



Developer Guide

AWS Step Functions



AWS Step Functions: Developer Guide

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is AWS Step Functions?	1
AWS SDK and Optimized integrations	2
Standard and Express workflows	2
Standard workflows specifications	2
Express workflows specifications	2
Use cases	3
Use case #1: Function orchestration	3
Use case #2: Branching	4
Use case #3: Error handling	4
Use case #4: Human in the loop	5
Use case #5: Parallel processing	6
Use case #6: Dynamic parallelism	6
Service integrations	6
Supported regions	10
Is this your first time using Step Functions?	10
Getting started	12
Key concepts	12
Tutorials in this series	14
Prerequisites	17
Sign up for an AWS account	17
Create a user with administrative access	18
Tutorial 1: Create the prototype for your state machine	19
Next steps	20
Tutorial 2: Define the first service integration using a Lambda function	20
Step 1: Create and test the Lambda function	21
Step 2: Update the workflow – configure the Get credit limit state	22
Next steps	22
Tutorial 3: Implement an if-else condition in your workflow	23
Step 1: Create an Amazon SNS topic that receives the callback token	23
Step 2: Create a Lambda function to handle the callback	24
Step 3: Update the workflow – add if-else condition logic in the Choice state	26
Next steps	29
Tutorial 4: Define multiple tasks to perform in parallel	29
Step 1: Create the Lambda functions to perform the required checks	29

Step 2: Update the workflow – Add parallel tasks to be performed	31
Tutorial 5: Concurrently iterate over a collection of items	32
Step 1: Create a DynamoDB table to store the name of all credit bureaus	33
Step 2: Update the state machine – Fetch results from the DynamoDB table	34
Step 3: Create a Lambda function that returns the credit scores for all credit bureaus	34
Step 4: Update the state machine – add a Map state to iteratively fetch credit scores	35
Tutorial 6: Save the workflow and execute the state machine	35
Step 1: Save the state machine	36
Step 2: Add the remaining IAM policies	37
Step 3: Run the state machine	37
Tutorial 7: Configure input and output	39
Select specific portions of the raw input using the InputPath filter	40
Manipulate the selected input using the Parameters filter	44
Configure output using the ResultSelector, ResultPath, and OutputPath filters	45
Tutorial 8: Debug errors in the console	47
Debugging the invalid path Choice state error	48
Debugging JSON path expression errors while applying input and output filters	50
Use cases	52
Data processing	52
Machine learning	54
Microservice orchestration	55
IT and security automation	56
How Step Functions works	58
Standard vs. Express Workflows	58
Synchronous and Asynchronous Express Workflows	62
Execution guarantees	63
Cost-optimization using Express Workflows	64
States	66
Amazon States Language	68
Pass	89
Task	91
Choice	112
Wait	118
Succeed	120
Fail	121
Parallel	123

Map	127
Map state processing modes	128
Inline mode and Distributed mode differences	129
Using Map state in Inline mode	131
Using Map state in Distributed mode	139
Tolerated failure threshold for Distributed Map state	150
Transitions	152
Transitions in Distributed Map state	153
State Machine Data	154
Data Format	154
State Machine Input/Output	155
State Input/Output	155
Input and Output Processing	156
Paths	159
InputPath, Parameters and ResultSelector	161
ResultPath	167
OutputPath	176
InputPath, ResultPath, and OutputPath Examples	177
Map state input and output fields	182
Context object	213
Data flow simulator	220
Using Data flow simulator	221
Data flow simulator considerations	222
Versions and aliases	223
Versions	224
Aliases	228
Authorization for versions and aliases	231
Associating executions with a version or alias	233
Deployment example	237
Gradual deployment of versions	240
Executions	249
Start Executions from a Task	250
Using EventBridge Scheduler	252
Standard and Express Workflow executions	258
Viewing and debugging executions	263
Redriving executions	285

Examining Map Run	295
Error handling	308
Error names	308
Retrying after an error	311
Fallback states	315
State machine examples using Retry and using Catch	318
Invoke Step Functions	322
Read Consistency	323
Tagging in Step Functions	323
Tagging for Cost Allocation	324
Tagging for Security	325
Viewing and Managing	326
Tagging API	326
Workflow Studio	327
Interface overview	328
Design mode	328
Code mode	334
Config mode	338
Keyboard shortcuts	342
Using Workflow Studio	342
Create a workflow	343
Design a workflow	345
Run your workflow	351
Edit your workflow	352
Export your workflow	354
Create your workflow prototype	355
Configure input and output	356
Configure input to a state	357
Configure output of a state	360
Execution roles in Workflow Studio	365
About auto-generated roles	366
Automatically generating roles	366
Resolving role generation problems	368
Role for testing HTTP Tasks in Workflow Studio	369
Role for testing an optimized service integration in Workflow Studio	369
Role for testing an AWS SDK service integration in Workflow Studio	369

Role for testing flow states in Workflow Studio	370
Error handling	371
Retry on errors	371
Catch errors	372
Timeouts	373
HeartbeatSeconds	373
Tutorial: Learn to use the AWS Step Functions Workflow Studio	373
Step 1: Navigate to Workflow Studio	374
Step 2: Create a state machine	374
Step 3: Review the auto-generated Amazon States Language definition	376
Step 4: Edit the workflow definition in Code mode	378
Step 5: Save the state machine	380
Step 6: Run the state machine	381
Step 7: Update your state machine	382
Step 8: Clean up	383
Tutorials	385
Create a Step Functions state machine that uses Lambda	385
Step 1: Create a Lambda function	386
Step 2: Test the Lambda function	387
Step 3: Create a state machine	387
Step 4: Run the state machine	390
Handling Error Conditions Using a State Machine	391
Step 1: Create a Lambda function that fails	392
Step 2: Test the Lambda function	393
Step 3: Create a state machine with a Catch field	393
Step 4: Run the state machine	396
Repeat an action using Inline Map state	397
Step 1: Create the workflow prototype	398
Step 2: Configure input and output	398
Step 3: Review the auto-generated Amazon States Language definition and save the workflow	399
Step 4: Run the state machine	401
Getting started with using Distributed Map state	402
Prerequisites	403
Step 1: Create the workflow prototype	404
Step 2: Configure the required fields for Map state	404

Step 3: Configure additional options	406
Step 4: Configure the Lambda function	406
Step 5: Update the workflow prototype	407
Step 6: Review the auto-generated Amazon States Language definition and save the workflow	408
Step 7: Run the state machine	410
Processing entire batch of data with a Lambda function	411
Step 1: Create the state machine	411
Step 2: Create the Lambda function	413
Step 3: Run the state machine	415
Processing individual data items with a Lambda function	416
Step 1: Create the state machine	417
Step 2: Create the Lambda function	419
Step 3: Run the state machine	415
Starting a State Machine Execution in Response to Amazon S3 Events	424
Prerequisite: Create a State Machine	424
Step 1: Create a Bucket in Amazon S3	425
Step 2: Enable Amazon S3 Event Notification with EventBridge	425
Step 3: Create an Amazon EventBridge Rule	426
Step 4: Test the Rule	427
Example of Execution Input	428
Creating a Step Functions API using API Gateway	429
Step 1: Create an IAM Role for API Gateway	429
Step 2: Create your API Gateway API	430
Step 3: Test and Deploy the API Gateway API	433
Create a Step Functions state machine using AWS SAM	435
Prerequisites	436
Step 1: Download a Sample AWS SAM Application	437
Step 2: Build Your Application	438
Step 3: Deploy Your Application to the AWS Cloud	439
Troubleshooting	440
Clean Up	441
Creating an Activity state machine	441
Step 1: Create an Activity	442
Step 2: Create a state machine	443
Step 3: Implement a Worker	444

Step 4: Run the state machine	447
Step 5: Run and Stop the Worker	448
Iterate a loop with Lambda	448
Step 1: Create a Lambda function to iterate a count	449
Step 2: Test the Lambda Function	450
Step 3: Create a State Machine	451
Step 4: Start a New Execution	454
Continuing Ongoing Work as a New Execution	455
Using a Step Functions API action (recommended)	455
Using a Lambda function	459
Deploying an Example Human Approval Project	471
Step 1: Create a Template	472
Step 2: Create a stack	472
Step 3: Approve the SNS subscription	473
Step 4: Run the state machine	474
Template Source Code	476
View X-Ray traces in Step Functions	486
Step 1: Create an IAM role for Lambda	486
Step 2: Create a Lambda function	487
Step 3: Create two more Lambda functions	489
Step 4: Create a state machine	489
Step 5: Run the state machine	491
Gather Amazon S3 bucket info using AWS SDK service integrations	495
Step 1: Create the state machine	495
Step 2: Add the necessary IAM role permissions	498
Step 3: Run a Standard state machine execution	498
Step 4: Run an Express state machine execution	499
Developer tools	501
Development Options	501
Step Functions console	502
AWS SDKs	502
Standard and Express workflows	503
HTTPS service API	503
Development environments	503
Endpoints	504
AWS CLI	504

Step Functions Local	504
AWS Toolkit for Visual Studio Code	505
AWS Serverless Application Model and Step Functions	505
Terraform and Step Functions	505
Definition format support	505
Step Functions and AWS SAM	512
Why use Step Functions with AWS SAM?	513
Step Functions integration with the AWS SAM specification	514
Step Functions integration with the SAM CLI	514
DefinitionSubstitutions in AWS SAM templates	515
Next steps	519
Using Workflow Studio in Application Composer	519
Using Workflow Studio in Application Composer	520
Dynamically reference resources using CloudFormation definition substitutions	520
Connect service integration tasks to enhanced component cards	521
Import existing projects and sync them locally	522
Unavailable Workflow Studio features in AWS Application Composer	522
Creating a Lambda State Machine Using AWS CloudFormation	523
Step 1: Set up your AWS CloudFormation template	523
Step 2: Use the AWS CloudFormation template to create a Lambda State Machine	529
Step 3: Start a State Machine execution	534
Creating a Lambda state machine using AWS CDK	535
Step 1: Set up your AWS CDK project	536
Step 2: Use AWS CDK to create a state machine	537
Step 3: Start a state machine execution	546
Step 4: Clean Up	547
Next steps	547
Creating an API Gateway REST API with Synchronous Express State Machine Using the AWS CDK	548
Step 1: Set Up Your AWS CDK Project	548
Step 2: Use the AWS CDK to create an API Gateway REST API with Synchronous Express State Machine backend integration	552
Step 3: Test the API Gateway	562
Step 4: Clean Up	565
Data Science SDK	565
Deploying state machines using Terraform	566

Prerequisites	566
Development lifecycle with Terraform	567
IAM roles and policies for your state machine	569
Testing and debugging	571
Using TestState API	571
Considerations about using the TestState API	572
Using inspection levels in TestState API	573
IAM permissions for using TestState API	580
Testing a state (Console)	581
Testing a state using AWS CLI	582
Testing and debugging input and output data flow	588
Testing state machines locally	592
Setting Up Step Functions Local (Downloadable Version) and Docker	593
Setting Up Step Functions Local (Downloadable Version) - Java Version	594
Setting Configuration Options for Step Functions Local	595
Running Step Functions Local on Your Computer	597
Testing Step Functions and AWS SAM CLI Local	599
Using Mocked Service Integrations	604
Best practices	622
Use timeouts to avoid stuck executions	622
Use Amazon S3 ARNs instead of passing large payloads	623
Avoid reaching the history quota	625
Handle Lambda service exceptions	626
Avoid latency when polling for activity tasks	627
Choosing Standard or Express Workflows	628
Amazon CloudWatch Logs resource policy size restrictions	628
Working with other services	630
Call other AWS services	630
Optimized integrations	631
AWS SDK integrations	631
Integration pattern support	631
Cross-account access	634
AWS SDK service integrations	634
Using AWS SDK service integrations	635
Supported services	636
Unsupported API actions for supported services	676

Deprecated AWS SDK service integrations	678
Optimized integrations	678
Amazon API Gateway	682
Amazon Athena	690
AWS Batch	693
Amazon Bedrock	694
AWS CodeBuild	699
Amazon DynamoDB	704
Amazon ECS/Fargate	707
Amazon EKS	711
Amazon EMR	725
Amazon EMR on EKS	737
Amazon EMR Serverless	741
Amazon EventBridge	750
AWS Glue	752
AWS Glue DataBrew	753
AWS Lambda	754
AWS Elemental MediaConvert	758
Amazon SageMaker	761
Amazon SNS	771
Amazon SQS	774
AWS Step Functions	777
Call third-party APIs	781
HTTP Task definition	781
HTTP Task fields	782
Authentication for an HTTP Task	788
Merging EventBridge connection and HTTP Task definition data	789
Applying URL-encoding on request body	792
IAM permissions to run an HTTP Task	794
HTTP Task example	795
Testing an HTTP Task	798
Unsupported HTTP Task responses	800
Service integration patterns	800
Request Response	800
Run a Job (.sync)	801
Wait for a Callback with the Task Token	803

Pass parameters to a service API	809
Pass static JSON as parameters	809
Pass state input as parameters using Paths	810
Pass Context Object Nodes as Parameters	810
Change log for integrations	811
Starter templates	835
Manage a batch job (AWS Batch, Amazon SNS)	836
Step 1: Create the state machine and provision resources	836
Step 2: Run the state machine	838
Example State Machine Code	839
IAM Example	841
Manage a container task (Amazon ECS, Amazon SNS)	842
Step 1: Create the state machine and provision resources	842
Step 2: Run the state machine	844
Example State Machine Code	845
IAM Example	847
Transfer data records (Lambda, DynamoDB, Amazon SQS)	848
Step 1: Create the state machine and provision resources	848
Step 2: Run the state machine	850
Example State Machine Code	851
IAM Example	853
Poll for Job Status (Lambda, AWS Batch)	854
Step 1: Create the state machine and provision resources	855
Step 2: Run the state machine	857
Example State Machine Code	859
Task Timer (Lambda, Amazon SNS)	861
Step 1: Create the state machine and provision resources	862
Step 2: Run the state machine	864
Callback Pattern Example (Amazon SQS, Amazon SNS, Lambda)	866
Step 1: Create the state machine and provision resources	866
Step 2: Run the state machine	868
Lambda Callback Example	870
Manage an Amazon EMR Job	871
Step 1: Create the State Machine and Provision Resources	871
Step 2: Run the state machine	844
Example State Machine Code	845

IAM Example	847
Run an EMR Serverless job	879
AWS CloudFormation template and additional resources	880
Step 1: Create the state machine and provision resources	880
Step 2: Run the state machine	882
Start a Workflow within a Workflow (Step Functions, Lambda)	883
Step 1: Create the state machine and provision resources	883
Step 2: Run the state machine	885
Example State Machine Code	886
Dynamically process data with a Map state	889
Step 1: Create the state machine and provision resources	889
Step 2: Subscribe to the Amazon SNS topic	892
Step 3: Add messages to the Amazon SQS queue	892
Step 4: Run the state machine	893
Example state machine code	894
IAM example	896
Process a CSV file with Distributed Map	897
AWS CloudFormation template and additional resources	898
Step 1: Create the state machine and provision resources	898
Step 2: Run the state machine	901
Process data in an Amazon S3 bucket with Distributed Map	902
AWS CloudFormation template and additional resources	903
Step 1: Create the state machine and provision resources	904
Step 2: Run the state machine	907
Train a Machine Learning Model	908
Step 1: Create the state machine and provision resources	908
Step 2: Run the state machine	910
Example State Machine Code	911
IAM Example	914
Tune a Machine Learning Model	915
Step 1: Create the state machine and provision resources	916
Step 2: Run the state machine	918
Example State Machine Code	919
IAM Examples	924
Process High-Volume Messages from Amazon SQS (Express Workflows)	926
Step 1: Create the state machine and provision resources	927

Step 2: Trigger the state machine execution	929
Example Lambda Function Code	930
Example State Machine Code	931
IAM Example	932
Selective Checkpointing Example (Express Workflows)	933
Step 1: Create the State Machine and Provision Resources	934
Step 2: Run the state machine	936
Example State Machine Code for the Parent (Standard Workflows)	937
Example IAM Role for the Parent State Machine	940
Example State Machine Code for the Nested State Machine (Express Workflows)	937
Example IAM Role for Child State Machine	944
Build an AWS CodeBuild Project (CodeBuild, Amazon SNS)	945
Step 1: Create the state machine and provision resources	945
Step 2: Run the state machine	947
Example State Machine Code	948
Preprocess data and train a machine learning model	950
Step 1: Create the state machine and provision resources	950
Step 2: Run the state machine	953
Example State Machine Code	954
IAM Example	957
Lambda orchestration example	958
Step 1: Create the state machine and provision resources	959
Step 2: Run the state machine	961
About the state machine and its execution	962
IAM Examples	966
Start an Athena query	968
Step 1: Create the state machine and provision resources	969
Step 2: Run the state machine	971
Example State Machine Code	972
IAM Example	974
Execute multiple queries (Amazon Athena, Amazon SNS)	976
Step 1: Create the state machine and provision resources	976
Step 2: Run the state machine	979
Example State Machine Code	980
IAM Examples	982
Query large datasets (Amazon Athena, Amazon S3, AWS Glue, Amazon SNS)	986

Step 1: Create the state machine and provision resources	986
Step 2: Run the state machine	989
Example State Machine Code	990
IAM Examples	991
Keep data up to date (Amazon Athena, Amazon S3, AWS Glue)	995
Step 1: Create the state machine and provision resources	995
Step 2: Run the state machine	997
Example State Machine Code	998
IAM Example	999
Manage an Amazon EKS cluster	1001
Step 1: Create the state machine and provision resources	1002
Step 2: Run the state machine	1005
Example State Machine Code	1006
IAM Example	1010
Make a call to API Gateway	1011
Step 1: Create the state machine and provision resources	1012
Step 2: Run the state machine	1014
Example State Machine Code	1015
IAM Example	1017
Call a microservice with API Gateway	1017
Step 1: Create the state machine and provision resources	1018
Step 2: Run the state machine	1020
Example State Machine Code	1021
IAM Example	1023
Send a custom event to EventBridge	1024
Step 1: Create the state machine and provision resources	1025
Step 2: Run the state machine	1027
Example State Machine Code	1028
IAM Example	1029
Invoke Synchronous Express Workflows	1030
Step 1: Create the state machine and provision resources	1030
Step 2: Run the state machine	1032
Example State Machine Code	1033
IAM Examples	1035
Run ETL/ELT workflows using Amazon Redshift	1036
Step 1: Create the state machine and provision resources	1037

Step 2: Run the state machine	1040
Example State Machine Code	1041
IAM Example	1061
Use Step Functions and AWS Batch with error handling	1062
Step 1: Create the state machine and provision resources	1062
Step 2: Run the state machine	1064
Example State Machine Code	1065
IAM Example	1067
Fan out an AWS Batch job	1068
Step 1: Create the state machine and provision resources	1068
Step 2: Run the state machine	1070
Example State Machine Code	1071
IAM Example	1073
AWS Batch with Lambda	1074
Step 1: Create the State Machine and Provision Resources	1074
Step 2: Run the state machine	1076
Example State Machine Code	1077
IAM Example	1078
Perform AI prompt-chaining with Amazon Bedrock	1080
AWS CloudFormation template and additional resources	1080
Prerequisites	1080
Step 1: Create the state machine and provision resources	1081
Step 2: Run the state machine	1083
Quotas	1085
General quotas	1086
Quotas related to accounts	1087
Quotas related to HTTP Task	1088
Quotas related to state throttling	1088
Quotas related to API action throttling	1089
Quota related to TestState API	1090
Other quotas	1090
Quotas related to state machine executions	1093
Quotas related to task executions	1094
Quotas related to versions and aliases	1095
Restrictions related to tagging	1096
Logging and monitoring	1097

Amazon CloudWatch Metrics	1097
Metrics that report a time interval	1098
Metrics that report a count	1099
Execution metrics	1099
Resource count metrics for versions and aliases	1102
Activity Metrics	1103
Lambda Function Metrics	1104
Service Integration Metrics	1105
Service Metrics	1106
API Metrics	1107
Best-effort CloudWatch metrics delivery	1107
Viewing Metrics for Step Functions	1108
Setting Alarms for Step Functions	1110
Amazon EventBridge Events	1112
EventBridge payloads	1113
Step Functions event examples	1114
Routing a Step Functions event to EventBridge	1118
Recording with CloudTrail	1120
Data events in CloudTrail	1121
Management events in CloudTrail	1122
Event examples	1124
Logging using CloudWatch Logs	1126
Configure logging	1126
CloudWatch Logs payloads	1127
IAM Policies for logging to CloudWatch Logs	1127
Log levels	1129
X-Ray	1133
Setup and configuration	1134
Concepts	1138
Service integrations	1139
Viewing the X-Ray console	1140
Viewing X-Ray tracing information for Step Functions	1140
Traces	1140
Service map	1141
Segments and subsegments	1142
Analytics	1144

Configuration	1145
What if there is no data in the trace map or service map?	1146
Using AWS User Notifications with Step Functions	1147
Security	1148
Data Protection	1148
Encryption	1149
Identity and Access Management	1149
Audience	1149
Authenticating with identities	1150
Managing access using policies	1153
Access Control	1156
Policy actions	1156
Policy resources	1157
Policy condition keys	1158
ACLs	1158
ABAC	1159
Temporary credentials	1159
Principal permissions	1160
Service roles	1160
Service-linked roles	1161
How AWS Step Functions works with IAM	1161
Identity-based policy examples	1162
Identity-based policies	1165
Resource-based policies	1165
AWS managed policies	1166
Creating a state machine IAM role	1168
Creating Granular IAM Permissions for Non-Admin Users	1171
Accessing cross-account AWS resources	1174
VPC Endpoints	1185
IAM Policies for integrated services	1188
IAM policies for using Distributed Map state	1279
Tag-based Policies	1284
Troubleshooting	1286
Logging and Monitoring	1287
Compliance Validation	1288
Resilience	1288

Infrastructure Security	1289
Configuration and Vulnerability Analysis	1289
Troubleshooting	1290
General troubleshooting	1290
I'm unable to create a state machine.	1290
I'm unable to use a JsonPath to reference the previous task's output.	1290
There was a delay in state transitions.	1291
When I start new Standard Workflow executions, they fail with the ExecutionLimitExceeded error.	1291
A failure on one branch in a parallel state causes the whole execution to fail.	1291
Troubleshooting service integrations	1291
My job is complete in the downstream service, but in Step Functions the task state remains "In progress" or its completion is delayed.	1291
I want to return a JSON output from a nested state machine execution.	1292
I can't invoke a Lambda function from another account.	1292
I'm unable to see task tokens passed from <code>.waitForTaskToken</code> states.	1293
Troubleshooting activities	1294
My state machine execution is stuck at an activity state.	1294
My activity worker times out while waiting for a task token.	1294
Troubleshooting Express Workflows	1295
My application times out before receiving a response from a <code>StartSyncExecution</code> API call.	1295
I'm unable to see the execution history in order to troubleshoot Express Workflow failures.	1295
Related information	1297
Recent feature launches	1298
Document history	1301

What is AWS Step Functions?

AWS Step Functions is a visual workflow service that helps you build distributed applications, automate processes, orchestrate microservices, and create data and machine learning (ML) pipelines.

In the Step Functions' graphical console, you can see your application's workflow as a series of event-driven steps.

Step Functions is based on *state machines* and *tasks*. In Step Functions, state machines are called *workflows*, which are a series of event-driven steps. Each step in a workflow is called a *state*. For example, a [Task state](#) represents a unit of work that another AWS service performs, such as calling another AWS service or API.

With Step Functions' built-in controls, you can examine the state of each step in your workflow to make sure that your application runs in order and as expected. Depending on your use case, you can have Step Functions call AWS services, such as Lambda, to perform tasks. You can create workflows that process and publish machine learning models. You can have Step Functions control AWS services, such as AWS Glue, to create extract, transform, and load (ETL) workflows. You also can create long-running, automated workflows for applications that require human interaction.

Tip

To learn how to use Step Functions, follow the interactive modules in the [AWS Step Functions Workshop](#), or read the [Getting Started](#) section in this guide to create a credit card application workflow.

Topics

- [AWS SDK and Optimized integrations](#)
- [Standard and Express workflows](#)
- [Use cases](#)
- [Service integrations](#)
- [Supported regions](#)
- [Is this your first time using Step Functions?](#)

AWS SDK and Optimized integrations

To call other AWS services, you can use Step Functions' AWS SDK integrations, or you can use one of Step Functions' Optimized integrations.

- The [AWS SDK integrations](#) let you call any of the over two hundred AWS services directly from your state machine, giving you access to over nine thousand API actions.
- [Step Functions' Optimized integrations](#) have been customized to simplify usage in your state machines.

Standard and Express workflows

Step Functions has two workflow types. Standard workflows have exactly-once workflow execution and can run for up to one year. This means that each step in a Standard workflow will execute exactly once. Express workflows, however, have at-least-once workflow execution and can run for up to five minutes. This means that one or more steps in an Express Workflow can potentially run more than once, while each step in the workflow executes at least once.

Executions are instances where you run your workflow to perform tasks. Standard workflows are ideal for long-running, auditable workflows, as they show execution history and visual debugging. Express workflows are ideal for high-event-rate workloads, such as streaming data processing and IoT data ingestion.

Standard workflows specifications

- 2,000 per second execution rate
- 4,000 per second state transition rate
- Priced by state transition
- Show execution history and visual debugging
- Support all service integrations and patterns

Express workflows specifications

- 100,000 per second execution rate
- Nearly unlimited state transition rate

- Priced by number and duration of executions
- Send execution history to [Amazon CloudWatch](#)
- Show execution history and visual debugging based on the **Log level** enabled
- Support all service integrations and most patterns

For more information about Standard and Express workflows, including Step Functions pricing, see the following:

- [Standard vs. Express Workflows](#)
- [AWS Step Functions pricing](#)

Use cases

Step Functions manages your application's components and logic, so you can write less code and focus on building and updating your application quickly. This section describes typical use cases for working with Step Functions.

Use case #1: Function orchestration

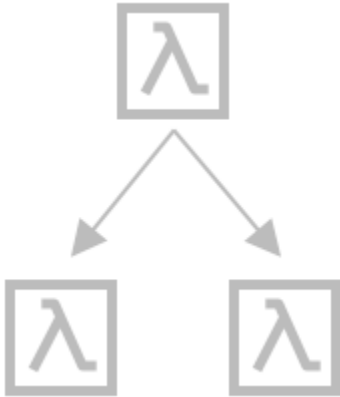


You create a workflow that runs a group of Lambda functions (steps) in a specific order. One Lambda function's output passes to the next Lambda function's input. The last step in your workflow gives a result. With Step Functions, you can see how each step in your workflow interacts with one other, so you can make sure that each step performs its intended function.

For a tutorial that shows you how to create a state machine with a group of functions, see the following:

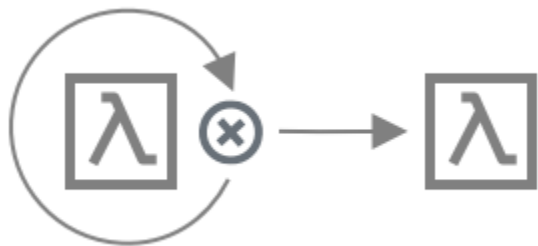
- [Getting started with AWS Step Functions](#)

Use case #2: Branching



A customer requests a credit limit increase. Using a [Choice](#) state, you can have Step Functions make decisions based on the Choice state's input. If the request is more than your customer's pre-approved credit limit, you can have Step Functions send your customer's request to a manager for sign-off. If the request is less than your customer's pre-approved credit limit, you can have Step Functions approve the request automatically.

Use case #3: Error handling



Retry

In this use case, a customer requests a username. The first time, your customer's request is unsuccessful. Using a `Retry` statement, you can have Step Functions try your customer's request again. The second time, your customer's request is successful.

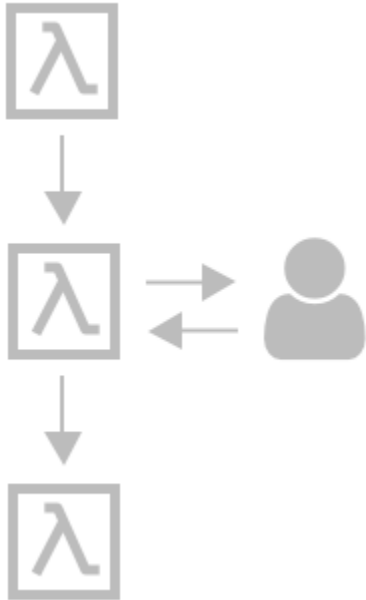
Catch

In a similar use case, a customer requests an unavailable username. Using a `Catch` statement, you have Step Functions suggest an available username. If your customer takes the available username, you can have Step Functions go to the next step in your workflow, which is to send a confirmation email. If your customer doesn't take the available username, you have Step Functions go to a different step in your workflow, which is to start the sign-up process over.

For more detailed examples of `Retry` and `Catch` statements, see the following:

- [Error handling in Step Functions](#)

Use case #4: Human in the loop

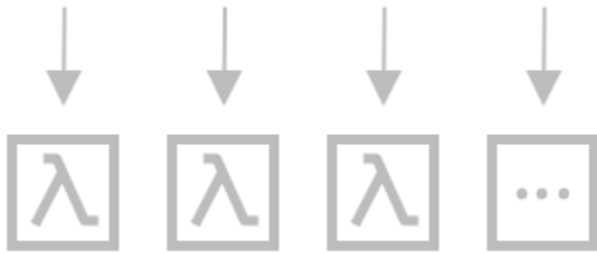


Using a banking app, one of your customers sends money to a friend. Your customer waits for a confirmation email. With [a callback and a task token](#), you have Step Functions tell Lambda to send your customer's money and report back when your customer's friend receives it. After Lambda reports back that your customer's friend received the money, you can have Step Functions go to the next step in your workflow, which is to send your customer a confirmation email.

To see a sample project that shows a callback with a task token, see the following:

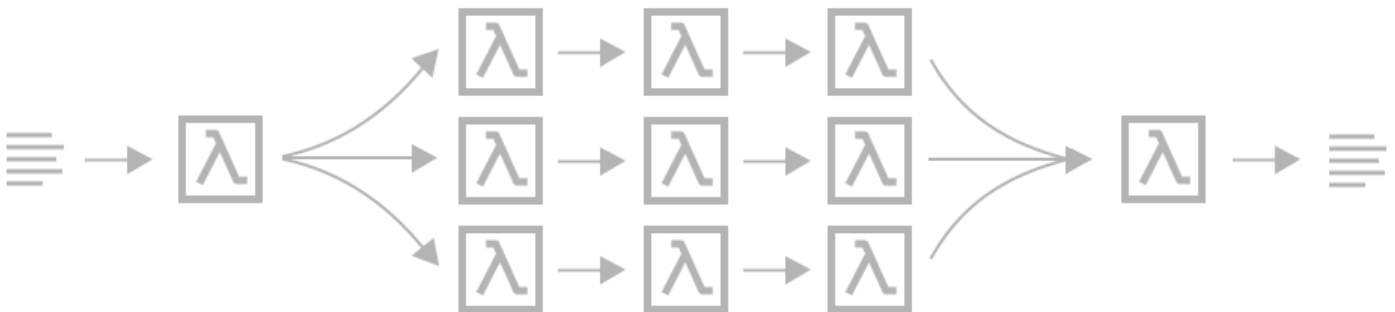
- [Callback Pattern Example \(Amazon SQS, Amazon SNS, Lambda\)](#)

Use case #5: Parallel processing



A customer converts a video file into five different display resolutions, so viewers can watch the video on multiple devices. Using a [Parallel](#) state, Step Functions inputs the video file, so Lambda can process it into the five display resolutions at the same time.

Use case #6: Dynamic parallelism



A customer orders three items, and you need to prepare each item for delivery. You check each item's availability, gather each item, and then package each item for delivery. Using a [Map](#) state, Step Functions has Lambda process each of your customer's items in parallel. Once all of your customer's items are packaged for delivery, Step Functions goes to the next step in your workflow, which is to send your customer a confirmation email with tracking information.

To see a sample project that shows dynamic parallelism using a Map state, see the following:

- [Dynamically process data with a Map state](#)

Service integrations

Step Functions integrates with multiple AWS services. To combine Step Functions with these services, use the following service integration patterns:

Request a response (default)

- Call a service, and let Step Functions progress to the next state after it gets an HTTP response.

Run a job (.sync)

- Call a service, and have Step Functions wait for a job to complete.

Wait for a callback with a task token (.waitForTaskToken)

- Call a service with a task token, and have Step Functions wait until the task token returns with a callback.

The table below shows the available service integrations and service integration patterns for Step Functions.

Standard Workflows and Express Workflows support the same integrations but not the same **integration patterns**.

- Optimized integrations pattern support is different for each integration.
- Express Workflows do **not** support Run a Job (.sync) or Wait for Callback (.waitForTaskToken).
- For more information, see [Standard vs. Express Workflows](#).

Standard Workflows

Supported service integrations

	Service	<u>Request Response</u>	<u>Run a Job (.sync)</u>	<u>Wait for Callback (.waitForTaskToken)</u>
Optimized integrations	Amazon API Gateway	✓		✓
	Amazon Athena	✓	✓	
	AWS Batch	✓	✓	

	Service	<u>Request Response</u>	<u>Run a Job (.sync)</u>	<u>Wait for Callback (.waitForTaskToken)</u>
	Amazon Bedrock	✓	✓	✓
	AWS CodeBuild	✓	✓	
	Amazon DynamoDB	✓		
	Amazon ECS/Fargate	✓	✓	✓
	Amazon EKS	✓	✓	✓
	Amazon EMR	✓	✓	
	Amazon EMR on EKS	✓	✓	
	Amazon EMR Serverless	✓	✓	
	Amazon EventBridge	✓		✓
	AWS Glue	✓	✓	
	AWS Glue DataBrew	✓	✓	
	AWS Lambda	✓		✓
	AWS Elemental MediaConvert	✓	✓	
	Amazon SageMaker	✓	✓	
	Amazon SNS	✓		✓
	Amazon SQS	✓		✓
	AWS Step Functions	✓	✓	✓

	Service	Request Response	Run a Job (.sync)	Wait for Callback (.waitForTaskToken)
AWS SDK integrations	Over two hundred	✓		✓

Express Workflows

Supported service integrations

	Service	Request Response	Run a Job (.sync)	Wait for Callback (.waitForTaskToken)
Optimized integrations	Amazon API Gateway	✓		
	Amazon Athena	✓		
	AWS Batch	✓		
	Amazon Bedrock	✓		
	AWS CodeBuild	✓		
	Amazon DynamoDB	✓		
	Amazon ECS/Fargate	✓		
	Amazon EKS	✓		
	Amazon EMR	✓		
	Amazon EMR on EKS	✓		
Amazon EMR Serverless	✓			

	Service	Request Response	Run a Job (.sync)	Wait for Callback (.waitForTaskToken)
	Amazon EventBridge	✓		
	AWS Glue	✓		
	AWS Glue DataBrew	✓		
	AWS Lambda	✓		
	AWS Elemental MediaConvert	✓		
	Amazon SageMaker	✓		
	Amazon SNS	✓		
	Amazon SQS	✓		
	AWS Step Functions	✓		
AWS SDK integrations	Over two hundred	✓		

Supported regions

Most AWS regions support Step Functions. For a complete list of AWS regions where Step Functions is available, see the [AWS Region Table](#).

Is this your first time using Step Functions?

If this is your first time using Step Functions, the following topics help you understand different parts of working with Step Functions, including how Step Functions combines with other AWS services:

- [Tutorials for Step Functions](#)
- [Starter templates for Step Functions](#)
- [AWS Step Functions Data Science SDK for Python](#)

Getting started with AWS Step Functions

Step Functions is a serverless orchestration service that lets you define an application workflow as a series of event-driven steps. Each step in the workflow is called a *state*. You most commonly use states, such as [Task state](#), [Choice](#), [Parallel](#), and [Map](#), to define your workflows. Within Task states, you can use the AWS SDK integrations that Step Functions supports and orchestrate multiple AWS services in your workflows.

Topics

- [Key concepts](#)
- [Tutorials in this series](#)
- [Prerequisites for getting started with AWS Step Functions](#)
- [Tutorial 1: Create the prototype for your state machine](#)
- [Tutorial 2: Define the first service integration using a Lambda function](#)
- [Tutorial 3: Implement an if-else condition in your workflow](#)
- [Tutorial 4: Define multiple tasks to perform in parallel](#)
- [Tutorial 5: Concurrently iterate over a collection of items](#)
- [Tutorial 6: Save the workflow and execute the state machine](#)
- [Tutorial 7: Configure input and output](#)
- [Tutorial 8: Debug errors in the console](#)

Key concepts

Before you start the tutorials, review the following key Step Functions terms for context.

Term	Description
Workflow	A sequence of steps that often reflect a business process.
States	Individual steps in your state machine that can make decisions based on their input, perform actions from those inputs, and pass output to other states. For more information, see States .

Term	Description
Workflow Studio	<p>A visual workflow designer that helps you to prototype and build workflows faster.</p> <p>For more information, see AWS Step Functions Workflow Studio.</p>
State machine	<p>A workflow defined using JSON text representing the individual states or steps in the workflow along with fields, such as <code>StartAt</code>, <code>TimeoutSeconds</code> , and <code>Version</code>.</p> <p>For more information, see State machine structure.</p>
Amazon States Language	<p>A JSON-based, structured language used to define your state machines. With ASL, you define a collection of states that can do work (Task state), determine which states to transition to next (Choice state), and stop an execution with an error (Fail state).</p> <p>For more information, see Amazon States Language.</p>
Input and output configuration	<p>States in a workflow receive JSON data as input and usually pass JSON data as output to the next state. Step Functions provides filters to control the data flow between states.</p> <p>For more information, see Input and Output Processing in Step Functions.</p>
Service integration	<p>You can call AWS service API actions from your workflow.</p> <p>For more information, see Using AWS Step Functions with other services.</p>
Service integration type	<ul style="list-style-type: none">• AWS SDK integrations – Standard way to call any of over two hundred AWS services and over nine thousand API actions directly from your state machine.• Optimized integrations – Custom integrations that streamline calling and exchanging data with certain services. For example, Lambda Invoke will automatically convert the <code>Payload</code> field of the response from an escaped JSON string into a JSON object.

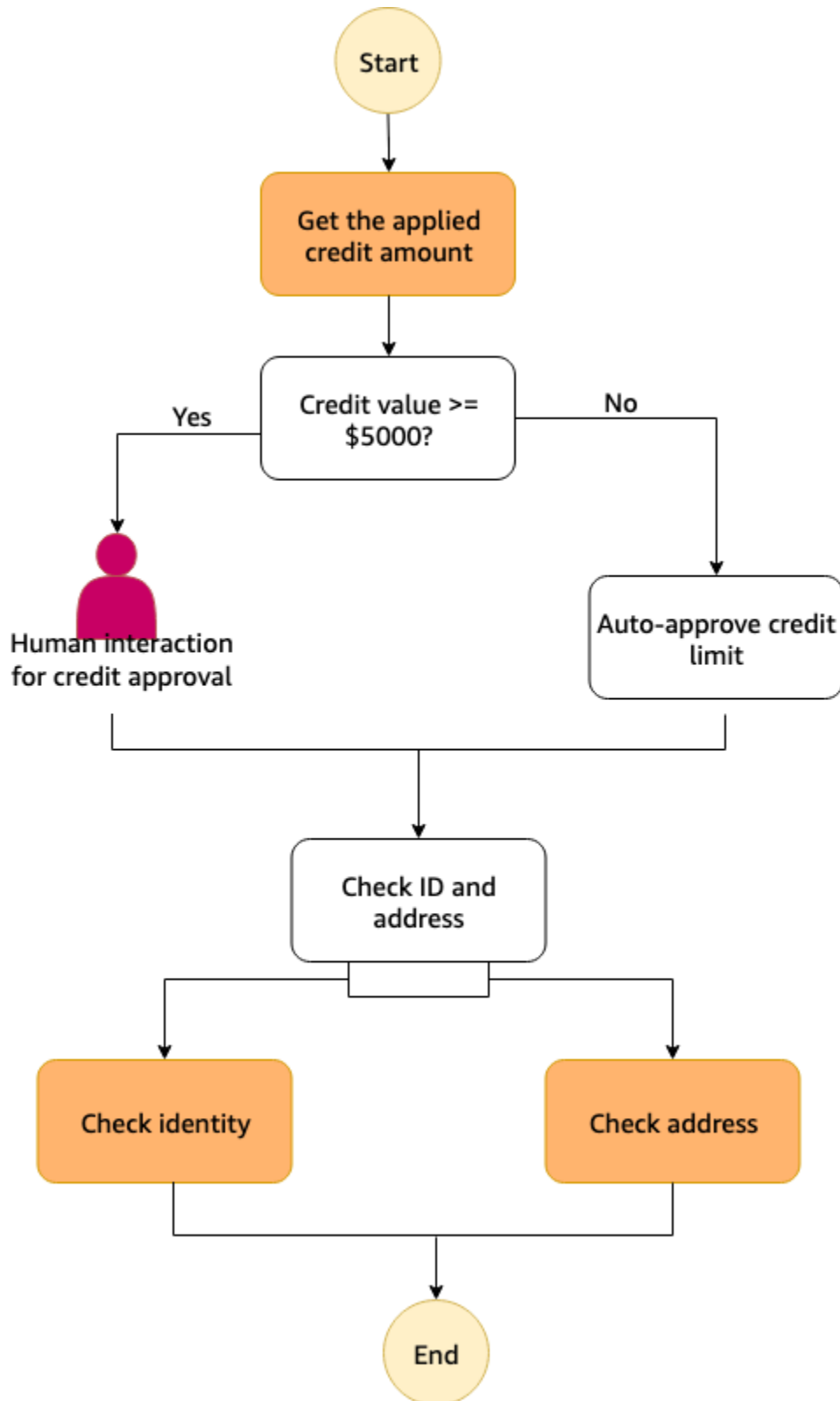
Term	Description
Service integration pattern	<p>When calling an AWS service, you use one of the following service integration patterns:</p> <ul style="list-style-type: none">• Request a response (default) – Call a service and move to the next state immediately after receiving an HTTP response.• Run a job (.sync) – Call a service and have Step Functions wait for a job to complete.• Wait for a callback with a task token (.waitForTaskToken) – Call a service with a task token and have Step Functions wait until the task token returns with a callback.
Execution	<p>State machine executions are instances where you run your workflow to perform tasks.</p> <p>For more information, see Executions in Step Functions.</p>

Tutorials in this series

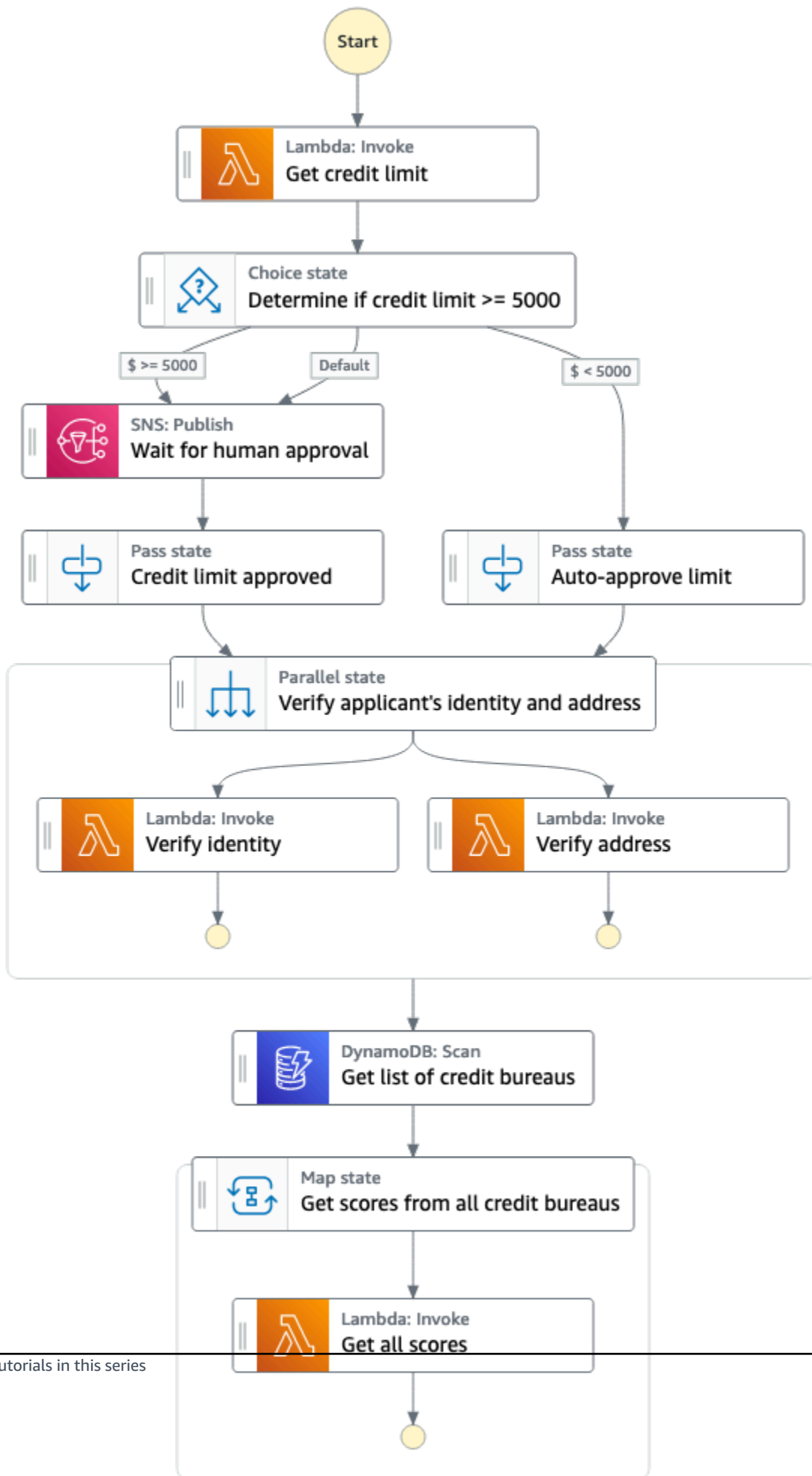
After completing these tutorials, you'll have a workflow that simulates processing a credit card application. You'll learn how to use common states and integrate your workflow with other AWS services

Step Functions can be used to create **many types** of workflows, such as data processing, IT automation, machine learning, and media encoding.

The following flowchart depicts steps for a business to process a credit card application. If the amount of requested credit is under \$5000, the credit limit will be automatically approved. If the request exceeds the limit, the workflow will add a human in the loop to verify the requestors identity and review credit scores.

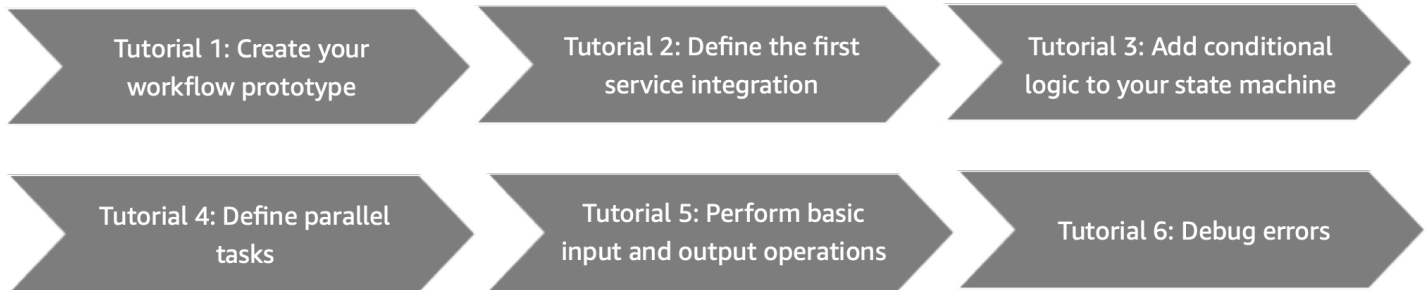


The following diagram shows how the credit application business process steps are represented by states in a Step Functions workflow.



In the following series of tutorials, you will build the credit card processing workflow.

We recommend completing these tutorials **in order** to learn the key features of Step Functions.



Before you get started, make sure to complete the [prerequisites](#).

Prerequisites for getting started with AWS Step Functions

Before you use AWS Step Functions for the first time, complete the following tasks.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform [tasks that require root user access](#).

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to <https://aws.amazon.com/> and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the [AWS Management Console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

For help signing in by using root user, see [Signing in as the root user](#) in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

For instructions, see [Enable a virtual MFA device for your AWS account root user \(console\)](#) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

For instructions, see [Enabling AWS IAM Identity Center](#) in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

For a tutorial about using the IAM Identity Center directory as your identity source, see [Configure user access with the default IAM Identity Center directory](#) in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see [Signing in to the AWS access portal](#) in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

For instructions, see [Create a permission set](#) in the *AWS IAM Identity Center User Guide*.

2. Assign users to a group, and then assign single sign-on access to the group.

For instructions, see [Add groups](#) in the *AWS IAM Identity Center User Guide*.

Tutorial 1: Create the prototype for your state machine

In this tutorial, you create the prototype for your credit card processing workflow using [Step Functions' Workflow Studio](#). You'll choose the required API actions and states from the **Actions** and **Flow** tabs respectively, and use the drag and drop feature of Workflow Studio to create the workflow prototype. In the subsequent tutorials, you'll learn how to configure the AWS services and the Step Functions' states you'll be using in this workflow.

To create the state machine prototype

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).
4. In Workflow Studio, from the **Actions** tab, drag an **AWS Lambda Invoke** API action and drop it to the empty state labelled **Drag first state here**. Configure it as follows:
 - In the **Configuration** tab, for **State name**, enter **Get credit limit**.
5. From the **Flow** tab, drag and drop a **Choice** state below the **Get credit limit** state. Rename the **Choice** state to **Credit applied >= 5000?**.
6. Drag and drop the following states as branches of the **Credit applied >= 5000?** state.
 - a. **Amazon SNS Publish** – From the **Actions** tab, drag and drop the **Amazon SNS Publish** API action. Rename this state to **Wait for human approval**.
 - b. **Pass** state — From the **Flow** tab, drag and drop the **Pass** state. Rename this branch to **Auto-approve limit**.
7. Drag and drop a **Pass** state below the **Wait for human approval** state. Rename this **Pass** state to **Credit limit approved**.

8. Drag and drop a **Parallel** state after the **Choice** state as follows:
 - a. Drop the **Parallel** state after the **Credit limit approved** state.
 - b. Rename the **Parallel** state to **Verify applicant's identity and address**.
 - c. Under both the branches of the **Parallel** state, drag and drop two **AWS Lambda Invoke** API actions.
 - d. Rename these states as **Verify identity** and **Verify address** respectively.
 - e. Choose the **Auto-approve limit** state and for **Next state**, select **Verify applicant's identity and address**.
9. Drag a **DynamoDB Scan** state and drop it below the **Verify applicant's identity and address** state. Rename the **DynamoDB Scan** state to **Get list of credit bureaus**.
10. Drag and drop a **Map** state after the **Get list of credit bureaus** state. Configure the **Map** state as follows:
 - a. Rename it to **Get scores from all credit bureaus**.
 - b. For **Processing mode**, keep the default selection of **Inline**.
 - c. Drag and drop an **AWS Lambda Invoke** API action to the empty state labelled **Drop state here**.
 - d. Rename the **AWS Lambda Invoke** state to **Get all scores**.
11. Keep this window open and proceed to the next tutorial for further actions.

Next steps

In the next tutorial, you learn how to integrate the Lambda function used by the **Get credit limit** state.

Tutorial 2: Define the first service integration using a Lambda function

In this tutorial, you learn how to define the first service integration for your workflow. You use the [Task](#) state named **Get credit limit** to invoke a Lambda function. Within Task states, you can use the AWS SDK integrations that Step Functions supports.

To define the first service integration for your workflow, first create a Lambda function. Then, update your workflow to specify the service integration with the Lambda function. The Lambda

function used in this tutorial returns a randomly generated integer representing the credit limit that an applicant has applied for.

Topics

- [Step 1: Create and test the Lambda function](#)
- [Step 2: Update the workflow – configure the Get credit limit state](#)
- [Next steps](#)

Step 1: Create and test the Lambda function

You can write code for the function in the AWS Management Console or your favorite editor. In the following steps, you create a Node.js Lambda function titled `RandomNumberforCredit`.

Important

Make sure that the workflow prototype you created in [Tutorial 1](#) is under the same AWS Region as the Lambda function you'll create in this tutorial.

1. In a new tab or window, open the [Lambda console](#) and create a **Node.js 16.x** Lambda function titled **`RandomNumberforCredit`**. For information about creating a Lambda function using the console, see [Create a Lambda function in the console](#) in the *AWS Lambda Developer Guide*.
2. On the **`RandomNumberforCredit`** page, choose **`index.mjs`** and replace the existing code in the **Code source** area with the following code.

```
export const handler = async function(event, context) {  
  
    const credLimit = Math.floor(Math.random() * 10000);  
    return (credLimit);  
  
};
```

3. From the **Function overview** section, copy the Amazon Resource Name of the Lambda function and save it in a text file. You'll need the function ARN while specifying the service integration for the **Get credit limit** state. The following is an example ARN:

```
arn:aws:lambda:us-east-2:123456789012:function:HelloWorld
```

4. Choose **Deploy** and then choose **Test** to deploy the changes and see the output of the Lambda function.

Step 2: Update the workflow – configure the Get credit limit state

In the Step Functions console, you'll update your workflow to specify service integration with the `RandomNumberforCredit` [Lambda function that you created in Step 1](#).

1. Open the [Step Functions console](#) window containing the workflow prototype you created in [Tutorial 1](#).
2. Choose the **Get credit limit** state, and in the **Configuration** tab, do the following:
 - a. For **Integration type**, keep the default selection of **Optimized**.

Using Step Functions, you can integrate with other AWS services and orchestrate them in your workflows. For more information about service integrations and their types, see [Using AWS Step Functions with other services](#).
 - b. For **Function name**, choose the **RandomNumberforCredit** Lambda function from the dropdown list.
 - c. Keep the default selections for rest of the items.
3. Keep this window open and proceed to the next tutorial for further actions.

Note

In this tutorial, you learned how to integrate with a Lambda function within a Task state in your workflows. You can also use other supported AWS SDK integrations in the Task state by specifying the service name and API call, as shown in the following syntax:

```
arn:aws:states:::aws-sdk:serviceName:apiAction
```

For more information, see [Using AWS Step Functions with other services](#).

Next steps

In the next tutorial, you'll implement *conditional logic* in your workflow. Conditional logic in Step Functions state machines behaves similar to an *if-else* statement in most common programming

languages. You'll use conditional logic in your workflow to determine the execution path based on conditional information.

Tutorial 3: Implement an if-else condition in your workflow

You can implement *if-else* conditions in your workflows by using the [Choice](#) state. It determines the workflow execution path based on whether a specified condition evaluates to true or false.

In this tutorial, you'll add conditional logic to determine if the applied credit amount returned by the `RandomNumberForCredit` Lambda function used in [Tutorial 2](#) exceeds a specific threshold limit. If the amount exceeds the threshold limit, the application requires a human interaction for approval. Otherwise, the application is auto-approved and moves to the next step.

You'll mimic the human interaction step by pausing the workflow execution until a task token is returned. To do this, you'll pass a task token to the AWS SDK integration you'll be using in this tutorial, which is Amazon Simple Notification Service. The workflow execution will be paused until it receives the task token back with a [SendTaskSuccess](#) API call. For more information about using task tokens, see [Wait for a Callback with the Task Token](#).

Because you've already defined the steps for human approval and auto-approval in your [workflow prototype](#), in this tutorial, you first create an Amazon SNS topic that receives the callback token. Then, you create a Lambda function to implement the callback functionality. Finally, you update your workflow prototype by adding the details of these AWS service integrations.

Topics

- [Step 1: Create an Amazon SNS topic that receives the callback token](#)
- [Step 2: Create a Lambda function to handle the callback](#)
- [Step 3: Update the workflow – add if-else condition logic in the Choice state](#)
- [Next steps](#)

Step 1: Create an Amazon SNS topic that receives the callback token

To implement the human interaction step, you'll publish to an Amazon Simple Notification Service topic and pass the callback task token to this topic. The callback task will pause the workflow execution until the task token is returned with a payload.

1. Open the [Amazon SNS console](#) and create a **Standard** topic type. For information about creating a topic, see [Create an Amazon SNS topic](#) in the *Amazon Simple Notification Service Developer Guide*.
2. Specify the topic name as **TaskTokenTopic**.
3. Make sure to copy the topic ARN and save it in a text file. You'll need the topic ARN while specifying the service integration for the **Wait for human approval** state. The following is an example topic ARN:

```
arn:aws:sns:us-east-2:123456789012:TaskTokenTopic
```

4. Create an email-based subscription for the topic and then confirm your subscription. For information about subscribing to a topic, see [Create a subscription to the topic](#) in the *Amazon Simple Notification Service Developer Guide*.

Step 2: Create a Lambda function to handle the callback

To handle callback functionality, you'll define a Lambda function and add the Amazon SNS topic you created in [Step 1](#) as a trigger for this function. When you publish to the Amazon SNS topic with a task token, the Lambda function is invoked with the payload of the published message.

- [Step 2.1: Create the Lambda function to handle callback](#)
- [Step 2.2: Add the Amazon SNS topic as a trigger for the Lambda function](#)
- [Step 2.3: Provide necessary permissions to the Lambda function IAM role](#)

Step 2.1: Create the Lambda function to handle callback

In this function, you'll process the credit limit approval request and return the request's result as successful with the [SendTaskSuccess](#) API call. This Lambda function will also return the task token it received from the Amazon SNS topic.

For simplicity, the Lambda function used for the human interaction step automatically approves any task and returns the task token with a `SendTaskSuccess` API call. You can name the Lambda function as **callback-human-approval**.

1. In a new tab or window, open the [Lambda console](#) and create a **Node.js 16.x** Lambda function titled **callback-human-approval**. For information about creating a Lambda function using the console, see [Create a Lambda function in the console](#) in the *AWS Lambda Developer Guide*.

2. On the **callback-human-approval** page, replace the existing code in the **Code source** area with the following code.

```
// Sample Lambda function that will automatically approve any task whenever a
// message is published to an Amazon SNS topic by Step Functions.

console.log('Loading function');
const AWS = require('aws-sdk');
const resultMessage = "Successful";

exports.handler = async (event, context) => {
  const stepfunctions = new AWS.StepFunctions();

  let message = JSON.parse(event.Records[0].Sns.Message);
  let taskToken = message.TaskToken;

  console.log('Message received from SNS:', message);
  console.log('Task token: ', taskToken);

  // Return task token to Step Functions

  let params = {
    output: JSON.stringify(resultMessage),
    taskToken: taskToken
  };

  console.log('JSON Returned to Step Functions: ', params);
  let myResult = await stepfunctions.sendTaskSuccess(params).promise();
  console.log('State machine - callback completed..');

  return myResult;
};
```

3. Keep this window open and perform the steps in the next section for further actions.

Step 2.2: Add the Amazon SNS topic as a trigger for the Lambda function

When you add the Amazon SNS topic you created in [Step 1 of this tutorial](#) as a trigger for the Lambda function you created in [Step 2.1 of this tutorial](#), the Lambda function is triggered each time you publish to the Amazon SNS topic. When you publish to the Amazon SNS topic with a task token, the Lambda function is invoked with the payload of the published message. For more

information about configuring triggers for Lambda functions, see [Configuring triggers](#) in the *AWS Lambda Developer Guide*.

1. In the **Function overview** section of the `callback-human-approval` Lambda function, choose **Add trigger**.
2. From the drop-down list of triggers, choose **SNS** as the trigger.
3. For **SNS topic**, start typing the name of the Amazon SNS topic you created in [Step 1 of this tutorial](#), and choose it from the dropdown list that appears.
4. Choose **Add**.
5. Keep this window open and perform the steps in the next section for further actions.

Step 2.3: Provide necessary permissions to the Lambda function IAM role

You must provide the `callback-human-approval` Lambda function the permissions to access Step Functions for returning the task token along with the `SendTaskSuccess` API call.

1. On the `callback-human-approval` page, choose the **Configuration** tab, and then choose **Permissions**.
2. Under **Execution role**, choose the **Role name** to navigate to the AWS Identity and Access Management console's **Roles** page.
3. To add the required permission, choose **Add permissions**, and then choose **Attach policies**.
4. In the search box, type **AWSStepFunctions** and then press Enter.
5. Choose **AWSStepFunctionsFullAccess** and then scroll down to choose **Attach policies**. This adds the policy containing the necessary permission for the `callback-human-approval` Lambda function role.

Step 3: Update the workflow – add if-else condition logic in the Choice state

In the Step Functions console, define conditional logic for your workflow using the Choice state. If the output returned by the `RandomNumberForCredit` Lambda function is less than 5000, the requested credit is auto-approved. If the output returned is greater than or equal to 5000, the workflow execution proceeds to the human interaction step for the credit limit approval.

In the `Choice` state, you use a comparison operator to compare an input variable with a specific value. You can specify the input variable as the execution input while starting a state machine execution or use the output of a preceding step as input for the current step. By default, the output of a step is stored in a variable called `Payload`. To use the `Payload` variable's value for comparison in the `Choice` state, use the `$` syntax as shown in the following procedure.

For information about how information flows from one state to another and specifying input and output in your workflows, see [Tutorial 7: Configure input and output](#) and [Input and Output Processing in Step Functions](#).

Note

If the `Choice` state uses an input variable specified in the state machine execution input for comparison, use the `$.variable_name` syntax to perform the comparison. For example, to compare a variable, such as `myAge`, use the syntax `$.myAge`.

Because in this step, the `Choice` state will receive input from the **Get credit limit** state, you'll use the `$` syntax for the `Choice` state configuration. To explore how the result of the state machine execution differs when you use the `$.variable_name` syntax in the `Choice` state configuration to refer to the output from a preceding step, see the [Debugging the invalid path Choice state error](#) section in [Tutorial 8](#).

To add *if-else* condition logic using the `Choice` state

1. Open the [Step Functions console](#) window containing the workflow prototype you created in [Tutorial 1: Create the prototype for your state machine](#).
2. Choose the **Credit applied >= 5000?** state and in the **Configuration** tab, specify the conditional logic as follows:
 - a. Under **Choice Rules**, choose the **Edit** icon in the **Rule #1** tile to define the first choice rule.
 - b. Choose **Add conditions**.
 - c. In the **Conditions for rule #1** dialog box, for **Variable**, enter `$`.
 - d. For **Operator**, choose **is less than**.
 - e. For **Value**, choose **Number constant**, and then enter **5000** in the field next to the **Value** dropdown list.
 - f. Choose **Save conditions**.

- g. For the **Then next state is:** dropdown list, choose **Auto-approve limit**.
 - h. Choose **Add new choice rule**, and then define the second choice rule when the credit amount is greater than or equal to 5000 by repeating substeps 2.b through 2.f. For **Operator**, choose **is greater than or equal to**.
 - i. For the **Then next state is:** dropdown list, choose **Wait for human approval**.
 - j. In the **Default rule** box, choose the **Edit** icon to define the default choice rule, and then choose **Wait for human approval** from the **Default state** dropdown list. You define the Default rule to specify the next state to transition to if none of the **Choice** state conditions evaluate to true or false.
3. Configure the **Wait for human approval** state as follows:
 - a. In the **Configuration** tab, for **Topic**, start typing the name of the Amazon SNS topic, **TaskTokenTopic**, and choose the name as it appears in the dropdown list.
 - b. For **Message**, choose **Enter message** from the dropdown list. In the **Message** field, you specify the message you want to publish to the Amazon SNS topic. For this tutorial, you publish a task token as the message.

A task token lets you pause a **Standard** type Step Functions workflow until an external process is complete and the task token is returned. When you specify a **Task** state as a callback task by specifying the [.waitForTaskToken service integration pattern](#), a task token is generated and placed in the context object when the task is started. The context object is an internal JSON structure that is available during an execution, and contains information about your state machine and its execution. For more information about context objects, see [Context object](#).

- c. In the box that appears, enter the following as message:

```
{
  "TaskToken.$": "$$.Task.Token"
}
```

- d. Choose the **Wait for callback** checkbox.
 - e. Choose **Done** in the dialog box that appears.
4. Keep this window open and proceed to the next tutorial for further actions.

Next steps

In the next tutorial, you'll learn how to perform multiple tasks in parallel.

Tutorial 4: Define multiple tasks to perform in parallel

So far you've learned how to run workflows in a sequential manner. However, you can run two or more steps in parallel using the [Parallel](#) state. A `Parallel` state causes the interpreter to execute each branch concurrently.

Both the branches in a `Parallel` state receive the same input, but each branch processes the parts of input specific for it. Step Functions waits until each branch completes executing before proceeding to the next step.

In this tutorial, you use the **Parallel** state to concurrently check the identity and address of the applicant.

Topics

- [Step 1: Create the Lambda functions to perform the required checks](#)
- [Step 2: Update the workflow – Add parallel tasks to be performed](#)

Step 1: Create the Lambda functions to perform the required checks

This credit card application workflow invokes two Lambda functions inside the **Parallel** state to check the applicant's identity and address. These checks are performed simultaneously using the **Parallel** state. The state machine completes execution only after both the parallel branches have completed executing.

To create the check-identity and check-address Lambda functions

1. In a new tab or window, open the [Lambda console](#) and create two **Node.js 16.x** Lambda functions titled `check-identity` and `check-address`. For information about creating a Lambda function using the console, see [Create a Lambda function in the console](#) in the *AWS Lambda Developer Guide*.
2. Open the **check-identity** function page and replace the existing code in the **Code source** area with the following code:

```
const ssnRegex = /^\\d{3}-?\\d{2}-?\\d{4}$/;
```

```

const emailRegex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/;

class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomValidationError";
  }
}

exports.handler = async (event) => {
  const {
    ssn,
    email
  } = event;
  console.log(`SSN: ${ssn} and email: ${email}`);

  const approved = ssnRegex.test(ssn) && emailRegex.test(email);

  if (!approved) {
    throw new ValidationError("Check Identity Validation Failed");
  }

  return {
    statusCode: 200,
    body: JSON.stringify({
      approved,
      message: `Identity validation ${approved ? 'passed' : 'failed'}`
    })
  }
};

```

3. Open the **check-address** function page and replace the existing code in the **Code source** area with the following code:

```

class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomAddressValidationError";
  }
}

exports.handler = async event => {
  const {

```

```
    street,
    city,
    state,
    zip
  } = event;
  console.log(`Address information: ${street}, ${city}, ${state} - ${zip}`);

  const approved = [street, city, state, zip].every(i => i?.trim().length > 0);

  if (!approved) {
    throw new ValidationError("Check Address Validation Failed");
  }

  return {
    statusCode: 200,
    body: JSON.stringify({
      approved,
      message: `Address validation ${ approved ? 'passed' : 'failed'}`
    })
  }
};
```

4. For both the Lambda functions, from the **Function overview** section, copy their respective Amazon Resource Names (ARN) and save them in a text file. You'll need the function ARNs while specifying the service integration for the **Verify applicant's identity and address** state. The following is an example ARN:

```
arn:aws:lambda:us-east-2:123456789012:function:HelloWorld
```

Step 2: Update the workflow – Add parallel tasks to be performed

In the Step Functions console, you'll update your workflow to specify service integration with the **check-identity** and **check-address** Lambda functions you created in [Step 1](#).

To add parallel tasks in the workflow

1. Open the [Step Functions console](#) window containing the workflow prototype you created in [Tutorial 1: Create the prototype for your state machine](#).
2. Choose the **Verify identity** state, and in the **Configuration** tab, do the following:
 - a. For **Integration type**, keep the default selection of **Optimized**.

Note

Using Step Functions, you can integrate with other AWS services and orchestrate them in your workflows. For more information about service integrations and their types, see [Using AWS Step Functions with other services](#)

- b. For **Function name**, choose the **check-identity** Lambda function from the dropdown list.
- c. For **Payload**, choose **Enter payload** and then replace the example payload with the following as payload:

```
{
  "email": "janedoe@example.com",
  "ssn": "012-00-0000"
}
```

3. Choose the **Verify address** state, and in the **Configuration** tab, do the following:
 - a. For **Integration type**, keep the default selection of **Optimized**.
 - b. For **Function name**, choose the **check-address** Lambda function from the dropdown list.
 - c. For **Payload**, choose **Enter payload** and then replace the example payload with the following as payload:

```
{
  "street": "123 Any St",
  "city": "Any Town",
  "state": "AT",
  "zip": "01000"
}
```

4. Choose **Next**.

Tutorial 5: Concurrently iterate over a collection of items

In the previous tutorial, you learned how to run separate branches of steps in parallel using the [Parallel](#) state. Using the [Map](#) state, you can run a set of workflow steps for each item in a dataset. The Map state's iterations run in parallel, which makes it possible to process a dataset quickly.

By including the Map state in your workflows you can perform tasks, such as data processing, using one of the two [Map state processing modes](#): Inline mode and Distributed mode. To configure a Map state, you define an [ItemProcessor](#), which contains JSON objects that specify the Map state processing mode and its definition. In this tutorial, you run the Map state in the default [Inline mode](#), which supports up to 40 concurrent iterations. When you run the Map state in [Distributed mode](#), it supports up to 10,000 parallel child workflow executions.

When your workflow execution enters the Map state, it will iterate over a JSON array specified in the state input. For each array item, its corresponding iteration runs in the context of the workflow that contains the Map state. When all iterations are complete, the Map state will return an array containing the output for each item processed by the `ItemProcessor`.

In this tutorial, you learn how to use the Map state in Inline mode to fetch the credit score of an applicant by iterating over a set of credit bureaus. To do this, you first fetch the names of all the credit bureaus stored in a Amazon DynamoDB table, and then use the Map state to loop through the credit bureau list to fetch the applicant's credit score reported by each of these bureaus.

Topics

- [Step 1: Create a DynamoDB table to store the name of all credit bureaus](#)
- [Step 2: Update the state machine – Fetch results from the DynamoDB table](#)
- [Step 3: Create a Lambda function that returns the credit scores for all credit bureaus](#)
- [Step 4: Update the state machine – add a Map state to iteratively fetch credit scores](#)

Step 1: Create a DynamoDB table to store the name of all credit bureaus

In this step, you create a table named **GetCreditBureau** using the DynamoDB console. The table uses the string attribute **Name** as the **Partition** key. In this table, you store the name of all the credit bureaus from which you want to fetch the applicant's credit score.

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the console, choose **Tables**, and then choose **Create table**.
3. Enter the table details as follows:
 - a. For the **Table name**, enter **GetCreditBureau**.

- b. For the **Partition** key, enter **Name**.
 - c. Keep the default selections, and choose **Create table**.
4. After your table is created, in the **Tables** list, choose the **GetCreditBureau** table.
5. Choose **Actions**, and then choose **Create item**.
6. For **Value**, enter the name of a credit bureau. For example, **CredTrack**.
7. Choose **Create item**.
8. Repeat this process and create items for names of other credit bureaus. For example, **KapFinn** and **CapTrust**.

Step 2: Update the state machine – Fetch results from the DynamoDB table

In the Step Functions console, you'll add a [Task](#) state and use the [AWS SDK integration](#) to fetch the names of credit bureaus from the DynamoDB table you created in [Step 1](#). You'll use the output of this step as the input for the Map state you'll add later in your workflow in this tutorial.

1. Open the **CreditCardWorkflow** state machine to update it.
2. Choose the **Get list of credit bureaus** state.
3. For **API Parameters**, specify the **Table name** value as **GetCreditBureau**.

Step 3: Create a Lambda function that returns the credit scores for all credit bureaus

In this step, you create a Lambda function that receives the names of all credit bureaus as input, and returns the credit score of the applicant for each of these credit bureaus. This Lambda function will be invoked from the Map state you'll add in your workflow in Step 4 of this tutorial.

1. Create a **Node.js 16.x** Lambda function and name it **get-credit-score**.
2. On the page titled **get-credit-score**, paste the following code into the **Code source** area.

```
function getScore(arr) {
  let temp;
  let i = Math.floor((Math.random() * arr.length));
  temp = arr[i];
}
```

```
    console.log(i);
    console.log(temp);
    return temp;
}

const arrScores = [700, 820, 640, 460, 726, 850, 694, 721, 556];

exports.handler = (event, context, callback) => {
  let creditScore = getScore(arrScores);
  callback(null, "Credit score pulled is: " + creditScore + ".");
};
```

3. Deploy the Lambda function.

Step 4: Update the state machine – add a Map state to iteratively fetch credit scores

In the Step Functions console, you add a Map state that invokes the **get-credit-score** Lambda function to check the applicant's credit score for all the credit bureaus returned by the **Get list of credit bureaus** state.

1. Open the **CreditCardWorkflow** state machine to update it.
2. Choose the **Get scores from all credit bureaus** state.
3. In the **Configuration** tab, choose **Provide a path to items array** and then enter **\$.Items**.
4. Choose **Get all scores** step inside the Map state.
5. In the **Configuration** tab, make sure for **Integration type**, **Optimized** is selected.
6. For **Function** name, start typing the name of the **get-credit-score** Lambda function and choose it from the dropdown list that appears.
7. For **Payload**, choose **No payload**.

Tutorial 6: Save the workflow and execute the state machine

Now that you've configured the resources of all the AWS services you're using in the workflow prototype, you can save it as a Step Functions state machine and start executing it.

Topics

- [Step 1: Review the auto-generated state machine definition and save the state machine](#)

- [Step 2: Add the remaining IAM policies](#)
- [Step 3: Run the state machine](#)

Step 1: Review the auto-generated state machine definition and save the state machine

As you drag and drop states from the **Flow** tab onto the canvas in Workflow Studio to build the workflow prototype, Step Functions automatically composes the [Amazon States Language](#) (ASL) definition of your workflow in real-time. You can edit this definition as required in the [Code editor](#).

To review the ASL definition and save the state machine

1. (Optional) Choose **Definition** on the [Inspector](#) to view the state machine's [Amazon States Language](#) (ASL) definition, which is automatically generated based on your selections in the **Actions** and **Flow** tabs and **Inspector** panel.

Tip

To edit the definition, you can open the code editor by choosing **Code** on top of the page. For this tutorial, continue with the auto-generated definition.

2. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

For this tutorial, enter the name **CreditCardWorkflow**.

3. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

For this tutorial, keep all the default selections in **State machine settings**.

Note

(Optional) Step Functions automatically creates an execution role for the state machine with the least privileges required to invoke the `RandomNumberforCredit` Lambda function and publish to the Amazon SNS topic.

If you've [previously created an IAM role](#) with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

4. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

Note

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

Step 2: Add the remaining IAM policies

Because Step Functions doesn't auto-generate the permissions to invoke the Lambda functions used in the `Parallel` state, you need to add the necessary policy.

To add the remaining policy

1. On the **CreditCardWorkflow** page, choose the IAM role for your state machine to navigate to the IAM console. You'll add the necessary permissions for the remaining Lambda functions on this page.
2. Choose **Add permissions**, and then choose **Attach policies**.
3. In the search box, type **AWSLambdaRole** and then press Enter.
4. Choose **AWSLambdaRole** and then choose **Attach policies**. This policy is now added to the execution role of your state machine. This policy lets you invoke any Lambda function in your state machine.

Step 3: Run the state machine

State machine executions are instances where you run your workflow to perform tasks.

To execute the state machine

1. On the **CreditCardWorkflow** page, choose **Start execution**.

The **Start execution** dialog box is displayed.

2. In the **Start execution** dialog box, do the following:
 - a. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions automatically generates a unique execution name.

Note

Step Functions allows you to create names for state machines, executions, activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

Note

You don't need to provide any input to execute this state machine. But you can specify an execution input, if required, in the **Input** area of the **Start execution** dialog box for other state machines. For an example of how to provide execution input to a state machine, see [Step 4: Start a new execution](#) of the *Learn to use the AWS Step Functions Workflow Studio tutorial*.

- b. Choose **Start execution**.
3. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Tutorial 7: Configure input and output

A Step Functions execution receives a JSON text as input and passes that input to the first state in the workflow. Individual states in a workflow receive JSON data as input and usually pass JSON data as output to the next state. By default, data passes from one state to the next state in the workflow unless you've configured the input and/or output for one or more states in the workflow. Understanding how the information flows from state to another, and learning how to filter and manipulate this data, is key to effectively designing and implementing workflows in Step Functions.

Step Functions provides multiple filters to control the input and output data flow between states. The following filters are available for use in your workflows:

Note

Based on your use case, you may not need to apply all of these filters in your workflows.

InputPath

Selects *WHAT* portion of the entire input payload to be used as a task's input. If you specify this field, Step Functions first applies this field.

Parameters

Specifies *HOW* the input should look like before invoking the task. With the `Parameters` field, you can create a collection of key-value pairs that are passed as input to an [AWS service integration](#), such as an AWS Lambda function. These values can be static, or dynamically selected from either the state input or the [workflow context object](#).

ResultSelector

Determines *WHAT to choose* from a task's output. With the `ResultSelector` field, you can create a collection of key-value pairs that replace a state's result and pass that collection to `ResultPath`.

ResultPath

Determines *WHERE to put* a task's output. Use the `ResultPath` to determine whether the output of a state is a copy of its input, the result it produces, or a combination of both.

OutputPath

Determines *WHAT to send* to the next state. With `OutputPath`, you can filter out unwanted information, and pass only the portion of JSON data that you care about.

Tip

The `Parameters` and `ResultSelector` filters work by constructing JSON, whereas the `InputPath` and `OutputPath` filters work by filtering specific nodes within a JSON data object, and the `ResultPath` filter works by creating a field under which the output can be added.

In this tutorial, you learn how to perform the following tasks:

- [Select specific portions of the raw input using the `InputPath` filter](#)
- [Manipulate the selected input using the `Parameters` filter](#)
- [Configure output using the `ResultSelector`, `ResultPath`, and `OutputPath` filters](#)

For more information about configuring input and output in your workflows, see [Input and Output Processing in Step Functions](#).

Select specific portions of the raw input using the `InputPath` filter

Use the `InputPath` filter to select a specific portion of the input payload.

If you don't specify `InputPath`, its value defaults to `$`, which causes the state's task to refer to the entire raw input instead of a specific portion.

To learn how to use the `InputPath` filter, perform the following steps:

- [Step 1: Create a state machine](#)
- [Step 2: Run the state machine](#)
- [Step 3: Use the `InputPath` filter to select specific parts of an execution input](#)

Step 1: Create a state machine

Important

Ensure that your state machine is under the same AWS account and Region as the Lambda function you created earlier.

1. Use the `Parallel` state example you learned about in [Tutorial 4](#) to create a new state machine. Make sure your workflow prototype looks similar to the following prototype.
2. Configure the integrations for the `check-identity` and `check-address` Lambda functions. For information about creating the Lambda functions and using them in your state machine, see [Step 1: Create the Lambda functions to perform the required checks](#) and [Step 2: Update the workflow – Add parallel tasks to be performed](#).
3. For **Payload**, make sure you keep the default selection of **Use state input as payload**.
4. Choose **Next** and then do the steps 1 through 3 in [Step 1: Save the state machine](#) of [Tutorial 5](#) to create a new state machine. For this tutorial, name your state machine **WorkflowInputOutput**.

Step 2: Run the state machine

1. On the **WorkflowInputOutput** page, choose **Start execution**.
2. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions automatically generates a unique execution name.

Note

Step Functions allows you to create names for state machines, executions, activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

3. In the **Input** area, add the following JSON data as the execution input.

```
{
  "data": {
    "firstname": "Jane",
```

```
"lastname": "Doe",
"identity": {
  "email": "jdoe@example.com",
  "ssn": "123-45-6789"
},
"address": {
  "street": "123 Main St",
  "city": "Columbus",
  "state": "OH",
  "zip": "43219"
}
}
```

4. Choose **Start execution**.
5. The state machine execution results in an error because you've not specified what parts of the execution input the check-identity and check-address Lambda functions must use to perform the required identity and address verification.
6. Continue to [Step 3](#) of this tutorial to fix the error.

Step 3: Use the InputPath filter to select specific parts of an execution input

1. On the [Execution Details](#) page, choose **Edit state machine**.
2. To verify the applicant's identity as mentioned in the execution input provided in [Step 2: Run the state machine](#), edit the **Verify identity** task definition as follows:

```
...
{
  "StartAt": "Verify identity",
  "States": {
    "Verify identity": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "InputPath": "$.data.identity",
      "Parameters": {
        "Payload.$": "$",
        "FunctionName": "arn:aws:lambda:us-east-2:123456789012:function:check-identity:$LATEST"
      },
      "End": true
    }
  }
}
```

```
    }  
  }  
  ...
```

Consequently, the following JSON data becomes available as input for the `check-identity` function.

```
{  
  "email": "jdoe@example.com",  
  "ssn": "123-45-6789"  
}
```

3. To verify the applicant's address as mentioned in the execution input, edit the `Verify address` task definition as follows:

```
...  
{  
  "StartAt": "Verify address",  
  "States": {  
    "Verify address": {  
      "Type": "Task",  
      "Resource": "arn:aws:states:::lambda:invoke",  
      "InputPath": "$.data.address",  
      "Parameters": {  
        "Payload.$": "$",  
        "FunctionName": "arn:aws:lambda:us-east-1:123456789012:function:check-  
address:$LATEST"  
      },  
      "End": true  
    }  
  }  
}
```

Consequently, the following JSON data becomes available as input for the `check-address` function.

```
{  
  "street": "123 Main St",  
  "city": "Columbus",  
  "state": "OH",
```

```
"zip": "43219"
}
```

4. Choose **Start execution**. The state machine execution now completes successfully.

Manipulate the selected input using the Parameters filter

While the `InputPath` filter helps you limit the raw JSON input you provide, using the `Parameters` filter, you can pass a collection of key-value pairs as input. These key-value pairs can either be static values that you define in your state machine definition, or values that are selected from the raw input using `InputPath`.

In your workflows, `Parameters` are applied after `InputPath`. `Parameters` help you specify *how* the underlying task accepts its input payload. For example, if the `check-address` Lambda function accepts a string parameter as input instead of the JSON data, you can use the `Parameters` filter to transform the input.

In the following example, the `Parameters` filter receives the input you selected using `InputPath` in [Step 3: Use the `InputPath` filter to select specific parts of an execution input](#) and applies the intrinsic function `States.Format` on the input items to create a string called `addressString`. Intrinsic functions help you perform basic data processing operations on a given input. For more information, see [Intrinsic functions](#).

```
"Parameters": {
  "addressString.$": "States.Format('{} . {}, {} - {}'.format($.street, $.city, $.state,
$.zip)"
}
```

Consequently, the following string gets created and is provided to the `check-address` Lambda function as input.

```
{
  "addressString": "123 Main St. Columbus, OH - 43219"
}
```


Configure output using the ResultSelector, ResultPath, and OutputPath filters

When the check-address Lambda function is invoked in the **WorkflowInputOutput** state machine, the function returns an output payload after performing the address verification. On the [Execution Details](#) page, choose the **Verify address** step and view the output payload inside **Task result** on the [Step details](#) pane.

```
{
  "ExecutedVersion": "$LATEST",
  "Payload": {
    "statusCode": 200,
    "body": "{\"approved\":true,\"message\":\"identity validation passed\"}"
  },
  "SdkHttpMetadata": {
    "AllHttpHeaders": {
      "X-Amz-Executed-Version": [
        "$LATEST"
      ],
      ...
      ...
    }
  }
  "StatusCode": 200
}
```

Using ResultSelector

Now if you need to provide the result of the identity and address verification checks to the following states in your workflow, you can select the **Payload.body** node in the output JSON and use the `StringToJson` [intrinsic function](#) in the `ResultSelector` filter to format the data as required.

`ResultSelector` selects *what* is needed from the task output. In the following example, `ResultSelector` takes the string in **\$.Payload.body** and applies the `States.StringToJson` intrinsic function to convert the string to JSON and puts the resulting JSON inside the **identity** node.

```
"ResultSelector": {
  "identity.$": "States.StringToJson($.Payload.body)"
}
```

Consequently, the following JSON data is created.

```
{
  "identity": {
    "approved": true,
    "message": "Identity validation passed"
  }
}
```

As you work with these input and output filters, you can also encounter runtime errors arising because of specifying invalid JSON path expressions. For more information, see.

Using ResultPath

You can specify a location in the initial input payload to save a state's task processing result using the `ResultPath` field. If you don't specify `ResultPath`, its value defaults to `$`, which causes the initial input payload to be replaced with the raw task result. If you specify `ResultPath` as `null`, the raw result is discarded and the initial input payload becomes the effective output.

If you apply the `ResultPath` field on the JSON data created using the `ResultSelector` field, the task result is added inside the results node in the input payload as shown in the following example:

```
{
  "data": {
    "firstname": "Jane",
    "lastname": "Doe",
    "identity": {
      "email": "jdoe@example.com",
      "ssn": "123-45-6789"
    },
    "address": {
      "street": "123 Main St",
      "city": "Columbus",
      "state": "OH",
      "zip": "43219"
    },
    "results": {
      "identity": {
        "approved": true
      }
    }
  }
}
```

Using OutputPath

You can select a portion of the state output after the application of `ResultPath` to pass to the next state. This enables you to filter out unwanted information, and pass only the portion of JSON that you care about.

In the following example, the `OutputPath` field saves the state output inside the results node: `"OutputPath": "$.results"`. Consequently, the final output of the state, which you can pass to the next state is as follows:

```
{
  "addressResult": {
    "approved": true,
    "message": "address validation passed"
  },
  "identityResult": {
    "approved": true,
    "message": "identity validation passed"
  }
}
```

Using console features to visualize the input and output data flows

You can visualize the input and output data flow between the states in your workflows using the Step Functions console's [Data flow simulator](#) or Advanced view option in the Execution Details page.

Tutorial 8: Debug errors in the console

As you work with Step Functions, you might encounter runtime errors arising because of reasons, such as:

- An invalid JSON path for the **Variable** field in the Choice state.
- State machine definition issue, such as no matching rule defined for a Choice state.
- Invalid JSON path expressions while applying filters to manipulate input and output.
- Task failures because of a Lambda function exception.
- IAM permission errors.

In this tutorial, you'll learn about debugging some of these errors using the Step Functions console. For more information, see [Error handling in Step Functions](#).

Topics

- [Debugging the invalid path Choice state error](#)
- [Debugging JSON path expression errors while applying input and output filters](#)

Debugging the invalid path Choice state error

When you specify an incorrect or unresolvable JSON path in the **Variable** field of the Choice state or do not define a matching rule in the Choice state, you receive an error while running your workflow.

To illustrate the invalid path error, this tutorial introduces a Choice state error in your workflow. You'll use the **CreditCardWorkflow** state machine and edit its definition to introduce the error.

1. Open the Step Functions console and then choose the **CreditCardWorkflow** state machine.
2. Choose **Edit** to edit the state machine definition. Make the change highlighted in the following code to your state machine definition.

```
{
  "Comment": "A description of my state machine",
  "StartAt": "Get credit limit",
  "States": {
    "Get credit limit": {
      ...
    },
    "Credit applied >= 5000?": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.Payload",
          "NumericLessThan": 5000,
          "Next": "Auto-approve limit"
        },
        {
          "Variable": "$.Payload",
          "NumericGreaterThanEquals": 5000,
          "Next": "Wait for human approval"
        }
      ]
    }
  }
}
```

```

    }
  ],
  "Default": "Wait for human approval"
},
...
...
}
}

```

3. Choose **Save** and then choose **Save anyway**.
4. Run the state machine.
5. On the Execution Details page of your state machine execution, do one of the following:
 - a. Choose **Cause** on the error message to view the reason for execution failure.
 - b. Choose **Show step detail** on the error message to view the step that caused the error.
6. In the **Input & Output** tab of the **Step details** section, choose the **Advanced view** toggle button to see the input and output data transfer path for a selected state.
7. In **Graph view**, make sure **Credit applied >= 5000?** is selected and do the following:
 - a. View the state's input value in **Input** box.
 - b. Choose the **Definition** tab, and notice the JSON path specified for the **Variable** field.

The input value for the **Credit applied >= 5000?** state is a numeric value, while you've specified the JSON path for the input value as `$.Payload`. During the state machine execution, the Choice state cannot resolve this JSON path because it doesn't exist.

8. Edit the state machine to specify the **Variable** field value as `$`.

```

{
  "Comment": "A description of my state machine",
  "StartAt": "Get credit limit",
  "States": {
    "Get credit limit": {
      ...
      ...
    },
    "Credit applied >= 5000?": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$",

```

```

        "NumericLessThan": 5000,
        "Next": "Auto-approve limit"
    },
    {
        "Variable": "$",
        "NumericGreaterThanEquals": 5000,
        "Next": "Wait for human approval"
    }
],
"Default": "Wait for human approval"
},
...
...
}
}

```

Debugging JSON path expression errors while applying input and output filters

As you work with the input and output filters, you might encounter runtime errors arising because of specifying invalid JSON path expressions.

The following example uses the **WorkflowInputOutput** state machine you created in [Tutorial 5](#) and demonstrates a scenario where you use the `ResultSelector` filter to select portions of the task output.

1. Apply the `ResultSelector` filter to choose a portion of the task output for the **Verify identity** step. To do this, edit your state machine definition as follows:

```

{
  "StartAt": "Verify identity",
  "States": {
    "Verify identity": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Parameters": {
        "FunctionName": "arn:aws:lambda:us-east-2:123456789012:function:check-identity",
        "Payload": {
          "email": "jdoe@example.com",
          "ssn": "123-45-6789"
        }
      }
    }
  }
}

```

```
    }
  },
  ...
  ...
  "ResultSelector": {
    "identity.$": "$.Payload.body.message"
  },
  "End": true
}
}
```

2. Run the state machine.
3. On the Execution Details page of your state machine execution, do the following:
 - a. Choose **Cause** on the error message to view the reason for execution failure.
 - b. Choose **Show step detail** on the error message to view the step that caused the error.
4. In the error message, note that the contents of the **\$.Payload.body** node is an escaped JSON string. The error has occurred because you cannot refer to a string using the JSON path notation.
5. To refer to the **\$.Payload.body.message** node, do the following:
 - a. Use the [States.StringToJson](#) intrinsic function to first convert the string to a JSON format.
 - b. Specify the JSON path for the **\$.Payload.body.message** node inside the intrinsic function.

```
"ResultSelector": {
  "identity.$": "States.StringToJson($.Payload.body.message)"
}
```

6. Run the state machine again.

Use cases

AWS Step Functions lets you build visual workflows that help rapidly translate business requirements into applications. Step Functions manages state, checkpoints and restarts for you, and provides built-in capabilities to automatically deal with errors and exceptions. To better understand the capabilities Step Functions can provide you with, read through the following use cases:

Topics

- [Data processing](#)
- [Machine learning](#)
- [Microservice orchestration](#)
- [IT and security automation](#)

Data processing

As the volume of data grows, coming from increasingly diverse sources, organizations find they need to move quickly to process this data to ensure they make faster, well-informed business decisions. To process data at scale, organizations need to elastically provision resources to manage the information they receive from mobile devices, applications, satellites, marketing and sales, operational data stores, infrastructure, and more.

Step Functions provides the scalability, reliability, and availability needed to successfully manage your data processing workflows. You can manage millions of concurrent executions with Step Functions as it scales horizontally and provides fault-tolerant workflows. Process data faster using parallel executions like Step Functions' [Parallel](#) state type, or dynamic parallelism using its [Map](#) state type. As part of your workflow, you can use the [Map](#) state to iterate over objects in a static data store like an Amazon S3 bucket. Step Functions also lets you easily retry failed executions, or choose a specific way to handle errors without the need to manage a complex process.

Depending upon your data processing needs, Step Functions directly integrates with other data processing services provided by AWS such as [AWS Batch](#) for batch processing, [Amazon EMR](#) for big data processing, [AWS Glue](#) for data preparation, [Athena](#) for data analysis, and [AWS Lambda](#) for compute.

Examples of the types of data processing workflows that customers use Step Functions to accomplish include:

File, video, and image processing

- Take a collection of video files and convert them to other sizes or resolutions that are ideal for the device they will be displayed on, such as mobile phones, laptops, or a television.
- Take a large collection of photos uploaded by users and convert them into thumbnails or various resolution images that can then be displayed on users' websites.
- Take semi-structured data, such as a CSV file, and combine it with unstructured data, such as an invoice, to produce a business report that is sent to business stakeholders monthly.
- Take earth observing data collected from satellites, convert it into formats that align with each other and then add other data sources collected on earth for additional insight.
- Take the transportation logs from various modes of transportation for products and look for optimizations using Monte Carlo Simulations and then send reports back to the organizations and people that are relying on you to ship their goods.

Coordinate extract, transform and load (ETL) jobs:

- Combine sales opportunity records with marketing metric datasets through a series of data preparation steps using AWS Glue, and produce business intelligence reports that can be used across the organization.
- Create, start, and terminate an Amazon EMR cluster for big data processing.

Batch processing and High Performance Computing (HPC) workloads:

- Build a genomics secondary analysis pipeline that processes raw whole genome sequences into variant calls. Align raw files to a reference sequence, and call variants on a specified list of chromosomes using dynamic parallelism.
- Find efficiencies in the production of your next mobile device or other electronics by simulating various layouts using different electric and chemical compounds. Run large batch processing of your workloads through various simulations to get the optimal design.

Machine learning

Machine learning enables organizations to quickly analyze collected data to identify patterns, then make decisions with minimal human intervention. Machine learning starts with an initial set of data, known as training data. This training data helps to increase a machine learning model's prediction accuracy, and serves as the foundation through which this model learns. Once the model is considered accurate enough to meet business needs, it's deployed to production. The [AWS Step Functions Data Science Software Development Kit \(SDK\)](#) is an open-source library that allows you to easily create workflows that preprocess data, train and then publish your models using Amazon SageMaker and Step Functions.

Preprocessing existing data sets is how an organization often creates training data. This method adds information, such as by labeling objects in an image, annotating text or processing audio. To preprocess data you can use AWS Glue, or you can create an SageMaker notebook instance that runs the Jupyter Notebook app. Once your data is ready, it can be uploaded to Amazon S3 for easy access. As machine learning models are trained, you can make adjustments to each model's parameters to improve accuracy until it's ready for deployment.

Step Functions lets you to orchestrate end-to-end machine learning workflows on SageMaker. These workflows can include data preprocessing, post-processing, feature engineering, data validation, and model evaluation. Once the model has been deployed to production, you can refine and test new approaches to continually improve business outcomes. You can create production-ready workflows directly in Python, or you can use the Step Functions Data Science SDK to copy that workflow, experiment with new options, and place the refined workflow in production.

Some types of machine learning workflows that customers use Step Functions for include:

Fraud Detection

- Identify and prevent fraudulent transactions, such as credit fraud, from occurring.
- Detect and prevent account takeovers using trained machine learning models.
- Identify promotional abuse, including the creation of fake accounts, so you can quickly take action.

Personalization and Recommendations

- Recommend products to targeted customers based upon what is predicted to attract their interest.

- Predict whether a customer will upgrade their account from a free tier to a paid subscription.

Data Enrichment

- Use data enrichment as part of preprocessing to provide better training data for more accurate machine learning models.
- Annotate text and audio excerpts to add syntactical information, such as sarcasm and slang.
- Label additional objects in images to provide critical information for the model to learn from, such as whether an object is an apple, a basketball, a rock, or an animal.

Microservice orchestration

Microservice architecture breaks applications into loosely coupled services. Benefits include improved scalability, increased resiliency, and faster time to market. Each microservice is independent, making it easy to scale up a single service or function without needing to scale the entire application. Individual services are loosely coupled, letting independent teams focus on a single business process, without the need for them to understand the entire application. Microservices also let you choose which individual components suit your business needs, giving you the flexibility to change your selection without rewriting your entire workflow. Different teams can use the programming languages and frameworks of their choice to work with their microservice, and this microservice can still communicate with any other in the application through application programming interfaces (APIs).

Step Functions gives you several ways to manage your microservice workflows. For long-running workflows you can use Standard Workflows with the AWS Fargate integration to orchestrate applications running in containers. For short-duration, high-volume workflows that require an immediate response, [Synchronous Express Workflows](#) are ideal. These can be used for web-based or mobile applications, which often have workflows of short duration, and require the completion of a series of steps before they return a response. You can directly trigger a Synchronous Express Workflows from Amazon API Gateway, and the connection is held open until the workflow completes or timeouts. For short duration workflows that do not require an immediate response, Step Functions provides Asynchronous Express Workflows.

Examples of some API orchestrations that use Step Functions include:

Synchronous or real-time workflows

- Change a value in a record such as updating an employee's last name and have the change immediately visible on the screen.
- Update an order during checkout, such as adding, removing, or changing the quantity of an item, then immediately reflect the update back to the customer.
- Run a quick processing job and immediately return the result back to the requester.

Container Orchestration

- Run jobs on Kubernetes with Amazon Elastic Kubernetes Service or on Amazon Elastic Container Service (ECS) with Fargate and integrate with other AWS services, such as sending notifications with Amazon SNS, as part of the same workflow.

IT and security automation

IT automation can help manage increasingly complex and time-consuming operations, such as upgrading and patching software, deploying security updates to address vulnerabilities, selecting infrastructure, synchronizing data, routing support tickets, and more. The automation of repetitive and time-consuming tasks can allow your organization to complete routine operations quickly and consistently on a large scale. This lets you focus on strategic work such as feature development, complex support requests, and innovation while meeting these growing demands.

Step Functions allows you to create workflows that automatically scale to meet the needs of your business without requiring manual intervention. In cases where an error occurs in your workflow, it often does not require manual intervention. Step Functions lets you automatically [retry failed tasks](#) and [an exponential backoff](#) that can manage errors in your workflow.

There can be situations where human intervention is required before the workflow can progress. For example, approving a substantial credit increase may require human approval. To manage this, you can define branching logic in Step Functions, so that only requests over a defined amount require human approval, while all other requests are automatically completed. In cases where human approval is required, Step Functions lets you pause the workflow at a specific step, wait for a response, and then continue the workflow once the response is received.

Some examples of the types of automation workflows that customers use Step Functions for include:

IT automation

- Auto-remediate incidents like opening an SSH port, low disk space, or when a public access is given to a Amazon S3 bucket.
- Automate the deployment of AWS CloudFormation StackSets

Security automation

- Automate the response to a scenario where a user and user access key has been exposed.
- Auto-remediate security incident responses according to policy actions defined such as restricting actions to specific ARNs or applying other actions.
- Warn employees of phishing emails within seconds of receipt.

Human Approval

- Automate the training of machine learning model and then require manual approval of the model by a data scientist before then automatically deploying or rejecting the model based upon the response received.
- Automate the routing of customer feedback received based on sentiment analysis so that those with a negative sentiment are immediately escalated for manual review.

How Step Functions works

This section describes important concepts to help you get familiar with AWS Step Functions and understand how it works.

Topics

- [Standard vs. Express Workflows](#)
- [States](#)
- [Map state processing modes](#)
- [Tolerated failure threshold for Distributed Map state](#)
- [Transitions](#)
- [State Machine Data](#)
- [Input and Output Processing in Step Functions](#)
- [Data flow simulator](#)
- [Manage continuous deployments with versions and aliases](#)
- [Executions in Step Functions](#)
- [Error handling in Step Functions](#)
- [Invoke AWS Step Functions from other services](#)
- [Read Consistency in Step Functions](#)
- [Tagging in Step Functions](#)

Standard vs. Express Workflows

When you create a state machine, you select a **Type** of either **Standard** or **Express**. The default **Type** for state machines is **Standard**. A state machine whose **Type** is **Standard** is called a *Standard workflow* and a state machine whose **Type** is **Express** is called an *Express workflow*.

For both Standard and Express workflows, you define your state machine using the [Amazon States Language](#). Your state machine executions will behave differently depending on the **Type** that you select.

⚠ Important

The **Type** you choose can't be changed after you create the state machine.

ℹ Note

If you define your state machines outside the Step Functions' console, such as in an editor of your choice, you must save your state machine definitions with the extension `.asl.json`.

Standard Workflows are ideal for long-running (up to one year), durable, and auditable workflows. You can retrieve the full execution history using the [Step Functions API](#) for up to 90 days after your execution completes. Standard Workflows follow an *exactly-once* model, where your tasks and states are never run more than once, unless you have specified `Retry` behavior in ASL. This makes Standard Workflows suited to orchestrating non-idempotent actions, such as starting an Amazon EMR cluster or processing payments. Standard Workflow executions are billed according to the number of state transitions processed.

Express Workflows are ideal for high-volume, event-processing workloads such as IoT data ingestion, streaming data processing and transformation, and mobile application backends. They can run for up to five minutes. Express Workflows employ an *at-least-once* model, where an execution could potentially run more than once. This makes Express Workflows ideal for orchestrating idempotent actions such as transforming input data and storing by way of a `PUT` action in Amazon DynamoDB. Express Workflow executions are billed by the number of executions, the duration of execution, and the memory consumed while the execution ran.

Standard and Express Workflows can automatically start in response to events such as HTTP requests from Amazon API Gateway (fully-managed APIs at scale), IoT Rules and over 140 other event sources in Amazon EventBridge.

ℹ Tip

To deploy an example of an Express workflow to your AWS account, see [Module 7 - API Gateway, Parallel State, Express workflows](#) of *The AWS Step Functions Workshop*.

For information about the console experience for Standard and Express Workflow executions, see [Standard and Express Workflow executions in the console](#).

Standard vs Express Workflows

	Standard Workflows	Express Workflows: Synchronous and Asynchronous
Maximum duration	One year	Five minutes
Supported execution start rate	For information about quotas related to supported execution start rate, see Quotas related to API action throttling .	For information about quotas related to supported execution start rate, see Quotas related to API action throttling .
Supported state transition rate	For information about quotas related to supported state transition rate, see Quotas related to state throttling .	No limit
Pricing	Priced by number of state transitions. A <i>state transition</i> is counted each time a step in your execution is completed.	Priced by the number of executions you run, their duration, and memory consumption.
Execution history	<p>Executions can be listed and described with Step Functions APIs. Executions can be visually debugged through the console. They can also be inspected in CloudWatch Logs by enabling logging on your state machine.</p> <p>For more information about debugging Standard Workflow executions in the</p>	<p>Unlimited execution history, that is, as many execution history entries are maintained as you can generate within a 5-minute period.</p> <p>Executions can be inspected in CloudWatch Logs or the Step Functions console by enabling logging on your state machine.</p>

	Standard Workflows	Express Workflows: Synchronous and Asynchronous
	console, see Standard and Express Workflow executions in the console and Viewing and debugging executions .	For more information about debugging Express Workflow executions in the console, see Standard and Express Workflow executions in the console and Viewing and debugging executions .
Execution semantics	<i>Exactly-once</i> workflow execution.	<i>Asynchronous Express Workflows: At-least-once</i> workflow execution. <i>Synchronous Express Workflows: At-most-once</i> workflow execution.
Service integrations	Supports all service integrations and patterns.	Supports all service integrations. <div style="border: 1px solid #0070C0; border-radius: 10px; padding: 10px;">Note Express Workflows do not support Job-run (.sync) or Callback (.waitForTaskToken) service integration patterns.</div>
Step Functions activities	Supports Step Functions activities.	Doesn't support Step Functions activities.

Synchronous and Asynchronous Express Workflows

There are two types of Express Workflows that you can choose: Asynchronous Express Workflows and Synchronous Express Workflows.

- **Asynchronous Express Workflows** return confirmation that the workflow was started, but don't wait for the workflow to complete. To get the result, you must poll the service's [CloudWatch Logs](#). You can use Asynchronous Express Workflows when you don't require immediate response output, such as messaging services or data processing that other services don't depend on. You can start Asynchronous Express Workflows in response to an event, by a nested workflow in Step Functions, or by using the [StartExecution](#) API call.
- **Synchronous Express Workflows** start a workflow, wait until it completes, and then return the result. Synchronous Express Workflows can be used to orchestrate microservices. With Synchronous Express Workflows, you can develop applications without the need to develop additional code to handle errors, retries, or run parallel tasks. You can run Synchronous Express Workflows invoked from Amazon API Gateway, AWS Lambda, or by using the [StartSyncExecution](#) API call.

Note

If you run Step Functions Express Workflows synchronously from the console, the `StartSyncExecution` request elapses after 60 seconds. To run the Express Workflows synchronously for a duration of up to five minutes, make the `StartSyncExecution` request using the AWS SDK or AWS Command Line Interface (AWS CLI) instead of the Step Functions console.

Synchronous Express execution API calls don't contribute to existing account capacity limits. Step Functions provides capacity on demand and automatically scales with sustained workload. Surges in workload may be throttled until capacity is available.

Execution guarantees

Standard Workflows	Asynchronous Express Workflows	Synchronous Express Workflows	
<i>Exactly-once</i> workflow execution	<i>At-least-once</i> workflow execution	<i>At-most-once</i> workflow execution	
Execution state internally persists between state transitions.	Execution state doesn't persist between state transitions.	Execution state doesn't persist between state transitions.	
Automatically returns an idempotent response on starting an execution with the same name as a currently-running workflow. The new workflow doesn't start and an exception is thrown once the currently-running workflow is complete.	Idempotency is not automatically managed. Starting multiple workflows with the same name results in concurrent executions. Can result in loss of internal workflow state if state machine logic is not idempotent.	Idempotency is not automatically managed. Step Functions waits once an execution starts and returns the state machine's result on completion. Workflows don't restart if an exception occurs.	
Execution history data removed after 90 days. Workflow names can be reused after removal of out-of-date execution data. To meet compliance, organizational, or regulatory requirements,	Execution history is not captured by Step Functions. Logging must be enabled through Amazon CloudWatch Logs.	Execution history is not captured by Step Functions. Logging must be enabled through Amazon CloudWatch Logs.	

Standard Workflows	Asynchronous Express Workflows	Synchronous Express Workflows	
<p>nts, you can reduce the execution history retention period to 30 days by sending a quota request. To do this, use the AWS Support Center Console and create a new case.</p>			

Cost-optimization using Express Workflows

Step Functions determines pricing for Standard and Express workflows based on the workflow type you use to build your state machines. To optimize the cost of your serverless workflows, you can follow either or both of the following recommendations:

Topics

- [Tip #1: Nesting Express workflows inside Standard workflows](#)
- [Tip #2: Convert Standard workflows into Express workflows](#)

For information about how choosing a Standard or Express workflow type affects billing, see [AWS Step Functions Pricing](#).

Tip #1: Nesting Express workflows inside Standard workflows

Step Functions runs workflows that have a finite duration and number of steps. Some workflows may complete execution within a short period of time. Others may require a combination of both long-running and high-event-rate workflows. With Step Functions, you can build large, complex workflows out of multiple smaller, simpler workflows.

For example, to build an order processing workflow, you can include all non-idempotent actions into a Standard workflow. This could include actions, such as approving order through human interaction and processing payments. You can then combine a series of idempotent actions, such as sending payment notifications and updating product inventory, in an Express workflow. You can

nest this Express workflow within the Standard workflow. In this example, the Standard workflow is known as the *parent state machine*. The nested Express workflow is known as a *child state machine*.

Tip #2: Convert Standard workflows into Express workflows

You can convert your existing Standard workflows into Express workflows if they meet the following requirements:

- The workflow must complete its execution within five minutes.
- The workflow conforms to an *at-least-once* execution model. This means that each step in the workflow may run more than exactly once.
- The workflow doesn't use the [.waitForTaskToken](#) or [.sync](#) service integration patterns.

Important

Express workflows use Amazon CloudWatch Logs to record execution histories. You will incur additional costs when using CloudWatch Logs.

To convert a Standard workflow into an Express workflow using the console

1. Open the [Step Functions console](#).
2. On the **State machines** page, choose a Standard type state machine to open it.

Tip

From the **Any type** dropdown list, choose **Standard** to filter the state machines list and view only Standard workflows.

3. Choose **Copy to new**.

Workflow Studio opens in [Design mode](#) displaying workflow of the state machine you selected.

4. (Optional) Update the workflow design.
5. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

6. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

Make sure that for **Type**, you choose **Express**. Keep all the other default selections on **State machine settings**.

Note

If you're converting a Standard workflow previously defined in [AWS CDK](#) or AWS SAM, you must change the value of Type and Resource name.

7. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

Note

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

For more information about best practices and guidelines when you manage cost-optimization for your workflows, see [Building cost-effective AWS Step Functions workflows](#).

States

Individual states can make decisions based on their input, perform actions from those inputs, and pass output to other states. In AWS Step Functions, you define your workflows in the Amazon States Language (ASL). The Step Functions console provides a graphical representation of your state machine to help visualize your application's logic.

Note

If you define your state machines outside the Step Functions' console, such as in an editor of your choice, you must save your state machine definitions with the extension `.asl.json`.

States are elements in your state machine. A state is referred to by its *name*, which can be any string, but which must be unique within the scope of the entire state machine.

States can perform a variety of functions in your state machine:

- Do some work in your state machine (a [Task](#) state)
- Make a choice between branches of execution (a [Choice](#) state)
- Stop an execution with a failure or success (a [Fail](#) or [Succeed](#) state)
- Pass its input to its output, or inject some fixed data into the workflow (a [Pass](#) state)
- Provide a delay for a certain amount of time, or until a specified date and time (a [Wait](#) state)
- Begin parallel branches of execution (a [Parallel](#) state)
- Dynamically iterate steps (a [Map](#) state)

The following is an example state named `HelloWorld` that performs an AWS Lambda function.

```
"HelloWorld": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloFunction",
  "Next": "AfterHelloWorldState",
  "Comment": "Run the HelloWorld Lambda function"
}
```

States share many common features:

- A `Type` field indicating what type of state it is.
- An optional `Comment` field to hold a human-readable comment about, or description of, the state.
- Each state (except a `Succeed` or `Fail` state) requires a `Next` field or, alternatively, can become a terminal state by specifying an `End` field.

Note

A `Choice` state may have more than one `Next`, but only one within each `Choice Rule`. A `Choice` state can't use `End`.

Certain state types require additional fields, or may redefine common field usage.

After you have created and run Standard workflows, you can access information about each state, its input and output, when it was active and for how long, by viewing the **Execution Details** page in the [Step Functions console](#). For more information, see [Viewing and debugging executions on the Step Functions console](#).

After you have created and run Express Workflow executions and if logging is enabled for your Express Workflow, you can [access information about the execution in Amazon CloudWatch Logs](#) or the Step Functions console. For more information, see [Viewing and debugging executions on the Step Functions console](#).

Topics

- [Amazon States Language](#)
- [Pass](#)
- [Task state](#)
- [Choice](#)
- [Wait](#)
- [Succeed](#)
- [Fail](#)
- [Parallel](#)
- [Map](#)

Amazon States Language

The Amazon States Language is a JSON-based, structured language used to define your state machine, a collection of [states](#), that can do work (Task states), determine which states to transition to next (Choice states), stop an execution with an error (Fail states), and so on.

For more information, see the [Amazon States Language Specification](#) and [Statelint](#), a tool that validates Amazon States Language code.

To create a state machine on the [Step Functions console](#) using Amazon States Language, see [Getting Started](#).

Note

If you define your state machines outside the Step Functions' console, such as in an editor of your choice, you must save your state machine definitions with the extension `.asl.json`.

Example Amazon States Language Specification

```
{
  "Comment": "An example of the Amazon States Language using a choice state.",
  "StartAt": "FirstState",
  "States": {
    "FirstState": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FUNCTION_NAME",
      "Next": "ChoiceState"
    },
    "ChoiceState": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.foo",
          "NumericEquals": 1,
          "Next": "FirstMatchState"
        },
        {
          "Variable": "$.foo",
          "NumericEquals": 2,
          "Next": "SecondMatchState"
        }
      ],
      "Default": "DefaultState"
    },
    "FirstMatchState": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:OnFirstMatch",
      "Next": "NextState"
    },
    "SecondMatchState": {
      "Type": "Task",

```

```
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:OnSecondMatch",
    "Next": "NextState"
  },

  "DefaultState": {
    "Type": "Fail",
    "Error": "DefaultStateError",
    "Cause": "No Matches!"
  },

  "NextState": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FUNCTION_NAME",
    "End": true
  }
}
```

Topics

- [State machine structure](#)
- [Intrinsic functions](#)
- [Common State Fields](#)

State machine structure

State machines are defined using JSON text that represents a structure containing the following fields.

Comment (Optional)

A human-readable description of the state machine.

StartAt (Required)

A string that must exactly match (is case sensitive) the name of one of the state objects.

TimeoutSeconds (Optional)

The maximum number of seconds an execution of the state machine can run. If it runs longer than the specified time, the execution fails with a `States.Timeout` [Error Name](#).

Version (Optional)

The version of the Amazon States Language used in the state machine (default is "1.0").

States (Required)

An object containing a comma-delimited set of states.

The States field contains [States](#).

```
{
  "State1" : {
  },
  "State2" : {
  },
  ...
}
```

A state machine is defined by the states it contains and the relationships between them.

The following is an example.

```
{
  "Comment": "A Hello World example of the Amazon States Language using a Pass state",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Pass",
      "Result": "Hello World!",
      "End": true
    }
  }
}
```

When an execution of this state machine is launched, the system begins with the state referenced in the `StartAt` field ("HelloWorld"). If this state has an `"End": true` field, the execution stops and returns a result. Otherwise, the system looks for a `"Next":` field and continues with that state next. This process repeats until the system reaches a terminal state (a state with `"Type": "Succeed", "Type": "Fail", or "End": true`), or a runtime error occurs.

The following rules apply to states within a state machine:

- States can occur in any order within the enclosing block, but the order in which they're listed doesn't affect the order in which they're run. The contents of the states determines this order.
- Within a state machine, there can be only one state that's designated as the `start` state, designated by the value of the `StartAt` field in the top-level structure. This state is the one that is executed first when the execution starts.
- Any state for which the `End` field is `true` is considered an end (or `terminal`) state. Depending on your state machine logic—for example, if your state machine has multiple branches of execution—you might have more than one end state.
- If your state machine consists of only one state, it can be both the `start` state and the end state.

Intrinsic functions

The Amazon States Language provides several intrinsic functions, also known as *intrinsic*s, that help you perform basic data processing operations without using a Task state. Intrinsic functions are constructs that look similar to functions in programming languages. They can be used to help payload builders process the data going to and from the `Resource` field of a Task state.

In Amazon States Language, intrinsic functions are grouped into the following categories, based on the type of data processing task that you want to perform:

- [Intrinsic functions for arrays](#)
- [Intrinsic functions for data encoding and decoding](#)
- [Intrinsic function for hash calculation](#)
- [Intrinsic functions for JSON data manipulation](#)
- [Intrinsic functions for Math operations](#)
- [Intrinsic function for String operation](#)
- [Intrinsic function for unique identifier generation](#)
- [Intrinsic function for generic operation](#)

Note

- To use intrinsic functions you must specify `.$` in the key value in your state machine definitions, as shown in the following example:

```
"KeyId.$": "States.Array($.Id)"
```

- You can nest up to 10 intrinsic functions within a field in your workflows. The following example shows a field named *myArn* that includes nine nested intrinsic functions:

```
"myArn.$": "States.Format('{}.{}.{}'.  
States.ArrayGetItem(States.StringSplit(States.ArrayGetItem(States.StringSplit($.ImageRe  
'/'), 2), '.'), 0),  
States.ArrayGetItem(States.StringSplit(States.ArrayGetItem(States.StringSplit($.ImageRe
```

Tip

If you use Step Functions in a [local development environment](#), make sure you're using [version 1.12.0](#) or higher to be able to include all the intrinsic functions in your workflows.

Fields that support intrinsic functions

The following table shows which fields support intrinsic functions for each state.

Fields that support intrinsic functions

State

	Pass	Task	Choice	Wait	Succeed	Fail	Parallel	Map
InputPath								
Parameters	✓	✓					✓	✓
ResultSelector		✓					✓	✓
ResultPath								

State

Pass Task Choice Wait Succeed Fail Parallel Map

OutputPath

Variable

**<Comparison
Operator>
Path**

TimeoutSecondsPath

HeartbeatSecondsPath

Credentials ✓

Intrinsics for arrays

Use the following intrinsics for performing array manipulations.

States.Array

The `States.Array` intrinsic function takes zero or more arguments. The interpreter returns a JSON array containing the values of the arguments in the order provided. For example, given the following input:

```
{
  "Id": 123456
}
```

You could use

```
"BuildId.$": "States.Array($.Id)"
```

Which would return the following result:

```
"BuildId": [123456]
```

States.ArrayPartition

Use the `States.ArrayPartition` intrinsic function to partition a large array. You can also use this intrinsic to slice the data and then send the payload in smaller chunks.

This intrinsic function takes two arguments. The first argument is an array, while the second argument defines the chunk size. The interpreter chunks the input array into multiple arrays of the size specified by chunk size. The length of the last array chunk may be less than the length of the previous array chunks if the number of remaining items in the array is smaller than the chunk size.

Input validation

- You must specify an array as the input value for the function's first argument.
- You must specify a non-zero, positive integer for the second argument representing the chunk size value.

If you specify a non-integer value for the second argument, Step Functions will round it off to the nearest integer.

- The input array can't exceed Step Functions' payload size limit of 256 KB.

For example, given the following input array:

```
{"inputArray": [1,2,3,4,5,6,7,8,9] }
```

You could use the `States.ArrayPartition` function to divide the array into chunks of four values:

```
"inputArray.$": "States.ArrayPartition($.inputArray,4)"
```

Which would return the following array chunks:

```
{"inputArray": [ [1,2,3,4], [5,6,7,8], [9]] }
```

In the previous example, the `States.ArrayPartition` function outputs three arrays. The first two arrays each contain four values, as defined by the chunk size. A third array contains the remaining value and is smaller than the defined chunk size.

States.ArrayContains

Use the `States.ArrayContains` intrinsic function to determine if a specific value is present in an array. For example, you can use this function to detect if there was an error in a Map state iteration.

This intrinsic function takes two arguments. The first argument is an array, while the second argument is the value to be searched for within the array.

Input validation

- You must specify an array as the input value for function's first argument.
- You must specify a valid JSON object as the second argument.
- The input array can't exceed Step Functions' payload size limit of 256 KB.

For example, given the following input array:

```
{
  "inputArray": [1,2,3,4,5,6,7,8,9],
  "lookingFor": 5
}
```

You could use the `States.ArrayContains` function to find the `lookingFor` value within the `inputArray`:

```
"contains.$": "States.ArrayContains($.inputArray, $.lookingFor)"
```

Because the value stored in `lookingFor` is included in the `inputArray`, `States.ArrayContains` returns the following result:

```
{"contains": true }
```


States.ArrayRange

Use the `States.ArrayRange` intrinsic function to create a new array containing a specific range of elements. The new array can contain up to 1000 elements.

This function takes three arguments. The first argument is the first element of the new array, the second argument is the final element of the new array, and the third argument is the increment value between the elements in the new array.

Input validation

- You must specify integer values for all of the arguments.

If you specify a non-integer value for any of the arguments, Step Functions will round it off to the nearest integer.

- You must specify a non-zero value for the third argument.
- The newly generated array can't contain more than 1000 items.

For example, the following use of the `States.ArrayRange` function will create an array with a first value of 1, a final value of 9, and values in between the first and final values increase by two for each item:

```
"array.$": "States.ArrayRange(1, 9, 2)"
```

Which would return the following array:

```
{"array": [1,3,5,7,9] }
```

States.ArrayGetItem

This intrinsic function returns a specified index's value. This function takes two arguments. The first argument is an array of values and the second argument is the array index of the value to return.

For example, use the following `inputArray` and `index` values:

```
{
  "inputArray": [1,2,3,4,5,6,7,8,9],
  "index": 5
}
```

```
}
```

From these values, you can use the `States.ArrayGetItem` function to return the value in the index position 5 within the array:

```
"item.$": "States.ArrayGetItem($.inputArray, $.index)"
```

In this example, `States.ArrayGetItem` would return the following result:

```
{ "item": 6 }
```

States.ArrayLength

The `States.ArrayLength` intrinsic function returns the length of an array. It has one argument, the array to return the length of.

For example, given the following input array:

```
{  
  "inputArray": [1,2,3,4,5,6,7,8,9]  
}
```

You can use `States.ArrayLength` to return the length of `inputArray`:

```
"length.$": "States.ArrayLength($.inputArray)"
```

In this example, `States.ArrayLength` would return the following JSON object that represents the array length:

```
{ "length": 9 }
```

States.ArrayUnique

The `States.ArrayUnique` intrinsic function removes duplicate values from an array and returns an array containing only unique elements. This function takes an array, which can be unsorted, as its sole argument.

For example, the following `inputArray` contains a series of duplicate values:

```
{"inputArray": [1,2,3,3,3,3,3,3,4] }
```

You could use the `States.ArrayUnique` function as and specify the array you want to remove duplicate values from:

```
"array.$": "States.ArrayUnique($.inputArray)"
```

The `States.ArrayUnique` function would return the following array containing only unique elements, removing all duplicate values:

```
{"array": [1,2,3,4] }
```

Intrinsics for data encoding and decoding

Use the following intrinsic functions to encode or decode data based on the Base64 encoding scheme.

`States.Base64Encode`

Use the `States.Base64Encode` intrinsic function to encode data based on MIME Base64 encoding scheme. You can use this function to pass data to other AWS services without using an AWS Lambda function.

This function takes a data string of up to 10,000 characters to encode as its only argument.

For example, consider the following input string:

```
{"input": "Data to encode" }
```

You can use the `States.Base64Encode` function to encode the input string as a MIME Base64 string:

```
"base64.$": "States.Base64Encode($.input)"
```

The `States.Base64Encode` function returns the following encoded data in response:

```
{"base64": "RGF0YSB0byB1bmNvZGU=" }
```

States.Base64Decode

Use the `States.Base64Decode` intrinsic function to decode data based on MIME Base64 decoding scheme. You can use this function to pass data to other AWS services without using a Lambda function.

This function takes a Base64 encoded data string of up to 10,000 characters to decode as its only argument.

For example, given the following input:

```
{"base64": "RGF0YSB0byB1bmNvZGU=" }
```

You can use the `States.Base64Decode` function to decode the base64 string to a human-readable string:

```
"data.$": "States.Base64Decode($.base64)"
```

The `States.Base64Decode` function would return the following decoded data in response:

```
{"data": "Decoded data" }
```

Intrinsic for hash calculation

States.Hash

Use the `States.Hash` intrinsic function to calculate the hash value of a given input. You can use this function to pass data to other AWS services without using a Lambda function.

This function takes two arguments. The first argument is the data you want to calculate the hash value of. The second argument is the hashing algorithm to use to perform the hash calculation. The data you provide must be an object string containing 10,000 characters or less.

The hashing algorithm you specify can be any of the following algorithms:

- MD5
- SHA-1
- SHA-256

- SHA-384
- SHA-512

For example, you can use this function to calculate the hash value of the `Data` string using the specified `Algorithm`:

```
{
  "Data": "input data",
  "Algorithm": "SHA-1"
}
```

You can use the `States.Hash` function to calculate the hash value:

```
"output.$": "States.Hash($.Data, $.Algorithm)"
```

The `States.Hash` function returns the following hash value in response:

```
{"output": "aaff4a450a104cd177d28d18d7485e8cae074b7" }
```

Intrinsics for JSON data manipulation

Use these functions to perform basic data processing operations on JSON objects.

`States.JsonMerge`

Use the `States.JsonMerge` intrinsic function to merge two JSON objects into a single object. This function takes three arguments. The first two arguments are the JSON objects that you want to merge. The third argument is a boolean value of `false`. This boolean value determines if the deep merging mode is enabled.

Currently, Step Functions only supports the shallow merging mode; therefore, you must specify the boolean value as `false`. In the shallow mode, if the same key exists in both JSON objects, the latter object's key overrides the same key in the first object. Additionally, objects nested within a JSON object aren't merged when you use shallow merging.

For example, you can use the `States.JsonMerge` function to merge the following JSON objects that share the key `a`.

```
{
```

```
"json1": { "a": {"a1": 1, "a2": 2}, "b": 2 },
"json2": { "a": {"a3": 1, "a4": 2}, "c": 3 }
}
```

You can specify the `json1` and `json2` objects as inputs in the `States.JsonMerge` function to merge them together:

```
"output.$": "States.JsonMerge($.json1, $.json2, false)"
```

The `States.JsonMerge` returns the following merged JSON object as result. In the merged JSON object output, the `json2` object's key `a` replaces the `json1` object's key `a`. Also, the nested object in `json1` object's key `a` is discarded because shallow mode doesn't support merging nested objects.

```
{
  "output": {
    "a": {"a3": 1, "a4": 2},
    "b": 2,
    "c": 3
  }
}
```

States.StringToJson

The `States.StringToJson` function takes a reference path to an escaped JSON string as its only argument.

The interpreter applies a JSON parser and returns the input's parsed JSON form. For example, you can use this function to escape the following input string:

```
{
  "escapedJsonString": "{\"foo\": \"bar\"}"
}
```

Use the `States.StringToJson` function and specify the `escapedJsonString` as the input argument:

```
States.StringToJson($.escapedJsonString)
```

The `States.StringToJson` function returns the following result:

```
{ "foo": "bar" }
```

States.JsonToString

The `States.JsonToString` function takes only one argument, which is the `Path` that contains the JSON data to return as an unescaped string. The interpreter returns a string that contains JSON text representing the data specified by the `Path`. For example, you can provide the following JSON `Path` containing an escaped value:

```
{
  "unescapedJson": {
    "foo": "bar"
  }
}
```

Provide the `States.JsonToString` function with the data contained within `unescapedJson`:

```
States.JsonToString($.unescapedJson)
```

The `States.JsonToString` function returns the following response:

```
{\"foo\": \"bar\"}
```

Intrinsics for Math operations

Use these functions to perform Math operations.

States.MathRandom

Use the `States.MathRandom` intrinsic function to return a random number between the specified start number (inclusive) and end number (exclusive).

You can use this function to distribute a specific task between two or more resources.

This function takes three arguments. The first argument is the start number, the second argument is the end number, and the last argument controls the seed value. The seed value

argument is optional. If you use this function with the same seed value, it returns an identical number.

Important

Because the `States.MathRandom` function doesn't return cryptographically secure random numbers, we recommend that you don't use it for security sensitive applications.

Input validation

- You must specify integer values for the start number and end number arguments.

If you specify a non-integer value for the start number or end number argument, Step Functions will round it off to the nearest integer.

For example, to generate a random number between one and 999, you can use the following input values:

```
{
  "start": 1,
  "end": 999
}
```

To generate the random number, provide the `start` and `end` values to the `States.MathRandom` function:

```
"random.$": "States.MathRandom($.start, $.end)"
```

The `States.MathRandom` function returns the following random number as a response:

```
{"random": 456 }
```

States.MathAdd

Use the `States.MathAdd` intrinsic function to return the sum of two numbers. For example, you can use this function to increment values inside a loop without invoking a Lambda function.

Input validation

- You must specify integer values for all the arguments.

If you specify a non-integer value for one or both the arguments, Step Functions will round it off to the nearest integer.

- You must specify integer values in the range of -2147483648 and 2147483647.

For example, you can use the following values to subtract one from 111:

```
{
  "value1": 111,
  "step": -1
}
```

Then, use the `States.MathAdd` function defining `value1` as the starting value, and `step` as the value to increment `value1` by:

```
"value1.$": "States.MathAdd($.value1, $.step)"
```

The `States.MathAdd` function would return the following number in response:

```
{"value1": 110 }
```

Intrinsic for String operation

`States.StringSplit`

Use the `States.StringSplit` intrinsic function to split a string into an array of values. This function takes two arguments. The first argument is a string and the second argument is the delimiting character that the function will use to divide the string.

Example - Split an input string using a single delimiting character

For this example, use `States.StringSplit` to divide the following `inputString`, which contains a series of comma separated values:

```
{
```

```
"inputString": "1,2,3,4,5",  
"splitter": ",",  
}
```

Use the `States.StringSplit` function and define `inputString` as the first argument, and the delimiting character `splitter` as the second argument:

```
"array.$": "States.StringSplit($.inputString, $.splitter)"
```

The `States.StringSplit` function returns the following string array as result:

```
{"array": ["1","2","3","4","5"] }
```

Example - Split an input string using multiple delimiting characters

For this example, use `States.StringSplit` to divide the following `inputString`, which contains multiple delimiting characters:

```
{  
  "inputString": "This.is+a,test=string",  
  "splitter": ".+,=",  
}
```

Use the `States.StringSplit` function as follows:

```
{  
  "myStringArray.$": "States.StringSplit($.inputString, $.splitter)"  
}
```

The `States.StringSplit` function returns the following string array as result:

```
{"myStringArray": [  
  "This",  
  "is",  
  "a",  
  "test",  
  "string"  
]}
```

Intrinsic for unique identifier generation

States.UUID

Use the `States.UUID` intrinsic function to return a version 4 universally unique identifier (v4 UUID) generated using random numbers. For example, you can use this function to call other AWS services or resources that need a UUID parameter or insert items in a DynamoDB table.

The `States.UUID` function is called with no arguments specified:

```
"uuid.$": "States.UUID()"
```

The function returns a randomly generated UUID, as in the following example:

```
{"uuid": "ca4c1140-dcc1-40cd-ad05-7b4aa23df4a8" }
```

Intrinsic for generic operation

States.Format

Use the `States.Format` intrinsic function to construct a string from both literal and interpolated values. This function takes one or more arguments. The value of the first argument must be a string, and may include zero or more instances of the character sequence `{}`. There must be as many remaining arguments in the intrinsic's invocation as there are occurrences of `{}`. The interpreter returns the string defined in the first argument with each `{}` replaced by the value of the positionally-corresponding argument in the Intrinsic invocation.

For example, you can use the following inputs of an individual's name, and a template sentence to have their name inserted into:

```
{
  "name": "Arnav",
  "template": "Hello, my name is {}."
}
```

Use the `States.Format` function and specify the template string and the string to insert in place of the `{}` characters:

```
States.Format('Hello, my name is {}.', $.name)
```

or

```
States.Format($.template, $.name)
```

With either of the previous inputs, the `States.Format` function returns the completed string in response:

```
Hello, my name is Arnav.
```

Reserved characters in intrinsic functions

The following characters are reserved for intrinsic functions, and must be escaped with a backslash (`\`) if you want them to appear in the `Value`: `{}`, and `\`.

If the character `\` needs to appear as part of the value without serving as an escape character, you must escape it with a backslash. The following escaped character sequences are used with intrinsic functions:

- The literal string `\'` represents `'`.
- The literal string `\{` represents `{`.
- The literal string `\}` represents `}`.
- The literal string `\\` represents `\`.

In JSON, backslashes contained in a string literal value must be escaped with another backslash. The equivalent list for JSON is:

- The escaped string `\\'` represents `\'`.
- The escaped string `\\{` represents `\{`.
- The escaped string `\\}` represents `\}`.
- The escaped string `\\\\` represents `\\`.

Note

If an open escape backslash `\` is found in the intrinsic invocation string, the interpreter will return a runtime error.

Common State Fields

Type (Required)

The state's type.

Next

The name of the next state that is run when the current state finishes. Some state types, such as `Choice`, allow multiple transition states.

If the current state is the last state in your workflow, or a terminal state, such as [Succeed](#) or [Fail](#), you don't need to specify the `Next` field.

End

Designates this state as a terminal state (ends the execution) if set to `true`. There can be any number of terminal states per state machine. Only one of `Next` or `End` can be used in a state. Some state types, such as `Choice`, or terminal states, such as [Succeed](#) and [Fail](#), don't support or use the `End` field.

Comment (Optional)

Holds a human-readable description of the state.

InputPath (Optional)

A [path](#) that selects a portion of the state's input to be passed to the state's task for processing. If omitted, it has the value `$` which designates the entire input. For more information, see [Input and Output Processing](#).

OutputPath (Optional)

A [path](#) that selects a portion of the state's output to be passed to the next state. If omitted, it has the value `$` which designates the entire output. For more information, see [Input and Output Processing](#).

Pass

A `Pass` state (`"Type": "Pass"`) passes its input to its output, without performing work. `Pass` states are useful when constructing and debugging state machines.

You can also use a Pass state to transform JSON state input using filters, and then pass the transformed data to the next state in your workflows. For information about input transformation, see [InputPath, Parameters and ResultSelector](#).

In addition to the [common state fields](#), Pass states allow the following fields.

Result (Optional)

Refers to the output of a virtual task that is passed on to the next state. If you include the `ResultPath` field in your state machine definition, `Result` is placed as specified by `ResultPath` and passed on to the next state.

ResultPath (Optional)

Specifies where to place the *output* (relative to the input) of the virtual task specified in `Result`. The input is further filtered as specified by the `OutputPath` field (if present) before being used as the state's output. For more information, see [Input and Output Processing](#).

Parameters (Optional)

Creates a collection of key-value pairs that will be passed as input. You can specify `Parameters` as a static value or select from the input using a path. For more information, see [InputPath, Parameters and ResultSelector](#).

Pass State Example

Here is an example of a Pass state that injects some fixed data into the state machine, probably for testing purposes.

```
"No-op": {
  "Type": "Pass",
  "Result": {
    "x-datum": 0.381018,
    "y-datum": 622.2269926397355
  },
  "ResultPath": "$.coords",
  "End": true
}
```

Suppose the input to this state is the following.

```
{
```

```
"georefOf": "Home"
}
```

Then the output would be this.

```
{
  "georefOf": "Home",
  "coords": {
    "x-datum": 0.381018,
    "y-datum": 622.2269926397355
  }
}
```

Task state

A Task state ("Type": "Task") represents a single unit of work performed by a state machine. A task performs work by using an activity or an AWS Lambda function, by integrating with other [supported AWS services](#), or by invoking a third-party API, such as Stripe.

The [Amazon States Language](#) represents tasks by setting a state's type to Task and by providing the task with the Amazon Resource Name (ARN) of the activity, Lambda function, or the third-party API endpoint. The following Task state definition invokes a Lambda function named *HelloFunction*.

```
"Lambda Invoke": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "Parameters": {
    "Payload.$": "$",
    "FunctionName": "arn:aws:lambda:us-east-2:123456789012:function:HelloFunction:
$LATEST"
  },
  "End": true
}
```

In this topic

- [Task types](#)
- [Task state fields](#)
- [Task state definition examples](#)

- [Activities](#)

Task types

Step Functions supports the following task types that you can specify in a Task state definition:

- [Activity](#)
- [Lambda functions](#)
- [A supported AWS service](#)
- [An HTTP Task](#)

You specify a task type by providing its ARN in the `Resource` field of a Task state definition. The following example shows the syntax of the `Resource` field. All Task types except the one that invokes an third-party API, use the following syntax. For information about syntax of the HTTP Task, see [Call third-party APIs](#).

In your Task state definition, replace the italicized text in the following syntax with the AWS resource-specific information.

```
arn:partition:service:region:account:task_type:name
```

The following list explains the individual components in this syntax:

- `partition` is the AWS Step Functions partition to use, most commonly `aws`.
- `service` indicates the AWS service used to execute the task, and can be one of the following values:
 - `states` for an [activity](#).
 - `lambda` for a [Lambda function](#). If you integrate with other AWS services, for example, Amazon SNS or Amazon DynamoDB, use `sns` or `dynamodb`.
- `region` is the [AWS Region code](#) in which the Step Functions activity or state machine type, Lambda function, or any other AWS resource has been created.
- `account` is the AWS account ID in which you've defined the resource.
- `task_type` is the type of task to run. It can be one of the following values:
 - `activity` – An [activity](#).
 - `function` – A [Lambda function](#).

- *servicename* – The name of a supported connected service (see [Optimized integrations for Step Functions](#)).
- name is the registered resource name (activity name, Lambda function name, or service API action).

Note

Step Functions doesn't support referencing ARNs across partitions or regions. For example, `aws-cn` can't invoke tasks in the `aws` partition, and the other way around.

The following sections provide more detail about each task type.

Activity

Activities represent workers (processes or threads), implemented and hosted by you, that perform a specific task. They are supported only by Standard Workflows, not Express Workflows.

Activity Resource ARNs use the following syntax.

```
arn:partition:states:region:account:activity:name
```

Note

You must create activities with Step Functions (using a [CreateActivity](#), API action, or the [Step Functions console](#)) before their first use.

For more information about creating an activity and implementing workers, see [Activities](#).

Lambda functions

Lambda tasks execute a function using AWS Lambda. To specify a Lambda function, use the ARN of the Lambda function in the Resource field.

Depending on the type of integration ([Optimized integration](#) or [AWS SDK integration](#)) you use for specifying a Lambda function, the syntax of your Lambda function's Resource field varies.

The following Resource field syntax is an example of an optimized integration with a Lambda function.

```
"arn:aws:states:::lambda:invoke"
```

The following Resource field syntax is an example of an AWS SDK integration with a Lambda function.

```
"arn:aws:states:::aws-sdk:lambda:invoke"
```

The following Task state definition shows an example of an optimized integration with a Lambda function named *HelloWorld*.

```
"LambdaState": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "OutputPath": "$.Payload",
  "Parameters": {
    "Payload.$": "$",
    "FunctionName": "arn:aws:lambda:us-east-1:function:HelloWorld:$LATEST"
  },
  "Next": "NextState"
}
```

After the Lambda function specified in the Resource field completes, its output is sent to the state identified in the Next field ("NextState").

A supported AWS service

When you reference a connected resource, Step Functions directly calls the API actions of a supported service. Specify the service and action in the Resource field.

Connected service Resource ARNs use the following syntax.

```
arn:partition:states:region:account:servicename:APIname
```

Note

To create a synchronous connection to a connected resource, append `.sync` to the *APIname* entry in the ARN. For more information, see [Working with other services](#).

For example:

```
{
  "StartAt": "BATCH_JOB",
  "States": {
    "BATCH_JOB": {
      "Type": "Task",
      "Resource": "arn:aws:states:::batch:submitJob.sync",
      "Parameters": {
        "JobDefinition": "preprocessing",
        "JobName": "PreprocessingBatchJob",
        "JobQueue": "SecondaryQueue",
        "Parameters.$": "$.batchjob.parameters",
        "RetryStrategy": {
          "attempts": 5
        }
      },
      "End": true
    }
  }
}
```

Task state fields

In addition to the [common state fields](#), Task states have the following fields.

Resource (Required)

A URI, especially an ARN that uniquely identifies the specific task to execute.

Parameters (Optional)

Used to pass information to the API actions of connected resources. The parameters can use a mix of static JSON and [JsonPath](#). For more information, see [Pass parameters to a service API](#).

Credentials (Optional)

Specifies a target role the state machine's execution role must assume before invoking the specified Resource. Alternatively, you can also specify a JSONPath value or an [intrinsic function](#) that resolves to an IAM role ARN at runtime based on the execution input. If you specify a JSONPath value, you must prefix it with the \$. notation.

For examples of using this field in the Task state, see [Task state's Credentials field examples](#). For an example of using this field to access a cross-account AWS resource from your state machine, see [Tutorial: Accessing cross-account AWS resources](#).

Note

This field is supported by the [Task types](#) that use [Lambda functions](#) and [a supported AWS service](#).

ResultPath (Optional)

Specifies where (in the input) to place the results of executing the task that's specified in Resource. The input is then filtered as specified by the OutputPath field (if present) before being used as the state's output. For more information, see [Input and Output Processing](#).

ResultSelector (Optional)

Pass a collection of key value pairs, where the values are static or selected from the result. For more information, see [ResultSelector](#).

Retry (Optional)

An array of objects, called Retriers, that define a retry policy if the state encounters runtime errors. For more information, see [State machine examples using Retry and using Catch](#).

Catch (Optional)

An array of objects, called Catchers, that define a fallback state. This state is executed if the state encounters runtime errors and its retry policy is exhausted or isn't defined. For more information, see [Fallback States](#).

TimeoutSeconds (Optional)

Specifies the maximum time an activity or a task can run before it times out with the [States.Timeout](#) error and fails. The timeout value must be positive, non-zero integer. The default value is 99999999.

The timeout count begins after a task starts, for example, when ActivityStarted or LambdaFunctionStarted events are logged in the execution event history. For Activities, the count begins when GetActivityTask receives a token and ActivityStarted is logged in the execution event history.

When a task starts, Step Functions waits for a success or failure response from the task or activity worker within the specified `TimeoutSeconds` duration. If the task or activity worker fails to respond within this time, Step Functions marks the workflow execution as failed.

TimeoutSecondsPath (Optional)

If you want to provide a timeout value dynamically from the state input using a reference path, use `TimeoutSecondsPath`. When resolved, the reference path must select fields whose values are positive integers.

Note

A Task state cannot include both `TimeoutSeconds` and `TimeoutSecondsPath`.

HeartbeatSeconds (Optional)

Determines the frequency of heartbeat signals an activity worker sends during the execution of a task. Heartbeats indicate that a task is still running and it needs more time to complete. Heartbeats prevent an activity or task from timing out within the `TimeoutSeconds` duration.

`HeartbeatSeconds` must be a positive, non-zero integer value less than the `TimeoutSeconds` field value. The default value is 99999999. If more time than the specified seconds elapses between heartbeats from the task, the Task state fails with a [States.Timeout](#) error.

For Activities, the count begins when `GetActivityTask` receives a token and `ActivityStarted` is logged in the execution event history.

HeartbeatSecondsPath (Optional)

If you want to provide a heartbeat value dynamically from the state input using a reference path, use `HeartbeatSecondsPath`. When resolved, the reference path must select fields whose values are positive integers.

Note

A Task state cannot include both `HeartbeatSeconds` and `HeartbeatSecondsPath`.

A Task state must set either the `End` field to `true` if the state ends the execution, or must provide a state in the `Next` field that is run when the Task state is complete.

Task state definition examples

The following examples show how you can specify the Task state definition based on your requirement.

- [Specifying Task state timeouts and heartbeat intervals](#)
 - [Static timeout and heartbeat notification example](#)
 - [Dynamic task timeout and heartbeat notification example](#)
- [Using Credentials field](#)
 - [Specifying hard-coded IAM role ARN](#)
 - [Specifying JSONPath as IAM role ARN](#)
 - [Specifying an intrinsic function as IAM role ARN](#)

Task state timeouts and heartbeat intervals

It's a good practice to set a timeout value and a heartbeat interval for long-running activities. This can be done by specifying the timeout and heartbeat values, or by setting them dynamically.

Static timeout and heartbeat notification example

When `HelloWorld` completes, the next state (here called `NextState`) will be run.

If this task fails to complete within 300 seconds, or doesn't send heartbeat notifications in intervals of 60 seconds, the task is marked as `failed`.

```
"ActivityState": {
  "Type": "Task",
  "Resource": "arn:aws:states:us-east-1:123456789012:activity:HelloWorld",
  "TimeoutSeconds": 300,
  "HeartbeatSeconds": 60,
  "Next": "NextState"
}
```

Dynamic task timeout and heartbeat notification example

In this example, when the AWS Glue job completes, the next state will be run.

If this task fails to complete within the interval set dynamically by the AWS Glue job, the task is marked as failed.

```
"GlueJobTask": {
  "Type": "Task",
  "Resource": "arn:aws:states:::glue:startJobRun.sync",
  "Parameters": {
    "JobName": "myGlueJob"
  },
  "TimeoutSecondsPath": "$.params.maxTime",

  "Next": "NextState"
}
```

Task state's Credentials field examples

Specifying hard-coded IAM role ARN

The following example specifies a target IAM role that a state machine's execution role must assume to access a cross-account Lambda function named Echo. In this example, the target role ARN is specified as a hard-coded value.

```
{
  "StartAt": "Cross-account call",
  "States": {
    "Cross-account call": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Credentials": {
        "RoleArn": "arn:aws:iam::111122223333:role/LambdaRole"
      },
      "Parameters": {
        "FunctionName": "arn:aws:lambda:us-east-2:111122223333:function:Echo"
      },
      "End": true
    }
  }
}
```

Specifying JSONPath as IAM role ARN

The following example specifies a JSONPath value, which will resolve to an IAM role ARN at runtime.

```
{
  "StartAt": "Lambda",
  "States": {
    "Lambda": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Credentials": {
        "RoleArn.$": "$.roleArn"
      },
      ...
    }
  }
}
```

Specifying an intrinsic function as IAM role ARN

The following example uses the [States.Format](#) intrinsic function, which resolves to an IAM role ARN at runtime.

```
{
  "StartAt": "Lambda",
  "States": {
    "Lambda": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Credentials": {
        "RoleArn.$": "States.Format('arn:aws:iam::{}:role/ROLENAME', $.accountId)"
      },
      ...
    }
  }
}
```

Activities

Activities are an AWS Step Functions feature that enables you to have a task in your state machine where the work is performed by a *worker* that can be hosted on Amazon Elastic Compute Cloud

(Amazon EC2), Amazon Elastic Container Service (Amazon ECS), mobile devices—basically anywhere.

Overview

In AWS Step Functions, activities are a way to associate code running somewhere (known as an *activity worker*) with a specific task in a state machine. You can create an activity using the Step Functions console, or by calling [CreateActivity](#). This provides an Amazon Resource Name (ARN) for your task state. Use this ARN to poll the task state for work in your activity worker.

Note

Activities are not versioned and are expected to be backward compatible. If you must make a backward-incompatible change to an activity, create a *new* activity in Step Functions using a unique name.

An activity worker can be an application running on an Amazon EC2 instance, an AWS Lambda function, a mobile device: any application that can make an HTTP connection, hosted anywhere. When Step Functions reaches an activity task state, the workflow waits for an activity worker to poll for a task. An activity worker polls Step Functions by using [GetActivityTask](#), and sending the ARN for the related activity. [GetActivityTask](#) returns a response including input (a string of JSON input for the task) and a [taskToken](#) (a unique identifier for the task). After the activity worker completes its work, it can provide a report of its success or failure by using [SendTaskSuccess](#) or [SendTaskFailure](#). These two calls use the [taskToken](#) provided by [GetActivityTask](#) to associate the result with that task.

APIs Related to Activity Tasks

Step Functions provides APIs for creating and listing activities, requesting a task, and for managing the flow of your state machine based on the results of your worker.

The following are the Step Functions APIs that are related to activities:

- [CreateActivity](#)
- [GetActivityTask](#)
- [ListActivities](#)
- [SendTaskFailure](#)

- [SendTaskHeartbeat](#)
- [SendTaskSuccess](#)

Note

Polling for activity tasks with `GetActivityTask` can cause latency in some implementations. See [Avoid latency when polling for activity tasks](#).

Waiting for an Activity Task to Complete

Configure how long a state waits by setting `TimeoutSeconds` in the task definition. To keep the task active and waiting, periodically send a heartbeat from your activity worker using [SendTaskHeartbeat](#) within the time configured in `TimeoutSeconds`. By configuring a long timeout duration and actively sending a heartbeat, an activity in Step Functions can wait up to a year for an execution to complete.

For example, if you need a workflow that waits for the outcome of a long process, do the following:

1. Create an activity by using the console, or by using [CreateActivity](#). Make a note of the activity ARN.
2. Reference that ARN in an activity task state in your state machine definition and set `TimeoutSeconds`.
3. Implement an activity worker that polls for work by using [GetActivityTask](#), referencing that activity ARN.
4. Use [SendTaskHeartbeat](#) periodically within the time you set in [HeartbeatSeconds](#) in your state machine task definition to keep the task from timing out.
5. Start an execution of your state machine.
6. Start your activity worker process.

The execution pauses at the activity task state and waits for your activity worker to poll for a task. Once a `taskToken` is provided to your activity worker, your workflow will wait for [SendTaskSuccess](#) or [SendTaskFailure](#) to provide a status. If the execution doesn't receive either of these or a [SendTaskHeartbeat](#) call before the time configured in `TimeoutSeconds`, the execution will fail and the execution history will contain an `ExecutionTimedOut` event.

Next Steps

For a more detailed look at creating state machines that use an activity workers, see:

- [Creating an Activity state machine using Step Functions](#)
- [Example Activity Worker in Ruby](#)

Example Activity Worker in Ruby

The following is an example activity worker that uses the AWS SDK for Ruby to show you how to use best practices and implement your own activity worker.

The code implements a consumer-producer pattern with a configurable number of threads for pollers and activity workers. The poller threads are constantly long polling the activity task. Once an activity task is retrieved, it's passed through a bounded blocking queue for the activity thread to pick it up.

- For more information about the AWS SDK for Ruby, see the [AWS SDK for Ruby API Reference](#).
- To download this code and related resources, see the [step-functions-ruby-activity-worker](#) repository on GitHub.

The following Ruby code is the main entry point for this example Ruby activity worker.

```
require_relative '../lib/step_functions/activity'
credentials = Aws::SharedCredentials.new
region = 'us-west-2'
activity_arn = 'ACTIVITY_ARN'

activity = StepFunctions::Activity.new(
  credentials: credentials,
  region: region,
  activity_arn: activity_arn,
  workers_count: 1,
  pollers_count: 1,
  heartbeat_delay: 30
)

# The start method takes as argument the block that is the actual logic of your custom
activity.start do |input|
```

```
{ result: :SUCCESS, echo: input['value'] }
end
```

The code includes defaults you can change to reference your activity, and to adapt it to your specific implementation. This code takes as input the actual implementation logic, allows you to reference your specific activity and credentials, and enables you to configure the number of threads and heartbeat delay. For more information and to download the code, see [Step Functions Ruby Activity Worker](#).

Item	Description
<code>require_relative</code>	Relative path to the following example activity worker code.
<code>region</code>	AWS Region of your activity.
<code>workers_count</code>	The number of threads for your activity worker. For most implementations, between 10 and 20 threads should be sufficient. The longer the activity takes to process, the more threads it might need. As an estimate, multiply the number of process activities per second by the 99th percentile activity processing latency, in seconds.
<code>pollers_count</code>	The number of threads for your pollers. Between 10 and 20 threads should be sufficient for most implementations.
<code>heartbeat_delay</code>	The delay in seconds between heartbeats.
<code>input</code>	Implementation logic of your activity.

The following is the Ruby activity worker, referenced with `../lib/step_functions/activity` in your code.

```
require 'set'
```

```
require 'json'
require 'thread'
require 'logger'
require 'aws-sdk'

module Validate
  def self.positive(value)
    raise ArgumentError, 'Argument has to be positive' if value <= 0
    value
  end

  def self.required(value)
    raise ArgumentError, 'Argument is required' if value.nil?
    value
  end
end

module StepFunctions
  class RetryError < StandardError
    def initialize(message)
      super(message)
    end
  end
end

def self.with_retries(options = {}, &block)
  retries = 0
  base_delay_seconds = options[:base_delay_seconds] || 2
  max_retries = options[:max_retries] || 3
  begin
    block.call
  rescue => e
    puts e
    if retries < max_retries
      retries += 1
      sleep base_delay_seconds**retries
      retry
    end
    raise RetryError, 'All retries of operation had failed'
  end
end

class Activity
  def initialize(options = {})
    @states = Aws::States::Client.new(
```

```
    credentials: Validate.required(options[:credentials]),
    region: Validate.required(options[:region]),
    http_read_timeout: Validate.positive(options[:http_read_timeout] || 60)
  )
  @activity_arn = Validate.required(options[:activity_arn])
  @heartbeat_delay = Validate.positive(options[:heartbeat_delay] || 60)
  @queue_max = Validate.positive(options[:queue_max] || 5)
  @pollers_count = Validate.positive(options[:pollers_count] || 1)
  @workers_count = Validate.positive(options[:workers_count] || 1)
  @max_retry = Validate.positive(options[:workers_count] || 3)
  @logger = Logger.new(STDOUT)
end

def start(&block)
  @sink = SizedQueue.new(@queue_max)
  @activities = Set.new
  start_heartbeat_worker(@activities)
  start_workers(@activities, block, @sink)
  start_pollers(@activities, @sink)
  wait
end

def queue_size
  return 0 if @sink.nil?
  @sink.size
end

def activities_count
  return 0 if @activities.nil?
  @activities.size
end

private

def start_pollers(activities, sink)
  @pollers = Array.new(@pollers_count) do
    PollerWorker.new(
      states: @states,
      activity_arn: @activity_arn,
      sink: sink,
      activities: activities,
      max_retry: @max_retry
    )
  end
end
```

```
    @pollers.each(&:start)
  end

  def start_workers(activities, block, sink)
    @workers = Array.new(@workers_count) do
      ActivityWorker.new(
        states: @states,
        block: block,
        sink: sink,
        activities: activities,
        max_retry: @max_retry
      )
    end
    @workers.each(&:start)
  end

  def start_heartbeat_worker(activities)
    @heartbeat_worker = HeartbeatWorker.new(
      states: @states,
      activities: activities,
      heartbeat_delay: @heartbeat_delay,
      max_retry: @max_retry
    )
    @heartbeat_worker.start
  end

  def wait
    sleep
  rescue Interrupt
    shutdown
  ensure
    Thread.current.exit
  end

  def shutdown
    stop_workers(@pollers)
    wait_workers(@pollers)
    wait_activities_drained
    stop_workers(@workers)
    wait_activities_completed
    shutdown_workers(@workers)
    shutdown_worker(@heartbeat_worker)
  end
end
```

```
def shutdown_workers(workers)
  workers.each do |worker|
    shutdown_worker(worker)
  end
end

def shutdown_worker(worker)
  worker.kill
end

def wait_workers(workers)
  workers.each(&:wait)
end

def wait_activities_drained
  wait_condition { @sink.empty? }
end

def wait_activities_completed
  wait_condition { @activities.empty? }
end

def wait_condition(&block)
  loop do
    break if block.call
    sleep(1)
  end
end

def stop_workers(workers)
  workers.each(&:stop)
end

class Worker
  def initialize
    @logger = Logger.new(STDOUT)
    @running = false
  end

  def run
    raise 'Method run hasn\'t been implemented'
  end

  def process
```



```
    loop do
      begin
        break unless @running
        run
      rescue => e
        puts e
        @logger.error('Unexpected error has occurred')
        @logger.error(e)
      end
    end
  end
end

def start
  return unless @thread.nil?
  @running = true
  @thread = Thread.new do
    process
  end
end

def stop
  @running = false
end

def kill
  return if @thread.nil?
  @thread.kill
  @thread = nil
end

def wait
  @thread.join
end

class PollerWorker < Worker
  def initialize(options = {})
    @states = options[:states]
    @activity_arn = options[:activity_arn]
    @sink = options[:sink]
    @activities = options[:activities]
    @max_retry = options[:max_retry]
    @logger = Logger.new(STDOUT)
  end
end
```

```
def run
  activity_task = StepFunctions.with_retries(max_retry: @max_retry) do
    begin
      @states.get_activity_task(activity_arn: @activity_arn)
    rescue => e
      @logger.error('Failed to retrieve activity task')
      @logger.error(e)
    end
  end
  return if activity_task.nil? || activity_task.task_token.nil?
  @activities.add(activity_task.task_token)
  @sink.push(activity_task)
end

class ActivityWorker < Worker
  def initialize(options = {})
    @states = options[:states]
    @block = options[:block]
    @sink = options[:sink]
    @activities = options[:activities]
    @max_retry = options[:max_retry]
    @logger = Logger.new(STDOUT)
  end

  def run
    activity_task = @sink.pop
    result = @block.call(JSON.parse(activity_task.input))
    send_task_success(activity_task, result)
  rescue => e
    send_task_failure(activity_task, e)
  ensure
    @activities.delete(activity_task.task_token) unless activity_task.nil?
  end

  def send_task_success(activity_task, result)
    StepFunctions.with_retries(max_retry: @max_retry) do
      begin
        @states.send_task_success(
          task_token: activity_task.task_token,
          output: JSON.dump(result)
        )
      rescue => e
    end
  end
end
```

```
        @logger.error('Failed to send task success')
        @logger.error(e)
      end
    end
  end

  def send_task_failure(activity_task, error)
    StepFunctions.with_retries do
      begin
        @states.send_task_failure(
          task_token: activity_task.task_token,
          cause: error.message
        )
      rescue => e
        @logger.error('Failed to send task failure')
        @logger.error(e)
      end
    end
  end
end

class HeartbeatWorker < Worker
  def initialize(options = {})
    @states = options[:states]
    @activities = options[:activities]
    @heartbeat_delay = options[:heartbeat_delay]
    @max_retry = options[:max_retry]
    @logger = Logger.new(STDOUT)
  end

  def run
    sleep(@heartbeat_delay)
    @activities.each do |token|
      send_heartbeat(token)
    end
  end

  def send_heartbeat(token)
    StepFunctions.with_retries(max_retry: @max_retry) do
      begin
        @states.send_task_heartbeat(token)
      rescue => e
        @logger.error('Failed to send heartbeat for activity')
        @logger.error(e)
      end
    end
  end
end
```

```
        end
      end
    rescue => e
      @logger.error('Failed to send heartbeat for activity')
      @logger.error(e)
    end
  end
end
end
end
```

Choice

A Choice state ("Type": "Choice") adds conditional logic to a state machine.

In addition to most of the [common state fields](#), Choice states contains the following additional fields.

Choices (Required)

An array of [Choice Rules](#) that determines which state the state machine transitions to next. You use a comparison operator in a Choice Rule to compare an input variable with a specific value. For example, using Choice Rules you can compare if an input variable is greater than or less than 100.

When a Choice state is run, it evaluates each Choice Rule to true or false. Based on the result of this evaluation, Step Functions transitions to the next state in the workflow.

You must define at least one rule in the Choice state.

Default (Optional, Recommended)

The name of the state to transition to if none of the transitions in Choices is taken.

Important

Choice states don't support the End field. In addition, they use Next only inside their Choices field.

Tip

To deploy an example of a workflow that uses a Choice state to your AWS account, see [Module 5 - Choice State and Map State](#) of *The AWS Step Functions Workshop*.

Choice Rules

A Choice state must have a `Choices` field whose value is a non-empty array. Each element in this array is an object called Choice Rule, which contains the following:

- A **comparison** – Two fields that specify an input variable to compare, the type of comparison, and the value to compare the variable to. Choice Rules support comparison between two variables. Within a Choice Rule, the value of variable can be compared with another value from the state input by appending `Path` to name of supported comparison operators. The values of `Variable` and `Path` fields in a comparison must be valid [Reference Paths](#).
- A **Next field** – The value of this field must match a state name in the state machine.

The following example checks whether the numerical value is equal to 1.

```
{
  "Variable": "$.foo",
  "NumericEquals": 1,
  "Next": "FirstMatchState"
}
```

The following example checks whether the string is equal to `MyString`.

```
{
  "Variable": "$.foo",
  "StringEquals": "MyString",
  "Next": "FirstMatchState"
}
```

The following example checks whether the string is greater than `MyStringABC`.

```
{
  "Variable": "$.foo",
  "StringGreaterThan": "MyStringABC",
}
```

```
"Next": "FirstMatchState"
}
```

The following example checks whether the string is null.

```
{
  "Variable": "$.possiblyNullValue",
  "IsNull": true
}
```

The following example shows how the `StringEquals` rule is only evaluated when `$.keyThatMightNotExist` exists because of the preceding `IsPresent` Choice Rule.

```
"And": [
  {
    "Variable": "$.keyThatMightNotExist",
    "IsPresent": true
  },
  {
    "Variable": "$.keyThatMightNotExist",
    "StringEquals": "foo"
  }
]
```

The following example checks whether a pattern with a wildcard matches.

```
{
  "Variable": "$.foo",
  "StringMatches": "log-*.txt"
}
```

The following example checks whether the timestamp is equal to `2001-01-01T12:00:00Z`.

```
{
  "Variable": "$.foo",
  "TimestampEquals": "2001-01-01T12:00:00Z",
  "Next": "FirstMatchState"
}
```

The following example compares a variable with another value from the state input.

```
{
  "Variable": "$.foo",
  "StringEqualsPath": "$.bar"
}
```

Step Functions examines each of the Choice Rules in the order listed in the Choices field. Then it transitions to the state specified in the Next field of the first Choice Rule in which the variable matches the value according to the comparison operator.

The following comparison operators are supported:

- And
- BooleanEquals, BooleanEqualsPath
- IsBoolean
- IsNull
- IsNumeric
- IsPresent
- IsString
- IsTimestamp
- Not
- NumericEquals, NumericEqualsPath
- NumericGreaterThan, NumericGreaterThanPath
- NumericGreaterThanEquals, NumericGreaterThanEqualsPath
- NumericLessThan, NumericLessThanPath
- NumericLessThanEquals, NumericLessThanEqualsPath
- Or
- StringEquals, StringEqualsPath
- StringGreaterThan, StringGreaterThanPath
- StringGreaterThanEquals, StringGreaterThanEqualsPath
- StringLessThan, StringLessThanPath
- StringLessThanEquals, StringLessThanEqualsPath

- `StringMatches`
- `TimestampEquals`, `TimestampEqualsPath`
- `TimestampGreaterThan`, `TimestampGreaterThanPath`
- `TimestampGreaterThanEquals`, `TimestampGreaterThanEqualsPath`
- `TimestampLessThan`, `TimestampLessThanPath`
- `TimestampLessThanEquals`, `TimestampLessThanEqualsPath`

For each of these operators, the corresponding value must be of the appropriate type: string, number, Boolean, or timestamp. Step Functions doesn't attempt to match a numeric field to a string value. However, because timestamp fields are logically strings, it's possible that a field considered to be a timestamp can be matched by a `StringEquals` comparator.

Note

For interoperability, don't assume that numeric comparisons work with values outside the magnitude or precision that the [IEEE 754-2008 binary64 data type](#) represents. In particular, integers outside of the range $[-2^{53}+1, 2^{53}-1]$ might fail to compare in the expected way.

Timestamps (for example, `2016-08-18T17:33:00Z`) must conform to [RFC3339 profile ISO 8601](#), with further restrictions:

- An uppercase T must separate the date and time portions.
- An uppercase Z must denote that a numeric time zone offset isn't present.

To understand the behavior of string comparisons, see the [Java compareTo documentation](#).

The values of the `And` and `Or` operators must be non-empty arrays of Choice Rules that must not themselves contain `Next` fields. Likewise, the value of a `Not` operator must be a single Choice Rule that must not contain `Next` fields.

You can create complex, nested Choice Rules using `And`, `Not`, and `Or`. However, the `Next` field can appear only in a top-level Choice Rule.

String comparison against patterns with one or more wildcards ("`*`") can be performed with the `StringMatches` comparison operator. The wildcard character is escaped by using the standard `\\` (Ex: "`*`"). No characters other than "`*`" have any special meaning during matching.

Choice State Example

The following is an example of a Choice state and other states that it transitions to.

Note

You must specify the `$.type` field. If the state input doesn't contain the `$.type` field, the execution fails and an error is displayed in the execution history. You can only specify a string in the `StringEquals` field that matches a literal value. For example, `"StringEquals": "Buy"`.

```
"ChoiceStateX": {
  "Type": "Choice",
  "Choices": [
    {
      "Not": {
        "Variable": "$.type",
        "StringEquals": "Private"
      },
      "Next": "Public"
    },
    {
      "Variable": "$.value",
      "NumericEquals": 0,
      "Next": "ValueIsZero"
    },
    {
      "And": [
        {
          "Variable": "$.value",
          "NumericGreaterThanEquals": 20
        },
        {
          "Variable": "$.value",
          "NumericLessThan": 30
        }
      ],
      "Next": "ValueInTwenties"
    }
  ],
  "Default": "DefaultState"
}
```

```
},  
  
"Public": {  
  "Type" : "Task",  
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Foo",  
  "Next": "NextState"  
},  
  
"ValueIsZero": {  
  "Type" : "Task",  
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Zero",  
  "Next": "NextState"  
},  
  
"ValueInTwenties": {  
  "Type" : "Task",  
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Bar",  
  "Next": "NextState"  
},  
  
"DefaultState": {  
  "Type": "Fail",  
  "Cause": "No Matches!"  
}
```

In this example, the state machine starts with the following input value.

```
{  
  "type": "Private",  
  "value": 22  
}
```

Step Functions transitions to the `ValueInTwenties` state, based on the `value` field.

If there are no matches for the Choice state's Choices, the state provided in the `Default` field runs instead. If the `Default` state isn't specified, the execution fails with an error.

Wait

A `Wait` state (`"Type": "Wait"`) delays the state machine from continuing for a specified time. You can choose either a relative time, specified in seconds from when the state begins, or an absolute end time, specified as a timestamp.

In addition to the [common state fields](#), Wait states have one of the following fields.

Seconds

A time, in seconds, to wait before beginning the state specified in the Next field. You must specify time as a positive, integer value from 0 up to 99999999.

Timestamp

An absolute time to wait until beginning the state specified in the Next field.

Timestamps must conform to the RFC3339 profile of ISO 8601, with the further restrictions that an uppercase T must separate the date and time portions, and an uppercase Z must denote that a numeric time zone offset is not present, for example, 2024-08-18T17:33:00Z.

Note

Currently, if you specify the wait time as a timestamp, Step Functions considers the time value up to seconds and truncates milliseconds.

SecondsPath

A time, in seconds, to wait before beginning the state specified in the Next field, specified using a [path](#) from the state's input data.

You must specify an integer value for this field.

TimestampPath

An absolute time to wait until beginning the state specified in the Next field, specified using a [path](#) from the state's input data.

Note

You must specify exactly one of Seconds, Timestamp, SecondsPath, or TimestampPath. In addition, the maximum wait time that you can specify for Standard Workflows and Express workflows is one year and five minutes respectively.

Wait State Examples

The following Wait state introduces a 10-second delay into a state machine.

```
"wait_ten_seconds": {
  "Type": "Wait",
  "Seconds": 10,
  "Next": "NextState"
}
```

In the next example, the Wait state waits until an absolute time: March 14, 2024, at 1:59 AM UTC.

```
"wait_until" : {
  "Type": "Wait",
  "Timestamp": "2024-03-14T01:59:00Z",
  "Next": "NextState"
}
```

You don't have to hard-code the wait duration. For example, given the following input data:

```
{
  "expirydate": "2024-03-14T01:59:00Z"
}
```

You can select the value of "expirydate" from the input using a reference [path](#) to select it from the input data.

```
"wait_until" : {
  "Type": "Wait",
  "TimestampPath": "$.expirydate",
  "Next": "NextState"
}
```

Succeed

A Succeed state ("Type": "Succeed") stops an execution successfully. The Succeed state is a useful target for Choice state branches that don't do anything but stop the execution.

Because Succeed states are terminal states, they have no Next field, and don't need an End field, as shown in the following example.

```
"SuccessState": {  
  "Type": "Succeed"  
}
```

Fail

A `Fail` state (`"Type": "Fail"`) stops the execution of the state machine and marks it as a failure, unless it is caught by a `Catch` block.

The `Fail` state only allows the use of `Type` and `Comment` fields from the set of [common state fields](#). In addition, the `Fail` state allows the following fields.

Cause (Optional)

A custom string that describes the cause of the error. You can specify this field for operational or diagnostic purposes.

CausePath (Optional)

If you want to provide a detailed description about the cause of the error dynamically from the state input using a [reference path](#), use `CausePath`. When resolved, the reference path must select a field that contains a string value.

You can also specify `CausePath` using an [intrinsic function](#) that returns a string. These intrinsics are: [States.Format](#), [States.JsonToString](#), [States.ArrayGetItem](#), [States.Base64Encode](#), [States.Base64Decode](#), [States.Hash](#), and [States.UUID](#).

Important

- You can specify either `Cause` or `CausePath`, but not both in your `Fail` state definition.
- As an information security best practice, we recommend that you remove any sensitive information or internal system details from the cause description.

Error (Optional)

An error name that you can provide to perform error handling using [Retry](#) or [Catch](#) fields. You can also provide an error name for operational or diagnostic purposes.

ErrorPath (Optional)

If you want to provide a name for the error dynamically from the state input using a [reference path](#), use `ErrorPath`. When resolved, the reference path must select a field that contains a string value.

You can also specify `ErrorPath` using an [intrinsic function](#) that returns a string. These intrinsics are: [States.Format](#), [States.JsonToString](#), [States.ArrayGetItem](#), [States.Base64Encode](#), [States.Base64Decode](#), [States.Hash](#), and [States.UUID](#).

Important

- You can specify either `Error` or `ErrorPath`, but not both in your Fail state definition.
- As an information security best practice, we recommend that you remove any sensitive information or internal system details from the error name.

Because Fail states always exit the state machine, they have no `Next` field and don't require an `End` field.

Fail state definition examples

The following Fail state definition example specifies static `Error` and `Cause` field values.

```
"FailState": {
  "Type": "Fail",
  "Cause": "Invalid response.",
  "Error": "ErrorA"
}
```

The following Fail state definition example uses reference paths dynamically to resolve the `Error` and `Cause` field values.

```
"FailState": {
  "Type": "Fail",
  "CausePath": "$.Cause",
  "ErrorPath": "$.Error"
}
```

The following Fail state definition example uses the [States.Format](#) intrinsic function to specify the `Error` and `Cause` field values dynamically.

```
"FailState": {
  "Type": "Fail",
  "CausePath": "States.Format('This is a custom error message for {}, caused by {}. ',
$.Error, $.Cause)",
  "ErrorPath": "States.Format('{}', $.Error)"
}
```

Parallel

The `Parallel` state (`"Type": "Parallel"`) can be used to add separate branches of execution in your state machine.

In addition to the [common state fields](#), `Parallel` states include these additional fields.

Branches (Required)

An array of objects that specify state machines to execute in parallel. Each such state machine object must have fields named `States` and `StartAt`, whose meanings are exactly like those in the top level of a state machine.

ResultPath (Optional)

Specifies where (in the input) to place the output of the branches. The input is then filtered as specified by the `OutputPath` field (if present) before being used as the state's output. For more information, see [Input and Output Processing](#).

ResultSelector (Optional)

Pass a collection of key value pairs, where the values are static or selected from the result. For more information, see [ResultSelector](#).

Retry (Optional)

An array of objects, called `Retriers`, that define a retry policy in case the state encounters runtime errors. For more information, see [State machine examples using Retry and using Catch](#).

Catch (Optional)

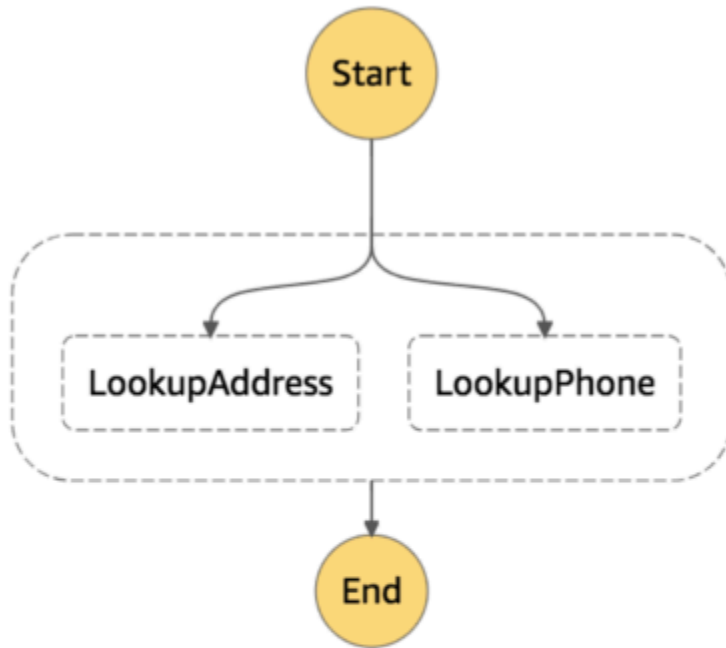
An array of objects, called `Catchers`, that define a fallback state that is executed if the state encounters runtime errors and its retry policy is exhausted or isn't defined. For more information, see [Fallback States](#).

A `Parallel` state causes AWS Step Functions to execute each branch, starting with the state named in that branch's `StartAt` field, as concurrently as possible, and wait until all branches terminate (reach a terminal state) before processing the `Parallel` state's `Next` field.

Parallel State Example

```
{
  "Comment": "Parallel Example.",
  "StartAt": "LookupCustomerInfo",
  "States": {
    "LookupCustomerInfo": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "LookupAddress",
          "States": {
            "LookupAddress": {
              "Type": "Task",
              "Resource":
                "arn:aws:lambda:us-east-1:123456789012:function:AddressFinder",
              "End": true
            }
          }
        },
        {
          "StartAt": "LookupPhone",
          "States": {
            "LookupPhone": {
              "Type": "Task",
              "Resource":
                "arn:aws:lambda:us-east-1:123456789012:function:PhoneFinder",
              "End": true
            }
          }
        }
      ]
    }
  }
}
```


In this example, the `LookupAddress` and `LookupPhone` branches are executed in parallel. Here is how the visual workflow looks in the Step Functions console.



Each branch must be self-contained. A state in one branch of a `Parallel` state must not have a `Next` field that targets a field outside of that branch, nor can any other state outside the branch transition into that branch.

Parallel State Input and Output Processing

A `Parallel` state provides each branch with a copy of its own input data (subject to modification by the `InputPath` field). It generates output that is an array with one element for each branch, containing the output from that branch. There is no requirement that all elements be of the same type. The output array can be inserted into the input data (and the whole sent as the `Parallel` state's output) by using a `ResultPath` field in the usual way (see [Input and Output Processing](#)).

```
{
  "Comment": "Parallel Example.",
  "StartAt": "FunWithMath",
  "States": {
    "FunWithMath": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "Add",
          "States": {
            "Add": {
              "Type": "Task",
              "Resource": "arn:aws:states:us-east-1:123456789012:activity:Add",
              "End": true
            }
          }
        },
        {
          "StartAt": "Subtract",
          "States": {
            "Subtract": {
              "Type": "Task",
              "Resource": "arn:aws:states:us-east-1:123456789012:activity:Subtract",
              "End": true
            }
          }
        }
      ]
    }
  }
}
```

If the `FunWithMath` state was given the array `[3, 2]` as input, then both the `Add` and `Subtract` states receive that array as input. The output of the `Add` and `Subtract` tasks would be the sum of and difference between the array elements 3 and 2, which is 5 and 1, while the output of the `Parallel` state would be an array.

```
[ 5, 1 ]
```

Tip

If the `Parallel` or `Map` state you use in your state machines returns an array of arrays, you can transform them into a flat array with the [ResultSelector](#) field. For more information, see [Flattening an array of arrays](#).

Error Handling

If any branch fails, because of an unhandled error or by transitioning to a `Fail` state, the entire `Parallel` state is considered to have failed and all its branches are stopped. If the error is not handled by the `Parallel` state itself, Step Functions stops the execution with an error.

Note

When a parallel state fails, invoked Lambda functions continue to run and activity workers processing a task token are not stopped.

- To stop long-running activities, use heartbeats to detect if its branch has been stopped by Step Functions, and stop workers that are processing tasks. Calling [SendTaskHeartbeat](#), [SendTaskSuccess](#), or [SendTaskFailure](#) will throw an error if the state has failed. See [Heartbeat Errors](#).
- Running Lambda functions cannot be stopped. If you have implemented a fallback, use a `Wait` state so that cleanup work happens after the Lambda function has finished.

Map

Use the `Map` state to run a set of workflow steps for each item in a dataset. The `Map` state's iterations run in parallel, which makes it possible to process a dataset quickly. `Map` states can use a variety of input types, including a JSON array, a list of Amazon S3 objects, or a CSV file.

Step Functions provides two types of processing modes for using the `Map` state in your workflows: *Inline* mode and *Distributed* mode.

For information about these modes, and how to use the `Map` state in either mode, see the following topics:

- [Map state processing modes](#)

- [Using Map state in Inline mode](#)
- [Using Map state in Distributed mode](#)

Tip

To deploy an example of a workflow that uses a Map state to your AWS account, see [Module 5 - Choice State and Map State](#) of *The AWS Step Functions Workshop*.

Map state processing modes

Step Functions provides the following processing modes for the Map state depending on how you want to process the items in a dataset.

- **Inline** – Limited-concurrency mode. In this mode, each iteration of the Map state runs in the context of the workflow that contains the Map state. Step Functions adds the execution history of these iterations to the parent workflow's execution history. By default, Map states run in Inline mode.

In this mode, the Map state accepts only a JSON array as input. Also, this mode supports up to 40 concurrent iterations.

For more information, see [Using Map state in Inline mode](#).

- **Distributed** – High-concurrency mode. In this mode, the Map state runs each iteration as a child workflow execution, which enables high concurrency of up to 10,000 parallel child workflow executions. Each child workflow execution has its own, separate execution history from that of the parent workflow.

In this mode, the Map state can accept either a JSON array or an Amazon S3 data source, such as a CSV file, as its input.

For more information, see [Using Map state in Distributed mode](#).

The mode you should use depends on how you want to process the items in a dataset. Use the Map state in Inline mode if your workflow's execution history won't exceed 25,000 entries, or if you don't require more than 40 concurrent iterations.

Use the Map state in Distributed mode when you need to orchestrate large-scale parallel workloads that meet any combination of the following conditions:

- The size of your dataset exceeds 256 KB.
- The workflow's execution event history exceeds 25,000 entries.
- You need a concurrency of more than 40 parallel iterations.

Topics

- [Inline mode and Distributed mode differences](#)
- [Using Map state in Inline mode](#)
- [Using Map state in Distributed mode to orchestrate large-scale parallel workloads](#)

Inline mode and Distributed mode differences

The following table highlights the differences between the Inline and Distributed modes.

Inline mode

Supported data sources

Accepts a JSON array passed from a previous step in the workflow as input.

Map iterations

In this mode, each iteration of the Map state runs in the context of the workflow that contains the Map state. Step Functions adds

Distributed mode

Accepts the following data sources as input:

- JSON array passed from a previous step in the workflow
- JSON file in an Amazon S3 bucket that contains an array
- CSV file in an Amazon S3 bucket
- Amazon S3 object list
- Amazon S3 inventory

In this mode, the Map state runs each iteration as a child workflow execution, which enables high concurrency of up to 10,000 parallel child workflow executions. Each child workflow

Inline mode

the execution history of these iterations to the parent workflow's execution history.

Maximum concurrency for parallel iterations

Lets you run up to 40 iterations as concurrently as possible.

Input payload and event history sizes

Enforces a limit of 256 KB on the input payload size and 25,000 entries in the execution event history.

Monitoring and observability

You can review the workflow's execution history from the console or by invoking the [GetExecutionHistory](#) API action.

You can also view the execution history through CloudWatch and X-Ray.

Distributed mode

execution has its own, separate execution history from that of the parent workflow.

Lets you run up to 10,000 parallel child workflow executions to process millions of data items at one time.

Lets you overcome the payload size limitation because the Map state can read input directly from Amazon S3 data sources.

In this mode, you can also overcome execution history limitations because the child workflow executions started by the Map state maintain their own, separate execution histories from the parent workflow's execution history.

When you run a Map state in Distributed mode, Step Functions creates a Map Run resource. A Map Run refers to a set of child workflow executions that a *Distributed Map state* starts. You can view a Map Run in the Step Functions console. You can also invoke the [DescribeMapRun](#) API action. A Map Run also emits metrics to CloudWatch.

For more information, see [Examining Map Run of a Distributed Map state execution](#).

Using Map state in Inline mode

By default, Map states runs in **Inline** mode. In Inline mode, the Map state accepts only a JSON array as input. It receives this array from a previous step in the workflow. In this mode, each iteration of the Map state runs in the context of the workflow that contains the Map state. Step Functions adds the execution history of these iterations to the parent workflow's execution history.

In this mode, the Map state supports up to 40 concurrent iterations.

A Map state set to **Inline** is known as an *Inline Map state*. Use the Map state in Inline mode if your workflow's execution history won't exceed 25,000 entries, or if you don't require more than 40 concurrent iterations.

For an introduction to using the *Inline Map state*, see the tutorial [Repeat an action using Inline Map state](#).

Contents

- [Key concepts in this topic](#)
- [Inline Map state fields](#)
- [Deprecated fields](#)
- [Inline Map state example](#)
- [Inline Map state example with ItemSelector](#)
- [Inline Map state input and output processing](#)

Key concepts in this topic

Inline mode

A limited-concurrency mode of the Map state. In this mode, each iteration of the Map state runs in the context of the workflow that contains the Map state. Step Functions adds the execution history of these iterations to the parent workflow's execution history. Map states run in the Inline mode by default.

This mode accepts only a JSON array as input and supports up to 40 concurrent iterations.

Inline Map state

A Map state set to the **Inline** mode.

Map workflow

The set of steps that the Map state runs for each iteration.

Map state iteration

A repetition of the workflow defined inside of the Map state.

Inline Map state fields

To use the *Inline Map state* in your workflows, specify one or more of these fields. You specify these fields in addition to the [common state fields](#).

Type (Required)

Sets the type of state, such as Map.

ItemProcessor (Required)

Contains the following JSON objects that specify the Map state processing mode and definition.

The definition contains the set of steps to repeat for processing each array item.

- `ProcessorConfig` – An optional JSON object that specifies the processing mode for the Map state. This object contains the `Mode` sub-field. This field defaults to `INLINE`, which uses the Map state in Inline mode.

In this mode, the failure of any iteration causes the Map state to fail. All iterations stop when the Map state fails.

- `StartAt` – Specifies a string that indicates the first state in a workflow. This string is case-sensitive and must match the name of one of the state objects. This state runs first for each item in the dataset. Any execution input that you provide to the Map state passes to the `StartAt` state first.
- `States` – A JSON object containing a comma-delimited set of [states](#). In this object, you define the [Map workflow](#).

Note

- States within the `ItemProcessor` field can only transition to each other. No state outside the `ItemProcessor` field can transition to a state within it.

- The `ItemProcessor` field replaces the now deprecated [Iterator](#) field. Although you can continue to include Map states that use the `Iterator` field, we highly recommend that you replace this field with `ItemProcessor`.

[Step Functions Local](#) doesn't currently support the `ItemProcessor` field. We recommend that you use the `Iterator` field with Step Functions Local.

ItemsPath (Optional)

Specifies a [reference path](#) using the [JsonPath](#) syntax. This path selects the JSON node that contains the array of items inside the state input. For more information, see [ItemsPath](#).

ItemSelector (Optional)

Overrides the values of the input array items before they're passed on to each Map state iteration.

In this field, you specify a valid JSON that contains a collection of key-value pairs. These pairs can contain any of the following:

- Static values you define in your state machine definition.
- Values selected from the state input using a [path](#).
- Values accessed from the [context object](#).

For more information, see [ItemSelector](#).

The `ItemSelector` field replaces the now deprecated [Parameters](#) field. Although you can continue to include Map states that use the `Parameters` field, we highly recommend that you replace this field with `ItemSelector`.

MaxConcurrency (Optional)

Specifies an integer value that provides the upper bound on the number of Map state iterations that can run in parallel. For example, a `MaxConcurrency` value of 10 limits the Map state to 10 concurrent iterations running at one time.

Note

Concurrent iterations may be limited. When this occurs, some iterations won't begin until previous iterations are complete. The likelihood of this occurring increases when your input array has more than 40 items.

To achieve a higher concurrency, consider [Using Map state in Distributed mode](#).

The default value is 0, which places no limit on concurrency. Step Functions invokes iterations as concurrently as possible.

A `MaxConcurrency` value of 1 invokes the `ItemProcessor` once for each array element. Items in the array are processed in the order of their appearance in the input. Step Functions doesn't start a new iteration until it completes the previous iteration.

MaxConcurrencyPath (Optional)

If you want to provide a maximum concurrency value dynamically from the state input using a reference path, use `MaxConcurrencyPath`. When resolved, the reference path must select a field whose value is a non-negative integer.

Note

A Map state cannot include both `MaxConcurrency` and `MaxConcurrencyPath`.

ResultPath (Optional)

Specifies where in the input to store the output of the Map state's iterations. The Map state then filters the input as specified by the `OutputPath` field, if specified. Then, it uses the filtered input as the state's output. For more information, see [Input and Output Processing](#).

ResultSelector (Optional)

Pass a collection of key value pairs, where the values are either static or selected from the result. For more information, see [ResultSelector](#).

Tip

If the Parallel or Map state you use in your state machines returns an array of arrays, you can transform them into a flat array with the `ResultSelector` field. For more information, see [Flattening an array of arrays](#).

Retry (Optional)

An array of objects, called Retriers, that define a retry policy. States use a retry policy when they encounter runtime errors. For more information, see [State machine examples using Retry and using Catch](#).

Note

If you define Retriers for the *Inline Map state*, the retry policy applies to all Map state iterations, instead of only failed iterations. For example, your Map state contains two successful iterations and one failed iteration. If you've defined the `Retry` field for the Map state, the retry policy applies to all three Map state iterations instead of only the failed iteration.

Catch (Optional)

An array of objects, called Catchers, that define a fallback state. States run a catcher if they encounter runtime errors and either don't have a retry policy, or their retry policy is exhausted. For more information, see [Fallback States](#).

Deprecated fields

Note

Although you can continue to include Map states that use the following fields, we highly recommend that you replace `Iterator` with [ItemProcessor](#) and `Parameters` with [ItemSelector](#).

Iterator

Specifies a JSON object that defines a set of steps that process each element of the array.

Parameters

Specifies a collection of key-value pairs, where the values can contain any of the following:

- Static values that you define in your state machine definition.
- Values selected from the input using a [path](#).

Inline Map state example

Consider the following input data for a Map state running in **Inline** mode.

```
{
  "ship-date": "2016-03-14T01:59:00Z",
  "detail": {
    "delivery-partner": "UQS",
    "shipped": [
      { "prod": "R31", "dest-code": 9511, "quantity": 1344 },
      { "prod": "S39", "dest-code": 9511, "quantity": 40 },
      { "prod": "R31", "dest-code": 9833, "quantity": 12 },
      { "prod": "R40", "dest-code": 9860, "quantity": 887 },
      { "prod": "R40", "dest-code": 9511, "quantity": 1220 }
    ]
  }
}
```

Given the previous input, the Map state in the following example invokes an AWS Lambda function named `ship-val` once for each item of the array in the `shipped` field.

```
"Validate All": {
  "Type": "Map",
  "InputPath": "$.detail",
  "ItemProcessor": {
    "ProcessorConfig": {
      "Mode": "INLINE"
    },
    "StartAt": "Validate",
    "States": {
      "Validate": {
        "Type": "Task",
        "Resource": "arn:aws:states:::lambda:invoke",
        "OutputPath": "$.Payload",
        "Parameters": {
          "FunctionName": "arn:aws:lambda:us-
east-2:123456789012:function:ship-val:$LATEST"
        },
        "End": true
      }
    }
  },
  "End": true,
```

```
"ResultPath": "$.detail.shipped",
"ItemsPath": "$.shipped"
}
```

Each iteration of the Map state sends an item in the array, selected with the [ItemsPath](#) field, as input to the `ship-val` Lambda function. The following values are an example of input the Map state sends to an invocation of the Lambda function:

```
{
  "prod": "R31",
  "dest-code": 9511,
  "quantity": 1344
}
```

When complete, the output of the Map state is a JSON array, where each item is the output of an iteration. In this case, this array contains the output of the `ship-val` Lambda function.

Inline Map state example with ItemSelector

Suppose that the `ship-val` Lambda function in the previous example also needs information about the shipment's courier. This information is in addition to the items in the array for each iteration. You can include information from the input, along with information specific to the current iteration of the Map state. Note the `ItemSelector` field in the following example:

```
"Validate-All": {
  "Type": "Map",
  "InputPath": "$.detail",
  "ItemsPath": "$.shipped",
  "MaxConcurrency": 0,
  "ResultPath": "$.detail.shipped",
  "ItemSelector": {
    "parcel.$": "$$.Map.Item.Value",
    "courier.$": "$.delivery-partner"
  },
  "ItemProcessor": {
    "StartAt": "Validate",
    "States": {
      "Validate": {
        "Type": "Task",

```

```
    }
  },
  "End": true
}
```

The `ItemSelector` block replaces the input to the iterations with a JSON node. This node contains both the current item data from the [context object](#) and the courier information from the Map state input's `delivery-partner` field. The following is an example of input to a single iteration. The Map state passes this input to an invocation of the `ship-val` Lambda function.

```
{
  "parcel": {
    "prod": "R31",
    "dest-code": 9511,
    "quantity": 1344
  },
  "courier": "UQS"
}
```

In the previous *Inline Map state* example, the `ResultPath` field produces output in the same format as the input. However, it overwrites the `detail.shipped` field with an array in which each element is the output of each iteration's `ship-val` Lambda invocation.

For more information about using the *Inline Map state* state and its fields, see the following.

- [Repeat an action using Inline Map state](#)
- [Input and Output Processing in Step Functions](#)
- [ItemsPath](#)
- [Context Object Data for Map States](#)

Inline Map state input and output processing

For a given Map state, [InputPath](#) selects a subset of the state's input.

The input of a Map state must include a JSON array. The Map state runs the `ItemProcessor` section once for each item in the array. If you specify the [ItemsPath](#) field, the Map state selects where in the input to find the array to iterate over. If not specified, the value of `ItemsPath` is `$`, and the `ItemProcessor` section expects that the array is the only input. If you specify the `ItemsPath` field, its value must be a [Reference Path](#). The Map state applies this path to the

effective input after it applies the `InputPath`. The `ItemsPath` must identify a field whose value is a JSON array.

The input to each iteration, by default, is a single element of the array field identified by the `ItemsPath` value. You can override this value with the [ItemSelector](#) field.

When complete, the output of the Map state is a JSON array, where each item is the output of an iteration.

For more information about Inline Map state inputs and outputs, see the following:

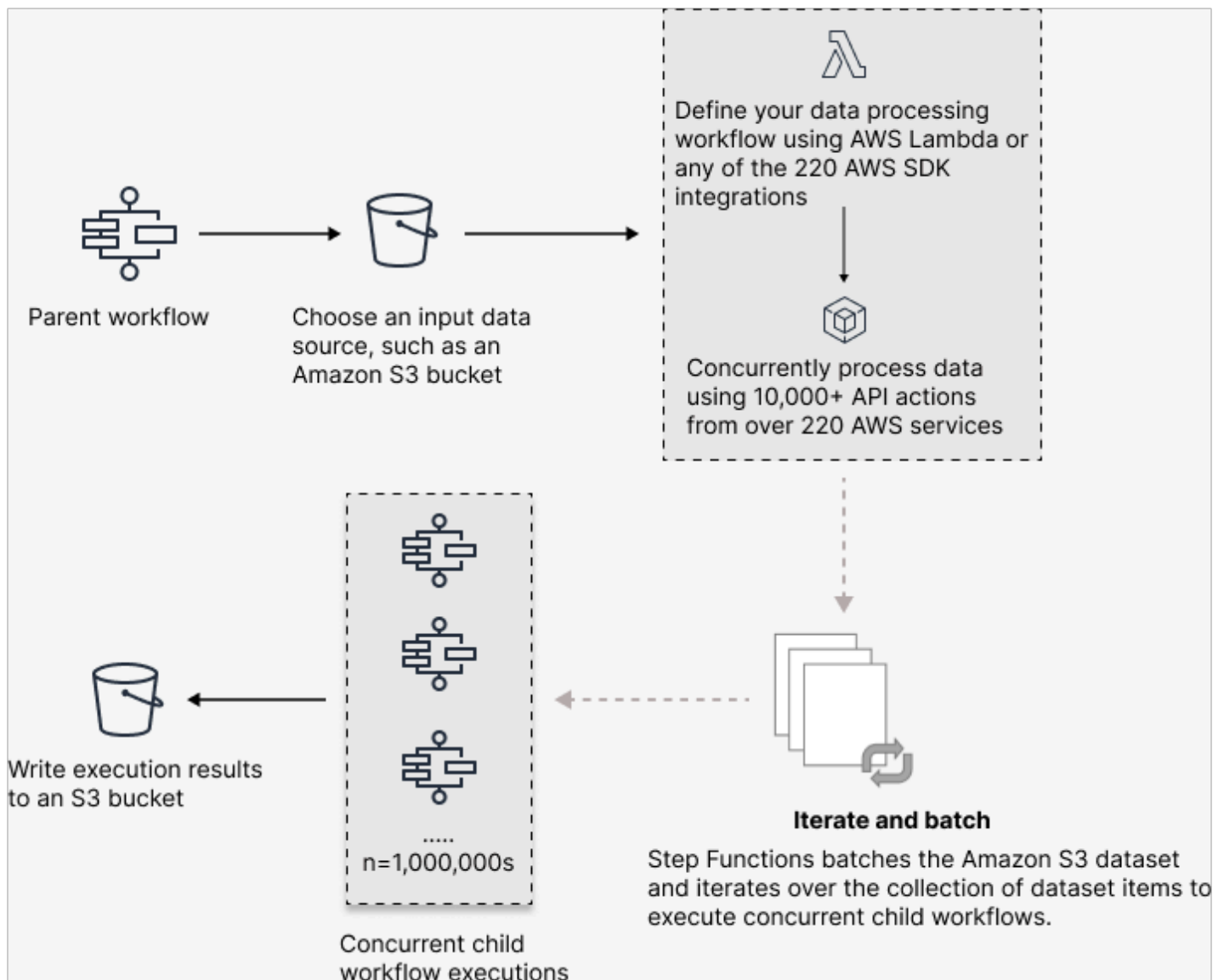
- [Repeat an action using Inline Map state](#)
- [Inline Map state example with ItemSelector](#)
- [Input and Output Processing in Step Functions](#)
- [Context Object Data for Map States](#)
- [Dynamically process data with a Map state](#)

Using Map state in Distributed mode to orchestrate large-scale parallel workloads

With Step Functions, you can orchestrate large-scale parallel workloads to perform tasks, such as on-demand processing of semi-structured data. These parallel workloads let you concurrently process large-scale data sources stored in Amazon S3. For example, you might process a single JSON or CSV file that contains large amounts of data. Or you might process a large set of Amazon S3 objects.

To set up a large-scale parallel workload in your workflows, include a Map state in Distributed mode. The *Map state* processes items in a dataset concurrently. A Map state set to **Distributed** is known as a *Distributed Map state*. In *Distributed mode*, the Map state allows high-concurrency processing. In Distributed mode, the Map state processes the items in the dataset in iterations called *child workflow executions*. You can specify the number of child workflow executions that can run in parallel. Each child workflow execution has its own, separate execution history from that of the parent workflow. If you don't specify, Step Functions runs 10,000 parallel child workflow executions in parallel.

The following illustration explains how you can set up large-scale parallel workloads in your workflows.



In this topic

- [Key terms](#)
- [Distributed Map state definition example](#)
- [Permissions to run Distributed Map](#)
- [Distributed Map state fields](#)
- [Next steps](#)

Key terms

Distributed mode

A processing mode of the [Map state](#). In this mode, each iteration of the Map state runs as a child workflow execution that enables high concurrency. Each child workflow execution has its own execution history, which is separate from the parent workflow's execution history. This mode supports reading input from large-scale Amazon S3 data sources.

Distributed Map state

A Map state set to **Distributed** [processing mode](#).

Map workflow

A set of steps that a Map state runs.

Parent workflow

A workflow that contains one or more Distributed Map states.

Child workflow execution

An iteration of the *Distributed Map state*. A child workflow execution has its own execution history, which is separate from the parent workflow's execution history.

Map Run

When you run a Map state in Distributed mode, Step Functions creates a Map Run resource. A Map Run refers to a set of child workflow executions that a *Distributed Map state* starts, and the runtime settings that control these executions. Step Functions assigns an Amazon Resource Name (ARN) to your Map Run. You can examine a Map Run in the Step Functions console. You can also invoke the [DescribeMapRun](#) API action. A Map Run also emits metrics to CloudWatch.

For more information, see [Examining Map Run](#).

Distributed Map state definition example

Use the Map state in Distributed mode when you need to orchestrate large-scale parallel workloads that meet any combination of the following conditions:

- The size of your dataset exceeds 256 KB.

- The workflow's execution event history exceeds 25,000 entries.
- You need a concurrency of more than 40 parallel iterations.

The following *Distributed Map state* definition example specifies the dataset as a CSV file stored in an Amazon S3 bucket. It also specifies a Lambda function that processes the data in each row of the CSV file. Because this example uses a CSV file, it also specifies the location of the CSV column headers. To view the complete state machine definition of this example, see the tutorial [Copying large-scale CSV data using Distributed Map](#).

```
{
  "Map": {
    "Type": "Map",
    "ItemReader": {
      "ReaderConfig": {
        "InputType": "CSV",
        "CSVHeaderLocation": "FIRST_ROW"
      },
      "Resource": "arn:aws:states:::s3:getObject",
      "Parameters": {
        "Bucket": "Database",
        "Key": "csv-dataset/ratings.csv"
      }
    },
    "ItemProcessor": {
      "ProcessorConfig": {
        "Mode": "DISTRIBUTED",
        "ExecutionType": "EXPRESS"
      },
      "StartAt": "LambdaTask",
      "States": {
        "LambdaTask": {
          "Type": "Task",
          "Resource": "arn:aws:states:::lambda:invoke",
          "OutputPath": "$.Payload",
          "Parameters": {
            "Payload.$": "$",
            "FunctionName": "arn:aws:lambda:us-east-2:123456789012:function:processCSVData"
          },
          "End": true
        }
      }
    }
  }
}
```

```

    },
    "Label": "Map",
    "End": true,
    "ResultWriter": {
      "Resource": "arn:aws:states:::s3:putObject",
      "Parameters": {
        "Bucket": "myOutputBucket",
        "Prefix": "csvProcessJobs"
      }
    }
  }
}
}
}
}

```

Permissions to run Distributed Map

When you include a *Distributed Map state* in your workflows, Step Functions needs appropriate permissions to allow the state machine role to invoke the [StartExecution](#) API action for the *Distributed Map state*.

The following IAM policy example grants the least privileges required to your state machine role for running the *Distributed Map state*.

Note

Make sure that you replace *stateMachineName* with the name of the state machine in which you're using the *Distributed Map state*. For example, `arn:aws:states:us-east-2:123456789012:stateMachine:mystateMachine`.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": [
        "arn:aws:states:region:accountID:stateMachine:stateMachineName"
      ]
    }
  ],
}

```

```

{
  "Effect": "Allow",
  "Action": [
    "states:DescribeExecution",
    "states:StopExecution"
  ],
  "Resource": "arn:aws:states:region:accountID:execution:stateMachineName:*"
}
]
}

```

In addition, you need to make sure that you have the least privileges necessary to access the AWS resources used in the *Distributed Map state*, such as Amazon S3 buckets. For information, see [IAM policies for using Distributed Map state](#).

Distributed Map state fields

To use the *Distributed Map state* in your workflows, specify one or more of these fields. You specify these fields in addition to the [common state fields](#).

Type (Required)

Sets the type of state, such as Map.

ItemProcessor (Required)

Contains the following JSON objects that specify the Map state processing mode and definition.

- ProcessorConfig – A JSON object that specifies the configuration for the Map state. This object contains the following sub-fields:
 - Mode – Set to **DISTRIBUTED** to use the Map state in Distributed mode.

Note

Currently, if you use the Map state inside Express workflows, you can't set the Mode to DISTRIBUTED. However, if you use the Map state inside Standard workflows, you can set the Mode to DISTRIBUTED.

- ExecutionType – Specifies the execution type for the Map workflow as either **STANDARD** or **EXPRESS**. You must provide this field if you specified DISTRIBUTED for the Mode sub-field. For more information about workflow types, see [Standard vs. Express Workflows](#).

- **StartAt** – Specifies a string that indicates the first state in a workflow. This string is case-sensitive and must match the name of one of the state objects. This state runs first for each item in the dataset. Any execution input that you provide to the Map state passes to the StartAt state first.
- **States** – A JSON object containing a comma-delimited set of [states](#). In this object, you define the [Map workflow](#).

ItemReader

Specifies a dataset and its location. The Map state receives its input data from the specified dataset.

In Distributed mode, you can use either a JSON payload passed from a previous state or a large-scale Amazon S3 data source as the dataset. For more information, see [ItemReader](#).

ItemsPath (Optional)

Specifies a [reference path](#) using the [JsonPath](#) syntax to select the JSON node that contains an array of items inside the state input.

In Distributed mode, you specify this field only when you use a JSON array from a previous step as your state input. For more information, see [ItemsPath](#).

ItemSelector (Optional)

Overrides the values of individual dataset items before they're passed on to each Map state iteration.

In this field, you specify a valid JSON input that contains a collection of key-value pairs. These pairs can either be static values that you define in your state machine definition, values selected from the state input using a [path](#), or values accessed from the [context object](#). For more information, see [ItemSelector](#).

ItemBatcher (Optional)

Specifies to process the dataset items in batches. Each child workflow execution then receives a batch of these items as input. For more information, see [ItemBatcher](#).

MaxConcurrency (Optional)

Specifies the number of child workflow executions that can run in parallel. The interpreter only allows up to the specified number of parallel child workflow executions. If you don't specify

a concurrency value or set it to zero, Step Functions doesn't limit concurrency and runs 10,000 parallel child workflow executions.

Note

While you can specify a higher concurrency limit for parallel child workflow executions, we recommend that you don't exceed the capacity of a downstream AWS service, such as AWS Lambda.

MaxConcurrencyPath (Optional)

If you want to provide a maximum concurrency value dynamically from the state input using a reference path, use `MaxConcurrencyPath`. When resolved, the reference path must select a field whose value is a non-negative integer.

Note

A Map state cannot include both `MaxConcurrency` and `MaxConcurrencyPath`.

ToleratedFailurePercentage (Optional)

Defines the percentage of failed items to tolerate in a Map Run. The Map Run automatically fails if it exceeds this percentage. Step Functions calculates the percentage of failed items as the result of the total number of failed or timed out items divided by the total number of items. You must specify a value between zero and 100. For more information, see [Tolerated failure threshold for Distributed Map state](#).

ToleratedFailurePercentagePath (Optional)

If you want to provide a tolerated failure percentage value dynamically from the state input using a reference path, use `ToleratedFailurePercentagePath`. When resolved, the reference path must select a field whose value is between zero and 100.

ToleratedFailureCount (Optional)

Defines the number of failed items to tolerate in a Map Run. The Map Run automatically fails if it exceeds this number. For more information, see [Tolerated failure threshold for Distributed Map state](#).

ToleratedFailureCountPath (Optional)

If you want to provide a tolerated failure count value dynamically from the state input using a reference path, use `ToleratedFailureCountPath`. When resolved, the reference path must select a field whose value is a non-negative integer.

Label (Optional)

A string that uniquely identifies a Map state. For each Map Run, Step Functions adds the label to the Map Run ARN. The following is an example of a Map Run ARN with a custom label named `demoLabel`:

```
arn:aws:states:us-east-1:123456789012:mapRun:demoWorkflow/  
demoLabel:3c39a231-69bb-3d89-8607-9e124eddbb0b
```

If you don't specify a label, Step Functions automatically generates a unique label.

Note

Labels can't exceed 40 characters in length, must be unique within a state machine definition, and can't contain any of the following characters:

- Whitespace characters
- Wildcard characters (? *)
- Bracket characters (< > { } [])
- Special characters (: ; , \ | ^ ~ \$ # % & ` ")
- Control characters (\\u0000 - \\u001f or \\u007f - \\u009f).

Step Functions allows you to create names for state machines, executions, activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

ResultWriter (Optional)

Specifies the Amazon S3 location where Step Functions writes all child workflow execution results.

Step Functions consolidates all child workflow execution data, such as execution input and output, ARN, and execution status. It then exports executions with the same status to their respective files in the specified Amazon S3 location. For more information, see [ResultWriter](#).

If you don't export the Map state results, it returns an array of all the child workflow execution results. For example:

```
[1, 2, 3, 4, 5]
```

ResultPath (Optional)

Specifies where in the input to place the output of the iterations. The input is then filtered as specified by the [OutputPath](#) field if present, before it is passed as the state's output. For more information, see [Input and Output Processing](#).

ResultSelector (Optional)

Pass a collection of key-value pairs, where the values are static or selected from the result. For more information, see [ResultSelector](#).

Tip

If the Parallel or Map state you use in your state machines returns an array of arrays, you can transform them into a flat array with the [ResultSelector](#) field. For more information, see [Flattening an array of arrays](#).

Retry (Optional)

An array of objects, called Retriers, that define a retry policy. An execution uses the retry policy if the state encounters runtime errors. For more information, see [State machine examples using Retry and using Catch](#).

Note

If you define Retriers for the *Distributed Map state*, the retry policy applies to all of the child workflow executions the Map state started. For example, imagine your Map state started three child workflow executions, out of which one fails. When the failure occurs, the execution uses the `RETRY` field, if defined, for the Map state. The retry policy applies to all the child workflow executions and not just the failed execution. If one or more child workflow executions fails, the Map Run fails.

When you retry a Map state, it creates a new Map Run.

Catch (Optional)

An array of objects, called Catchers, that define a fallback state. Step Functions uses the Catchers defined in `Catch` if the state encounters runtime errors. When an error occurs, the execution first uses any retriers defined in `Retry`. If the retry policy isn't defined or is exhausted, the execution uses its Catchers, if defined. For more information, see [Fallback States](#).

Next steps

To continue learning more about *Distributed Map state*, see the following resources:

- **Input and output processing**

To configure the input that a *Distributed Map state* receives and the output that it generates, Step Functions provides the following fields:

- [ItemReader](#)
- [ItemsPath](#)
- [ItemSelector](#)
- [ItemBatcher](#)
- [ResultWriter](#)
- [Parsing an input CSV file](#)

In addition to these fields, Step Functions also provides you the ability to define a tolerated failure threshold for Distributed Map. This value lets you specify the maximum number of, or percentage of, failed items as a failure threshold for a [Map Run](#). For more information about configuring the tolerated failure threshold, see [Tolerated failure threshold for Distributed Map state](#).

- **Using Distributed Map state**

Refer the following tutorials and sample projects to get started with using *Distributed Map state*.

- [Getting started with using Distributed Map state](#)
- [Processing entire batch of data with a Lambda function](#)
- [Processing individual data items with a Lambda function](#)

- [Sample project: Process a CSV file with Distributed Map](#)
- [Sample project: Process data in an Amazon S3 bucket with Distributed Map](#)
- **Examine Distributed Map state execution**

The Step Functions console provides a *Map Run Details* page, which displays all the information related to a *Distributed Map state* execution. For information about how to examine the information displayed on this page, see [Examining Map Run](#).

Tolerated failure threshold for Distributed Map state

When you orchestrate large-scale parallel workloads, you can also define a tolerated failure threshold. This value lets you specify the maximum number of, or percentage of, failed items as a failure threshold for a [Map Run](#). Depending on which value you specify, your Map Run fails automatically if it exceeds the threshold. If you specify both values, the workflow fails when it exceeds either value.

Specifying a threshold helps you fail a specific number of items before the entire Map Run fails. Step Functions returns a [States.ExceedToleratedFailureThreshold](#) error when the Map Run fails because the specified threshold is exceeded.

Note

Step Functions may continue to run child workflows in a Map Run even after the tolerated failure threshold is exceeded, but before the Map Run fails.

To specify the threshold value in Workflow Studio, select **Set a tolerated failure threshold** in **Additional configuration** under the **Runtime settings** field.

Tolerated failure percentage

Defines the percentage of failed items to tolerate. Your Map Run fails if this value is exceeded. Step Functions calculates the percentage of failed items as the result of the total number of failed or timed out items divided by the total number of items. You must specify a value between zero and 100. The default percentage value is zero, which means that the workflow fails if any one of its child workflow executions fails or times out. If you specify the percentage as 100, the workflow won't fail even if all child workflow executions fail.

Alternatively, you can specify the percentage as a [reference path](#) to an existing key-value pair in your *Distributed Map state* input. This path must resolve to a positive integer between 0 and 100 at runtime. You specify the reference path in the `ToleratedFailurePercentagePath` sub-field.

For example, given the following input:

```
{
  "percentage": 15
}
```

You can specify the percentage using a reference path to that input as follows:

```
{
  ...
  "Map": {
    "Type": "Map",
    ...
    "ToleratedFailurePercentagePath": "$.percentage"
    ...
  }
}
```

Important

You can specify either `ToleratedFailurePercentage` or `ToleratedFailurePercentagePath`, but not both in your *Distributed Map state* definition.

Tolerated failure count

Defines the number of failed items to tolerate. Your Map Run fails if this value is exceeded.

Alternatively, you can specify the count as a [reference path](#) to an existing key-value pair in your *Distributed Map state* input. This path must resolve to a positive integer at runtime. You specify the reference path in the `ToleratedFailureCountPath` sub-field.

For example, given the following input:

```
{
  "count": 10
}
```

You can specify the number using a reference path to that input as follows:

```
{
  ...
  "Map": {
    "Type": "Map",
    ...
    "ToleratedFailureCountPath": "$.count"
    ...
  }
}
```

Important

You can specify either `ToleratedFailureCount` or `ToleratedFailureCountPath`, but not both in your *Distributed Map state* definition.

Transitions

When you start a new execution of your state machine, the system begins with the state referenced in the top-level `StartAt` field. This field, given as a string, must exactly match, including case, the name of a state in the workflow.

After a state runs, AWS Step Functions uses the value of the `Next` field to determine the next state to advance to.

`Next` fields also specify state names as strings. This string is case-sensitive and must match the name of a state specified in the state machine description exactly

For example, the following state includes a transition to `NextState`.

```
"SomeState" : {
  ...
  "Next" : "NextState"
```

```
}
```

Most states permit only a single transition rule with the `Next` field. However, certain flow-control states, such as a `Choice` state, allow you to specify multiple transition rules, each with its own `Next` field. The [Amazon States Language](#) provides details about each of the state types you can specify, including information about how to specify transitions.

States can have multiple incoming transitions from other states.

The process repeats until it either reaches a terminal state (a state with `"Type": Succeed`, `"Type": Fail`, or `"End": true`), or a runtime error occurs.

When you [redrive](#) an execution, it's considered as a state transition. In addition, all states that are rerun in a redrive are also considered as state transitions.

The following rules apply to states within a state machine:

- States can occur in any order within the enclosing block. However, the order in which they're listed doesn't affect the order in which they're run. That order is determined by the contents of the states.
- Within a state machine, there can be only one state designated as the `start` state. The `start` state is defined by the value of the `StartAt` field in the top-level structure.
- Depending on your state machine logic — for example, if your state machine has multiple logic branches — you may have more than one end state.
- If your state machine consists of only one state, it can be both the `start` and end state.

Transitions in Distributed Map state

When you use the `Map` state in `Distributed` mode, you'll be charged one state transition for each child workflow execution that the *Distributed Map state* starts. When you use the `Map` state in `Inline` mode, you aren't charged a state transition for each iteration of the *Inline Map state*.

You can optimize cost by using the `Map` state in `Distributed` mode and include a nested workflow in the `Map` state definition. The *Distributed Map state* also adds more value when you start child workflow executions of type **Express**. Step Functions stores the response and status of the `Express` child workflow executions, which reduces the need to store execution data in CloudWatch Logs. You can also get access to flow controls available with a *Distributed Map state*, such as defining

error thresholds or batching a group of items. For information about Step Functions pricing, see [AWS Step Functions pricing](#).

State Machine Data

State machine data takes the following forms:

- The initial input into a state machine
- Data passed between states
- The output from a state machine

This section describes how state machine data is formatted and used in AWS Step Functions.

Topics

- [Data Format](#)
- [State Machine Input/Output](#)
- [State Input/Output](#)

Data Format

State machine data is represented by JSON text. You can provide values to a state machine using any data type supported by JSON.

Note

- Numbers in JSON text format conform to JavaScript semantics. These numbers typically correspond to double-precision [IEEE-854](#) values.
- The following is valid JSON text:
 - Standalone, quote-delimited strings
 - Objects
 - Arrays
 - Numbers
 - Boolean values
 - `null`

- The output of a state becomes the input for the next state. However, you can restrict states to work on a subset of the input data by using [Input and Output Processing](#).

State Machine Input/Output

You can give your initial input data to an AWS Step Functions state machine in one of two ways. You can pass the data to a [StartExecution](#) action when you start an execution. You can also pass the data to the state machine from the [Step Functions console](#). Initial data is passed to the state machine's `StartAt` state. If no input is provided, the default is an empty object (`{}`).

The output of the execution is returned by the last state (`terminal`). This output appears as JSON text in the execution's result.

For Standard Workflows, you can retrieve execution results from the execution history using external callers, such as the [DescribeExecution](#) action. You can view execution results on the [Step Functions console](#).

For Express Workflows, if you enabled logging, you can retrieve results from CloudWatch Logs, or view and debug the executions in the Step Functions console. For more information, see [Logging using CloudWatch Logs](#) and [Viewing and debugging executions on the Step Functions console](#).

You should also consider quotas related to your state machine. For more information, see [Quotas](#)

State Input/Output

Each state's input consists of JSON text from the preceding state or, for the `StartAt` state, the input into the execution. Certain flow-control states echo their input to their output.

In the following example, the state machine adds two numbers together.

1. Define the AWS Lambda function.

```
function Add(input) {
  var numbers = JSON.parse(input).numbers;
  var total = numbers.reduce(
    function(previousValue, currentValue, index, array) {
      return previousValue + currentValue;
    });
  return JSON.stringify({ result: total });
}
```

2. Define the state machine.

```
{
  "Comment": "An example that adds two numbers together.",
  "StartAt": "Add",
  "Version": "1.0",
  "TimeoutSeconds": 10,
  "States": {
    {
      "Add": {
        "Type": "Task",
        "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Add",
        "End": true
      }
    }
  }
}
```

3. Start an execution with the following JSON text.

```
{ "numbers": [3, 4] }
```

The Add state receives the JSON text and passes it to the Lambda function.

The Lambda function returns the result of the calculation to the state.

The state returns the following value in its output.

```
{ "result": 7 }
```

Because Add is also the final state in the state machine, this value is returned as the state machine's output.

If the final state returns no output, then the state machine returns an empty object ({}).

For more information, see [Input and Output Processing in Step Functions](#).

Input and Output Processing in Step Functions

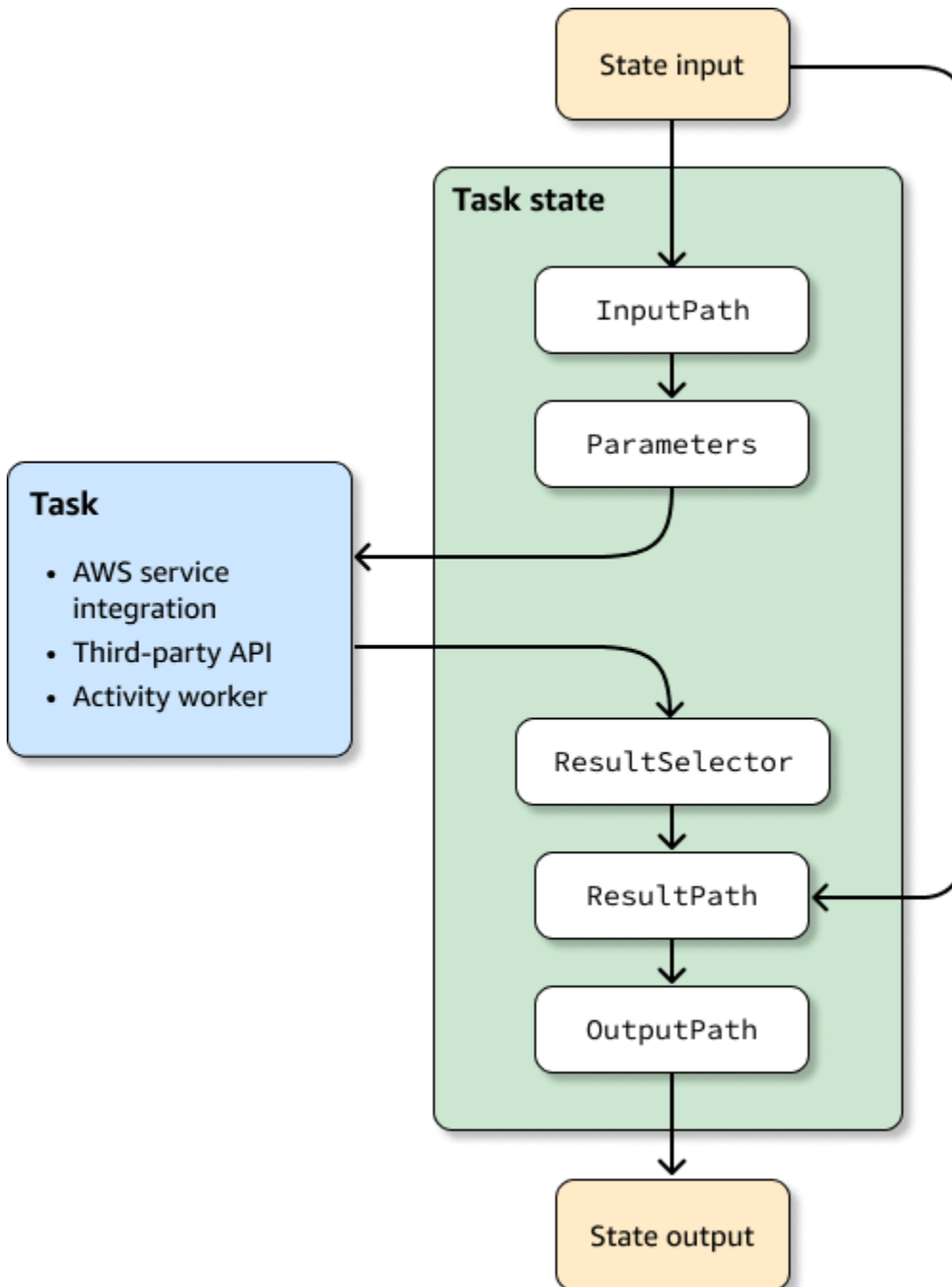
A Step Functions execution receives a JSON text as input and passes that input to the first state in the workflow. Individual states receive JSON as input and usually pass JSON as output to the

next state. Understanding how this information flows from state to state, and learning how to filter and manipulate this data, is key to effectively designing and implementing workflows in AWS Step Functions.

In the Amazon States Language, these fields filter and control the flow of JSON from state to state:

- `InputPath`
- `Parameters`
- `ResultSelector`
- `ResultPath`
- `OutputPath`

The following diagram shows how JSON information moves through a task state. `InputPath` selects which parts of the JSON input to pass to the task of the Task state (for example, an AWS Lambda function). `ResultPath` then selects what combination of the state input and the task result to pass to the output. `OutputPath` can filter the JSON output to further limit the information that's passed to the output.



`InputPath`, `Parameters`, `ResultSelector`, `ResultPath`, and `OutputPath` each manipulate JSON as it moves through each state in your workflow.

Each can use [paths](#) to select portions of the JSON from the input or the result. A path is a string, beginning with `$`, that identifies nodes within JSON text. Step Functions paths use [JsonPath](#) syntax.

Tip

Use the [data flow simulator in the Step Functions console](#) to test JSON path syntax, to better understand how data is manipulated within a state, and to see how data is passed between states.

Tip

To deploy an example of a workflow that includes input and output processing to your AWS account, see [Module 6 - Input and Output Processing](#) of *The AWS Step Functions Workshop*.

Topics

- [Paths](#)
- [InputPath, Parameters and ResultSelector](#)
- [ResultPath](#)
- [OutputPath](#)
- [InputPath, ResultPath, and OutputPath Examples](#)
- [Map state input and output fields](#)
- [Context object](#)

Paths

In the Amazon States Language, a *path* is a string beginning with \$ that you can use to identify components within JSON text. Paths follow [JsonPath](#) syntax. You can specify a path to access subsets of the input when specifying values for `InputPath`, `ResultPath`, and `OutputPath`. For more information see [Input and Output Processing in Step Functions](#).

Note

You can also specify a JSON node of the input or the context object by using paths within the `Parameters` field of a state definition. See [Pass parameters to a service API](#).

You must use square bracket notation if your field name contains any character that is not included in the `member-name-shorthand` definition of the [JsonPath ABNF](#) rule. Therefore, to encode special characters, such as punctuation marks (excluding `_`), you must use square bracket notation. For example, `$.abc.['def ghi']`.

Reference Paths

A *reference path* is a path whose syntax is limited in such a way that it can identify only a single node in a JSON structure:

- You can access object fields using only dot (`.`) and square bracket (`[]`) notation.
- Functions such as `length()` aren't supported.
- Lexical operators, which are non-symbolic, such as `subsetof` aren't supported.
- Filtering by regular expression or by referencing another value in the JSON structure isn't supported.
- The operators `@`, `,`, `:`, and `?` are not supported

For example, if state input data contains the following values:

```
{
  "foo": 123,
  "bar": ["a", "b", "c"],
  "car": {
    "cdr": true
  }
}
```

The following reference paths would return the following.

```
$.foo => 123
$.bar => ["a", "b", "c"]
$.car.cdr => true
```

Certain states use paths and reference paths to control the flow of a state machine or configure a state's settings or options. For more information, see [Modeling workflow input and output path processing with data flow simulator](#) and [Using JSONPath effectively in AWS Step Functions](#).

Flattening an array of arrays

If the [Parallel](#) or [Map](#) state in your state machines return an array of arrays, you can transform them into a flat array with the [ResultSelector](#) field. You can include this field inside the [Parallel](#) or [Map](#) state definition to manipulate the result of these states.

To flatten arrays, use the [JMESPath syntax \[* \]](#) in the `ResultSelector` field as shown in the following example.

```
"ResultSelector": {
  "flattenArray.$": "$[*][*]"
}
```

For examples that show how to flatten an array, see *Step 3* in the following tutorials:

- [Processing entire batch of data with a Lambda function](#)
- [Processing individual data items with a Lambda function](#)

InputPath, Parameters and ResultSelector

The `InputPath`, `Parameters` and `ResultSelector` fields provide a way to manipulate JSON as it moves through your workflow. `InputPath` can limit the input that is passed by filtering the JSON notation by using a path (see [Paths](#)). The `Parameters` field enables you to pass a collection of key-value pairs, where the values are either static values that you define in your state machine definition, or that are selected from the input using a path. The `ResultSelector` field provides a way to manipulate the state's result before `ResultPath` is applied.

AWS Step Functions applies the `InputPath` field first, and then the `Parameters` field. You can first filter your raw input to a selection you want using `InputPath`, and then apply `Parameters` to manipulate that input further, or add new values. You can then use the `ResultSelector` field to manipulate the state's output before `ResultPath` is applied.

Tip

Use the [data flow simulator in the Step Functions console](#) to test JSON path syntax, to better understand how data is manipulated within a state, and to see how data is passed between states.

InputPath

Use `InputPath` to select a portion of the state input.

For example, suppose the input to your state includes the following.

```
{
  "comment": "Example for InputPath.",
  "dataset1": {
    "val1": 1,
    "val2": 2,
    "val3": 3
  },
  "dataset2": {
    "val1": "a",
    "val2": "b",
    "val3": "c"
  }
}
```

You could apply the `InputPath`.

```
"InputPath": "$.dataset2",
```

With the previous `InputPath`, the following is the JSON that is passed as the input.

```
{
  "val1": "a",
  "val2": "b",
  "val3": "c"
}
```

Note

A path can yield a selection of values. Consider the following example.

```
{ "a": [1, 2, 3, 4] }
```

If you apply the path `$.a[0:2]`, the following is the result.

```
[ 1, 2 ]
```

Parameters

This section describes the different ways you can use the Parameters field.

Key-value pairs

Use the Parameters field to create a collection of key-value pairs that are passed as input. The values of each can either be static values that you include in your state machine definition, or selected from either the input or the context object with a path. For key-value pairs where the value is selected using a path, the key name must end in `.$`.

For example, suppose you provide the following input.

```
{
  "comment": "Example for Parameters.",
  "product": {
    "details": {
      "color": "blue",
      "size": "small",
      "material": "cotton"
    },
    "availability": "in stock",
    "sku": "2317",
    "cost": "$23"
  }
}
```

To select some of the information, you could specify these parameters in your state machine definition.

```
"Parameters": {
  "comment": "Selecting what I care about.",
  "MyDetails": {
    "size.$": "$.product.details.size",
    "exists.$": "$.product.availability",
    "StaticValue": "foo"
  }
}
```

```
},
```

Given the previous input and the `Parameters` field, this is the JSON that is passed.

```
{
  "comment": "Selecting what I care about.",
  "MyDetails": {
    "size": "small",
    "exists": "in stock",
    "StaticValue": "foo"
  }
},
```

In addition to the input, you can access a special JSON object, known as the context object. The context object includes information about your state machine execution. See [Context object](#).

Connected resources

The `Parameters` field can also pass information to connected resources. For example, if your task state is orchestrating an AWS Batch job, you can pass the relevant API parameters directly to the API actions of that service. For more information, see:

- [Pass parameters to a service API](#)
- [Working with other services](#)

Amazon S3

If the Lambda function data you are passing between states might grow to more than 262,144 bytes, we recommend using Amazon S3 to store the data, and implement one of the following methods:

- Use the *Distributed Map state* in your workflow so that the Map state can read input directly from Amazon S3 data sources. For more information, see [Using Map state in Distributed mode](#).
- Parse the Amazon Resource Name (ARN) of the bucket in the `Payload` parameter to get the bucket name and key value. For more information, see [Use Amazon S3 ARNs instead of passing large payloads](#).

Alternatively, you can adjust your implementation to pass smaller payloads in your executions.

ResultSelector

Use the `ResultSelector` field to manipulate a state's result before `ResultPath` is applied. The `ResultSelector` field lets you create a collection of key value pairs, where the values are static or selected from the state's result. Using the `ResultSelector` field, you can choose what parts of a state's result you want to pass to the `ResultPath` field.

Note

With the `ResultPath` field, you can add the output of the `ResultSelector` field to the original input.

`ResultSelector` is an optional field in the following states:

- [Map](#)
- [Task state](#)
- [Parallel](#)

For example, Step Functions service integrations return metadata in addition to the payload in the result. `ResultSelector` can select portions of the result and merge them with the state input with `ResultPath`. In this example, we want to select just the `resourceType` and `ClusterId`, and merge that with the state input from an Amazon EMR `createCluster.sync`. Given the following:

```
{
  "resourceType": "elasticmapreduce",
  "resource": "createCluster.sync",
  "output": {
    "SdkHttpMetadata": {
      "HttpHeaders": {
        "Content-Length": "1112",
        "Content-Type": "application/x-amz-JSON-1.1",
        "Date": "Mon, 25 Nov 2019 19:41:29 GMT",
        "x-amzn-RequestId": "1234-5678-9012"
      },
      "HttpStatusCode": 200
    },
    "SdkResponseMetadata": {
      "RequestId": "1234-5678-9012"
    }
  },
}
```

```

    "ClusterId": "AKIAIOSFODNN7EXAMPLE"
  }
}

```

You can then select the `resourceType` and `ClusterId` using `ResultSelector`:

```

"Create Cluster": {
  "Type": "Task",
  "Resource": "arn:aws:states:::elasticmapreduce:createCluster.sync",
  "Parameters": {
    <some parameters>
  },
  "ResultSelector": {
    "ClusterId.$": "$.output.ClusterId",
    "ResourceType.$": "$.resourceType"
  },
  "ResultPath": "$.EMROutput",
  "Next": "Next Step"
}

```

With the given input, using `ResultSelector` produces:

```

{
  "OtherDataFromInput": {},
  "EMROutput": {
    "ResourceType": "elasticmapreduce",
    "ClusterId": "AKIAIOSFODNN7EXAMPLE"
  }
}

```

Flattening an array of arrays

If the [Parallel](#) or [Map](#) state in your state machines return an array of arrays, you can transform them into a flat array with the [ResultSelector](#) field. You can include this field inside the `Parallel` or `Map` state definition to manipulate the result of these states.

To flatten arrays, use the [JMESPath syntax \[* \]](#) in the `ResultSelector` field as shown in the following example.

```

"ResultSelector": {
  "flattenArray.$": "$[*][*]"
}

```

For examples that show how to flatten an array, see *Step 3* in the following tutorials:

- [Processing entire batch of data with a Lambda function](#)
- [Processing individual data items with a Lambda function](#)

ResultPath

The output of a state can be a copy of its input, the result it produces (for example, output from a Task state's Lambda function), or a combination of its input and result. Use `ResultPath` to control which combination of these is passed to the state output.

The following state types can generate a result and can include `ResultPath`:

- [Pass](#)
- [Task state](#)
- [Parallel](#)
- [Map](#)

Use `ResultPath` to combine a task result with task input, or to select one of these. The path you provide to `ResultPath` controls what information passes to the output.

Note

`ResultPath` is limited to using [reference paths](#), which limit scope so that it can identify only a single node in JSON. See [Reference Paths](#) in the [Amazon States Language](#).

These examples are based on the state machine and Lambda function described in the [Creating a Step Functions state machine that uses Lambda](#) tutorial. Work through that tutorial and test different outputs by trying various paths in a `ResultPath` field.

Use `ResultPath` to:

- [Use `ResultPath` to Replace the Input with the Result](#)
- [Discard the Result and Keep the Original Input](#)
- [Use `ResultPath` to Include the Result with the Input](#)
- [Use `ResultPath` to Update a Node in the Input with the Result](#)

- [Use ResultPath to Include Both Error and Input in a Catch](#)

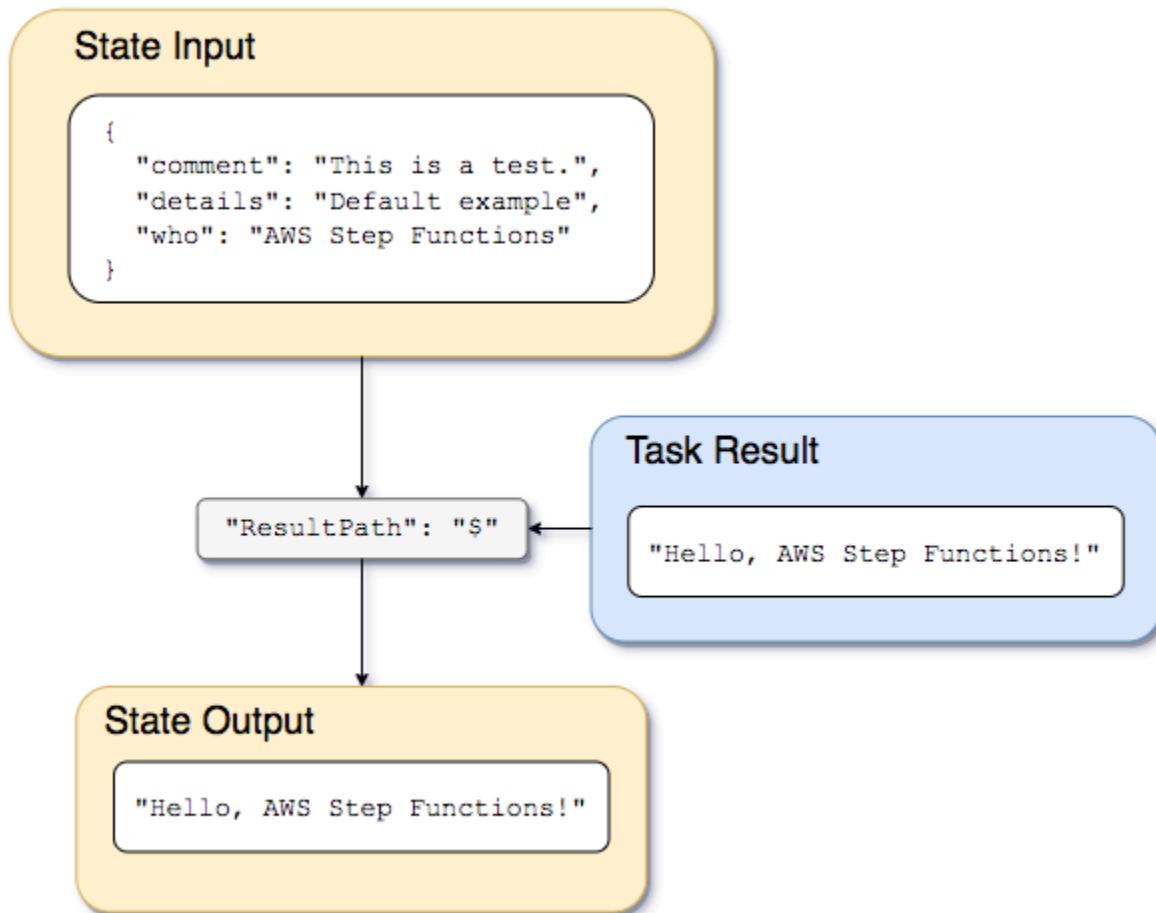
Tip

Use the [data flow simulator in the Step Functions console](#) to test JSON path syntax, to better understand how data is manipulated within a state, and to see how data is passed between states.

Use ResultPath to Replace the Input with the Result

If you don't specify a `ResultPath`, the default behavior is as if you had specified `"ResultPath": "$"`. Because this tells the state to replace the entire input with the result, the state input is completely replaced by the result coming from the task result.

The following diagram shows how `ResultPath` can completely replace the input with the result of the task.



Use the state machine and Lambda function described in [Creating a Step Functions state machine that uses Lambda](#), and change the service integration type to [AWS SDK integration](#) for the Lambda function. To do this, specify the Lambda function Amazon Resource Name (ARN) in the Resource field of the Task state as shown in the following example. Using AWS SDK integration ensures that the Task state result only contains the Lambda function output without any metadata.

```
{  "StartAt": "CallFunction",  "States": {    "CallFunction": {      "Type": "Task",      "Resource": "arn:aws:lambda:us-east-2:123456789012:function:HelloFunction",      "End": true    }  }}
```

```
}
```

Then, pass the following input:

```
{  
  "comment": "This is a test of the input and output of a Task state.",  
  "details": "Default example",  
  "who": "AWS Step Functions"  
}
```

The Lambda function provides the following result.

```
"Hello, AWS Step Functions!"
```

Tip

You can view this result on the [Step Functions console](#). To do this, on the [Execution Details](#) page of the console, choose the Lambda function in the **Graph view**. Then, choose the **Output** tab in the [Step details](#) pane to see this result.

If `ResultPath` isn't specified in the state, or if `"ResultPath": "$` is set, the input of the state is replaced by the result of the Lambda function, and the output of the state is the following.

```
"Hello, AWS Step Functions!"
```

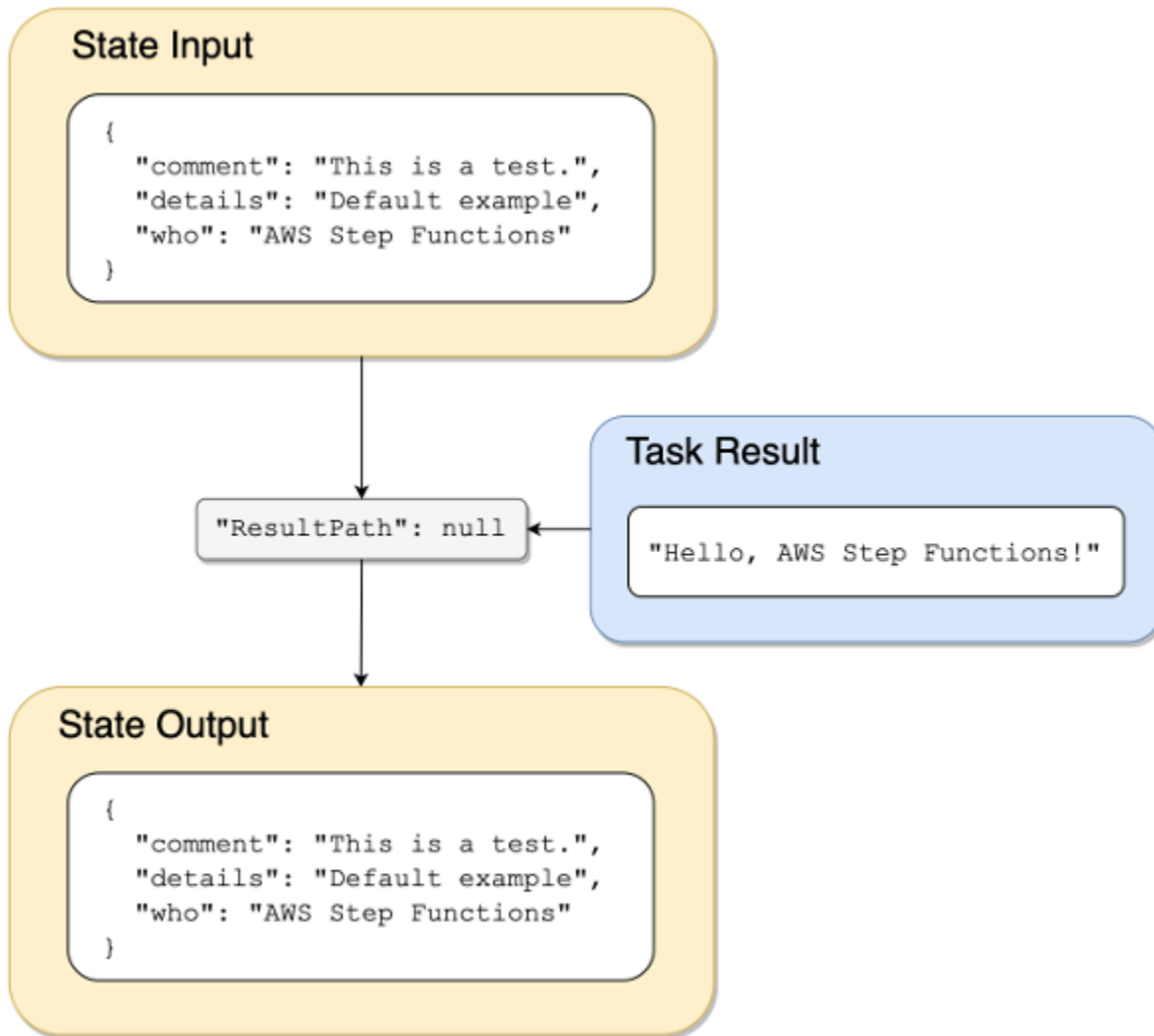
Note

`ResultPath` is used to include content from the result with the input, before passing it to the output. But, if `ResultPath` isn't specified, the default is to replace the entire input.

Discard the Result and Keep the Original Input

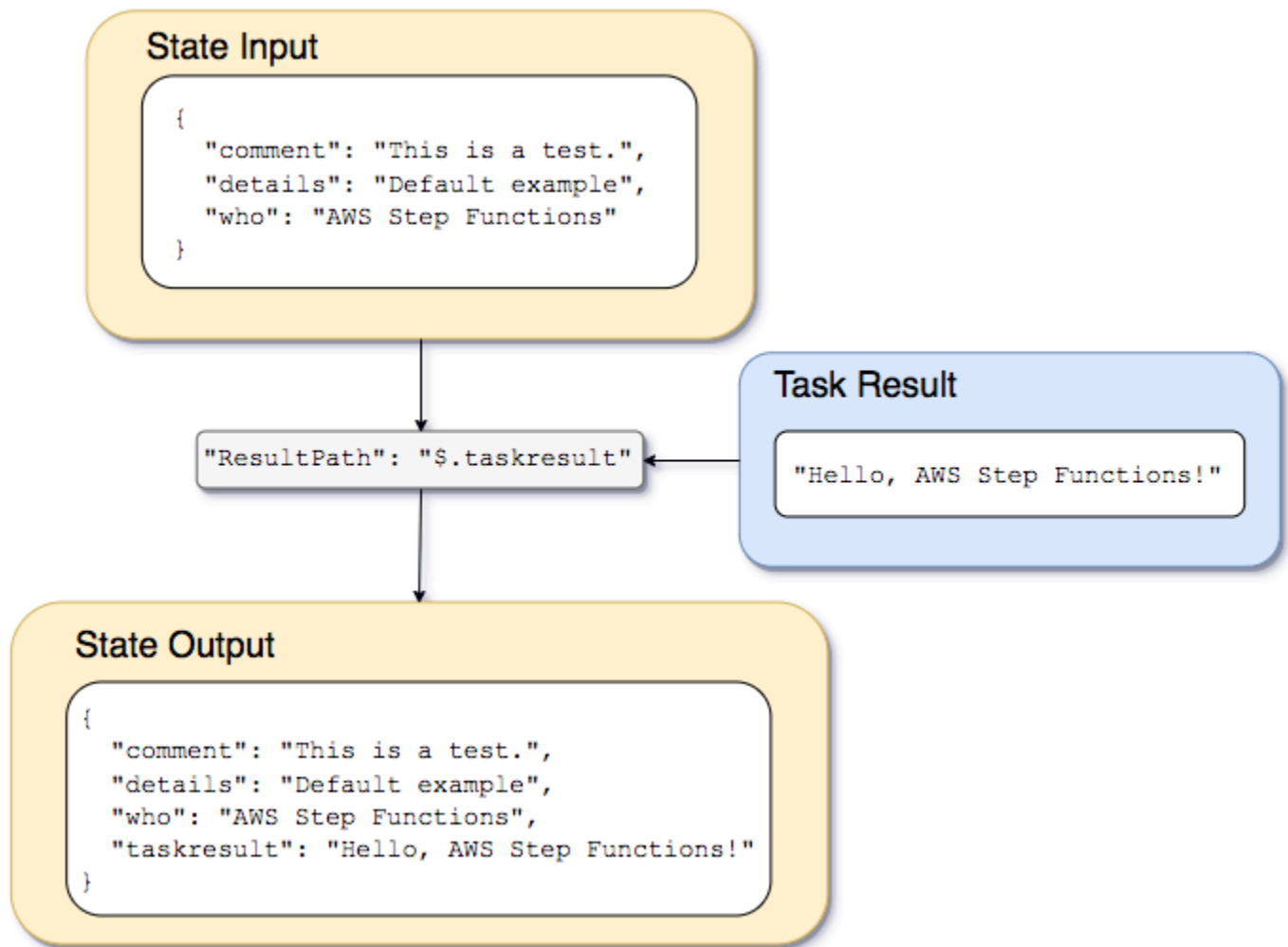
If you set `ResultPath` to `null`, it will pass the original input to the output. Using `"ResultPath": null`, the state's input payload will be copied directly to the output, with no regard for the result.

The following diagram shows how a null `ResultPath` will copy the input directly to the output.



Use `ResultPath` to Include the Result with the Input

The following diagram shows how `ResultPath` can include the result with the input.



Using the state machine and Lambda function described in the [Creating a Step Functions state machine that uses Lambda](#) tutorial, we could pass the following input.

```
{  "comment": "This is a test of the input and output of a Task state.",  "details": "Default example",  "who": "AWS Step Functions"}
```

The result of the Lambda function is the following.

```
"Hello, AWS Step Functions!"
```


To preserve the input, insert the result of the Lambda function, and then pass the combined JSON to the next state, we could set `ResultPath` to the following.

```
"ResultPath": "$.taskresult"
```

This includes the result of the Lambda function with the original input.

```
{
  "comment": "This is a test of input and output of a Task state.",
  "details": "Default behavior example",
  "who": "AWS Step Functions",
  "taskresult": "Hello, AWS Step Functions!"
}
```

The output of the Lambda function is appended to the original input as a value for `taskresult`. The input, including the newly inserted value, is passed to the next state.

You can also insert the result into a child node of the input. Set the `ResultPath` to the following.

```
"ResultPath": "$.strings.lambdaresult"
```

Start an execution using the following input.

```
{
  "comment": "An input comment.",
  "strings": {
    "string1": "foo",
    "string2": "bar",
    "string3": "baz"
  },
  "who": "AWS Step Functions"
}
```

The result of the Lambda function is inserted as a child of the `strings` node in the input.

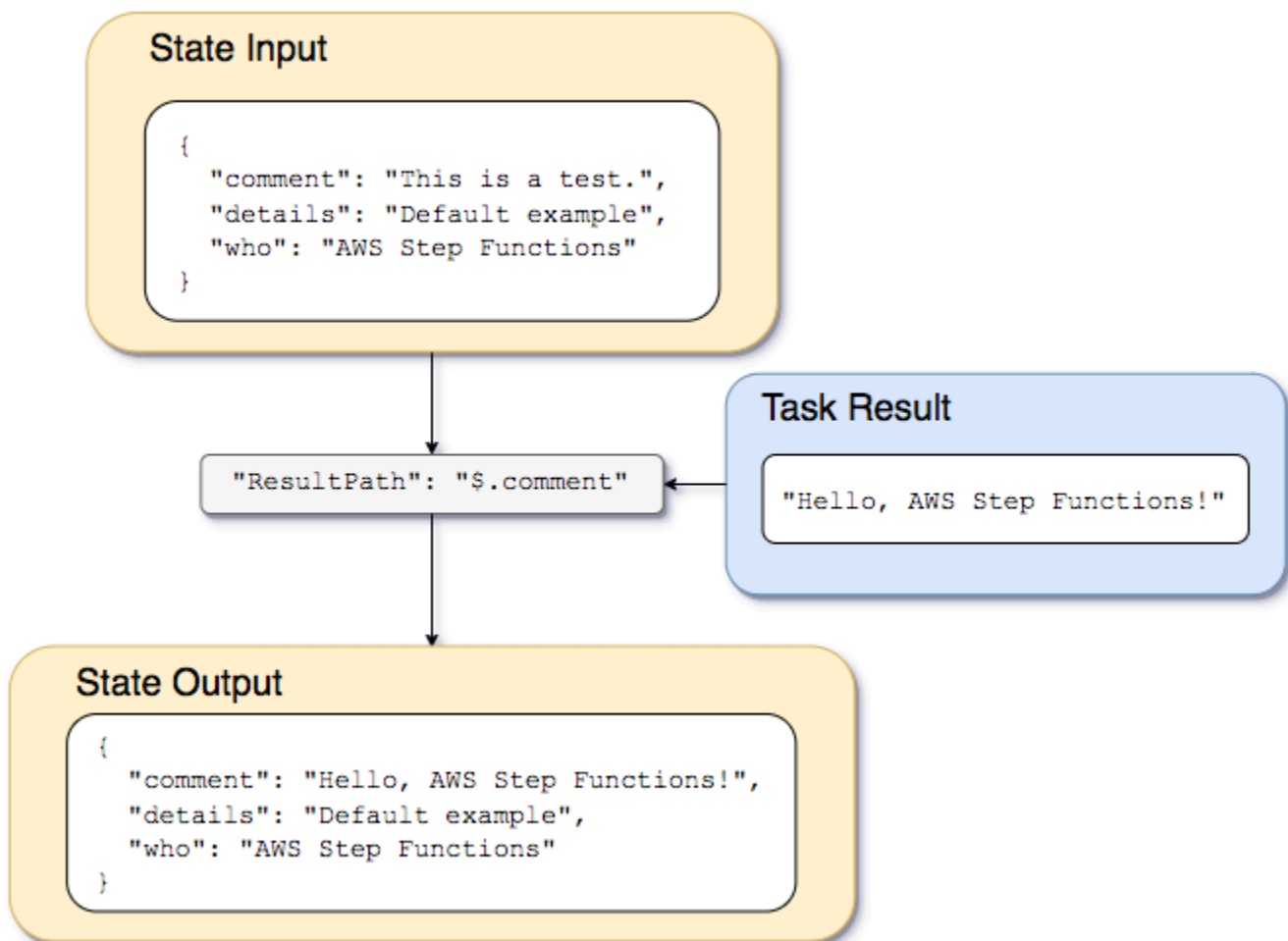
```
{
  "comment": "An input comment.",
  "strings": {
    "string1": "foo",
    "string2": "bar",
    "string3": "baz",
```

```
"lambdaresult": "Hello, AWS Step Functions!"
},
"who": "AWS Step Functions"
}
```

The state output now includes the original input JSON with the result as a child node.

Use ResultPath to Update a Node in the Input with the Result

The following diagram shows how ResultPath can update the value of existing JSON nodes in the input with values from the task result.



Using the example of the state machine and Lambda function described in the [Creating a Step Functions state machine that uses Lambda](#) tutorial, we could pass the following input.

```
{
```

```
"comment": "This is a test of the input and output of a Task state.",
"details": "Default example",
"who": "AWS Step Functions"
}
```

The result of the Lambda function is the following.

```
Hello, AWS Step Functions!
```

Instead of preserving the input and inserting the result as a new node in the JSON, we can overwrite an existing node.

For example, just as omitting or setting `"ResultPath": "$"` overwrites the entire node, you can specify an individual node to overwrite with the result.

```
"ResultPath": "$.comment"
```

Because the comment node already exists in the state input, setting `ResultPath` to `$.comment` replaces that node in the input with the result of the Lambda function. Without further filtering by `OutputPath`, the following is passed to the output.

```
{
  "comment": "Hello, AWS Step Functions!",
  "details": "Default behavior example",
  "who": "AWS Step Functions",
}
```

The value for the comment node, "This is a test of the input and output of a Task state.", is replaced by the result of the Lambda function: "Hello, AWS Step Functions!" in the state output.

Use ResultPath to Include Both Error and Input in a Catch

The [Handling error conditions using a Step Functions state machine](#) tutorial shows how to use a state machine to catch an error. In some cases, you might want to preserve the original input with the error. Use `ResultPath` in a `Catch` to include the error with the original input, instead of replacing it.

```
"Catch": [{
```

```
"ErrorEquals": ["States.ALL"],
"Next": "NextTask",
"ResultPath": "$.error"
}]
```

If the previous `Catch` statement catches an error, it includes the result in an `error` node within the state input. For example, with the following input:

```
{"foo": "bar"}
```

The state output when catching an error is the following.

```
{
  "foo": "bar",
  "error": {
    "Error": "Error here"
  }
}
```

For more information about error handling, see the following:

- [Error handling in Step Functions](#)
- [Handling error conditions using a Step Functions state machine](#)

OutputPath

`OutputPath` enables you to select a portion of the state output to pass to the next state. This enables you to filter out unwanted information, and pass only the portion of JSON that you care about.

If you don't specify an `OutputPath` the default value is `$`. This passes the entire JSON node (determined by the state input, the task result, and `ResultPath`) to the next state.

Tip

Use the [data flow simulator in the Step Functions console](#) to test JSON path syntax, to better understand how data is manipulated within a state, and to see how data is passed between states.

For more information, see the following:

- [Paths in the Amazon States Language](#)
- [InputPath, ResultPath, and OutputPath Examples](#)
- [Pass static JSON as parameters](#)
- [Input and Output Processing in Step Functions](#)

InputPath, ResultPath, and OutputPath Examples

Any state other than a [Fail](#) state or a [Succeed](#) state can include the input and output processing fields, such as `InputPath`, `ResultPath`, or `OutputPath`. Additionally, the [Wait](#) and [Choice](#) states don't support the `ResultPath` field. With these fields, you can use a [JsonPath](#) to filter the JSON data as it moves through your workflow.

You can also use the `Parameters` field to manipulate the JSON data as it moves through your workflow. For information about using `Parameters`, see [InputPath, Parameters and ResultSelector](#).

For example, start with the AWS Lambda function and state machine described in the [Creating a Step Functions state machine that uses Lambda](#) tutorial. Modify the state machine so that it includes the following `InputPath`, `ResultPath`, and `OutputPath`.

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloFunction",
      "InputPath": "$.lambda",
      "ResultPath": "$.data.lambdaresult",
      "OutputPath": "$.data",
      "End": true
    }
  }
}
```

Start an execution using the following input.

```
{
  "comment": "An input comment.",
  "data": {
    "val1": 23,
    "val2": 17
  },
  "extra": "foo",
  "lambda": {
    "who": "AWS Step Functions"
  }
}
```

Assume that the `comment` and `extra` nodes can be discarded, but that we want to include the output of the Lambda function, and preserve the information in the `data` node.

In the updated state machine, the `Task` state is altered to process the input to the task.

```
"InputPath": "$.lambda",
```

This line in the state machine definition limits the task input to only the `lambda` node from the state input. The Lambda function receives only the JSON object `{"who": "AWS Step Functions"}` as input.

```
"ResultPath": "$.data.lambdaresult",
```

This `ResultPath` tells the state machine to insert the result of the Lambda function into a node named `lambdaresult`, as a child of the `data` node in the original state machine input. Because we aren't performing any other manipulation on the original input and the result using `OutputPath`, the output of the state now includes the result of the Lambda function with the original input.

```
{
  "comment": "An input comment.",
  "data": {
    "val1": 23,
    "val2": 17,
    "lambdaresult": "Hello, AWS Step Functions!"
  },
  "extra": "foo",
```

```
"lambda": {
  "who": "AWS Step Functions"
}
```

But, our goal was to preserve only the data node, and include the result of the Lambda function. `OutputPath` filters this combined JSON before passing it to the state output.

```
"OutputPath": "$.data",
```

This selects only the data node from the original input (including the `lambdaresult` child inserted by `ResultPath`) to be passed to the output. The state output is filtered to the following.

```
{
  "val1": 23,
  "val2": 17,
  "lambdaresult": "Hello, AWS Step Functions!"
}
```

In this Task state:

1. `InputPath` sends only the `lambda` node from the input to the Lambda function.
2. `ResultPath` inserts the result as a child of the `data` node in the original input.
3. `OutputPath` filters the state input (which now includes the result of the Lambda function) so that it passes only the data node to the state output.

Example to manipulate original state machine input, result, and final output using JsonPath

Consider the following state machine that verifies an insurance applicant's identity and address.

Note

To view the complete example, see [How to use JSON Path in Step Functions](#).

```
{
  "Comment": "Sample state machine to verify an applicant's ID and address",
  "StartAt": "Verify info",
  "States": {
```

```

"Verify info": {
  "Type": "Parallel",
  "End": true,
  "Branches": [
    {
      "StartAt": "Verify identity",
      "States": {
        "Verify identity": {
          "Type": "Task",
          "Resource": "arn:aws:states:::lambda:invoke",
          "Parameters": {
            "Payload.$": "$",
            "FunctionName": "arn:aws:lambda:us-east-2:111122223333:function:check-
identity:$LATEST"
          },
          "End": true
        }
      }
    },
    {
      "StartAt": "Verify address",
      "States": {
        "Verify address": {
          "Type": "Task",
          "Resource": "arn:aws:states:::lambda:invoke",
          "Parameters": {
            "Payload.$": "$",
            "FunctionName": "arn:aws:lambda:us-east-2:111122223333:function:check-
address:$LATEST"
          },
          "End": true
        }
      }
    }
  ]
}
}
}

```

If you run this state machine using the following input, the execution fails because the Lambda functions that perform verification only expect the data that needs to be verified as input. Therefore, you must specify the nodes that contain the information to be verified using an appropriate `JsonPath`.


```
{
  "data": {
    "firstname": "Jane",
    "lastname": "Doe",
    "identity": {
      "email": "jdoe@example.com",
      "ssn": "123-45-6789"
    },
    "address": {
      "street": "123 Main St",
      "city": "Columbus",
      "state": "OH",
      "zip": "43219"
    },
    "interests": [
      {
        "category": "home",
        "type": "own",
        "yearBuilt": 2004
      },
      {
        "category": "boat",
        "type": "snowmobile",
        "yearBuilt": 2020
      },
      {
        "category": "auto",
        "type": "RV",
        "yearBuilt": 2015
      }
    ]
  }
}
```

To specify the node that the *check-identity* Lambda function must use, use the `InputPath` field as follows:

```
"InputPath": "$.data.identity"
```

And to specify the node that the *check-address* Lambda function must use, use the `InputPath` field as follows:

```
"InputPath": "$.data.address"
```

Now if you want to store the verification result within the original state machine input, use the `ResultPath` field as follows:

```
"ResultPath": "$.results"
```

However, if you only need the identity and verification results and discard the original input, use the `OutputPath` field as follows:

```
"OutputPath": "$.results"
```

For more information, see [Input and Output Processing in Step Functions](#).

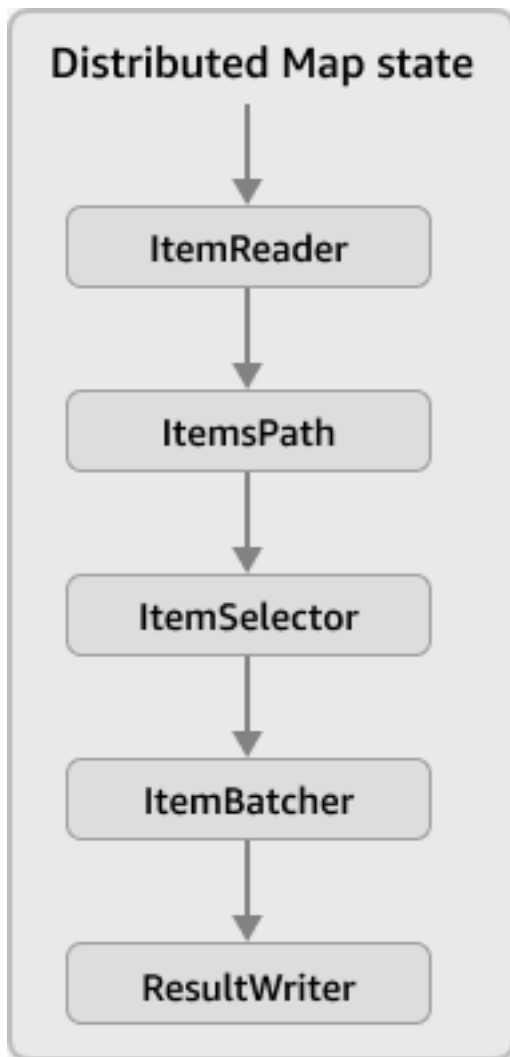
Map state input and output fields

Map states concurrently iterate over a collection of items in a dataset, such as a JSON array, a list of Amazon S3 objects, or the rows of a CSV file in an Amazon S3 bucket. It repeats a set of steps for each item in the collection. You can configure the input that the Map state receives and the output it generates using these fields. Step Functions applies each field in your *Distributed Map state* in the order shown in the following list and illustration:

Note

Based on your use case, you may not need to apply all of these fields.

1. [ItemReader](#)
2. [ItemsPath](#)
3. [ItemSelector](#)
4. [ItemBatcher](#)
5. [ResultWriter](#)

**Note**

These Map state input and output fields are currently unavailable in the [data flow simulator in the Step Functions console](#).

ItemReader

The `ItemReader` field is a JSON object, which specifies a dataset and its location. A *Distributed Map state* uses this dataset as its input. The following example shows the syntax of the `ItemReader` field if your dataset is a CSV file that's stored in an Amazon S3 bucket.

```
"ItemReader": {  
  "ReaderConfig": {
```

```
    "InputType": "CSV",
    "CSVHeaderLocation": "FIRST_ROW"
  },
  "Resource": "arn:aws:states:::s3:getObject",
  "Parameters": {
    "Bucket": "myBucket",
    "Key": "csvDataset/ratings.csv"
  }
}
```

Tip

In Workflow Studio, you specify the dataset and its location in the **Item source** field.

Contents

- [Contents of the ItemReader field](#)
- [Examples of datasets](#)
- [IAM policies for datasets](#)

Contents of the ItemReader field

Depending on your dataset, the contents of the `ItemReader` field varies. For example, if your dataset is a JSON array passed from a previous step in the workflow, the `ItemReader` field is omitted. If your dataset is an Amazon S3 data source, this field contains the following sub-fields.

ReaderConfig

A JSON object that specifies the following details:

- `InputType`

Specifies the type of Amazon S3 data source, such as CSV file, object, JSON file, or an Amazon S3 inventory list. In Workflow Studio, you can select an input type from the **Amazon S3 item source** dropdown list under the **Item source** field.

- `CSVHeaderLocation`

Note

You must specify this field only if you use a CSV file as dataset.

Accepts one of the following values to specify the location of the column header:

Important

Currently, Step Functions supports CSV headers of up to 10 KB.

- **FIRST_ROW** – Use this option if the first line of the file is the header.
- **GIVEN** – Use this option to specify the header within the state machine definition. For example, if your CSV file contains the following data.

```
1,307,3.5,1256677221
1,481,3.5,1256677456
1,1091,1.5,1256677471
...
```

Provide the following JSON array as a CSV header.

```
"ItemReader": {
  "ReaderConfig": {
    "InputType": "CSV",
    "CSVHeaderLocation": "GIVEN",
    "CSVHeaders": [
      "userId",
      "movieId",
      "rating",
      "timestamp"
    ]
  }
}
```

Tip

In Workflow Studio, you can find this option under **Additional configuration** in the **Item source** field.

- **MaxItems**

Limits the number of data items passed to the Map state. For example, suppose that you provide a CSV file that contains 1000 rows and specify a limit of 100. Then, the interpreter passes *only* 100 rows to the Map state. The Map state processes items in sequential order, starting after the header row.

By default, the Map state iterates over all the items in the specified dataset.

Note

Currently, you can specify a limit of up to 100,000,000. The *Distributed Map state* stops reading items beyond this limit.

Tip

In Workflow Studio, you can find this option under **Additional configuration** in the **Item source** field.

Alternatively, you can specify a [reference path](#) to an existing key-value pair in your *Distributed Map state* input. This path must resolve to a positive integer. You specify the reference path in the `MaxItemsPath` sub-field.

Important

You can specify either the `MaxItems` or the `MaxItemsPath` sub-field, but not both.

Resource

The Amazon S3 API action that Step Functions must invoke depending on the specified dataset.

Parameters

A JSON object that specifies the Amazon S3 bucket name and object key that the dataset is stored in.

Important

Make sure that your Amazon S3 buckets are under the same AWS account and AWS Region as your state machine.

Examples of datasets

You can specify one of the following options as your dataset:

- [JSON array from a previous step](#)
- [A list of Amazon S3 objects](#)
- [JSON file in an Amazon S3 bucket](#)
- [CSV file in an Amazon S3 bucket](#)
- [Amazon S3 inventory list](#)

Important

Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see [IAM policies for datasets](#).

JSON array from a previous step

A *Distributed Map state* can accept a JSON input passed from a previous step in the workflow. This input must either be an array, or must contain an array within a specific node. To select a node that contains the array, you can use the [ItemsPath](#) field.

To process individual items in the array, the *Distributed Map state* starts a child workflow execution for each array item. The following tabs show examples of the input passed to the Map state and the corresponding input to a child workflow execution.

Note

Step Functions omits the `ItemReader` field when your dataset is a JSON array from a previous step.

Input passed to the Map state

Consider the following JSON array of three items.

```
"facts": [  
  {  
    "verdict": "true",  
    "statement_date": "6/11/2008",  
    "statement_source": "speech"  
  },  
  {  
    "verdict": "false",  
    "statement_date": "6/7/2022",  
    "statement_source": "television"  
  },  
  {  
    "verdict": "mostly-true",  
    "statement_date": "5/18/2016",  
    "statement_source": "news"  
  }  
]
```

Input passed to a child workflow execution

The *Distributed Map state* starts three child workflow executions. Each execution receives an array item as input. The following example shows the input received by a child workflow execution.

```
{  
  "verdict": "true",  
  "statement_date": "6/11/2008",  
  "statement_source": "speech"  
}
```


Amazon S3 objects example

A *Distributed Map state* can iterate over the objects that are stored in an Amazon S3 bucket. When the workflow execution reaches the Map state, Step Functions invokes the [ListObjectsV2](#) API action, which returns an array of the Amazon S3 object metadata. In this array, each item contains data, such as **ETag** and **Key**, for the data stored in the bucket.

To process individual items in the array, the *Distributed Map state* starts a child workflow execution. For example, suppose that your Amazon S3 bucket contains 100 images. Then, the array returned after invoking the ListObjectsV2 API action contains 100 items. The *Distributed Map state* then starts 100 child workflow executions to process each array item.

Note

- Currently, Step Functions also includes an item for each folder you create in a specific Amazon S3 bucket using the Amazon S3 console. This results in an extra child workflow execution started by the *Distributed Map state*. To avoid creating an extra child workflow execution for the folder, we recommend that you use the AWS CLI to create folders. For more information, see [High-level Amazon S3 commands](#) in the *AWS Command Line Interface User Guide*.
- Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see [IAM policies for datasets](#).

The following tabs show examples of the `ItemReader` field syntax and the input passed to a child workflow execution for this dataset.

ItemReader syntax

In this example, you've organized your data, which includes images, JSON files, and objects, within a prefix named `processData` in an Amazon S3 bucket named `myBucket`.

```
"ItemReader": {
  "Resource": "arn:aws:states:::s3:listObjectsV2",
  "Parameters": {
    "Bucket": "myBucket",
    "Prefix": "processData"
  }
}
```

Input passed to a child workflow execution

The *Distributed Map state* starts as many child workflow executions as the number of items present in the Amazon S3 bucket. The following example shows the input received by a child workflow execution.

```
{
  "Etag": "\"05704fbdccb224cb01c59005bebbad28\"",
  "Key": "processData/images/n02085620_1073.jpg",
  "LastModified": 1668699881,
  "Size": 34910,
  "StorageClass": "STANDARD"
}
```

JSON file in an Amazon S3 bucket

A *Distributed Map state* can accept a JSON file that's stored in an Amazon S3 bucket as a dataset. The JSON file must contain an array.

When the workflow execution reaches the Map state, Step Functions invokes the [GetObject](#) API action to fetch the specified JSON file. The Map state then iterates over each item in the array and starts a child workflow execution for each item. For example, if your JSON file contains 1000 array items, the Map state starts 1000 child workflow executions.

Note

- The execution input used to start a child workflow execution can't exceed 256 KB. However, Step Functions supports reading an item of up to 8 MB from a CSV or JSON file if you then apply the optional `ItemSelector` field to reduce the item's size.
- Currently, Step Functions supports 10 GB as the maximum size of an individual file in an Amazon S3 inventory report. However, Step Functions can process more than 10 GB if each individual file is under 10 GB.
- Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see [IAM policies for datasets](#).

The following tabs show examples of the `ItemReader` field syntax and the input passed to a child workflow execution for this dataset.

For this example, imagine you have a JSON file named *factcheck.json*. You've stored this file within a prefix named *jsonDataset* in an Amazon S3 bucket. The following is an example of the JSON dataset.

```
[
  {
    "verdict": "true",
    "statement_date": "6/11/2008",
    "statement_source": "speech"
  },
  {
    "verdict": "false",
    "statement_date": "6/7/2022",
    "statement_source": "television"
  },
  {
    "verdict": "mostly-true",
    "statement_date": "5/18/2016",
    "statement_source": "news"
  },
  ...
]
```

ItemReader syntax

```
"ItemReader": {
  "Resource": "arn:aws:states:::s3:getObject",
  "ReaderConfig": {
    "InputType": "JSON"
  },
  "Parameters": {
    "Bucket": "myBucket",
    "Key": "jsonDataset/factcheck.json"
  }
}
```

Input to a child workflow execution

The *Distributed Map state* starts as many child workflow executions as the number of array items present in the JSON file. The following example shows the input received by a child workflow execution.

```
{
  "verdict": "true",
  "statement_date": "6/11/2008",
  "statement_source": "speech"
}
```

CSV file in an Amazon S3 bucket

A *Distributed Map state* can accept a CSV file that's stored in an Amazon S3 bucket as a dataset. If you use a CSV file as your dataset, you need to specify a CSV column header. For information about how to specify a CSV header, see [Contents of the ItemReader field](#).

Because there isn't a standardized format to create and maintain data in CSV files, Step Functions parses CSV files based on the following rules:

- Commas (,) are a delimiter that separates individual fields.
- Newlines are a delimiter that separates individual records.
- Fields are treated as strings. For data type conversions, use the [States.StringToJson](#) intrinsic function in [ItemSelector](#).
- Double quotation marks (" ") aren't required to enclose strings. However, strings that are enclosed by double quotation marks can contain commas and newlines without them functioning as delimiters.
- Escape double quotes by repeating them.
- If the number of fields in a row is less than the number of fields in the header, Step Functions provides empty strings for the missing values.
- If the number of fields in a row is more than the number of fields in the header, Step Functions skips the additional fields.

For more information about how Step Functions parses a CSV file, see [Example of parsing an input CSV file](#).

When the workflow execution reaches the Map state, Step Functions invokes the [GetObject](#) API action to fetch the specified CSV file. The Map state then iterates over each row in the CSV file and starts a child workflow execution to process the items in each row. For example, suppose that you provide a CSV file that contains 100 rows as input. Then, the interpreter passes each row to the Map state. The Map state processes items in serial order, starting after the header row.

Note

- The execution input used to start a child workflow execution can't exceed 256 KB. However, Step Functions supports reading an item of up to 8 MB from a CSV or JSON file if you then apply the optional `ItemSelector` field to reduce the item's size.
- Currently, Step Functions supports 10 GB as the maximum size of an individual file in an Amazon S3 inventory report. However, Step Functions can process more than 10 GB if each individual file is under 10 GB.
- Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see [IAM policies for datasets](#).

The following tabs show examples of the `ItemReader` field syntax and the input passed to a child workflow execution for this dataset.

ItemReader syntax

For example, say that you have a CSV file named *ratings.csv*. Then, you've stored this file within a prefix that's named *csvDataset* in an Amazon S3 bucket.

```
{
  "ItemReader": {
    "ReaderConfig": {
      "InputType": "CSV",
      "CSVHeaderLocation": "FIRST_ROW"
    },
    "Resource": "arn:aws:states:::s3:getObject",
    "Parameters": {
      "Bucket": "myBucket",
      "Key": "csvDataset/ratings.csv"
    }
  }
}
```

Input to a child workflow execution

The *Distributed Map state* starts as many child workflow executions as the number of rows present in the CSV file, excluding the header row, if in the file. The following example shows the input received by a child workflow execution.

```
{
  "rating": "3.5",
  "movieId": "307",
  "userId": "1",
  "timestamp": "1256677221"
}
```

S3 inventory example

A *Distributed Map state* can accept an Amazon S3 inventory manifest file that's stored in an Amazon S3 bucket as a dataset.

When the workflow execution reaches the Map state, Step Functions invokes the [GetObject](#) API action to fetch the specified Amazon S3 inventory manifest file. The Map state then iterates over the objects in the inventory to return an array of Amazon S3 inventory object metadata.

Note

- Currently, Step Functions supports 10 GB as the maximum size of an individual file in an Amazon S3 inventory report. However, Step Functions can process more than 10 GB if each individual file is under 10 GB.
- Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see [IAM policies for datasets](#).

The following is an example of an inventory file in CSV format. This file includes the objects named `csvDataset` and `imageDataset`, which are stored in an Amazon S3 bucket that's named `sourceBucket`.

```
"sourceBucket","csvDataset/","0","2022-11-16T00:27:19.000Z"
"sourceBucket","csvDataset/titles.csv","3399671","2022-11-16T00:29:32.000Z"
"sourceBucket","imageDataset/","0","2022-11-15T20:00:44.000Z"
"sourceBucket","imageDataset/n02085620_10074.jpg","27034","2022-11-15T20:02:16.000Z"
...
```

⚠ Important

Currently, Step Functions doesn't support user-defined Amazon S3 inventory report as a dataset. You must also make sure that the output format of your Amazon S3 inventory report is CSV. For more information about Amazon S3 inventories and how to set them up, see [Amazon S3 Inventory](#) in the *Amazon S3 User Guide*.

The following example of an inventory manifest file shows the CSV headers for the inventory object metadata.

```
{
  "sourceBucket" : "sourceBucket",
  "destinationBucket" : "arn:aws:s3:::inventory",
  "version" : "2016-11-30",
  "creationTimestamp" : "1668560400000",
  "fileFormat" : "CSV",
  "fileSchema" : "Bucket, Key, Size, LastModifiedDate",
  "files" : [ {
    "key" : "source-bucket/destination-prefix/
data/20e55de8-9c21-45d4-99b9-46c732000228.csv.gz",
    "size" : 7300,
    "MD5checksum" : "a7ff4a1d4164c3cd55851055ec8f6b20"
  } ]
}
```

The following tabs show examples of the `ItemReader` field syntax and the input passed to a child workflow execution for this dataset.

ItemReader syntax

```
{
  "ItemReader": {
    "ReaderConfig": {
      "InputType": "MANIFEST"
    },
    "Resource": "arn:aws:states:::s3:getObject",
    "Parameters": {
      "Bucket": "destinationBucket",
      "Key": "destination-prefix/source-bucket/config-ID/YYYY-MM-DDTHH-MMZ/
manifest.json"
    }
  }
}
```

```
    }  
  }  
}
```

Input to a child workflow execution

```
{  
  "LastModifiedDate": "2022-11-16T00:29:32.000Z",  
  "Bucket": "sourceBucket",  
  "Size": "3399671",  
  "Key": "csvDataset/titles.csv"  
}
```

Depending on the fields you selected while configuring the Amazon S3 inventory report, the contents of your `manifest.json` file may vary from the example shown.

IAM policies for datasets

When you create workflows with the Step Functions console, Step Functions can automatically generate IAM policies based on the resources in your workflow definition. These policies include the least privileges necessary to allow the state machine role to invoke the [StartExecution](#) API action for the *Distributed Map state*. These policies also include the least privileges necessary for Step Functions to access AWS resources, such as Amazon S3 buckets and objects and Lambda functions. We highly recommend that you include only those permissions that are necessary in your IAM policies. For example, if your workflow includes a Map state in Distributed mode, scope your policies down to the specific Amazon S3 bucket and folder that contains your dataset.

Important

If you specify an Amazon S3 bucket and object, or prefix, with a [reference path](#) to an existing key-value pair in your *Distributed Map state* input, make sure that you update the IAM policies for your workflow. Scope the policies down to the bucket and object names the path resolves to at runtime.

The following IAM policy examples grant the least privileges required to access your Amazon S3 datasets using the [ListObjectsV2](#) and [GetObject](#) API actions.

Example IAM policy for Amazon S3 object as dataset

The following example shows an IAM policy that grants the least privileges to access the objects organized within *processImages* in an Amazon S3 bucket named *myBucket*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::myBucket"
      ],
      "Condition": {
        "StringLike": {
          "s3:prefix": [
            "processImages"
          ]
        }
      }
    }
  ]
}
```

Example IAM policy for a CSV file as dataset

The following example shows an IAM policy that grants least privileges to access a CSV file named *ratings.csv*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::myBucket/csvDataset/ratings.csv"
      ]
    }
  ]
}
```

```

    }
  ]
}

```

Example IAM policy for an Amazon S3 inventory as dataset

The following example shows an IAM policy that grants least privileges to access an Amazon S3 inventory report.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::destination-prefix/source-bucket/config-ID/YYYY-MM-DDTHH-MMZ/manifest.json",
        "arn:aws:s3:::destination-prefix/source-bucket/config-ID/data/*"
      ]
    }
  ]
}

```

ItemsPath

Use the `ItemsPath` field to select an array within a JSON input provided to a Map state. The Map state repeats a set of steps for each item in the array. By default, the Map state sets `ItemsPath` to `$`, which selects the entire input. If the input to the Map state is a JSON array, it runs an iteration for each item in the array, passing that item to the iteration as input.

Note

You can use `ItemsPath` in the *Distributed Map state* only if you use a JSON input passed from a previous state in the workflow.

You can use the `ItemsPath` field to specify a location in the input that points to JSON array used for iterations. The value of `ItemsPath` must be a [Reference Path](#), and that path must point to

JSON array. For instance, consider input to a Map state that includes two arrays, like the following example.

```
{
  "ThingsPiratesSay": [
    {
      "say": "Avast!"
    },
    {
      "say": "Yar!"
    },
    {
      "say": "Walk the Plank!"
    }
  ],
  "ThingsGiantsSay": [
    {
      "say": "Fee!"
    },
    {
      "say": "Fi!"
    },
    {
      "say": "Fo!"
    },
    {
      "say": "Fum!"
    }
  ]
}
```

In this case, you could specify which array to use for Map state iterations by selecting it with `ItemsPath`. The following state machine definition specifies the `ThingsPiratesSay` array in the input using `ItemsPath`. It then runs an iteration of the `SayWord` pass state for each item in the `ThingsPiratesSay` array.

```
{
  "StartAt": "PiratesSay",
  "States": {
    "PiratesSay": {
      "Type": "Map",
      "ItemsPath": "$.ThingsPiratesSay",

```

```
    "ItemProcessor": {
      "StartAt": "SayWord",
      "States": {
        "SayWord": {
          "Type": "Pass",
          "End": true
        }
      }
    },
    "End": true
  }
}
```

When processing input, the Map state applies `ItemsPath` after [InputPath](#). It operates on the effective input to the state after `InputPath` filters the input.

For more information on Map states, see the following:

- [Map state](#)
- [Map state processing modes](#)
- [Repeat an action using Inline Map state](#)
- [Inline Map state input and output processing](#)

ItemSelector

By default, the effective input for the Map state is the set of individual data items present in the raw state input. The `ItemSelector` field lets you override the data items' values before they're passed on to the Map state. To override the values, specify a valid JSON input that contains a collection of key-value pairs. These pairs can be static values provided in your state machine definition, values selected from the state input using a [path](#), or values accessed from the [context object](#).

If you specify key-value pairs using a path or context object, the key name must end in `.$`.

Note

The `ItemSelector` field replaces the `Parameters` field within the `Map` state. If you use the `Parameters` field in your `Map` state definitions to create custom input, we highly recommend that you replace them with `ItemSelector`.

You can specify the `ItemSelector` field in both an *Inline Map state* and a *Distributed Map state*.

For example, consider the following JSON input that contains an array of three items within the `imageData` node. For each *Map state iteration*, an array item is passed to the iteration as input.

```
[
  {
    "resize": "true",
    "format": "jpg"
  },
  {
    "resize": "false",
    "format": "png"
  },
  {
    "resize": "true",
    "format": "jpg"
  }
]
```

Using the `ItemSelector` field, you can define a custom JSON input to override the original input as shown in the following example. Step Functions then passes this custom input to each *Map state iteration*. The custom input contains a static value for `size` and the value of a context object `data` for `Map` state. The `$$.Map .Item .Value` context object contains the value of each individual data item.

```
{
  "ItemSelector": {
    "size": 10,
    "value.$": "$$.Map.Item.Value"
  }
}
```

The following example shows the input received by one iteration of the *Inline Map state*:

```
{
  "size": 10,
  "value": {
    "resize": "true",
    "format": "jpg"
  }
}
```

ⓘ Tip

For a complete example of a *Distributed Map state* that uses the `ItemSelector` field, see [Getting started with using Distributed Map state](#).

ItemBatcher

The `ItemBatcher` field is a JSON object, which specifies to process a group of items in a single child workflow execution. Use batching when processing large CSV files or JSON arrays, or large sets of Amazon S3 objects.

The following example shows the syntax of the `ItemBatcher` field. In the following syntax, the maximum number of items that each child workflow execution should process is set to 100.

```
{
  "ItemBatcher": {
    "MaxItemsPerBatch": 100
  }
}
```

By default, each item in a dataset is passed as input to individual child workflow executions. For example, assume you specify a JSON file as input that contains the following array:

```
[
  {
    "verdict": "true",
    "statement_date": "6/11/2008",
    "statement_source": "speech"
  },
  {
```

```
    "verdict": "false",
    "statement_date": "6/7/2022",
    "statement_source": "television"
  },
  {
    "verdict": "true",
    "statement_date": "5/18/2016",
    "statement_source": "news"
  },
  ...
]
```

For the given input, each child workflow execution receives an array item as its input. The following example shows the input of a child workflow execution:

```
{
  "verdict": "true",
  "statement_date": "6/11/2008",
  "statement_source": "speech"
}
```

To help optimize the performance and cost of your processing job, select a batch size that balances the number of items against the items processing time. If you use batching, Step Functions adds the items to an **Items** array. It then passes the array as input to each child workflow execution. The following example shows a batch of two items passed as input to a child workflow execution:

```
{
  "Items": [
    {
      "verdict": "true",
      "statement_date": "6/11/2008",
      "statement_source": "speech"
    },
    {
      "verdict": "false",
      "statement_date": "6/7/2022",
      "statement_source": "television"
    }
  ]
}
```

Tip

To learn more about using the `ItemBatcher` field in your workflows, try the following tutorials and workshop:

- [Process an entire batch of data within a Lambda function](#)
- [Iterate over items in a batch inside child workflow executions](#)
- [Large-Scale Parallelization with Distributed Map](#) in *Module 14 - Data Processing of The AWS Step Functions Workshop*

Contents

- [Fields to specify item batching](#)

Fields to specify item batching

To batch items, specify the maximum number of items to batch, the maximum batch size, or both. You must specify one of these values to batch items.

Max items per batch

Specifies the maximum number of items that each child workflow execution processes. The interpreter limits the number of items batched in the `Items` array to this value. If you specify both a batch number and size, the interpreter reduces the number of items in a batch to avoid exceeding the specified batch size limit.

If you don't specify this value but provide a value for maximum batch size, Step Functions processes as many items as possible in each child workflow execution without exceeding the maximum batch size in bytes.

For example, imagine you run an execution with an input JSON file that contains 1130 nodes. If you specify a maximum items value for each batch of 100, Step Functions creates 12 batches. Of these, 11 batches contain 100 items each, while the twelfth batch contains the remaining 30 items.

Alternatively, you can specify the maximum items for each batch as a [reference path](#) to an existing key-value pair in your *Distributed Map state* input. This path must resolve to a positive integer.

For example, given the following input:

```
{
  "maxBatchItems": 500
}
```

You can specify the maximum number of items to batch using a reference path as follows:

```
{
  ...
  "Map": {
    "Type": "Map",
    "MaxConcurrency": 2000,
    "ItemBatcher": {
      "MaxItemsPerBatchPath": "$.maxBatchItems"
    }
  }
  ...
  ...
}
```

Important

You can specify either the `MaxItemsPerBatch` or the `MaxItemsPerBatchPath` sub-field, but not both.

Max KBs per batch

Specifies the maximum size of a batch in bytes, up to 256 KBs. If you specify both a maximum batch number and size, Step Functions reduces the number of items in a batch to avoid exceeding the specified batch size limit.

Alternatively, you can specify the maximum batch size as a [reference path](#) to an existing key-value pair in your *Distributed Map state* input. This path must resolve to a positive integer.

Note

If you use batching and don't specify a maximum batch size, the interpreter processes as many items it can process up to 256 KB in each child workflow execution.

For example, given the following input:

```
{
  "batchSize": 131072
}
```

You can specify the maximum batch size using a reference path as follows:

```
{
  ...
  "Map": {
    "Type": "Map",
    "MaxConcurrency": 2000,
    "ItemBatcher": {
      "MaxInputBytesPerBatchPath": "$.batchSize"
    }
    ...
    ...
  }
}
```

Important

You can specify either the `MaxInputBytesPerBatch` or the `MaxInputBytesPerBatchPath` sub-field, but not both.

Batch input

Optionally, you can also specify a fixed JSON input to include in each batch passed to each child workflow execution. Step Functions merges this input with the input for each individual child workflow executions. For example, given the following fixed input of a fact check date on an array of items:

```
"ItemBatcher": {
  "BatchInput": {
    "factCheck": "December 2022"
  }
}
```

Each child workflow execution receives the following as input:

```
{
  "BatchInput": {
    "factCheck": "December 2022"
  },
  "Items": [
    {
      "verdict": "true",
      "statement_date": "6/11/2008",
      "statement_source": "speech"
    },
    {
      "verdict": "false",
      "statement_date": "6/7/2022",
      "statement_source": "television"
    },
    ...
  ]
}
```

ResultWriter

The `ResultWriter` field is a JSON object that specifies the Amazon S3 location where Step Functions writes the results of the child workflow executions started by a *Distributed Map state*. By default, Step Functions doesn't export these results.

Important

Make sure that the Amazon S3 bucket you use to export the results of a Map Run is under the same AWS account and AWS Region as your state machine. Otherwise, your state machine execution will fail with the `States.ResultWriterFailed` error.

Exporting the results to an Amazon S3 bucket is helpful if your output payload size exceeds 256 KB. Step Functions consolidates all child workflow execution data, such as execution input and output, ARN, and execution status. It then exports executions with the same status to their respective files in the specified Amazon S3 location. The following example shows the syntax of the `ResultWriter` field if you export the child workflow execution results. In this example, you store the results in a bucket named `myOutputBucket` within a prefix called `csvProcessJobs`.

```
{
  "ResultWriter": {
    "Resource": "arn:aws:states:::s3:putObject",
    "Parameters": {
      "Bucket": "myOutputBucket",
      "Prefix": "csvProcessJobs"
    }
  }
}
```

ⓘ Tip

In Workflow Studio, you can export the child workflow execution results by selecting **Export Map state results to Amazon S3**. Then, provide the name of the Amazon S3 bucket and prefix where you want to export the results to.

Step Functions needs appropriate permissions to access the bucket and folder where you want to export the results. For information about the required IAM policy, see [IAM policies for ResultWriter](#).

If you export the child workflow execution results, the *Distributed Map state* execution returns the Map Run ARN and data about the Amazon S3 export location in the following format:

```
{
  "MapRunArn": "arn:aws:states:us-
east-2:123456789012:mapRun:csvProcess/Map:ad9b5f27-090b-3ac6-9beb-243cd77144a7",
  "ResultWriterDetails": {
    "Bucket": "myOutputBucket",
    "Key": "csvProcessJobs/ad9b5f27-090b-3ac6-9beb-243cd77144a7/manifest.json"
  }
}
```

Step Functions exports executions with the same status to their respective files. For example, if your child workflow executions resulted in 500 success and 200 failure results, Step Functions creates two files in the specified Amazon S3 location for the success and failure results. In this example, the success results file contains the 500 success results, while the failure results file contains the 200 failure results.

For a given execution attempt, Step Functions creates the following files in the specified Amazon S3 location depending on your execution output:

- `manifest.json` – Contains Map Run metadata, such as export location, Map Run ARN, and information about the result files.

If you've [redriven](#) a Map Run, the `manifest.json` file, contains references to all the successful child workflow executions across all the attempts of a Map Run. However, this file contains references to the failed and pending executions for a specific redrive.

- `SUCCEEDED_n.json` – Contains the consolidated data for all successful child workflow executions. *n* represents the index number of the file. The index number starts from 0. For example, `SUCCEEDED_1.json`.
- `FAILED_n.json` – Contains the consolidated data for all failed, timed out, and aborted child workflow executions. Use this file to recover from failed executions. *n* represents the index of the file. The index number starts from 0. For example, `FAILED_1.json`.
- `PENDING_n.json` – Contains the consolidated data for all child workflow executions that weren't started because the Map Run failed or aborted. *n* represents the index of the file. The index number starts from 0. For example, `PENDING_1.json`.

Step Functions supports individual result files of up to 5 GB. If a file size exceeds 5 GB, Step Functions creates another file to write the remaining execution results and appends an index number to the file name. For example, if size of the `Succeeded_0.json` file exceeds 5 GB, Step Functions creates `Succeeded_1.json` file to record the remaining results.

If you didn't specify to export the child workflow execution results, the state machine execution returns an array of child workflow execution results as shown in the following example:

Note

If the returned output size exceeds 256 KB, the state machine execution fails and returns a [States.DataLimitExceeded](#) error.

```
[
  {
    "statusCode": 200,
    "inputReceived": {
      "show_id": "s1",
      "release_year": "2020",
      "rating": "PG-13",
```

```
    "type": "Movie"
  }
},
{
  "statusCode": 200,
  "inputReceived": {
    "show_id": "s2",
    "release_year": "2021",
    "rating": "TV-MA",
    "type": "TV Show"
  }
},
...
]
```

IAM policies for ResultWriter

When you create workflows with the Step Functions console, Step Functions can automatically generate IAM policies based on the resources in your workflow definition. These policies include the least privileges necessary to allow the state machine role to invoke the [StartExecution](#) API action for the *Distributed Map state*. These policies also include the least privileges necessary Step Functions to access AWS resources, such as Amazon S3 buckets and objects and Lambda functions. We highly recommend that you include only those permissions that are necessary in your IAM policies. For example, if your workflow includes a Map state in Distributed mode, scope your policies down to the specific Amazon S3 bucket and folder that contains your dataset.

Important

If you specify an Amazon S3 bucket and object, or prefix, with a [reference path](#) to an existing key-value pair in your *Distributed Map state* input, make sure that you update the IAM policies for your workflow. Scope the policies down to the bucket and object names the path resolves to at runtime.

The following IAM policy example grants the least privileges required to write your child workflow execution results to a folder named *csvJobs* in an Amazon S3 bucket using the [PutObject](#) API action.

```
{
  "Version": "2012-10-17",
```

```
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "s3:PutObject",
      "s3:GetObject",
      "s3:ListMultipartUploadParts",
      "s3:AbortMultipartUpload"
    ],
    "Resource": [
      "arn:aws:s3:::resultBucket/csvJobs/*"
    ]
  }
]
```

If the Amazon S3 bucket to which you're writing the child workflow execution result is encrypted using an AWS Key Management Service (AWS KMS) key, you must include the necessary AWS KMS permissions in your IAM policy. For more information, see [IAM permissions for AWS KMS key encrypted Amazon S3 bucket](#).

Parsing an input CSV file

Because there isn't a standardized format to create and maintain data in CSV files, Step Functions parses CSV files based on the following rules:

- Commas (,) are a delimiter that separates individual fields.
- Newlines are a delimiter that separates individual records.
- Fields are treated as strings. For data type conversions, use the [States.StringToJson](#) intrinsic function in [ItemSelector](#).
- Double quotation marks (" ") aren't required to enclose strings. However, strings that are enclosed by double quotation marks can contain commas and newlines without them functioning as delimiters.
- Escape double quotes by repeating them.
- If the number of fields in a row is less than the number of fields in the header, Step Functions provides empty strings for the missing values.
- If the number of fields in a row is more than the number of fields in the header, Step Functions skips the additional fields.

Example of parsing an input CSV file

Say that you have provided a CSV file named *myCSVInput.csv* that contains one row as input. Then, you've stored this file in an Amazon S3 bucket that's named *my-bucket*. The CSV file is as follows.

```
abc,123,"This string contains commas, a double quotation marks (""), and a newline (
)",{"MyKey":"MyValue"},[1,2,3]"
```

The following state machine reads this CSV file and uses [ItemSelector](#) to convert the data types of some of the fields.

```
{
  "StartAt": "Map",
  "States": {
    "Map": {
      "Type": "Map",
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "DISTRIBUTED",
          "ExecutionType": "STANDARD"
        },
        "StartAt": "Pass",
        "States": {
          "Pass": {
            "Type": "Pass",
            "End": true
          }
        }
      },
      "End": true,
      "Label": "Map",
      "MaxConcurrency": 1000,
      "ItemReader": {
        "Resource": "arn:aws:states:::s3:getObject",
        "ReaderConfig": {
          "InputType": "CSV",
          "CSVHeaderLocation": "GIVEN",
          "CSVHeaders": [
            "MyLetters",
            "MyNumbers",
            "MyString",
```



```
        "MyObject",
        "MyArray"
    ]
},
"Parameters": {
    "Bucket": "my-bucket",
    "Key": "myCSVInput.csv"
}
},
"ItemSelector": {
    "MyLetters.$": "$$.Map.Item.Value.MyLetters",
    "MyNumbers.$": "States.StringToJson($$.Map.Item.Value.MyNumbers)",
    "MyString.$": "$$.Map.Item.Value.MyString",
    "MyObject.$": "States.StringToJson($$.Map.Item.Value.MyObject)",
    "MyArray.$": "States.StringToJson($$.Map.Item.Value.MyArray)"
}
}
}
```

When you run this state machine, it produces the following output.

```
[
  {
    "MyNumbers": 123,
    "MyObject": {
      "MyKey": "MyValue"
    },
    "MyString": "This string contains commas, a double quote (\"), and a newline (\n)",
    "MyLetters": "abc",
    "MyArray": [
      1,
      2,
      3
    ]
  }
]
```

Context object

The context object is an internal JSON structure that is available during an execution, and contains information about your state machine and execution. This allows your workflows access to

information about their specific execution. You can access the context object from the following fields:

- InputPath
- OutputPath
- ItemsPath (in Map states)
- Variable (in Choice states)
- ResultSelector
- Parameters
- Variable to variable comparison operators

Context Object Format

The context object includes information about the state machine, state, execution, and task. This JSON object includes nodes for each type of data, and is in the following format.

```
{
  "Execution": {
    "Id": "String",
    "Input": {},
    "Name": "String",
    "RoleArn": "String",
    "StartTime": "Format: ISO 8601",
    "RedriveCount": Number,
    "RedriveTime": "Format: ISO 8601"
  },
  "State": {
    "EnteredTime": "Format: ISO 8601",
    "Name": "String",
    "RetryCount": Number
  },
  "StateMachine": {
    "Id": "String",
    "Name": "String"
  },
  "Task": {
    "Token": "String"
  }
}
```

During an execution, the context object is populated with relevant data for the `Parameters` field from where it is accessed. The value for a `Task` field is null if the `Parameters` field is outside of a task state.

The value of the `RedriveCount` context object is 0, if you've not yet [redriven](#) an execution. Further, the `RedriveTime` context object is only available if you've redriven an execution. If you've [redriven a Map Run](#), the `RedriveTime` context object is only available for child workflows of type `Standard`. For a redriven `Map Run` with child workflows of type `Express`, `RedriveTime` isn't available.

Content from a running execution includes specifics in the following format.

```
{
  "Execution": {
    "Id": "arn:aws:states:us-east-1:123456789012:execution:stateMachineName:executionName",
    "Input": {
      "key": "value"
    },
    "Name": "executionName",
    "RoleArn": "arn:aws:iam::123456789012:role...",
    "StartTime": "2019-03-26T20:14:13.192Z"
  },
  "State": {
    "EnteredTime": "2019-03-26T20:14:13.192Z",
    "Name": "Test",
    "RetryCount": 3
  },
  "StateMachine": {
    "Id": "arn:aws:states:us-east-1:123456789012:stateMachine:stateMachineName",
    "Name": "stateMachineName"
  },
  "Task": {
    "Token": "h7XRiCdLtd/83p1E0dMccoxlzfHglSDKzPK9mBVKZsp7d9yrT1W"
  }
}
```

Note

For context object data related to Map states, see [Context Object Data for Map States](#).

Accessing the Context Object

To access the context object, first specify the parameter name by appending `.$` to the end, as you do when selecting state input with a path. Then, to access context object data instead of the input, prepend the path with `$$..` This tells AWS Step Functions to use the path to select a node in the context object.

The following examples show how you can access context objects, such as execution ID, name, start time, and redrive count.

- [Retrieve and pass execution ARN to a downstream service](#)
- [Access the execution start time and name in a Pass state](#)
- [Access the redrive count of an execution](#)

Retrieve and pass execution ARN to a downstream service

This example Task state uses a path to retrieve and pass the execution Amazon Resource Name (ARN) to an Amazon Simple Queue Service (Amazon SQS) message.

```
{
  "Order Flight Ticket Queue": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sqs:sendMessage",
    "Parameters": {
      "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/flight-purchase",
      "MessageBody": {
        "From": "YVR",
        "To": "SEA",
        "Execution.$": "$$.Execution.Id"
      }
    }
  },
  "Next": "NEXT_STATE"
}
```

For more information about using the task token when calling an integrated service, see [Wait for a Callback with the Task Token](#).

Access the execution start time and name in a Pass state

```
{
```

```

"Comment": "Accessing context object in a state machine",
"StartAt": "Get execution context data",
"States": {
  "Get execution context data": {
    "Type": "Pass",
    "Parameters": {
      "startTime.$": "$$.Execution.StartTime",
      "execName.$": "$$.Execution.Name"
    },
    "ResultPath": "$.executionContext",
    "End": true
  }
}
}
}

```

Access the redrive count of an execution

The following example of a Task state definition shows how you can access the [redrive](#) count of an execution.

```

{
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "OutputPath": "$.Payload",
  "Parameters": {
    "Payload": {
      "Number.$": "$$.Execution.RedriveCount"
    },
    "FunctionName": "arn:aws:lambda:us-east-2:123456789012:function:functionName"
  },
  "End": true
}

```

Context Object Data for Map States

There are two additional items available in the context object when processing a [Map state](#): `Index` and `Value`. For each Map state iteration, `Index` contains the index number for the array item that is being currently processed, while `Value` contains the array item being processed. Within a Map state, the context object includes the following data:


```

"Map": {

```

```
"Item": {
  "Index": Number,
  "Value": "String"
}
```

These are available only in a Map state, and can be specified in the [ItemSelector](#) field.

 **Note**

You must define parameters from the context object in the ItemSelector block of the main Map state, not within the states included in the ItemProcessor section.

Given a state machine with a simple Map state, we can inject information from the context object as follows.

```
{
  "StartAt": "ExampleMapState",
  "States": {
    "ExampleMapState": {
      "Type": "Map",
      "ItemSelector": {
        "ContextIndex.$": "$$.Map.Item.Index",
        "ContextValue.$": "$$.Map.Item.Value"
      },
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "INLINE"
        },
        "StartAt": "TestPass",
        "States": {
          "TestPass": {
            "Type": "Pass",
            "End": true
          }
        }
      },
      "End": true
    }
  }
}
```

If you execute the previous state machine with the following input, `Index` and `Value` are inserted in the output.

```
[
  {
    "who": "bob"
  },
  {
    "who": "meg"
  },
  {
    "who": "joe"
  }
]
```

The output for the execution returns the values of `Index` and `Value` items for each of the three iterations as follows:

```
[
  {
    "ContextIndex": 0,
    "ContextValue": {
      "who": "bob"
    }
  },
  {
    "ContextIndex": 1,
    "ContextValue": {
      "who": "meg"
    }
  },
  {
    "ContextIndex": 2,
    "ContextValue": {
      "who": "joe"
    }
  }
]
```

Data flow simulator

You can design, implement, and debug workflows in the [Step Functions console](#). You can also control the flow of data in your workflows with [JsonPath](#) input and output data processing. With the [Data flow simulator](#), you can simulate the order that the [Task state](#) states in your workflow process data at runtime. Using the simulator, you can understand how to filter and manipulate data as it flows from one state to another. It simulates each of the following fields that Step Functions uses to process and control the flow of JSON data:

[InputPath](#)

Selects *WHAT* portion of the entire input payload to be used as a task's input. If you specify this field, Step Functions first applies this field.

[Parameters](#)

Specifies *HOW* the input should look like before invoking the task. With the `Parameters` field, you can create a collection of key-value pairs that are passed as input to an [AWS service integration](#), such as an AWS Lambda function. These values can be static, or dynamically selected from either the state input or the [workflow context object](#).

[ResultSelector](#)

Determines *WHAT to choose* from a task's output. With the `ResultSelector` field, you can create a collection of key-value pairs that replace a state's result and pass that collection to `ResultPath`.

[ResultPath](#)

Determines *WHERE to put* a task's output. Use the `ResultPath` to determine whether the output of a state is a copy of its input, the result it produces, or a combination of both.

[OutputPath](#)

Determines *WHAT to send* to the next state. With `OutputPath`, you can filter out unwanted information, and pass only the portion of JSON data that you care about.

In this topic

- [Using Data flow simulator](#)
- [Considerations about using the Data flow simulator](#)

Using Data flow simulator

The simulator provides a real-time, side-by-side comparison of your data before and after you apply an [input and output data processing](#) field. To use the simulator, specify a JSON input. Then, evaluate it through each of the input and output processing fields. The simulator automatically validates your JSON input and highlights any syntax errors.

To use the data flow simulator

In the following steps, you provide JSON input and apply the [InputPath](#) and [Parameters](#) fields. You can also apply the other available fields and view their outputs.

1. Open the [Step Functions console](#).
2. In the navigation pane, choose **Data flow simulator**.
3. In the **State input** area, replace the prepopulated example JSON data with the following JSON data. Then, choose **Next**.

```
{
  "data": {
    "firstname": "Jane",
    "lastname": "Doe",
    "identity": {
      "email": "jdoe@example.com",
      "ssn": "123-45-6789"
    },
    "address": {
      "street": "123 Main St",
      "city": "Columbus",
      "state": "OH",
      "zip": "43219"
    }
  }
}
```

4. For **InputPath**, enter **\$.data.address** to select the address node of the input JSON data.

The **State input after InputPath** box displays the following results.

```
{
  "street": "123 Main St",
  "city": "Columbus",
```

```
"state": "OH",
"zip": "43219"
}
```

5. Choose **Next**.
6. Apply the **Parameters** field to convert the resulting JSON data to a string. To convert the data, do the following:
 - In the **Parameters** box, enter the following code to create a string called `addressString`.

```
{
  "addressString.$": "States.Format('{} . {}, {} - {}', $.street, $.city,
  $.state, $.zip)"
}
```

7. View the result of the **Parameters** field application in the **Filtered input after Parameters** box.

Considerations about using the Data flow simulator

Before you use the Data flow simulator, consider its limitations, including, but not limited to:

- **Unsupported filter expressions**

Filter expressions in the simulator behave differently than in the Step Functions service. This includes expressions that use the following syntax: `[?(expression)]`. The following is a list of unsupported expressions. If used, these expressions may not return the expected outcome after their evaluation.

- `$.book[?(@.isInStock==true)]`
- `$.book[?(@.price > 10.0)].title`

- **Incorrect JsonPath evaluation for single item arrays**

If you filter your data with a JsonPath expression that'd return a single array item, the simulator returns the item without the array. For example, consider the following array of data, called `items`:

```
{
  "items": [
```

```
{
  "name": "shoe",
  "color": "blue",
  "comment": "nice shoe"
},
{
  "name": "hat",
  "color": "red"
},
{
  "name": "shirt",
  "color": "yellow"
}
]
```

Given this `items` array, if you enter `$.comment` in the [InputPath](#) field, you'd expect the following output:

```
[
  "nice shoe"
]
```

However, the Data flow simulator returns the following output instead:

```
"nice shoe"
```

For JsonPath evaluation of an array that contains multiple items, the simulator returns the expected output.

Manage continuous deployments with versions and aliases

You can use Step Functions to manage continuous deployments of your workflows through state machine *versions* and *aliases*. A *version* is a numbered, immutable snapshot of a state machine that you can run. An *alias* is a pointer for up to two versions of a state machine.

You can maintain multiple versions of your state machines and manage their deployment in your production workflow. With aliases, you can route traffic between different workflow versions and gradually deploy those workflows to the production environment.

Additionally, you can start state machine executions using a version or an alias. If you don't use a version or alias when you start a state machine execution, Step Functions uses the latest revision of the state machine definition.

State machine revision

A state machine can have one or more revisions. When you update a state machine using the [UpdateStateMachine](#) API action, it creates a new state machine revision. A *revision* is an immutable, read-only snapshot of a state machine's definition and configuration. You can't start a state machine execution from a revision, and revisions don't have an ARN. Revisions have a `revisionId`, which is a universally unique identifier (UUID).

Contents

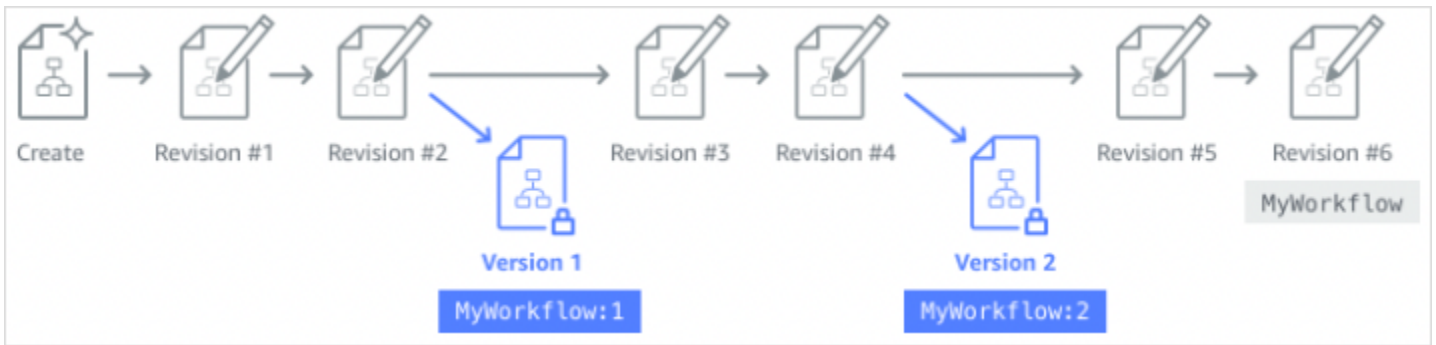
- [State machine versions](#)
- [State machine aliases](#)
- [Authorization for versions and aliases](#)
- [Associating state machine executions with a version or alias](#)
- [Alias and version deployment example](#)
- [Perform gradual deployment of state machine versions](#)

State machine versions

A *version* is a numbered, immutable snapshot of a state machine. You publish versions from the most recent revision made to that state machine. Each version has a unique Amazon Resource Name (ARN). This ARN is a combination of state machine ARN and the version number separated by a colon (:). The following example shows the format of a state machine version ARN.

```
arn:partition:states:region:account-id:stateMachine:myStateMachine:1
```

To start using state machine versions, you must publish the first version. After you publish a version, you can invoke the [StartExecution](#) API action with the version ARN. You can't edit a version, but you can update a state machine and publish a new version. You can also publish multiple versions of your state machine.



When you publish a new version of your state machine, Step Functions assigns it a version number. Version numbers start at 1 and increase monotonically for each new version. Version numbers aren't reused for a given state machine. If you delete version 10 of your state machine and then publish a new version, Step Functions publishes it as version 11.

The following properties are the same for all versions of a state machine:

- All versions of a state machine share the same type ([Standard or Express](#)).
- You can't change the name or creation date of a state machine between versions.
- Tags apply globally to state machines. You can manage tags for state machines using the [TagResource](#) and [UntagResource](#) API actions.

State machines also contain properties that are a part of each version and [revision](#), but these properties can differ between two given versions or revisions. These properties include [State machine definition](#), [IAM role](#), [tracing configuration](#), and [logging configuration](#).

Contents

- [Publishing a state machine version \(Console\)](#)
- [Managing versions with Step Functions API operations](#)
- [Running a state machine version from the console](#)

Publishing a state machine version (Console)

You can publish up to 1000 versions of a state machine. To request an increase to this soft limit, use the **Support Center** page in the [AWS Management Console](#). You can manually delete unused versions from the console or by invoking the [DeleteStateMachineVersion](#) API action.

To publish a state machine version

1. Open the [Step Functions console](#), and then choose an existing state machine.
2. On the **State machine detail** page, choose **Edit**.
3. Edit the state machine definition as required, and then choose **Save**.
4. Choose **Publish version**.
5. (Optional) In the **Description** field of the dialog box that appears, enter a brief description about the state machine version.
6. Choose **Publish**.

Note

When you publish a new version of your state machine, Step Functions assigns it a version number. Version numbers start at 1 and increase monotonically for each new version. Version numbers aren't reused for a given state machine. If you delete version 10 of your state machine and then publish a new version, Step Functions publishes it as version 11.

Managing versions with Step Functions API operations

Step Functions provides the following API operations to publish and manage state machine versions:

- [PublishStateMachineVersion](#) – Publishes a version from the current [revision](#) of a state machine.
- [UpdateStateMachine](#) – Publishes a new state machine version if you update a state machine and set the `publish` parameter to `true` in the same request.
- [CreateStateMachine](#) – Publishes the first revision of the state machine if you set the `publish` parameter to `true`.
- [ListStateMachineVersions](#) – Lists versions for the specified state machine ARN.
- [DescribeStateMachine](#) – Returns the state machine version details for a version ARN specified in `stateMachineArn`.
- [DeleteStateMachineVersion](#) – Deletes a state machine version.

To publish a new version from the current revision of a state machine called *myStateMachine* using the AWS Command Line Interface, use the `publish-state-machine-version` command:

```
aws stepfunctions publish-state-machine-version --state-machine-arn arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine
```

The response returns the `stateMachineVersionArn`. For example, the previous command returns a response of `arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine:1`.

Note

When you publish a new version of your state machine, Step Functions assigns it a version number. Version numbers start at 1 and increase monotonically for each new version. Version numbers aren't reused for a given state machine. If you delete version 10 of your state machine and then publish a new version, Step Functions publishes it as version 11.

Running a state machine version from the console

To start using state machine versions, you must first publish a version from the current state machine [revision](#). To publish a version, use the Step Functions console or invoke the [PublishStateMachineVersion](#) API action. You can also invoke the [UpdateStateMachineAlias](#) API action with an optional parameter named `publish` to update a state machine and publish its version.

You can start executions of a version by using the console or by invoking the [StartExecution](#) API action and providing the version ARN. You can also use an [alias](#) to start executions of a version. Based on its [routing configuration](#), an alias routes traffic to a specific version.

If you start a state machine execution without using a version, Step Functions uses the most recent revision of the state machine for the execution. For information about how Step Functions associates an execution with a version, see [Associating executions with a version or alias](#).

To start an execution using a state machine version

1. Open the [Step Functions console](#), and then choose an existing state machine that you've published one or more versions for. To learn how to publish a version, see [Publishing a state machine version \(Console\)](#).
2. On the **State machine detail** page, choose the **Versions** tab.
3. In the **Versions** section, do the following:

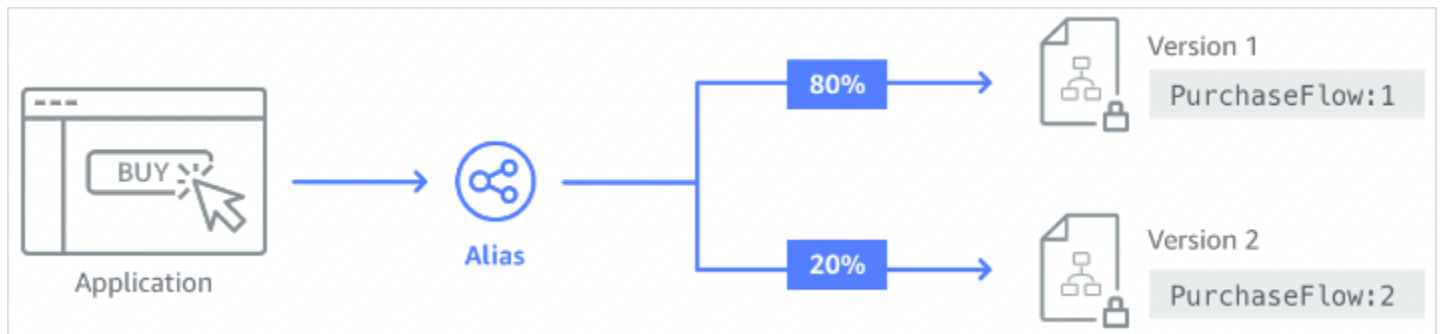
- a. Select the version that you want to start the execution with.
 - b. Choose **Start execution**.
4. (Optional) In the **Start execution** dialog box, enter a name for the execution.
 5. (Optional) , enter the execution input, and then choose **Start execution**.

State machine aliases

An *alias* is a pointer for up to two versions of the same state machine. You can create multiple aliases for your state machines. Each alias has a unique Amazon Resource Name (ARN). The alias ARN is a combination of the state machine's ARN and the alias name, separated by a colon (:). The following example shows the format of a state machine alias ARN.

```
arn:partition:states:region:account-id:stateMachine:myStateMachine:aliasName
```

You can use an alias to [route traffic](#) between one of the two state machine versions. You can also create an alias that points to a single version. Aliases can only point to state machine versions. You can't use an alias to point to another alias. You can also update an alias to point to a different version of the state machine.



Contents

- [Creating a state machine alias \(Console\)](#)
- [Managing aliases with Step Functions API operations](#)
- [Alias routing configuration](#)
- [Running a state machine using an alias \(Console\)](#)

Creating a state machine alias (Console)

You can create up to 100 aliases for each state machine by using the Step Functions console or by invoking the [CreateStateMachineAlias](#) API action. To request an increase to this soft limit, use the **Support Center** page in the [AWS Management Console](#). Delete unused aliases from the console or by invoking the [DeleteStateMachineAlias](#) API action.

To create a state machine alias

1. Open the [Step Functions console](#), and then choose an existing state machine.
2. On the **State machine detail** page, choose the **Aliases** tab.
3. Choose **Create new alias**.
4. On the **Create alias** page, do the following:
 - a. Enter an **Alias name**.
 - b. (Optional) Enter a **Description** for the alias.
5. To configure routing on the alias, see [Alias routing configuration](#).
6. Choose **Create alias**.

Managing aliases with Step Functions API operations

Step Functions provides the following API operations that you can use to create and manage state machine aliases or get information about the aliases:

- [CreateStateMachineAlias](#) – Creates an alias for a state machine.
- [DescribeStateMachineAlias](#) – Returns details about a state machine alias.
- [ListStateMachineAliases](#) – Lists aliases for the specified state machine ARN.
- [UpdateStateMachineAlias](#) – Updates the configuration of an existing state machine alias by modifying its `description` or `routingConfiguration`.
- [DeleteStateMachineAlias](#) – Deletes a state machine version.

To create an alias named *PROD* that points to version 1 of a state machine named *myStateMachine* using the AWS Command Line Interface, use the `create-state-machine-alias` command.

```
aws stepfunctions create-state-machine-alias --name PROD --routing-configuration "[{\"stateMachineVersionArn\":\"arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine:1\"}, {\"weight\":100}]"
```

Alias routing configuration

You can use an alias to route execution traffic between two versions of a state machine. For example, say you want to launch a new version of your state machine. You can reduce the risks involved in deploying the new version by configuring routing on an alias. By configuring routing, you can send most of your traffic to an earlier, tested version of your state machine. The new version can then receive a smaller percentage, until you can confirm that it's safe to roll forward the new version.

To define routing configuration, make sure that you publish both state machine versions that your alias points to. When you start an execution from an alias, Step Functions randomly chooses the state machine version to run from the versions specified in the routing configuration. It bases this choice on the traffic percentage that you assign to each version in the alias routing configuration.

To configure routing configuration on an alias

- On the **Create alias** page, under **Routing configuration**, do the following:
 - a. For **Version**, choose the first state machine version that the alias points to.
 - b. Select the **Split traffic between two versions** check box.

Tip

To point to a single version, clear the **Split traffic between two versions** check box.

- c. For **Version**, choose the second version that the alias must point to.
- d. In the **Traffic percentage** fields, specify the percentage of traffic to route to each version. For example, enter **60** and **40** to route 60 percent of the execution traffic to the first version and 40 percent traffic to the second version.

The combined traffic percentages must equal to 100 percent.

Running a state machine using an alias (Console)

You can start state machine executions with an alias from either the console or by invoking the [StartExecution](#) API action with the alias' ARN. Step Functions then runs the version specified by the alias. By default, if you don't specify a version or alias when you start a state machine execution, Step Functions uses the most recent revision.

To start a state machine execution using an alias

1. Open the [Step Functions console](#), then choose an existing state machine that you've created an alias for. For information about creating an alias, see [Creating a state machine alias \(Console\)](#).
2. On the **State machine detail** page, choose the **Aliases** tab.
3. In the **Aliases** section, do the following:
 - a. Select the alias that you want to start the execution with.
 - b. Choose **Start execution**.
4. (Optional) In the **Start execution** dialog box, enter a name for the execution.
5. If required, enter the execution input, and then choose **Start execution**.

Authorization for versions and aliases

To invoke Step Functions API actions with a version or an alias, you need appropriate permissions. To authorize a version or an alias to invoke an API action, Step Functions uses the state machine's ARN instead of using the version ARN or alias ARN. You can also scope down the permissions for a specific version or alias. For more information, see [Scoping down permissions](#).

You can use the following IAM policy example of a state machine named *myStateMachine* to invoke the [CreateStateMachineAlias](#) API action to create a state machine alias.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "states:CreateStateMachineAlias",
      "Resource": "arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine"
    }
  ]
}
```

```
]
}
```

When you set permissions to allow or deny access to API actions using state machine versions or aliases, consider the following:

- If you use the `publish` parameter of the [CreateStateMachine](#) and [UpdateStateMachine](#) API actions to publish a new state machine version, you also need the `ALLOW` permission on the [PublishStateMachineVersion](#) API action.
- The [DeleteStateMachine](#) API action deletes all versions and aliases associated with a state machine.

In this topic

- [Scoping down permissions for a version or alias](#)

Scoping down permissions for a version or alias

You can use a qualifier to further scope down the authorization permission needed by a version or an alias. A qualifier refers to a version number or an alias name. You use the qualifier to qualify a state machine. The following example is a state machine ARN that uses an alias named `PROD` as the qualifier.

```
arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine:PROD
```

For more information about qualified and unqualified ARNs, see [Associating executions with a version or alias](#).

You scope down the permissions using the optional context key named `states:StateMachineQualifier` in an IAM policy's `Condition` statement. For example, the following IAM policy for a state machine named `myStateMachine` denies access to invoke the [DescribeStateMachine](#) API action with an alias named as `PROD` or the version `1`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
```

```
"Action": "states:DescribeStateMachine",
"Resource": "arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine",
"Condition": {
  "ForAnyValue:StringEquals": {
    "states:StateMachineQualifier": [
      "PROD",
      "1"
    ]
  }
}
```

The following list specifies the API actions on which you can scope down the permissions with the `StateMachineQualifier` context key.

- [CreateStateMachineAlias](#)
- [DeleteStateMachineAlias](#)
- [DeleteStateMachineVersion](#)
- [DescribeStateMachine](#)
- [DescribeStateMachineAlias](#)
- [ListExecutions](#)
- [ListStateMachineAliases](#)
- [StartExecution](#)
- [StartSyncExecution](#)
- [UpdateStateMachineAlias](#)

Associating state machine executions with a version or alias

Step Functions associates an execution with a version or alias based on the Amazon Resource Name (ARN) that you use to invoke the [StartExecution](#) API action. Step Functions performs this action at the execution start time.

You can start a state machine execution using a qualified or an unqualified ARN.

- **Qualified ARN** – Refers to a state machine ARN suffixed with a version number or an alias name.

The following qualified ARN example refers to version 3 of a state machine named `myStateMachine`.

```
arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine:3
```

The following qualified ARN example refers to an alias named `PROD` of a state machine named `myStateMachine`.

```
arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine:PROD
```

- **Unqualified ARN** – Refers to a state machine ARN without a version number or an alias name suffix.

```
arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine
```

For example, if your qualified ARN refers to version 3, Step Functions associates the execution with this version. It doesn't associate the execution with any aliases that point to the version 3.

If your qualified ARN refers to an alias, Step Functions associates the execution with that alias and the version to which the alias points. An execution can only be associated with one alias.

Note

If you start an execution with an unqualified ARN, Step Functions doesn't associate that execution with a version even if the version uses the same state machine [revision](#). For example, if version 3 uses the latest revision, but you start an execution with an unqualified ARN, Step Functions doesn't associate that execution with the version 3.

In this topic

- [Viewing executions started with a version or an alias](#)

Viewing executions started with a version or an alias

Step Functions provides the following ways in which you can view the executions started with a version or an alias:

Using API actions

You can view all the executions associated with a version or an alias by invoking the [DescribeExecution](#) and [ListExecutions](#) API actions. These API actions return the ARN of the version or alias that was used to start the execution. These actions also return other details including status and ARN of the execution.

You can also provide a state machine alias ARN or version ARN to list the executions associated with a specific alias or version.

The following example response of the [ListExecutions](#) API action shows the ARN of the alias used to start a state machine execution named *myFirstExecution*.

The *italicized* text in the following code snippet represents resource-specific information.

```
{
  "executions": [
    {
      "executionArn": "arn:aws:states:us-
east-1:123456789012:execution:myStateMachine:myFirstExecution",
      "stateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:myStateMachine",
      "stateMachineAliasArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:myStateMachine:PROD",
      "name": "myFirstExecution",
      "status": "SUCCEEDED",
      "startDate": "2023-04-20T23:07:09.477000+00:00",
      "stopDate": "2023-04-20T23:07:09.732000+00:00"
    }
  ]
}
```

Using Step Functions console

You can also view the executions started by a version or an alias from the [Step Functions console](#). The following procedure shows how you can view the executions started with a specific version:

1. Open the [Step Functions console](#), and then choose an existing state machine for which you've published a [version](#) or created an [alias](#). This example shows how to view the executions started with a specific state machine version.
2. Choose the **Versions** tab, and then choose a version from the **Versions** list.

Tip

Filter by property or value box to search for a specific version.

3. On the *Version details page*, you can see a list of all the in-progress and past state machine executions started with the selected version.

The following image shows the *Version Details* console page. This page lists executions started by the version 4 of a state machine named *MathAddDemo*. This list also displays an execution that was started by an alias named *PROD*. This alias routed the execution traffic to version 4.

Step Functions > State machines > MathAddDemo > Version: 4

Version: 4 Switch version ▼ Info Delete Start execution

Details

ARN
arn:aws:states:us-east-1:123456789012:stateMachine:MathAddDemo:4

IAM role ARN
arn:aws:iam::123456789012:role/service-role/StepFunctions-MathAddDemo-role-3d6c9a40

Publish date
Jun 5, 2023 01:31:29.626 PM PDT

Description
Added a terminal state.

Executions | Definition | Used by alias | Metrics | Logs

Executions (3) Refresh View details Stop execution Start execution

Filter executions by property or value All Last 1 year 3 matches < 1 >

Name	Status	Version	Alias	Started	End Time
MathDemo-PROD-2	Succeeded	4	PROD	Jun 5, 2023, 14:31:13.461 (UTC-07:00)	Jun 5, 2023, 14:31:13.567 (UTC-07:00)
MathAddDemo-ver4-2	Succeeded	4	-	Jun 5, 2023, 13:34:53.666 (UTC-07:00)	Jun 5, 2023, 13:34:53.742 (UTC-07:00)
MathAddDemo-ver4-1	Failed	4	-	Jun 5, 2023, 13:33:31.122 (UTC-07:00)	Jun 5, 2023, 13:33:31.198 (UTC-07:00)

Using CloudWatch metrics

For each state machine execution that you start with a [Qualified ARN](#), Step Functions emits additional metrics with the same name and value as the metrics emitted currently. These additional metrics contain dimensions for each of the version identifier and alias name with which you start an execution. With these metrics, you can monitor state machine executions at the version level and determine when a rollback scenario might be necessary. You can also [create Amazon CloudWatch alarms](#) based on these metrics.

Step Functions emits the following metrics for executions that you start with an alias or a version:

- ExecutionTime
- ExecutionsAborted
- ExecutionsFailed
- ExecutionsStarted
- ExecutionsSucceeded
- ExecutionsTimedOut

If you started the execution with a version ARN, Step Functions publishes the metric with the `StateMachineArn` and a second metric with `StateMachineArn` and `Version` dimensions.

If you started the execution with an alias ARN, Step Functions emits the following metrics:

- Two metrics for the unqualified ARN and version.
- A metric with the `StateMachineArn` and `Alias` dimensions.

Alias and version deployment example

The following example of the Canary deployment technique shows how you can deploy a new state machine version with the AWS Command Line Interface. In this example, the alias you create routes 20 percent of execution traffic to the new version. It then routes the remaining 80 percent to the earlier version. To deploy a new state machine [version](#) and shift execution traffic with an [alias](#), complete the following steps:

1. Publish a version from the current state machine revision.

Use the **publish-state-machine-version** command in the AWS CLI to publish a version from the current revision of a state machine called *myStateMachine*:

```
aws stepfunctions publish-state-machine-version --state-machine-arn
arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine
```

The response returns the `stateMachineVersionArn` of the version that you published. For example, `arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine:1`.

2. Create an alias that points to the state machine version.

Use the `create-state-machine-alias` command to create an alias named *PROD* that points to version 1 of *myStateMachine*:

```
aws stepfunctions create-state-machine-alias --name PROD --routing-configuration "[{\"stateMachineVersionArn\":\"arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine:1\"}, {\"weight\":100}]"
```

3. Verify that executions started by the alias use correct published version.

Start a new execution of *myStateMachine* by providing the ARN of the alias **PROD** in the `start-execution` command:

```
aws stepfunctions start-execution --state-machine-arn arn:aws:states:us-east-1:123456789012:stateMachineAlias:myStateMachine:PROD --input "{}"
```

If you provide the state machine ARN in the [StartExecution](#) request, it uses the most recent [revision](#) of the state machine instead of the version specified in your alias for starting the execution.

4. Update the state machine definition and publish a new version.

Update *myStateMachine* and publish its new version. To do this, use the optional `publish` parameter of the `update-state-machine` command:

```
aws stepfunctions update-state-machine --state-machine-arn arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine --definition $UPDATED_STATE_MACHINE_DEFINITION --publish
```

The response returns the `stateMachineVersionArn` for the new version. For example, `arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine:2`.

5. Update the alias to point to both the versions and set the alias' [routing configuration](#).

Use the **update-state-machine-alias** command to update the routing configuration of the alias PROD. Configure the alias so that 80 percent of the execution traffic goes to version 1 and the remaining 20 percent goes to version 2:

```
aws stepfunctions update-state-machine-alias --state-machine-alias-arn
arn:aws:states:us-east-1:123456789012:stateMachineAlias:myStateMachine:PROD
--routing-configuration "[{"stateMachineVersionArn":
\"arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine:1\",
\"weight\":80}, {"stateMachineVersionArn\":\"arn:aws:states:us-
east-1:123456789012:stateMachine:myStateMachine:2\", \"weight\":20}]"
```

6. Replace version 1 with version 2.

After you verify that your new state machine version works correctly, you can deploy the new state machine version. To do this, update the alias again to assign 100 percent of execution traffic to the new version.

Use the **update-state-machine-alias** command to set the routing configuration of the alias PROD to 100 percent for version 2:

```
aws stepfunctions update-state-machine-alias --state-machine-alias-arn
arn:aws:states:us-east-1:123456789012:stateMachineAlias:myStateMachine:PROD
--routing-configuration "[{"stateMachineVersionArn\":\"arn:aws:states:us-
east-1:123456789012:stateMachine:myStateMachine:2\", \"weight\":100}]"
```

Tip

To roll back the deployment of version 2, edit the alias' routing configuration to shift 100 percent of traffic to the newly deployed version.

```
aws stepfunctions update-state-machine-alias
--state-machine-alias-arn arn:aws:states:us-
east-1:123456789012:stateMachineAlias:myStateMachine:PROD
--routing-configuration "[{"stateMachineVersionArn\":\"arn:aws:states:us-
east-1:123456789012:stateMachine:myStateMachine:1\", \"weight\":100}]"
```

You can use versions and aliases to perform other types of deployments. For instance, you can perform a *rolling deployment* of a new version of your state machine. To do so, gradually increase the weighted percentage in the routing configuration of the alias that points to the new version.

You can also use versions and aliases to perform a *blue/green deployment*. To do so, create an alias named `green` that runs the current version 1 of your state machine. Then, create another alias named `blue` that runs the new version, for example, `2`. To test the new version, send execution traffic to the `blue` alias. When you're confident that your new version works correctly, update the `green` alias to point to your new version.

Perform gradual deployment of state machine versions

A rolling deployment is a deployment strategy that slowly replaces previous versions of an application with new versions of an application. To perform a rolling deployment of a state machine version, gradually send an increasing amount of execution traffic to the new version. The amount of traffic and rate of increase are parameters that you configure.

You can perform rolling deployment of a version using one of the following options:

- [Step Functions console](#) – Create an alias that points to two versions of the same state machine. For this alias, you configure the routing configuration to shift traffic between the two versions. For more information about using the console to roll out versions, see [Versions](#) and [Aliases](#).
- **Scripts for AWS CLI and SDK** – Create a shell script using the AWS CLI or the AWS SDK. For more information, see the following sections for using AWS CLI and AWS SDK.
- **AWS CloudFormation templates** – Use the [AWS::StepFunctions::StateMachineVersion](#) and [AWS::StepFunctions::StateMachineAlias](#) resources to publish multiple state machine versions and create an alias to point to one or two of these versions.

Use the AWS CLI to deploy a new state machine version

The example script in this section shows how you can use the AWS CLI to gradually shift traffic from a previous state machine version to a new state machine version. You can either use this example script or update it according to your requirements.

This script shows a Canary deployment for deploying a new state machine version using an alias. The following steps outline the tasks that the script performs:

1. If the `publish_revision` parameter is set to `true`, publish the most recent [revision](#) as the next version of the state machine. This version becomes the new, live version if the deployment succeeds.

If you set the `publish_revision` parameter to `false`, the script deploys the last published version of the state machine.

2. Create an alias if it doesn't exist yet. If the alias doesn't exist, point 100 percent of traffic for this alias to the new version, and then exit the script.
3. Update the routing configuration of the alias to shift a small percentage of traffic from the previous version to the new version. You set this canary percentage with the `canary_percentage` parameter.
4. By default, monitor the configurable CloudWatch alarms every 60 seconds. If any of these alarms set off, rollback the deployment immediately by pointing 100 percent of traffic to the previous version.

After every time interval, in seconds, defined in `alarm_polling_interval`, continue monitoring the alarms. Continue monitoring until the time interval defined in `canary_interval_seconds` has passed.

5. If no alarms were set off during `canary_interval_seconds`, shift 100 percent of traffic to the new version.
6. If the new version deploys successfully, delete any versions older than the number specified in the `history_max` parameter.

```
#!/bin/bash
#
# AWS StepFunctions example showing how to create a canary deployment with a
# State Machine Alias and versions.
#
# Requirements: AWS CLI installed and credentials configured.
#
# A canary deployment deploys the new version alongside the old version, while
# routing only a small fraction of the overall traffic to the new version to
# see if there are any errors. Only once the new version has cleared a testing
# period will it start receiving 100% of traffic.
#
# For a Blue/Green or All at Once style deployment, you can set the
# canary_percentage to 100. The script will immediately shift 100% of traffic
```

```
# to the new version, but keep on monitoring the alarms (if any) during the
# canary_interval_seconds time interval. If any alarms raise during this period,
# the script will automatically rollback to the previous version.
#
# Step Functions allows you to keep a maximum of 1000 versions in version history
# for a state machine. This script has a version history deletion mechanism at
# the end, where it will delete any versions older than the limit specified.
#
# For a fuller example, that also demonstrates linear (or rolling) deployments,
# please see
# https://github.com/aws-samples/aws-stepfunctions-examples/blob/main/gradual-deploy/
# sfndeploy.py

set -euo pipefail

# *****
# you can safely change the variables in this block to your values
state_machine_name="my-state-machine"
alias_name="alias-1"
region="us-east-1"

# array of cloudwatch alarms to poll during the test period.
# to disable alarm checking, set alarm_names=()
alarm_names=("alarm1" "alarm name with a space")

# true to publish the current revision as the next version before deploy.
# false to deploy the latest version from the state machine's version history.
publish_revision=true

# true to force routing configuration update even if the current routing
# for the alias does not have a 100% routing config.
# false will abandon deploy attempt if current routing config not 100% to a
# single version.
# Be careful when you combine this flag with publish_revision - if you just
# rerun the script you might deploy the newly published revision from the
# previous run.
force=false

# percentage of traffic to route to the new version during the test period
canary_percentage=10

# how many seconds the canary deployment lasts before full deploy to 100%
canary_interval_seconds=300
```

```

# how often to poll the alarms
alarm_polling_interval=60

# how many versions to keep in history. delete versions prior to this.
# set to 0 to disable old version history deletion.
history_max=0
# *****

#####
# Update alias routing configuration.
#
# If you don't specify version 2 details, will only create 1 routing entry. In
# this case the routing entry weight must be 100.
#
# Globals:
#   alias_arn
# Arguments:
#   1. version 1 arn
#   2. version 1 weight
#   3. version 2 arn (optional)
#   4. version 2 weight (optional)
#####
function update_routing() {
    if [[ $# -eq 2 ]]; then
        local routing_config="{\"stateMachineVersionArn\": \"$1\", \"weight\":$2}"
    elif [[ $# -eq 4 ]]; then
        local routing_config="{\"stateMachineVersionArn\": \"$1\", \"weight\":$2},
{\"stateMachineVersionArn\": \"$3\", \"weight\":$4}"
    else
        echo "You have to call update_routing with either 2 or 4 input arguments." >&2
        exit 1
    fi

    ${aws} update-state-machine-alias --state-machine-alias-arn ${alias_arn} --routing-
configuration "${routing_config}"
}

# *****
# pre-run validation
if [[ (( "${#alarm_names[@]}" -gt 0 )) ]]; then
    alarm_exists_count=$(aws cloudwatch describe-alarms --alarm-names "${alarm_names[@]}"
--alarm-types "CompositeAlarm" "MetricAlarm" --query "length([MetricAlarms,
CompositeAlarms][])" --output text)

```

```

if [[ ("${#alarm_names[@]}" -ne "${alarm_exists_count}") ]]; then
    echo All of the alarms to monitor do not exist in CloudWatch: $(IFS=,; echo
"${alarm_names[*]}") >&2
    echo Only the following alarm names exist in CloudWatch:
    aws cloudwatch describe-alarms --alarm-names "${alarm_names[@]}" --alarm-types
"CompositeAlarm" "MetricAlarm" --query "join(', ', [MetricAlarms, CompositeAlarms]
[].AlarmName)" --output text
    exit 1
fi
fi

if [[ ("${history_max}" -gt 0) && ("${history_max}" -lt 2) ]]; then
    echo The minimum value for history_max is 2. This is the minimum number of older
state machine versions to be able to rollback in the future. >&2
    exit 1
fi
# *****
# main block follows

account_id=$(aws sts get-caller-identity --query Account --output text)

sm_arn="arn:aws:states:${region}:${account_id}:stateMachine:${state_machine_name}"

# the aws command we'll be invoking a lot throughout.
aws="aws stepfunctions"

# promote the latest revision to the next version
if [[ "${publish_revision}" = true ]]; then
    new_version=$((${aws} publish-state-machine-version --state-machine-arn=$sm_arn --
query stateMachineVersionArn --output text)
    echo Published the current revision of state machine as the next version with arn:
${new_version}
else
    new_version=$((${aws} list-state-machine-versions --state-machine-arn ${sm_arn} --max-
results 1 --query "stateMachineVersions[0].stateMachineVersionArn" --output text)
    echo "Since publish_revision is false, using the latest version from the state
machine's version history: ${new_version}"
fi

# find the alias if it exists
alias_arn_expected="${sm_arn}:${alias_name}"
alias_arn=$((${aws} list-state-machine-aliases --state-machine-arn
${sm_arn} --query "stateMachineAliases[?stateMachineAliasArn==\`
${alias_arn_expected}\`].stateMachineAliasArn" --output text)

```



```
if [[ "${alias_arn_expected}" == "${alias_arn}" ]]; then
    echo Found alias ${alias_arn}

    echo Current routing configuration is:
    ${aws} describe-state-machine-alias --state-machine-alias-arn "${alias_arn}" --query
routingConfiguration
else
    echo Alias does not exist. Creating alias ${alias_arn_expected} and routing 100%
traffic to new version ${new_version}

    ${aws} create-state-machine-alias --name "${alias_name}" --routing-configuration
["{\"stateMachineVersionArn\": \"${new_version}\", \"weight\":100}"]

    echo Done!
    exit 0
fi

# find the version to which the alias currently points (the current live version)
old_version=$((${aws} describe-state-machine-alias --state-machine-alias-arn $alias_arn
--query "routingConfiguration[?weight==`100`].stateMachineVersionArn" --output text))

if [[ -z "${old_version}" ]]; then
    if [[ "${force}" = true ]]; then
        echo Force setting is true. Will force update to routing config for alias to point
100% to new version.
        update_routing "${new_version}" 100

        echo Alias ${alias_arn} now pointing 100% to ${new_version}.
        echo Done!
        exit 0
    else
        echo Alias ${alias_arn} does not have a routing config entry with 100% of the
traffic. This means there might be a deploy in progress, so not starting another
deploy at this time. >&2
        exit 1
    fi
fi

if [[ "${old_version}" == "${new_version}" ]]; then
    echo The alias already points to this version. No update necessary.
    exit 0
fi
```

```
echo Switching ${canary_percentage}% to new version ${new_version}
(( old_weight = 100 - ${canary_percentage} ))
update_routing "${new_version}" ${canary_percentage} "${old_version}" ${old_weight}

echo New version receiving ${canary_percentage}% of traffic.
echo Old version ${old_version} is still receiving ${old_weight}%.

if [[ $#alarm_names[@] -eq 0 ]]; then
    echo No alarm_names set. Skipping cloudwatch monitoring.
    echo Will sleep for ${canary_interval_seconds} seconds before routing 100% to new
    version.
    sleep ${canary_interval_seconds}
    echo Canary period complete. Switching 100% of traffic to new version...
else
    echo Checking if alarms fire for the next ${canary_interval_seconds} seconds.

    (( total_wait = canary_interval_seconds + $(date +%s) ))

    now=$(date +%s)
    while [[ ((${now} -lt ${total_wait})) ]]; do
        alarm_result=$(aws cloudwatch describe-alarms --alarm-names "${alarm_names[@]}"
--state-value ALARM --alarm-types "CompositeAlarm" "MetricAlarm" --query "join(', ',
[MetricAlarms, CompositeAlarms][].AlarmName)" --output text)

        if [[ ! -z "${alarm_result}" ]]; then
            echo The following alarms are in ALARM state: ${alarm_result}. Rolling back
            deploy. >&2
            update_routing "${old_version}" 100

            echo Rolled back to ${old_version}
            exit 1
        fi

        echo Monitoring alarms...no alarms have triggered.
        sleep ${alarm_polling_interval}
        now=$(date +%s)
    done

    echo No alarms detected during canary period. Switching 100% of traffic to new
    version...
fi

update_routing "${new_version}" 100
```

```
echo Version ${new_version} is now receiving 100% of traffic.

if [[ ("${history_max}" -eq 0 )]]; then
    echo Version History deletion is disabled. Remember to prune your history, the
    default limit is 1000 versions.
    echo Done!
    exit 0
fi

echo Keep the last ${history_max} versions. Deleting any versions older than that...

# the results are sorted in descending order of the version creation time
version_history=$((${aws} list-state-machine-versions --state-
machine-arn ${sm_arn} --max-results 1000 --query "join('\n',
stateMachineVersions[].stateMachineVersionArn)" --output text)

counter=0

while read line; do
    ((counter=${counter} + 1))

    if [[ ( ${counter} -gt ${history_max}) ]]; then
        echo Deleting old version ${line}
        ${aws} delete-state-machine-version --state-machine-version-arn ${line}
    fi
done <<< "${version_history}"

echo Done!
```

Use the AWS SDK to deploy a new state machine version

The example script at [aws-stepfunctions-examples](#) shows how to use the AWS SDK for Python to gradually shift traffic from a previous version to a new version of a state machine. You can either use this example script or update it according to your requirements.

The script shows the following deployment strategies:

- **Canary** – Shifts traffic in two increments.

In the first increment, a small percentage of traffic, for example, 10 percent is shifted to the new version. In the second increment, before a specified time interval in seconds gets over, the remaining traffic is shifted to the new version. The switch to the new version for the remaining traffic takes place only if no CloudWatch alarms are set off during the specified time interval.

- **Linear or Rolling** – Shifts traffic to the new version in equal increments with an equal number of seconds between each increment.

For example, if you specify the increment percent as **20** with an `--interval` of **600** seconds, this deployment increases traffic by 20 percent every 600 seconds until the new version receives 100 percent of the traffic.

This deployment immediately rolls back the new version if any CloudWatch alarms are set off.

- **All at Once or Blue/Green** – Shifts 100 percent of traffic to the new version immediately. This deployment monitors the new version and rolls it back automatically to the previous version if any CloudWatch alarms are set off.

Use AWS CloudFormation to deploy a new state machine version

The following CloudFormation template example publishes two versions of a state machine named *MyStateMachine*. It creates an alias named *PROD*, which points to both these versions, and then deploys the version 2.

In this example, 10 percent of traffic is shifted to the version 2 every five minutes until this version receives 100 percent of the traffic. This example also shows how you can set CloudWatch alarms. If any of the alarms you set go into the ALARM state, the deployment fails and rolls back immediately.

```
MyStateMachine:
  Type: AWS::StepFunctions::StateMachine
  Properties:
    Type: STANDARD
    StateMachineName: MyStateMachine
    RoleArn: arn:aws:iam::123456789012:role/myIamRole
  Definition:
    StartAt: PassState
    States:
      PassState:
        Type: Pass
        Result: Result
        End: true

MyStateMachineVersionA:
  Type: AWS::StepFunctions::StateMachineVersion
  Properties:
    Description: Version 1
    StateMachineArn: !Ref MyStateMachine
```

```
MyStateMachineVersionB:
  Type: AWS::StepFunctions::StateMachineVersion
  Properties:
    Description: Version 2
    StateMachineArn: !Ref MyStateMachine

PROD:
  Type: AWS::StepFunctions::StateMachineAlias
  Properties:
    Name: PROD
    Description: The PROD state machine alias taking production traffic.
    DeploymentPreference:
      StateMachineVersionArn: !Ref MyStateMachineVersionB
      Type: LINEAR
      Percentage: 10
      Interval: 5
    Alarms:
      # A list of alarms that you want to monitor. If any of these alarms trigger,
      # rollback the deployment immediately by pointing 100 percent of traffic to the previous
      # version.
      - !Ref CloudWatchAlarm1
      - !Ref CloudWatchAlarm2
```

Executions in Step Functions

A state machine *execution* occurs when an AWS Step Functions state machine runs and performs its tasks. Each Step Functions state machine can have multiple simultaneous executions, which you can initiate from the [Step Functions console](#), or by using the AWS SDKs, the Step Functions API actions, or the AWS Command Line Interface (AWS CLI). An execution receives JSON input and produces JSON output. You can start a Step Functions execution in the following ways:

- Call the [StartExecution](#) API action.
- [Start a new execution](#) in the Step Functions console.
- Use Amazon EventBridge to [start an execution](#) in response to an event.
- Use Amazon EventBridge Scheduler to [start a state machine execution](#) on a schedule.
- Start an execution with [Amazon API Gateway](#).
- Start a [nested workflow execution](#) from a Task state.

For more information about the different ways of working with Step Functions, see [Development Options](#).

Start Workflow Executions from a Task State

AWS Step Functions can start workflow executions directly from a Task state of a state machine. This allows you to break your workflows into smaller state machines, and to start executions of these other state machines. By starting these new workflow executions you can:

- Separate higher level workflow from lower level, task-specific workflows.
- Avoid repetitive elements by calling a separate state machine multiple times.
- Create a library of modular reusable workflows for faster development.
- Reduce complexity and make it easier to edit and troubleshoot state machines.

Step Functions can start these workflow executions by calling its own API as an [integrated service](#). Simply call the `StartExecution` API action from your Task state and pass the necessary parameters. You can call the Step Functions API using any of the [service integration patterns](#).

Tip

To deploy an example of a nested workflow to your AWS account, see [Module 13 - Nested Express Workflows](#).

To start a new execution of a state machine, use a Task state similar to the following example:

```
{
  "Type": "Task",
  "Resource": "arn:aws:states:::states:startExecution",
  "Parameters": {
    "StateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:HelloWorld",
    "Input": {
      "Comment": "Hello world!"
    },
  },
  "Retry": [
    {
      "ErrorEquals": [
```

```

        "StepFunctions.ExecutionLimitExceeded"
    ]
}
],
"End":true
}

```

This Task state will start a new execution of the HelloWorld state machine, and will pass the JSON comment as input.

Note

The `StartExecution` API action quotas can limit the number of executions that you can start. Use the `Retry` on `StepFunctions.ExecutionLimitExceeded` to ensure your execution is started. See the following.

- [Quotas related to API action throttling](#)
- [Error handling in Step Functions](#)

Associate Workflow Executions

To associate a started workflow execution with the execution that started it, pass the execution ID from the [context object](#) to the execution input. You can access the ID from the context object from your Task state in a running execution. Pass the execution ID by appending `.$` to the parameter name, and referencing the ID in the context object with `$$`.`Execution.Id`.

```
"AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
```

You can use a special parameter named `AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID` when you start an execution. If included, this association provides links in the **Step details** section of the Step Functions console. When provided, you can easily trace the executions of your workflows from starting executions to their started workflow executions. Using the previous example, associate the execution ID with the started execution of the HelloWorld state machine, as follows.

```
{
  "Type": "Task",
```

```
"Resource": "arn:aws:states:::states:startExecution",
"Parameters": {
  "StateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:HelloWorld",
  "Input": {
    "Comment": "Hello world!",
    "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
  }
},
"End": true
}
```

For more information, see the following:

- [Working with other services](#)
- [Pass parameters to a service API](#)
- [Accessing the Context Object](#)
- [AWS Step Functions](#)

Using Amazon EventBridge Scheduler with AWS Step Functions

[Amazon EventBridge Scheduler](#) is a serverless scheduler that allows you to create, run, and manage tasks from one central, managed service. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for failed API invocations.

For example, with EventBridge Scheduler, you can start a state machine execution on a schedule when a security related event occurs or to automate a data processing job.

This page explains how to use EventBridge Scheduler to start execution of a Step Functions state machine on a schedule.

Topics

- [Set up the execution role](#)
- [Create a schedule](#)
- [Related resources](#)

Set up the execution role

When you create a new schedule, EventBridge Scheduler must have permission to invoke its target API operation on your behalf. You grant these permissions to EventBridge Scheduler using an *execution role*. The permission policy you attach to your schedule's execution role defines the required permissions. These permissions depend on the target API you want EventBridge Scheduler to invoke.

When you use the EventBridge Scheduler console to create a schedule, as in the following procedure, EventBridge Scheduler automatically sets up an execution role based on your selected target. If you want to create a schedule using one of the EventBridge Scheduler SDKs, the AWS CLI, or AWS CloudFormation, you must have an existing execution role that grants the permissions EventBridge Scheduler requires to invoke a target. For more information about manually setting up an execution role for your schedule, see [Setting up an execution role](#) in the *EventBridge Scheduler User Guide*.

Create a schedule

To create a schedule by using the console

1. Open the Amazon EventBridge Scheduler console at <https://console.aws.amazon.com/scheduler/home>.
2. On the **Schedules** page, choose **Create schedule**.
3. On the **Specify schedule detail** page, in the **Schedule name and description** section, do the following:
 - a. For **Schedule name**, enter a name for your schedule. For example, **MyTestSchedule**.
 - b. (Optional) For **Description**, enter a description for your schedule. For example, **My first schedule**.
 - c. For **Schedule group**, choose a schedule group from the dropdown list. If you don't have a group, choose **default**. To create a schedule group, choose **create your own schedule**.

You use schedule groups to add tags to groups of schedules.

4. • Choose your schedule options.

Occurrence	Do this...	
<p>One-time schedule</p> <p>A one-time schedule invokes a target only once at the date and time that you specify.</p>	<p>For Date and time, do the following:</p> <ul style="list-style-type: none">• Enter a valid date in YYYY/MM/DD format.• Enter a timestamp in 24-hour hh:mm format.• For Timezone, choose the timezone.	

Occurrence	Do this...	
<p>Recurring schedule</p> <p>A recurring schedule invokes a target at a rate that you specify using a cron expression or rate expression.</p>	<p>a. For Schedule type, do one of the following:</p> <ul style="list-style-type: none">• To use a cron expression to define the schedule, choose Cron-based schedule and enter the cron expression.• To use a rate expression to define the schedule, choose Rate-based schedule and enter the rate expression. <p>For more information about cron and rate expressions, see Schedule types on EventBridge Scheduler in the <i>Amazon EventBridge Scheduler User Guide</i>.</p> <p>b. For Flexible time window, choose Off to turn off the option, or choose one of the pre-defined time windows. For example, if you choose 15 minutes and you set a recurring schedule to invoke its target once every hour, the schedule runs within</p>	

Occurrence	Do this...	
	15 minutes after the start of every hour.	

5. (Optional) If you chose **Recurring schedule** in the previous step, in the **Timeframe** section, do the following:
 - a. For **Timezone**, choose a timezone.
 - b. For **Start date and time**, enter a valid date in YYYY/MM/DD format, and then specify a timestamp in 24-hour hh:mm format.
 - c. For **End date and time**, enter a valid date in YYYY/MM/DD format, and then specify a timestamp in 24-hour hh:mm format.
6. Choose **Next**.
7. On the **Select target** page, choose the AWS API operation that EventBridge Scheduler invokes:
 - a. Choose **AWS Step Functions StartExecution**.
 - b. In the **StartExecution** section, select a state machine or choose **Create new state machine**.

Currently, you can't run Synchronous Express workflows on a schedule.

- c. Enter a JSON payload for the execution. Even if your state machine doesn't require any JSON payload, you must still include input in JSON format as shown in the following example.

```
{
  "Comment": "sampleJSONData"
}
```

8. Choose **Next**.
9. On the **Settings** page, do the following:
 - a. To turn on the schedule, under **Schedule state**, toggle **Enable schedule**.
 - b. To configure a retry policy for your schedule, under **Retry policy and dead-letter queue (DLQ)**, do the following:
 - Toggle **Retry**.

- For **Maximum age of event**, enter the maximum **hour(s)** and **min(s)** that EventBridge Scheduler must keep an unprocessed event.
- The maximum time is 24 hours.
- For **Maximum retries**, enter the maximum number of times EventBridge Scheduler retries the schedule if the target returns an error.

The maximum value is 185 retries.

With retry policies, if a schedule fails to invoke its target, EventBridge Scheduler re-runs the schedule. If configured, you must set the maximum retention time and retries for the schedule.

- c. Choose where EventBridge Scheduler stores undelivered events.

Dead-letter queue (DLQ) option	Do this...	
Don't store	Choose None .	
Store the event in the same AWS account where you're creating the schedule	a. Choose Select an Amazon SQS queue in my AWS account as a DLQ . b. Choose the Amazon Resource Name (ARN) of the Amazon SQS queue.	
Store the event in a different AWS account from where you're creating the schedule	a. Choose Specify an Amazon SQS queue in other AWS accounts as a DLQ . b. Enter the Amazon Resource Name (ARN) of the Amazon SQS queue.	

- d. To use a customer managed key to encrypt your target input, under **Encryption**, choose **Customize encryption settings (advanced)**.

If you choose this option, enter an existing KMS key ARN or choose **Create an AWS KMS key** to navigate to the AWS KMS console. For more information about how EventBridge Scheduler encrypts your data at rest, see [Encryption at rest](#) in the *Amazon EventBridge Scheduler User Guide*.

- e. To have EventBridge Scheduler create a new execution role for you, choose **Create new role for this schedule**. Then, enter a name for **Role name**. If you choose this option, EventBridge Scheduler attaches the required permissions necessary for your templated target to the role.

10. Choose **Next**.

11. In the **Review and create schedule** page, review the details of your schedule. In each section, choose **Edit** to go back to that step and edit its details.

12. Choose **Create schedule**.

You can view a list of your new and existing schedules on the **Schedules** page. Under the **Status** column, verify that your new schedule is **Enabled**.

To confirm that EventBridge Scheduler invoked the state machine, check the [state machine's Amazon CloudWatch logs](#).

Related resources

For more information about EventBridge Scheduler, see the following:

- [EventBridge Scheduler User Guide](#)
- [EventBridge Scheduler API Reference](#)
- [EventBridge Scheduler Pricing](#)

Standard and Express Workflow executions in the console

When you create a state machine, you select a **Type** of either **Standard** or **Express**. The default **Type** for state machines is **Standard**. A state machine whose **Type** is **Standard** is called a *Standard workflow* and a state machine whose **Type** is **Express** is called an *Express workflow*.

For both Standard and Express workflows, you define your state machine using the [Amazon States Language](#). Your state machine executions will behave differently depending on the **Type** that you select.

⚠ Important

The **Type** you choose can't be changed after you create the state machine.

For more information about Standard and Express workflows, see [Standard vs. Express Workflows](#).

The history of Standard workflow executions are recorded in Step Functions, while the history of Express workflow executions aren't recorded in Step Functions. To record the history of an Express workflow execution, you must configure it to send logs to Amazon CloudWatch. For more information, see [Logging using CloudWatch Logs](#).

Once logging is configured on an Express workflow, you can view its executions in the Step Functions console. The console experience to view Express workflow executions and Standard workflow executions is similar, except for the following differences and limitations.

📘 Note

Because execution data for Express workflows are displayed using CloudWatch Logs Insights, scanning the logs will incur charges. By default, your log group only lists executions completed in the last three hours. If you specify a larger time range that includes more execution events, your costs will increase. For more information, see **Vended Logs** under the **Logs** tab on the [CloudWatch Pricing page](#) and [Logging using CloudWatch Logs](#).

Contents

- [Console experience differences](#)
- [Considerations and limitations for viewing Express workflow executions](#)

Console experience differences

For all Standard and Express workflows, you can view details, such as the state machine and its IAM role ARN, on the *State machine detail* page in the Step Functions console.

On the *State machine detail* page, you can also see a list of your state machine's execution histories under the **Executions** tab. Use the Filter executions by property or value box to search for a specific execution, [version](#), or [alias](#) of the chosen state machine. Use the All dropdown to filter execution

histories by their status. You can also choose an execution history and select the **View details** button to open its *Execution details* page.

Standard workflows

The execution histories for Standard Workflows are always available for executions completed in the last 90 days.

redriveInlineMap

Edit
Actions ▾
Start execution

Details

<p>Arn</p> <p>arn:aws:states:us-east-1:123456789012:stateMachine:redriveInlineMap</p> <p>IAM role ARN</p> <p>arn:aws:iam::123456789012:role/service-role/StepFunctions-redriveInlineMap-role-sh73cx890</p>	<p>Type</p> <p>Standard</p> <p>Status</p> <p>Active</p> <p>Creation date</p> <p><u>Oct 8, 2023, 13:48:02 (UTC-08:00)</u></p>
--	---

Executions
Logging
Definition
Aliases
Versions
Tags

Executions (1/6)

↻
View details
Stop execution
Redrive
Start execution

6 matches
< 1 >
⚙️

Name	Status	Started	End Time
<input checked="" type="radio"/> redriveInlineMap-6	⊗ Failed	<u>Oct 19, 2023, 14:16:07 (UTC-08:00)</u>	<u>Oct 19, 2023, 14:16:10 (UTC-08:00)</u>
<input type="radio"/> redriveInlineMap-5	✔ Succeeded	<u>Oct 19, 2023, 08:50:51 (UTC-08:00)</u>	<u>Oct 19, 2023, 11:29:23 (UTC-08:00)</u>
<input type="radio"/> redriveInlineMap-4	⊗ Failed	<u>Oct 19, 2023, 08:48:15 (UTC-08:00)</u>	<u>Oct 19, 2023, 08:48:26 (UTC-08:00)</u>
<input type="radio"/> redriveInlineMap-3	✔ Succeeded	<u>Oct 8, 2023, 13:53:21 (UTC-08:00)</u>	<u>Oct 8, 2023, 13:55:11 (UTC-08:00)</u>
<input type="radio"/> redriveInlineMap-2	⊗ Failed	<u>Oct 8, 2023, 13:52:23 (UTC-08:00)</u>	<u>Oct 8, 2023, 13:52:23 (UTC-08:00)</u>
<input type="radio"/> redriveInlineMap-1	⊗ Failed	<u>Oct 8, 2023, 13:48:57 (UTC-08:00)</u>	<u>Oct 8, 2023, 13:48:57 (UTC-08:00)</u>

Express workflows

To display execution history for Express workflows, the Step Functions console retrieves log data gathered through a CloudWatch Logs log group.

You must also enable the new console experience to view Express workflow executions. To do this, choose the **Enable** button displayed inside the banner on the **Executions** tab. Once you choose this button, it won't appear again.

Tip

To switch between enabling or disabling the console experience, use the **Enable express execution history** toggle button.

The histories for executions completed in the last three hours are available by default. You can adjust this time range or specify a custom range. If you specify a larger time range that includes more execution events, the cost to scan the logs will increase. For more information, see **Vended Logs** under the **Logs** tab on the [CloudWatch Pricing page](#) and [Logging using CloudWatch Logs](#).

The screenshot displays the AWS Step Functions console for an Express State Machine named `ExpressStateMachineForTextProcessing-UaZFv1uprIT`. At the top, there are buttons for `Edit`, `Actions`, and `Start execution`. The `Details` section shows the ARN `arn:aws:states:us-east-1:123456789012:stateMachine:ExpressStateMachineForTextProcessing-UaZFv1uprIT` and the IAM role ARN `arn:aws:iam::123456789012:role/StepFunctionsSample-ExpressSQS-StatesExecutionRole-1XKDX1B5V7ICB`. The state machine is of type `Express` and is currently `ACTIVE`. It was created on `Aug 11, 2022 10:53:22.441`.

Below the details, there are tabs for `Executions`, `Monitoring`, `Logging`, `Definition`, `Aliases`, `Versions`, and `Tags`. The `Executions` tab is active, showing a list of executions. The list has a search bar, a filter dropdown set to `All`, and a time range selector set to `Last 3 hours`. There is `1 match` shown. The execution table has columns for `Name`, `Status`, `Started`, and `End Time`. One execution is listed with the name `ExpressStateMachineForTextProcessing-1:22d01...`, a status of `Failed`, and a start time of `May 19, 2023 05:59:55.628 PM PDT`.

Considerations and limitations for viewing Express workflow executions

When viewing Express workflow executions on the Step Functions console, keep in mind the following considerations and limitations.

- [Availability of Express workflow execution details relies on Amazon CloudWatch Logs](#)
- [Partial Express workflow execution details are available if logging level is ERROR or FATAL](#)
- [State machine definition of an older execution can't be viewed once it has been updated](#)

Availability of Express workflow execution details relies on Amazon CloudWatch Logs

Note

If you don't enable the new console experience to view Express workflow executions, the execution histories and their corresponding execution details are unavailable in the Step Functions console. To enable the new console experience, choose the **Enable** button displayed inside the banner on the **Executions** tab.

For Express workflows, their execution history and detailed execution information are gathered through CloudWatch Logs Insights. This information is kept in the CloudWatch Logs log group that you specify when you create the state machine. The state machine's execution history is shown under the **Executions** tab on the Step Functions console. Detailed information about each execution of the state machine is displayed on the **Execution details** page for the chosen execution.

Warning

If you delete the CloudWatch Logs for an Express workflow, it won't be listed under the **Executions** tab.

We recommend that you use the default log level of **ALL** for logging all execution event types. You can update the log level as required for your existing state machines when you edit them. For more information, see [Logging using CloudWatch Logs](#) and [Log levels](#).

Partial Express workflow execution details are available if logging level is ERROR or FATAL

By default, the logging level for Express workflow executions is set to **ALL**. If you change the log level, the execution histories and execution details for completed executions won't be affected. However, all new executions will emit logs based on the updated log level. For more information, see [Logging using CloudWatch Logs](#) and [Log levels](#).

For example, if you change the log level from **ALL** to either **ERROR** or **FATAL**, the **Executions** tab on the Step Functions console only lists failed executions. In the **Event view** tab, the console shows only the event details for the state machine steps that failed.

We recommend that you use the default log level of **ALL** for logging all execution event types. You can update the log level as required for your existing state machines when you edit the state machine.

State machine definition of an older execution can't be viewed once it has been updated

State machine definitions for past executions aren't stored for Express workflows. If you change the state machine definition, you can only view the state machine definition for executions using the most current definition.

For example, if you remove one or more steps from your state machine definition, Step Functions detects a mismatch between the definition and prior execution events. Because previous definitions are not stored for Express workflows, Step Functions can't display the state machine definition for executions run on an earlier version of the state machine definition. As a result, the **Execution input & output**, **Definition**, **Graph view**, and **Table view** tabs are unavailable for executions run on previous versions of a state machine definition.

Viewing and debugging executions on the Step Functions console

The *Execution Details* page on the Step Functions console presents information about past and in-progress state machine executions for Standard and Express Workflows. This information is shown in a dashboard format. For example, you can find the state machine's Amazon States Language definition, its execution status, ARN, and total number of state transitions. You can also view the execution details for any individual state in the state machine.

Contents

- [Execution Details page – Interface overview](#)
 - [Execution summary](#)
 - [Error message](#)
 - [View mode](#)
 - [Step details](#)
 - [Events](#)
- [Tutorial: Examining state machine executions using the Step Functions console](#)
 - [Step 1: Create and test the required Lambda functions](#)
 - [Step 2: Create and execute the state machine](#)
 - [Step 3: View the state machine execution details](#)

- [Step 4: Explore the different View modes](#)

Execution Details page – Interface overview

You can find the details for all your in-progress and past state machine executions for both Standard and Express Workflows on the *Execution Details* page. If you specified an execution ID while starting your execution, this page is titled with that execution ID. Otherwise, it's titled with the unique execution ID that Step Functions automatically generates for you.

Besides the execution metrics, the *Execution Details* page provides the following options for managing your state machine and its execution:

Button	Choose this button to:
Edit state machine	Edit your state machine's Amazon States Language definition.
New execution	Start a new execution of your state machine.
Actions	<p>Provides the following options to choose from:</p> <ul style="list-style-type: none"> • Stop execution – Stop an in-progress execution. This option is unavailable for completed executions. • Redrive – Redrive executions of Standard Workflows that didn't complete successfully in the last 14 days. These include failed, aborted, or timed out executions. For more information, see Redriving executions. • Export – Export the execution details in JSON format to share it with others or perform offline analysis. • Send feedback – Share feedback about the interface.

 Viewing executions started with a version or alias

You can also view the executions started with a version or an alias in the Step Functions console. For more information, see [Listing executions for versions and aliases](#).

The *Execution Details* console page contains the following sections:

Execution: retriesRedrives-1

[Edit state machine](#) [New execution](#) [Actions](#) ▾

1

Details | Execution input and output | Definition

Execution status
⊗ Failed

Redrive details
 Redrive #1 completed

Redrive count [Info](#)
 1

Execution type
 Standard

Execution ARN
 arn:aws:states:us-east-1:123456789012:execution:retriesRedrives:retriesRedrives-1

State transitions [Learn more](#) [↗](#)
 14

Execution Logs [Learn more](#) [↗](#)
[CloudWatch Logs](#) [↗](#)

Start time
 Oct 18, 2023, 20:14:29.353 (UTC-07:00)

Last redrive time
 Oct 18, 2023, 20:14:47.040 (UTC-07:00)

End time
 Oct 18, 2023, 20:14:55.006 (UTC-07:00)

Duration [Info](#)
 00:00:25.653

Alias
 -

Version
 -

⊗ Error in step: Generate random number. [View step details](#) 2

▶ Cause

[Recover](#) ▾

Graph view | Table view 3

Graph view [Actions](#) ▾

```

graph TD
  Start((Start)) --> Parallel[Parallel  
2 Branches]
  Parallel --> Random[Generate random number]
  Parallel --> Wait[Wait 5 sec]
  Random --> End((End))
  Wait --> End
  style Random fill:#f8d7da,stroke:#c0392b
  style Wait fill:#d4edda,stroke:#28a745
  
```

Step details

Choose a step to view its details.

Legend: ⌚ In progress ⊗ Failed ⚠ Caught error ⊖ Canceled ✔ Succeeded

Events (37) 4

< 1 >

ID	Type	Step	Resource	Redrive attempt	Started After	Timestamp
▶ 1	ExecutionStarted			-	0	Oct 18, 2023, 20:14:29.353 (UTC-07:00)
▶ 2	ParallelStateEntered	Parallel		-	00:00:00.032	Oct 18, 2023, 20:14:29.385 (UTC-07:00)
▶ 3	ParallelStateStarted	Parallel		-	00:00:00.032	Oct 18, 2023, 20:14:29.385 (UTC-07:00)
▶ 4	TaskStateEntered	Generate random number		-	00:00:00.032	Oct 18, 2023, 20:14:29.385 (UTC-07:00)
▶ 5	TaskScheduled	Generate random number	Lambda Log group	-	00:00:00.032	Oct 18, 2023, 20:14:29.385 (UTC-07:00)
▶ 6	WaitStateEntered	Wait 5 sec		-	00:00:00.032	Oct 18, 2023, 20:14:29.385 (UTC-07:00)
▶ 7	TaskStarted	Generate random number		-	00:00:00.096	Oct 18, 2023, 20:14:29.449 (UTC-07:00)
▶ 8	⊗ TaskFailed	Generate random number		-	00:00:00.163	Oct 18, 2023, 20:14:29.516 (UTC-07:00)
▶ 9	TaskScheduled	Generate random number	Lambda Log group	-	00:00:01.236	Oct 18, 2023, 20:14:30.589 (UTC-07:00)

1. [Execution summary](#)
2. [Error message](#)
3. [View mode](#)
4. [Step details](#)
5. [Events](#)

Execution summary

The *Execution summary* section appears at the top of the *Execution Details* page. This section provides you with an overview of the execution details of your workflow. This information is divided between the following three tabs:

Details

Shows information, such as the execution's status, ARN, and timestamps for execution start and end time. You can also view the total count of **State transitions** that occurred while running the state machine execution. You can also view the links for **X-Ray trace map** and Amazon CloudWatch **Execution Logs** if you enabled tracing or logs for your state machine.

If your state machine execution was started by another state machine, you can view the link for the parent state machine on this tab.

If your state machine execution was [redriven](#), this tab displays redrive related information, for example **Redrive count**.

Execution input and output

Shows the state machine execution input and output side-by-side.

Definition

Shows the state machine's Amazon States Language definition.

Error message

If your state machine execution failed, the *Execution Details* page displays an error message. Choose **Cause** or **View step details** in the error message to view the reason for execution failure or the step that caused the error.

If you choose **View step details**, Step Functions highlights the step that caused the error in the [Step details](#), [Graph view](#), and [Table view](#) tabs. If the step is a Task, Map, or Parallel state for which

you've defined retries, the **Step details** pane displays the **Retry** tab for the step. Additionally, if you've redriven the execution, you can see the retries and redrive execution details in the **Retries & redrives** tab of the **Step details** pane.

From the **Recover** dropdown button on this error message, you can either redrive your unsuccessful executions or start a new execution. For more information, see [Redriving executions](#).

Details	Execution input and output	Definition
<p>Execution status</p> <p>✘ Failed</p> <p>Execution type</p> <p>Standard</p> <p>Execution ARN</p> <p> arn:aws:states:us-east-1:123456789012:execution:retriesRedrives:retriesRedrives-1</p> <p>State transitions Learn more </p> <p>8</p> <p>Execution Logs Learn more </p> <p>CloudWatch Logs </p>		<p>Start time</p> <p>Oct 18, 2023, 22:41:41.283 (UTC-07:00)</p> <p>End time</p> <p>Oct 18, 2023, 22:41:49.495 (UTC-07:00)</p> <p>Duration</p> <p>00:00:08.212</p> <p>Alias</p> <p>-</p> <p>Version</p> <p>-</p>

✘ **Error in step: Generate random number.** [View step details](#)

▶ Cause

Recover ▼

View mode

The *View mode* section contains two different visualizations for your state machine. You can choose to view a graphic representation of the workflow, a table outlining the states in your workflow, or a list of the events associated with your state machine's execution:

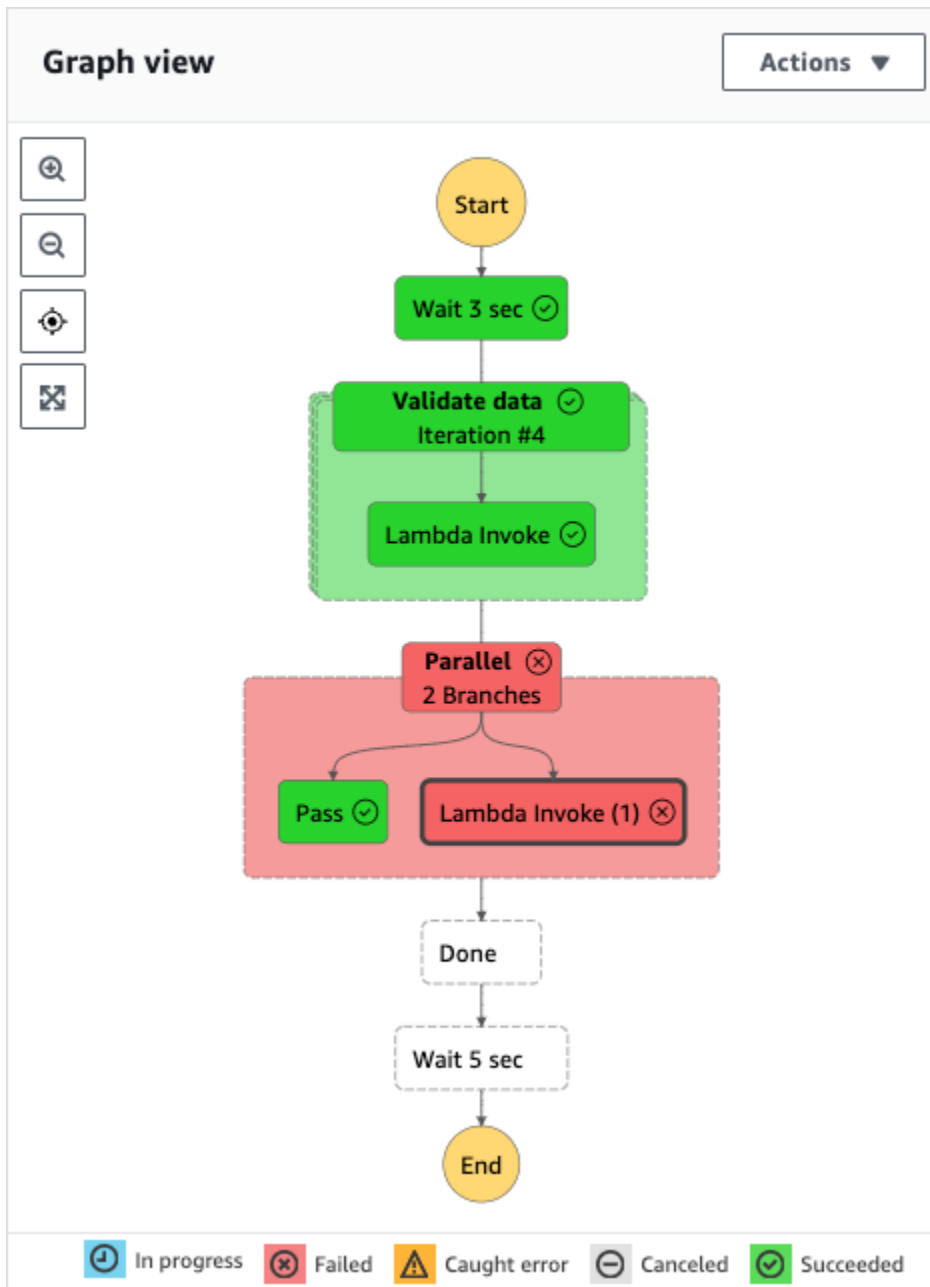
Note

Choose a tab to view its contents.

Graph view

The **Graph view** mode displays a graphical representation of your workflow. A legend is included at the bottom that indicates the execution status of the state machine. It also contains

buttons that let you zoom in, zoom out, center align the full workflow, or view the workflow in full-screen mode.



From this view, you can choose any step in your workflow to view details about its execution in the [Step details](#) component. When you chose a step in the **Graph view**, the **Table view** also shows that step. This is true in reverse as well. If you choose a step from **Table view**, the **Graph view** shows the same step.

If your state machine contains a Map state, Parallel state, or both, you can view their names in the workflow in the **Graph view**. In addition, for the Map state, the **Graph view** lets you move

across different iterations of the **Map** state execution data. For example, if your **Map** state has five iterations and you want to view the execution data for the third and fourth iterations, do the following:

1. Choose the **Map** state whose iteration data you want to view.
2. From **Map iteration viewer**, choose **#2** from the dropdown list for third iteration. This is because iterations are counted from zero. Likewise, choose **#3** from the dropdown list for the fourth iteration of the **Map** state.

Alternatively, use the



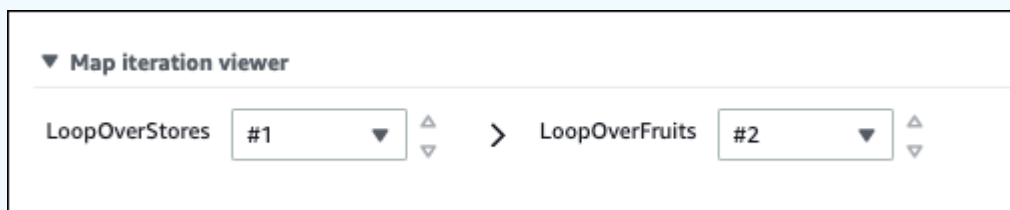
and



controls to move between different iterations of the **Map** state.

Note

If your state machine contains nested Map states, the dropdown lists for the parent and child Map state iterations will be displayed as shown in the following example:



3. (Optional) If one or more of your **Map** state iterations failed to execute, or the execution was stopped, you can view its data by choosing those iteration numbers under **Failed** or **Aborted** in the dropdown list.

Finally, you can use the **Export** and **Layout** buttons to export the workflow graph as an SVG or PNG image. You can also switch between horizontal and vertical views of your workflow.

Table view

The **Table view** mode displays a tabular representation of the states in your workflow. In this *View mode*, you can see the details of each state that was executed in your workflow, including its name, the name of any resource it used (such as an AWS Lambda function), and if the state executed successfully.

From this view, you can choose any state in your workflow to view details about its execution in the [Step details](#) component. When you chose a step in the **Table view**, the **Graph view** also shows that step. This is true in reverse as well. If you choose a step from **Graph view**, the **Table view** shows the same step.

You can also limit the amount of data displayed in the **Table view** mode by applying filters to the view. You can create a filter for a specific property, such as **Status** or **Redrive attempt**. For more information, see [Tutorial: Examining state machine executions using the Step Functions console](#).

Table view							
		Data flow simulator					
<input type="text" value="Filter by properties or search by keyword"/>				<input type="text" value="Filter by a date and time range"/>			
	Name	Type	Status	Resource	Duration	Timeline	Started after
<input type="radio"/>	<input type="checkbox"/> Parallel	Parallel	Succeeded	-	32 sec		35 ms
<input type="radio"/>	<input type="checkbox"/> #1	ParallelBranch	Succeeded	-	32 sec		147 ms
<input type="radio"/>	<input type="checkbox"/> Lambd	Task	Succeeded 	Lambda ...	31 sec		147 ms
<input type="radio"/>	<input type="checkbox"/> Wait	Wait	Succeeded	-	1 sec		31 sec
<input type="radio"/>	<input type="checkbox"/> Choice	Choice	Succeeded	-	0 ms		32 sec
<input type="radio"/>	<input type="checkbox"/> Pass	Pass	Succeeded	-	0 ms		32 sec
<input type="radio"/>	<input type="checkbox"/> #0	ParallelBranch	Succeeded	-	9 sec		156 ms
<input type="radio"/>	<input type="checkbox"/> Map	Map	Succeeded	-	134 ms		156 ms
<input type="radio"/>	<input type="checkbox"/> #0	MapIteration	Succeeded	-	103 ms		156 ms
<input type="radio"/>	<input type="checkbox"/> #1	MapIteration	Succeeded	-	113 ms		156 ms
<input type="radio"/>	<input type="checkbox"/> #2	MapIteration	Succeeded	-	122 ms		156 ms
<input type="radio"/>	<input type="checkbox"/> #3	MapIteration	Succeeded	-	134 ms		156 ms
<input type="radio"/>	<input type="checkbox"/> Pass	Pass	Succeeded	-	0 ms		290 ms
<input type="radio"/>	<input type="checkbox"/> FailAct	Parallel	Caught error	-	5 sec		302 ms
<input type="radio"/>	<input type="checkbox"/> #0	ParallelBranch	Caught error	-	0 ms		405 ms
<input type="radio"/>	<input type="checkbox"/> Task	Task	Aborted	-	32 sec		405 ms
<input type="radio"/>	<input type="checkbox"/> #1	ParallelBranch	Caught error	-	0 ms		419 ms

By default, this mode displays the **Name**, **Type**, **Status**, **Resource**, and **Started After** columns. You can configure the columns you want to view using the **Preferences** dialog box. The selections that you make on this dialog box persist for future state machine executions until they are changed again.

If you add the **Timeline** column, the execution duration of each state is shown with respect to the runtime for the entire execution. This is displayed as a color-coded, linear timeline. This can help you identify any performance-related issues with a specific state's execution. The color-coded segments for each state on the timeline help you identify the state's execution status, such as in-progress, failed, or aborted.

For example, if you've defined execution retries for a state in your state machine, these retries are shown in the timeline. Red segments represent the failed `Retry` attempts, while light gray segments represent the `BackoffRate` between each `Retry` attempt.

	Name	Type	Status	Resource	Duration	Timeline	Started After
	LoopOverStr	Map	⊗ Failed	-	8 sec		69 ms
	#0	MapIteration	⊗ Failed	-	8 sec		69 ms
	GetList	Task	⊗ Failed ■■■■	Lambda ...	8 sec		69 ms
	#1	MapIteration	✔ Succeeded	-	1 sec		69 ms
	#2	MapIteration	⊖ Aborted	-	8 sec		69 ms
	GetList	Task	✔ Succeeded ■■■	Lambda ...	8 sec		69 ms
	#3	MapIteration	✔ Succeeded	-	5 sec		69 ms

If your state machine contains a `Map` state, `Parallel` state, or both, you can view their names in the workflow in **Table view**. For `Map` and `Parallel` states, the **Table view** mode displays the execution data for their iterations and parallel branches as nodes inside a tree view. You can choose each node in these states to view their individual details in the [Step details](#) section. For example, you can review the data for a specific **Map** state iteration that caused the state to fail. Expand the node for the **Map** state, and then view the status for each iteration in the **Status** column.

Step details

The *Step details* section opens up on the right when you choose a state in the **Graph view** or **Table view**. This section contains the following tabs, which provide you in-depth information about the selected state:

Input

Shows the input details of the selected state. If there is an error in the input, it is indicated with a



on the tab header. In addition, you can view the reason for the error in this tab.

You can also choose the **Advanced view** toggle button to see the input data transfer path as the data passed through the selected state. This lets you identify how your input was processed as one or more of the fields, such as `InputPath`, `Parameters`, `ResultSelector`, `OutputPath`, and `ResultPath`, were applied to the data.

Output

Shows the output of the selected state. If there is an error in the output, it is indicated with a



on the tab header. In addition, you can view the reason for the error in the this tab.

You can also choose the **Advanced view** toggle button to see the output data transfer path as the data passed through the selected state. This lets you identify how your input was processed as one or more of the fields, such as `InputPath`, `Parameters`, `ResultSelector`, `OutputPath`, and `ResultPath`, were applied to the data.

Details

Shows information, such as the state type, its execution status, and execution duration.

For Task states that use a resource, such as AWS Lambda, this tab provides links to the resource definition page and Amazon CloudWatch logs page for the resource invocation. It also shows values, if specified, for the Task state's `TimeoutSeconds` and `HeartbeatSeconds` fields.

For Map states, this tab shows you information regarding the total count of a Map state's iterations. Iterations are categorized as **Failed**, **Aborted**, **Succeeded**, or **InProgress**.

Definition

Shows the Amazon States Language definition corresponding to the selected state.

Retry

Note

This tab appears only if you've defined a `Retry` field in your state machine's `Task` or `Parallel` state.

Shows the initial and subsequent retry attempts for a selected state in its original execution attempt. For the initial and all the subsequent failed attempts, choose the



next to **Type** to view the **Reason** for failure that appears in a dropdown box. If the retry attempt succeeds, you can view the **Output** that appears in a dropdown box.

If you've redriven your execution, this tab header displays the name **Retries & redrives** and displays the retry attempt details for each redrive.

Events

Shows a filtered list of the events associated with the selected state in an execution. The information you see on this tab is a subset of the complete execution event history you see in the [Events](#) table.

Events

The **Events** table displays the complete history for the selected execution as a list of events spanning multiple pages. Each page contains up to 25 events. This section also displays the total event count, which can help you determine if you exceeded the maximum event history count of 25,000 events.

Events (109)						
<input type="text" value="Filter by properties or search by keyword"/>			<input type="text" value="Filter by a date and time range"/>		< 1 >	
ID ▲	Type	Step	Resource	Redrive attempt	Started After	Timestamp ▼
▶ 95	⊗ TaskStateAborted			#2	02:37:37.672	Oct 19, 2023, 11:28:28.958 (UTC-07:00)
▶ 96	⊗ ParallelStateFailed	Parallel		#2	02:37:37.672	Oct 19, 2023, 11:28:28.958 (UTC-07:00)
▶ 97	⊗ ExecutionFailed			#2	02:37:37.713	Oct 19, 2023, 11:28:28.999 (UTC-07:00)
▶ 98	⊖ ExecutionRedriven			#3	02:38:24.882	Oct 19, 2023, 11:29:16.168 (UTC-07:00)
▶ 99	⊙ TaskScheduled	Lambda Invoke (1)	Lambda Log group	#3	02:38:24.904	Oct 19, 2023, 11:29:16.190 (UTC-07:00)
▶ 100	⊖ TaskStarted	Lambda Invoke (1)		#3	02:38:24.985	Oct 19, 2023, 11:29:16.271 (UTC-07:00)
▶ 101	⊙ TaskSucceeded	Lambda Invoke (1)		#3	02:38:27.260	Oct 19, 2023, 11:29:18.546 (UTC-07:00)
▶ 102	⊖ TaskStateExited	Lambda Invoke (1)		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
▶ 103	⊙ ParallelStateSucceeded	Parallel		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
▶ 104	⊖ ParallelStateExited	Parallel		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
▶ 105	⊖ PassStateEntered	Done		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
▶ 106	⊖ PassStateExited	Done		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
▶ 107	⊙ WaitStateEntered	Wait 5 sec		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
▶ 108	⊖ WaitStateExited	Wait 5 sec		#3	02:38:32.345	Oct 19, 2023, 11:29:23.631 (UTC-07:00)
▶ 109	⊙ ExecutionSucceeded			#3	02:38:32.394	Oct 19, 2023, 11:29:23.680 (UTC-07:00)

By default, the results in the **Events** table are displayed in ascending order based on the **Timestamp** of the events. You can change the execution event history's sorting to descending order by clicking on the **Timestamp** column header.

In the **Events** table, each event is color-coded to indicate its execution status. For example, events that failed appear in red. To view additional details about an event, choose the



next to the event ID. Once open, the event details show the input, output, and resource invocation for the event.

In addition, in the **Events** table, you can apply filters to limit the execution event history results that are displayed. You can choose properties such as **ID**, or **Redrive attempt**. For more information, see [Tutorial: Examining state machine executions using the Step Functions console](#).

Tutorial: Examining state machine executions using the Step Functions console

In this tutorial, you will learn how to inspect the execution information displayed on the *Execution Details* page and view the reason for a failed execution. Then, you'll learn how to access different iterations of a Map state execution. Finally, you'll learn how to configure the columns on the **Table view** and apply suitable filters to view only the information of interest to you.

In this tutorial, you create a Standard type state machine, which obtains the price of a set of fruits. To do this, the state machine uses three AWS Lambda functions which return a random list of four fruits, the price of each fruit, and the average cost of the fruits. The Lambda functions are designed to throw an error if the price of the fruits is less than or equal to a threshold value.

Note

While the following procedure contains instructions for how to examine the details of a Standard workflow execution, you can also examine the details for Express workflow executions. For information about the differences between the execution details for Standard and Express workflow types, see [Standard and Express Workflow executions in the console](#).

Contents

- [Step 1: Create and test the required Lambda functions](#)
- [Step 2: Create and execute the state machine](#)
- [Step 3: View the state machine execution details](#)
- [Step 4: Explore the different View modes](#)

Step 1: Create and test the required Lambda functions

1. Open the [Lambda console](#) and then perform steps 1 through 4 in the [Step 1: Create a Lambda function](#) section. Make sure to name the Lambda function **GetListOfFruits**.
2. After you create your Lambda function, copy the function's Amazon Resource Name (ARN) displayed in the upper-right corner of the page. To copy the ARN, click the



The following is an example ARN, where *function-name* is the name of the Lambda function (in this case, GetListOfFruits):

```
arn:aws:lambda:us-east-1:123456789012:function:function-name
```

3. Copy the following code for the Lambda function into the **Code source** area of the **GetListOfFruits** page.

```
function getRandomSubarray(arr, size) {  
    var shuffled = arr.slice(0), i = arr.length, temp, index;
```



```

    while (i--) {
      index = Math.floor((i + 1) * Math.random());
      temp = shuffled[index];
      shuffled[index] = shuffled[i];
      shuffled[i] = temp;
    }
    return shuffled.slice(0, size);
  }
}

exports.handler = async function(event, context) {

  const fruits = ['Abiu', 'Açaí', 'Acerola', 'Ackee', 'African
cucumber', 'Apple', 'Apricot', 'Avocado', 'Banana', 'Bilberry', 'Blackberry', 'Blackcurrant', 'Jos

  const errorChance = 45;

  const waitTime = Math.floor( 100 * Math.random() );

  await new Promise( r => setTimeout(() => r(), waitTime));

  const num = Math.floor( 100 * Math.random() );
  // const num = 51;
  if (num <= errorChance) {
    throw(new Error('Error'));
  }

  return getRandomSubarray(fruits, 4);
};

```

4. Choose **Deploy**, and then choose **Test**, to deploy the changes and see the output of your Lambda function.
5. Create two additional Lambda functions, named **GetFruitPrice** and **CalculateAverage** respectively, with the following steps:
 - a. Copy the following code into the **Code source** area of the **GetFruitPrice** Lambda function:

```

exports.handler = async function(event, context) {

  const errorChance = 0;
  const waitTime = Math.floor( 100 * Math.random() );

  await new Promise( r => setTimeout(() => r(), waitTime));

```

```

const num = Math.floor( 100 * Math.random() );
if (num <= errorChance) {
    throw(new Error('Error'));
}

return Math.floor(Math.random()*100)/10;
};

```

- b. Copy the following code into the **Code source** area of the **CalculateAverage** Lambda function:

```

function getRandomSubarray(arr, size) {
    var shuffled = arr.slice(0), i = arr.length, temp, index;
    while (i-->0) {
        index = Math.floor((i + 1) * Math.random());
        temp = shuffled[index];
        shuffled[index] = shuffled[i];
        shuffled[i] = temp;
    }
    return shuffled.slice(0, size);
}

const average = arr => arr.reduce( ( p, c ) => p + c, 0 ) / arr.length;

exports.handler = async function(event, context) {
    const errors = [
        "Error getting data from DynamoDB",
        "Error connecting to DynamoDB",
        "Network error",
        "MemoryError - Low memory"
    ]

    const errorChance = 0;

    const waitTime = Math.floor( 100 * Math.random() );

    await new Promise( r => setTimeout(() => r(), waitTime));

    const num = Math.floor( 100 * Math.random() );
    if (num <= errorChance) {
        throw(new Error(getRandomSubarray(errors, 1)[0]));
    }
}

```

```
    return average(event);  
};
```

- c. Make sure to copy the ARNs of these two Lambda functions, and then **Deploy** and **Test** them.

Step 2: Create and execute the state machine

Use the [Step Functions console](#) to create a state machine that invokes the [Lambda functions you created in Step 1](#). In this state machine, three Map states are defined. Each of these Map states contains a Task state that invokes one of your Lambda functions. Additionally, a Retry field is defined in each Task state with a number of retry attempts defined for each state. If a Task state encounters a runtime error, it's executed again up to the number of retry attempts defined for that Task.

1. Open the [Step Functions console](#) and choose **Write your workflow in code**.

Important

Ensure that your state machine is under the same AWS account and Region as the Lambda function you created earlier.

2. For **Type**, keep the default selection of **Standard**.
3. Copy the following Amazon States Language definition and paste it under **Definition**. Make sure to replace the ARNs shown with those of the Lambda functions that you previously created.

```
{  
  "StartAt": "LoopOverStores",  
  "States": {  
    "LoopOverStores": {  
      "Type": "Map",  
      "Iterator": {  
        "StartAt": "GetListOfFruits",  
        "States": {  
          "GetListOfFruits": {  
            "Type": "Task",  
            "Resource": "arn:aws:states:::lambda:invoke",  
            "OutputPath": "$$.Payload",
```

```
    "Parameters": {
      "FunctionName": "arn:aws:lambda:us-
east-1:123456789012:function:GetListofFruits:$LATEST",
      "Payload": {
        "storeName.$": "$"
      }
    },
    "Retry": [
      {
        "ErrorEquals": [
          "States.ALL"
        ],
        "IntervalSeconds": 2,
        "MaxAttempts": 1,
        "BackoffRate": 1.3
      }
    ],
    "Next": "LoopOverFruits"
  },
  "LoopOverFruits": {
    "Type": "Map",
    "Iterator": {
      "StartAt": "GetFruitPrice",
      "States": {
        "GetFruitPrice": {
          "Type": "Task",
          "Resource": "arn:aws:states:::lambda:invoke",
          "OutputPath": "$.Payload",
          "Parameters": {
            "FunctionName": "arn:aws:lambda:us-
east-1:123456789012:function:GetFruitPrice:$LATEST",
            "Payload": {
              "fruitName.$": "$"
            }
          }
        },
        "Retry": [
          {
            "ErrorEquals": [
              "States.ALL"
            ],
            "IntervalSeconds": 2,
            "MaxAttempts": 3,
            "BackoffRate": 1.3
          }
        ]
      }
    }
  }
}
```

```

        ],
        "End": true
    }
}
},
"ItemsPath": "$",
"End": true
}
},
},
"ItemsPath": "$.stores",
"Next": "LoopOverStoreFruitsPrice",
"ResultPath": "$.storesFruitsPrice"
},
"LoopOverStoreFruitsPrice": {
    "Type": "Map",
    "End": true,
    "Iterator": {
        "StartAt": "CalculateAverage",
        "States": {
            "CalculateAverage": {
                "Type": "Task",
                "Resource": "arn:aws:states:::lambda:invoke",
                "OutputPath": "$.Payload",
                "Parameters": {
                    "FunctionName": "arn:aws:lambda:us-
east-1:123456789012:function:Calculate-average:$LATEST",
                    "Payload.$": "$"
                }
            },
            "Retry": [
                {
                    "ErrorEquals": [
                        "States.ALL"
                    ],
                    "IntervalSeconds": 2,
                    "MaxAttempts": 2,
                    "BackoffRate": 1.3
                }
            ],
            "End": true
        }
    }
},
"ItemsPath": "$.storesFruitsPrice",

```

```
        "ResultPath": "$.storesPriceAverage",
        "MaxConcurrency": 1
    }
}
```

4. Enter a name for your state machine. Keep the default selections for the other options on this page and choose **Create state machine**.
5. Open the page titled with your state machine name. Perform steps 1 through 4 in the [Step 4: Run the state machine](#) section, but use the following data as the execution input:

```
{
  "stores": [
    "Store A",
    "Store B",
    "Store C",
    "Store D"
  ]
}
```

Step 3: View the state machine execution details

On the page titled with your execution ID, you can review the results of your execution and debug any errors.

1. (Optional) Choose from the tabs displayed on the *Execution Details* page to see the information present in each of them. For example, to view the state machine input and its execution output, choose **Execution input & output** on the [Execution summary](#) section.
2. If your state machine execution failed, choose **Cause** or **Show step detail** on the error message. Details about the error are displayed in the [Step details](#) section. Notice that the step that caused the error, which is a Task state named **GetListofFruits**, is highlighted in the **Graph view** and **Table view**.

Note

Because the **GetListofFruits** step is defined inside a Map state, and the step failed to execute successfully, the **Status** of Map state step is displayed as **Failed**.

Step 4: Explore the different *View modes*

You can choose a preferred mode to view either the state machine workflow or the execution event history. Some of the tasks that you can perform in these *View modes* are as follows:

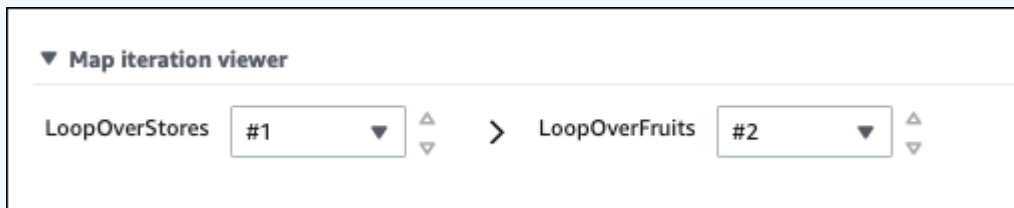
Graph view – Switch between different Map state iterations

If your **Map** state has five iterations and you want to view the execution details for the third and fourth iterations, do the following:

1. Choose the Map state that you want to view the iteration data for.
2. From **Map iteration viewer**, choose the iteration that you want to view. Iterations are counted from zero. To choose the third iteration out of five, choose **#2** from the dropdown list next to the **Map** state's name.

Note

If your state machine contains nested Map states, Step Functions displays the parent and child Map state iterations as two separate dropdown lists:



3. (Optional) If one or more of your Map state iterations failed to execute or was stopped in an aborted state, you can view details about the failed iteration. To see these details, choose the affected iteration numbers under **Failed** or **Aborted** in the dropdown list.

Table view – Switch between different Map state iterations

If your **Map** state has five iterations and you want to view the execution details for the iteration number three and four, do the following:

1. Choose the Map state for which you want to view the different iteration data.
2. In the tree view display of the Map state iterations, choose the row for iteration named **#2** for iteration number three. Similarly, choose the row named **#3** for iteration number four.

Table view – Configure the columns to display

Choose



Then, in the **Preferences** dialog box, choose the columns you want to display under **Select visible columns**.

By default, this mode displays the **Name**, **Type**, **Status**, **Resource**, and **Started After** columns.

Table view – Filter the results

Limit the amount of information displayed by applying one or more filters based on a property, such as **Status**, or a date and time range. For example, to view the steps that failed execution, apply the following filter:

1. Choose **Filter by properties or search by keyword**, and then choose **Status** under **Properties**.
2. Under **Operators**, choose **Status =**.
3. Choose **Status = Failed**.
4. (Optional) Choose **Clear filters** to remove the applied filters.

Event view – Filter the results

Limit the amount of information displayed by applying one or more filters based on a property, such as **Type**, or a date and time range. For example, to view the Task state steps that failed execution, apply the following filter:

1. Choose **Filter by properties or search by keyword**, and then choose **Type** under **Properties**.
2. Under **Operators**, choose **Type =**.
3. Choose **Type = TaskFailed**.
4. (Optional) Choose **Clear filters** to remove the applied filters.

Event view – Inspect a TaskFailed event detail

Choose the



next to the ID of a **TaskFailed** event to inspect its details, including input, output, and resource invocation that appear in a dropdown box.

Redriving executions

You can use `redrive` to restart executions of [Standard Workflows](#) that didn't complete successfully in the last 14 days. These include failed, aborted, or timed out executions.

When you `redrive` an execution, it continues the failed execution from the unsuccessful step and uses the same input. Step Functions preserves the results and execution history of the successful steps, and these steps aren't rerun when you `redrive` an execution. For example, say that your workflow contains two states: a [Pass](#) state followed by a [Task state](#) state. If your workflow execution fails at the Task state, and you `redrive` the execution, the execution reschedules and then reruns the Task state.

Redriven executions use the same state machine definition and execution ARN that was used for the original execution attempt. If your original execution attempt was associated with a [version](#), [alias](#), or both, the redriven execution is associated with the same version, alias, or both. Even if you update your alias to point to a different version, the redriven execution continues to use the version associated with the original execution attempt. Because redriven executions use the same state machine definition, you must start a new execution if you update your state machine definition.

When you `redrive` an execution, the state machine level timeout, if defined, is reset to 0. For more information about state machine level timeout, see [TimeoutSeconds](#).

Execution `redrives` are considered as state transitions. For information about how state transitions affect billing, see [Step Functions Pricing](#).

Topics

- [Redrive eligibility for unsuccessful executions](#)
- [Redrive behavior of individual states](#)
- [IAM permission to redrive an execution](#)
- [Redriving executions in console](#)
- [Redriving executions using API](#)
- [Examining redriven executions](#)
- [Retry behavior of redriven executions](#)

Redrive eligibility for unsuccessful executions

You can `redrive` executions if your original execution attempt meets the following conditions:

- You started the execution on or after November 15, 2023. Executions that you started prior to this date aren't eligible for redrive.
- The execution status isn't SUCCEEDED.
- The workflow execution hasn't exceeded the redrivable period of 14 days. Redrivable period refers to the time during which you can redrive a given execution. This period starts from the day a state machine completes its execution.
- The workflow execution hasn't exceeded the maximum open time of one year. For information about state machine execution quotas, see [Quotas related to state machine executions](#).
- The execution event history count is less than 24,999. Redriven executions append their event history to the existing event history. Make sure your workflow execution contains less than 24,999 events to accommodate the ExecutionRedriven history event and at least one other history event.

Redrive behavior of individual states

Depending on the state that failed in your workflow, the redrive behavior for all unsuccessful states varies. The following table describes the redrive behavior for all the states.

State name	Redrive execution behavior
Pass	If a preceding step fails or the state machine times out, the Pass state is exited and isn't executed on redrive.
Task state	Schedules and starts the Task state again. When you redrive an execution that reruns a Task state, the <code>TimeoutSeconds</code> for the state, if defined, is reset to 0. For more information about timeout, see Task state .
Choice	Reevaluates the Choice state rules.
Wait	If the state specifies <code>Timestamp</code> or <code>TimestampPath</code> that refers to a timestamp in the past, redrive causes the Wait state to

State name	Redrive execution behavior
	be exited and enters the state specified in the Next field.
Succeed	Doesn't redrive state machine executions that enter the Succeed state.
Fail	Reenters the Fail state and fails again.
Parallel	<p>Reschedules and redrives only those branches that failed or aborted.</p> <p>If the state failed because of a States.DataLimitExceeded error, the Parallel state is rerun, including the branches that were successful in the original execution attempt.</p>
Inline Map state	<p>Reschedules and redrives only those iterations that failed or aborted.</p> <p>If the state failed because of a States.DataLimitExceeded error, the Inline Map state is rerun, including the iterations that were successful in the original execution attempt.</p>
Distributed Map state	<p>redrives the unsuccessful child workflow executions in a Map Run. For more information, see Redriving Map Runs.</p> <p>If the state failed because of a States.DataLimitExceeded error, the Distributed Map state is rerun. This includes the child workflows that were successful in the original execution attempt.</p>

IAM permission to redrive an execution

Step Functions needs appropriate permission to redrive an execution. The following IAM policy example grants the least privilege required to your state machine for redriving an execution.

Remember to replace the *italicized* text with your resource-specific information.

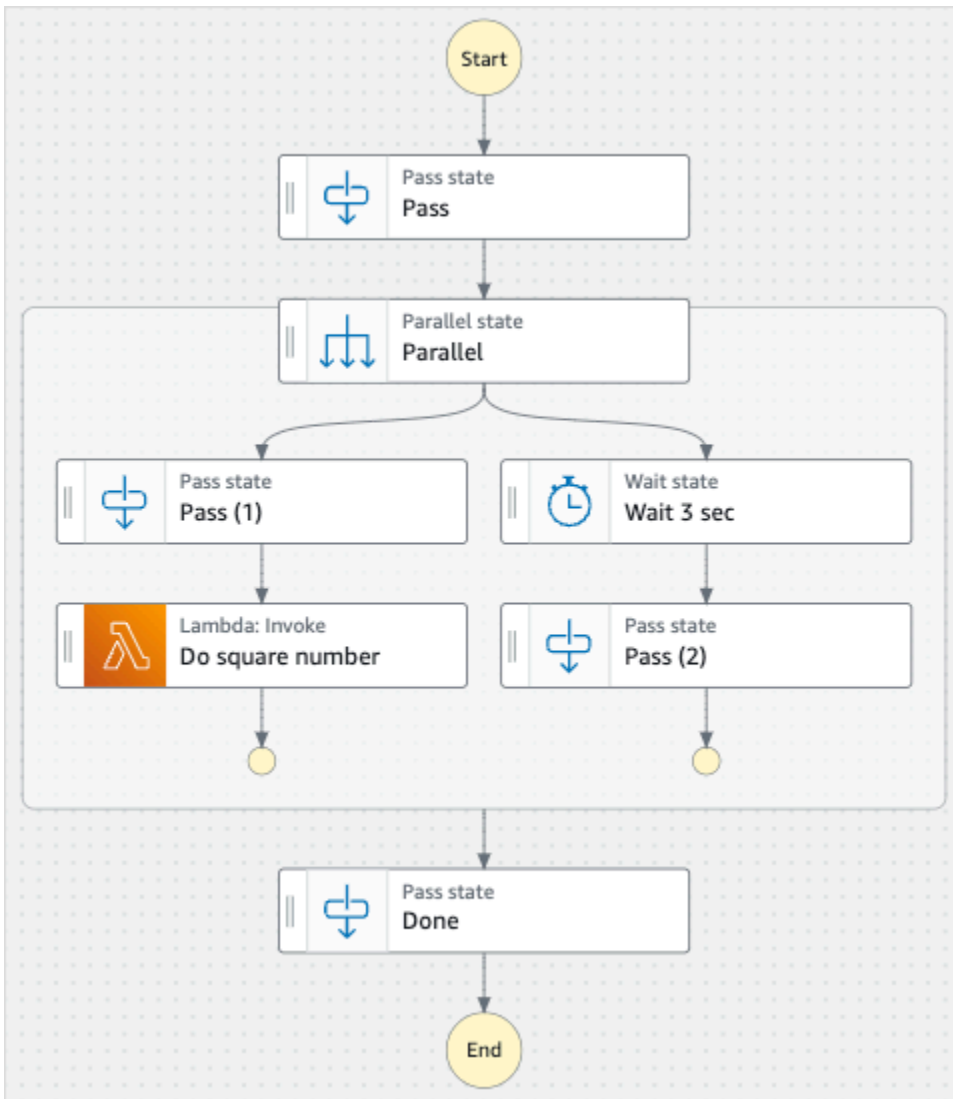
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:RedriveExecution"
      ],
      "Resource": "arn:aws:states:us-
east-2:123456789012:execution:myStateMachine:*"
    }
  ]
}
```

For an example of the permission you need to redrive a Map Run, see [Example of IAM policy for redriving a Distributed Map](#).

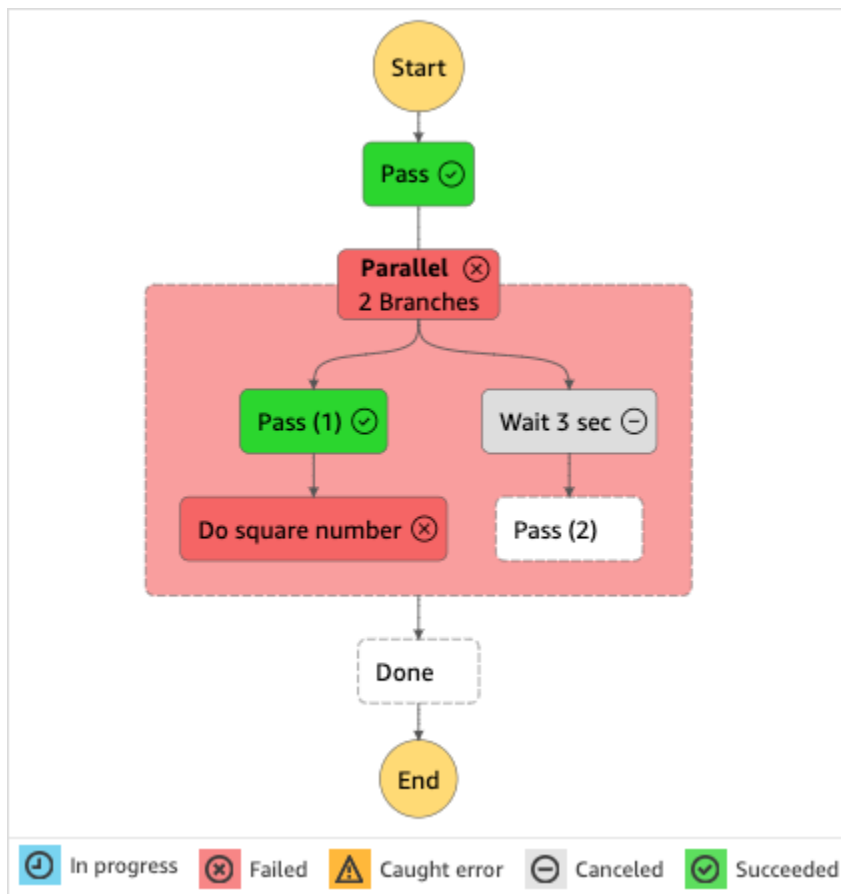
Redriving executions in console

You can redrive [eligible](#) executions from the Step Functions console.

For example, say that the following image represents the workflow graph of your state machine.



Imagine you run this state machine. The following image shows the graph for the state machine execution.



As shown in this image, the **Lambda Invoke** step named **Do square number** inside the **Parallel** state has returned an error. This caused the **Parallel** state to fail. The branches whose execution were in progress or not started are stopped and the state machine execution fails.

To redrive an execution from the console

1. Open the [Step Functions console](#), and then choose an existing state machine that failed execution.
2. On the state machine detail page, under **Executions**, choose a failed execution instance.
3. Choose **Redrive**.
4. In the **Redrive** dialog box, choose **Redrive execution**.

Tip

If you're on the *Execution Details* page of a failed execution, do one of the following to redrive the execution:

- Choose **Recover**, and then select **Redrive from failure**.

- Choose **Actions**, and then select **Redrive**.

Notice that `redrive` uses the same state machine definition and ARN. It continues running the execution from the step that failed in the original execution attempt. In this example, it's the **Do square number** step and **Wait 3 sec** branch inside the **Parallel** state. After restarting the execution of these unsuccessful steps in the **Parallel** state, `redrive` will continue execution for the **Done** step.

5. Choose the execution to open the *Execution Details* page.

On this page, you can view the results of the redriven execution. For example, in the [Execution summary](#) section, you can see **Redrive count**, which represents the number of times an execution has been redriven. In the **Events** section, you can see the `redrive` related execution events appended to the events of the original execution attempt. For example, the `ExecutionRedriven` event.

Redriving executions using API

You can redrive [eligible](#) executions using the [RedriveExecution](#) API. This API restarts unsuccessful executions of Standard Workflows from the step that failed, aborted, or timed out.

In the AWS Command Line Interface (AWS CLI), run the following command to redrive an unsuccessful state machine execution. Remember to replace the *italicized* text with your resource-specific information.

```
aws stepfunctions redrive-execution --execution-arn arn:aws:states:us-east-2:123456789012:execution:myStateMachine:foo
```

Examining redriven executions

You can examine a redriven execution in the console or using the APIs: [GetExecutionHistory](#) and [DescribeExecution](#).

Examine redriven executions on console

1. Open the [Step Functions console](#), and then choose an existing state machine for which you've redriven an execution.
2. Open the *Execution Details* page.

On this page, you can view the results of the redriven execution. For example, in the [Execution summary](#) section, you can see **Redrive count**, which represents the number of times an execution has been redriven. In the **Events** section, you can see the redrive related execution events appended to the events of the original execution attempt. For example, the `ExecutionRedriven` event.

Examine redriven executions using APIs

If you've redriven a state machine execution, you can use one of the following APIs to view details about the redriven execution. Remember to replace the *italicized* text with your resource-specific information.

- `GetExecutionHistory` – Returns the history of the specified execution as a list of events. This API also returns the details about the redrive attempt of an execution, if available.

In the AWS CLI, run the following command.

```
aws stepfunctions get-execution-history --execution-arn arn:aws:states:us-east-2:123456789012:execution:myStateMachine:foo
```

- `DescribeExecution` – Provides information about a state machine execution. This can be the state machine associated with the execution, the execution input and output, execution redrive details, if available, and relevant execution metadata.


In the AWS CLI, run the following command.

```
aws stepfunctions describe-execution --execution-arn arn:aws:states:us-east-2:123456789012:execution:myStateMachine:foo
```


Retry behavior of redriven executions

If your redriven execution reruns a [Task state](#), [Parallel](#), or [Inline Map](#) state, for which you have defined retries, the retry attempt count for these states is reset to 0. This allows for the maximum number of attempts on redrive. For a redriven execution, you can track individual retry attempts of these states using the console.

To examine the individual retry attempts in the console

1. On the *Execution Details* page of the [Step Functions console](#), choose a state that was retried on redrive.
2. Choose the **Retries & redrives** tab.
3. Choose the  next to each retry attempt to view its details. If the retry attempt succeeded, you can view the results in **Output** that appears in a dropdown box.

The following image shows an example of the retries performed for a state in the original execution attempt and the redrives of that execution. In this image, three retries are performed in the original and redrive execution attempts. The execution succeeds in the fourth redrive attempt and returns an output of 16.

Input	Output	Details	Definition	Retries & redrives	Events
Type	Status	Resource		Duration	Time
▶ Original execution	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.151	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.139	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.164	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.149	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ Redrive #1	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.187	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.147	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.154	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.170	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ Redrive #2	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.206	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.184	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.188	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.219	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ Redrive #3	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.198	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.142	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.174	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▶ - Retry	⊗ Failed	Logs Lambda ↗ Log group ↗		00:00:00.208	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
▼ Redrive #4	✔ Succeeded	Logs Lambda ↗ Log group ↗		00:00:00.195	<div style="width: 100%; height: 10px; background-color: #ccc;"></div>
Output Learn more ↗					
<pre> 1 { 2 "Squared": 16 3 }</pre> <div style="text-align: right;">  Formatted ↗ </div>					

Examining Map Run of a Distributed Map state execution

When you run a Map state in Distributed mode, Step Functions creates a Map Run resource. A Map Run refers to a set of child workflow executions that a *Distributed Map state* starts, and the runtime settings that control these executions. Step Functions assigns an Amazon Resource Name (ARN) to your Map Run. You can examine a Map Run in the Step Functions console. You can also invoke the [DescribeMapRun](#) API action. A Map Run also emits metrics to CloudWatch.

The Step Functions console provides a *Map Run Details* page which displays all the information related to a *Distributed Map state* execution. For example, you can view the status of the *Distributed Map state's* execution, the Map Run's ARN, and the statuses of the items processed in the child workflow executions started by the *Distributed Map state*. You can also view a list of all child workflow executions and access their details. Additionally, if your Map Run was [redriven](#), you can see the redrive details of the Map Run in the [Map Run execution summary](#) section. For example, **Last redrive time**. The console displays this information in a dashboard format.

The *Map Run Details* page contains the following sections:

[Step Functions](#) > [State machines](#) > [SampleMapRunRedrive](#) > [Execution:SampleMapRunRedrive-1](#) > Map Run: Map:c79b2b00-70be-3d97-9291-de25e847efa2

Map Run: Map:c79b2b00-70be-3d97-9291-de25e847efa2

Details

Input and output

<p>Status 🔄 Running</p> <p>Redrive details Redrive #1 in progress</p> <p>Redrive count Info 1</p> <p>Child workflow type Info Standard</p> <p>Map Run ARN 📄 <code>arn:aws:states:us-east-1:123456789012:mapRun:SampleMapRunRedrive/Map:c79b2b00-70be-3d97-9291-de25e847efa2</code></p>	<p>Maximum concurrency Info 1000 ↗</p> <p>Item batching Info -</p> <p>Tolerated failure threshold Info 3 items ↗</p>	<p>Start time Oct 26, 2023, 1:48:06 PM PDT</p> <p>Last redrive time Oct 26, 2023, 1:48:42 PM PDT</p> <p>End time -</p>
--	--	--

Item processing status

80% processed Duration: 00:01:32.490

🕒 Pending

2

🔄 Running

0

✅ Succeeded

16

❌ Failed

2 / 20%

Threshold: 3 items

⊖ Aborted

0

Total: 20

Executions (20)

Any status ▼

<
1
>
⚙️

	Name	Number of items	Status	Start time	End time
<input type="radio"/>	1a3f52ac-036f-3c65-9f93-0dbe822ef862	1	❌ Failed	Oct 26, 2023, 1:48:07 PM PDT	Oct 26, 2023, 1:48:08 PM PDT
<input type="radio"/>	4cf0edf2-5668-3bab-98d6-c811f2165bd8	1	❌ Failed	Oct 26, 2023, 1:48:07 PM PDT	Oct 26, 2023, 1:48:08 PM PDT
<input type="radio"/>	633b5bd8-a16f-355f-8c45-c0aa381d339d	1	✅ Succeeded	Oct 26, 2023, 1:48:07 PM PDT	Oct 26, 2023, 1:48:08 PM PDT
<input type="radio"/>	a2493e43-58be-360f-9344-7a4091b52f89	1	✅ Succeeded	Oct 26, 2023, 1:48:07 PM PDT	Oct 26, 2023, 1:48:08 PM PDT

Contents

- [Map Run execution summary](#)
- [Error message](#)

- [Item processing status](#)
- [Executions listing](#)
- [Redriving Map Runs](#)
 - [Redrive eligibility for child workflows in a Map Run](#)
 - [Child workflow execution redrive behavior](#)
 - [Scenarios of input used on Map Run redrive](#)
 - [IAM permission to redrive a Map Run](#)
 - [Redriving Map Run in console](#)
 - [Redriving Map Run using API](#)

Map Run execution summary

The *Map Run Execution summary* section appears at the top of the *Map Run Details* page. This section provides you with an overview of the execution details of the *Distributed Map state*. This information is divided between the following tabs:

Details

Shows information, such as the execution status of the *Distributed Map state*, the Map Run ARN, and type of the child workflow executions started by the *Distributed Map state*. You can view additional configurations, such as tolerated failure threshold for the Map Run and the maximum concurrency specified for child workflow executions. You can also edit these configurations.

Input and output

Shows the input received by the *Distributed Map state* and the corresponding output that it generates. For example, you can view the input dataset and its location, and the input filters applied to the individual data items in that dataset. If you export the output of the *Distributed Map state* execution, this tab shows the path to the Amazon S3 bucket that contains the execution results. Otherwise, it points you to the parent workflow's *Execution Details* page to view the execution output.

Error message

If your Map Run failed, the *Map Run Details* page displays an error message with the reason for failure.

From the **Recover** dropdown button on this error message, you can either redrive the unsuccessful child workflow executions started by this Map Run or start a new execution of the parent workflow. For more information, see [Redriving Map Runs](#).

Details

Input and output

<p>Status ⊗ Failed</p> <p>Child workflow type Info Standard</p> <p>Map Run ARN <code>arn:aws:states:us-east-1:123456789012:mapRun:redriveMapRun/Map:0d641cc0-8ed7-3d10-b605-3337eb56027d</code></p>	<p>Maximum concurrency Info 1000 </p> <p>Item batching Info -</p> <p>Tolerated failure threshold Info 3 items </p>	<p>Start time Oct 25, 2023, 5:59:36 PM PDT</p> <p>End time Oct 25, 2023, 5:59:39 PM PDT</p>
--	--	---

⊗

Tolerated failure threshold exceeded

4 child workflow executions containing 4 (20%) items failed or timed out. Because the Map Run failed, 0 executions containing 0 items were aborted. [Learn more about recovery from Map Run failures](#)

Recover ▼

Item processing status

The **Item processing status** section displays the status of the items processed in a Map Run. For example, **Pending** indicates that a child workflow execution hasn't started processing the item yet.

Item statuses are dependent on the status of the child workflow executions processing the items. If a child workflow execution failed, times out, or if a user cancels the execution, Step Functions doesn't receive any information about the processing result of the items inside that child workflow execution. All items processed by that execution share the child workflow execution's status.

For example, say that you want to process 100 items in two child workflow executions, where each execution processes a batch of 50 items. If one of the executions fails and the other succeeds, you'll have 50 successful and 50 failed items.

The following table explains the types of processing statuses available for all items:

Status	Description
Pending	<p>Indicates an item that the child workflow execution hasn't started processing. If a Map Run stops, fails, or a user cancels the execution before processing of an item starts, the item remains in Pending status.</p> <p>For example, if a Map Run fails with 10 items pending to process, these 10 items remain in the Pending status.</p>
Running	<p>Indicates an item currently being processed by the child workflow execution.</p>
Succeeded	<p>Indicates that the child workflow execution successfully processed the item.</p> <p>A successful child workflow execution can't have any failed items. If one item in the dataset fails during execution, the entire child workflow execution fails.</p>
Failed	<p>Indicates that the child workflow execution either failed to process the item, or the execution timed out. If any one item processed by a child workflow execution fails, the entire child workflow execution fails.</p> <p>For example, consider a child workflow execution that processed 1000 items. If any one item in that dataset fails during execution, then Step Functions considers the entire child workflow execution as failed.</p> <p>When you redrive a Map Run, the count of items with this status is reset to 0.</p>

Status	Description
Aborted	<p>Indicates that the child workflow execution started processing the item, but either the user cancelled the execution, or Step Functions stopped the execution because the Map Run failed.</p> <p>For example, consider a Running child workflow execution that's processing 50 items. If the Map Run stops because of a failure or because a user cancelled the execution, the child workflow execution and the status of all 50 items changes to Aborted.</p> <p>If you use a child workflow execution of the Express type, you can't stop the execution.</p> <p>When you redrive a Map Run that starts child workflow executions of type Express, the count of items with this status is reset to 0. This is because Express child workflows are restarted using the StartExecution API action instead of being redriven.</p>

Executions listing

The **Executions** section lists all of the child workflow executions for a specific Map Run. Use the **Search by exact execution name** field to search for a specific child workflow execution. You can also use the **Any status** dropdown to filter child workflow execution histories by their status. To see details about a specific execution, select a child workflow execution from the list and choose the **View details** button to open its [Execution details](#) page.

Important

The retention policy for child workflow executions is 90 days. Completed child workflow executions that are older than this retention period aren't displayed in the **Executions** table. This is true even if the *Distributed Map state* or the parent workflow continues to run

longer than the retention period. You can view execution details, including results, of these child workflow executions if you export the *Distributed Map state* output to an Amazon S3 bucket using [ResultWriter](#).

Tip

Choose the refresh button



to view the most current list of all child workflow executions.

Redriving Map Runs

You can restart unsuccessful child workflow executions in a Map Run by [redriving](#) your [parent workflow](#). A redriven parent workflow redrives all the unsuccessful states, including Distributed Map. A parent workflow redrives unsuccessful states if there's no `<stateType>Exited` event corresponding to the `<stateType>Entered` event for a state when the parent workflow completed its execution. For example, if the event history doesn't contain the `MapStateExited` event for a `MapStateEntered` event, you can redrive the parent workflow to redrive all the unsuccessful child workflow executions in the Map Run.

A Map Run is either not started or fails in the original execution attempt when the state machine doesn't have the required permission to access the [ItemReader](#), [ResultWriter](#), or both. If the Map Run wasn't started in the original execution attempt of the parent workflow, redriving the parent workflow starts the Map Run for the first time. To resolve this, add the required permissions to your state machine role, and then redrive the parent workflow. If you redrive the parent workflow without adding the required permissions, it attempts to start a new Map Run run that will fail again. For information about the permissions that you might need, see [IAM policies for using Distributed Map state](#).

Topics

- [Redrive eligibility for child workflows in a Map Run](#)
- [Child workflow execution redrive behavior](#)
- [Scenarios of input used on Map Run redrive](#)
- [IAM permission to redrive a Map Run](#)

- [Redriving Map Run in console](#)
- [Redriving Map Run using API](#)

Redrive eligibility for child workflows in a Map Run

You can redrive the unsuccessful child workflow executions in a Map Run if the following conditions are met:

- You started the parent workflow execution on or after November 15, 2023. Executions that you started prior to this date aren't eligible for redrive.
- You haven't exceeded the hard limit of 1000 redrives of a given Map Run. If you've exceeded this limit, you'll receive the [States.Runtime](#) error.
- The parent workflow is redrivable. If the parent workflow isn't redrivable, you can't redrive the child workflow executions in a Map Run. For more information about redrive eligibility of a workflow, see [Redrive eligibility for unsuccessful executions](#).
- The child workflow executions of type Standard in your Map Run haven't exceeded the 25,000 execution event history limit. Child workflow executions that have exceeded the event history limit are counted towards the [tolerated failure threshold](#) and considered as failed. For more information about the redrive eligibility of an execution, see [Redrive eligibility for unsuccessful executions](#).

A new Map Run is started and the existing Map Run isn't redriven in the following cases even if the Map Run failed in the original execution attempt:

- Map Run failed because of the [States.DataLimitExceeded](#) error.
- Map Run failed because of the JSON data interpolation error, [States.Runtime](#). For example, you selected a non-existent JSON node in [OutputPath](#).

A Map Run can continue to run even after the parent workflow stops or times out. In these scenarios, the redrive doesn't happen immediately:

- Map Run might still be canceling in progress child workflow executions of type Standard, or waiting for child workflow executions of type Express to complete their executions.
- Map Run might still be writing results to the [ResultWriter](#), if you configured it to export results.

In these cases, the running Map Run completes its operations before attempting to redrive.

Child workflow execution redrive behavior

The redriven child workflow executions in a Map Run exhibit the behavior as described in the following table.

Express child workflow	Standard child workflow
<p>All child workflow executions that failed or timed out in the original execution attempt are started using the StartExecution API action. The first state in ItemProcessor is run first.</p>	<p>All child workflow executions that failed, timed out, or canceled in the original execution attempt are redriven using the RedriveExecution API action. These child workflows are redriven from the last state in ItemProcessor that resulted in their unsuccessful execution.</p>
<p>Unsuccessful executions can always be redriven. This is because Express child workflow executions are always started as a new execution using the StartExecution API action.</p>	<p>Unsuccessful Standard child workflow executions can't always be redriven. If an execution isn't redrivable, it won't be attempted again. The last error or output of the execution is permanent. This is possible when an execution exceeds 25,000 history events, or its redrivable period of 14 days has expired.</p> <p>A Standard child workflow execution might not be redrivable if the parent workflow execution has closed within 14 days, but the child workflow execution closed earlier than 14 days.</p>
<p>Express child workflow executions use the same execution ARN as the original execution attempt, but you can't distinctly identify their individual redrives.</p>	<p>Standard child workflow executions use the same execution ARN as the original execution attempt. You can distinctly identify the individual redrives in the console and using APIs, such as GetExecutionHistory and</p>

Express child workflow**Standard child workflow**

[DescribeExecution](#). For more information, see [Examining redriven executions](#).

If you've redriven a Map Run, and it has reached its concurrency limit, the child workflow executions in that Map Run transition to the pending state. The execution status of the Map Run also transitions to the **Pending redrive** state. Until the specified concurrency limit can allow for more child workflow executions to run, the execution remains in the **Pending redrive** state.

For example, say that the concurrency limit of the Distributed Map in your workflow is 3000, and the number of child workflows to be rerun is 6000. This causes 3000 child workflows to run in parallel while the remaining 3000 workflows remain in the **Pending redrive** state. After the first batch of 3000 child workflows complete their execution, the remaining 3000 child workflows are run.

When a Map Run has completed its execution or is aborted, the count of child workflow executions in the **Pending redrive** state is reset to 0.

Scenarios of input used on Map Run redrive

Depending on how you provided input to the Distributed Map in the original execution attempt, a redriven Map Run will use the input as described in the following table.

Input in the original execution attempt	Input used on Map Run redrive
Input passed from a previous state or the execution input.	The redriven Map Run uses the same input.
Input passed using ItemReader and the Map Run didn't start the child workflow execution s because one of the following conditions is true: <ul style="list-style-type: none"> • Map Run failed with the States.ItemReaderFailed error. • Map Run failed with the States.ResultWriterFailed error. 	The redriven Map Run uses the input in the Amazon S3 bucket.

Input in the original execution attempt	Input used on Map Run redrive
<ul style="list-style-type: none"> The parent workflow execution was timed out or canceled before the Map Run was started. 	
<p>Input passed using ItemReader. The Map Run failed after starting or attempting to start child workflow executions.</p>	<p>The redriven Map Run uses the same input provided in the original execution attempt.</p>

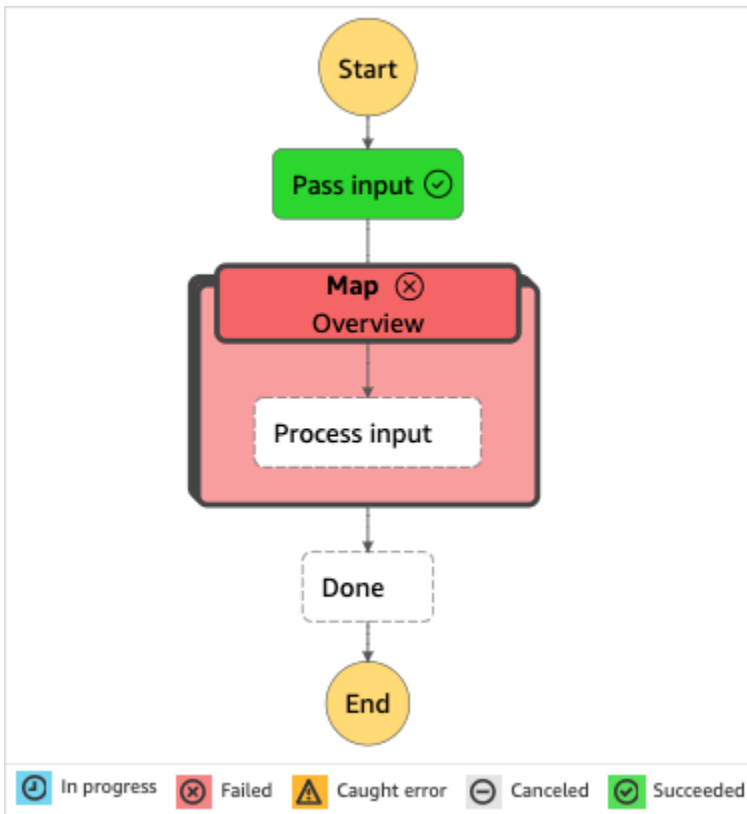
IAM permission to redrive a Map Run

Step Functions needs appropriate permission to redrive a Map Run. The following IAM policy example grants the least privilege required to your state machine for redriving a Map Run. Remember to replace the *italicized* text with your resource-specific information.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:RedriveExecution"
      ],
      "Resource": "arn:aws:states:us-east-2:123456789012:execution:myStateMachine/myMapRunLabel:*"
    }
  ]
}
```

Redriving Map Run in console

The following image shows the execution graph of a state machine that contains a Distributed Map. This execution failed because the Map Run failed. To redrive the Map Run, you must redrive the parent workflow.



To redrive a Map Run from the console

1. Open the [Step Functions console](#), and then choose an existing state machine that contains a Distributed Map that failed execution.
2. On the state machine detail page, under **Executions**, choose a failed execution instance of this state machine.
3. Choose **Redrive**.
4. In the **Redrive** dialog box, choose **Redrive execution**.

i Tip

You can also redrive a Map Run from the *Execution Details* or *Map Run Details* page. If you're on the *Execution Details* page, do one of the following to redrive the execution:

- Choose **Recover**, and then select **Redrive from failure**.
- Choose **Actions**, and then select **Redrive**.

If you're on the *Map Run Details* page, choose **Recover**, and then select **Redrive from failure**.

Notice that redrive uses the same state machine definition and ARN. It continues running the execution from the step that failed in the original execution attempt. In this example, it's the Distributed Map step named **Map** and the **Process input** step inside it. After restarting the unsuccessful child workflow executions of the Map Run, redrive will continue execution for the **Done** step.

5. From the *Execution Details* page, choose **Map Run** to see the details of the redriven Map Run.

On this page, you can view the results of the redriven execution. For example, in the [Map Run execution summary](#) section, you can see **Redrive count**, which represents the number of times the Map Run has been redriven. In the **Events** section, you can see the redrive related execution events appended to the events of the original execution attempt. For example, the `MapRunRedriven` event.

After you've redriven a Map Run, you can examine its redrive details in the console or using the [GetExecutionHistory](#) and [DescribeExecution](#) API actions. For more information about examining a redriven execution, see [Examining redriven executions](#).

Redriving Map Run using API

You can redrive an [eligible](#) Map Run using the [RedriveExecution](#) API on the parent workflow. This API restarts unsuccessful child workflow executions in a Map Run.

In the AWS Command Line Interface (AWS CLI), run the following command to redrive an unsuccessful state machine execution. Remember to replace the *italicized* text with your resource-specific information.

```
aws stepfunctions redrive-execution --execution-arn arn:aws:states:us-east-2:123456789012:execution:myStateMachine:foo
```

After you've redriven a Map Run, you can examine its redrive details in the console or using the [DescribeMapRun](#) API action. To examine the redrive details of Standard workflow executions in a Map Run, you can use the [GetExecutionHistory](#) or [DescribeExecution](#) API action. For more

information about examining a redriven execution, see [the section called “Examining redriven executions”](#).

You can examine the redrive details of Express workflow executions in a Map Run on the [Step Functions console](#) if you've enabled logging on the parent workflow. For more information, see [Logging using CloudWatch Logs](#).

Error handling in Step Functions

All states, except Pass and Wait states, can encounter runtime errors. Errors can happen for various reasons, such as the following examples:

- State machine definition issues (for example, no matching rule in a Choice state)
- Task failures (for example, an exception in a AWS Lambda function)
- Transient issues (for example, network partition events)

By default, when a state reports an error, AWS Step Functions causes the execution to fail entirely.

Tip

To deploy an example of a workflow that includes error handling to your AWS account, see [Module 8 - Error Handling](#) of *The AWS Step Functions Workshop*.

Topics

- [Error names](#)
- [Retrying after an error](#)
- [Fallback states](#)
- [State machine examples using Retry and using Catch](#)

Error names

Step Functions identifies errors in the Amazon States Language using case-sensitive strings, known as *error names*. The Amazon States Language defines a set of built-in strings that name well-known errors, all beginning with the `States.` prefix.

States.ALL

A wildcard that matches any known error name.

Note

This error type can't catch the `States.DataLimitExceeded` terminal error type and runtime error types. For more information about these error types, see [States.DataLimitExceeded](#) and [States.Runtime](#).

States.DataLimitExceeded

Step Functions reports a `States.DataLimitExceeded` exception under the following conditions:

- When the output of a connector is larger than payload size quota.
- When the output of a state is larger than payload size quota.
- When, after Parameters processing, the input of a state is larger than the payload size quota.

For more information on quotas, see [Quotas](#).

Note

This is a terminal error that can't be caught by the `States.ALL` error type.

States.ExceedToleratedFailureThreshold

A Map state failed because the number of failed items exceeded the threshold specified in the state machine definition. For more information, see [Tolerated failure threshold for Distributed Map state](#).

States.HeartbeatTimeout

A Task state failed to send a heartbeat for a period longer than the `HeartbeatSeconds` value.

Note

This error is only available inside the `Catch` and `Retry` fields.

States.IntrinsicFailure

An attempt to invoke an intrinsic function within a payload template failed.

States.ItemReaderFailed

A Map state failed because it couldn't read from the item source specified in the `ItemReader` field. For more information, see [ItemReader](#).

States.NoChoiceMatched

A Choice state failed to match the input with the conditions defined in the Choice Rule and a Default transition isn't specified.

States.ParameterPathFailure

An attempt to replace a field, within a state's `Parameters` field, whose name ends in `.$` using a path fails.

States.Permissions

A Task state failed because it had insufficient privileges to run the specified code.

States.ResultPathMatchFailure

Step Functions failed to apply a state's `ResultPath` field to the input the state received.

States.ResultWriterFailed

A Map state failed because it couldn't write results to the destination specified in the `ResultWriter` field. For more information, see [ResultWriter](#).

States.Runtime

An execution failed due to some exception that it couldn't process. Often these are caused by errors at runtime, such as attempting to apply `InputPath` or `OutputPath` on a null JSON payload. A `States.Runtime` error isn't retrievable, and will always cause the execution to fail. A retry or catch on `States.ALL` won't catch `States.Runtime` errors.

States.TaskFailed

A Task state failed during the execution. When used in a retry or catch, `States.TaskFailed` acts as a wildcard that matches any known error name except for `States.Timeout`.

States.Timeout

A Task state either ran longer than the `TimeoutSeconds` value, or failed to send a heartbeat for a period longer than the `HeartbeatSeconds` value.

Additionally, if a state machine runs longer than the specified `TimeoutSeconds` value, the execution fails with a `States.Timeout` error.

States can report errors with other names. However, these error names can't begin with the `States.` prefix.

As a best practice, ensure production code can handle AWS Lambda service exceptions (`Lambda.ServiceException` and `Lambda.SdkClientException`). For more information, see [Handle Lambda service exceptions](#).

Note

Unhandled errors in Lambda are reported as `Lambda.Unknown` in the error output. These include out-of-memory errors and function timeouts. You can match on `Lambda.Unknown`, `States.ALL`, or `States.TaskFailed` to handle these errors. When Lambda hits the maximum number of invocations, the error is `Lambda.TooManyRequestsException`. For more information about Lambda function errors, see [Error handling and automatic retries](#) in the *AWS Lambda Developer Guide*.

Retrying after an error

`Task`, `Parallel`, and `Map` states can have a field named `Retry`, whose value must be an array of objects known as *retriers*. An individual retrier represents a certain number of retries, usually at increasing time intervals.

When one of these states reports an error and there's a `Retry` field, Step Functions scans through the retriers in the order listed in the array. When the error name appears in the value of a retrier's `ErrorEquals` field, the state machine makes retry attempts as defined in the `Retry` field.

If your redriven execution reruns a [Task state](#), [Parallel](#), or [Inline Map state](#), for which you have defined [retries](#), the retry attempt count for these states is reset to 0 to allow for the maximum number of attempts on redrive. For a redriven execution, you can track individual retry attempts of these states using the console. For more information, see [Retry behavior of redriven executions](#) in [Redriving executions](#).

A retrier contains the following fields:

Note

Retries are treated as state transitions. For information about how state transitions affect billing, see [Step Functions Pricing](#).

ErrorEquals (Required)

A non-empty array of strings that match error names. When a state reports an error, Step Functions scans through the retriers. When the error name appears in this array, it implements the retry policy described in this retrier.

IntervalSeconds (Optional)

A positive integer that represents the number of seconds before the first retry attempt (1 by default). `IntervalSeconds` has a maximum value of 99999999.

MaxAttempts (Optional)

A positive integer that represents the maximum number of retry attempts (3 by default). If the error recurs more times than specified, retries cease and normal error handling resumes. A value of 0 specifies that the error is never retried. `MaxAttempts` has a maximum value of 99999999.

BackoffRate (Optional)

The multiplier by which the retry interval denoted by `IntervalSeconds` increases after each retry attempt. By default, the `BackoffRate` value increases by 2.0.

For example, say your `IntervalSeconds` is 3, `MaxAttempts` is 3, and `BackoffRate` is 2. The first retry attempt takes place three seconds after the error occurs. The second retry takes place six seconds after the first retry attempt. While the third retry takes place 12 seconds after the second retry attempt.

MaxDelaySeconds (Optional)

A positive integer that sets the maximum value, in seconds, up to which a retry interval can increase. This field is helpful to use with the `BackoffRate` field. The value you specify in this field limits the exponential wait times resulting from the backoff rate multiplier applied to each consecutive retry attempt. You must specify a value greater than 0 and less than 31622401 for `MaxDelaySeconds`.

If you don't specify this value, Step Functions doesn't limit the wait times between retry attempts.

JitterStrategy (Optional)

A string that determines whether or not to include jitter in the wait times between consecutive retry attempts. Jitter reduces simultaneous retry attempts by spreading these out over a randomized delay interval. This string accepts FULL or NONE as its values. The default value is NONE.

For example, say you have set `MaxAttempts` as 3, `IntervalSeconds` as 2, and `BackoffRate` as 2. The first retry attempt takes place two seconds after the error occurs. The second retry takes place four seconds after the first retry attempt and the third retry takes place eight seconds after the second retry attempt. If you set `JitterStrategy` as FULL, the first retry interval is randomized between 0 and 2 seconds, the second retry interval is randomized between 0 and 4 seconds, and the third retry interval is randomized between 0 and 8 seconds.

Retry field examples

This section includes the following `Retry` field examples.

- [Retry with BackoffRate](#)
- [Retry with MaxDelaySeconds](#)
- [Retry all errors except States.Timeout](#)
- [Complex retry scenario](#)

Tip

To deploy an example of an error handling workflow to your AWS account, see [Error Handling](#) module of *The AWS Step Functions Workshop*.

Example 1 – Retry with BackoffRate

The following example of a `Retry` makes two retry attempts with the first retry taking place after waiting for three seconds. Based on the `BackoffRate` you specify, Step Functions increases the interval between each retry until the maximum number of retry attempts is reached. In the following example, the second retry attempt starts after waiting for three seconds after the first retry.

```
"Retry": [ {
```

```
"ErrorEquals": [ "States.Timeout" ],
"IntervalSeconds": 3,
"MaxAttempts": 2,
"BackoffRate": 1
} ]
```

Example 2 – Retry with MaxDelaySeconds

The following example makes three retry attempts and limits the wait time resulting from `BackoffRate` at 5 seconds. The first retry takes place after waiting for three seconds. The second and third retry attempts take place after waiting for five seconds after the preceding retry attempt because of the maximum wait time limit set by `MaxDelaySeconds`.

```
"Retry": [ {
  "ErrorEquals": [ "States.Timeout" ],
  "IntervalSeconds": 3,
  "MaxAttempts": 3,
  "BackoffRate": 2,
  "MaxDelaySeconds": 5,
  "JitterStrategy": "FULL"
} ]
```

Without `MaxDelaySeconds`, the second retry attempt would take place six seconds after the first retry, and the third retry attempt would take place 12 seconds after the second retry.

Example 3 – Retry all errors except States.Timeout

The reserved name `States.ALL` that appears in a retriever's `ErrorEquals` field is a wildcard that matches any error name. It must appear alone in the `ErrorEquals` array and must appear in the last retriever in the `Retry` array. The name `States.TaskFailed` also acts a wildcard and matches any error except for `States.Timeout`.

The following example of a `Retry` field retries any error except `States.Timeout`.

```
"Retry": [ {
  "ErrorEquals": [ "States.Timeout" ],
  "MaxAttempts": 0
}, {
  "ErrorEquals": [ "States.ALL" ]
} ]
```

Example 4 – Complex retry scenario

A retriever's parameters apply across all visits to the retriever in the context of a single-state execution.

Consider the following Task state.

```
"X": {
  "Type": "Task",
  "Resource": "arn:aws:states:us-east-1:123456789012:task:X",
  "Next": "Y",
  "Retry": [ {
    "ErrorEquals": [ "ErrorA", "ErrorB" ],
    "IntervalSeconds": 1,
    "BackoffRate": 2.0,
    "MaxAttempts": 2
  }, {
    "ErrorEquals": [ "ErrorC" ],
    "IntervalSeconds": 5
  } ],
  "Catch": [ {
    "ErrorEquals": [ "States.ALL" ],
    "Next": "Z"
  } ]
}
```

This task fails four times in succession, outputting these error names: `ErrorA`, `ErrorB`, `ErrorC`, and `ErrorB`. The following occurs as a result:

- The first two errors match the first retriever and cause waits of one and two seconds.
- The third error matches the second retriever and causes a wait of five seconds.
- The fourth error also matches the first retriever. However, it already reached its maximum of two retries (`MaxAttempts`) for that particular error. Therefore, that retriever fails and the execution redirects the workflow to the Z state through the `Catch` field.

Fallback states

`Task`, `Map` and `Parallel` states can each have a field named `Catch`. This field's value must be an array of objects, known as *catchers*.

A catcher contains the following fields.

ErrorEquals (Required)

A non-empty array of strings that match error names, specified exactly as they are with the `retrier` field of the same name.

Next (Required)

A string that must exactly match one of the state machine's state names.

ResultPath (Optional)

A [path](#) that determines what input the catcher sends to the state specified in the `Next` field.

When a state reports an error and either there is no `Retry` field, or if retries fail to resolve the error, Step Functions scans through the catchers in the order listed in the array. When the error name appears in the value of a catcher's `ErrorEquals` field, the state machine transitions to the state named in the `Next` field.

The reserved name `States.ALL` that appears in a catcher's `ErrorEquals` field is a wildcard that matches any error name. It must appear alone in the `ErrorEquals` array and must appear in the last catcher in the `Catch` array. The name `States.TaskFailed` also acts a wildcard and matches any error except for `States.Timeout`.

The following example of a `Catch` field transitions to the state named `RecoveryState` when a Lambda function outputs an unhandled Java exception. Otherwise, the field transitions to the `EndState` state.

```
"Catch": [ {
  "ErrorEquals": [ "java.lang.Exception" ],
  "ResultPath": "$.error-info",
  "Next": "RecoveryState"
}, {
  "ErrorEquals": [ "States.ALL" ],
  "Next": "EndState"
} ]
```

Note

Each catcher can specify multiple errors to handle.

Error output

When Step Functions transitions to the state specified in a catch name, the object usually contains the field `Cause`. This field's value is a human-readable description of the error. This object is known as the *error output*.

In this example, the first catcher contains a `ResultPath` field. This works similarly to a `ResultPath` field in a state's top level, resulting in two possibilities:

- It takes the results of that state's execution and overwrites either all of, or a portion of, the state's input.
- It takes the results and adds them to the input. In the case of an error handled by a catcher, the result of the state's execution is the error output.

Thus, for the first catcher in the example, the catcher adds the error output to the input as a field named `error-info` if there isn't already a field with this name in the input. Then, the catcher sends the entire input to `RecoveryState`. For the second catcher, the error output overwrites the input and the catcher only sends the error output to `EndState`.

Note

If you don't specify the `ResultPath` field, it defaults to `$`, which selects and overwrites the entire input.

When a state has both `Retry` and `Catch` fields, Step Functions uses any appropriate retriers first. If the retry policy fails to resolve the error, Step Functions applies the matching catcher transition.

Cause payloads and service integrations

A catcher returns a string payload as an output. When working with service integrations such as Amazon Athena or AWS CodeBuild, you may want to convert the `Cause` string to JSON. The following example of a `Pass` state with intrinsic functions shows how to convert a `Cause` string to JSON.

```
"Handle escaped JSON with JSONtoString": {
  "Type": "Pass",
  "Parameters": {
```

```
"Cause.$": "States.StringToJson($.Cause)"
},
"Next": "Pass State with Pass Processing"
},
```

State machine examples using Retry and using Catch

The state machines defined in the following examples assume the existence of two Lambda functions: one that always fails and one that waits long enough to allow a timeout defined in the state machine to occur.

This is a definition of a Node.js Lambda function that always fails, returning the message `error`. In the state machine examples that follow, this Lambda function is named `FailFunction`. For information about creating a Lambda function, see [Step 1: Create a Lambda function](#) section.

```
exports.handler = (event, context, callback) => {
  callback("error");
};
```

This is a definition of a Node.js Lambda function that sleeps for 10 seconds. In the state machine examples that follow, this Lambda function is named `sleep10`.

Note

When you create this Lambda function in the Lambda console, remember to change the **Timeout** value in the **Advanced settings** section from 3 seconds (default) to 11 seconds.

```
exports.handler = (event, context, callback) => {
  setTimeout(function(){
  }, 11000);
};
```

Handling a failure using Retry

This state machine uses a `Retry` field to retry a function that fails and outputs the error name `HandledError`. It retries this function twice with an exponential backoff between retries.

```
{
```

```
"Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
function",
"StartAt": "HelloWorld",
"States": {
  "HelloWorld": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FailFunction",
    "Retry": [ {
      "ErrorEquals": ["HandledError"],
      "IntervalSeconds": 1,
      "MaxAttempts": 2,
      "BackoffRate": 2.0
    } ],
    "End": true
  }
}
```

This variant uses the predefined error code `States.TaskFailed`, which matches any error that a Lambda function outputs.

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FailFunction",
      "Retry": [ {
        "ErrorEquals": ["States.TaskFailed"],
        "IntervalSeconds": 1,
        "MaxAttempts": 2,
        "BackoffRate": 2.0
      } ],
      "End": true
    }
  }
}
```

Note

As a best practice, tasks that reference a Lambda function should handle Lambda service exceptions. For more information, see [Handle Lambda service exceptions](#).

Handling a failure using Catch

This example uses a `Catch` field. When a Lambda function outputs an error, it catches the error and the state machine transitions to the fallback state.

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FailFunction",
      "Catch": [ {
        "ErrorEquals": ["HandledError"],
        "Next": "fallback"
      } ],
      "End": true
    },
    "fallback": {
      "Type": "Pass",
      "Result": "Hello, AWS Step Functions!",
      "End": true
    }
  }
}
```

This variant uses the predefined error code `States.TaskFailed`, which matches any error that a Lambda function outputs.

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
function",
  "StartAt": "HelloWorld",
  "States": {
```

```
"HelloWorld": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FailFunction",
  "Catch": [ {
    "ErrorEquals": ["States.TaskFailed"],
    "Next": "fallback"
  } ],
  "End": true
},
"fallback": {
  "Type": "Pass",
  "Result": "Hello, AWS Step Functions!",
  "End": true
}
}
```

Handling a timeout using Retry

This state machine uses a `Retry` field to retry a `Task` state that times out, based on the timeout value specified in `TimeoutSeconds`. Step Functions retries the Lambda function invocation in this `Task` state twice, with an exponential backoff between retries.

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:sleep10",
      "TimeoutSeconds": 2,
      "Retry": [ {
        "ErrorEquals": ["States.Timeout"],
        "IntervalSeconds": 1,
        "MaxAttempts": 2,
        "BackoffRate": 2.0
      } ],
      "End": true
    }
  }
}
```

Handling a timeout using Catch

This example uses a `Catch` field. When a timeout occurs, the state machine transitions to the fallback state.

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:sleep10",
      "TimeoutSeconds": 2,
      "Catch": [ {
        "ErrorEquals": ["States.Timeout"],
        "Next": "fallback"
      } ],
      "End": true
    },
    "fallback": {
      "Type": "Pass",
      "Result": "Hello, AWS Step Functions!",
      "End": true
    }
  }
}
```

Note

You can preserve the state input and the error by using `ResultPath`. See [Use ResultPath to Include Both Error and Input in a Catch](#).

Invoke AWS Step Functions from other services

You can configure several other services to invoke state machines. Based on the state machine's [workflow type](#), you can invoke state machines asynchronously or synchronously. To invoke state machines synchronously, use the [StartSyncExecution](#) API call or Amazon API Gateway integration with Express Workflows. With asynchronous invocation, Step Functions pauses the

workflow execution until a task token is returned. However, waiting for a task token does make the workflow synchronous.

Services that you can configure to invoke Step Functions include:

- AWS Lambda, using the [StartExecution](#) call.
- [Amazon API Gateway](#)
- [Amazon EventBridge](#)
- [AWS CodePipeline](#)
- [AWS IoT Rules Engine](#)
- [AWS Step Functions](#)

Step Functions invocations are governed by the `StartExecution` quota. For more information, see:

- [Quotas](#)

Read Consistency in Step Functions

State machine updates in AWS Step Functions are eventually consistent. All `StartExecution` calls within a few seconds will use the updated definition and `roleArn` (the Amazon Resource Name for the IAM role). Executions started immediately after calling `UpdateStateMachine` might use the previous state machine definition and `roleArn`.

For more information, see the following:

- [UpdateStateMachine](#) in the *AWS Step Functions API Reference*
- [Update a workflow](#) in [Getting started with AWS Step Functions](#).

Tagging in Step Functions

AWS Step Functions supports tagging of state machines (both Standard and Express) and activities. This can help you track and manage the costs associated with your resources, and provide better security in your AWS Identity and Access Management (IAM) policies. Tagging Step Functions resources allows them to be managed by AWS Resource Groups. For more information on Resource Groups, see the [AWS Resource Groups User Guide](#).

For tag-based authorization, state machine execution resources as shown in the following example inherit the tags associated with a state machine.

```
arn:<partition>:states:<Region>:<account-id>:execution:<StateMachineName>:<ExecutionId>
```

When you call [DescribeExecution](#) or other APIs in which you specify the execution resource ARN, Step Functions uses tags associated with the state machine to accept or deny the request while performing tag-based authorization. This helps you allow or deny access to state machine executions at the state machine level.

To review the restrictions related to resource tagging, see [Restrictions related to tagging](#).

Topics

- [Tagging for Cost Allocation](#)
- [Tagging for Security](#)
- [Viewing and Managing Tags in the Step Functions Console](#)
- [Manage Tags with Step Functions API Actions](#)

Tagging for Cost Allocation

To organize and identify your Step Functions resources for cost allocation, you can add metadata *tags* that identify the purpose of a state machine or activity. This is especially useful when you have many resources. You can use cost allocation tags to organize your AWS bill to reflect your own cost structure. To do this, sign up to get your AWS account bill to include the tag keys and values. For more information, see [Setting Up a Monthly Cost Allocation Report](#) in the *AWS Billing User Guide*.

For example, you could add tags that represent the cost center and purpose of your Step Functions resources, as follows.

Resource	Key	Value
StateMachine1	Cost Center	34567
	Application	Image processing
StateMachine2	Cost Center	34567

Resource	Key	Value
	Application	Rekognition processing
Activity1	Cost Center	12345
	Application	Legacy database

This tagging scheme allows you to group two state machines performing related tasks in the same cost center, while tagging an unrelated activity with a different cost allocation tag.

Tagging for Security

IAM supports controlling access to resources based on tags. To control access based on tags, provide information about your resource tags in the condition element of an IAM policy.

For example, you could restrict access to all Step Functions resources that include a tag with the key `environment` and the value `production`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "states:TagResource",
        "states>DeleteActivity",
        "states>DeleteStateMachine",
        "states:StopExecution"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {"aws:ResourceTag/environment": "production"}
      }
    }
  ]
}
```

For more information, see [Controlling Access Using Tags](#) in the IAM User Guide.

Viewing and Managing Tags in the Step Functions Console

Step Functions allows you to view and manage the tags for your state machines in the Step Functions console. From the **Details** page of a state machine, select **Tags**. Here, you can view the existing tags associated with your state machine.

Note

To manage tags for activities, see [Manage Tags with Step Functions API Actions](#).

To add or delete tags that are associated with your state machine, select the **Manage Tags** button.

1. Browse to the details page of a state machine.
2. Select **Tags**, next to **Executions** and **Definition**.
3. Choose **Manage tags**.
 - To modify existing tags, edit the **Key** and **Value**.
 - To remove existing tags, choose **Remove tag**.
 - To add a new tag, choose **Add tag** and enter a **Key** and **Value**.
4. Choose **Save**.

Manage Tags with Step Functions API Actions

To manage tags using the Step Functions API, use the following API actions:

- [ListTagsForResource](#)
- [TagResource](#)
- [UntagResource](#)

AWS Step Functions Workflow Studio

Workflow Studio for AWS Step Functions is a low-code visual workflow designer that lets you create serverless workflows by orchestrating AWS services. Using its drag-and-drop feature or the built-in code editor, you can create and edit workflows, control how input and output is filtered or transformed for each state, and configure error handling. As you drag and drop states to build your workflow, Workflow Studio validates your work and auto-generates code. You can review the generated code or update the state machine definition within the code editor. When you're finished, you can save your workflow, run it, then examine the results in the Step Functions console. You can visually add and modify workflows to orchestrate the multiple services in your application.

To use Step Functions Workflow Studio, you will need an AWS account, and credentials that provide the correct permissions for any resources you want to use. For more information, see [Prerequisites for getting started with AWS Step Functions](#).

Note

Workflow Studio doesn't support Internet Explorer 11. If you're you using Internet Explorer 11 and encounter issues using Workflow Studio, try using a different browser.

You can access Workflow Studio from the [Step Functions console](#), when you create or edit a workflow in Step Functions. You can also [access](#) Workflow Studio within AWS Application Composer. Workflow Studio in Application Composer provides a visual IaC environment that makes it easy for you to incorporate workflows in your serverless applications built using IaC tools, such as AWS CloudFormation templates. Using Workflow Studio in Application Composer, you can build workflows using AWS CloudFormation templates. Within Application Composer, you can add a new workflow, modify an existing workflow, and connect individual workflow steps to other application resources. Application Composer automatically creates and updates the needed CloudFormation resources and configurations. This helps you create and manage all resources used in your workflows in one place. This also helps you accelerate your path from workflow prototyping to production deployment.

When you use Workflow Studio in Application Composer, you can also directly connect to your local project. This helps you to work in your integrated development environment (IDE) alongside the visual canvas. For more information, see [Using Workflow Studio in Application Composer](#).

Topics

- [Interface overview](#)
- [Using Workflow Studio](#)
- [Configure inputs and outputs for your states](#)
- [Execution roles in Workflow Studio](#)
- [Error handling](#)
- [Tutorial: Learn to use the AWS Step Functions Workflow Studio](#)

Interface overview

Workflow Studio for AWS Step Functions is a low-code visual workflow designer that lets you create serverless workflows by orchestrating AWS services. With its drag and drop feature, Workflow Studio makes it easy for you to build, edit, and visualize your workflow prototypes. Workflow Studio also offers a built-in code editor for writing and editing your workflow definitions using [Amazon States Language](#) (ASL) within the Step Functions console.

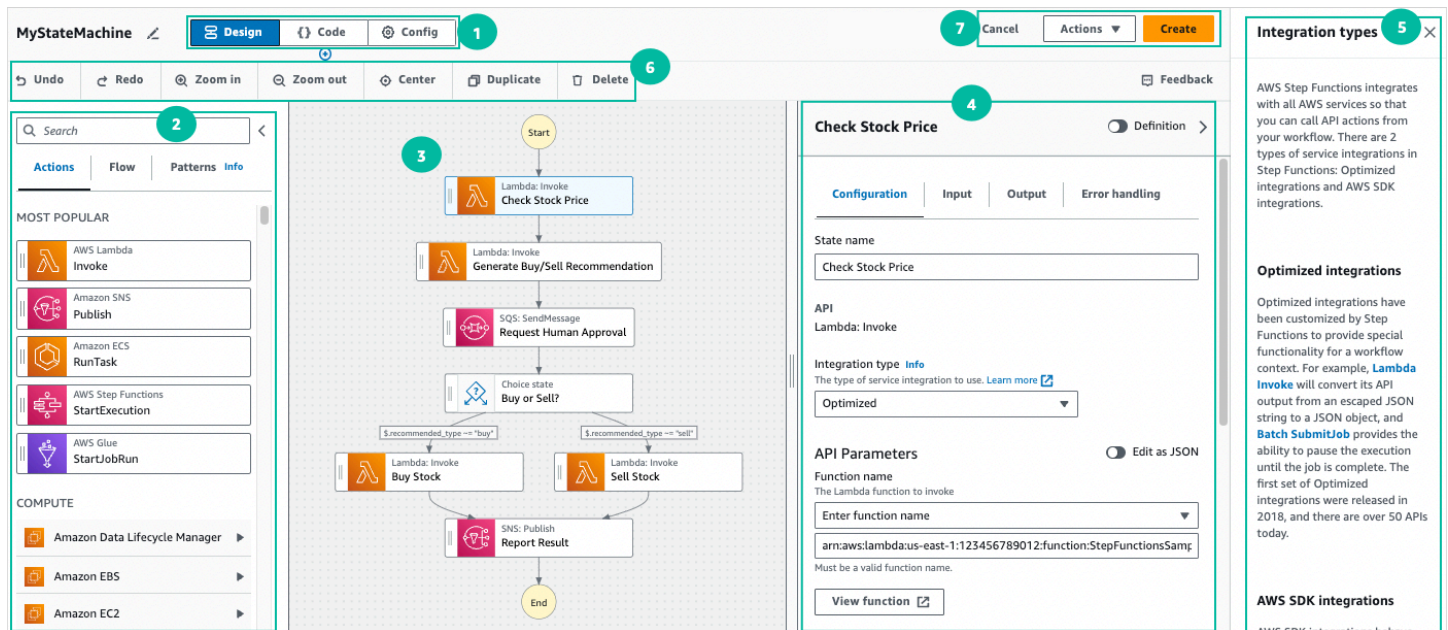
To help you build and visualize your workflows, edit their definitions, and manage their configuration, Workflow Studio provides three modes: **Design**, **Code**, and **Config**. The following sections describe these modes in detail.

In this topic

- [Design mode](#)
- [Code mode](#)
- [Config mode](#)
- [Keyboard shortcuts](#)

Design mode

The **Design** mode of Workflow Studio provides a graphical interface to visualize your workflows as you build their prototypes. The following image shows the different components available in the **Design** mode.

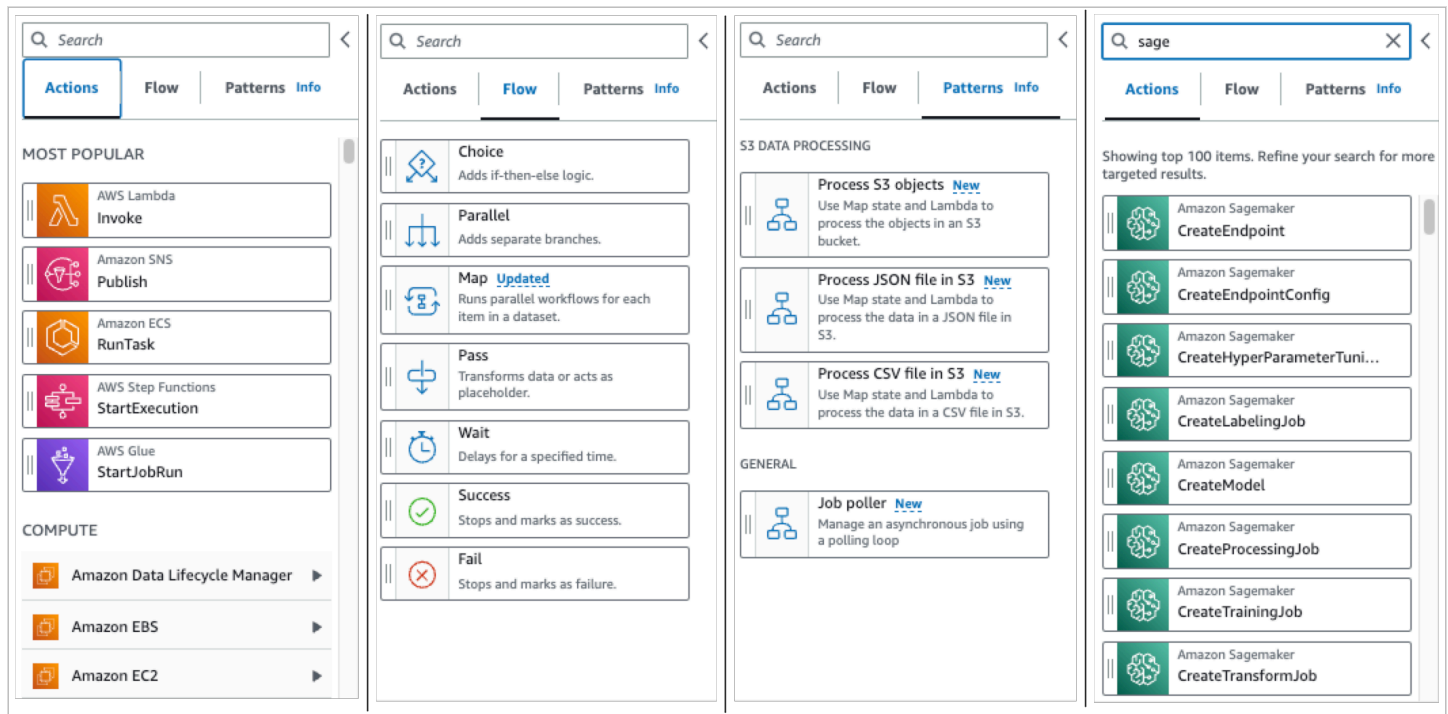


1. Mode buttons – Switch between the **Design**, **Code**, and **Config** modes of the Workflow Studio using the mode buttons. You can't switch modes if the JSON in the ASL definition of your workflow is invalid.
2. The [States browser](#) contains the following three tabs:
 - The **Actions** tab provides a list of AWS APIs that you can drag and drop into your workflow graph in the canvas. Each action represents a [Task state](#) state.
 - The **Flow** tab provides a list of flow states that you can drag and drop into your workflow graph in the canvas.
 - The **Patterns** tab provides several ready-to-use, reusable building blocks that you can use for a variety of use cases. For example, you can use these patterns to iteratively process data in an Amazon S3 bucket.
3. The [Canvas](#) is where you drag and drop states into your workflow graph, change the order of states, and select states to configure or view.
4. The [Inspector](#) panel is where you can view and edit the properties of any state you've selected on the canvas. Turn on the **Definition** toggle to view the Amazon States Language code for your workflow, with the currently selected state highlighted.
5. **Info** links open a panel with contextual information when you need help. These panels also include links to related topics in the Step Functions documentation.
6. Design toolbar – Contains a set of buttons to perform common actions, such as undo, delete, and zoom in.

- Utility buttons – A set of buttons to perform tasks, such as saving your workflows or exporting their ASL definitions in a JSON or YAML file.

States browser

The **States browser** is where you select states to drag and drop into your workflow graph. The **Actions** tab provides a list of AWS APIs, and the **Flow** tab provides a list of flow states. While the **Patterns** tab provides several ready-to-use, reusable building blocks that you can use for a variety of use cases. You can search all states in the **States Browser** using the Search box at the top.



There are seven flow states that you can use to direct and control your workflow. All of them take input from the previous state, and many let you filter the input from the preceding state, and the output to the state that follows. The flow states are:

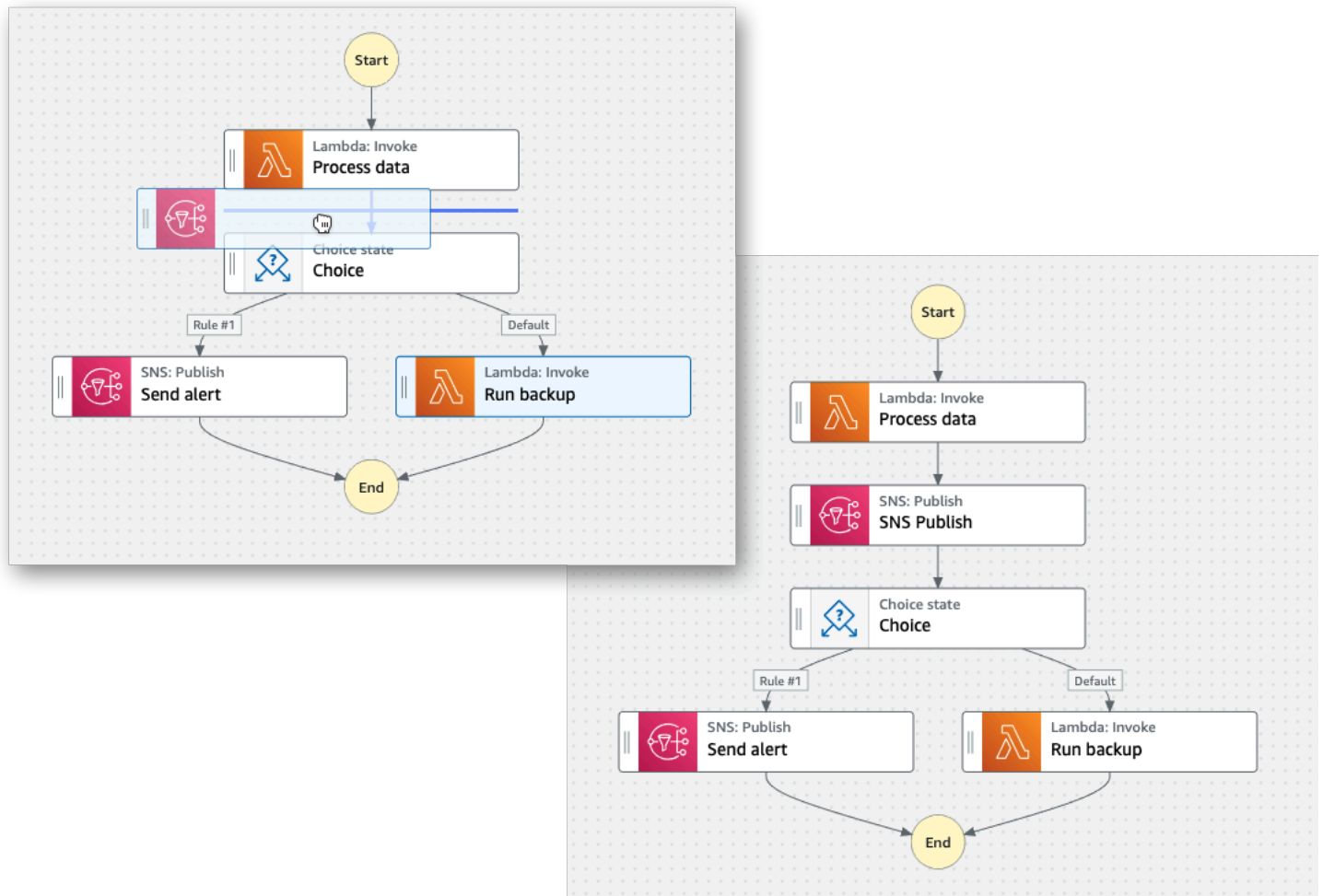
- Choice:** Add a choice between branches of execution to your workflow. In the **Configuration** tab of the Inspector, you can configure rules to determine which state the workflow will transition to.
- Parallel:** Add parallel branches of execution to your workflow.
- Map:** Dynamically iterate steps for each element of an input array. Unlike a **Parallel** flow state, a **Map** state will execute the same steps for multiple entries of an array in the state input.
- Pass:** Lets you pass its input to its output. (Optional) You can add fixed data into the output.
- Wait:** Have your workflow pause for a certain amount of time or until a specified time or date.

- [Succeed](#): Stops your workflow with a success.
- [Fail](#): Stops your workflow with a failure.

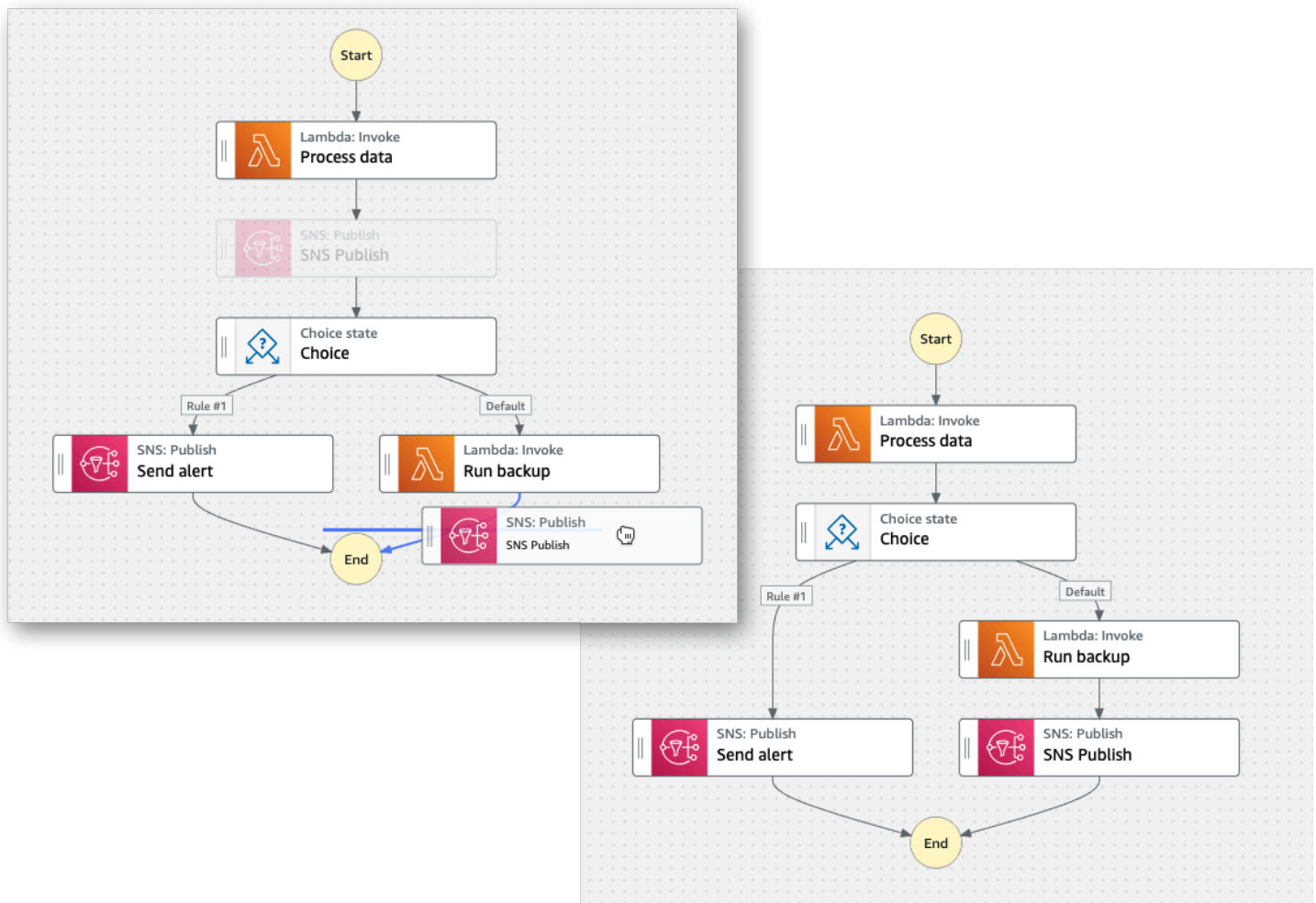
Canvas

After you choose a state to add to your workflow, drag it to the canvas and drop it into your workflow graph. You can also drag and drop states to move them to different places in your workflow. If your workflow is complex, you may not be able to view all of it in the canvas panel. Use the controls at the top of the canvas to zoom in or out. To view different parts of a workflow graph, you can drag the workflow graph in the canvas.

Drag a workflow state from the **Actions** or **Flow** tab and drop it into your workflow. A line shows where it will be placed in your workflow. The new workflow state has been added to your workflow, and its code is auto-generated.

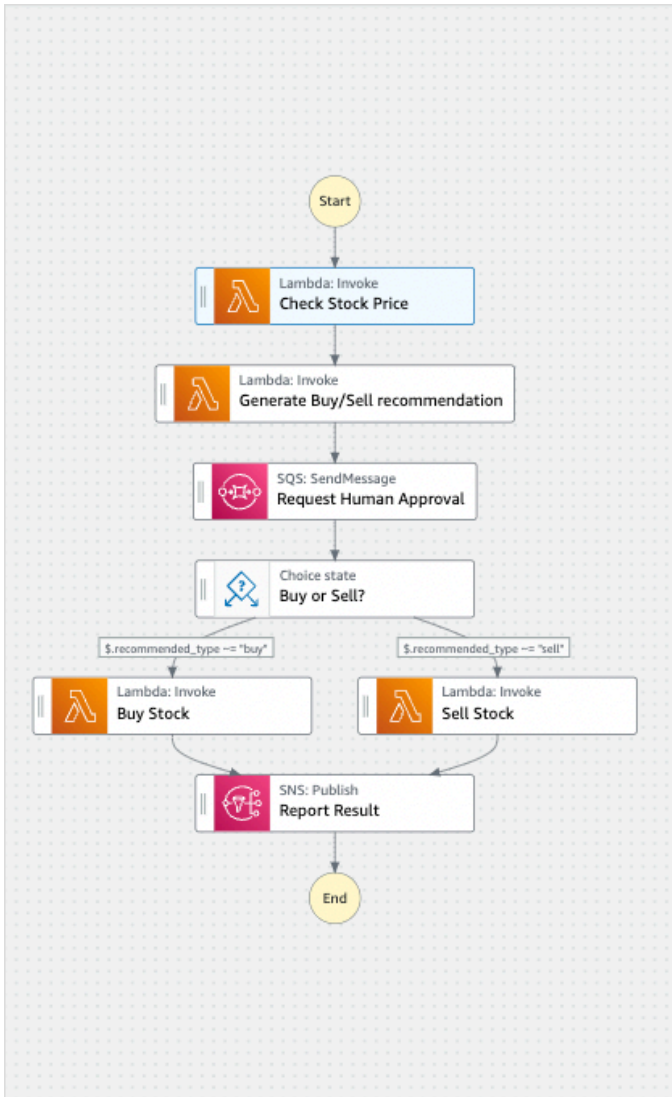


To change the order of a state, you can drag it to a different place in your workflow.



Inspector

You can configure any states that you add to your workflow. Choose the state you want to configure, and you will see its configuration options in the **Inspector** panel. To see the auto-generated [ASL definition](#) for your workflow code, turn on the **Definition** toggle. The ASL definition associated with the state you've selected will appear highlighted.



Check Stock Price Definition >

Configuration | Input | Output | Error handling

State name
Check Stock Price

API
Lambda: Invoke

Integration type [Info](#)
The type of service integration to use. [Learn more](#)

Optimized

API Parameters Edit as JSON

Function name
The Lambda function to invoke

Enter function name

arn:aws:lambda:us-east-1: :function:StepFunctionsSample-Hello

Must be a valid function name.

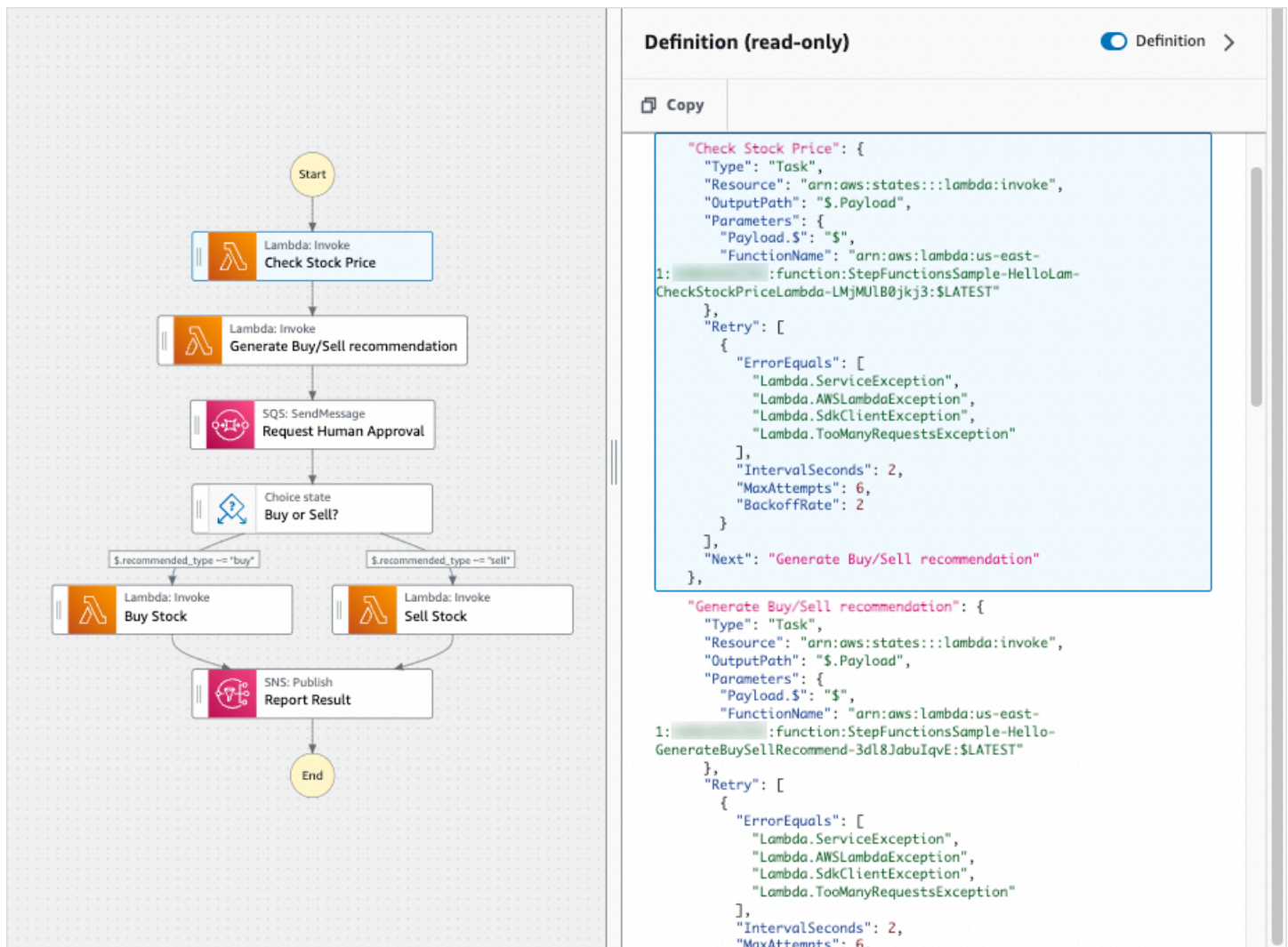
[View function](#)

Payload
The JSON that you want to provide to your Lambda function.

Use state input as payload

Additional configuration

- Wait for callback - *optional*
Pause the execution at this state until the execution receives a callback from SendTaskSuccess or SendTaskFailure APIs with the task token.



The screenshot displays the AWS Step Functions console in Code mode. On the left, a workflow diagram shows the following steps:

- Start** (yellow circle)
- Check Stock Price** (Lambda: Invoke)
- Generate Buy/Sell recommendation** (Lambda: Invoke)
- Request Human Approval** (SQS: SendMessage)
- Buy or Sell?** (Choice state)
 - Path 1: `$.recommended_type == "buy"` leads to **Buy Stock** (Lambda: Invoke)
 - Path 2: `$.recommended_type == "sell"` leads to **Sell Stock** (Lambda: Invoke)
- Report Result** (SNS: Publish)
- End** (yellow circle)

On the right, the ASL definition is shown in Code mode. The definition includes the following tasks:

```

"Check Stock Price": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "OutputPath": "$$.Payload",
  "Parameters": {
    "Payload.$": "$",
    "FunctionName": "arn:aws:lambda:us-east-1:123456789012:function:StepFunctionsSample-HelloLambda-CheckStockPrice-Lambda-LMjMULB0jkj3:$LATEST"
  },
  "Retry": [
    {
      "ErrorEquals": [
        "Lambda.ServiceException",
        "Lambda.AWSLambdaException",
        "Lambda.SdkClientException",
        "Lambda.TooManyRequestsException"
      ],
      "IntervalSeconds": 2,
      "MaxAttempts": 6,
      "BackoffRate": 2
    }
  ],
  "Next": "Generate Buy/Sell recommendation"
},
"Generate Buy/Sell recommendation": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "OutputPath": "$$.Payload",
  "Parameters": {
    "Payload.$": "$",
    "FunctionName": "arn:aws:lambda:us-east-1:123456789012:function:StepFunctionsSample-HelloLambda-GenerateBuySellRecommend-3dL8JabuIqvE:$LATEST"
  },
  "Retry": [
    {
      "ErrorEquals": [
        "Lambda.ServiceException",
        "Lambda.AWSLambdaException",
        "Lambda.SdkClientException",
        "Lambda.TooManyRequestsException"
      ],
      "IntervalSeconds": 2,
      "MaxAttempts": 6
    }
  ]
}

```

Code mode

The **Code** mode of Workflow Studio provides an integrated code editor to view, write, and edit the [Amazon States Language](#) (ASL) definition of your workflows within the Step Functions console. The following image shows the different components available in the **Code** mode.

The screenshot displays the AWS Step Functions console interface for a state machine named "MyStateMachine". The interface is divided into two main panes: the Code editor on the left and the Graph visualization pane on the right. The Code editor shows the ASL definition in JSON format, with syntax highlighting and a toolbar for editing (Undo, Redo, Format, Copy, Commands, View docs). The Graph visualization pane shows a flowchart of the state machine's execution path, starting with a "Start" node, followed by "Check Stock Price", "Generate Buy/Sell Recommendation", "Request Human Approval", a "Choice state" for "Buy or Sell?", and finally "Buy Stock" or "Sell Stock" tasks leading to "Report Result" and "End".

```

1 {
2   "Comment": "A description of my state machine",
3   "StartAt": "Check Stock Price",
4   "States": {
5     "Check Stock Price": {
6       "Type": "Task",
7       "Resource": "arn:aws:states:::lambda:invoke",
8       "OutputPath": "$$.Payload",
9       "Parameters": {
10        "Payload.$": "$",
11        "FunctionName": "arn:aws:lambda:us-east-1:123456789012:function:Step
12      },
13      "Retry": [
14        {
15          "ErrorEquals": [
16            "Lambda.ServiceException",
17            "Lambda.AWSLambdaException",
18            "Lambda.SdkClientException",
19            "Lambda.TooManyRequestsException"
20          ],
21          "IntervalSeconds": 2,
22          "MaxAttempts": 6,
23          "BackoffRate": 2
24        }
25      ],
26      "Next": "Generate Buy/Sell Recommendation"
27    },
28    "Generate Buy/Sell Recommendation": {
29      "Type": "Task",
30      "Resource": "arn:aws:states:::lambda:invoke",
31      "OutputPath": "$$.Payload",
32      "Parameters": {
33        "Payload.$": "$",
34        "FunctionName": "arn:aws:lambda:us-east-1:123456789012:function:Step
35      },
36      "Retry": [

```

1. Mode buttons – Switch between the **Design**, **Code**, and **Config** modes of the Workflow Studio using the mode buttons. You can't switch modes if the JSON in the ASL definition of your workflow is invalid.
2. The [Code editor](#) is where you write and edit the [ASL definition](#) of your workflows within the Workflow Studio. The code editor also provides features, such as syntax highlighting and auto-completion.
3. [Graph visualization pane](#) – Shows a real-time graphical visualization of your workflow.
4. Utility buttons – A set of buttons to perform tasks, such as saving your workflows or exporting their ASL definitions in a JSON or YAML file.
5. Code toolbar – Contains a set of buttons to perform common actions, such as undoing an action or formatting the code.
6. Graph toolbar – Contains a set of buttons to perform common actions, such as zooming in and zooming out the workflow graph.

Code editor

The code editor provides an IDE-like experience to write and edit your workflow definitions using JSON within the Workflow Studio. The code editor includes several features, such as syntax highlighting, auto-complete suggestions, [ASL definition](#) validation, and context-sensitive help display. As you update your workflow definition, the [Graph visualization pane](#) renders a real-time graph of your workflow. You can also see the updated workflow graph in the [Design mode](#).

If you select a state in the [Design mode](#) or the graph visualization pane, the ASL definition of that state appears highlighted in the code editor. The ASL definition of your workflow is automatically updated if you reorder, delete, or add a state in the **Design** mode or the graph visualization pane.

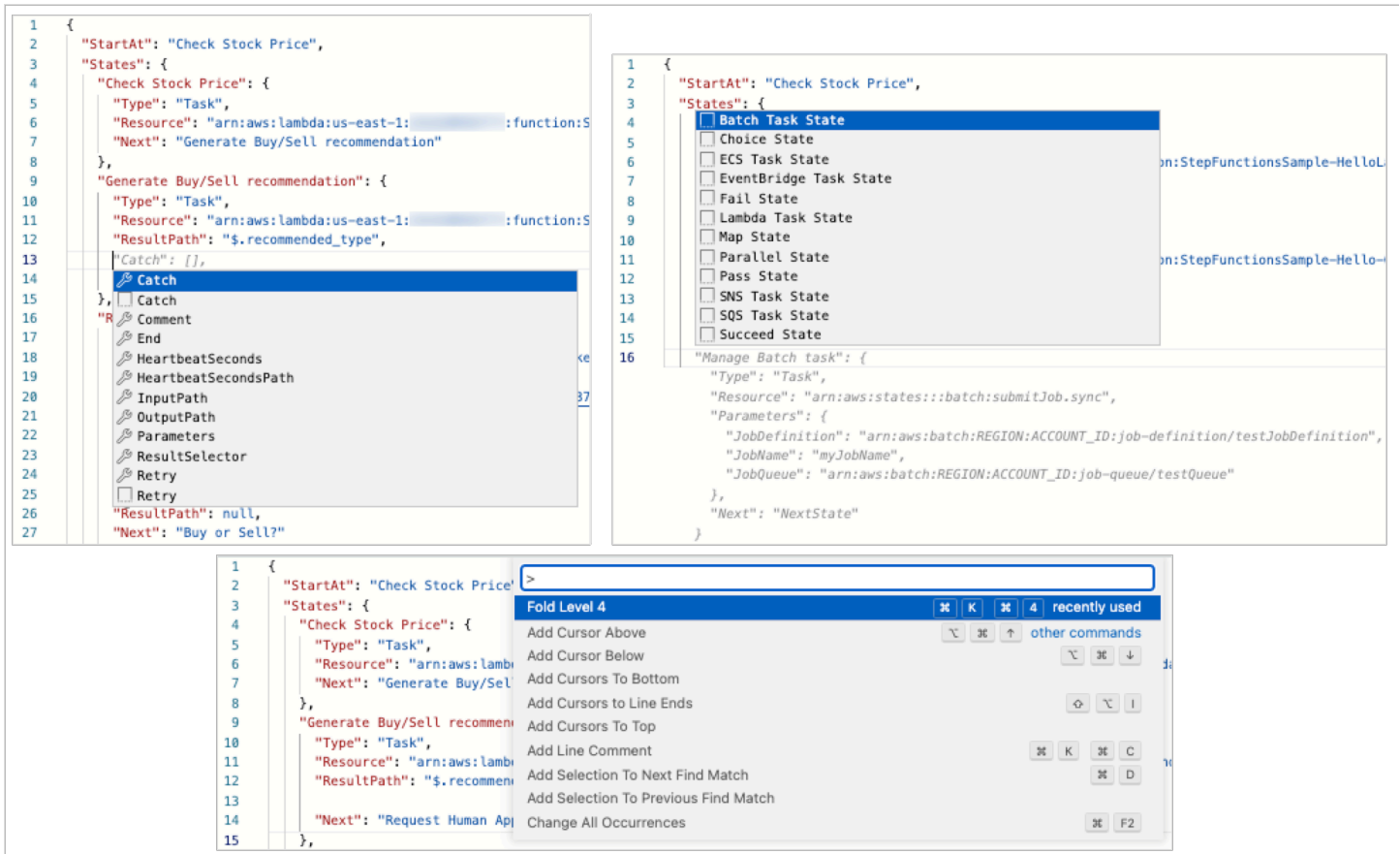
```
1  {
2    "StartAt": "Check Stock Price",
3    "States": {
4      "Check Stock Price": {
5        "Type": "Task",
6        "Resource": "arn:aws:lambda:us-east-1: :function:StepFun
7        "Next": "Generate Buy/Sell recommendation"
8      },
9      "Generate Buy/Sell recommendation": {
10       "Type": "Task",
11       "Resource": "arn:aws:lambda:us-east-1: :function:StepFun
12       "ResultPath": "$.recommended_type",
13       "Next": "Request Human Approval"
14     },
15     "Request Human Approval": {
16       "Type": "Task",
17       "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
18       "Parameters": {
19         "QueueUrl": "https://sqs.us-east-1.amazonaws.com/ /Sto
20         "MessageBody": {
21           "Input.$": "$",
22           "TaskToken.$": "$$.Task.Token"
23         }
24     }
25   }
26 }
```

Place the cursor over any field in the workflow definition to view its context-sensitive help as a tooltip.

```
1 {
2   "StartAt": "Check Stock Price",
3   "States": {
4     "Check Stock Price": {
5       "Type": "Task",
6       "Resource": "arn:aws:lambda:us-east-1: :function:StepFunctionsSampl",
7       "Next": "Generate Buy/Sell recommendation"
8     },
9     "Generate Buy/Sell recommendation": {
10      "Type": "Task",
11      "Resource": "arn:aws:lambda:us-east-1: :function:StepFunctionsSampl",
12      "ResultPath": "$.recommended_type",
13      "Next": "Request Human Approval"
14    },
15    "R"
16  }
17  "Parameters": {
18    "QueueUrl": "https://sqs.us-east-1.amazonaws.com/ /StepFunctions",
19    "MessageBody": {
20      "Input.$": "$",
21      "TaskToken.$": "$$.Task.Token"
22    }
23  }
```

Used to pass information to the API actions of connected resources. The Parameters can use a mix of static JSON, JsonPath and intrinsic functions.

The auto-complete suggestions displays code snippets for the fields or states that you can include in your workflows. To see a list of fields you can include within a specific state, press **Ctrl+Space**. To generate code snippet for a new state in your workflow press **Ctrl+Space** after the current state's definition. You can also press **F1** to display a list of available commands.



Graph visualization pane

Graph visualizations let you see what your workflow looks like in graphical format. When you write your workflow definitions in the [Code editor](#) of Workflow Studio, the graph visualization pane renders a real-time graph of your workflow. As you reorder, delete, or duplicate a state in the graph visualization pane, the workflow definition in the Code editor is automatically updated. Similarly, as you update your workflow definitions, reorder, delete, or add a state in the Code editor, the visualization is automatically updated.

If the JSON in the ASL definition of your workflow is invalid, the graph visualization pane pauses the rendering and displays a status message at the bottom of the pane.

Config mode

The **Config** mode of Workflow Studio lets you manage the configuration of your state machines. In this mode, you can specify details, such as state machine name and its type, IAM permissions, and logging configuration for the state machine. Other additional configurations that you can specify in this mode include enabling AWS X-Ray tracing and publishing a version when you

create the state machine. After you've created the state machine, you can edit all state machine configuration options except the state machine name and type. The following image shows some of the configurations you can specify in the **Config** mode.

The screenshot shows the AWS Step Functions Config mode interface. At the top, there are tabs for 'Design', 'Code', and 'Config', with 'Config' selected. The main content area is titled 'State machine configuration' and includes a 'Feedback' button. The configuration is divided into three sections: 'Details', 'Permissions', and 'Logging'.

Details

State machine name
State machine name cannot be changed after creation.
WorkflowStudio
Must be 1-80 characters. Can use alphanumeric characters, dashes, and underscores.

Type [Info](#)
State machine type cannot be changed after creation.

- Standard**
Durable workflows for ETL, ML, e-commerce and automation. They can run for up to 1 year, and history is stored in Step Functions for auditing and playback. Supported by a feature-rich console debugger. Recommended for new users.
- Express**
Low cost, high scale workflows for streaming data processing and microservice APIs. They can run for up to 5 minutes, and history can be streamed to CloudWatch Logs.

Permissions [Info](#)

Execution role
The IAM role that defines which resources your state machine has permission to access during execution. To create a custom role, go to the [IAM console](#).

Create new role

An execution role will be created with full permissions.
A new execution role named `StepFunctions-WorkflowStudio-role-591phu8kk` will be created. All required permissions for the actions specified in your state machine will be auto-generated.

► [Review auto-generated permissions](#)

Logging [Info](#)
You can log your state machine's execution history to CloudWatch Logs. For Express state machines, you must enable logging to inspect and debug executions. CloudWatch Logs charges apply. [Learn more](#)

Log level
Indicates which execution history events to log
OFF

Manage state machine configuration

To manage your state machine configuration, do the following:

1. Enter a name for your state machine in the **State machine name** box.

 **Tip**

Alternatively, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, under **State machine configuration**, specify a name.

 **Important**

You can't edit the state machine name after you've created the state machine.

2. In **Type**, choose a state machine type of **Standard** or **Express**. For information about state machine types, see [Standard vs. Express Workflows](#).

 **Important**

You can't edit the state machine type after you've created the state machine.

3. In **Permissions**, select the IAM role to be used as the execution role for the state machine.

- **Create new role (Recommended)**: If you select this option, Step Functions automatically creates an execution role for your state machines with the least privileges required when you create the state machines. These automatically generated IAM roles are valid for the AWS Region in which you create the state machine.

 **Tip**

To review the permissions that Step Functions will automatically generate for your state machine, choose **Review auto-generated permissions**.

 **Note**

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

- **Choose an existing role:** Create your own IAM role for the state machine and then choose it from the options listed below **Choose an existing role**. Make sure the role's policy includes the permissions that you would like the state machine to assume.

For information about creating IAM policies, see [Creating IAM policies](#) in the *IAM User Guide*.

- **Enter a role ARN:** Specify the Amazon Resource Name (ARN) of an existing IAM role to use for this state machine. For example, `arn:aws:iam::123456789012:role/service-role/StepFunctions-WorkflowStudio-role-777f4027`.
4. In **Logging**, set the log level for your state machine. Step Functions logs the execution history events based on your selection. You can select one of the following options:
- **ALL:** All event types are logged.
 - **ERROR:** All error event types are logged, such as `TaskFailed` and `ExecutionFailed`.
 - **FATAL:** All fatal error event types are logged, such as `ExecutionAborted` and `ExecutionFailed`.
 - **OFF:** No event types are logged.

For more information about log levels, see [Log levels](#).

5. In **Additional configuration**, set one or more of the following optional configurations:
- **Enable X-Ray tracing:** Choose this checkbox for Step Functions to send traces to X-Ray for state machine executions, even when a trace ID is not passed by an upstream service. For more information, see [AWS X-Ray and Step Functions](#).
 - **Publish version on creation:** A *version* is a numbered, immutable snapshot of a state machine that you can run. Choose this checkbox to publish a version of your state machine while creating the state machine. Step Functions publishes version 1 as the first revision of the state machine.

For more information about versions, see [State machine versions](#).

- **Add new tag:** Choose this box to add tags to your state machine. Adding tags can help you track and manage the costs associated with your resources, and provide better security in your IAM policies. For more information about tags, see [Tagging in Step Functions](#).
6. Choose **Create**.
7. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role configuration** to go back to the **Config** mode.

Keyboard shortcuts

Workflow Studio supports the following keyboard shortcuts:

Keyboard shortcut	Function
Shortcuts for the Code mode	
Ctrl+space	Auto-complete suggestions
F1	Display a list of available commands
Common shortcuts for the Design and Code modes	
Ctrl+Z	Undo the last operation
Ctrl+Shift+Z	Redo the last operation
Alt+C	Center the workflow in the canvas
Backspace	Remove all selected states
Delete	Remove all selected states
Ctrl+D	Duplicate selected state

Using Workflow Studio

Learn to create, edit, and run workflows using Step Functions Workflow Studio. After your workflow is ready, you can export it. You can also use Workflow Studio for rapid prototyping.

In this topic

- [Create a workflow](#)
- [Design a workflow](#)
- [Run your workflow](#)
- [Edit your workflow](#)
- [Export your workflow](#)
- [Create your workflow prototype](#)

Create a workflow

In Workflow Studio, you can either choose a starter template, or choose a blank template to create a workflow from scratch. For blank templates, you can use the [Design](#) or [Code](#) mode to create your workflow.

A starter template is a ready-to-run sample project that automatically creates the workflow prototype and definition, and deploys all the related AWS resources that your project needs to your AWS account. You can use these starter templates to deploy and run them as is, or use the workflow prototypes to build on them. For more information about starter templates, see [Starter templates for Step Functions](#).

Create a workflow using starter templates

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. In the **Choose a template** dialog box, do one of the following to choose a sample project, for example, the **Task Timer** sample project:
 - Type **Task Timer** in the Search by keyword box, and then choose **Task Timer** from the search results that are returned.
 - Browse through the sample projects listed under **All** on the right pane, and then choose **Task Timer**.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.
5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition. In [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) for updating the [Amazon States Language](#) (ASL) definition of your workflow.

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

ℹ Note

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

⚠ Important

Standard charges apply for each service used in the CloudFormation template.

Create a workflow using a blank template

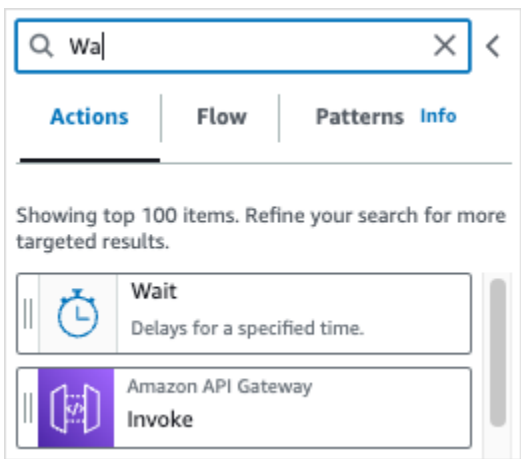
1. Open the [Step Functions console](#).
2. Choose **Create state machine**.
3. In the **Choose a template** dialog box, select **Blank**.
4. Choose **Select**. This opens Workflow Studio in [Design mode](#).

You can now start designing your workflow in [Design mode](#) or writing your workflow definition in [Code mode](#).

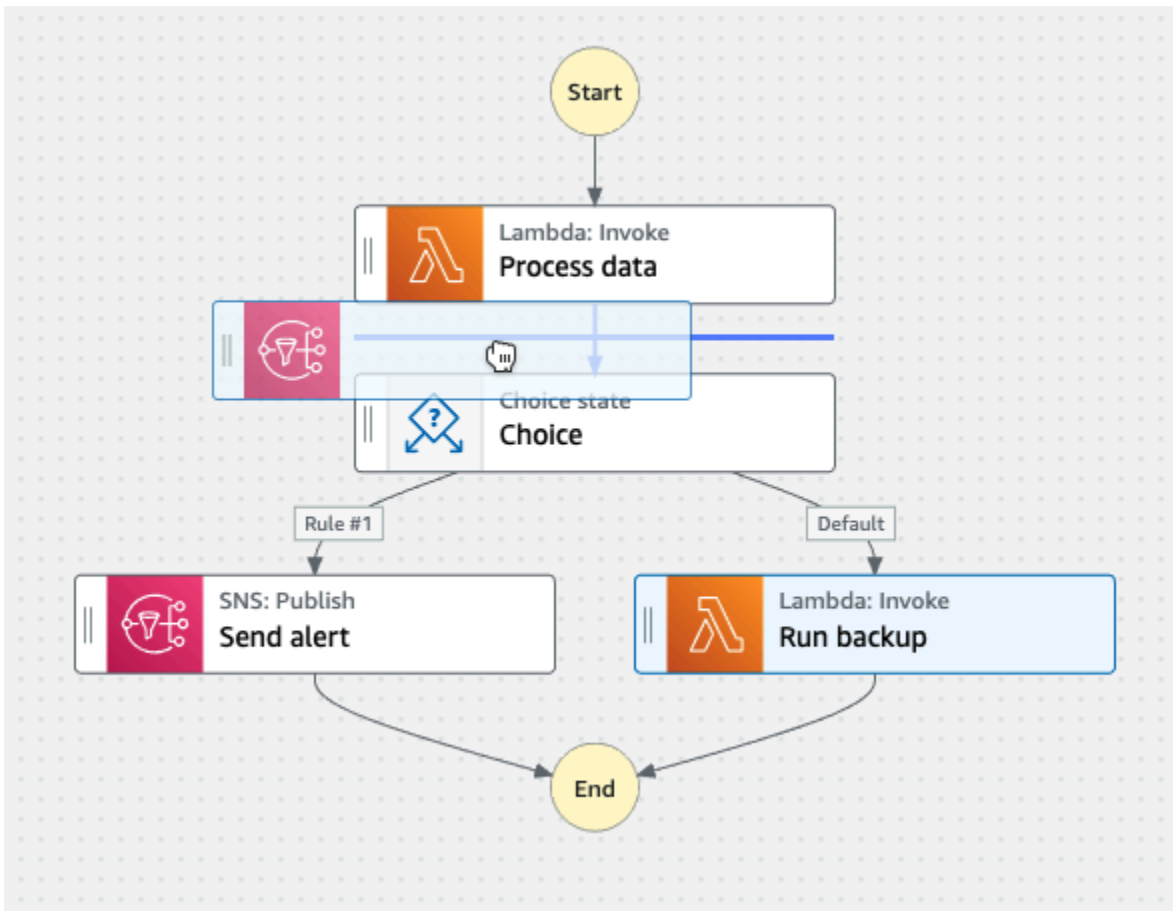
5. Choose **Config** to manage the configuration of your workflow in the [Config mode](#). For example, provide a name for your workflow and choose its type.

Design a workflow

If you know the name of the state you want to add, use the search box at the top of the [States browser](#) to find that state in the **Actions** and **Flow** tabs of the [Design mode](#).



Otherwise, choose a state from the states browser and drag and drop it onto the canvas, placing it where you want in your workflow. You can also reorder states in your workflow by dragging them to a different location in your workflow. As you drag a state onto the canvas, a line appears wherever you can drop it in your workflow. After a state is dropped onto the canvas, its code is auto-generated and added inside your workflow definition. To see the definition, turn on the **Definition** toggle on the [Inspector panel](#). To make edits to your workflow definition, choose the [Code mode](#) that offers an integrated code editor.



After you drop a state onto the canvas, you can configure it in the [Inspector](#) panel on the right. This panel contains the **Configuration**, **Input**, **Output**, and **Error Handling** tabs for each of the state or API action that you place on the canvas. You configure the states you include in your workflows in the **Configuration** tab. For example, the **Configuration** tab for Lambda Invoke API action consists of the following options:

State name 1
Lambda Invoke

API 2
Lambda: Invoke

Integration type Info 3
The type of service integration to use. [Learn more](#)

Optimized

API Parameters Edit as JSON

Function name 4
The Lambda function to invoke

Choose an option

Payload 5
The JSON that you want to provide to your Lambda function.

Use state input as payload

Additional configuration

Wait for callback - optional 6
Pause the execution at this state until the execution receives a callback from `SendTaskSuccess` or `SendTaskFailure` APIs with the task token.

IAM role for cross-account access - optional Info 7
When your execution reaches this state, it can access cross-account resources by assuming an IAM role with appropriate permissions in the target account.

Choose an option

Next state 8
Go to end

Comment - optional 9
Enter comment

1. The **State name** identifies the state. You can use your own name or accept the default generated name.
2. The **API** shows the API action used by the state.
3. The **Integration type** dropdown list provides options to choose the [type](#) of service integrations available in Step Functions. The integration type you choose is used to call API actions of a specific AWS service from your workflow.

4. The **Function name** provides options to:

- **Enter a function name:** You can enter your function name or its ARN.
- **Get function name at runtime from state input:** You can use this option to dynamically get the function name from the state input based on the path you specify.
- **Select function name:** You can directly select from the functions available in your account and region.

5. The **Payload** lets you select from the following options:

- **Use state input as payload:** You can use this option to pass the state's input as the payload provided to your Lambda function.
- **Enter your own payload:** You can use this option to construct a JSON object to pass as the payload to your Lambda function. This JSON can include both static values and values selected from the state input.
- **No payload:** You can use this option if you don't want to pass any payload to your Lambda function.

6. (Optional) Some states will have an option to select **Wait for task to complete** or **Wait for callback**. When available, these options select one of the following [service integration patterns](#):

- **No option selected:** Step Functions will use the [Request Response](#) integration pattern. Step Functions will wait for an HTTP response and then progress to the next state. Step Functions will not wait for a job to complete. When no options are available, the state will use this pattern.
- **Wait for task to complete:** Step Functions will use the [Run a Job \(.sync\)](#) integration pattern.
- **Wait for callback:** Step Functions will use the [Wait for a Callback with the Task Token](#) integration pattern.

7. (Optional) To access resources configured in different AWS accounts within your workflows, Step Functions provides [cross-account access](#). **IAM role for cross-account access** provides options to:

- **Provide IAM role ARN:** Specify the IAM role that contains appropriate resource access permissions. These resources are available in a target account, which is an AWS account to which you make cross-account calls.
- **Get IAM role ARN at runtime from state input:** Specify a reference path to an existing key-value pair in the state's JSON input which contains the IAM role.

8. **Next state** lets you to select the state you want to transition to next.

9. (Optional) The **Comment** field can be used to add your own comment. It will not affect the workflow, but can be used to annotate your workflow.

Some states will have more generic configuration options. For example, the Amazon ECS RunTask state configuration contains an `API Parameters` field populated with placeholder values.

Run configuration
Definition >

Configuration
Input
Output
Error handling

State name

API
ECS: RunTask

Integration type [Info](#)
The type of service integration to use. [Learn more](#)

API Parameters
JSON object containing the parameters to pass into this API. Contains sample values. Update the JSON with your own parameter values. Note: parameter names must be in PascalCase.

```

1 {
2   "LaunchType": "FARGATE",
3   "Cluster": "arn:aws:ecs:REGION:ACCOUNT_ID:cluster/MyECSCluster",
4   "TaskDefinition": "arn:aws:ecs:REGION:ACCOUNT_ID:task-definition/MyTaskDefinition",
5 }
```

Must be valid JSON. To reference a node in this state's JSON input, the key must end with ".\$" (for example "key2.\$": "\$.inputValue"). [Info](#)

Wait for task to complete - optional
Pause the execution at this state and monitor the task. Resume the execution once the task is complete. Additional permissions required. [Learn more](#)

Wait for callback - optional
Pause the execution at this state until the execution receives a callback from `SendTaskSuccess` or `SendTaskFailure` APIs with the task token.

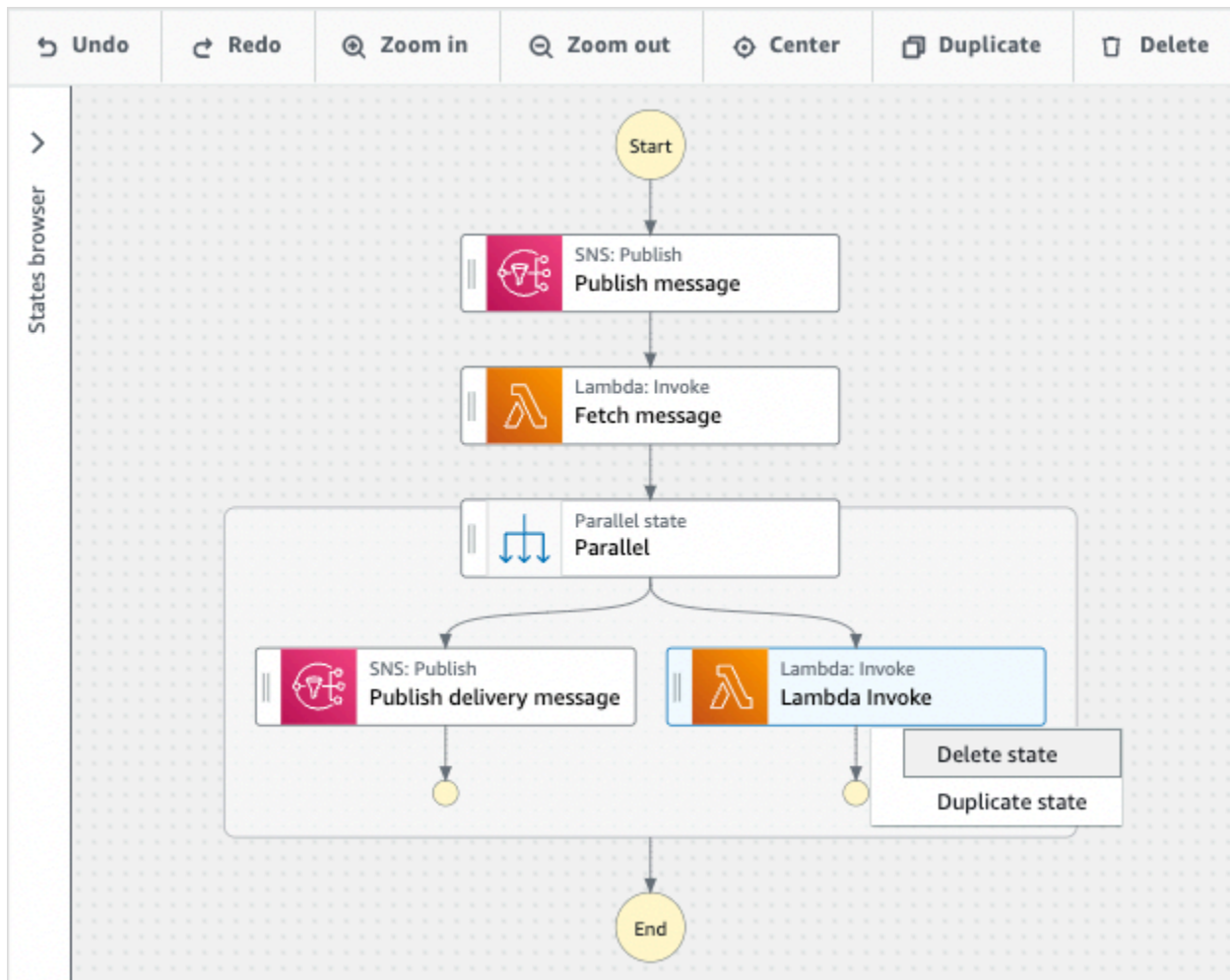
IAM role for cross-account access - optional [Info](#)
When your execution reaches this state, it can access cross-account resources by assuming an IAM role with appropriate permissions in the target account.

Next state

Comment - optional

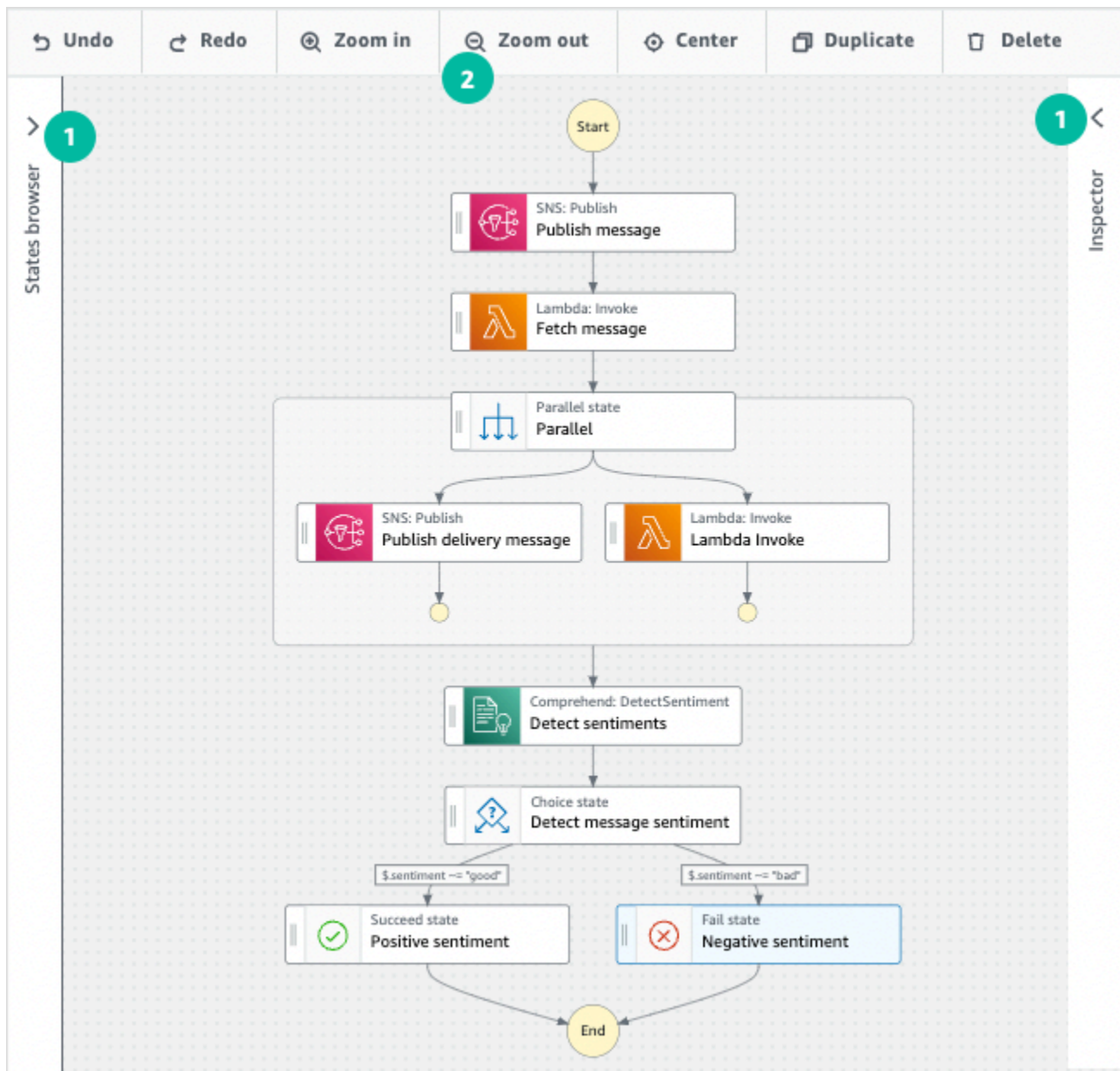
For these states, you can replace the placeholder values with configurations that are suited to your needs.

To delete a state, you can use backspace, right-click and choose **Delete state**, or choose **Delete** on the [Design toolbar](#).



As your workflow grows, it may not fit in the canvas. You can:

1. Use the controls on the side panels to resize or close the panels.
2. Use the Design toolbar controls at the top of the [Canvas](#) to zoom the workflow graph in or out.



Run your workflow

After you create or edit your workflow with the Workflow Studio, you can run it and view its execution in the [Step Functions console](#).


To run a workflow in Workflow Studio

1. In the **Design**, **Code**, or **Config** mode, choose **Execute**.

The **Start execution** dialog box opens in a new tab.

2. In the **Start execution** dialog box, do the following:

1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

 **Note**

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.
3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

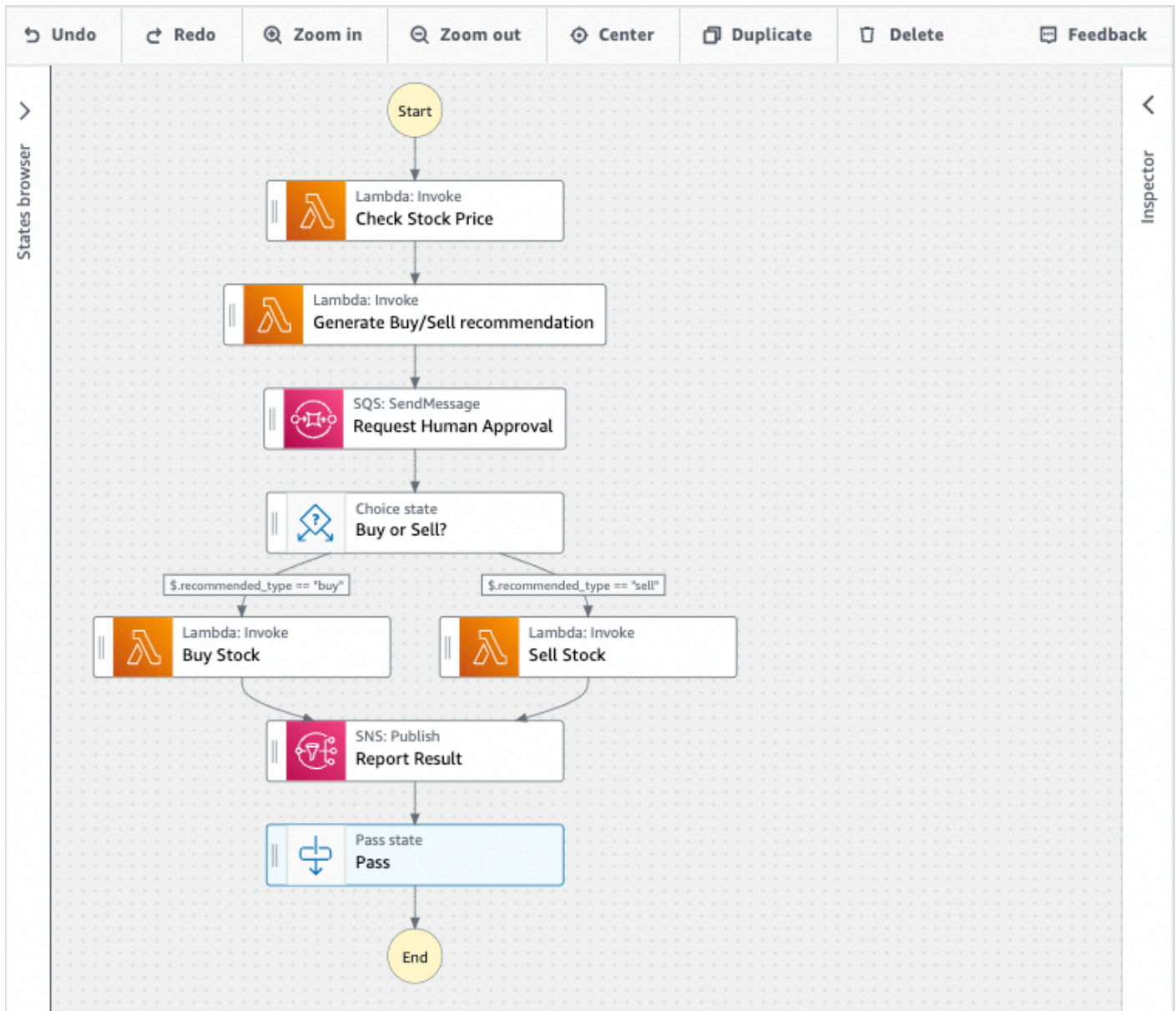
To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Edit your workflow

You can edit an existing workflow visually in the [Design mode](#) of Workflow Studio. You can also edit the workflow definition in [Code mode](#) of Workflow Studio.

To edit an existing workflow:

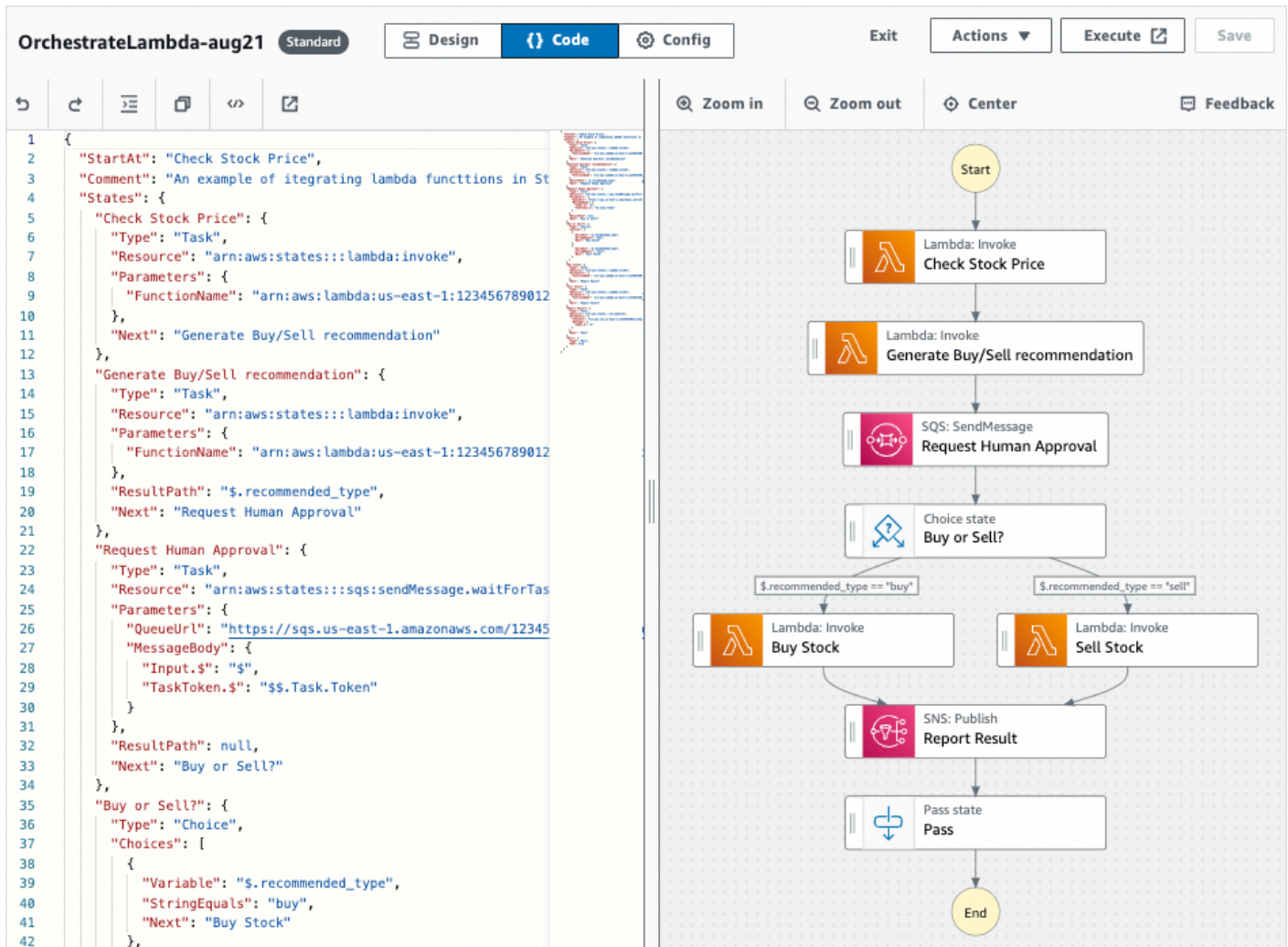
1. Open the [Step Functions console](#).
2. On the **State machines** page, choose the workflow you want to edit.
3. On the *State machine detail* page, choose **Edit**.
4. The workflow opens in **Design** mode of Workflow Studio. Edit the workflow as required.



Note

If you see errors in your workflow, you must fix them in **Design** mode. You can't switch to the **Code** or **Config** mode if any errors exist in your workflow.

5. (Optional) Choose the **Code** button to view or edit the workflow definition in Workflow Studio.



6. When you are done, choose **Save** to save your updated workflow.
7. (Optional) To run your updated workflow, choose **Execute**. The **Start execution** dialog box opens in a new tab.

Export your workflow

You can export your workflow's [Amazon States Language](#) (ASL) definition and your workflow graph:

1. Choose your workflow in the [Step Functions console](#).
2. On the *State machine detail* page, choose **Edit**.
3. (Optional) Your workflow opens in **Design** mode of Workflow Studio. [Edit your workflow](#) in **Design** mode or switch to the **Code** mode.
4. Choose the **Actions** dropdown button, and then do one or both of the following:

- To export the workflow graph to an SVG or PNG file, under **Export graph**, select the format you want.
- To export the workflow definition as a JSON or YAML file, under **Export definition**, select the format you want.

Create your workflow prototype

You can use Workflow Studio to create prototypes of new workflows that contain placeholder resources. You can also build your workflows using [Workflow Studio in Application Composer](#). To create a prototype:

1. Sign in to the [Step Functions console](#).
2. Choose **Create state machine**.
3. In the **Choose a template** dialog box, select **Blank**.
4. Choose **Select**. This opens Workflow Studio in [Design mode](#).
5. The [Design mode](#) of Workflow Studio opens. Design your workflow in Workflow Studio. To include placeholder resources:
 - a. Choose the state for which you want to include a placeholder resource, and then in **Configuration**:
 - For Lambda Invoke states, choose **Function name**, then choose **Enter function name**. You can also enter a custom name for your function.
 - For Amazon SQS Send Message states, choose **Queue URL**, then choose **Enter queue URL**. Enter a placeholder queue URL.
 - For Amazon SNS Publish states, from **Topic**, choose a topic ARN.
 - For all other states listed under **Actions**, you can use the default configuration.
 - b. (Optional) To view the auto-generated ASL definition of your workflow, choose **Definition**.
 - c. (Optional) To update the workflow definition in Workflow Studio, choose the **Code** button.

Note

If you see errors in your workflow, you must fix them in **Design** mode. You can't switch to the **Code** or **Config** mode if any errors exist in your workflow.

Note

If you see errors in your workflow definition, you must fix them in **Code** mode. You can't switch to the **Design** or **Config** mode if any errors exist in your workflow definition.

6. (Optional) To edit the state machine name, choose the edit icon next to the default state machine name of **MyStateMachine** and specify a name in the **State machine name** box.

You can also switch to the [Config mode](#) to edit the default state machine name.

7. Specify your workflow settings, such as state machine type and its execution role.

8. Choose **Create**.

You've now created a new workflow with placeholder resources that can be used to prototype. You can [export](#) your workflow definition and the workflow graph.

- To export your workflow definition as a JSON or YAML file, in the **Design** or **Code** mode, choose the **Actions** dropdown button. Then, under **Export definition**, select the format you want to export. You can use this exported definition as the starting point for local development with the [AWS Toolkit for Visual Studio Code](#).
- To export your workflow graph to an SVG or PNG file, in the **Design** or **Code** mode, choose the **Actions** dropdown button. Then, under **Export definition**, select the format you want.

Configure inputs and outputs for your states

Each state makes a decision or performs an action based on input that it receives. In most cases, it then passes output to other states. In Workflow Studio, you can configure how a state filters and manipulates its input and output data in the **Input** and **Output** tabs of the [Inspector](#) panel. Use the **Info** links to access contextual help when configuring inputs and outputs.

For detailed information about how Step Functions processes input and output, see [Input and Output Processing in Step Functions](#).

Configure input to a state

Each state receives input from the previous state as JSON. If you want to filter the input, you can use the [InputPath](#) filter under the **Input** tab in the [Inspector](#) panel. The InputPath is a string, beginning with \$, that identifies a specific JSON node. These are called [reference paths](#), and they follow JsonPath syntax.

Configuration | **Input** | Output | Error handling

During workflow execution, a task state's input comes from the previous state's output. [Info](#)

Filter input with InputPath - optional [Info](#)
Use the InputPath filter to select a portion of the state input to use.

To filter the input:

- Choose **Filter input with InputPath**.
- Enter a valid [JsonPath](#) for the InputPath filter. For example, `$.data`.

Your InputPath filter will be added to your workflow.

Example Example 1: Use InputPath filter in Workflow Studio

Say the input to your state includes the following JSON data.

```
{
  "comment": "Example for InputPath",
  "dataset1": {
    "val1": 1,
    "val2": 2,
    "val3": 3
  },
  "dataset2": {
    "val1": "a",
    "val2": "b",
    "val3": "c"
  }
}
```

To apply the InputPath filter, choose **Filter input with InputPath**, then enter an appropriate reference path. If you enter `$.dataset2.val1`, the following JSON is passed as input to the state.

```
{"a"}
```

A reference path can also have a selection of values. If the data you reference is `{ "a": [1, 2, 3, 4] }` and you apply the reference path `$.a[0:2]` as the InputPath filter, the following is the result.

```
[ 1, 2 ]
```

[Parallel](#), [Map](#), and [Pass](#) flow states have an additional input filtering option called Parameters under their **Input** tab. This filter takes effect after the InputPath filter and can be used to construct a custom JSON object consisting of one or more key-value pairs. The values of each pair can either be static values, can be selected from the input, or can be selected from the [Context object](#) with a path.

Note

To specify that a parameter uses a reference path to point to a JSON node in the input, the parameter name must end with `.$`.

Example Example 2: Create custom JSON input for Parallel state

Say the following JSON data is the input to a Parallel state.

```
{
  "comment": "Example for Parameters",
  "product": {
    "details": {
      "color": "blue",
      "size": "small",
      "material": "cotton"
    },
    "availability": "in stock",
    "sku": "2317",
    "cost": "$23"
  }
}
```

To select part of this input and pass additional key-value pairs with a static value, you can specify the following in the **Parameters** field, under the **Parallel** state's **Input** tab.

```
{
  "comment": "Selecting what I care about.",
  "MyDetails": {
    "size.$": "$.product.details.size",
    "exists.$": "$.product.availability",
    "StaticValue": "foo"
  }
}
```

The following JSON data will be the result.

```
{
  "comment": "Selecting what I care about.",
  "MyDetails": {
```

```
"size": "small",
"exists": "in stock",
"StaticValue": "foo"
}
}
```

Configure output of a state

Each state produces JSON output that can be filtered before it is passed to the next state. There are several filters available, and each affects the output in a different way. Output filters available for each state are listed under the **Output** tab in the **Inspector** panel. For [Task state](#) states, any output filters you select are processed in this order:

1. [ResultSelector](#): Use this filter to manipulate the state's result. You can construct a new JSON object with parts of the result.
2. [ResultPath](#): Use this filter to select a combination of the state input and the task result to pass to the output.
3. [OutputPath](#): Use this filter to filter the JSON output to choose which information from the result will be passed to the next state.

Configuration

Input

Output

Error handling

During execution, the task state calls an API and the response goes into the *task result*. The result can be manipulated with filters before it is passed as output to the next state [Info](#)

- Transform result with ResultSelector - optional** [Info](#)
Use the ResultSelector filter to construct a new JSON object using parts of the task result.
- Combine input and result with ResultPath - optional** [Info](#)
Use the ResultPath filter to add the result into the original state input. The specified path indicates where to add the result.
- Filter output with OutputPath - optional** [Info](#)
Use the OutputPath filter to select a portion of the effective output to pass to the next state.

Use ResultSelector

ResultSelector is an optional output filter for the following states:

- [Task state](#) states, which are all states listed in the **Actions** tab of the [States browser](#).
- [Map](#) states, in the **Flow** tab of the States browser.
- [Parallel](#) states, in the **Flow** tab of the States browser.

ResultSelector can be used to construct a custom JSON object consisting of one or more key-value pairs. The values of each pair can either be static values or selected from the state's result with a path.

Note

To specify that a parameter uses a path to reference a JSON node in the result, the parameter name must end with `.$`.

Example Example to use ResultSelector filter

In this example, you use ResultSelector to manipulate the response from the Amazon EMR CreateCluster API call for an Amazon EMR CreateCluster state. The following is the result from the Amazon EMR CreateCluster API call.

```
{
  "resourceType": "elasticmapreduce",
  "resource": "createCluster.sync",
  "output": {
    "SdkHttpMetadata": {
      "HttpHeaders": {
        "Content-Length": "1112",
        "Content-Type": "application/x-amz-JSON-1.1",
        "Date": "Mon, 25 Nov 2019 19:41:29 GMT",
        "x-amzn-RequestId": "1234-5678-9012"
      },
      "HttpStatusCode": 200
    },
    "SdkResponseMetadata": {
      "RequestId": "1234-5678-9012"
    }
  },
}
```

```
"ClusterId": "AKIAIOSFODNN7EXAMPLE"
}
```

To select part of this information and pass an additional key-value pair with a static value, specify the following in the **ResultSelector** field, under the state's **Output** tab.

```
{
  "result": "found",
  "ClusterId.$": "$.output.ClusterId",
  "ResourceType.$": "$.resourceType"
}
```

Using `ResultSelector` produces the following result.

```
{
  "result": "found",
  "ClusterId": "AKIAIOSFODNN7EXAMPLE",
  "ResourceType": "elasticmapreduce"
}
```

Use ResultPath

The output of a state can be a copy of its input, the result it produces, or a combination of its input and result. Use `ResultPath` to control which combination of these is passed to the state output. For more use cases of `ResultPath`, see [ResultPath](#).

`ResultPath` is an optional output filter for the following states:

- [Task state](#) states, which are all states listed in the **Actions** tab of the States browser.
- [Map](#) states, in the **Flow** tab of the States browser.
- [Parallel](#) states, in the **Flow** tab of the States browser.
- [Pass](#) states, in the **Flow** tab of the States browser.

`ResultPath` can be used to add the result into the original state input. The specified path indicates where to add the result.

Example Example to use ResultPath filter

Say the following is the input to a Task state.

```
{
  "details": "Default example",
  "who": "AWS Step Functions"
}
```

The result of the Task state is the following.

```
Hello, AWS Step Functions
```

You can add this result to the state's input by applying `ResultPath` and entering a reference [path](#) that indicates where to add the result, such as `$.taskresult`:

With this `ResultPath`, the following is the JSON that is passed as the state's output.

```
{
  "details": "Default example",
  "who": "AWS Step Functions",
  "taskresult": "Hello, AWS Step Functions!"
}
```

Use OutputPath

The `OutputPath` filter lets you filter out unwanted information, and pass only the portion of JSON that you care about. The `OutputPath` is a string, beginning with `$`, that identifies nodes within JSON text.

Example Example to use OutputPath filter

A Lambda Invoke API call returns metadata in addition to the payload, which is the Lambda function's result. An example of the response from this API call is shown under the **Output** tab of the state.

Lambda Invoke

Configuration

Input

Output

Error handling

During execution, the task state calls an API and the response goes into the task result. The result can be manipulated with filters before it is passed as output to the next state. [Info](#)

Lambda:Invoke task result example

A read-only example of the kind of task result to expect from this API:

```
{
  "ExecutedVersion": "$LATEST",
  "Payload": {
    "foo": "bar",
    "colors": [
      "red",
      "blue",
      "green"
    ],
    "car": {
      "year": 2008,
      "make": "Toyota",
      "model": "Matrix"
    }
  },
  "SdkHttpMetadata": {
    "AllHttpHeaders": {
      "X-Amz-Executed-Version": [
        "$LATEST"
      ]
    }
  }
}
```

Transform result with ResultSelector - *optional* [Info](#)

Use the ResultSelector filter to construct a new JSON object using parts of the task result.

You can use OutputPath to filter out the additional metadata. By default, the value of **OutputPath** filter for Lambda Invoke states created through the Workflow Studio is \$.Payload. This default value removes the additional metadata and returns an output equivalent to running the Lambda function directly.

The Lambda Invoke task result example and the value of `$.Payload` for the **Output** filter pass the following JSON data as the output.

```
{
  "foo": "bar",
  "colors": [
    "red",
    "blue",
    "green"
  ],
  "car": {
    "year": 2008,
    "make": "Toyota",
    "model": "Matrix"
  }
}
```

Note

Since the `OutputPath` filter is the last output filter to take effect, if you use additional output filters such as `ResultSelector` or `ResultPath`, you should modify the default value of `$.Payload` for the `OutputPath` filter accordingly.

Execution roles in Workflow Studio

Every Step Functions state machine requires an AWS Identity and Access Management (IAM) role which grants the state machine permission to perform actions on AWS services and resources or call third-party APIs. This role is called an *execution role*. This role must contain IAM policies for each action, for example, policies that allow the state machine to invoke an AWS Lambda function, run an AWS Batch job, or call the Stripe API. Step Functions requires you to provide an execution role in the following cases:

- You create a state machine in the console, AWS SDKs or AWS CLI using the [CreateStateMachine](#) API.
- You [test](#) a state in the console, AWS SDKs, or AWS CLI using the [TestState](#) API.

Workflow Studio has capabilities that makes it easy to manage execution roles for your workflows.

Topics

- [About auto-generated roles](#)
- [Automatically generating roles](#)
- [Resolving role generation problems](#)
- [Role for testing HTTP Tasks in Workflow Studio](#)
- [Role for testing an optimized service integration in Workflow Studio](#)
- [Role for testing an AWS SDK service integration in Workflow Studio](#)
- [Role for testing flow states in Workflow Studio](#)

About auto-generated roles

When you create a state machine in the Step Functions console, [Workflow Studio](#) can automatically create an execution role for you which contains the necessary IAM policies. Workflow Studio analyzes your state machine definition and generates policies with the least privileges necessary to execute your workflow.

Workflow Studio can generate IAM policies for the following:

- [HTTP Tasks](#) that call third-party APIs.
- Task states that call other AWS services using [optimized integrations](#), such as [Lambda Invoke](#), [DynamoDB GetItem](#), or [AWS Glue StartJobRun](#).
- Task states that run [nested workflows](#).
- [Distributed Map states](#), including [policies](#) to start child workflow executions, list Amazon S3 buckets, and read or write S3 objects.
- [X-Ray](#) tracing. Every role that is auto-generated in Workflow Studio contains a [policy](#) which grants permissions for the state machine to send traces to X-Ray.
- [Logging using CloudWatch Logs](#) when logging is enabled on the state machine.

Workflow Studio can't generate IAM policies for Task states that call other AWS services using [AWS SDK integrations](#).

Automatically generating roles

1. Open the [Step Functions console](#) and choose **Create state machine**.

You can also update an existing state machine. Refer Step 4 if you're updating a state machine.

2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).
4. Choose the **Config** tab.
5. Scroll down to the **Permissions** section, and do the following:
 - a. For **Execution role**, make sure you keep the default selection of **Create new role**.

Workflow Studio automatically generates all the required IAM policies for every valid state in your state machine definition. It displays a banner in with the message, **An execution role will be created with full permissions**.

MyStateMachine-zt9v7smr7 Design Code Config Cancel Actions Create

State machine configuration Feedback

Permissions [Info](#)

Execution role
The IAM role that defines which resources your state machine has permission to access during execution. To create a custom role, go to the [IAM console](#).

Create new role ↻

An execution role will be created with full permissions.
A new execution role named `StepFunctions-MyStateMachine-zt9v7smr7-role-w8u477ccc` will be created. All required permissions for the actions specified in your state machine will be auto-generated.

▼ Review auto-generated permissions

Service	Action(s)	Status	Documentation links
AWS Glue	glue:StartJobRun	✔ Policy will be generated to perform the action for any Glue resource	Call Glue with Step Functions Glue policies for Step Functions
Amazon SNS	sns:Publish	✔ Policy will be generated to perform the action for any SNS resource	Call SNS with Step Functions SNS policies for Step Functions
AWS Lambda	lambda:InvokeFunction	✔ Policy will be generated to perform the action for specified Lambda resources only	Call Lambda with Step Functions Lambda policies for Step Functions
AWS X-Ray	xray:PutTraceSegments xray:PutTelemetryRecords xray:GetSamplingRules xray:GetSamplingTargets	✔ Policies will be generated for X-Ray tracing	X-Ray policies for Step Functions

Tip

To review the permissions that Workflow Studio automatically generates for your state machine, choose **Review auto-generated permissions**.

Note

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

If Workflow Studio can't generate all the required IAM policies, it displays a banner with the message **Permissions for certain actions cannot be auto-generated. An IAM role will be created with partial permissions only.** For information about how to add the missing permissions, see [Resolving role generation problems](#).

- b. Choose **Create** if you're creating a state machine. Otherwise, choose **Save**.
- c. Choose **Confirm** in the dialog box that appears.

Workflow Studio saves your state machine and creates the new execution role.

Resolving role generation problems

Workflow Studio can't automatically generate an execution role with all the required permissions in the following cases:

- There're errors in your state machine. Make sure to resolve all validation errors in Workflow Studio. Also, make sure that you address any server-side errors you encounter in the course of saving.
- Your state machine contains tasks use AWS SDK integrations. Workflow Studio can't [auto-generate](#) IAM policies in this case. Workflow Studio displays a banner with the message, **Permissions for certain actions cannot be auto-generated. An IAM role will be created with partial permissions only.** In the **Review auto-generated permissions** table, choose the content in **Status** for more information about the policies your execution role is missing. Workflow Studio can still generate an execution role, but this role will not contain IAM policies for all actions. See the links under **Documentation links** to write your own policies and add them to the role after it is generated. These links are available even after you save the state machine.

Role for testing HTTP Tasks in Workflow Studio

You require an execution role to [test](#) an HTTP Task state. If you don't have a role with sufficient permissions, use one of the following options to create a role:

- **Auto-generate a role with Workflow Studio (recommended)** – This is the secure option. Close the **Test state** dialog box and follow the instructions in [Automatically generating roles](#). This will require you to create or update your state machine first, then go back into Workflow Studio to test your state.
- **Use a role with Administrator access** – If you have permissions to create a role with full access to all services and resources in AWS, you can use that role to test any type of state in your workflow. To do this, you can create a Step Functions service role and add the [AdministratorAccess policy](#) to it in the IAM console <https://console.aws.amazon.com/iam/>.

Role for testing an optimized service integration in Workflow Studio

You require an execution role to Task states that call [optimized service integrations](#). If you don't have a role with sufficient permissions, use one of the following options to create a role:

- **Use the documentation links in Workflow Studio to write your own IAM policies (recommended)** – This is the secure option. Close the **Test state** dialog box and follow the instructions in [Automatically generating roles](#). This will require you to create or update your state machine first, then go back into Workflow Studio to test your state.
- **Use a role with Administrator access** – If you have permissions to create a role with full access to all services and resources in AWS, you can use that role to test any type of state in your workflow. To do this, you can create a Step Functions service role and add the [AdministratorAccess policy](#) to it in the IAM console <https://console.aws.amazon.com/iam/>.

Role for testing an AWS SDK service integration in Workflow Studio

You require an execution role to Task states that call [AWS SDK integrations](#). If you don't have a role with sufficient permissions, use one of the following options to create a role:

- **Use the documentation links in Workflow Studio to write your own IAM policies (recommended)** – This is the secure option. Close the **Test state** dialog box and follow the instructions in [Automatically generating roles](#). This will require you to create or update your state machine first, then go back into Workflow Studio to test your state. Do the following:

1. Close the **Test state** dialog box
 2. Choose the **Config** tab to view the Config mode.
 3. Scroll down to the **Permissions** section.
 4. Workflow Studio displays a banner with the message, **Permissions for certain actions cannot be auto-generated. An IAM role will be created with partial permissions only.** Choose **Review auto-generated permissions.**
 5. The **Review auto-generated permissions** table displays a row that shows the action corresponding to the task state you want to test. See the links under **Documentation links** to write your own IAM policies into a custom role.
- **Use a role with Administrator access** – If you have permissions to create a role with full access to all services and resources in AWS, you can use that role to test any type of state in your workflow. To do this, you can create a Step Functions service role and add the [AdministratorAccess policy](https://console.aws.amazon.com/iam/) to it in the IAM console <https://console.aws.amazon.com/iam/>.

Role for testing flow states in Workflow Studio

You require an execution role to test flow states in Workflow Studio. Flow states are those states that direct execution flow, such as [Choice](#), [Parallel](#), [Map](#), [Pass](#), [Wait](#), [Succeed](#), or [Fail](#). The [TestState](#) API doesn't work with Map or Parallel states. Use one of the following options to create a role for testing a flow state:

- **Use any role in your AWS account (recommended)** – Flow states do not require any specific IAM policies, because they don't call AWS actions or resources. Therefore, you can use any IAM role in your AWS account.
 1. In the **Test state** dialog box, select any role from the **Execution role** dropdown list.
 2. If no roles appear in the dropdown list, do the following:
 - a. In the IAM console <https://console.aws.amazon.com/iam/>, choose **Roles**.
 - b. Choose a role from the list, and copy its ARN from the role details page. You will need to provide this ARN in the **Test state** dialog box.
 - c. In the **Test state** dialog box, select **Enter a role ARN** from the **Execution role** dropdown list.
 - d. Paste the ARN in **Role ARN**.
- **Use a role with Administrator access** – If you have permissions to create a role with full access to all services and resources in AWS, you can use that role to test any type of state

in your workflow. To do this, you can create a Step Functions service role and add the [AdministratorAccess policy](#) to it in the IAM console <https://console.aws.amazon.com/iam/>.

Error handling

By default, when a state reports an error, Step Functions causes the workflow execution to fail entirely. For actions and some flow states, you can configure how Step Functions handles errors. Even if you have configured error handling, some errors may still cause a workflow execution to fail. For more information, see [Error handling in Step Functions](#). In Workflow Studio, configure error handling in the the **Error handling** tab of the [Inspector](#) panel.

The screenshot displays the 'Error handling' configuration panel in the AWS Step Functions Inspector. It features four main sections:

- Retry on errors**: Includes a description and a list of 'Retriers'. One 'Retrier #1' is shown with an edit icon. A '+ Add new retriever' button is located below the list.
- Catch errors**: Includes a description and a '+ Add new catcher' button.
- Timeouts**: Contains a dropdown menu for 'TimeoutSeconds - optional' with the text 'Choose an option'.
- HeartbeatSeconds - optional**: Contains a dropdown menu with the text 'Choose an option'.

Retry on errors

You can add one or more rules to action states and the [Parallel](#) flow state to retry the task when an error occurs. These rules are called *retriers*. To add a retriever, choose the edit icon in **Retrier #1** box, then configure its options:

- (Optional) In the **Comment** field, add your comment. It will not affect the workflow, but can be used to annotate your workflow.
- Place the cursor in the **Errors** field and choose an error that will trigger the retriever, or enter a custom error name. You can choose or add multiple errors.
- (Optional) Set an **Interval**. This is the time in seconds before Step Functions make its first retry. Additional retries will follow at intervals that you can configure with **Max attempts** and **Backoff rate**.
- (Optional) Set **Max attempts**. This is the maximum number of retries before Step Functions will cause the execution to fail.
- (Optional) Set the **Backoff rate**. This is a multiplier that determines by how much the retry interval will increase with each attempt.

Note

Not all error handling options are available for all states. Lambda Invoke has one retriever configured by default.

Catch errors

You can add one or more rules to action states and to the [Parallel](#) and [Map](#) flow states to catch an error. These rules are called *catchers*. To add a catcher, choose **Add new catcher**, then configure its options:

- (Optional) In the **Comment** field, add your comment. It will not affect the workflow, but can be used to annotate your workflow.
- Place the cursor in **Errors** field and choose an error that will trigger the catcher, or enter a custom error name. You can choose or add multiple errors.
- In the **Fallback state** field, choose a [fallback state](#). This is the state that the workflow will move to next, after an error is caught.
- (Optional) In the **ResultPath** field, add a `ResultPath` filter to add the error to the original state input. The [ResultPath](#) must be a valid [JsonPath](#). This will be sent to the fallback state.

Timeouts

You can configure a timeout for action states to set the maximum number of seconds your state can run before it fails. Use timeouts to prevent stuck executions. To configure a timeout, enter the number of seconds your state should wait before the execution fails. For more information about timeouts, see `TimeoutSeconds` in [Task state](#) state.

HeartbeatSeconds

You can configure a *Heartbeat* or periodic notification sent by your task. If you set a heartbeat interval, and your state doesn't send heartbeat notifications in the configured intervals, the task is marked as failed. To configure a heartbeat, set a positive, non-zero integer number of seconds. For more information, see `HeartBeatSeconds` in [Task state](#) state.

Tutorial: Learn to use the AWS Step Functions Workflow Studio

In this tutorial, you will learn the basics of working with Workflow Studio for AWS Step Functions. In [Design mode](#) of Workflow Studio, you'll create a state machine containing multiple states, including `Pass`, `Choice`, `Fail`, `Wait`, and `Parallel`. You'll use the drag and drop feature to search for, select, and configure these states. Then, you'll view the auto-generated [Amazon States Language](#) (ASL) definition of your workflow. You'll also use the [Code mode](#) of Workflow Studio to edit the workflow definition. Then, you'll exit Workflow Studio, run the state machine, and review the execution details.

In this tutorial, you'll also learn how to update the state machine and view the changes in the execution output. Finally, you'll perform a clean-up step and delete your state machine.

After you complete this tutorial, you'll know how to use Workflow Studio to create and configure a workflow using both the **Design** and **Code** modes. You'll also know how to update, run, and delete your state machine.

Note

Before you start, make sure to complete the [prerequisites for this tutorial](#).

Topics

- [Step 1: Navigate to Workflow Studio](#)

- [Step 2: Create a state machine](#)
- [Step 3: Review the auto-generated Amazon States Language definition](#)
- [Step 4: Edit the workflow definition in Code mode](#)
- [Step 5: Save the state machine](#)
- [Step 6: Run the state machine](#)
- [Step 7: Update your state machine](#)
- [Step 8: Clean up](#)

Step 1: Navigate to Workflow Studio

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).

Step 2: Create a state machine

In Workflow Studio, a state machine is a graphical representation of your workflow. With Workflow Studio, you can define, configure, and examine the individual steps of your workflow. In the following steps, you use the [Design mode](#) of Workflow Studio to create your state machine.

To create a state machine

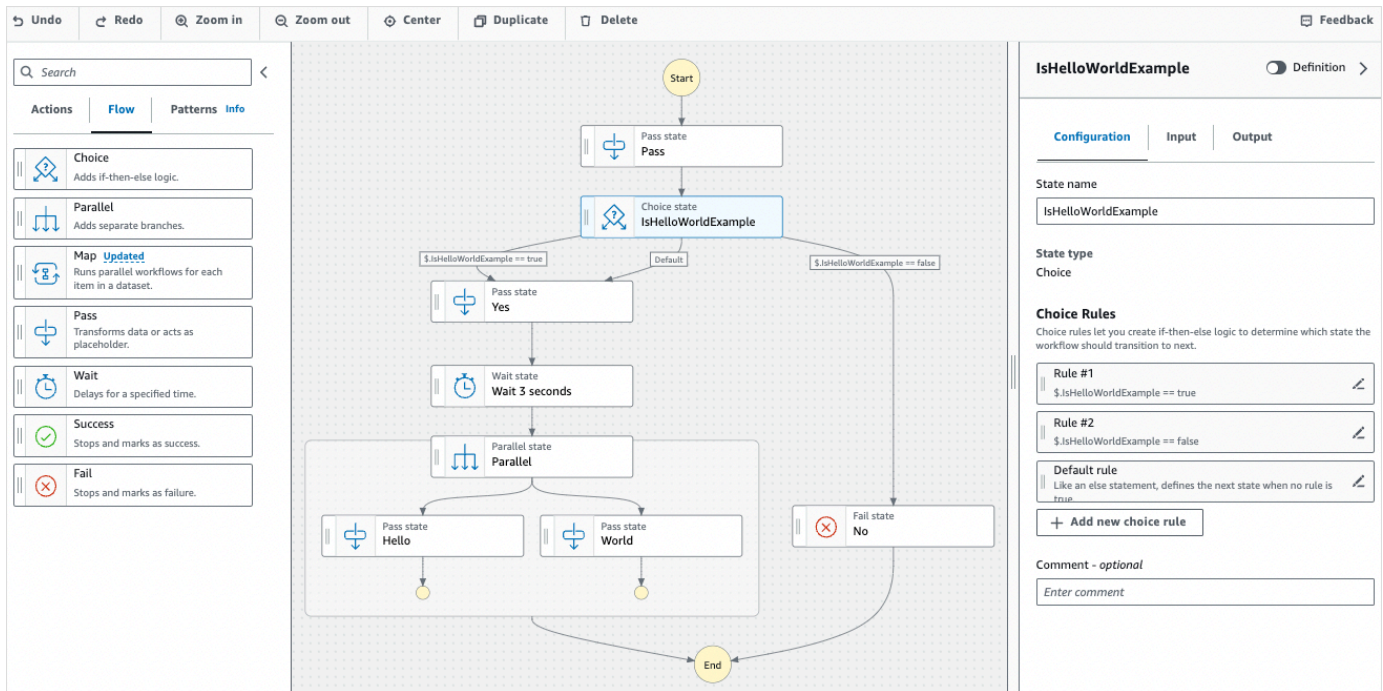
1. Make sure you're in the **Design** mode of Workflow Studio.
2. From the [States browser](#) on the left, choose the **Flow** tab. Then, drag a **Pass** state to the empty state labelled **Drag first state here**.
3. Drag a **Choice** state from the **Flow** tab and drop it below the **Pass** state.
4. For **State name**, replace the default name of **Choice**. For this tutorial, use the name **IsHelloWorldExample**.
5. Drag another **Pass** state and drop it to one branch of the **IsHelloWorldExample** state. Then, drag a **Fail** state and drop it below the other branch of the **IsHelloWorldExample** state.
6. Choose the **Pass (1)** state, and rename it to **Yes**. Rename the **Fail** state as **No**.
7. Specify the **IsHelloWorldExample** state's branching logic using the boolean variable `IsHelloWorldExample`.

If `IsHelloWorldExample` is `False`, the workflow will enter the **No** state. Otherwise, the workflow will continue its execution flow in the **Yes** state.

To define the branching logic, do the following:

- a. Choose the **IsHelloWorldExample** state on the [Canvas](#), and then under **Choice Rules**, choose the edit icon in the **Rule #1** box to define the first choice rule.
 - b. Choose **Add conditions**.
 - c. In the **Conditions for rule #1** dialog box, enter `$.IsHelloWorldExample` under **Variable**.
 - d. Choose **is equal to** under **Operator**.
 - e. Choose **Boolean constant** under **Value**, and then choose **true** from the dropdown list.
 - f. Choose **Save conditions**.
 - g. Make sure the **Then next state is:** dropdown list has **Yes** selected.
 - h. Choose **Add new choice rule**, then choose **Add conditions**.
 - i. In the **Rule #2** box, define the second choice rule when the `IsHelloWorldExample` variable's value is false by repeating substeps 7.c through 7.f. For step 7.e, choose **false** instead of **true**.
 - j. In the **Rule #2** box, choose **No** from the **Then next state is:** dropdown list.
 - k. In the **Default rule** box, choose the edit icon to define the default choice rule, and then choose **Yes** from the dropdown list.
8. Add a **Wait** state after the **Yes** state, and name it **Wait 3 sec**. Then, configure the wait time to be three seconds by doing the following steps:
- a. Under **Options**, keep the default selection of **Wait for a fixed interval**.
 - b. Under **Seconds**, make sure **Enter seconds** is selected, and then enter **3** in the box.
9. After the **Wait 3 sec** state, add a **Parallel** state. Add two **Pass** states in its two branches. Name the first **Pass** state **Hello**. Name the second **Pass** state **World**.

The completed workflow will look like this:



Step 3: Review the auto-generated Amazon States Language definition

As you drag and drop states from the **Flow** tab onto the canvas, Workflow Studio automatically composes the [Amazon States Language](#) (ASL) definition of your workflow in real-time. In the [Inspector](#) panel, choose the **Definition** toggle button to view this definition or switch to the [Code mode](#) to edit this definition as required. For information about editing the workflow definition, see [Step 4](#) of this tutorial.

- (Optional) Choose **Definition** on the **Inspector** panel and view the state machine's workflow.

The following example code shows the auto-generated Amazon States Language definition for the `IsHelloWorldExample` state machine. The Choice state that you added in Workflow Studio is used to determine the execution flow based on [the branching logic you defined in Step 2](#).

```
{
  "Comment": "A Hello World example of the Amazon States Language using Pass states",
  "StartAt": "Pass",
  "States": {
    "Pass": {
      "Type": "Pass",
```

```
    "Next": "IsHelloWorldExample",
    "Comment": "A Pass state passes its input to its output, without performing
work. Pass states are useful when constructing and debugging state machines."
  },
  "IsHelloWorldExample": {
    "Type": "Choice",
    "Comment": "A Choice state adds branching logic to a state machine. Choice
rules can implement 16 different comparison operators, and can be combined using
And, Or, and Not\\\"",
    "Choices": [
      {
        "Variable": "$.IsHelloWorldExample",
        "BooleanEquals": false,
        "Next": "No"
      },
      {
        "Variable": "$.IsHelloWorldExample",
        "BooleanEquals": true,
        "Next": "Yes"
      }
    ],
    "Default": "Yes"
  },
  "No": {
    "Type": "Fail",
    "Cause": "Not Hello World"
  },
  "Yes": {
    "Type": "Pass",
    "Next": "Wait 3 sec"
  },
  "Wait 3 sec": {
    "Type": "Wait",
    "Seconds": 3,
    "Next": "Parallel"
  },
  "Parallel": {
    "Type": "Parallel",
    "End": true,
    "Branches": [
      {
        "StartAt": "Hello",
        "States": {
          "Hello": {
```

```
        "Type": "Pass",
        "End": true
      }
    },
    {
      "StartAt": "World",
      "States": {
        "World": {
          "Type": "Pass",
          "End": true
        }
      }
    }
  ]
}
```

Step 4: Edit the workflow definition in Code mode

The **Code** mode of Workflow Studio provides an integrated code editor to view and edit the ASL definition of your workflows.

1. Choose **Code** to switch to the **Code** mode.
2. After the **Parallel** state's definition, place the cursor and press **Enter**.
3. Press **Ctrl+space** to see the list of states that you can add after the **Parallel** state.
4. Choose **Pass State** from the list of options. The code editor adds boilerplate code for the **Pass State**.
5. The addition of this state results in errors in your workflow definition. In the **Parallel** state's definition, replace `"End": true` with `"Next": "PassState"`.
6. In the **Pass State** definition you added, make the following changes:
 - a. Remove the **Result** node.
 - b. Remove `"ResultPath": "$.result"`, and `"Next": "NextState"`.
 - c. After `"Type": "Pass"`, enter `"End": true`.
 - d. Add a `,` after the **Pass State** definition.

Your workflow definition should now look similar to the following definition.

```
{
  "Comment": "A description of my state machine",
  "StartAt": "Pass",
  "States": {
    "Pass": {
      "Type": "Pass",
      "Next": "IsHelloWorldExample"
    },
    "IsHelloWorldExample": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.IsHelloWorldExample",
          "BooleanEquals": true,
          "Next": "Yes"
        },
        {
          "Variable": "$.IsHelloWorldExample",
          "BooleanEquals": false,
          "Next": "No"
        }
      ],
      "Default": "Yes"
    },
    "Yes": {
      "Type": "Pass",
      "Next": "Wait 3 seconds"
    },
    "Wait 3 seconds": {
      "Type": "Wait",
      "Seconds": 3,
      "Next": "Parallel"
    },
    "Parallel": {
      "Type": "Parallel",
      "Branches": [
        {
          "StartAt": "Hello",
          "States": {
            "Hello": {
              "Type": "Pass",
              "End": true
            }
          }
        }
      ]
    }
  }
}
```

```
    }
  }
},
{
  "StartAt": "World",
  "States": {
    "World": {
      "Type": "Pass",
      "End": true
    }
  }
},
],
"Next": "PassState"
},
"PassState": {
  "Type": "Pass",
  "End": true
},
"No": {
  "Type": "Fail"
}
}
}
```

Step 5: Save the state machine

1. Choose the **Config** more or choose the edit icon next to the default state machine name of **MyStateMachine**. In **State machine configuration**, specify a name. For example, enter **HelloWorld**.
2. (Optional) Specify other workflow settings, such as state machine type and its execution role. For this tutorial, keep all the default selections in **State machine configuration**.
3. Choose **Create**.
4. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role configuration** to go back to the **Config** mode.

For more information about the **Config** mode, see [Config mode of Workflow Studio](#).

Step 6: Run the state machine

State machine executions are instances where you run your workflow to perform tasks.

1. On the **State machines** page, choose the **HelloWorld** state machine.
2. On the **HelloWorld** page, choose **Start execution**.
3. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

4. In the **Input** box, enter input values for your execution in JSON format. Based on your input, the `IsHelloWorldExample` variable determines which state machine flow will be executed. For now, use the following input value:

```
{
  "IsHelloWorldExample": true
}
```

Note

While specifying an execution input is optional, in this tutorial, it is mandatory to specify an execution input similar to the above example input. This input value is referenced in the `Choice` state when you run the state machine.

5. Choose **Start execution**.
6. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output,

and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

For this tutorial, if you entered an input value of `"IsHelloWorldExample": true`, you should see the following output:

```
{
  "IsHelloWorldExample": true
},
{
  "IsHelloWorldExample": true
}
```

Step 7: Update your state machine

When you update a state machine, your updates are *eventually consistent*. After a short amount of time, all newly started executions will reflect your state machine's updated definition. All currently running executions will run to completion under the previous definition.

In this step, you'll update your state machine in the [Design mode](#) mode of Workflow Studio. You'll add a `Result` field in the **Pass** state named **World**.

1. On the page titled with your execution ID, choose **Edit state machine**.
2. Make sure you're in the **Design** mode.
3. Choose the **Pass** state named **World** on the canvas, and then choose **Output**.
4. In the **Result** box, enter **"World has been updated!"**.
5. Choose **Save**.
6. (Optional) In the **Definition** area, view the updated Amazon States Language definition of your workflow.

```
{
  "Type": "Parallel",
  "End": true,
  "Branches": [
    {
      "StartAt": "Hello",
      "States": {
        "Hello": {
```

```
        "Type": "Pass",
        "End": true
      }
    },
    {
      "StartAt": "World",
      "States": {
        "World": {
          "Type": "Pass",
          "Result": "World has been updated!",
          "End": true
        }
      }
    }
  ],
  "Next": "PassState"
}
```

7. Choose **Execute**.
8. In the **Start execution** dialog box that opens in a new tab, provide the following execution input.

```
{
  "IsHelloWorldExample": true
}
```

9. Choose **Start Execution**.
10. (Optional) In the **Graph view**, choose the **World** step, and then choose **Output**. The output is **World has been updated!**

Step 8: Clean up

To delete your state machine

1. From the navigation menu, choose **State machines**.
2. On the **State machines** page, select **HelloWorld**, and then choose **Delete**.
3. In the **Delete state machine** dialog box, type **delete** to confirm deletion.
4. Choose **Delete**.

If deletion is successful, a green status bar appears at the top of your screen. The green status bar tells you that your state machine is marked for deletion. Your state machine will be deleted when all of its in-progress executions stop running.

To delete your execution role

1. Open the [Roles page](#) for IAM.
2. Choose the IAM role that Step Functions created for you. For example, **StepFunctions-HelloWorld-role-EXAMPLE**.
3. Choose **Delete role**.
4. Choose **Yes, delete**.

Tutorials for Step Functions

The tutorials in this section can help you understand different aspects of working with AWS Step Functions.

To complete these tutorials, you need an AWS account. If you don't have an AWS account, navigate to <https://aws.amazon.com/> and choose **Create an AWS Account**.

Topics

- [Creating a Step Functions state machine that uses Lambda](#)
- [Handling error conditions using a Step Functions state machine](#)
- [Using Inline Map state to repeat an action](#)
- [Copying large-scale CSV data using Distributed Map](#)
- [Processing entire batch of data with a Lambda function](#)
- [Processing individual data items with a Lambda function](#)
- [Starting a State Machine Execution in Response to Amazon S3 Events](#)
- [Creating a Step Functions API using API Gateway](#)
- [Create a Step Functions state machine using AWS SAM](#)
- [Creating an Activity state machine using Step Functions](#)
- [Iterate a loop with Lambda](#)
- [Continuing Long-running Workflow Executions as a New Execution](#)
- [Deploying an Example Human Approval Project](#)
- [View X-Ray traces in Step Functions](#)
- [Gather Amazon S3 bucket info using AWS SDK service integrations](#)

Creating a Step Functions state machine that uses Lambda

In this tutorial, you will create a single-step workflow using AWS Step Functions to invoke an AWS Lambda function.

Note

Step Functions is based on *state machines* and *tasks*. In Step Functions, state machines are called *workflows*, which are a series of event-driven steps. Each step in a workflow is

called a *state*. For example, a [Task state](#) represents a unit of work that another AWS service performs, such as calling another AWS service or API.

For more information, see:

- [What is AWS Step Functions?](#)
- [Call other AWS services](#)

Lambda is well-suited for Task states, because Lambda functions are *serverless* and easy to write. You can write code in the AWS Management Console or your favorite editor. AWS handles the details of providing a computing environment for your function and running it.

In this topic:

- [Step 1: Create a Lambda function](#)
- [Step 2: Test the Lambda function](#)
- [Step 3: Create a state machine](#)
- [Step 4: Run the state machine](#)

Step 1: Create a Lambda function

Your Lambda function receives event data and returns a greeting message.

Important

Ensure that your Lambda function is under the same AWS account and AWS Region as your state machine.

1. Open the [Lambda console](#) and choose **Create function**.
2. On the **Create function** page, choose **Author from scratch**.
3. For **Function name**, enter `HelloFunction`.
4. Keep the default selections for all other options, and then choose **Create function**.
5. After your Lambda function is created, copy the function's Amazon Resource Name (ARN) displayed in the upper-right corner of the page. To copy the ARN, click



The following is an example ARN:

```
arn:aws:lambda:us-east-1:123456789012:function:HelloFunction
```

6. Copy the following code for the Lambda function into the **Code source** section of the *HelloFunction* page.

```
export const handler = async(event, context, callback) => {  
    callback(null, "Hello from " + event.who + "!");  
};
```

This code assembles a greeting using the `who` field of the input data, which is provided by the event object passed into your function. You add input data for this function later, when you [start a new execution](#). The `callback` method returns the assembled greeting from your function.

7. Choose **Deploy**.

Step 2: Test the Lambda function

Test your Lambda function to see it in operation.

1. Choose **Test**.
2. For **Event name**, enter `HelloEvent`.
3. Replace the **Event JSON** data with the following.

```
{  
  "who": "AWS Step Functions"  
}
```

The `"who"` entry corresponds to the `event.who` field in your Lambda function, completing the greeting. You will input the same input data when you run your state machine.

4. Choose **Save** and then choose **Test**.
5. To review the test results, under **Execution result**, expand **Details**.

Step 3: Create a state machine

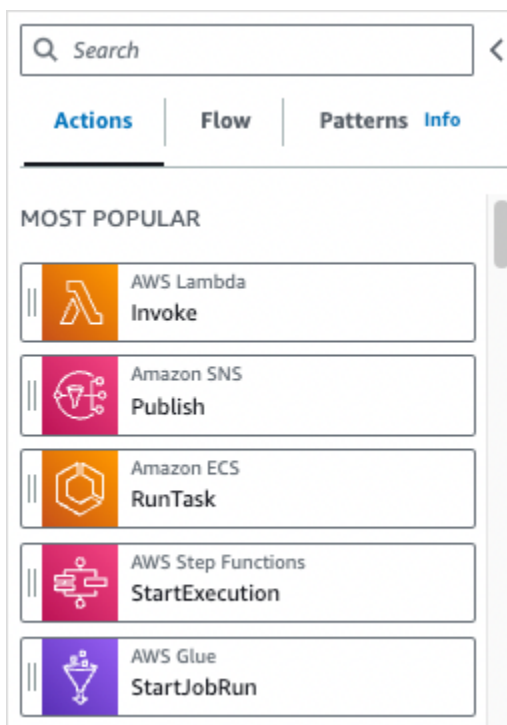
Use the Step Functions console to create a state machine that invokes the Lambda function that you created in [Step 1](#).

1. Open the [Step Functions console](#) and choose **Create state machine**.

⚠ Important

Make sure that your state machine is under the same AWS account and Region as the Lambda function you created earlier.

2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).
4. In the [States browser](#) on the left, make sure you've chosen the **Actions** tab. Then, do the following:
 - Drag and drop the **AWS Lambda Invoke** API into the empty state labelled **Drag first state here**.



5. In the [Inspector](#) panel on the right, configure the Lambda function:
 - a. In the **API Parameters** section, choose [the Lambda function that you created earlier](#) in the **Function name** dropdown list.
 - b. Keep the default selection in the **Payload** dropdown list.

6. (Optional) Choose **Definition** to view the state machine's [Amazon States Language](#) (ASL) definition, which is automatically generated based on your selections in the **Actions** tab and **Inspector** panel.
7. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

For example, enter the name **LambdaStateMachine**.

Note

Names of state machines, executions, and activity tasks must not exceed 80 characters in length. These names must be unique for your account and AWS Region, and must not contain any of the following:

- Whitespace
- Wildcard characters (? *)
- Bracket characters (< > { } [])
- Special characters (" # % \ ^ | ~ ` \$ & , ; : /)
- Control characters (\\u0000 - \\u001f or \\u007f - \\u009f).

If your state machine is of type Express, you can provide the same name to multiple executions of the state machine. Step Functions generates a unique execution ARN for each Express state machine execution, even if multiple executions have the same name.


Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

8. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

For this tutorial, keep all the default selections in **State machine settings**.

9. Choose **Create**.
10. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

 **Note**

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.


Step 4: Run the state machine

After you create your state machine, you can run it.

1. On the **State machines** page, choose **LambdaStateMachine**.
2. Choose **Start execution**.

The **Start execution** dialog box is displayed.

3. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

 **Note**

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

4. In the **Input** area, replace the example execution data with the following.

```
{
  "who" : "AWS Step Functions"
}
```

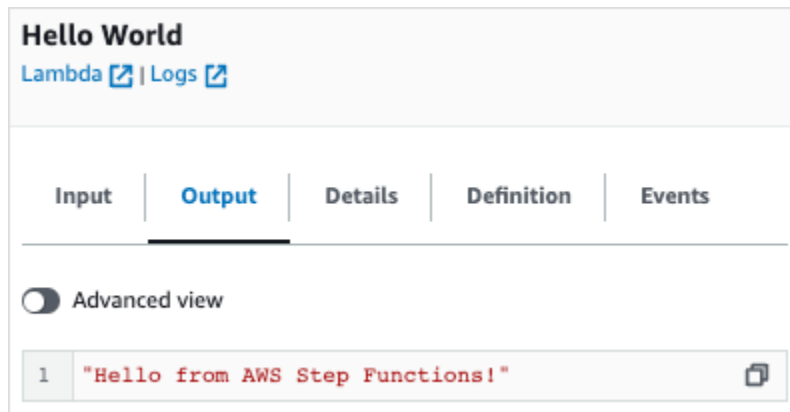
"who" is the key name that your Lambda function uses to get the name of the person to greet.

5. Choose **Start Execution**.

Your state machine's execution starts, and a new page showing your running execution is displayed.

- The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).



Note

You can also pass payloads while invoking Lambda from a state machine. For more information and examples about invoking Lambda by passing payload in the `Parameters` field, see [Invoke Lambda with Step Functions](#).

Handling error conditions using a Step Functions state machine

In this tutorial, you create an AWS Step Functions state machine with a [Fallback states](#) field. The `Catch` field uses an AWS Lambda function to respond with conditional logic based on error message type. This is a technique called *function error handling*.

For more information, see [AWS Lambda function errors in Node.js](#) in the *AWS Lambda Developer Guide*.

Note

You can also create state machines that [Retry](#) on timeouts or those that use `Catch` to transition to a specific state when an error or timeout occurs. For examples of these error handling techniques, see [Examples Using Retry and Using Catch](#).

In this topic:

- [Step 1: Create a Lambda function that fails](#)
- [Step 2: Test the Lambda function](#)
- [Step 3: Create a state machine with a `Catch` field](#)
- [Step 4: Run the state machine](#)

Step 1: Create a Lambda function that fails

Use a Lambda function to simulate an error condition.

Important

Ensure that your Lambda function is under the same AWS account and AWS Region as your state machine.

1. Open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.
2. Choose **Create function**.
3. Choose **Use a blueprint**, enter `step-functions` into the search box, and then choose the **Throw a custom error** blueprint.
4. For **Function name**, enter `FailFunction`.
5. For **Role**, keep the default selection (**Create a new role with basic Lambda permissions**).
6. The following code is displayed in the **Lambda function code** pane.

```
exports.handler = async (event, context) => {
  function CustomError(message) {
    this.name = 'CustomError';
    this.message = message;
  }
```

```
CustomError.prototype = new Error();

throw new CustomError('This is a custom error!');
};
```

The context object returns the error message `This is a custom error!`.

7. Choose **Create function**.
8. After your Lambda function is created, copy the function's Amazon Resource Name (ARN) displayed in the upper-right corner of the page. To copy the ARN, click



The following is an example ARN:

```
arn:aws:lambda:us-east-1:123456789012:function:FailFunction
```

9. Choose **Deploy**.

Step 2: Test the Lambda function

Test your Lambda function to see it in operation.

1. On the **FailFunction** page, choose the **Test** tab, and then choose **Test**. You don't need to create a test event.
2. To review the test results (the simulated error), under **Execution result**, expand **Details**.

Step 3: Create a state machine with a Catch field

Use the Step Functions console to create a state machine that uses a [Task state](#) state with a Catch field. Add a reference to your Lambda function in the Task state. The state machine invokes the Lambda function, which fails during execution. Step Functions retries the function twice using exponential backoff between retries.


1. Open the [Step Functions console](#) and choose **Create state machine**.
2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).
4. Choose **Code** to open the code editor. In the code editor, you write and edit the [Amazon States Language](#) (ASL) definition of your workflows.

5. Paste the following code, but replace the ARN of [the Lambda function that you created earlier](#) in the Resource field.

```
{
  "Comment": "A Catch example of the Amazon States Language using an AWS Lambda
function",
  "StartAt": "CreateAccount",
  "States": {
    "CreateAccount": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FailFunction",
      "Catch": [ {
        "ErrorEquals": ["CustomError"],
        "Next": "CustomErrorFallback"
      }, {
        "ErrorEquals": ["States.TaskFailed"],
        "Next": "ReservedTypeFallback"
      }, {
        "ErrorEquals": ["States.ALL"],
        "Next": "CatchAllFallback"
      } ],
      "End": true
    },
    "CustomErrorFallback": {
      "Type": "Pass",
      "Result": "This is a fallback from a custom Lambda function exception",
      "End": true
    },
    "ReservedTypeFallback": {
      "Type": "Pass",
      "Result": "This is a fallback from a reserved error code",
      "End": true
    },
    "CatchAllFallback": {
      "Type": "Pass",
      "Result": "This is a fallback from any error code",
      "End": true
    }
  }
}
```

This is a description of your state machine using the Amazon States Language. It defines a single Task state named `CreateAccount`. For more information, see [State Machine Structure](#).

For more information about the syntax of the `Retry` field, see [State machine examples using Retry and using Catch](#).

 **Note**

Unhandled errors in Lambda are reported as `Lambda.Unknown` in the error output. These include out-of-memory errors and function timeouts. You can match on `Lambda.Unknown`, `States.ALL`, or `States.TaskFailed` to handle these errors. When Lambda hits the maximum number of invocations, the error is `Lambda.TooManyRequestsException`. For more information about Lambda function errors, see [Error handling and automatic retries](#) in the *AWS Lambda Developer Guide*.

6. (Optional) In the [Graph visualization pane](#), see the real-time graphical visualization of your workflow.
7. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.


For this tutorial, enter **CatchFailure**.

8. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

For this tutorial, keep all the default selections in **State machine settings**.

9. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

 **Note**

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

Step 4: Run the state machine

After you create your state machine, you can run it.

1. On the **State machines** page, choose **Catchfailure**.
2. On the **Catchfailure** page, choose **Start execution**. The **Start execution** dialog box is displayed.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.
3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

For example, to view your custom error message, choose the **CreateAccount** step in **Graph view**, and then choose the **Output** tab.

The screenshot displays the AWS Step Functions console interface. At the top, there are tabs for 'Details', 'Execution input and output', and 'Definition'. The 'Execution input and output' tab is active, showing the input JSON on the left and the output error message on the right. Below this, there are tabs for 'Graph view' and 'Table view'. The 'Graph view' shows a workflow diagram starting with a 'Start' state, followed by a 'CreateAccount' state (highlighted in orange with a warning icon). From 'CreateAccount', the workflow branches into three parallel paths leading to 'CustomErrorFallback', 'ReservedTypeFallback', and 'CatchAllFallback' states, all of which converge to an 'End' state. To the right of the graph, the 'CreateAccount' state is expanded, showing its 'Input', 'Output', 'Details', 'Definition', and 'Events' tabs. The 'Output' tab is active, displaying the error details in an 'Advanced view'.

```

1 {
2   "Error": "CustomError",
3   "Cause": "{\n  \"errorType\": \"CustomError\",\n  \"errorMessage\": \"This is a custom error!\",\n  \"trace\": [\n    {\n      \"Error\": \"\n      at Runtime.handler\n      (file:///var/task/index.mjs:7:27)\",\n    },\n    {\n      \"Error\": \"\n      at Runtime.handleOnceNonStreaming\n      (file:///var/runtime/index.mjs:1083:29)\",\n    }\n  ]\n}"
4 }

```

Note

You can preserve the state input with the error by using `ResultPath`. See [Use ResultPath to Include Both Error and Input in a Catch](#).

Using Inline Map state to repeat an action

This tutorial helps you get started with using the Map state in Inline mode. You use the *Inline Map state* in your workflows to repeatedly perform an action. For more information about Inline mode, see [Map state in Inline mode](#).

In this tutorial, you use the *Inline Map state* to repeatedly generate version 4 universally unique identifiers (v4 UUID). You start by creating a workflow that contains two [Pass](#) states and an *Inline Map state* in the Workflow Studio. Then, you configure the input and output, including the input JSON array for the Map state. The Map state returns an output array that contains the v4 UUIDs generated for each item in the input array.

Contents

- [Step 1: Create the workflow prototype](#)
- [Step 2: Configure input and output](#)
- [Step 3: Review the auto-generated Amazon States Language definition and save the workflow](#)

- [Step 4: Run the state machine](#)

Step 1: Create the workflow prototype

In this step, you create the prototype for your workflow using Workflow Studio. Workflow Studio is a visual workflow designer available in the Step Functions console. You'll choose the required states from the **Flow** tab and use the drag and drop feature of Workflow Studio to create the workflow prototype.

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).
4. From the **Flow** tab, drag a **Pass** state and drop it to the empty state labelled **Drag first state here**.
5. Drag a **Map** state and drop it below the **Pass** state. Rename the **Map** state to **Map demo**.
6. Drag a second **Pass** state and drop it inside of the **Map demo** state.
7. Rename the second **Pass** state to **Generate UUID**.

Step 2: Configure input and output

In this step, you configure input and output for all the states in your workflow prototype. First, you inject some fixed data into the workflow using the first **Pass** state. This **Pass** state passes on this data as input to the **Map demo** state. Within this input, you specify the node that contains the input array the **Map demo** state should iterate over. Then you define the step that the **Map demo** state should repeat to generate the v4 UUIDs. Finally, you configure the output to return for each repetition.

1. Choose the first **Pass** state in your workflow prototype. In the **Output** tab, enter the following under **Result**:

```
{
  "foo": "bar",
  "colors": [
    "red",
    "green",
    "blue",
```

```
    "yellow",  
    "white"  
  ]  
}
```

2. Choose the **Map demo** state and in the **Configuration** tab, do the following:
 - a. Choose **Provide a path to items array**.
 - b. Specify the following [reference path](#) to select the node that contains the input array:

```
$.colors
```

3. Choose the **Generate UUID** state and in the **Input** tab, do the following:
 - a. Choose **Transform input with Parameters**.
 - b. Enter the following JSON input to generate the v4 UUIDs for each of the input array items. You use the [States.UUID](#) intrinsic function to generate the UUIDs.

```
{  
  "uuid.$": "States.UUID()  
}
```

4. For the **Generate UUID** state, choose the **Output** tab and do the following:
 - a. Choose **Filter output with OutputPath**.
 - b. Enter the following reference path to select the JSON node that contains the output array items:

```
$.uuid
```

Step 3: Review the auto-generated Amazon States Language definition and save the workflow

As you drag and drop states from the **Flow** panel onto the canvas, Workflow Studio automatically composes the [Amazon States Language](#) (ASL) definition of your workflow in real-time. You can edit this definition as required.

1. (Optional) Choose **Definition** on the [Inspector](#) panel to view the automatically-generated Amazon States Language definition of your workflow.

Tip

You can also view the ASL definition in the [Code editor](#) of Workflow Studio. In the code editor, you can also edit the ASL definition of your workflow.

The following example shows the automatically generated Amazon States Language definition for your workflow.

```
{
  "Comment": "Using Map state in Inline mode",
  "StartAt": "Pass",
  "States": {
    "Pass": {
      "Type": "Pass",
      "Next": "Map demo",
      "Result": {
        "foo": "bar",
        "colors": [
          "red",
          "green",
          "blue",
          "yellow",
          "white"
        ]
      }
    },
    "Map demo": {
      "Type": "Map",
      "ItemsPath": "$.colors",
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "INLINE"
        }
      },
      "StartAt": "Generate UUID",
      "States": {
        "Generate UUID": {
          "Type": "Pass",
          "End": true,
          "Parameters": {
            "uuid.$": "States.UUID()"
          }
        }
      }
    }
  }
}
```

```
        },
        "OutputPath": "$.uuid"
      }
    },
    "End": true
  }
}
```

2. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

For this tutorial, enter the name **InlineMapDemo**.

3. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

For this tutorial, keep all the default selections in **State machine configuration**.

4. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

Note

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

Step 4: Run the state machine

State machine executions are instances where you run your workflow to perform tasks.

1. On the **InlineMapDemo** page, choose **Start execution**.
2. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions automatically generates a unique execution name.

Note

Step Functions allows you to create names for state machines, executions, activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.
3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

To view the execution input and output side-by-side, choose **Execution input and output**. Under **Output**, view the output array returned by the Map state. The following is an example of the output array:

```
[  
  "a85cbc7b-4e65-4ac2-97af-80ed504adc1d",  
  "b05bca11-d481-414e-aa9a-88285ec6590d",  
  "f42d59f7-bd32-480f-b270-caddb518ce2a",  
  "15f18616-517d-4b69-b7c3-bf22222d2efd",  
  "690bcfee-6d58-408c-a6b4-1995ccafdbd2"  
]
```

Copying large-scale CSV data using Distributed Map

This tutorial helps you start using the Map state in Distributed mode. A Map state set to **Distributed** is known as a *Distributed Map state*. You use the *Distributed Map state* in your workflows to iterate over large-scale Amazon S3 data sources. The Map state runs each iteration as a child workflow

execution, which enables high concurrency. For more information about Distributed mode, see [Map state in Distributed mode](#).

In this tutorial, you use the *Distributed Map state* to iterate over a CSV file in an Amazon S3 bucket. You then return its contents, along with the ARN of a child workflow execution, in another Amazon S3 bucket. You start by creating a workflow prototype in the Workflow Studio. Next, you set the [Map state's processing mode](#) to Distributed, specify the CSV file as the dataset, and provide its location to the Map state. You also specify the workflow type for the child workflow executions that the *Distributed Map state* starts as **Express**.

In addition to these settings, you also specify other configurations, such as the maximum number of concurrent child workflow executions and the location to export the Map result, for the example workflow used in this tutorial.

Contents

- [Prerequisites](#)
- [Step 1: Create the workflow prototype](#)
- [Step 2: Configure the required fields for Map state](#)
- [Step 3: Configure additional options](#)
- [Step 4: Configure the Lambda function](#)
- [Step 5: Update the workflow prototype](#)
- [Step 6: Review the auto-generated Amazon States Language definition and save the workflow](#)
- [Step 7: Run the state machine](#)

Prerequisites

- Upload a CSV file to an Amazon S3 bucket. You must define a header row within your CSV file. For information about size limits imposed on the CSV file and how to specify the header row, see [CSV file in an Amazon S3 bucket](#).
- Create another Amazon S3 bucket and a folder within that bucket to export the Map state result to.

⚠ Important

Make sure that your Amazon S3 buckets are under the same AWS account and AWS Region as your state machine.

Step 1: Create the workflow prototype

In this step, you create the prototype for your workflow using Workflow Studio. Workflow Studio is a visual workflow designer available in the Step Functions console. You choose the required state and API action from the **Flow** and **Actions** tabs respectively. You'll use the drag and drop feature of Workflow Studio to create the workflow prototype.

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).
4. From the **Flow** tab, drag a **Map** state and drop it to the empty state labelled **Drag first state here**.
5. In the **Configuration** tab, for **State name**, enter **Process data**.
6. From the **Actions** tab, drag an **AWS Lambda Invoke** API action and drop it inside the **Process data** state.
7. Rename the **AWS Lambda Invoke** state to **Process CSV data**.

Step 2: Configure the required fields for Map state

In this step, you configure the following required fields of the *Distributed Map state*:

- [ItemReader](#) – Specifies the dataset and its location from which the Map state can read input.
- [ItemProcessor](#) – Specifies the following values:
 - **ProcessorConfig** – Set the **Mode** and **ExecutionType** to **DISTRIBUTED** and **EXPRESS** respectively. This sets the Map state's processing mode and the workflow type for child workflow executions that the *Distributed Map state* starts.
 - **StartAt** – The first state in the Map workflow.
 - **States** – Defines the Map workflow, which is a set of steps to repeat in each child workflow execution.

- [ResultWriter](#) – Specifies the Amazon S3 location where Step Functions writes the *Distributed Map state* results.

Important

Make sure that the Amazon S3 bucket you use to export the results of a Map Run is under the same AWS account and AWS Region as your state machine. Otherwise, your state machine execution will fail with the `States.ResultWriterFailed` error.

To configure the required fields:

1. Choose the **Process data** state and, in the **Configuration** tab, do the following:
 - a. For **Processing mode**, choose **Distributed**.
 - b. For **Item source**, choose **Amazon S3**, and then choose **CSV file in S3** from the **S3 item source** dropdown list.
 - c. Do the following to specify the Amazon S3 location of your CSV file:
 - i. For **S3 object**, select **Enter bucket and key** from the dropdown list.
 - ii. For **Bucket**, enter the name of the Amazon S3 bucket, which contains the CSV file. For example, **sourceBucket**.
 - iii. For **Key**, enter the name of the Amazon S3 object in which you saved the CSV file. You must also specify the name of the CSV file in this field. For example, **csvDataset/ratings.csv**.
 - d. For CSV files, you must also specify the location of the column header. To do this, choose **Additional configuration**, and then for **CSV header location** keep the default selection of **First row** if the first row of your CSV file is the header. Otherwise, choose **Given** to specify the header within the state machine definition. For more information, see [ReaderConfig](#).
 - e. For **Child execution type**, choose **Express**.
2. In **Export location**, to export the Map Run results to a specific Amazon S3 location, choose **Export Map state's output to Amazon S3**.
3. Do the following:
 - a. For **S3 bucket**, choose **Enter bucket name and prefix** from the dropdown list.

- b. For **Bucket**, enter the name of the Amazon S3 bucket where you want to export the results to. For example, **mapOutputs**.
- c. For **Prefix**, enter the folder name where you want to save the results to. For example, **resultData**.

Step 3: Configure additional options

In addition to the required settings for a *Distributed Map state*, you can also specify other options. These can include the maximum number of concurrent child workflow executions and the location to export the Map state result to.

1. Choose the **Process data** state. Then, in **Item source**, choose **Additional configuration**.
2. Do the following:
 - a. Choose **Modify items with ItemSelector** to specify a custom JSON input for each child workflow execution.
 - b. Enter the following JSON input:

```
{
  "index.$": "$$.Map.Item.Index",
  "value.$": "$$.Map.Item.Value"
}
```

For information about how to create a custom input, see [ItemSelector](#).

3. In **Runtime settings**, for **Concurrency limit**, specify the number of concurrent child workflow executions that the *Distributed Map state* can start. For example, enter **100**.
4. Open a new window or tab on your browser and complete the configuration of the Lambda function you'll use in this workflow, as explained in [Step 4: Configure the Lambda function](#).

Step 4: Configure the Lambda function

Important

Ensure that your Lambda function is under the same AWS Region as your state machine.

1. Open the [Lambda console](#) and choose **Create function**.
2. On the **Create function** page, choose **Author from scratch**.
3. In the **Basic information** section, configure your Lambda function:
 - a. For **Function name**, enter **distributedMapLambda**.
 - b. For **Runtime**, choose **Node.js 16.x**.
 - c. Keep all of the default selections and choose **Create function**.
 - d. After you create your Lambda function, copy the function's Amazon Resource Name (ARN) displayed in the upper-right corner of the page. You'll need to provide this in your workflow prototype. To copy the ARN, click



The following is an example ARN:

```
arn:aws:lambda:us-east-2:123456789012:function:distributedMapLambda
```

4. Copy the following code for the Lambda function and paste it into the **Code source** section of the **distributedMapLambda** page.

```
exports.handler = async function(event, context) {
  console.log("Received Input:\n", event);

  return {
    'statusCode' : 200,
    'inputReceived' : event //returns the input that it received
  }
};
```

5. Choose **Deploy**. Once your function deploys, choose **Test** to see the output of your Lambda function.

Step 5: Update the workflow prototype

In the Step Functions console, you'll update your workflow to add the Lambda function's ARN.

1. Return to the tab or window where you created the workflow prototype.
2. Choose the **Process CSV data** step, and in the **Configuration** tab, do the following:
 - a. For **Integration type**, choose **Optimized**.

- b. For **Function name**, start to enter the name of your Lambda function. Choose the function from the dropdown list that appears, or choose **Enter function name** and provide the Lambda function ARN.

Step 6: Review the auto-generated Amazon States Language definition and save the workflow

As you drag and drop states from the **Action** and **Flow** tabs onto the canvas, Workflow Studio automatically composes the [Amazon States Language](#) definition of your workflow in real-time. You can edit this definition as required.

1. (Optional) Choose **Definition** on the [Inspector](#) panel and view the state machine definition.

Tip

You can also view the ASL definition in the [Code editor](#) of Workflow Studio. In the code editor, you can also edit the ASL definition of your workflow.

The following example code shows the automatically generated Amazon States Language definition for your workflow.

```
{
  "Comment": "Using Map state in Distributed mode",
  "StartAt": "Process data",
  "States": {
    "Process data": {
      "Type": "Map",
      "MaxConcurrency": 100,
      "ItemReader": {
        "ReaderConfig": {
          "InputType": "CSV",
          "CSVHeaderLocation": "FIRST_ROW"
        },
        "Resource": "arn:aws:states:::s3:getObject",
        "Parameters": {
          "Bucket": "sourceBucket",
          "Key": "csvDataset/ratings.csv"
        }
      }
    }
  }
}
```

```
    },
    "ItemProcessor": {
      "ProcessorConfig": {
        "Mode": "DISTRIBUTED",
        "ExecutionType": "EXPRESS"
      },
      "StartAt": "Process CSV data",
      "States": {
        "Process CSV data": {
          "Type": "Task",
          "Resource": "arn:aws:states:::lambda:invoke",
          "OutputPath": "$.Payload",
          "Parameters": {
            "Payload.$": "$",
            "FunctionName": "arn:aws:lambda:us-
east-2:123456789012:function:distributedMapLambda"
          },
          "End": true
        }
      }
    },
    "Label": "Processdata",
    "End": true,
    "ResultWriter": {
      "Resource": "arn:aws:states:::s3:putObject",
      "Parameters": {
        "Bucket": "mapOutputs",
        "Prefix": "resultData"
      }
    },
    "ItemSelector": {
      "index.$": "$$.Map.Item.Index",
      "value.$": "$$.Map.Item.Value"
    }
  }
}
```

2. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

For this tutorial, enter the name **DistributedMapDemo**.

3. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

For this tutorial, keep all the default selections in **State machine configuration**.

4. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

Note

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

Step 7: Run the state machine

An *execution* is an instance of your state machine where you run your workflow to perform tasks.

1. On the **DistributedMapDemo** page, choose **Start execution**.
2. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions automatically generates a unique execution name.

Note

Step Functions allows you to create names for state machines, executions, activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.
3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

For example, choose the Map state, and then choose **Map Run** to open the *Map Run Details* page. On this page, you can view all the execution details of the *Distributed Map state* and the child workflow executions that it started. For information about this page, see [Examining Map Run](#).

Processing entire batch of data with a Lambda function

In this tutorial, you use the *Distributed Map state*'s [ItemBatcher](#) field to process an entire batch of items inside a Lambda function. Each batch contains a maximum of three items. The *Distributed Map state* starts four child workflow executions, where each execution processes three items, while one execution processes a single item. Each child workflow execution invokes a Lambda function that iterates over the individual items present in the batch.

You'll create a state machine that performs multiplication on an array of integers. Say that the integer array you provide as input is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] and the multiplication factor is 7. Then, the resulting array formed after multiplying these integers with a factor of 7, will be [7, 14, 21, 28, 35, 42, 49, 56, 63, 70].

Topics

- [Step 1: Create the state machine](#)
- [Step 2: Create the Lambda function](#)
- [Step 3: Run the state machine](#)

Step 1: Create the state machine

In this step, you create the workflow prototype of the state machine that passes an entire batch of data to the Lambda function you'll create in [Step 2](#).

- Use the following definition to create a state machine using the [Step Functions console](#). For information about creating a state machine, see [Step 1: Create the workflow prototype](#) in the [Getting started with using Distributed Map state](#) tutorial.

In this state machine, you define a *Distributed Map state* that accepts an array of 10 integers as input and passes this array to a Lambda function in batches of 3. The Lambda function iterates over the individual items present in the batch and returns an output array named `multiplied`. The output array contains the result of the multiplication performed on the items passed in the input array.

Important

Make sure to replace the Amazon Resource Name (ARN) of the Lambda function in the following code with the ARN of the function you'll create in [Step 2](#).

```
{
  "StartAt": "Pass",
  "States": {
    "Pass": {
      "Type": "Pass",
      "Next": "Map",
      "Result": {
        "MyMultiplicationFactor": 7,
        "MyItems": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      }
    },
    "Map": {
      "Type": "Map",
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "DISTRIBUTED",
          "ExecutionType": "STANDARD"
        },
        "StartAt": "Lambda Invoke",
        "States": {
          "Lambda Invoke": {
            "Type": "Task",
            "Resource": "arn:aws:states:::lambda:invoke",
            "OutputPath": "$.Payload",
            "Parameters": {
```


⚠ Important

Ensure that your Lambda function is under the same AWS Region as your state machine.

To create the Lambda function

1. Use the [Lambda console](#) to create a **Python 3.9** Lambda function named **ProcessEntireBatch**. For information about creating a Lambda function, see [Step 4: Configure the Lambda function](#) in the [Getting started with using Distributed Map state](#) tutorial.
2. Copy the following code for the Lambda function and paste it into the **Code source** section of your Lambda function.

```
import json

def lambda_handler(event, context):
    multiplication_factor = event['BatchInput']['MyMultiplicationFactor']
    items = event['Items']

    results = [multiplication_factor * item for item in items]

    return {
        'statusCode': 200,
        'multiplied': results
    }
```

3. After you create your Lambda function, copy the function's ARN displayed in the upper-right corner of the page. To copy the ARN, click the



The following is an example ARN, where *function-name* is the name of the Lambda function (in this case, ProcessEntireBatch):

```
arn:aws:lambda:us-east-1:123456789012:function:function-name
```

You'll need to provide the function ARN in the state machine you created in [Step 1](#).

4. Choose **Deploy** to deploy the changes.

Step 3: Run the state machine

When you run the [state machine](#), the *Distributed Map state* starts four child workflow executions, where each execution processes three items, while one execution processes a single item.

The following example shows the data passed to the [ProcessEntireBatch](#) function by one of the child workflow executions.

```
{
  "BatchInput": {
    "MyMultiplicationFactor": 7
  },
  "Items": [1, 2, 3]
}
```

Given this input, the following example shows the output array named `multiplied` that is returned by the Lambda function.

```
{
  "statusCode": 200,
  "multiplied": [7, 14, 21]
}
```

The state machine returns the following output that contains four arrays named `multiplied` for the four child workflow executions. These arrays contain the multiplication results of the individual input items.

```
[
  {
    "statusCode": 200,
    "multiplied": [7, 14, 21]
  },
  {
    "statusCode": 200,
    "multiplied": [28, 35, 42]
  },
  {
    "statusCode": 200,
    "multiplied": [49, 56, 63]
  },
  {
```

```
    "statusCode": 200,  
    "multiplied": [70]  
  }  
]
```

To combine all the array items returned into a single output array, you can use the [ResultSelector](#) field. Define this field inside the *Distributed Map state* to find all the `multiplied` arrays, extract all the items inside these arrays, and then combine them into a single output array.

To use the `ResultSelector` field, update your state machine definition as shown in the following example.

```
{  
  "StartAt": "Pass",  
  "States": {  
    ...  
    ...  
    "Map": {  
      "Type": "Map",  
      ...  
      ...  
      "ItemsPath": "$.MyItems",  
      "ResultSelector": {  
        "multiplied.$": "$..multiplied[*]"  
      }  
    }  
  }  
}
```

The updated state machine returns a consolidated output array as shown in the following example.

```
{  
  "multiplied": [7, 14, 21, 28, 35, 42, 49, 56, 63, 70]  
}
```

Processing individual data items with a Lambda function

In this tutorial, you use the *Distributed Map state's* [ItemBatcher](#) field to iterate over individual items present in a batch using a Lambda function. The *Distributed Map state* starts four child workflow executions. Each of these child workflows runs an *Inline Map state*. For its each iteration, the *Inline*

Map state invokes a Lambda function and passes a single item from the batch to the function. The Lambda function then processes the item and returns the result.

You'll create a state machine that performs multiplication on an array of integers. Say that the integer array you provide as input is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] and the multiplication factor is 7. Then, the resulting array formed after multiplying these integers with a factor of 7, will be [7, 14, 21, 28, 35, 42, 49, 56, 63, 70].

Topics

- [Step 1: Create the state machine](#)
- [Step 2: Create the Lambda function](#)
- [Step 3: Run the state machine](#)

Step 1: Create the state machine

In this step, you create the workflow prototype of the state machine that passes a single item from a batch of items to each invocation of the Lambda function you'll create in [Step 2](#).

- Use the following definition to create a state machine using the [Step Functions console](#). For information about creating a state machine, see [Step 1: Create the workflow prototype](#) in the [Getting started with using Distributed Map state](#) tutorial.

In this state machine, you define a *Distributed Map state* that accepts an array of 10 integers as input and passes these array items to the child workflow executions in batches. Each child workflow execution receives a batch of three items as input and runs an *Inline Map state*. Every iteration of the *Inline Map state* invokes a Lambda function and passes an item from the batch to the function. This function then multiplies the item with a factor of 7 and returns the result.

The output of each child workflow execution is a JSON array that contains the multiplication result for each of the items passed.

Important

Make sure to replace the Amazon Resource Name (ARN) of the Lambda function in the following code with the ARN of the function you'll create in [Step 2](#).

```
{
  "StartAt": "Pass",
  "States": {
    "Pass": {
      "Type": "Pass",
      "Next": "Map",
      "Result": {
        "MyMultiplicationFactor": 7,
        "MyItems": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      }
    },
    "Map": {
      "Type": "Map",
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "DISTRIBUTED",
          "ExecutionType": "STANDARD"
        },
        "StartAt": "InnerMap",
        "States": {
          "InnerMap": {
            "Type": "Map",
            "ItemProcessor": {
              "ProcessorConfig": {
                "Mode": "INLINE"
              },
              "StartAt": "Lambda Invoke",
              "States": {
                "Lambda Invoke": {
                  "Type": "Task",
                  "Resource": "arn:aws:states:::lambda:invoke",
                  "OutputPath": "$.Payload",
                  "Parameters": {
                    "Payload.$": "$",
                    "FunctionName": "arn:aws:lambda:us-
east-1:123456789012:function: functionName"
                  },
                  "Retry": [
                    {
                      "ErrorEquals": [
                        "Lambda.ServiceException",
                        "Lambda.AWSLambdaException",
```

```
        "Lambda.SdkClientException",
        "Lambda.TooManyRequestsException"
    ],
    "IntervalSeconds": 2,
    "MaxAttempts": 6,
    "BackoffRate": 2
  }
],
"End": true
}
}
},
"End": true,
"ItemsPath": "$.Items",
"ItemSelector": {
  "MyMultiplicationFactor.$": "$.BatchInput.MyMultiplicationFactor",
  "MyItem.$": "$$.Map.Item.Value"
}
}
},
"End": true,
"Label": "Map",
"MaxConcurrency": 1000,
"ItemsPath": "$.MyItems",
"ItemBatcher": {
  "MaxItemsPerBatch": 3,
  "BatchInput": {
    "MyMultiplicationFactor.$": "$.MyMultiplicationFactor"
  }
}
}
}
}
```

Step 2: Create the Lambda function

In this step, you create the Lambda function that processes each item passed from the batch.

⚠ Important

Ensure that your Lambda function is under the same AWS Region as your state machine.

To create the Lambda function

1. Use the [Lambda console](#) to create a **Python 3.9** Lambda function named **ProcessSingleItem**. For information about creating a Lambda function, see [Step 4: Configure the Lambda function](#) in the [Getting started with using Distributed Map state](#) tutorial.
2. Copy the following code for the Lambda function and paste it into the **Code source** section of your Lambda function.

```
import json

def lambda_handler(event, context):

    multiplication_factor = event['MyMultiplicationFactor']
    item = event['MyItem']

    result = multiplication_factor * item

    return {
        'statusCode': 200,
        'multiplied': result
    }
```

3. After you create your Lambda function, copy the function's ARN displayed in the upper-right corner of the page. To copy the ARN, click the



The following is an example ARN, where *function-name* is the name of the Lambda function (in this case, ProcessSingleItem):

```
arn:aws:lambda:us-east-1:123456789012:function:function-name
```

You'll need to provide the function ARN in the state machine you created in [Step 1](#).

4. Choose **Deploy** to deploy the changes.

Step 3: Run the state machine

When you run the [state machine](#), the *Distributed Map state* starts four child workflow executions, where each execution processes three items, while one execution processes a single item.

The following example shows the data passed to one of the [ProcessSingleItem](#) function invocations inside a child workflow execution.

```
{
  "MyMultiplicationFactor": 7,
  "MyItem": 1
}
```

Given this input, the following example shows the output that is returned by the Lambda function.

```
{
  "statusCode": 200,
  "multiplied": 7
}
```

The following example shows the output JSON array for one of the child workflow executions.

```
[
  {
    "statusCode": 200,
    "multiplied": 7
  },
  {
    "statusCode": 200,
    "multiplied": 14
  },
  {
    "statusCode": 200,
    "multiplied": 21
  }
]
```

The state machine returns the following output that contains four arrays for the four child workflow executions. These arrays contain the multiplication results of the individual input items.

Finally, the state machine output is an array named `multiplied` that combines all the multiplication results returned for the four child workflow executions.

```
[
  [
    {
      "statusCode": 200,
      "multiplied": 7
    },
    {
      "statusCode": 200,
      "multiplied": 14
    },
    {
      "statusCode": 200,
      "multiplied": 21
    }
  ],
  [
    {
      "statusCode": 200,
      "multiplied": 28
    },
    {
      "statusCode": 200,
      "multiplied": 35
    },
    {
      "statusCode": 200,
      "multiplied": 42
    }
  ],
  [
    {
      "statusCode": 200,
      "multiplied": 49
    },
    {
      "statusCode": 200,
      "multiplied": 56
    },
    {
      "statusCode": 200,
```

```
    "multiplied": 63
  }
],
[
  {
    "statusCode": 200,
    "multiplied": 70
  }
]
]
```

To combine all the multiplication results returned by the child workflow executions into a single output array, you can use the [ResultSelector](#) field. Define this field inside the *Distributed Map state* to find all the results, extract the individual results, and then combine them into a single output array named `multiplied`.

To use the `ResultSelector` field, update your state machine definition as shown in the following example.

```
{
  "StartAt": "Pass",
  "States": {
    ...
    ...
    "Map": {
      "Type": "Map",
      ...
      ...
      "ItemBatcher": {
        "MaxItemsPerBatch": 3,
        "BatchInput": {
          "MyMultiplicationFactor.$": "$.MyMultiplicationFactor"
        }
      },
      "ItemsPath": "$.MyItems",
      "ResultSelector": {
        "multiplied.$": "$..multiplied"
      }
    }
  }
}
```

The updated state machine returns a consolidated output array as shown in the following example.

```
{
  "multiplied": [7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
}
```

Starting a State Machine Execution in Response to Amazon S3 Events

You can execute an AWS Step Functions state machine in response to an Amazon EventBridge rule.

This tutorial shows you how to configure a state machine as a target for an Amazon EventBridge rule. This rule will start a state machine execution when files are added to an Amazon Simple Storage Service (Amazon S3) bucket.

For a practical application, you could launch a state machine that performs operations on files that you add to the bucket, such as creating thumbnails or running Amazon Rekognition analysis on image and video files.

In this tutorial, you start the execution of a HelloWorld state machine by uploading a file to an Amazon S3 bucket. Then you review the example input of that execution to identify the information that is included in input from the Amazon S3 event notification delivered to EventBridge.

Topics

- [Prerequisite: Create a State Machine](#)
- [Step 1: Create a Bucket in Amazon S3](#)
- [Step 2: Enable Amazon S3 Event Notification with EventBridge](#)
- [Step 3: Create an Amazon EventBridge Rule](#)
- [Step 4: Test the Rule](#)
- [Example of Execution Input](#)

Prerequisite: Create a State Machine

Before you can configure a state machine as an Amazon EventBridge target, you must create the state machine.

- To create a basic state machine, use the [Creating state machine that uses a Lambda function](#) tutorial.
- If you already have a HelloWorld state machine, proceed to the next step.

Step 1: Create a Bucket in Amazon S3

Now that you have a HelloWorld state machine, you need to create an Amazon S3 bucket which stores your files. In Step 3 of this tutorial, you set up a rule so that when a file is uploaded to this bucket, EventBridge triggers an execution of your state machine.

1. Navigate to the [Amazon S3 console](#), and then choose **Create bucket** to create the bucket in which you want to store your files and trigger an Amazon S3 event rule.
2. Enter a **Bucket name**, such as *username*-sfn-tutorial.

Note

Bucket names must be unique across all existing bucket names in all AWS Regions in Amazon S3. Use your own *username* to make this name unique. You need to create all resources in the same AWS Region.

3. Keep all the default selections on the page, and choose **Create bucket**.

Step 2: Enable Amazon S3 Event Notification with EventBridge

After you create the Amazon S3 bucket, configure it to send events to EventBridge whenever certain events happen in your S3 bucket, such as file uploads.

1. Navigate to the [Amazon S3 console](#).
2. In the **Buckets** list, choose the name of the bucket that you want to enable events for.
3. Choose **Properties**.
4. Scroll down the page to view the **Event Notifications** section, and then choose **Edit** in the **Amazon EventBridge** subsection.
5. Under **Send notifications to Amazon EventBridge for all events in this bucket**, choose **On**.
6. Choose **Save changes**.

Note

After you enable EventBridge, it takes around five minutes for the changes to take effect.

Step 3: Create an Amazon EventBridge Rule

After you have a state machine, and have created the Amazon S3 bucket and configured it to send event notifications to EventBridge, create an EventBridge rule.

Note

You must configure EventBridge rule in the same AWS Region as the Amazon S3 bucket.

To create the rule

1. Navigate to the [Amazon EventBridge console](#), choose **Create rule**.

Tip

Alternatively, in the navigation pane on the EventBridge console, choose **Rules** under **Buses**, and then choose **Create rule**.

2. Enter a **Name** for your rule (for example, *S3Step Functions*) and optionally enter a **Description** for the rule.
3. For **Event bus** and **Rule type**, keep the default selections.
4. Choose **Next**. This opens the **Build event pattern** page.
5. Scroll down to the **Event pattern** section, and do the following:
 - a. For **Event source**, keep the default selection of **AWS events or EventBridge partner events**.
 - b. For **AWS service**, choose **Simple Storage Service (S3)**.
 - c. For **Event type**, choose **Amazon S3 Event Notification**.
 - d. Choose **Specific event(s)**, and then choose **Object Created**.

- e. Choose **Specific bucket(s) by name** and enter the bucket name you created in [Step 1](#) (*username-sfn-tutorial*) to store your files.
- f. Choose **Next**. This opens the **Select target(s)** page.

To create the target

1. In **Target 1**, keep the default selection of **AWS service**.
2. In the **Select a target** dropdown list, select **Step Functions state machine**.
3. In the **State machine** list, select the state machine that you [created earlier](#) (for example, `HelloWorld`).
4. Keep all the default selections on the page, and choose **Next**. This opens the **Configure tags** page.
5. Choose **Next** again. This opens the **Review and create** page.
6. Review the details of the rule and choose **Create rule**.

The rule is created and the **Rules** page is displayed, listing all your Amazon EventBridge rules.

Step 4: Test the Rule

Now that everything is in place, test adding a file to the Amazon S3 bucket, and then look at the input of the resulting state machine execution.

1. Add a file to your Amazon S3 bucket.

Navigate to the [Amazon S3 console](#), choose the bucket you created to store files (*username-sfn-tutorial*), and then choose **Upload**.

2. Add a file, for example *test.png*, and then choose **Upload**.

This launches an execution of your state machine, passing information from AWS CloudTrail as the input.

3. Check the execution for your state machine.

Navigate to the [Step Functions console and select the state machine used in your Amazon EventBridge rule \(HelloWorld\)](#).

4. Select the most recent execution of that state machine and expand the **Execution Input** section.

This input includes information such as the bucket name and the object name. In a real-world use case, a state machine can use this input to perform actions on that object.

Example of Execution Input

The following example shows a typical input to the state machine execution.

```
{
  "version": "0",
  "id": "6c540ad4-0671-9974-6511-756fbd7771c3",
  "detail-type": "Object Created",
  "source": "aws.s3",
  "account": "123456789012",
  "time": "2023-06-23T23:45:48Z",
  "region": "us-east-2",
  "resources": [
    "arn:aws:s3:::username-sfn-tutorial"
  ],
  "detail": {
    "version": "0",
    "bucket": {
      "name": "username-sfn-tutorial"
    },
    "object": {
      "key": "test.png",
      "size": 800704,
      "etag": "f31d8546bb67845b4d3048cde533b937",
      "sequencer": "00621049BA9A8C712B"
    },
    "request-id": "79104EXAMPLEB723",
    "requester": "123456789012",
    "source-ip-address": "200.0.100.11",
    "reason": "PutObject"
  }
}
```


Creating a Step Functions API using API Gateway

You can use Amazon API Gateway to associate your AWS Step Functions APIs with methods in an API Gateway API. When an HTTPS request is sent to an API method, API Gateway invokes your Step Functions API actions.

This tutorial shows you how to create an API that uses one resource and the POST method to communicate with the [StartExecution](#) API action. You'll use the AWS Identity and Access Management (IAM) console to create a role for API Gateway. Then, you'll use the API Gateway console to create an API Gateway API, create a resource and method, and map the method to the [StartExecution](#) API action. Finally, you'll deploy and test your API.

Note

Although Amazon API Gateway can start a Step Functions execution by calling [StartExecution](#), you must call [DescribeExecution](#) to get the result.

Topics

- [Step 1: Create an IAM Role for API Gateway](#)
- [Step 2: Create your API Gateway API](#)
- [Step 3: Test and Deploy the API Gateway API](#)

Step 1: Create an IAM Role for API Gateway

Before you create your API Gateway API, you need to give API Gateway permission to call Step Functions API actions.

To set up permissions for API Gateway

1. Sign in to the [IAM console](#) and choose **Roles, Create role**.
2. On the **Select trusted entity** page, do the following:
 - a. For **Trusted entity type**, keep the default selection of **AWS service**.
 - b. For **Use case**, choose **API Gateway** from the dropdown list.
3. Select **API Gateway**, and then choose **Next**.
4. On the **Add permissions** page, choose **Next**.

5. (Optional) On the **Name, review, and create** page, enter details, such as the role name. For example, enter **APIGatewayToStepFunctions**.
6. Choose **Create role**.

The IAM role appears in the list of roles.

7. Choose the name of your role and note the **Role ARN**, as shown in the following example.

```
arn:aws:iam::123456789012:role/APIGatewayToStepFunctions
```

To attach a policy to the IAM role

1. On the **Roles** page, search for your role (APIGatewayToStepFunctions), and then choose the role.
2. On the **Permissions** tab, choose **Add permissions**, and then choose **Attach policies**.
3. On the **Attach Policy** page, search for AWSStepFunctionsFullAccess, choose the policy, and then choose **Add permissions**.

Step 2: Create your API Gateway API

After you create your IAM role, you can create your custom API Gateway API.

To create the API

1. Open the [Amazon API Gateway console](#), and then choose **Create API**.
2. On the **Choose an API type** page, in the **REST API** pane, choose **Build**.
3. On the **Create REST API** page, select **New API**, and then enter **StartExecutionAPI** for the **API name**.
4. Keep **API endpoint type** as **Regional**, and then choose **Create API**.

To create a resource

1. On the **Resources** page of **StartExecutionAPI**, choose **Create resource**.
2. On the **Create resource** page, enter **execution** for **Resource name**, and then choose **Create resource**.

To create a POST method

1. Choose the **/execution** resource, and then choose **Create method**.
2. For **Method type**, choose POST.
3. For **Integration type**, choose **AWS service**.
4. For **AWS Region**, choose a Region from the list.

Note

For Regions that currently support Step Functions, see [Supported Regions](#).

5. For **AWS service**, choose **Step Functions** from the list.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, choose **POST** from the list.

Note

All Step Functions API actions use the HTTP POST method.

8. For **Action type**, select **Use action name**.
9. For **Action name**, enter **StartExecution**.
10. For **Execution role**, enter [the role ARN of the IAM role that you created earlier](#), as shown in the following example.

```
arn:aws:iam::123456789012:role/APIGatewayToStepFunctions
```

Method type

POST

Integration type

Lambda function
Integrate your API with a Lambda function.

HTTP
Integrate with an existing HTTP endpoint.

Mock
Generate a response based on API Gateway mappings and transformations.

AWS service
Integrate with an AWS Service.

VPC link
Integrate with a resource that isn't accessible over the public internet.

AWS Region

us-west-2

AWS service

Step Functions

AWS subdomain

HTTP method

POST

Action type

Use action name

Use path override

Action name - optional

StartExecution

Execution role

arn:aws:iam::555555555555:role/APIGatewayToStepFunctions

Credential cache

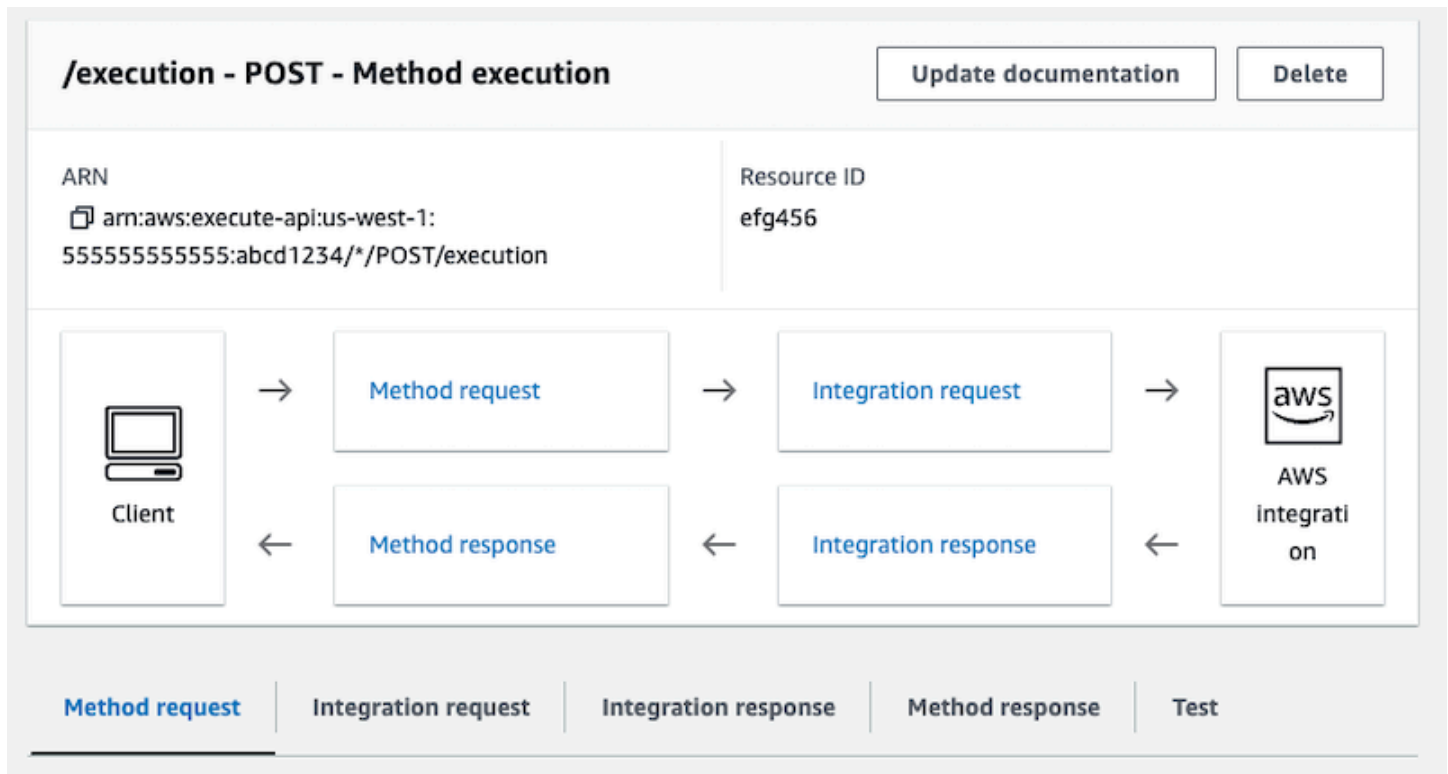
Do not add caller credentials to cache key

Default timeout
The default timeout is 29 seconds.

Cancel Create method

11. Keep the default options for **Credential cache** and **Default timeout**, and then choose **Save**.

The visual mapping between API Gateway and Step Functions is displayed on the **/execution - POST - Method execution** page.



Step 3: Test and Deploy the API Gateway API

Once you have created the API, test and deploy it.

To test the communication between API Gateway and Step Functions

1. On the **/execution - POST - Method Execution** page, choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. On the **/execution - POST - Method Test** tab, copy the following request parameters into the **Request body** section using the ARN of an existing state machine (or [create a new state machine that uses a Lambda function](#)), and then choose **Test**.

```
{
  "input": "{}",
  "name": "MyExecution",
  "stateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine>HelloWorld"
}
```

For more information, see the `StartExecution` [Request Syntax](#) in the *AWS Step Functions API Reference*.

Note

If you don't want to include the ARN of your state machine in the body of your API Gateway call, you can configure a mapping template in the **Integration request** tab, as shown in the following example.

```
{
  "input": "$util.escapeJavaScript($input.json('$'))",
  "stateMachineArn": "$util.escapeJavaScript($stageVariables.arn)"
}
```

With this approach, you can specify ARNs of different state machines based on your development stage (for example, dev, test, and prod). For more information about specifying stage variables in a mapping template, see [\\$stageVariables](#) in the *API Gateway Developer Guide*.

3. The execution starts and the execution ARN and its epoch date are displayed under **Response body**.

```
{
  "executionArn": "arn:aws:states:us-east-1:123456789012:execution>HelloWorld:MyExecution",
  "startDate": 1486768956.878
}
```

Note

You can view the execution by choosing your state machine on the [AWS Step Functions console](#).

To deploy your API

1. On the **Resources** page of *StartExecutionAPI*, choose **Deploy API**.

2. For **Stage**, select **New stage**.
3. For **Stage name**, enter **alpha**.
4. (Optional) For **Description**, enter a description.
5. Choose **Deploy**.

To test your deployment

1. On the **Stages** page of **StartExecutionAPI**, expand **alpha**, **/**, **/execution**, **POST**, and then choose the **POST** method.
2. Under **Method overrides**, choose the copy icon to copy your API's invoke URL. The full URL should look like the following example.

```
https://a1b2c3d4e5.execute-api.us-east-1.amazonaws.com/alpha/execution
```

3. From the command line, run the `curl` command using the ARN of your state machine, and then invoke the URL of your deployment, as shown in the following example.

```
curl -X POST -d '{"input": "{}", "name": "MyExecution", "stateMachineArn":  
  "arn:aws:states:us-east-1:123456789012:stateMachine>HelloWorld"}' https://  
a1b2c3d4e5.execute-api.us-east-1.amazonaws.com/alpha/execution
```

The execution ARN and its epoch date are returned, as shown in the following example.

```
{"executionArn": "arn:aws:states:us-  
east-1:123456789012:execution>HelloWorld:MyExecution", "startDate": 1.486772644911E9}
```

Note

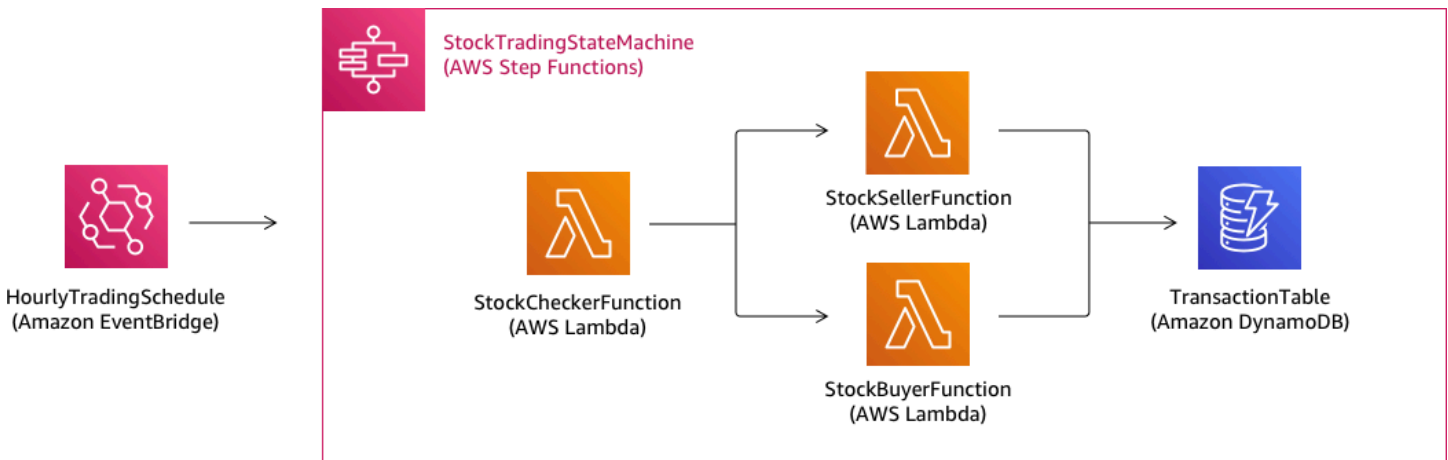
If you get a "Missing Authentication Token" error, make sure that the invoke URL ends with **/execution**.

Create a Step Functions state machine using AWS SAM

In this guide, you download, build, and deploy a sample AWS SAM application that contains an AWS Step Functions state machine. This application creates a mock stock trading workflow which

runs on a pre-defined schedule (note that the schedule is disabled by default to avoid incurring charges).

The following diagram shows the components of this application:



The following is a preview of commands that you run to create your sample application. For more details about each of these commands, see the sections later in this page

```
# Step 1 - Download a sample application. For this tutorial you
# will follow the prompts to select an AWS Quick Start Template
# called 'Multi-step workflow'
sam init

# Step 2 - Build your application
cd project-directory
sam build

# Step 3 - Deploy your application
sam deploy --guided
```

Prerequisites

This guide assumes that you've completed the steps in the [Installing the AWS SAM CLI](#) for your OS. It assumes that you've done the following:

1. Created an AWS account.
2. Configured IAM permissions.
3. Installed Homebrew. Note: Homebrew is only a prerequisite for Linux and macOS.

4. Installed the AWS SAM CLI. Note: Make sure you have version 0.52.0 or later. You can check which version you have by executing the command `sam --version`.

Step 1: Download a Sample AWS SAM Application

Command to run:

```
sam init
```

Follow the on-screen prompts to select the following:

1. **Template:** AWS Quick Start Templates
2. **Language:** Python, Ruby, NodeJS, Go, Java, or .NET
3. **Project name:** (name of your choice - default is sam-app)
4. **Quick start application:** Multi-step workflow

What AWS SAM is doing:

This command creates a directory with the name you provided for the 'Project name' prompt (default is sam-app). The specific contents of the directory will depend on the language you choose.

Following are the directory contents when you choose one of the Python runtimes:

```
### README.md
### functions
#   ### __init__.py
#   ### stock_buyer
# #   ### __init__.py
# #   ### app.py
# #   ### requirements.txt
#   ### stock_checker
# #   ### __init__.py
# #   ### app.py
# #   ### requirements.txt
#   ### stock_seller
#     ### __init__.py
#     ### app.py
#     ### requirements.txt
### statemachine
```

```
#   ### stock_trader.asl.json
### template.yaml
### tests
    ### unit
        ### __init__.py
        ### test_buyer.py
        ### test_checker.py
        ### test_seller.py
```

There are two especially interesting files that you can take a look at:

- `template.yaml`: Contains the AWS SAM template that defines your application's AWS resources.
- `statemachine/stockTrader.asl.json`: Contains the application's state machine definition, which is written in [Amazon States Language](#).

You can see the following entry in the `template.yaml` file, which points to the state machine definition file:

```
Properties:
  DefinitionUri: statemachine/stock_trader.asl.json
```

It can be helpful to keep the state machine definition as a separate file instead of embedding it in the AWS SAM template. For example, tracking changes to the state machine definition is easier if you don't include the definition in the template. You can use the Workflow Studio to create and maintain the state machine definition, and export the definition from the console directly to the Amazon States Language specification file without merging it into the template.

For more information about the sample application, see the `README.md` file in the project directory.

Step 2: Build Your Application

Command to run:

First change into the project directory (that is, the directory where the `template.yaml` file for the sample application is located; by default is `sam-app`), then run this command:

```
sam build
```

Example output:

```
Build Succeeded

Built Artifacts  : .aws-sam/build
Built Template   : .aws-sam/build/template.yaml

Commands you can use next
=====
[*] Invoke Function: sam local invoke
[*] Deploy: sam deploy --guided
```

What AWS SAM is doing:

The AWS SAM CLI comes with abstractions for a number of Lambda runtimes to build your dependencies, and copies all build artifacts into staging folders so that everything is ready to be packaged and deployed. The `sam build` command builds any dependencies that your application has, and copies the build artifacts to folders under `.aws-sam/build`.

Step 3: Deploy Your Application to the AWS Cloud

Command to run:

```
sam deploy --guided
```

Follow the on-screen prompts. You can just respond with Enter to accept the default options provided in the interactive experience.

What AWS SAM is doing:

This command deploys your application to the AWS cloud. It takes the deployment artifacts you build with the `sam build` command, packages and uploads them to an Amazon S3 bucket created by AWS SAM CLI, and deploys the application using AWS CloudFormation. In the output of the deploy command you can see the changes being made to your AWS CloudFormation stack.

You can verify the example Step Functions state machine was successfully deployed by following these steps:

1. Sign in to the AWS Management Console and open the Step Functions console at <https://console.aws.amazon.com/states/>.
2. In the left navigation, choose **State machines**.
3. Find and choose your new state machine in the list. It will be named `StockTradingStateMachine-<unique-hash>`.
4. Choose the **Definition** tab.

You should now see a visual representation of your state machine. You can verify that the visual representation matches the state machine definition found in the `statemachine/stockTrader.asl.json` file of your project directory.

Troubleshooting

SAM CLI error: "no such option: --guided"

When executing `sam deploy`, you see the following error:

```
Error: no such option: --guided
```

This means that you are using an older version of the AWS SAM CLI that does not support the `--guided` parameter. To fix this, you can either update your version of AWS SAM CLI to 0.33.0 or later, or omit the `--guided` parameter from the `sam deploy` command.

SAM CLI error: "Failed to create managed resources: Unable to locate credentials"

When executing `sam deploy`, you see the following error:

```
Error: Failed to create managed resources: Unable to locate credentials
```

This means that you have not set up AWS credentials to enable the AWS SAM CLI to make AWS service calls. To fix this, you must set up AWS credentials. For more information, see [Setting Up AWS Credentials](#) in the *AWS Serverless Application Model Developer Guide*.

Clean Up

If you no longer need the AWS resources you created by running this tutorial, you can remove them by deleting the AWS CloudFormation stack that you deployed.

To delete the AWS CloudFormation stack created with this tutorial using the AWS Management Console, follow these steps:

1. Sign in to the AWS Management Console and open the AWS CloudFormation console at <https://console.aws.amazon.com/cloudformation>.
2. In the left navigation pane, choose **Stacks**.
3. In the list of stacks, choose **sam-app** (or the name of stack you created).
4. Choose **Delete**.

When done, the status of the of the stack will change to **DELETE_COMPLETE**.

Alternatively, you can delete the AWS CloudFormation stack by executing the following AWS CLI command:

```
aws cloudformation delete-stack --stack-name sam-app --region region
```

Verify Deleted Stack

For both methods of deleting the AWS CloudFormation stack, you can verify it was deleted by going to the <https://console.aws.amazon.com/cloudformation>, choosing **Stacks** in the left navigation pane, and choosing **Deleted** in the dropdown to the right of the search text box. You should see your stack name **sam-app** (or the name of the stack you created) in the list of deleted stacks.

Creating an Activity state machine using Step Functions

This tutorial shows you how to create an activity-based state machine using Java and AWS Step Functions. Activities allow you to control worker code that runs somewhere else from your state machine. For an overview, see [Activities](#) in [How Step Functions works](#).

To complete this tutorial, you need the following:

- The [SDK for Java](#). The example activity in this tutorial is a Java application that uses the AWS SDK for Java to communicate with AWS.
- AWS credentials in the environment or in the standard AWS configuration file. For more information, see [Set Up Your AWS Credentials](#) in the *AWS SDK for Java Developer Guide*.

Topics

- [Step 1: Create an Activity](#)
- [Step 2: Create a state machine](#)
- [Step 3: Implement a Worker](#)
- [Step 4: Run the state machine](#)
- [Step 5: Run and Stop the Worker](#)

Step 1: Create an Activity

You must make Step Functions aware of the *activity* whose *worker* (a program) you want to create. Step Functions responds with an Amazon Resource Name(ARN) that establishes an identity for the activity. Use this identity to coordinate the information passed between your state machine and worker.

Important

Ensure that your activity task is under the same AWS account as your state machine.

1. In the [Step Functions console](#), in the navigation pane on the left, choose **Activities**.
2. Choose **Create activity**.
3. Enter a **Name** for the activity, for example, *get-greeting*, and then choose **Create activity**.
4. When your activity task is created, make a note of its ARN, as shown in the following example.

```
arn:aws:states:us-east-1:123456789012:activity:get-greeting
```

Step 2: Create a state machine

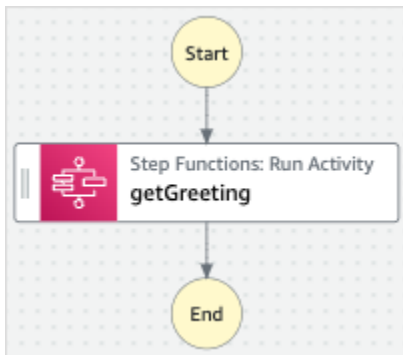
Create a state machine that determines when your activity is invoked and when your worker should perform its primary work, collect its results, and return them. To create the state machine, you'll use the [Code editor](#) of Workflow Studio.

1. In the [Step Functions console](#), in the navigation pane on the left, choose **State machines**.
2. On the **State machines** page, choose **Create state machine**.
3. In the **Choose a template** dialog box, select **Blank**.
4. Choose **Select**. This opens Workflow Studio in [Design mode](#).
5. For this tutorial, you'll write the [Amazon States Language](#) (ASL) definition of your state machine in the code editor. To do this, choose **Code**.
6. Remove the existing boilerplate code and paste the following code. Remember to replace the example ARN in this code with the ARN of [the activity task that you created earlier](#) in the Resource field.

```
{
  "Comment": "An example using a Task state.",
  "StartAt": "getGreeting",
  "Version": "1.0",
  "TimeoutSeconds": 300,
  "States": {
    {
      "getGreeting": {
        "Type": "Task",
        "Resource": "arn:aws:states:us-east-1:123456789012:activity:get-greeting",
        "End": true
      }
    }
  }
}
```

This is a description of your state machine using the [Amazon States Language](#) (ASL). It defines a single Task state named `getGreeting`. For more information, see [State Machine Structure](#).

7. On the [Graph visualization pane](#), make sure the workflow graph for the ASL definition you added looks similar to the following graph.



- Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

For this tutorial, enter the name **ActivityStateMachine**.

- (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

For this tutorial, keep all the default selections in **State machine settings**.

If you've [previously created an IAM role](#) with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

- In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

Note

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

Step 3: Implement a Worker

Create a *worker*. A worker is a program that is responsible for:

- Polling Step Functions for activities using the `GetActivityTask` API action.

- Performing the work of the activity using your code, (for example, the `getGreeting()` method in the following code).
- Returning the results using the `SendTaskSuccess`, `SendTaskFailure`, and `SendTaskHeartbeat` API actions.

Note

For a more complete example of an activity worker, see [Example Activity Worker in Ruby](#). This example provides an implementation based on best practices, which you can use as a reference for your activity worker. The code implements a consumer-producer pattern with a configurable number of threads for pollers and activity workers.

To implement the worker

1. Create a file named `GreeterActivities.java`.
2. Add the following code to it.

```
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.EnvironmentVariableCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.stepfunctions.AWSStepFunctions;
import com.amazonaws.services.stepfunctions.AWSStepFunctionsClientBuilder;
import com.amazonaws.services.stepfunctions.model.GetActivityTaskRequest;
import com.amazonaws.services.stepfunctions.model.GetActivityTaskResult;
import com.amazonaws.services.stepfunctions.model.SendTaskFailureRequest;
import com.amazonaws.services.stepfunctions.model.SendTaskSuccessRequest;
import com.amazonaws.util.json.Jackson;
import com.fasterxml.jackson.databind.JsonNode;
import java.util.concurrent.TimeUnit;

public class GreeterActivities {

    public String getGreeting(String who) throws Exception {
        return "{\"Hello\": \"" + who + "\"}";
    }

    public static void main(final String[] args) throws Exception {
        GreeterActivities greeterActivities = new GreeterActivities();
    }
}
```

```
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setSocketTimeout((int)TimeUnit.SECONDS.toMillis(70));

AWSStepFunctions client = AWSStepFunctionsClientBuilder.standard()
    .withRegion(Regions.US_EAST_1)
    .withCredentials(new EnvironmentVariableCredentialsProvider())
    .withClientConfiguration(clientConfiguration)
    .build();

while (true) {
    GetActivityTaskResult getActivityTaskResult =
        client.getActivityTask(
            new
GetActivityTaskRequest().withActivityArn(ACTIVITY_ARN));

    if (getActivityTaskResult.getTaskToken() != null) {
        try {
            JsonNode json =
Jackson.jsonNodeOf(getActivityTaskResult.getInput());
            String greetingResult =

greeterActivities.getGreeting(json.get("who").textValue());
            client.sendTaskSuccess(
                new SendTaskSuccessRequest().withOutput(

greetingResult).withTaskToken(getActivityTaskResult.getTaskToken()));
        } catch (Exception e) {
            client.sendTaskFailure(new
SendTaskFailureRequest().withTaskToken(
                getActivityTaskResult.getTaskToken()));
        }
    } else {
        Thread.sleep(1000);
    }
}
}
```

Note

The `EnvironmentVariableCredentialsProvider` class in this example assumes that the `AWS_ACCESS_KEY_ID` (or `AWS_ACCESS_KEY`) and `AWS_SECRET_KEY` (or `AWS_SECRET_ACCESS_KEY`) environment variables are set. For more information

about providing the required credentials to the factory, see [AWSCredentialsProvider](#) in the *AWS SDK for Java API Reference* and [Set Up AWS Credentials and Region for Development](#) in the *AWS SDK for Java Developer Guide*.

By default the AWS SDK will wait up to 50 seconds to receive data from the server for any operation. The `GetActivityTask` operation is a long-poll operation that will wait up to 60 seconds for the next available task. To prevent receiving a `SocketTimeoutException` error, set `SocketTimeout` to 70 seconds.

3. In the parameter list of the `GetActivityTaskRequest().withActivityArn()` constructor, replace the `ACTIVITY_ARN` value with the ARN of [the activity task that you created earlier](#).

Step 4: Run the state machine

When you start the execution of the state machine, your worker polls Step Functions for activities, performs its work (using the input that you provide), and returns its results.

1. On the **ActivityStateMachine** page, choose **Start execution**.

The **Start execution** dialog box is displayed.

2. In the **Start execution** dialog box, do the following:
 - a. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

- b. In the **Input** box, enter the following JSON input to run your workflow.

```
{
  "who": "AWS Step Functions"
}
```

- c. Choose **Start execution**.
- d. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Step 5: Run and Stop the Worker

To have the worker poll your state machine for activities, you must run the worker.

1. On the command line, navigate to the directory in which you created `GreeterActivities.java`.
2. To use the AWS SDK, add the full path of the `lib` and `third-party` directories to the dependencies of your build file and to your Java `CLASSPATH`. For more information, see [Downloading and Extracting the SDK](#) in the *AWS SDK for Java Developer Guide*.
3. Compile the file.

```
$ javac GreeterActivities.java
```

4. Run the file.

```
$ java GreeterActivities
```

5. On the [Step Functions console](#), navigate to the *Execution Details* page.
6. When the execution completes, examine the results of your execution.
7. Stop the worker.

Iterate a loop with Lambda

In this tutorial, you implement a design pattern that uses a state machine and an AWS Lambda function to iterate a loop a specific number of times.

Use this design pattern any time you need to keep track of the number of loops in a state machine. This implementation can help you break up large tasks or long-running executions into smaller chunks, or to end an execution after a specific number of events. You can use a similar implementation to periodically end and restart a long-running execution to avoid exceeding service quotas for AWS Step Functions, AWS Lambda, or other AWS services.

Before you begin, go through the [Creating a Step Functions state machine that uses Lambda](#) tutorial to ensure you are familiar with using Lambda and Step Functions together.

Step 1: Create a Lambda function to iterate a count

By using a Lambda function you can track the number of iterations of a loop in your state machine. The following Lambda function receives input values for `count`, `index`, and `step`. It returns these values with an updated `index` and a Boolean value named `continue`. The Lambda function sets `continue` to `true` if the `index` is less than `count`.

Your state machine then implements a `Choice` state that executes some application logic if `continue` is `true`, or exits if it is `false`.

To create the Lambda function

1. Sign in to the [Lambda console](#), and then choose **Create function**.
2. On the **Create function** page, choose **Author from scratch**.
3. In the **Basic information** section, configure your Lambda function, as follows:
 - a. For **Function name**, enter `Iterator`.
 - b. For **Runtime**, choose **Node.js**.
 - c. In **Change default execution role**, choose **Create a new role with basic Lambda permissions**.
 - d. Choose **Create function**.
4. Copy the following code for the Lambda function into the **Code source**.

```
export const handler = function (event, context, callback) {
  let index = event.iterator.index
  let step = event.iterator.step
  let count = event.iterator.count

  index = index + step
```

```
    callback(null, {
      index,
      step,
      count,
      continue: index < count
    })
  }
}
```

This code accepts input values for `count`, `index`, and `step`. It increments the `index` by the value of `step` and returns these values, and the Boolean `continue`. The value of `continue` is `true` if `index` is less than `count`.

5. Choose **Deploy**.

Step 2: Test the Lambda Function

Run your Lambda function with numeric values to see it in operation. You can provide input values for your Lambda function that mimic an iteration.

To test your Lambda function

1. Choose **Test**.
2. In the **Configure test event** dialog box, enter `TestIterator` in the **Event name** box.
3. Replace the example data with the following.

```
{
  "Comment": "Test my Iterator function",
  "iterator": {
    "count": 10,
    "index": 5,
    "step": 1
  }
}
```

These values mimic what would come from your state machine during an iteration. The Lambda function will increment the `index` and return `true` for `continue` when the `index` is less than `count`. For this test, the `index` has already incremented to 5. The test will increment `index` to 6 and set `continue` to `true`.

4. Choose **Create**.

5. Choose **Test** to test your Lambda function.

The results of the test are displayed in the **Execution results** tab.

6. Choose the **Execution results** tab to see the output.

```
{
  "index": 6,
  "step": 1,
  "count": 10,
  "continue": true
}
```

Note

If you set index to 9 and test again, the index increments to 10, and continue will be false.

Step 3: Create a State Machine

Before you leave the Lambda console...

Copy the Lambda function ARN. Paste it into a note. You'll need it in the next step.

Next, you will create a state machine with the following states:

- **ConfigureCount** – Sets default values for count, index, and step.
- **Iterator** – Refers to the Lambda function you created earlier, passing in the values configured in **ConfigureCount**.
- **IsCountReached** – A choice state that continues the loop or proceeds to Done state, based on the value returned from your **Iterator** function.
- **ExampleWork** – A stub for work that needs to be done. In this example, the workflow has a **Pass** state, but in a real solution, you would likely use a **Task**.
- **Done** – End state of your workflow.

To create the state machine in the console:

1. Open the [Step Functions console](#), and then choose **Create a state machine**.

⚠ Important

Your state machine must be in the same AWS account and Region as your Lambda function.

2. Select the **Blank** template.
3. In the **Code** pane, paste the following JSON which defines the state machine.

For more information about the Amazon States Language, see [State Machine Structure](#).

```
{
  "Comment": "Iterator State Machine Example",
  "StartAt": "ConfigureCount",
  "States": {
    "ConfigureCount": {
      "Type": "Pass",
      "Result": {
        "count": 10,
        "index": 0,
        "step": 1
      },
      "ResultPath": "$.iterator",
      "Next": "Iterator"
    },
    "Iterator": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Iterate",
      "ResultPath": "$.iterator",
      "Next": "IsCountReached"
    },
    "IsCountReached": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.iterator.continue",
          "BooleanEquals": true,
          "Next": "ExampleWork"
        }
      ]
    }
  }
}
```



```

        }
      ],
      "Default": "Done"
    },
    "ExampleWork": {
      "Comment": "Your application logic, to run a specific number of times",
      "Type": "Pass",
      "Result": {
        "success": true
      },
      "ResultPath": "$.result",
      "Next": "Iterator"
    },
    "Done": {
      "Type": "Pass",
      "End": true
    }
  }
}

```

4. Replace the `Iterator Resource` field with the ARN for your `Iterator Lambda` function that you created earlier.
5. Select **Config**, and enter a **Name** for your state machine, such as *IterateCount*.

Note

Names of state machines, executions, and activity tasks must not exceed 80 characters in length. These names must be unique for your account and AWS Region, and must not contain any of the following:

- Whitespace
- Wildcard characters (? *)
- Bracket characters (< > { } [])
- Special characters (" # % \ ^ | ~ ` \$ & , ; : /)
- Control characters (\\u0000 - \\u001f or \\u007f - \\u009f).

If your state machine is of type `Express`, you can provide the same name to multiple executions of the state machine. Step Functions generates a unique execution ARN

for each Express state machine execution, even if multiple executions have the same name.

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

6. For **Type**, accept default value of **Standard**. For **Permissions**, choose **Create new role**.
7. Choose **Create**, and then **Confirm** the role creations.

Step 4: Start a New Execution

After you create your state machine, you can start an execution.

1. On the **IterateCount** page, choose **Start execution**.
2. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

3. Choose **Start Execution**.

A new execution of your state machine starts, showing your running execution.

State machine graph view, showing Iterator state in blue to indicate in progress status.

The execution increments in steps, tracking the count using your Lambda function. On each iteration, it performs the example work referenced in the `ExampleWork` state in your state machine.

When the count reaches the number specified in the `ConfigureCount` state in your state machine, the execution quits iterating and ends.

State machine graph view, showing Iterator state and Done state in green to indicate both have succeeded.

Continuing Long-running Workflow Executions as a New Execution

AWS Step Functions is designed to run workflows that have a finite duration and number of steps. Executions have a maximum duration of one year, and a maximum of 25,000 events (see [Quotas](#)).

For long-running executions, to avoid reaching the hard quota of 25,000 entries in the execution event history, we recommend that you start a new workflow execution directly from the Task state of a state machine. This allows you to break your workflows into smaller state machines, and to continue your ongoing work in a new execution. To start these workflow executions, call the `StartExecution` API action from your Task state and pass the necessary parameters.

Alternatively, you can also implement a pattern that uses a Lambda function to start a new execution of your state machine to split ongoing work across multiple workflow executions.

This tutorial shows you both the approaches to continue workflow executions without exceeding service quotas.

Topics

- [Using a Step Functions API action to continue a new execution \(recommended\)](#)
- [Using a Lambda function to continue a new execution](#)

Using a Step Functions API action to continue a new execution (recommended)

Step Functions can start workflow executions by calling its own API as an [integrated service](#). We recommend that you use this approach to avoid exceeding service quotas for long-running executions.

Step 1: Create a long-running state machine

Create a long-running state machine that you want to start from the Task state of a different state machine. For this tutorial, use the [state machine that uses a Lambda function](#).

Note

Make sure to copy the name and Amazon Resource Name of this state machine in a text file for later use.

Step 2: Create a state machine to call the Step Functions API action

To start workflow executions from a Task state

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).
4. From the **Actions** tab, drag the **StartExecution** API action and drop it on the empty state labelled **Drag first state here**.
5. Choose the **StartExecution** state and do the following in the **Configuration** tab in [Design mode](#):
 - a. Rename the state to **Start nested execution**.
 - b. For **Integration type**, choose **AWS SDK - new** from the dropdown list.
 - c. In **API Parameters**, do the following:
 - i. For `StateMachineArn`, replace the sample Amazon Resource Name with the ARN of your state machine. For example, enter the ARN of the [state machine that uses Lambda](#).
 - ii. For `Input` node, replace the existing placeholder text with the following value:

```
"Comment": "Starting workflow execution using a Step Functions API action"
```

- iii. Make sure your inputs in **API Parameters** look similar to the following:

```
{
  "StateMachineArn": "arn:aws:states:us-
east-2:123456789012:stateMachine:LambdaStateMachine",
  "Input": {
    "Comment": "Starting workflow execution using a Step Functions API
action",
    "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
```

```
}
```

- (Optional) Choose **Definition** on the [Inspector](#) panel to view the automatically-generated [Amazon States Language](#) (ASL) definition of your workflow.

Tip

You can also view the ASL definition in the [Code editor](#) of Workflow Studio. In the code editor, you can also edit the ASL definition of your workflow.

- Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

For this tutorial, enter the name **ParentStateMachine**.

- (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

For this tutorial, keep all the default selections in **State machine settings**.

If you've [previously created an IAM role](#) with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

- In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

Note

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

Step 3: Update the IAM policy

To make sure your state machine has permissions to start the execution of the [state machine that uses a Lambda function](#), you need to attach an inline policy to your state machine's IAM role. For more information, see [Embedding Inline Policies](#) in the *IAM User Guide*.

1. On the **ParentStateMachine** page, choose the **IAM role ARN** to navigate to the **IAM Roles** page for your state machine.
2. Assign an appropriate permission to the IAM role of the **ParentStateMachine** for it to be able to start execution of another state machine. To assign the permission, do the following:
 - a. On the **IAM Roles** page, choose **Add permissions**, and then choose **Create inline policy**.
 - b. On the **Create policy** page, choose the **JSON** tab.
 - c. Replace the existing text with the following policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": [
        "arn:aws:states:us-
east-2:123456789012:stateMachine:LambdaStateMachine"
      ]
    }
  ]
}
```

- d. Choose **Review policy**.
- e. Specify a name for the policy, and then choose **Create policy**.

Step 4: Run the state machine

State machine executions are instances where you run your workflow to perform tasks.

1. On the **ParentStateMachine** page, choose **Start execution**.

The **Start execution** dialog box is displayed.

2. In the **Start execution** dialog box, do the following:
 - a. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

- b. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.
- c. Choose **Start execution**.
- d. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

3. Open the **LambdaStateMachine** page and notice a new execution triggered by the **ParentStateMachine**.

Using a Lambda function to continue a new execution

You can create a state machine that uses a Lambda function to start a new execution before the current execution terminates. Using this approach to continue your ongoing work in a new execution enables you to have a state machine that can break large jobs into smaller workflows, or to have a state machine that runs indefinitely.

This tutorial builds on the concept of using an external Lambda function to modify your workflow, which was demonstrated in the [Iterate a loop with Lambda](#) tutorial. You use the same Lambda function (`Iterator`) to iterate a loop for a specific number of times. In addition, you create another Lambda function to start a new execution of your workflow, and to decrement a count each time it starts a new execution. By setting the number of executions in the input, this state machine ends and restarts an execution a specified number of times.

The state machine you'll create implements the following states.

State	Purpose
ConfigureCount	A Pass state that configures the count, index, and step values that the <code>Iterator</code> Lambda function uses to step through iterations of work.
Iterator	A Task state that references the <code>Iterator</code> Lambda function.
IsCountReached	A Choice state that uses a Boolean value from the <code>Iterator</code> function to decide whether the state machine should continue the example work, or move to the <code>ShouldRestart</code> state.
ExampleWork	A Pass state that represents the <code>Task</code> state that would perform work in an actual implementation.
ShouldRestart	A Choice state that uses the <code>executionCount</code> value to decide whether it should end one execution and start another, or simply end.
Restart	A Task state that uses a Lambda function to start a new execution of your state machine. Like the <code>Iterator</code> function, this function also decrements a count. The <code>Restart</code> state passes the decremented value of the count to the input of the new execution.

Prerequisites

Before you begin, go through the [Creating a Step Functions state machine that uses Lambda](#) tutorial to ensure that you're familiar with using Lambda and Step Functions together.

Topics

- [Step 1: Create a Lambda function to iterate a count](#)
- [Step 2: Create a Restart Lambda function to start a new Step Functions execution](#)
- [Step 3: Create a state machine](#)
- [Step 4: Update the IAM Policy](#)
- [Step 5: Run the state machine](#)

Step 1: Create a Lambda function to iterate a count

Note

If you have completed the [Iterate a loop with Lambda](#) tutorial, you can skip this step and use that Lambda function.

This section and the [Iterate a loop with Lambda](#) tutorial show how you can use a Lambda function to track a count, for example, the number of iterations of a loop in your state machine.

The following Lambda function receives input values for `count`, `index`, and `step`. It returns these values with an updated `index` and a Boolean named `continue`. The Lambda function sets `continue` to `true` if the `index` is less than `count`.

Your state machine then implements a `Choice` state that executes some application logic if `continue` is `true`, or moves on to `ShouldRestart` if `continue` is `false`.

Create the Iterate Lambda function

1. Open the [Lambda console](#), and then choose **Create function**.
2. On the **Create function** page, choose **Author from scratch**.
3. In the **Basic information** section, configure your Lambda function, as follows:
 - a. For **Function name**, enter `Iterator`.
 - b. For **Runtime**, choose **Node.js 16.x**.
 - c. Keep all the default selections on the page, and then choose **Create function**.

When your Lambda function is created, make a note of its Amazon Resource Name (ARN) in the upper-right corner of the page, for example:

```
arn:aws:lambda:us-east-1:123456789012:function:Iterator
```

4. Copy the following code for the Lambda function into the **Code source** section of the ***Iterator*** page in the Lambda console.

```
exports.handler = function iterator (event, context, callback) {  
  let index = event.iterator.index;  
  let step = event.iterator.step;
```

```
let count = event.iterator.count;

index = index + step;

callback(null, {
  index,
  step,
  count,
  continue: index < count
})
}
```

This code accepts input values for `count`, `index`, and `step`. It increments the `index` by the value of `step` and returns these values, and the Boolean value of `continue`. The value of `continue` is `true` if `index` is less than `count`.

5. Choose **Deploy** to deploy the code.

Test the Iterate Lambda function

To see your `Iterate` function working, run it with numeric values. You can provide input values for your Lambda function that mimic an iteration to see what output you get with specific input values.

To test your Lambda function

1. In the **Configure test event** dialog box, choose **Create new test event**, and then type `TestIterator` for **Event name**.
2. Replace the example data with the following.

```
{
  "Comment": "Test my Iterator function",
  "iterator": {
    "count": 10,
    "index": 5,
    "step": 1
  }
}
```

These values mimic what would come from your state machine during an iteration. The Lambda function increments the `index` and returns `continue` as `true`. When the `index` is

not less than the count, it returns `continue` as `false`. For this test, the index has already incremented to 5. The results should increment the index to 6 and set `continue` to `true`.

3. Choose **Create**.
4. On the **Iterator** page in your Lambda console, be sure **TestIterator** is listed, and then choose **Test**.

The results of the test are displayed at the top of the page. Choose **Details** and review the result.

```
{
  "index": 6,
  "step": 1,
  "count": 10,
  "continue": true
}
```

 **Note**

If you set `index` to 9 for this test, the index increments to 10, and `continue` is `false`.

Step 2: Create a Restart Lambda function to start a new Step Functions execution

1. Open the [Lambda console](#), and then choose **Create function**.
2. On the **Create function** page, choose **Author from scratch**.
3. In the **Basic information** section, configure your Lambda function, as follows:
 - a. For **Function name**, enter `Restart`.
 - b. For **Runtime**, choose **Node.js 16.x**.
4. Keep all the default selections on the page, and then choose **Create function**.

When your Lambda function is created, make a note of its Amazon Resource Name (ARN) in the upper-right corner of the page, for example:

```
arn:aws:lambda:us-east-1:123456789012:function:Iterator
```

5. Copy the following code for the Lambda function into the **Code source** section of the **Restart** page in the Lambda console.

The following code decrements a count of the number of executions, and starts a new execution of your state machine, including the decremented value.

```
var aws = require('aws-sdk');
var sfn = new aws.StepFunctions();

exports.restart = function(event, context, callback) {

  let StateMachineArn = event.restart.StateMachineArn;
  event.restart.executionCount -= 1;
  event = JSON.stringify(event);

  let params = {
    input: event,
    stateMachineArn: StateMachineArn
  };

  sfn.startExecution(params, function(err, data) {
    if (err) callback(err);
    else callback(null, event);
  });
}
```

6. Choose **Deploy** to deploy the code.

Step 3: Create a state machine

Now that you've created your two Lambda functions, create a state machine. In this state machine, the `ShouldRestart` and `Restart` states are how you break your work across multiple executions.

Example `ShouldRestart` Choice state

The following excerpt shows the `ShouldRestart` [Choice](#) state. This state determines whether or not you should restart the execution.

```
"ShouldRestart": {
  "Type": "Choice",
  "Choices": [
```

```
{
  "Variable": "$.restart.executionCount",
  "NumericGreaterThan": 1,
  "Next": "Restart"
}
],
```

The `$.restart.executionCount` value is included in the input of the initial execution. It's decremented by one each time the `Restart` function is called, and then placed into the input for each subsequent execution.

Example Restart Task state

The following excerpt shows the `Restart` [Task](#) state. This state uses the Lambda function you created earlier to restart the execution, and to decrement the count to track the remaining number of executions to start.

```
"Restart": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Restart",
  "Next": "Done"
},
```

To create the state machine

1. Open the [Step Functions console](#) and choose **Create state machine**.

Important

Make sure that your state machine is under the same AWS account and Region as the Lambda functions you created earlier in [Step 1](#) and [Step 2](#).

2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).
4. For this tutorial, you'll write the [Amazon States Language](#) (ASL) definition of your state machine in the [Code editor](#). To do this, choose **Code**.
5. Remove the existing boilerplate code and paste the following code. Remember to replace the ARNs in this code with the ARNs of the Lambda functions you created.

```
{
```

```
"Comment": "Continue-as-new State Machine Example",
"StartAt": "ConfigureCount",
"States": {
  "ConfigureCount": {
    "Type": "Pass",
    "Result": {
      "count": 100,
      "index": -1,
      "step": 1
    },
    "ResultPath": "$.iterator",
    "Next": "Iterator"
  },
  "Iterator": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Iterator",
    "ResultPath": "$.iterator",
    "Next": "IsCountReached"
  },
  "IsCountReached": {
    "Type": "Choice",
    "Choices": [
      {
        "Variable": "$.iterator.continue",
        "BooleanEquals": true,
        "Next": "ExampleWork"
      }
    ],
    "Default": "ShouldRestart"
  },
  "ExampleWork": {
    "Comment": "Your application logic, to run a specific number of times",
    "Type": "Pass",
    "Result": {
      "success": true
    },
    "ResultPath": "$.result",
    "Next": "Iterator"
  },
  "ShouldRestart": {
    "Type": "Choice",
    "Choices": [
      {
        "Variable": "$.restart.executionCount",
```

```
        "NumericGreaterThan": 0,
        "Next": "Restart"
    }
  ],
  "Default": "Done"
},
"Restart": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Restart",
  "Next": "Done"
},
"Done": {
  "Type": "Pass",
  "End": true
}
}
```

- Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

For this tutorial, enter the name **ContinueAsNew**.

- (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

For this tutorial, keep all the default selections in **State machine settings**.

If you've [previously created an IAM role](#) with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

- In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

Note

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

9. Save the Amazon Resource Name (ARN) of this state machine in a text file. You'll need to provide the ARN while providing permission to the Lambda function to start a new Step Functions execution.

Step 4: Update the IAM Policy

To make sure your Lambda function has permissions to start a new Step Functions execution, attach an inline policy to the IAM role you use for your Restart Lambda function. For more information, see [Embedding Inline Policies](#) in the *IAM User Guide*.

Note

You can update the Resource line in the previous example to reference the ARN of your ContinueAsNew state machine. This restricts the policy so that it can only start an execution of that specific state machine.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": "arn:aws:states:us-east-2:123456789012stateMachine:ContinueAsNew"
    }
  ]
}
```


Step 5: Run the state machine

To start an execution, provide input that includes the ARN of the state machine and an executionCount for how many times it should start a new execution.

1. On the **ContinueAsNew** page, choose **Start execution**.

The **Start execution** dialog box is displayed.

2. In the **Start execution** dialog box, do the following:
 - a. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

 **Note**

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

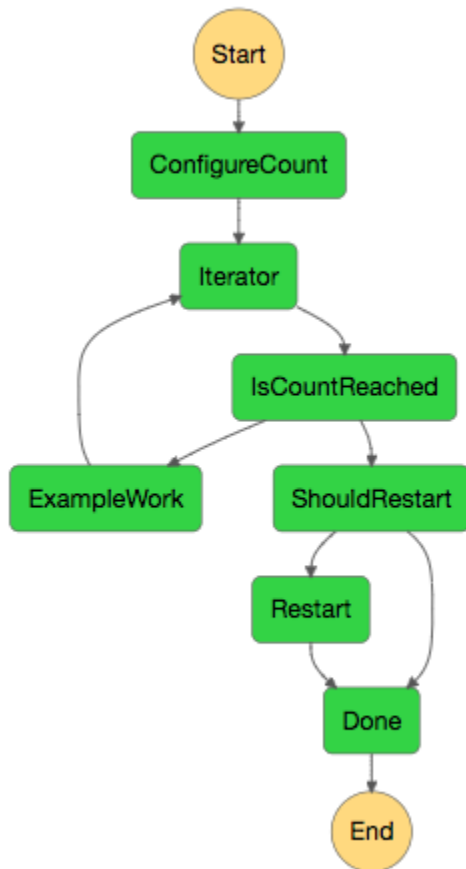
- b. In the **Input** box, enter the following JSON input to run your workflow.

```
{
  "restart": {
    "StateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:ContinueAsNew",
    "executionCount": 4
  }
}
```

- c. Update the `StateMachineArn` field with the ARN for your `ContinueAsNew` state machine.
 - d. Choose **Start execution**.
 - e. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

The **Graph view** displays the first of the four executions. Before it completes, it will pass through the `Restart` state and start a new execution.



As this execution completes, you can look at the next execution that's running. Select the **ContinueAsNew** link at the top to see the list of executions. You should see both the recently closed execution, and an ongoing execution that the `Restart` Lambda function started.

Succeeded

Running

When all the executions are complete, you should see four successful executions in the list. The first execution that was started displays the name you chose, and subsequent executions have a generated name.

8c4254e3-efa2-4b58-aa1a-fb85c8977516 arn:aws:states:us-east-1:██████████:execution:ContinueAsNew:8c4254e3-efa2-4b58-a...	Succeeded
0c9cfbd5-bf15-470b-b675-4d6ea0934afc arn:aws:states:us-east-1:██████████:execution:ContinueAsNew:0c9cfbd5-bf15-470b-b6...	Succeeded
67e10aef-693a-4abb-b7e6-2805a845ddd8 arn:aws:states:us-east-1:██████████:execution:ContinueAsNew:67e10aef-693a-4abb-b...	Succeeded
Test1 arn:aws:states:us-east-1:██████████:execution:ContinueAsNew:Test1	Succeeded

Deploying an Example Human Approval Project

This tutorial shows you how to deploy a human approval project that allows an AWS Step Functions execution to pause during a task, and wait for a user to respond to an email. The workflow progresses to the next state once the user has approved the task to proceed.

Deploying the AWS CloudFormation stack included in this tutorial will create all necessary resources, including:

- Amazon API Gateway resources
- An AWS Lambda functions
- An AWS Step Functions state machine
- An Amazon Simple Notification Service email topic
- Related AWS Identity and Access Management roles and permissions

Note

You will need to provide a valid email address that you have access to when you create the AWS CloudFormation stack.


For more information, see [Working with CloudFormation Templates](#) and the [AWS::StepFunctions::StateMachine](#) resource in the *AWS CloudFormation User Guide*.

Topics

- [Step 1: Create an AWS CloudFormation template](#)
- [Step 2: Create a stack](#)
- [Step 3: Approve the Amazon SNS subscription](#)
- [Step 4: Run the state machine](#)
- [AWS CloudFormation Template Source Code](#)

Step 1: Create an AWS CloudFormation template

1. Copy the example code from the [AWS CloudFormation Template Source Code](#) section.



```
    "URL": "http://www.example.com",
    "Subject": "Human approval."
  }
}
```

2. Paste the source of the AWS CloudFormation template into a file on your local machine.
For this example the file is called `human-approval.yaml`.

Step 2: Create a stack

1. Log into the [AWS CloudFormation console](#).
2. Choose **Create Stack**, and then choose **With new resources (standard)**.
3. On the **Create stack** page, do the following:
 - a. In the **Prerequisite - Prepare template** section, make sure **Template is ready** is selected.
 - b. In the **Specify template** section, choose **Upload a template file** and then choose **Choose file** to upload the `human-approval.yaml` file you created earlier that includes the [template source code](#).
4. Choose **Next**.
5. On the **Specify stack details** page, do the following:
 - a. For **Stack name**, enter a name for your stack.
 - b. Under **Parameters**, enter a valid email address. You'll use this email address to subscribe to the Amazon SNS topic.

- Choose **Next**, and then choose **Next** again.
- On the **Review** page, choose **I acknowledge that AWS CloudFormation might create IAM resources** and then choose **Create**.

AWS CloudFormation begins to create your stack and displays the **CREATE_IN_PROGRESS** status. When the process is complete, AWS CloudFormation displays the **CREATE_COMPLETE** status.

- (Optional) To display the resources in your stack, select the stack and choose the **Resources** tab.

▼ Resources

To view detailed drift information for specific resources, visit the [Drift Details](#) page.

Logical ID	Physical ID	Type	Drift Status	Status	Status Reason
ApiDeployment	zc8s70	AWS::ApiGateway::Depl...	NOT_CHECKED	CREATE_COMPL...	
ApiGatewayAccount	Human-ApiGa-TMBAQT11ZS4D	AWS::ApiGateway::Acc...	NOT_CHECKED	CREATE_COMPL...	
ApiGatewayCloud...	HumanApprovalExample-ApiGatewayCloudWatchLogsRole-1QZYONUOHAT2A	AWS::IAM::Role	NOT_CHECKED	CREATE_COMPL...	
ExecutionApi	dzn43w8x88	AWS::ApiGateway::Rest...	NOT_CHECKED	CREATE_COMPL...	
ExecutionApiStage	states	AWS::ApiGateway::Stage	NOT_CHECKED	CREATE_COMPL...	
ExecutionMethod	Human-Execu-LF06XD0FIW44	AWS::ApiGateway::Meth...	NOT_CHECKED	CREATE_COMPL...	
ExecutionResource	930an7	AWS::ApiGateway::Res...	NOT_CHECKED	CREATE_COMPL...	

Step 3: Approve the Amazon SNS subscription

Once the Amazon SNS topic is created, you will receive an email requesting that you confirm subscription.

- Open the email account you provided when you created the AWS CloudFormation stack.
- Open the message **AWS Notification - Subscription Confirmation** from **no-reply@sns.amazonaws.com**

The email will list the Amazon Resource Name for the Amazon SNS topic, and a confirmation link.

- Choose the **confirm subscription** link.



Simple Notification Service

Subscription confirmed!

You have subscribed [redacted]@amazon.com to the topic:
HumanApprovalExample-SNSHumanApprovalEmailTopic-AA1MNLKYAIM3.

Your subscription's id is:
arn:aws:sns:us-east-1:[redacted]:HumanApprovalExample-SNSHumanApprovalEmailTopic-AA1MNLKYAIM3:c358fd09-ce61-4cc7-b67f-52ccf3ee4e4f

If it was not your intention to subscribe, [click here to unsubscribe](#).

Step 4: Run the state machine

1. On the **HumanApprovalLambdaStateMachine** page, choose **Start execution**.

The **Start execution** dialog box is displayed.

2. In the **Start execution** dialog box, do the following:
 - a. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

- b. In the **Input** box, enter the following JSON input to run your workflow.

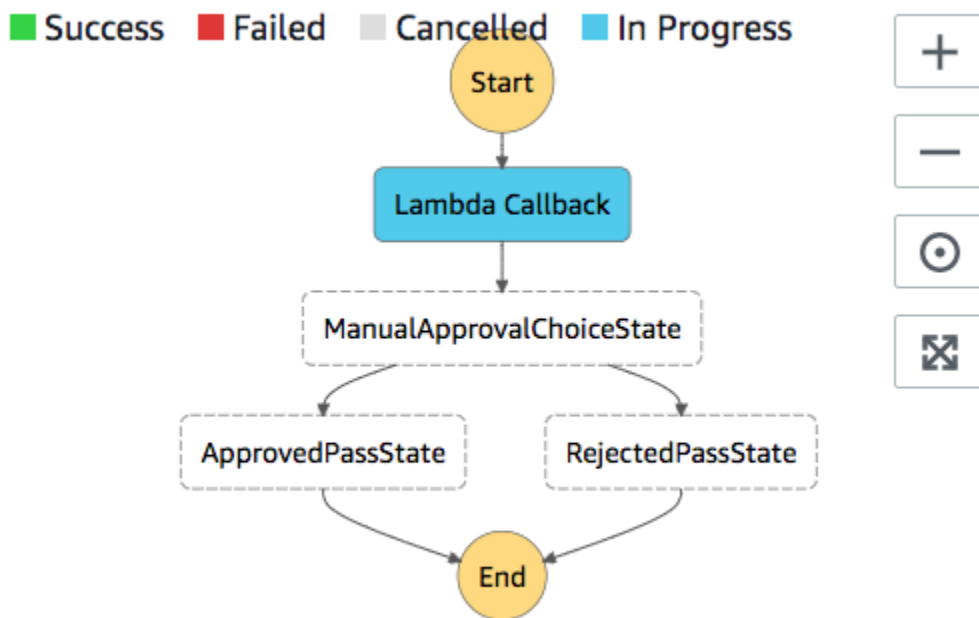
```
{
  "Comment": "Testing the human approval tutorial."
}
```

- c. Choose **Start execution**.

The **ApprovalTest** state machine execution starts, and pauses at the **Lambda Callback** task.

- d. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

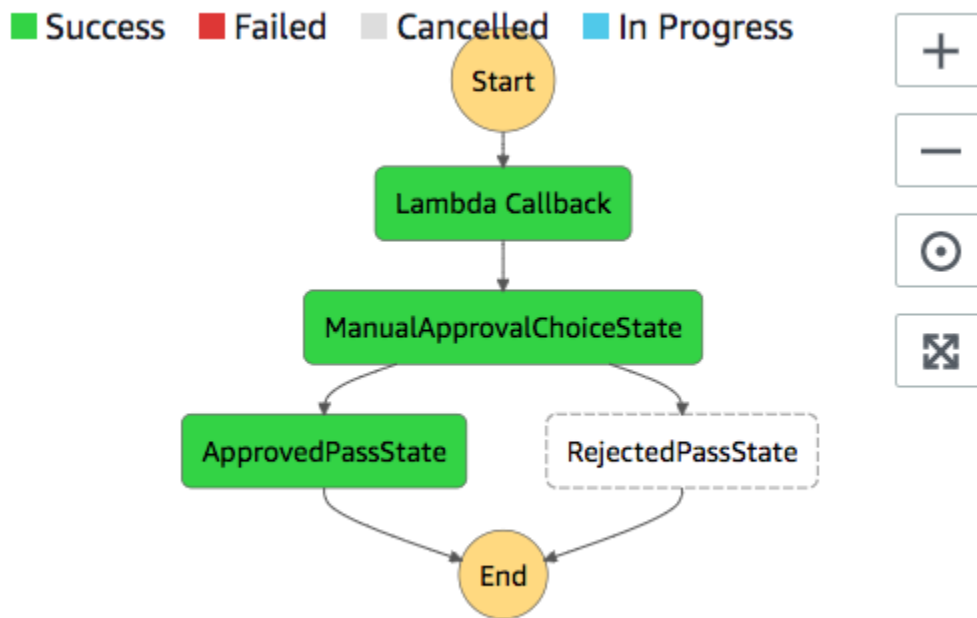


3. In the email account you used for the Amazon SNS topic earlier, open the message with the subject **Required approval from AWS Step Functions**.

The message includes separate URLs for **Approve** and **Reject**.

4. Choose the **Approve** URL.

The workflow continues based on your choice.



AWS CloudFormation Template Source Code

Use this AWS CloudFormation template to deploy an example of a human approval process workflow.

```

AWSTemplateFormatVersion: "2010-09-09"
Description: "AWS Step Functions Human based task example. It sends an email with an HTTP URL for approval."
Parameters:
  Email:
    Type: String
    AllowedPattern: "^[a-zA-Z0-9_+-.]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-]+\.$"
    ConstraintDescription: Must be a valid email address.
Resources:
  # Begin API Gateway Resources
  ExecutionApi:
    Type: "AWS::ApiGateway::RestApi"
    Properties:
      Name: "Human approval endpoint"
      Description: "HTTP Endpoint backed by API Gateway and Lambda"
      FailOnWarnings: true

  ExecutionResource:

```



```

Type: 'AWS::ApiGateway::Resource'
Properties:
  RestApiId: !Ref ExecutionApi
  ParentId: !GetAtt "ExecutionApi.RootResourceId"
  PathPart: execution

ExecutionMethod:
Type: "AWS::ApiGateway::Method"
Properties:
  AuthorizationType: NONE
  HttpMethod: GET
  Integration:
    Type: AWS
    IntegrationHttpMethod: POST
    Uri: !Sub "arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/
${LambdaApprovalFunction.Arn}/invocations"
    IntegrationResponses:
      - StatusCode: 302
        ResponseParameters:
          method.response.header.Location:
"integration.response.body.headers.Location"
    RequestTemplates:
      application/json: |
        {
          "body" : $input.json('$'),
          "headers": {
            #foreach($header in $input.params().header.keySet())
              "$header":
"$util.escapeJavaScript($input.params().header.get($header))"
            #if($foreach.hasNext),#end

            #end
          },
          "method": "$context.httpMethod",
          "params": {
            #foreach($param in $input.params().path.keySet())
              "$param": "$util.escapeJavaScript($input.params().path.get($param))"
            #if($foreach.hasNext),#end

            #end
          },
          "query": {
            #foreach($queryParam in $input.params().querystring.keySet())

```

```

        "$queryParam":
"$util.escapeJavaScript($input.params().querystring.get($queryParam))"
#if($foreach.hasNext),#end

        #end
    }
}

ResourceId: !Ref ExecutionResource
RestApiId: !Ref ExecutionApi
MethodResponses:
  - StatusCode: 302
    ResponseParameters:
      method.response.header.Location: true

ApiGatewayAccount:
  Type: 'AWS::ApiGateway::Account'
  Properties:
    CloudWatchRoleArn: !GetAtt "ApiGatewayCloudWatchLogsRole.Arn"

ApiGatewayCloudWatchLogsRole:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: Allow
          Principal:
            Service:
              - apigateway.amazonaws.com
          Action:
            - 'sts:AssumeRole'
    Policies:
      - PolicyName: ApiGatewayLogsPolicy
        PolicyDocument:
          Version: 2012-10-17
          Statement:
            - Effect: Allow
              Action:
                - "logs:*"
              Resource: !Sub "arn:${AWS::Partition}:logs:*:*:*"

ExecutionApiStage:
  DependsOn:
    - ApiGatewayAccount

```

```

Type: 'AWS::ApiGateway::Stage'
Properties:
  DeploymentId: !Ref ApiDeployment
  MethodSettings:
    - DataTraceEnabled: true
      HttpMethod: '*'
      LoggingLevel: INFO
      ResourcePath: /*
  RestApiId: !Ref ExecutionApi
  StageName: states

ApiDeployment:
  Type: "AWS::ApiGateway::Deployment"
  DependsOn:
    - ExecutionMethod
  Properties:
    RestApiId: !Ref ExecutionApi
    StageName: DummyStage
# End API Gateway Resources

# Begin
# Lambda that will be invoked by API Gateway
LambdaApprovalFunction:
  Type: 'AWS::Lambda::Function'
  Properties:
    Code:
      ZipFile:
        Fn::Sub: |
          const { SFN: StepFunctions } = require("@aws-sdk/client-sfn");
          var redirectToStepFunctions = function(lambdaArn, statemachineName,
executionName, callback) {
            const lambdaArnTokens = lambdaArn.split(":");
            const partition = lambdaArnTokens[1];
            const region = lambdaArnTokens[3];
            const accountId = lambdaArnTokens[4];

            console.log("partition=" + partition);
            console.log("region=" + region);
            console.log("accountId=" + accountId);

            const executionArn = "arn:" + partition + ":states:" + region + ":" +
accountId + ":execution:" + statemachineName + ":" + executionName;
            console.log("executionArn=" + executionArn);

```

```
    const url = "https://console.aws.amazon.com/states/home?region=" + region
+   + "#/executions/details/" + executionArn;
    callback(null, {
      statusCode: 302,
      headers: {
        Location: url
      }
    });
  });
};

exports.handler = (event, context, callback) => {
  console.log('Event= ' + JSON.stringify(event));
  const action = event.query.action;
  const taskToken = event.query.taskToken;
  const statemachineName = event.query.sm;
  const executionName = event.query.ex;

  const stepfunctions = new StepFunctions();

  var message = "";

  if (action === "approve") {
    message = { "Status": "Approved! Task approved by ${Email}" };
  } else if (action === "reject") {
    message = { "Status": "Rejected! Task rejected by ${Email}" };
  } else {
    console.error("Unrecognized action. Expected: approve, reject.");
    callback({"Status": "Failed to process the request. Unrecognized
Action."});
  }

  stepfunctions.sendTaskSuccess({
    output: JSON.stringify(message),
    taskToken: event.query.taskToken
  })
  .then(function(data) {
    redirectToStepFunctions(context.invokedFunctionArn, statemachineName,
executionName, callback);
  }).catch(function(err) {
    console.error(err, err.stack);
    callback(err);
  });
}
```

Description: Lambda function that callback to AWS Step Functions

```
    FunctionName: LambdaApprovalFunction
    Handler: index.handler
    Role: !GetAtt "LambdaApiGatewayIAMRole.Arn"
    Runtime: nodejs18.x

LambdaApiGatewayInvoke:
  Type: "AWS::Lambda::Permission"
  Properties:
    Action: "lambda:InvokeFunction"
    FunctionName: !GetAtt "LambdaApprovalFunction.Arn"
    Principal: "apigateway.amazonaws.com"
    SourceArn: !Sub "arn:aws:execute-api:${AWS::Region}:${AWS::AccountId}:
${ExecutionApi}/*"

LambdaApiGatewayIAMRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Action:
            - "sts:AssumeRole"
          Effect: "Allow"
          Principal:
            Service:
              - "lambda.amazonaws.com"
    Policies:
      - PolicyName: CloudWatchLogsPolicy
        PolicyDocument:
          Statement:
            - Effect: Allow
              Action:
                - "logs:*"
              Resource: !Sub "arn:${AWS::Partition}:logs:*:*:*"
      - PolicyName: StepFunctionsPolicy
        PolicyDocument:
          Statement:
            - Effect: Allow
              Action:
                - "states:SendTaskFailure"
                - "states:SendTaskSuccess"
              Resource: "*"
# End Lambda that will be invoked by API Gateway
```

```

# Begin state machine that publishes to Lambda and sends an email with the link for
approval
HumanApprovalLambdaStateMachine:
  Type: AWS::StepFunctions::StateMachine
  Properties:
    RoleArn: !GetAtt LambdaStateMachineExecutionRole.Arn
    DefinitionString:
      Fn::Sub: |
        {
          "StartAt": "Lambda Callback",
          "TimeoutSeconds": 3600,
          "States": {
            "Lambda Callback": {
              "Type": "Task",
              "Resource": "arn:
${AWS::Partition}:states:::lambda:invoke.waitForTaskToken",
              "Parameters": {
                "FunctionName": "${LambdaHumanApprovalSendEmailFunction.Arn}",
                "Payload": {
                  "ExecutionContext.$": "$$",
                  "APIGatewayEndpoint": "https://${ExecutionApi}.execute-api.
${AWS::Region}.amazonaws.com/states"
                }
              },
              "Next": "ManualApprovalChoiceState"
            },
            "ManualApprovalChoiceState": {
              "Type": "Choice",
              "Choices": [
                {
                  "Variable": "$.Status",
                  "StringEquals": "Approved! Task approved by ${Email}",
                  "Next": "ApprovedPassState"
                },
                {
                  "Variable": "$.Status",
                  "StringEquals": "Rejected! Task rejected by ${Email}",
                  "Next": "RejectedPassState"
                }
              ]
            },
            "ApprovedPassState": {
              "Type": "Pass",
              "End": true
            }
          }
        }

```

```

    },
    "RejectedPassState": {
      "Type": "Pass",
      "End": true
    }
  }
}

```

SNSHumanApprovalEmailTopic:

```

Type: AWS::SNS::Topic
Properties:
  Subscription:
    -
      Endpoint: !Sub ${Email}
      Protocol: email

```

LambdaHumanApprovalSendEmailFunction:

```

Type: "AWS::Lambda::Function"
Properties:
  Handler: "index.lambda_handler"
  Role: !GetAtt LambdaSendEmailExecutionRole.Arn
  Runtime: "nodejs18.x"
  Timeout: "25"
  Code:
    ZipFile:
      Fn::Sub: |
        console.log('Loading function');
        const { SNS } = require("@aws-sdk/client-sns");
        exports.lambda_handler = (event, context, callback) => {
          console.log('event= ' + JSON.stringify(event));
          console.log('context= ' + JSON.stringify(context));

          const executionContext = event.ExecutionContext;
          console.log('executionContext= ' + executionContext);

          const executionName = executionContext.Execution.Name;
          console.log('executionName= ' + executionName);

          const statemachineName = executionContext.StateMachine.Name;
          console.log('statemachineName= ' + statemachineName);

          const taskToken = executionContext.Task.Token;
          console.log('taskToken= ' + taskToken);

```

```
    const apigwEndpoint = event.APIGatewayEndpoint;
    console.log('apigwEndpoint = ' + apigwEndpoint)

    const approveEndpoint = apigwEndpoint + "/execution?
action=approve&ex=" + executionName + "&sm=" + statemachineName + "&taskToken=" +
    encodeURIComponent(taskToken);
    console.log('approveEndpoint= ' + approveEndpoint);

    const rejectEndpoint = apigwEndpoint + "/execution?
action=reject&ex=" + executionName + "&sm=" + statemachineName + "&taskToken=" +
    encodeURIComponent(taskToken);
    console.log('rejectEndpoint= ' + rejectEndpoint);

    const emailSnsTopic = "${SNSHumanApprovalEmailTopic}";
    console.log('emailSnsTopic= ' + emailSnsTopic);

    var emailMessage = 'Welcome! \n\n';
    emailMessage += 'This is an email requiring an approval for a step
functions execution. \n\n'
    emailMessage += 'Please check the following information and click
"Approve" link if you want to approve. \n\n'
    emailMessage += 'Execution Name -> ' + executionName + '\n\n'
    emailMessage += 'Approve ' + approveEndpoint + '\n\n'
    emailMessage += 'Reject ' + rejectEndpoint + '\n\n'
    emailMessage += 'Thanks for using Step functions!'

    const sns = new SNS();
    var params = {
        Message: emailMessage,
        Subject: "Required approval from AWS Step Functions",
        TopicArn: emailSnsTopic
    };

    sns.publish(params)
        .then(function(data) {
            console.log("MessageID is " + data.MessageId);
            callback(null);
        }).catch(
        function(err) {
            console.error(err, err.stack);
            callback(err);
        });
}
```



```
LambdaStateMachineExecutionRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: Allow
          Principal:
            Service: states.amazonaws.com
          Action: "sts:AssumeRole"
    Policies:
      - PolicyName: InvokeCallbackLambda
        PolicyDocument:
          Statement:
            - Effect: Allow
              Action:
                - "lambda:InvokeFunction"
              Resource:
                - !Sub "${LambdaHumanApprovalSendEmailFunction.Arn}"

LambdaSendEmailExecutionRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
          Action: "sts:AssumeRole"
    Policies:
      - PolicyName: CloudWatchLogsPolicy
        PolicyDocument:
          Statement:
            - Effect: Allow
              Action:
                - "logs:CreateLogGroup"
                - "logs:CreateLogStream"
                - "logs:PutLogEvents"
              Resource: !Sub "arn:${AWS::Partition}:logs:*:*:*"
      - PolicyName: SNSSendEmailPolicy
        PolicyDocument:
          Statement:
            - Effect: Allow
```

```
    Action:
      - "SNS:Publish"
    Resource:
      - !Sub "${SNSHumanApprovalEmailTopic}"

# End state machine that publishes to Lambda and sends an email with the link for
approval
Outputs:
  ApiGatewayInvokeURL:
    Value: !Sub "https://${ExecutionApi}.execute-api.${AWS::Region}.amazonaws.com/
states"
  StateMachineHumanApprovalArn:
    Value: !Ref HumanApprovalLambdaStateMachine
```

View X-Ray traces in Step Functions

In this tutorial, you will learn how to use X-Ray to trace errors that occur when running a state machine. You can use [AWS X-Ray](#) to visualize the components of your state machine, identify performance bottlenecks, and troubleshoot requests that resulted in an error. In this tutorial, you will create several Lambda functions that randomly produce errors, which you can then trace and analyze using X-Ray.

The [Creating a Step Functions state machine that uses Lambda](#) tutorial walks you through creating a state machine that calls a Lambda function. If you have completed that tutorial, skip to [Step 2](#) and use the AWS Identity and Access Management (IAM) role that you previously created.

Topics

- [Step 1: Create an IAM role for Lambda](#)
- [Step 2: Create a Lambda function](#)
- [Step 3: Create two more Lambda functions](#)
- [Step 4: Create a state machine](#)
- [Step 5: Run the state machine](#)

Step 1: Create an IAM role for Lambda

Both AWS Lambda and AWS Step Functions can execute code and access AWS resources (for example, data stored in Amazon S3 buckets). To maintain security, you must grant Lambda and Step Functions access to these resources.

Lambda requires you to assign an AWS Identity and Access Management (IAM) role when you create a Lambda function, in the same way Step Functions requires you to assign an IAM role when you create a state machine.

You use the IAM console to create a service-linked role.

To create a role (console)

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane of the IAM console, choose **Roles**. Then choose **Create role**.
3. Choose the **AWS Service** role type, and then choose **Lambda**.
4. Choose the **Lambda** use case. Use cases are defined by the service to include the trust policy required by the service. Then choose **Next: Permissions**.
5. Choose one or more permissions policies to attach to the role (for example, `AWSLambdaBasicExecutionRole`). See [AWS Lambda Permissions Model](#).

Select the box next to the policy that assigns the permissions that you want the role to have, and then choose **Next: Review**.

6. Enter a **Role name**.
7. (Optional) For **Role description**, edit the description for the new service-linked role.
8. Review the role, and then choose **Create role**.

Step 2: Create a Lambda function

Your Lambda function will randomly throw errors or time out, producing example data to view in X-Ray.

Important

Ensure that your Lambda function is under the same AWS account and AWS Region as your state machine.

1. Open the [Lambda console](#) and choose **Create function**.
2. In the **Create function** section, choose **Author from scratch**.
3. In the **Basic information** section, configure your Lambda function:

- a. For **Function name**, enter `TestFunction1`.
- b. For **Runtime**, choose **Node.js 18.x**.
- c. For **Role**, select **Choose an existing role**.
- d. For **Existing role**, select [the Lambda role that you created earlier](#).

 **Note**

If the IAM role that you created doesn't appear in the list, the role might still need a few minutes to propagate to Lambda.

- e. Choose **Create function**.

When your Lambda function is created, note its Amazon Resource Name (ARN) in the upper-right corner of the page. For example:

```
arn:aws:lambda:us-east-1:123456789012:function:TestFunction1
```

4. Copy the following code for the Lambda function into the **Function code** section of the ***TestFunction1*** page.

```
function getRandomSeconds(max) {
    return Math.floor(Math.random() * Math.floor(max)) * 1000;
}
function sleep(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}
export const handler = async (event) => {
    if(getRandomSeconds(4) === 0) {
        throw new Error("Something went wrong!");
    }
    let wait_time = getRandomSeconds(5);
    await sleep(wait_time);
    return { 'response': true }
};
```

This code creates randomly timed failures, which will be used to generate example errors in your state machine that can be viewed and analyzed using X-Ray traces.

5. Choose **Save**.

Step 3: Create two more Lambda functions

Create two more Lambda functions.

1. Repeat Step 2 to create two more Lambda functions. For the next function, in **Function name**, enter `TestFunction2`. For the last function, in **Function name**, enter `TestFunction3`.
2. In the Lambda console, check that you now have three Lambda functions, `TestFunction1`, `TestFunction2`, and `TestFunction3`.

Step 4: Create a state machine

In this step, you'll use the [Step Functions console](#) to create a state machine with three Task states. Each Task state will reference one of your three Lambda functions.

1. Open the [Step Functions console](#) and choose **Create state machine**.

Important

Make sure that your state machine is under the same AWS account and Region as the Lambda functions you created earlier in [Step 2](#) and [Step 3](#).

2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).
4. For this tutorial, you'll write the [Amazon States Language](#) (ASL) definition of your state machine in the [Code editor](#). To do this, choose **Code**.
5. Remove the existing boilerplate code and paste the following code. In the Task state definition, remember to replace the example ARNs with the ARNs of the Lambda functions you created.

```
{
  "StartAt": "CallTestFunction1",
  "States": {
    "CallTestFunction1": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:test-function1",
      "Catch": [
        {
          "ErrorEquals": [
            "States.TaskFailed"
          ]
        }
      ]
    }
  }
}
```

```
    ],
    "Next": "AfterTaskFailed"
  }
],
"Next": "CallTestFunction2"
},
"CallTestFunction2": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:test-function2",
  "Catch": [
    {
      "ErrorEquals": [
        "States.TaskFailed"
      ],
      "Next": "AfterTaskFailed"
    }
  ],
  "Next": "CallTestFunction3"
},
"CallTestFunction3": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:test-function3",
  "TimeoutSeconds": 5,
  "Catch": [
    {
      "ErrorEquals": [
        "States.Timeout"
      ],
      "Next": "AfterTimeout"
    },
    {
      "ErrorEquals": [
        "States.TaskFailed"
      ],
      "Next": "AfterTaskFailed"
    }
  ],
  "Next": "Succeed"
},
"Succeed": {
  "Type": "Succeed"
},
"AfterTimeout": {
  "Type": "Fail"
}
```

```
    },
    "AfterTaskFailed": {
      "Type": "Fail"
    }
  }
}
```

This is a description of your state machine using the Amazon States Language. It defines three Task states named `CallTestFunction1`, `CallTestFunction2` and `CallTestFunction3`. Each calls one of your three Lambda functions. For more information, see [State Machine Structure](#).

6. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

For this tutorial, enter the name **TraceFunctions**.

7. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

For this tutorial, under **Additional configuration**, choose **Enable X-Ray tracing**. Keep all the other default selections in **State machine settings**.

If you've [previously created an IAM role](#) with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

8. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

Note

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

Step 5: Run the state machine

State machine executions are instances where you run your workflow to perform tasks.

1. On the *TraceFunctions* page, choose **Start execution**.

The **New execution** page is displayed.

2. In the **Start execution** dialog box, do the following:
 - a. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

- b. Choose **Start execution**.
 - c. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Run several (at least three) executions.

3. After the executions have finished, follow the **X-Ray trace map** link. You can view the trace while an execution is still running, but you may want to see the execution results before viewing the X-Ray trace map.

Details | Execution Input | Execution Output | Definition

Execution Status
❌ Failed

Execution ARN
arn:aws:states:sa-east-1:123456789012:execution:TraceFunctions:1-23456789-0134567890123456789012

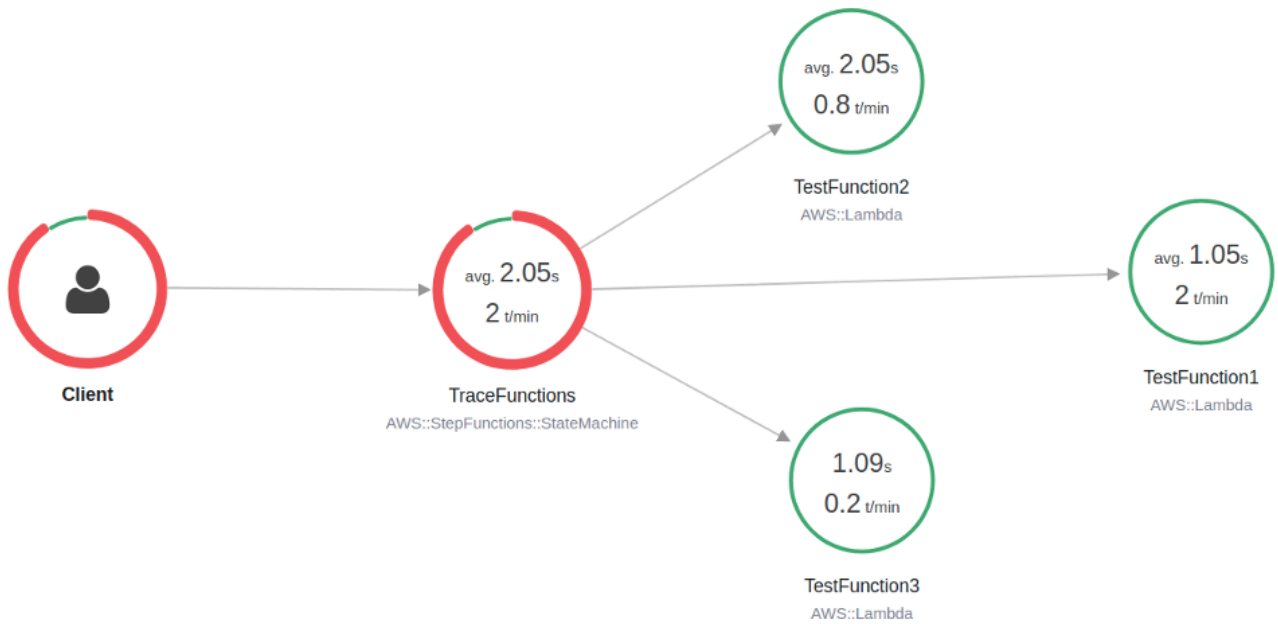
X-Ray trace map [Learn more](#)

1-23456789-0134567890123456789012

4. View the service map to identify where errors are occurring, connections with high latency, or traces for requests that were unsuccessful. In this example, you can see how much traffic each function is receiving. `TestFunction2` was called more often than `TestFunction3`, and `TestFunction1` was called more than twice as often as `TestFunction2`.

The service map indicates the health of each node by coloring it based on the ratio of successful calls to errors and faults:

- **Green** for successful calls
- **Red** for server faults (500 series errors)
- **Yellow** for client errors (400 series errors)
- **Purple** for throttling errors (429 Too Many Requests)



You can also choose a service node to view requests for that node, or an edge between two nodes to view requests that traveled that connection.

- View the X-Ray trace map to see what has happened for each execution. The Timeline view shows a hierarchy of segments and subsegments. The first entry in the list is the segment, which represents all data recorded by the service for a single request. Below the segment are subsegments. This example shows subsegments recorded by the Lambda functions.

Name	Res.	Duration	Status	0.0ms	200ms	400ms	600ms	800ms	1.0s	1.2s	1.4s	1.6s	1.8s	2.0s	2.2s	2.4s
▼ TraceFunctions AWS::StepFunctions::StateMachine																
TraceFunctions	-	2.3 sec	!	[Timeline bar]												
CallTestFunction1	-	2.1 sec	✓	[Timeline bar]												
Lambda	200	2.1 sec	✓	[Timeline bar] Invoke: TestFunction1												
CallTestFunction2	-	102 ms	!	[Timeline bar]												
Lambda	200	62.0 ms	!	[Timeline bar] Invoke: TestFunction2												
AfterTaskFailed	-	0.0 ms	!	[Timeline bar]												
▶ Lambda AWS::Lambda (Client Response)																

For more information on understanding X-Ray traces and using X-Ray with Step Functions, see the [AWS X-Ray and Step Functions](#)

Gather Amazon S3 bucket info using AWS SDK service integrations

This tutorial shows you how to perform an [AWS SDK integration](#) with Amazon Simple Storage Service. The state machine you create in this tutorial gathers information about your Amazon S3 buckets, then list your buckets along with version information for each bucket in the current region.

Topics

- [Step 1: Create the state machine](#)
- [Step 2: Add the necessary IAM role permissions](#)
- [Step 3: Run a Standard state machine execution](#)
- [Step 4: Run an Express state machine execution](#)

Step 1: Create the state machine

Using the Step Functions console, you'll create a state machine that includes a Task state to list all the Amazon S3 buckets in the current account and region. Then, you'll add another Task state that invokes the [HeadBucket](#) API to verify if the returned bucket is accessible in the current region. If the bucket isn't accessible, the HeadBucket API call returns the `S3.S3Exception` error. You'll include a Catch block to catch this exception and a Pass state as the fallback state.

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. In the **Choose a template** dialog box, select **Blank**.
3. Choose **Select**. This opens Workflow Studio in [Design mode](#).
4. For this tutorial, you'll write the [Amazon States Language](#) (ASL) definition of your state machine in the [Code editor](#). To do this, choose **Code**.
5. Remove the existing boilerplate code and paste the following state machine definition.

```
{
  "Comment": "A description of my state machine",
  "StartAt": "ListBuckets",
  "States": {
    "ListBuckets": {
      "Type": "Task",
      "Parameters": {}
    }
  }
}
```

```
    "Resource": "arn:aws:states:::aws-sdk:s3:listBuckets",
    "Next": "Map"
  },
  "Map": {
    "Type": "Map",
    "ItemsPath": "$.Buckets",
    "ItemProcessor": {
      "ProcessorConfig": {
        "Mode": "INLINE"
      },
      "StartAt": "HeadBucket",
      "States": {
        "HeadBucket": {
          "Type": "Task",
          "ResultPath": null,
          "Parameters": {
            "Bucket.$": "$.Name"
          },
          "Resource": "arn:aws:states:::aws-sdk:s3:headBucket",
          "Catch": [
            {
              "ErrorEquals": [
                "S3.S3Exception"
              ],
              "ResultPath": null,
              "Next": "Pass"
            }
          ],
          "Next": "GetBucketVersioning"
        },
        "GetBucketVersioning": {
          "Type": "Task",
          "End": true,
          "Parameters": {
            "Bucket.$": "$.Name"
          },
          "ResultPath": "$.BucketVersioningInfo",
          "Resource": "arn:aws:states:::aws-sdk:s3:getBucketVersioning"
        },
        "Pass": {
          "Type": "Pass",
          "End": true,
          "Result": {
            "Status": "Unknown"
          }
        }
      }
    }
  }
}
```

```
        },
        "ResultPath": "$.BucketVersioningInfo"
      }
    },
    "End": true
  }
}
```

6. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

For this tutorial, enter the name **Gather-S3-Bucket-Info-Standard**.

7. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

Keep all the default selections in **State machine settings**.

If you've [previously created an IAM role](#) with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

8. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

Note

If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

In [Step 2](#), you'll add the missing permissions to the state machine role.

Step 2: Add the necessary IAM role permissions

To gather information about the Amazon S3 buckets in your current region, you must provide your state machine the necessary permissions to access the Amazon S3 buckets.

1. On the state machine page, choose **IAM role ARN** to open the **Roles** page for the state machine role.
2. Choose **Add permissions** and then choose **Create inline policy**.
3. Choose the **JSON** tab, and then paste the following permissions into the JSON editor.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "s3:ListAllMyBuckets",
        "s3:ListBucket",
        "s3:GetBucketVersioning"
      ],
      "Resource": "*"
    }
  ]
}
```

4. Choose **Review policy**.
5. Under **Review policy**, for the policy **Name**, enter **s3-bucket-permissions**.
6. Choose **Create policy**.

Step 3: Run a Standard state machine execution

1. On the **Gather-S3-Bucket-Info-Standard** page, choose **Start execution**.
2. In the **Start execution** dialog box, do the following:
 - a. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

- b. Choose **Start execution**.
- c. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Step 4: Run an Express state machine execution

1. Create an Express state machine using the state machine definition provided in [Step 1](#). Make sure that you also include the necessary IAM role permissions as explained in [Step 2](#).

Tip

To distinguish from the Standard machine you created earlier, name the Express state machine as **Gather-S3-Bucket-Info-Express**.

2. On the **Gather-S3-Bucket-Info-Standard** page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 - a. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names

don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

- b. Choose **Start execution**.
- c. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Developer tools

The following resources contain additional information about building serverless workflows and working with state machines:

- [AWS CDK](#)
- [AWS Toolkit for VS Code](#)

The topics below contain information that teach you how to create, test, and debug state machines.

Topics

- [Development options](#)
- [AWS Step Functions and AWS SAM](#)
- [Using Workflow Studio in Application Composer](#)
- [Creating a Lambda state machine for Step Functions using AWS CloudFormation](#)
- [Creating a Lambda state machine for Step Functions using AWS CDK](#)
- [Creating an API Gateway REST API with Synchronous Express State Machine Using the AWS CDK](#)
- [AWS Step Functions Data Science SDK for Python](#)
- [Deploying state machines using Terraform](#)

Development options

You can implement your AWS Step Functions state machines in several ways, such as using the console, the SDKs, or a local version of Step Functions for testing and development.

Topics

- [Step Functions console](#)
- [AWS SDKs](#)
- [Standard and Express workflows](#)
- [HTTPS service API](#)
- [Development environments](#)

- [Endpoints](#)
- [AWS CLI](#)
- [Step Functions Local](#)
- [AWS Toolkit for Visual Studio Code](#)
- [AWS Serverless Application Model and Step Functions](#)
- [Terraform and Step Functions](#)
- [Definition format support](#)

Step Functions console

You can define a state machine using the [Step Functions console](#). You can write complex state machines in the cloud without using a local development environment by using AWS Lambda to supply code for your tasks. Once written, you can then use the Step Functions console to define your state machine using the Amazon States Language.

The [Creating a Lambda State Machine](#) tutorial uses this technique to create a simple state machine, execute it, and view its results.

Data flow simulator

You can design, implement and debug workflows in the Step Functions console. You can also control the flow of data in your workflows by using JsonPath input and output processing. Use the [data flow simulator in the Step Functions console](#) to learn how information flows from state to state, and to understand how to filter and manipulate data. This tool simulates each of the [fields](#) that Step Functions uses to process data, such as InputPath, Parameters, ResultSelector, OutputPath, and ResultPath.

For more information, see [Data flow simulator](#).

AWS SDKs

Step Functions is supported by the AWS SDKs for Java, .NET, Ruby, PHP, Python (Boto 3), JavaScript, Go, and C++. These SDKs provide a convenient way to use the Step Functions HTTPS API actions in multiple programming languages.

You can develop state machines, activities, or state machine starters using the API actions exposed by these SDK libraries. You can also access visibility operations using these libraries to develop your own Step Functions monitoring and reporting tools.

To use Step Functions with other AWS services, see the reference documentation for the current AWS SDKs and [Tools for Amazon Web Services](#).

Note

Step Functions only supports HTTPS endpoints.

Standard and Express workflows

When you create a new state machine, you must select a Type of either **Standard** or **Express**. In both cases, you define your state machine using the Amazon States Language. Your state machine executions will behave differently, depending on which **Type** you select. The **Type** you choose can't be changed after your state machine is created.

See [Logging using CloudWatch Logs](#) for more information.

HTTPS service API

Step Functions provides service operations that are accessible through HTTPS requests. You can use these operations to communicate directly with Step Functions and to develop your own libraries in any language that can communicate with Step Functions through HTTPS.

You can develop state machines, workers, or state machine starters using the service API actions. You can also access visibility operations through the API actions to develop your own monitoring and reporting tools.

For detailed information about API actions, see the [AWS Step Functions API Reference](#).

Development environments

You must set up a development environment that is compatible with the programming language that you plan to use.

For example, to develop for Step Functions using Java, you must install a Java development environment, such as the AWS SDK for Java, on each of your development workstations. If you use Eclipse IDE for Java Developers, you should also install the AWS Toolkit for Eclipse. This Eclipse plugin adds features that are useful for developing on AWS.

If your programming language requires a runtime environment, you must set up the environment on each computer where these processes will run.

Endpoints

To reduce latency and store data in a location that meets your requirements, Step Functions provides endpoints in different AWS Regions.

Each endpoint in Step Functions is completely independent. A state machine or activity exists only within the Region where it was created. Any state machines and activities that you create in one Region don't share any data or attributes with those created in another Region. For example, you can register a state machine named STATES-Flows-1 in two different Regions. The STATES-Flows-1 state machine in one region won't share data or attributes with the STATES-Flow-1 state machine in the other region.

For a list of Step Functions endpoints, see [AWS Step Functions Regions and Endpoints](#) in the *AWS General Reference*.

AWS CLI

You can access many Step Functions features from the AWS Command Line Interface (AWS CLI). The AWS CLI is an alternative to using the [Step Functions console](#) or, in some cases, to programming using the Step Functions API actions. For example, you can use the AWS CLI to create a state machine and then list your existing state machines.

You can use Step Functions commands in the AWS CLI to start and manage executions, poll for activities, record task heartbeats, and more. For a complete list of Step Functions commands, descriptions of the available arguments, and examples showing their use, see the *AWS CLI Command Reference*.

AWS CLI commands follow the Amazon States Language closely, so you can use the AWS CLI to learn about the Step Functions API actions. You can also use your existing API knowledge to prototype code or perform Step Functions actions from the command line.

Step Functions Local

For testing and development purposes, you can install and run Step Functions on your local machine. With Step Functions Local, you can start an execution on any machine.

The local version of Step Functions can invoke AWS Lambda functions, both in AWS and when running locally. You can also coordinate other [supported AWS services](#). For more information, see [Testing state machines locally](#).

Note

Step Functions Local uses dummy accounts to work.

AWS Toolkit for Visual Studio Code

You can use VS Code to interact with remote state machines and develop state machines locally. You can create or update state machines, list existing state machines, and execute or download a state machine. VS Code also lets you create new state machines from templates, see a visualization of your state machine, and provides code snippets, code completion, and code validation.

For more information, see [the AWS Toolkit for Visual Studio Code User Guide](#)

AWS Serverless Application Model and Step Functions

Step Functions is integrated with the AWS Serverless Application Model, which lets you integrate workflows with Lambda functions, APIs and events to create serverless applications.

You can also use the AWS SAM CLI in conjunction with the AWS Toolkit for Visual Studio Code as part of an integrated experience.

For more information, see [AWS Step Functions and AWS SAM](#).

Terraform and Step Functions

[Terraform](#) by HashiCorp is a framework for building applications using infrastructure as code (IaC). With Terraform, you can create state machines and use features, such as previewing infrastructure deployments and creating reusable templates. Terraform templates help you maintain and reuse the code by breaking it down into smaller chunks.

For more information, see [Deploying state machines using Terraform](#).

Definition format support

Step Functions offers a variety of tools that lets you provide your state machine definitions in different formats. An Amazon States Language (ASL) definition that specifies the details of your state machine can be provided as either a string, or as a serialized object using JSON or YAML.

Note

YAML allows single line comments. Any YAML comments provided in the state machine definition portion of a template will not be carried forward into the created resource's definition. Instead, you can use the `Comment` property within the state machine definition. For more information, see the [State machine structure](#) page.

The following table shows which tools support ASL-based definitions.

Definition format support by tool

	JSON	YAML	Stringified Amazon States Language		
Step Functions Console	✓				
HTTPS Service API			✓		
AWS CLI			✓		
Step Functions Local			✓		
AWS Toolkit for Visual Studio Code	✓	✓			
AWS SAM	✓	✓			
AWS CloudFormation	✓	✓	✓		

Note

AWS CloudFormation and AWS SAM also allow you to upload your state machine definitions to Amazon S3 in JSON or YAML format, and to provide the definition's Amazon S3 location in the template. This can improve the readability of your templates when your state machine definition is complex. For more information see the [AWS::StepFunctions::StateMachine S3Location](#) page.

The following example AWS CloudFormation templates show how you can provide the same state machine definition using different input formats.

JSON with Definition

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "AWS Step Functions sample template.",
  "Resources": {
    "MyStateMachine": {
      "Type": "AWS::StepFunctions::StateMachine",
      "Properties": {
        "RoleArn": {
          "Fn::GetAtt": [ "StateMachineRole", "Arn" ]
        },
        "TracingConfiguration": {
          "Enabled": true
        },
        "Definition": {
          "StartAt": "HelloWorld",
          "States": {
            "HelloWorld": {
              "Type": "Pass",
              "End": true
            }
          }
        }
      }
    },
    "StateMachineRole": {
      "Type": "AWS::IAM::Role",
      "Properties": {
        "AssumeRolePolicyDocument": {
```

```
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": [
          "sts:AssumeRole"
        ],
        "Effect": "Allow",
        "Principal": {
          "Service": [
            "states.amazonaws.com"
          ]
        }
      }
    ],
    "ManagedPolicyArns": [],
    "Policies": [
      {
        "PolicyName": "StateMachineRolePolicy",
        "PolicyDocument": {
          "Statement": [
            {
              "Action": [
                "lambda:InvokeFunction"
              ],
              "Resource": "*",
              "Effect": "Allow"
            }
          ]
        }
      }
    ]
  }
},
"Outputs": {
  "StateMachineArn": {
    "Value": {
      "Ref": "MyStateMachine"
    }
  }
}
}
```


JSON with DefinitionString

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "AWS Step Functions sample template.",
  "Resources": {
    "MyStateMachine": {
      "Type": "AWS::StepFunctions::StateMachine",
      "Properties": {
        "RoleArn": {
          "Fn::GetAtt": [ "StateMachineRole", "Arn" ]
        },
        "TracingConfiguration": {
          "Enabled": true
        },
        "DefinitionString": "{\n  \\"StartAt\\": \\"HelloWorld\\",\n  \\"States\\": {\n    \\"HelloWorld\\": {\n      \\"Type\\": \\"Pass\\",\n      \\"End\\": true\n    }\n  }\n}"
      }
    },
    "StateMachineRole": {
      "Type": "AWS::IAM::Role",
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
            {
              "Action": [
                "sts:AssumeRole"
              ],
              "Effect": "Allow",
              "Principal": {
                "Service": [
                  "states.amazonaws.com"
                ]
              }
            }
          ]
        }
      }
    },
    "ManagedPolicyArns": [],
    "Policies": [
      {
        "PolicyName": "StateMachineRolePolicy",
        "PolicyDocument": {
          "Statement": [
```



```

    HelloWorld:
      Type: Pass
      End: true
StateMachineRole:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Version: 2012-10-17
      Statement:
        - Action:
            - 'sts:AssumeRole'
          Effect: Allow
          Principal:
            Service:
              - states.amazonaws.com
    ManagedPolicyArns: []
    Policies:
      - PolicyName: StateMachineRolePolicy
        PolicyDocument:
          Statement:
            - Action:
                - 'lambda:InvokeFunction'
              Resource: "*"
              Effect: Allow

Outputs:
  StateMachineArn:
    Value:
      Ref: MyStateMachine

```

YAML with DefinitionString

```

AWSTemplateFormatVersion: 2010-09-09
Description: AWS Step Functions sample template.
Resources:
  MyStateMachine:
    Type: 'AWS::StepFunctions::StateMachine'
    Properties:
      RoleArn: !GetAtt
        - StateMachineRole
        - Arn
      TracingConfiguration:
        Enabled: true

```

```
DefinitionString: |
  {
    "StartAt": "HelloWorld",
    "States": {
      "HelloWorld": {
        "Type": "Pass",
        "End": true
      }
    }
  }
StateMachineRole:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Version: 2012-10-17
      Statement:
        - Action:
            - 'sts:AssumeRole'
          Effect: Allow
          Principal:
            Service:
              - states.amazonaws.com
    ManagedPolicyArns: []
  Policies:
    - PolicyName: StateMachineRolePolicy
      PolicyDocument:
        Statement:
          - Action:
              - 'lambda:InvokeFunction'
            Resource: "*"
            Effect: Allow

Outputs:
  StateMachineArn:
    Value:
      Ref: MyStateMachine
```

AWS Step Functions and AWS SAM

You can use the AWS SAM CLI in conjunction with the AWS Toolkit for Visual Studio Code as part of an integrated experience to create state machines locally. You can build a serverless application with AWS SAM, then build out your state machine in the VS Code IDE. Then you can validate,

package, and deploy your resources. Optionally, you can also publish to the AWS Serverless Application Repository.

Tip

To deploy a sample serverless application that starts a Step Functions workflow using AWS SAM to your AWS account, see [Module 11 - Deploy with AWS SAM](#) of *The AWS Step Functions Workshop*.

Topics

- [Why use Step Functions with AWS SAM?](#)
- [Step Functions integration with the AWS SAM specification](#)
- [Step Functions integration with the SAM CLI](#)
- [DefinitionSubstitutions in AWS SAM templates](#)
- [Next steps](#)

Why use Step Functions with AWS SAM?

When you use Step Functions with AWS SAM you can:

- Get started using a AWS SAM sample template.
- Build your state machine into your serverless application.
- Use variable substitution to substitute ARNs into your state machine at the time of deployment.

AWS CloudFormation supports [DefinitionSubstitutions](#) that let you add dynamic references in your workflow definition to a value that you provide in your CloudFormation template. You can add dynamic references by adding substitutions to your workflow definition using the `${dollar_sign_brace}` notation. You also need to define these dynamic references in the `DefinitionSubstitutions` property for the `StateMachine` resource in your CloudFormation template. These substitutions are replaced with actual values during the CloudFormation stack creation process. For more information, see [DefinitionSubstitutions in AWS SAM templates](#).

- Specify your state machine's role using AWS SAM policy templates.
- Initiate state machine executions with API Gateway, EventBridge events, or on a schedule within your AWS SAM template.

Step Functions integration with the AWS SAM specification

You can use the [AWS SAM Policy Templates](#) to add permissions to your state machine. With these permissions, you can orchestrate Lambda functions and other AWS resources to form complex and robust workflows.

Step Functions integration with the SAM CLI

Step Functions is integrated with the AWS SAM CLI. Use this to quickly develop a state machine into your serverless application.

Try the [Create a Step Functions state machine using AWS SAM](#) tutorial to learn how to use AWS SAM to create state machines.

Supported AWS SAM CLI functions include:

CLI Command	Description
sam init	Initializes a Serverless Application with an AWS SAM template. Can be used with a SAM template for Step Functions.
sam validate	Validates an AWS SAM template.
sam package	Packages an AWS SAM application. It creates a ZIP file of your code and dependencies, and then uploads it to Amazon S3. It then returns a copy of your AWS SAM template, replacing references to local artifacts with the Amazon S3 location where the command uploaded the artifacts.
sam deploy	Deploys an AWS SAM application.
sam publish	Publish an AWS SAM application to the AWS Serverless Application Repository. This command takes a packaged AWS SAM template and publishes the application to the specified region.

Note

When using AWS SAM local, you can emulate Lambda and API Gateway locally. However, you can't emulate Step Functions locally using AWS SAM.

DefinitionSubstitutions in AWS SAM templates

You can define state machines using CloudFormation templates with AWS SAM. Using AWS SAM, you can define the state machine inline in the template or in a separate file. The following AWS SAM template includes a state machine that simulates a stock trading workflow. This state machine invokes three Lambda functions to check the price of a stock and determine whether to buy or sell the stock. This transaction is then recorded in an Amazon DynamoDB table. The ARNs for the Lambda functions and DynamoDB table in the following template are specified using [DefinitionSubstitutions](#).

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: |
  step-functions-stock-trader
  Sample SAM Template for step-functions-stock-trader
Resources:
  StockTradingStateMachine:
    Type: AWS::Serverless::StateMachine
    Properties:
      DefinitionSubstitutions:
        StockCheckerFunctionArn: !GetAtt StockCheckerFunction.Arn
        StockSellerFunctionArn: !GetAtt StockSellerFunction.Arn
        StockBuyerFunctionArn: !GetAtt StockBuyerFunction.Arn
        DDBPutItem: !Sub arn:${AWS::Partition}:states:::dynamodb:putItem
        DDBTable: !Ref TransactionTable
      Policies:
        - DynamoDBWritePolicy:
            TableName: !Ref TransactionTable
        - LambdaInvokePolicy:
            FunctionName: !Ref StockCheckerFunction
        - LambdaInvokePolicy:
            FunctionName: !Ref StockBuyerFunction
        - LambdaInvokePolicy:
            FunctionName: !Ref StockSellerFunction
    DefinitionUri: statemachine/stock_trader.asl.json
```

```
StockCheckerFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: functions/stock-checker/
    Handler: app.lambdaHandler
    Runtime: nodejs18.x
    Architectures:
      - x86_64
StockSellerFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: functions/stock-seller/
    Handler: app.lambdaHandler
    Runtime: nodejs18.x
    Architectures:
      - x86_64
StockBuyerFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: functions/stock-buyer/
    Handler: app.lambdaHandler
    Runtime: nodejs18.x
    Architectures:
      - x86_64
TransactionTable:
  Type: AWS::DynamoDB::Table
  Properties:
    AttributeDefinitions:
      - AttributeName: id
      AttributeType: S
```

The following code is the state machine definition in the file `stock_trader.asl.json` which is used in the [Create a Step Functions state machine using AWS SAM](#) tutorial. This state machine definition contains several `DefinitionSubstitutions` denoted by the `${dollar_sign_brace}` notation. For example, instead of specifying a static Lambda function ARN for the `Check Stock Value` task, the substitution `${StockCheckerFunctionArn}` is used. This substitution is defined in the [DefinitionSubstitutions](#) property of the template. `DefinitionSubstitutions` is a map of key-value pairs for the state machine resource. In `DefinitionSubstitutions`, `${StockCheckerFunctionArn}` maps to the ARN of the `StockCheckerFunction` resource using the CloudFormation intrinsic function [!GetAtt](#). When

you deploy the AWS SAM template, the `DefinitionSubstitutions` in the template are replaced with the actual values.

```
{
  "Comment": "A state machine that does mock stock trading.",
  "StartAt": "Check Stock Value",
  "States": {
    "Check Stock Value": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "OutputPath": "$.Payload",
      "Parameters": {
        "Payload.$": "$",
        "FunctionName": "${StockCheckerFunctionArn}"
      },
      "Next": "Buy or Sell?"
    },
    "Buy or Sell?": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.stock_price",
          "NumericLessThanEquals": 50,
          "Next": "Buy Stock"
        }
      ],
      "Default": "Sell Stock"
    },
    "Buy Stock": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "OutputPath": "$.Payload",
      "Parameters": {
        "Payload.$": "$",
        "FunctionName": "${StockBuyerFunctionArn}"
      },
      "Retry": [
        {
          "ErrorEquals": [
            "Lambda.ServiceException",
            "Lambda.AWSLambdaException",
            "Lambda.SdkClientException",
            "Lambda.TooManyRequestsException"
          ]
        }
      ]
    }
  }
}
```

```
        ],
        "IntervalSeconds": 1,
        "MaxAttempts": 3,
        "BackoffRate": 2
    }
],
"Next": "Record Transaction"
},
"Sell Stock": {
    "Type": "Task",
    "Resource": "arn:aws:states:::lambda:invoke",
    "OutputPath": "$.Payload",
    "Parameters": {
        "Payload.$": "$",
        "FunctionName": "${StockSellerFunctionArn}"
    },
    "Next": "Record Transaction"
},
"Record Transaction": {
    "Type": "Task",
    "Resource": "arn:aws:states:::dynamodb:putItem",
    "Parameters": {
        "TableName": "${DDBTable}",
        "Item": {
            "Id": {
                "S.$": "$.id"
            },
            "Type": {
                "S.$": "$.type"
            },
            "Price": {
                "N.$": "$.price"
            },
            "Quantity": {
                "N.$": "$.qty"
            },
            "Timestamp": {
                "S.$": "$.timestamp"
            }
        }
    },
    "End": true
}
}
```

```
}
```

Next steps

You can learn more about using Step Functions with AWS SAM with the following resources:

- Complete the [Create a Step Functions state machine using AWS SAM](#) tutorial to create a state machine with AWS SAM.
- Specify a [AWS::Serverless::StateMachine](#) resource.
- Find [AWS SAM Policy Templates](#) to use.
- Use [AWS Toolkit for Visual Studio Code](#) with Step Functions.
- Review the [AWS SAM CLI reference](#) to learn more about the features available in AWS SAM.

You can also design and build your workflows in infrastructure as code (IaC) using visual builders, such as Workflow Studio in Application Composer. For more information, see [Using Workflow Studio in Application Composer](#).

Using Workflow Studio in Application Composer

AWS Application Composer is a visual builder that helps you develop AWS SAM and AWS CloudFormation templates using a simple graphical interface. With Application Composer, you design an application architecture by dragging, grouping, and connecting AWS services in a visual canvas. Application Composer then creates an infrastructure as code (IaC) template from your design that you can use to deploy your application with the AWS SAM Command Line Interface (AWS SAM CLI) or CloudFormation. To learn more about Application Composer, see [What is Application Composer](#).

Workflow Studio is available in Application Composer to help you design and build your workflows. Workflow Studio in Application Composer provides a visual IaC environment that makes it easy for you to incorporate workflows in your serverless applications built using IaC tools, such as CloudFormation templates. When you use Workflow Studio in Application Composer, it connects the individual workflow steps to AWS resources and generates the resource configurations in an AWS SAM template. It also adds the IAM permissions required for your workflow to run. Using Workflow Studio in Application Composer, you can create prototypes of your applications and turn them into production-ready applications.

When you use Workflow Studio in Application Composer, you can switch back and forth between the Application Composer canvas and Workflow Studio.

Topics

- [Using Workflow Studio in Application Composer to build a serverless workflow](#)
- [Dynamically reference resources using CloudFormation definition substitutions in Workflow Studio](#)
- [Connect service integration tasks to enhanced component cards](#)
- [Import existing projects and sync them locally](#)
- [Unavailable Workflow Studio features in AWS Application Composer](#)

Using Workflow Studio in Application Composer to build a serverless workflow

1. Open the [Application Composer console](#) and choose **Create project** to create a project.
2. In the search field in the **Resources** palette, enter **state machine**.
3. Drag the **Step Functions State machine** resource onto the canvas.
4. Choose **Edit in Workflow Studio** to edit your state machine resource.

The following animation shows how you can switch to the Workflow Studio for editing your state machine definition.

An animation that illustrates how you can use Workflow Studio in Application Composer.

The integration with Workflow Studio to edit state machines resources created in Application Composer is only available for [AWS::Serverless::StateMachine](#) resource. This integration is not available for templates that use the [AWS::StepFunctions::StateMachine](#) resource.

Dynamically reference resources using CloudFormation definition substitutions in Workflow Studio

In Workflow Studio, you can use CloudFormation definition substitutions in your workflow definition to dynamically reference resources that you've defined in your IaC template. You can add placeholder substitutions to your workflow definition using the `${dollar_sign_brace}` notation and they are replaced with actual values during the CloudFormation stack creation

process. For more information about definition substitutions, see [DefinitionSubstitutions in AWS SAM templates](#).

The following animation shows how you can add placeholder substitutions for the resources in your state machine definition.

An animation that illustrates how to dynamically reference resources, such as AWS Lambda functions, definition substitutions when you use Workflow Studio in Application Composer.

Connect service integration tasks to enhanced component cards

You can connect the tasks that call [optimized service integrations](#) to [enhanced component cards](#) in Application Composer canvas. Doing this automatically maps any placeholder substitutions specified by the `#{dollar_sign_brace}` notation in your workflow definition and the `DefinitionSubstitution` property for your `StateMachine` resource. It also adds the appropriate AWS SAM policies for the state machine.

If you map optimized service integration tasks with [standard component cards](#), the connection line doesn't appear on the Application Composer canvas.

The following animation shows how you can connect an optimized task to an enhanced component card and view the changes in [Change Inspector](#).

An animation that illustrates how to connect tasks that call optimized service integrations to enhanced component cards when you use Workflow Studio in Application Composer.

You can't connect [AWS SDK integrations](#) in your Task state with enhanced component cards or optimized service integrations with standard component cards. For these tasks, you can map the substitutions in the **Resource properties** panel in Application Composer canvas, and add policies in the AWS SAM template.

Tip

Alternatively, you can also map placeholder substitutions for your state machine under **Definition Substitutions** in the **Resource properties** panel. When you do this, you must add the required permissions for the AWS service your Task state calls in the state machine execution role. For information about permissions your execution role might need, see [Execution roles in Workflow Studio](#).

The following animation shows how you can manually update the placeholder substitution mapping in the **Resource properties** panel.

An animation that illustrates how to manually update the placeholder substitution mapping in the **Resource properties** panel when you use Workflow Studio in Application Composer.

Import existing projects and sync them locally

You can open existing CloudFormation and AWS SAM projects in Application Composer to visualize them for better understanding and modify their designs. Using Application Composer's [local sync](#) feature, you can automatically sync and save your template and code files to your local build machine. Using the local sync mode can compliment your existing development flows. Make sure that your browser supports the [File System Access API](#), which allows web applications to read, write, and save files in your local file system. We recommend using either Google Chrome or Microsoft Edge.

Unavailable Workflow Studio features in AWS Application Composer

When you use Workflow Studio in Application Composer, some of the Workflow Studio features are unavailable. In addition, the **API Parameters** section available in the [Inspector](#) panel supports CloudFormation definition substitutions. You can add the substitutions in the [Code mode](#) using the `${dollar_sign_brace}` notation. For more information about this notation, see [DefinitionSubstitutions in AWS SAM templates](#).

The following list describes the Workflow Studio features that are unavailable when you use Workflow Studio in Application Composer:

- [Starter templates](#) – Starter templates are ready-to-run sample projects that automatically create the workflow proptotypes and definitions. These templates deploys all the related AWS resources that your project needs to your AWS account.
- [Config mode](#) – This mode lets you manage the configuration of your state machines. You can update your state machine configurations in your IaC templates or use the **Resource properties** panel in Application Composer canvas. For information about updating configurations in the **Resource properties** panel, see [Connect service integration tasks to enhanced component cards](#).
- [TestState](#) API
- Option to import or export workflow definitions from the **Actions** dropdown button in Workflow Studio. Instead, from the Application Composer **menu**, select **Open > Project folder**. Make sure that you've enabled the [local sync](#) mode to automatically save your changes in the Application Composer canvas directly to your local machine.

- **Execute** button. When you use Workflow Studio in Application Composer, Application Composer generates the IaC code for your workflow. Therefore, you must first deploy the template. Then, run the workflow in the console or through the AWS Command Line Interface (AWS CLI).

Creating a Lambda state machine for Step Functions using AWS CloudFormation

This tutorial shows you how to create a basic AWS Lambda function using AWS CloudFormation. You'll use the AWS CloudFormation console and a YAML template to create the *stack* (IAM roles, the Lambda function, and the state machine). Then, you'll use the AWS Step Functions console to start the state machine execution.

For more information, see [Working with CloudFormation Templates](#) and the [AWS::StepFunctions::StateMachine](#) resource in the *AWS CloudFormation User Guide*.

Topics

- [Step 1: Set up your AWS CloudFormation template](#)
- [Step 2: Use the AWS CloudFormation template to create a Lambda State Machine](#)
- [Step 3: Start a State Machine execution](#)

Step 1: Set up your AWS CloudFormation template

Before you use the [example templates](#), you should understand how to declare the different parts of an AWS CloudFormation template.

Topics

- [To create an IAM role for Lambda](#)
- [To create a Lambda function](#)
- [To create an IAM role for the state machine execution](#)
- [To create a Lambda state machine](#)

To create an IAM role for Lambda

Define the trust policy associated with the IAM role for the Lambda function. The following examples define a trust policy using either YAML or JSON.

YAML

```
LambdaExecutionRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
          Action: "sts:AssumeRole"
```

JSON

```
"LambdaExecutionRole": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "lambda.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
}
```

To create a Lambda function

Define the following properties for a Lambda function that will print the message Hello World.

Important

Ensure that your Lambda function is under the same AWS account and AWS Region as your state machine.

YAML

```
MyLambdaFunction:
  Type: "AWS::Lambda::Function"
  Properties:
    Handler: "index.handler"
    Role: !GetAtt [ LambdaExecutionRole, Arn ]
    Code:
      ZipFile: |
        exports.handler = (event, context, callback) => {
          callback(null, "Hello World!");
        };
    Runtime: "nodejs12.x"
    Timeout: "25"
```

JSON

```
{
  "MyLambdaFunction": {
    "Type": "AWS::Lambda::Function",
    "Properties": {
      "Handler": "index.handler",
      "Role": {
        "Fn::GetAtt": [
          "LambdaExecutionRole",
          "Arn"
        ]
      },
      "Code": {
        "ZipFile": "exports.handler = (event, context, callback) => {\n
callback(null, \"Hello World!\");\n};\n"
      },
      "Runtime": "nodejs12.x",
      "Timeout": "25"
    }
  },
}
```

To create an IAM role for the state machine execution

Define the trust policy associated with the IAM role for the state machine execution.

YAML

```

StatesExecutionRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: "Allow"
          Principal:
            Service:
              - !Sub states.${AWS::Region}.amazonaws.com
          Action: "sts:AssumeRole"
    Path: "/"
    Policies:
      - PolicyName: StatesExecutionPolicy
        PolicyDocument:
          Version: "2012-10-17"
          Statement:
            - Effect: Allow
              Action:
                - "lambda:InvokeFunction"
              Resource: "*"

```

JSON

```

"StatesExecutionRole": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": [
              {
                "Fn::Sub": "states.
${AWS::Region}.amazonaws.com"
              }
            ]
          }
        }
      ],
      "Action": "sts:AssumeRole"
    }
  }
}

```



```

        "Resource": "${lambdaArn}",
        "End": true
    }
}
- {lambdaArn: !GetAtt [ MyLambdaFunction, Arn ]}
RoleArn: !GetAtt [ StatesExecutionRole, Arn ]

```

JSON

```

"MyStateMachine": {
  "Type": "AWS::StepFunctions::StateMachine",
  "Properties": {
    "DefinitionString": {
      "Fn::Sub": [
        "{\n  \"Comment\": \"A Hello World example using an\n  AWS Lambda function\",\n  \"StartAt\": \"HelloWorld\",\n  \"States\": {\n  \"HelloWorld\": {\n    \"Type\": \"Task\",\n    \"Resource\": \"${lambdaArn}\",\n    \"End\": true\n  } }\n}",
        {
          "lambdaArn": {
            "Fn::GetAtt": [
              "MyLambdaFunction",
              "Arn"
            ]
          }
        }
      ]
    },
    "RoleArn": {
      "Fn::GetAtt": [
        "StatesExecutionRole",
        "Arn"
      ]
    }
  }
}

```

Step 2: Use the AWS CloudFormation template to create a Lambda State Machine

Once you understand the components of the AWS CloudFormation template, you can put them together and use the template to create an AWS CloudFormation stack.

To create the Lambda state machine

1. Copy the following example data to a file named `MyStateMachine.yaml` for the YAML example, or `MyStateMachine.json` for JSON.

YAML

```
AWSTemplateFormatVersion: "2010-09-09"
Description: "An example template with an IAM role for a Lambda state machine."
Resources:
  LambdaExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: "sts:AssumeRole"

  MyLambdaFunction:
    Type: "AWS::Lambda::Function"
    Properties:
      Handler: "index.handler"
      Role: !GetAtt [ LambdaExecutionRole, Arn ]
      Code:
        ZipFile: |
          exports.handler = (event, context, callback) => {
            callback(null, "Hello World!");
          };
      Runtime: "nodejs12.x"
      Timeout: "25"

  StatesExecutionRole:
    Type: "AWS::IAM::Role"
```

```

Properties:
  AssumeRolePolicyDocument:
    Version: "2012-10-17"
    Statement:
      - Effect: "Allow"
        Principal:
          Service:
            - !Sub states.${AWS::Region}.amazonaws.com
        Action: "sts:AssumeRole"
  Path: "/"
  Policies:
    - PolicyName: StatesExecutionPolicy
      PolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Action:
              - "lambda:InvokeFunction"
            Resource: "*"

MyStateMachine:
  Type: "AWS::StepFunctions::StateMachine"
  Properties:
    DefinitionString:
      !Sub
      - |-
        {
          "Comment": "A Hello World example using an AWS Lambda function",
          "StartAt": "HelloWorld",
          "States": {
            "HelloWorld": {
              "Type": "Task",
              "Resource": "${lambdaArn}",
              "End": true
            }
          }
        }
      - {lambdaArn: !GetAtt [ MyLambdaFunction, Arn ]}
    RoleArn: !GetAtt [ StatesExecutionRole, Arn ]

```

JSON

```
{
```

```
"AWSTemplateFormatVersion": "2010-09-09",
  "Description": "An example template with an IAM role for a Lambda state
machine.",
  "Resources": {
    "LambdaExecutionRole": {
      "Type": "AWS::IAM::Role",
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
            {
              "Effect": "Allow",
              "Principal": {
                "Service": "lambda.amazonaws.com"
              },
              "Action": "sts:AssumeRole"
            }
          ]
        }
      }
    },
    "MyLambdaFunction": {
      "Type": "AWS::Lambda::Function",
      "Properties": {
        "Handler": "index.handler",
        "Role": {
          "Fn::GetAtt": [
            "LambdaExecutionRole",
            "Arn"
          ]
        },
        "Code": {
          "ZipFile": "exports.handler = (event, context, callback) =>
{\n  callback(null, \"Hello World!\");\n};\n"
        },
        "Runtime": "nodejs12.x",
        "Timeout": "25"
      }
    },
    "StatesExecutionRole": {
      "Type": "AWS::IAM::Role",
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Version": "2012-10-17",
```

```

        "Statement": [
            {
                "Effect": "Allow",
                "Principal": {
                    "Service": [
                        {
                            "Fn::Sub": "states.
${AWS::Region}.amazonaws.com"
                        }
                    ]
                },
                "Action": "sts:AssumeRole"
            }
        ],
        "Path": "/",
        "Policies": [
            {
                "PolicyName": "StatesExecutionPolicy",
                "PolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Action": [
                                "lambda:InvokeFunction"
                            ],
                            "Resource": "*"
                        }
                    ]
                }
            }
        ]
    },
    "MyStateMachine": {
        "Type": "AWS::StepFunctions::StateMachine",
        "Properties": {
            "DefinitionString": {
                "Fn::Sub": [
                    "{\n  \"Comment\": \"A Hello World example using
an AWS Lambda function\",
  \"StartAt\": \"HelloWorld\",
  \"States\":
{\n    \"HelloWorld\": {\n      \"Type\": \"Task\",
      \"Resource\":
\"${lambdaArn}\",
      \"End\": true\n    }\n  }"}
                ]
            }
        }
    }
}

```



```
{
  "lambdaArn": {
    "Fn::GetAtt": [
      "MyLambdaFunction",
      "Arn"
    ]
  },
  "RoleArn": {
    "Fn::GetAtt": [
      "StatesExecutionRole",
      "Arn"
    ]
  }
}
```

2. Open the [AWS CloudFormation console](#) and choose **Create Stack**.
3. On the **Select Template** page, choose **Upload a template to Amazon S3**. Choose your MyStateMachine file, and then choose **Next**.
4. On the **Specify Details** page, for **Stack name**, enter MyStateMachine, and then choose **Next**.
5. On the **Options** page, choose **Next**.
6. On the **Review** page, choose **I acknowledge that AWS CloudFormation might create IAM resources**. and then choose **Create**.

AWS CloudFormation begins to create the MyStateMachine stack and displays the **CREATE_IN_PROGRESS** status. When the process is complete, AWS CloudFormation displays the **CREATE_COMPLETE** status.

7. (Optional) To display the resources in your stack, select the stack and choose the **Resources** tab.

▼ Resources

To view detailed drift information for specific resources, visit the [Drift Details page](#).

Logical ID	Physical ID	Type	Drift Status	Status	Status Reason
LambdaExecutionRole	MyStateMachine-LambdaExecutionRole-1DNCMT8YQJTB4	AWS::IAM::Role	NOT_CHECKED	CREATE_COMPLETE	
MyLambdaFunction	MyStateMachine-MyLambdaFunction-VFQCSK4HCF	AWS::Lambda::Function	NOT_CHECKED	CREATE_COMPLETE	
MyStateMachine	arn:aws:states:us-east-1:999942473912:stateMachine:MyStateMachine-USWVRPCPE5	AWS::StepFunctions::State...	NOT_CHECKED	CREATE_COMPLETE	
StatesExecutionRole	MyStateMachine-StatesExecutionRole-VMS3WJADIE7	AWS::IAM::Role	NOT_CHECKED	CREATE_COMPLETE	

Step 3: Start a State Machine execution

After you create your Lambda state machine, you can start its execution.

To start the state machine execution

1. Open the [Step Functions console](#) and choose the name of the state machine that you created using AWS CloudFormation.
2. On the **MyStateMachine-ABCDEFGHIJK** page, choose **New execution**.

The **New execution** page is displayed.

3. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

4. Choose **Start Execution**.

A new execution of your state machine starts, and a new page showing your running execution is displayed.

5. (Optional) In the **Execution Details**, review the **Execution Status** and the **Started** and **Closed** timestamps.
6. To view the results of your execution, choose **Output**.

Creating a Lambda state machine for Step Functions using AWS CDK

This tutorial shows how to create an AWS Step Functions state machine that contains an AWS Lambda function using the AWS Cloud Development Kit (AWS CDK). The AWS CDK is an Infrastructure as Code (IAC) framework that lets you define AWS infrastructure using a full-fledged programming language. You can write an app in one of the CDK's supported languages containing one or more stacks. Then, you can synthesize it to an AWS CloudFormation template and deploy it to your AWS account. We'll use this method to define a Step Functions state machine containing a Lambda function, then use the AWS Management Console to run the state machine.

Before you begin this tutorial, you must set up your AWS CDK development environment as described in [Getting Started With the AWS CDK - Prerequisites](#) in the *AWS Cloud Development Kit (AWS CDK) Developer Guide*. Then, install the AWS CDK with the following command at the AWS CLI:

```
npm install -g aws-cdk
```

This tutorial produces the same result as [the section called "Creating a Lambda State Machine Using AWS CloudFormation"](#). However, in this tutorial, the AWS CDK doesn't require you to create any IAM roles; the AWS CDK does it for you. The AWS CDK version also includes a [Succeed](#) step to illustrate how to add additional steps to your state machine.

Tip

To deploy a sample serverless application that starts a Step Functions workflow using AWS CDK with TypeScript to your AWS account, see [Module 10 - Deploy with AWS CDK](#) of *The AWS Step Functions Workshop*.

Topics

- [Step 1: Set up your AWS CDK project](#)
- [Step 2: Use AWS CDK to create a state machine](#)
- [Step 3: Start a state machine execution](#)
- [Step 4: Clean Up](#)
- [Next steps](#)

Step 1: Set up your AWS CDK project

1. In your home directory, or another directory if you prefer, run the following command to create a directory for your new AWS CDK app.

Important

Be sure to name the directory `step`. The AWS CDK application template uses the name of the directory to generate names for source files and classes. If you use a different name, your app will not match this tutorial.

TypeScript

```
mkdir step && cd step
```

JavaScript

```
mkdir step && cd step
```

Python

```
mkdir step && cd step
```

Java

```
mkdir step && cd step
```

C#

Make sure you've installed .NET version 6.0 or higher. For information, see [Supported versions](#).

```
mkdir step && cd step
```

2. Initialize the app by using the **cdk init** command. Specify the desired template ("app") and programming language as shown in the following examples.

TypeScript

```
cdk init --language typescript
```

JavaScript

```
cdk init --language javascript
```

Python

```
cdk init --language python
```

After the project is initialized, activate the project's virtual environment and install the AWS CDK's baseline dependencies.

```
source .venv/bin/activate  
python -m pip install -r requirements.txt
```

Java

```
cdk init --language java
```

C#

```
cdk init --language csharp
```

Step 2: Use AWS CDK to create a state machine

First, we'll present the individual pieces of code that define the Lambda function and the Step Functions state machine. Then, we'll explain how to put them together in your AWS CDK app. Finally, you'll see how to synthesize and deploy these resources.

To create a Lambda function

The following AWS CDK code defines the Lambda function, providing its source code inline.

TypeScript

```
const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
  code: lambda.Code.fromInline(`
    exports.handler = (event, context, callback) => {
      callback(null, "Hello World!");
    }
  `),
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: "index.handler",
  timeout: cdk.Duration.seconds(3)
});
```

JavaScript

```
const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
  code: lambda.Code.fromInline(`
    exports.handler = (event, context, callback) => {
      callback(null, "Hello World!");
    }
  `),
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: "index.handler",
  timeout: cdk.Duration.seconds(3)
});
```

Python

```
hello_function = lambda_.Function(
    self, "MyLambdaFunction",
    code=lambda_.Code.from_inline("""
    exports.handler = (event, context, callback) => {
      callback(null, "Hello World!");
    }"""),
    runtime=lambda_.Runtime.NODEJS_18_X,
    handler="index.handler",
    timeout=Duration.seconds(25))
```

Java

```
final Function helloFunction = Function.Builder.create(this, "MyLambdaFunction")
    .code(Code.fromInline(
```

```
        "exports.handler = (event, context, callback) => { callback(null,
'Hello World!' );}"))
        .runtime(Runtime.NODEJS_18_X)
        .handler("index.handler")
        .timeout(Duration.seconds(25))
        .build();
```

C#

```
var helloFunction = new Function(this, "MyLambdaFunction", new FunctionProps
{
    Code = Code.FromInline(@"`
        exports.handler = (event, context, callback) => {
            callback(null, 'Hello World!');
        }"),
    Runtime = Runtime.NODEJS_12_X,
    Handler = "index.handler",
    Timeout = Duration.Seconds(25)
});
```

You can see in this short example code:

- The function's logical name, `MyLambdaFunction`.
- The source code for the function, embedded as a string in the source code of the AWS CDK app.
- Other function attributes, such as the runtime to be used (Node 18.x), the function's entry point, and a timeout.

To create a state machine

Our state machine has two states: a Lambda function task, and a [Succeed](#) state. The function requires that we create a Step Functions [the section called “Task”](#) that invokes our function. This Task state is used as the first step in the state machine. The success state is added to the state machine using the Task state's `next()` method. The following code first invokes the function named `MyLambdaTask`, then uses the `next()` method to define a success state named `GreetedWorld`.

TypeScript

```
const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
```

```

definition: new tasks.LambdaInvoke(this, "MyLambdaTask", {
    lambdaFunction: helloFunction
}).next(new sfm.Succeed(this, "GreetedWorld"))
});

```

JavaScript

```

const stateMachine = new sfm.StateMachine(this, 'MyStateMachine', {
    definition: new tasks.LambdaInvoke(this, "MyLambdaTask", {
        lambdaFunction: helloFunction
    }).next(new sfm.Succeed(this, "GreetedWorld"))
});

```

Python

```

state_machine = sfm.StateMachine(
    self, "MyStateMachine",
    definition=tasks.LambdaInvoke(
        self, "MyLambdaTask",
        lambda_function=hello_function)
    .next(sfm.Succeed(self, "GreetedWorld")))

```

Java

```

final StateMachine stateMachine = StateMachine.Builder.create(this,
    "MyStateMachine")
    .definition(LambdaInvoke.Builder.create(this, "MyLambdaTask")
        .lambdaFunction(helloFunction)
        .build())
    .next(new Succeed(this, "GreetedWorld"))
    .build();

```

C#

```

var stateMachine = new StateMachine(this, "MyStateMachine", new StateMachineProps {
    DefinitionBody = DefinitionBody.FromChainable(new LambdaInvoke(this,
    "MyLambdaTask", new LambdaInvokeProps
    {
        LambdaFunction = helloFunction
    })
    .Next(new Succeed(this, "GreetedWorld")))
});

```


To build and deploy the AWS CDK app

In your newly created AWS CDK project, edit the file that contains the stack's definition to look like the following example code. You'll recognize the definitions of the Lambda function and the Step Functions state machine from previous sections.

1. Update the stack as shown in the following examples.

TypeScript

Update `lib/step-stack.ts` with the following code.

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as sfm from 'aws-cdk-lib/aws-stepfunctions';
import * as tasks from 'aws-cdk-lib/aws-stepfunctions-tasks';

export class StepStack extends cdk.Stack {
  constructor(app: cdk.App, id: string) {
    super(app, id);

    const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
      code: lambda.Code.fromInline(`
        exports.handler = (event, context, callback) => {
          callback(null, "Hello World!");
        }
      `),
      runtime: lambda.Runtime.NODEJS_18_X,
      handler: "index.handler",
      timeout: cdk.Duration.seconds(3)
    });

    const stateMachine = new sfm.StateMachine(this, 'MyStateMachine', {
      definition: new tasks.LambdaInvoke(this, "MyLambdaTask", {
        lambdaFunction: helloFunction
      }).next(new sfm.Succeed(this, "GreetedWorld"))
    });
  }
}
```

JavaScript

Update `lib/step-stack.js` with the following code.

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as sfn from 'aws-cdk-lib/aws-stepfunctions';
import * as tasks from 'aws-cdk-lib/aws-stepfunctions-tasks';

export class StepStack extends cdk.Stack {
  constructor(app, id) {
    super(app, id);

    const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
      code: lambda.Code.fromInline(`
        exports.handler = (event, context, callback) => {
          callback(null, "Hello World!");
        }
      `),
      runtime: lambda.Runtime.NODEJS_18_X,
      handler: "index.handler",
      timeout: cdk.Duration.seconds(3)
    });

    const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
      definition: new tasks.LambdaInvoke(this, "MyLambdaTask", {
        lambdaFunction: helloFunction
      }).next(new sfn.Succeed(this, "GreetedWorld"))
    });
  }
}
```

Python

Update `step/step_stack.py` with the following code.

```
from aws_cdk import (
    Duration,
    Stack,
    aws_stepfunctions as sfn,
    aws_stepfunctions_tasks as tasks,
    aws_lambda as lambda_
)

class StepStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
```

```
super().__init__(scope, construct_id, **kwargs)

hello_function = lambda_.Function(
    self, "MyLambdaFunction",
    code=lambda_.Code.from_inline("""
exports.handler = (event, context, callback) => {
    callback(null, "Hello World!");
}"""),
    runtime=lambda_.Runtime.NODEJS_18_X,
    handler="index.handler",
    timeout=Duration.seconds(25))

state_machine = sfn.StateMachine(
    self, "MyStateMachine",
    definition=tasks.LambdaInvoke(
        self, "MyLambdaTask",
        lambda_function=hello_function)
        .next(sfn.Succeed(self, "GreetedWorld")))
```

Java

Update `src/main/java/com.myorg/StepStack.java` with the following code.

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.Duration;
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.stepfunctions.StateMachine;
import software.amazon.awscdk.services.stepfunctions.Succeed;
import software.amazon.awscdk.services.stepfunctions.tasks.LambdaInvoke;

public class StepStack extends Stack {
    public StepStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public StepStack(final Construct scope, final String id, final StackProps
        props) {
```

```

        super(scope, id, props);

        final Function helloFunction = Function.Builder.create(this,
            "MyLambdaFunction")
            .code(Code.fromInline(
                "exports.handler = (event, context, callback) =>
            { callback(null, 'Hello World!' );}"))
            .runtime(Runtime.NODEJS_18_X)
            .handler("index.handler")
            .timeout(Duration.seconds(25))
            .build();

        final StateMachine stateMachine = StateMachine.Builder.create(this,
            "MyStateMachine")
            .definition(LambdaInvoke.Builder.create(this, "MyLambdaTask")
                .lambdaFunction(helloFunction)
                .build()
                .next(new Succeed(this, "GreetedWorld")))
            .build();
    }
}

```

C#

Update `scr/Step/StepStack.cs` with the following code.

```

using Amazon.CDK;
using Constructs;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.StepFunctions;
using Amazon.CDK.AWS.StepFunctions.Tasks;

namespace Step
{
    public class StepStack : Stack
    {
        internal StepStack(Construct scope, string id, IStackProps props =
            null) : base(scope, id, props)
        {
            var helloFunction = new Function(this, "MyLambdaFunction", new
            FunctionProps
            {

```

```
        Code = Code.FromInline(@"exports.handler = (event, context,
callback) => {
            callback(null, 'Hello World!');
        }"),
        Runtime = Runtime.NODEJS_18_X,
        Handler = "index.handler",
        Timeout = Duration.Seconds(25)
    });

    var stateMachine = new StateMachine(this, "MyStateMachine", new
StateMachineProps
    {
        DefinitionBody = DefinitionBody.FromChainable(new
LambdaInvoke(this, "MyLambdaTask", new LambdaInvokeProps
        {
            LambdaFunction = helloFunction
        })
        .Next(new Succeed(this, "GreetedWorld")))
    });
}
}
```

2. Save the source file, and then run the `cdk synth` command in the app's main directory.

AWS CDK runs the app and synthesizes an AWS CloudFormation template from it. AWS CDK then displays the template.

Note

If you used TypeScript to create your AWS CDK project, running the `cdk synth` command may return the following error.

```
TSError: # Unable to compile TypeScript:
bin/step.ts:7:33 - error TS2554: Expected 2 arguments, but got 3.
```

Modify the `bin/step.ts` file as shown in the following example to resolve this error.

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { StepStack } from '../lib/step-stack';
```

```
const app = new cdk.App();
new StepStack(app, 'StepStack');
app.synth();
```

3. To deploy the Lambda function and the Step Functions state machine to your AWS account, issue `cdk deploy`. You'll be asked to approve the IAM policies the AWS CDK has generated.

Step 3: Start a state machine execution

After you create your state machine, you can start its execution.

To start the state machine execution

1. Open the [Step Functions console](#) and choose the name of the state machine that you created using AWS CDK.
2. On the state machine page, choose **Start execution**.

The **Start execution** dialog box is displayed.

3. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

4. Choose **Start Execution**.

Your state machine's execution starts, and a new page showing your running execution is displayed.

5. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Step 4: Clean Up

After you've tested your state machine, we recommend that you remove both your state machine and the related Lambda function to free up resources in your AWS account. Run the `cdk destroy` command in your app's main directory to remove your state machine.

Next steps

To learn more about developing AWS infrastructure using AWS CDK, see the [AWS CDK Developer Guide](#).

For information about writing AWS CDK apps in your language of choice, see:

TypeScript

[Working with AWS CDK in TypeScript](#)

JavaScript

[Working with AWS CDK in JavaScript](#)

Python

[Working with AWS CDK in Python](#)

Java

[Working with AWS CDK in Java](#)

C#

[Working with AWS CDK in C#](#)

For more information about the AWS Construct Library modules used in this tutorial, see the following AWS CDK API Reference overviews:

- [aws-lambda](#)
- [aws-stepfunctions](#)
- [aws-stepfunctions-tasks](#)

Creating an API Gateway REST API with Synchronous Express State Machine Using the AWS CDK

This tutorial shows you how to create an API Gateway REST API with Synchronous Express State Machine as the backend integration using the AWS Cloud Development Kit (AWS CDK). This tutorial will use the `StepFunctionsRestApi` construct to connect the State Machine to the API Gateway. The `StepFunctionsRestApi` construct will set up a default input/output mapping and the API Gateway REST API, with required permissions and an HTTP “ANY” method. The AWS CDK is an Infrastructure as Code (IAC) framework that lets you define AWS infrastructure using a full-fledged programming language. You write an app in one of the CDK's supported languages, containing one or more stacks, then synthesize it to an AWS CloudFormation template and deploy it to your AWS account. We'll use it to define an API Gateway REST API, which is integrated with Synchronous Express State Machine as the backend, then use the AWS Management Console to initiate execution.

Before embarking on this tutorial, set up your AWS CDK development environment as described in [Getting Started With the AWS CDK - Prerequisites](#), then install the AWS CDK by issuing:

```
npm install -g aws-cdk
```

Topics

- [Step 1: Set Up Your AWS CDK Project](#)
- [Step 2: Use the AWS CDK to create an API Gateway REST API with Synchronous Express State Machine backend integration](#)
- [Step 3: Test the API Gateway](#)
- [Step 4: Clean Up](#)

Step 1: Set Up Your AWS CDK Project

First, create a directory for your new AWS CDK app and initialize the project.

TypeScript

```
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language typescript
```

JavaScript

```
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language javascript
```

Python

```
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language python
```

After the project has been initialized, activate the project's virtual environment and install the AWS CDK's baseline dependencies.

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language java
```

C#

```
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language csharp
```

Go

```
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
```

```
cdk init --language go
```

Note

Be sure to name the directory `stepfunctions-rest-api`. The AWS CDK application template uses the name of the directory to generate names for source files and classes. If you use a different name, your app will not match this tutorial.

Now install the construct library modules for AWS Step Functions and Amazon API Gateway.

TypeScript

```
npm install @aws-cdk/aws-stepfunctions @aws-cdk/aws-apigateway
```

JavaScript

```
npm install @aws-cdk/aws-stepfunctions @aws-cdk/aws-apigateway
```

Python

```
python -m pip install aws-cdk.aws-stepfunctions  
python -m pip install aws-cdk.aws-apigateway
```

Java

Edit the project's `pom.xml` to add the following dependencies inside the existing `<dependencies>` container.

```
<dependency>  
  <groupId>software.amazon.awscdk</groupId>  
  <artifactId>stepfunctions</artifactId>  
  <version>${cdk.version}</version>  
</dependency>  
<dependency>  
  <groupId>software.amazon.awscdk</groupId>  
  <artifactId>apigateway</artifactId>  
  <version>${cdk.version}</version>  
</dependency>
```

Maven automatically installs these dependencies the next time you build your app. To build, issue `mvn compile` or use your Java IDE's **Build** command.

C#

```
dotnet add src/StepfunctionsRestApi package Amazon.CDK.AWS.Stepfunctions
dotnet add src/StepfunctionsRestApi package Amazon.CDK.AWS.APIGateway
```

You may also install the indicated packages using the Visual Studio NuGet GUI, available via **Tools > NuGet Package Manager > Manage NuGet Packages for Solution**.

Once you have installed the modules, you can use them in your AWS CDK app by importing the following packages.

TypeScript

```
@aws-cdk/aws-stepfunctions
@aws-cdk/aws-apigateway
```

JavaScript

```
@aws-cdk/aws-stepfunctions
@aws-cdk/aws-apigateway
```

Python

```
aws_cdk.aws_stepfunctions
aws_cdk.aws_apigateway
```

Java

```
software.amazon.awscdk.services.apigateway.StepFunctionsRestApi
software.amazon.awscdk.services.stepfunctions.Pass
software.amazon.awscdk.services.stepfunctions.StateMachine
software.amazon.awscdk.services.stepfunctions.StateMachineType
```

C#

```
Amazon.CDK.AWS.StepFunctions
Amazon.CDK.AWS.APIGateway
```

Go

Add the following to `import` inside `stepfunctions-rest-api.go`.

```
"github.com/aws/aws-cdk-go/awscdk/awsapigateway"  
"github.com/aws/aws-cdk-go/awscdk/awsstepfunctions"
```

Step 2: Use the AWS CDK to create an API Gateway REST API with Synchronous Express State Machine backend integration

First, we'll present the individual pieces of code that define the Synchronous Express State Machine and the API Gateway REST API, then explain how to put them together into your AWS CDK app. Then you'll see how to synthesize and deploy these resources.

Note

The State Machine that we will show here will be a simple State Machine with a Pass state.

To create an Express State Machine

This is the AWS CDK code that defines a simple state machine with a Pass state.

TypeScript

```
const machineDefinition = new stepfunctions.Pass(this, 'PassState', {  
  result: {value:"Hello!"},  
})  
  
const stateMachine = new stepfunctions.StateMachine(this, 'MyStateMachine', {  
  definition: machineDefinition,  
  stateMachineType: stepfunctions.StateMachineType.EXPRESS,  
});
```

JavaScript

```
const machineDefinition = new sfn.Pass(this, 'PassState', {  
  result: {value:"Hello!"},  
})
```

```
const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
  definition: machineDefinition,
  stateMachineType: stepfunctions.StateMachineType.EXPRESS,
});
```

Python

```
machine_definition = sfn.Pass(self, "PassState",
                             result = sfn.Result("Hello"))

state_machine = sfn.StateMachine(self, 'MyStateMachine',
                                  definition = machine_definition,
                                  state_machine_type = sfn.StateMachineType.EXPRESS)
```

Java

```
Pass machineDefinition = Pass.Builder.create(this, "PassState")
    .result(Result.fromString("Hello"))
    .build();

StateMachine stateMachine = StateMachine.Builder.create(this, "MyStateMachine")
    .definition(machineDefinition)
    .stateMachineType(StateMachineType.EXPRESS)
    .build();
```

C#

```
var machineDefinition = new Pass(this, "PassState", new PassProps
{
    Result = Result.FromString("Hello")
});

var stateMachine = new StateMachine(this, "MyStateMachine", new StateMachineProps
{
    Definition = machineDefinition,
    StateMachineType = StateMachineType.EXPRESS
});
```

Go

```
var machineDefinition = awsstepfunctions.NewPass(stack, jsii.String("PassState"),
&awsstepfunctions.PassProps
```

```
{
    Result: awsstepfunctions.NewResult(jsii.String("Hello")),
})

var stateMachine = awsstepfunctions.NewStateMachine(stack,
    jsii.String("StateMachine"), &awsstepfunctions.StateMachineProps
    {
        Definition: machineDefinition,
        StateMachineType: awsstepfunctions.StateMachineType_EXPRESS,
    })
```

You can see in this short snippet:

- The machine definition named `PassState`, which is a Pass State.
- The State Machine's logical name, `MyStateMachine`.
- The machine definition is used as the State Machine definition.
- The State Machine Type is set as `EXPRESS` because `StepFunctionsRestApi` will only allow a Synchronous Express state machine.

To create the API Gateway REST API using `StepFunctionsRestApi` construct

We will use `StepFunctionsRestApi` construct to create the API Gateway REST API with required permissions and default input/output mapping.

TypeScript

```
const api = new apigateway.StepFunctionsRestApi(this,
    'StepFunctionsRestApi', { stateMachine: stateMachine });
```

JavaScript

```
const api = new apigateway.StepFunctionsRestApi(this,
    'StepFunctionsRestApi', { stateMachine: stateMachine });
```

Python

```
api = apigw.StepFunctionsRestApi(self, "StepFunctionsRestApi",
    state_machine = state_machine)
```

Java

```
StepFunctionsRestApi api = StepFunctionsRestApi.Builder.create(this,
    "StepFunctionsRestApi")
    .stateMachine(stateMachine)
    .build();
```

C#

```
var api = new StepFunctionsRestApi(this, "StepFunctionsRestApi", new
    StepFunctionsRestApiProps
{
    StateMachine = stateMachine
});
```

Go

```
awsapigateway.NewStepFunctionsRestApi(stack, jsii.String("StepFunctionsRestApi"),
    &awsapigateway.StepFunctionsRestApiProps
{
    StateMachine = stateMachine,
})
```

To build and deploy the AWS CDK app

In the AWS CDK project you created, edit the file containing the definition of the stack to look like the code below. You'll recognize the definitions of the Step Functions state machine and the API Gateway from above.

TypeScript

Update `lib/stepfunctions-rest-api-stack.ts` to read as follows.

```
import * as cdk from 'aws-cdk-lib';
import * as stepfunctions from 'aws-cdk-lib/aws-stepfunctions';
import * as apigateway from 'aws-cdk-lib/aws-apigateway';

export class StepfunctionsRestApiStack extends cdk.Stack {
    constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
```

```
super(scope, id, props);

const machineDefinition = new stepfunctions.Pass(this, 'PassState', {
  result: {value:"Hello!"},
});

const stateMachine = new stepfunctions.StateMachine(this, 'MyStateMachine', {
  definition: machineDefinition,
  stateMachineType: stepfunctions.StateMachineType.EXPRESS,
});

const api = new apigateway.StepFunctionsRestApi(this,
  'StepFunctionsRestApi', { stateMachine: stateMachine });
```

JavaScript

Update `lib/stepfunctions-rest-api-stack.js` to read as follows.

```
const cdk = require('@aws-cdk/core');
const stepfunctions = require('@aws-cdk/aws-stepfunctions');
const apigateway = require('@aws-cdk/aws-apigateway');

class StepfunctionsRestApiStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const machineDefinition = new stepfunctions.Pass(this, "PassState", {
      result: {value:"Hello!"},
    })

    const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
      definition: machineDefinition,
      stateMachineType: stepfunctions.StateMachineType.EXPRESS,
    });

    const api = new apigateway.StepFunctionsRestApi(this,
      'StepFunctionsRestApi', { stateMachine: stateMachine });

  }
}

module.exports = { StepStack }
```


Python

Update `stepfunctions_rest_api/stepfunctions_rest_api_stack.py` to read as follows.

```
from aws_cdk import App, Stack
from constructs import Construct
from aws_cdk import aws_stepfunctions as sfn
from aws_cdk import aws_apigateway as apigw

class StepfunctionsRestApiStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        machine_definition = sfn.Pass(self, "PassState",
                                     result = sfn.Result("Hello"))

        state_machine = sfn.StateMachine(self, 'MyStateMachine',
                                         definition = machine_definition,
                                         state_machine_type = sfn.StateMachineType.EXPRESS)

        api = apigw.StepFunctionsRestApi(self,
                                         "StepFunctionsRestApi",
                                         state_machine = state_machine)
```

Java

Update `src/main/java/com.myorg/StepfunctionsRestApiStack.java` to read as follows.

```
package com.myorg;

import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.stepfunctions.Pass;
import software.amazon.awscdk.services.stepfunctions.StateMachine;
import software.amazon.awscdk.services.stepfunctions.StateMachineType;
import software.amazon.awscdk.services.apigateway.StepFunctionsRestApi;
```

```

public class StepfunctionsRestApiStack extends Stack {
    public StepfunctionsRestApiStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public StepfunctionsRestApiStack(final Construct scope, final String id, final
StackProps props) {
        super(scope, id, props);

        Pass machineDefinition = Pass.Builder.create(this, "PassState")
            .result(Result.fromString("Hello"))
            .build();

        StateMachine stateMachine = StateMachine.Builder.create(this,
"MyStateMachine")
            .definition(machineDefinition)
            .stateMachineType(StateMachineType.EXPRESS)
            .build();

        StepFunctionsRestApi api = StepFunctionsRestApi.Builder.create(this,
"StepFunctionsRestApi")
            .stateMachine(stateMachine)
            .build();
    }
}

```

C#

Update `src/StepfunctionsRestApi/StepfunctionsRestApiStack.cs` to read as follows.

```

using Amazon.CDK;
using Amazon.CDK.AWS.StepFunctions;
using Amazon.CDK.AWS.APIGateway;

namespace StepfunctionsRestApi
{
    public class StepfunctionsRestApiStack : Stack
    {
        internal StepfunctionsRestApi(Construct scope, string id, IStackProps props
= null) : base(scope, id, props)
        {

```

```

        var machineDefinition = new Pass(this, "PassState", new PassProps
        {
            Result = Result.FromString("Hello")
        });

        var stateMachine = new StateMachine(this, "MyStateMachine", new
        StateMachineProps
        {
            Definition = machineDefinition,
            StateMachineType = StateMachineType.EXPRESS
        });

        var api = new StepFunctionsRestApi(this, "StepFunctionsRestApi", new
        StepFunctionsRestApiProps
        {
            StateMachine = stateMachine
        });
    }
}
}
}
}

```

Go

Update `stepfunctions-rest-api.go` to read as follows.

```

package main
import (
    "github.com/aws/aws-cdk-go/awscdk"
    "github.com/aws/aws-cdk-go/awscdk/awsapigateway"
    "github.com/aws/aws-cdk-go/awscdk/awsstepfunctions"
    "github.com/aws/constructs-go/constructs/v3"
    "github.com/aws/jsii-runtime-go"
)

type StepfunctionsRestApiGoStackProps struct {
    awscdk.StackProps
}

func NewStepfunctionsRestApiGoStack(scope constructs.Construct, id string, props
*StepfunctionsRestApiGoStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {

```

```
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  // The code that defines your stack goes here
  var machineDefinition = awsstepfunctions.NewPass(stack,
jsii.String("PassState"), &awsstepfunctions.PassProps
  {
    Result: awsstepfunctions.NewResult(jsii.String("Hello")),
  })

  var stateMachine = awsstepfunctions.NewStateMachine(stack,
jsii.String("StateMachine"), &awsstepfunctions.StateMachineProps{
    Definition: machineDefinition,
    StateMachineType: awsstepfunctions.StateMachineType_EXPRESS,
  });

  awsapigateway.NewStepFunctionsRestApi(stack,
jsii.String("StepFunctionsRestApi"), &awsapigateway.StepFunctionsRestApiProps{
    StateMachine = stateMachine,
  })

  return stack
}

func main() {
  app := awscdk.NewApp(nil)

  NewStepfunctionsRestApiGoStack(app, "StepfunctionsRestApiGoStack",
&StepfunctionsRestApiGoStackProps{
    awscdk.StackProps{
      Env: env(),
    },
  })

  app.Synth(nil)
}

// env determines the AWS environment (account+region) in which our stack is to
// be deployed. For more information see: https://docs.aws.amazon.com/cdk/latest/guide/environments.html
func env() *awscdk.Environment {
  // If unspecified, this stack will be "environment-agnostic".
  // Account/Region-dependent features and context lookups will not work, but a
```

```
// single synthesized template can be deployed anywhere.
//-----
return nil

// Uncomment if you know exactly what account and region you want to deploy
// the stack to. This is the recommendation for production stacks.
//-----
// return &awscdk.Environment{
//   Account: jsii.String("123456789012"),
//   Region:  jsii.String("us-east-1"),
// }

// Uncomment to specialize this stack for the AWS Account and Region that are
// implied by the current CLI configuration. This is recommended for dev
// stacks.
//-----
// return &awscdk.Environment{
//   Account: jsii.String(os.Getenv("CDK_DEFAULT_ACCOUNT")),
//   Region:  jsii.String(os.Getenv("CDK_DEFAULT_REGION")),
// }
}
```

Save the source file, then issue `cdk synth` in the app's main directory. The AWS CDK runs the app and synthesizes an AWS CloudFormation template from it, then displays the template.

To actually deploy the Amazon API Gateway and the AWS Step Functions state machine to your AWS account, issue `cdk deploy`. You'll be asked to approve the IAM policies the AWS CDK has generated. The policies being created will look something like this:

```

IAM Statement Changes
+ | Resource | Effect | Action | Principal | Condition |
+ | ${SfnDemoCdkStack--StateMachine-apiRole.Arn} | Allow | sts:AssumeRole | Service:apigateway.amazonaws.com | |
+ | ${StateMachine} | Allow | states:StartSyncExecution | AWS:${SfnDemoCdkStack--StateMachine-apiRole} | |
+ | ${StateMachine/Role.Arn} | Allow | sts:AssumeRole | Service:states.${AWS::Region}.amazonaws.com | |
+ | ${StepFunctions-rest-api/CloudWatchRole.Arn} | Allow | sts:AssumeRole | Service:apigateway.amazonaws.com | |

IAM Policy Changes
+ | Resource | Managed Policy ARN |
+ | ${StepFunctions-rest-api/CloudWatchRole} | arn:${AWS::Partition}:iam::aws:policy/service-role/AmazonAPIGatewayPushToCloudWatchLogs |

(NOTE: There may be security-related changes not in this list. See https://github.com/aws/aws-cdk/issues/1299)
Do you wish to deploy these changes (y/n)? 

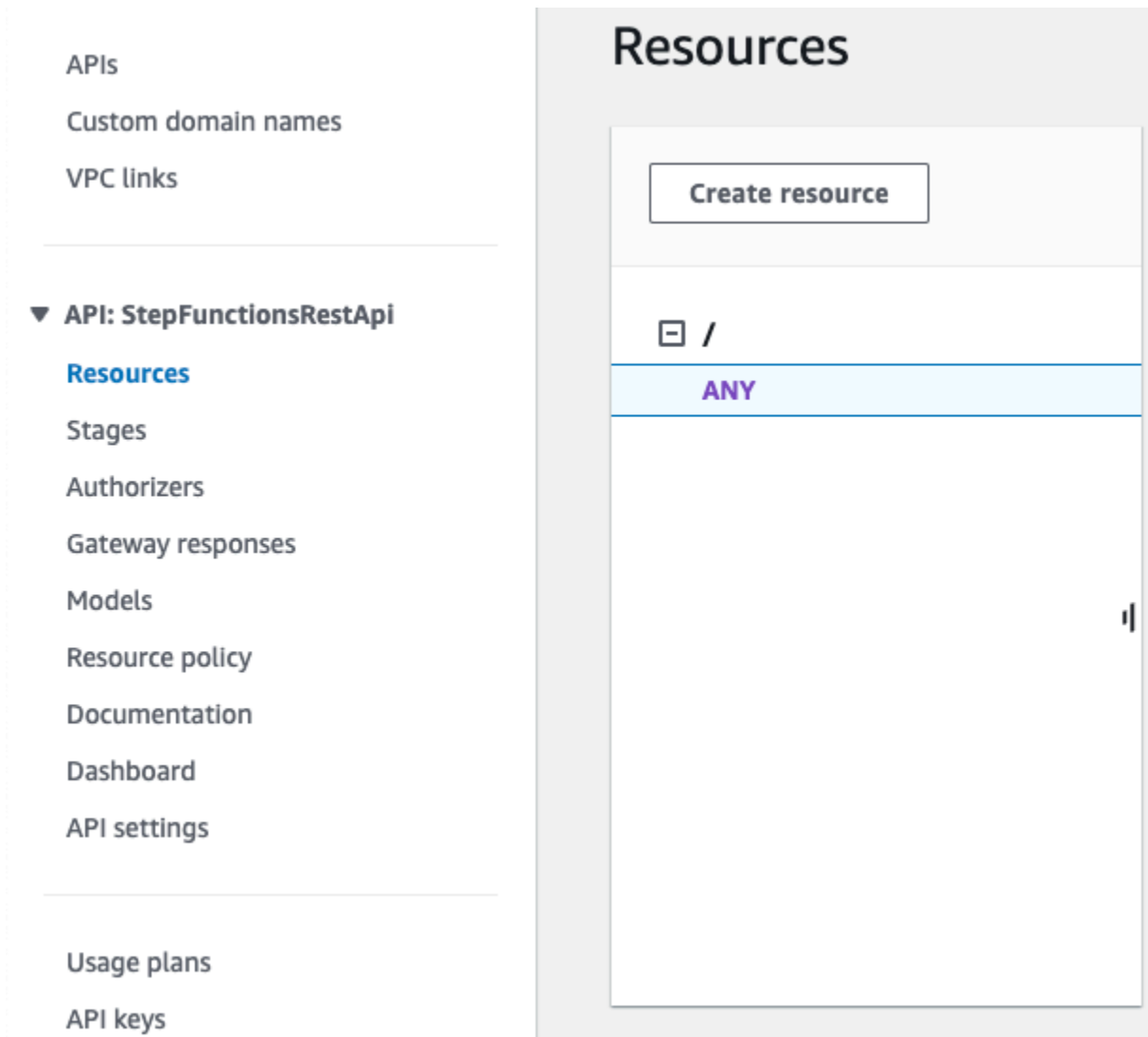
```

Step 3: Test the API Gateway

After you create your API Gateway REST API with Synchronous Express State Machine as the backend integration, you can test the API Gateway.

To test the deployed API Gateway using API Gateway console

1. Open the [Amazon API Gateway console](#) and sign in.
2. Choose your REST API named `StepFunctionsRestApi`.
3. In the **Resources** pane, choose the ANY method.



4. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
5. For **Method**, choose **POST**.
6. For **Request body**, copy the following request parameters.

```
{
  "key": "Hello"
}
```

7. Choose **Test**. The following information will be displayed:
 - **Request** is the resource's path that was called for the method.
 - **Status** is the response's HTTP status code.

- **Latency** is the time between the receipt of the request from the caller and the returned response.
- **Response body** is the HTTP response body.
- **Response headers** are the HTTP response headers.
- **Log** shows the simulated Amazon CloudWatch Logs entries that would have been written if this method were called outside of the API Gateway console.

Note

Although the CloudWatch Logs entries are simulated, the results of the method call are real.

The **Response body** output should be something like this:

```
"Hello"
```

Tip

Try the API Gateway with different methods and an invalid input to see the error output. You may want to change the state machine to look for a particular key and during testing provide the wrong key to fail the State Machine execution and generate an error message in the **Response body** output.

To test the deployed API using cURL

1. Open a terminal window.
2. Copy the following cURL command and paste it into the terminal window, replacing `<api-id>` with your API's API ID and `<region>` with the region where your API is deployed.

```
curl -X POST\  
  'https://<api-id>.execute-api.<region>.amazonaws.com/prod' \  
  -d '{"key":"Hello"}' \  
  -H 'Content-Type: application/json'
```


The **Response Body** output should be something like this:

```
"Hello"
```

 **Tip**

Try the API Gateway with different methods and an invalid input to see the error output. You may want to change the state machine to look for a particular key and during testing provide the wrong key to fail the State Machine execution and generate an error message in the **Response Body** output.

Step 4: Clean Up

When you're done trying out your API Gateway, you can tear down both the state machine and the API Gateway using the AWS CDK. Issue `cdk destroy` in your app's main directory.

AWS Step Functions Data Science SDK for Python

The AWS Step Functions Data Science SDK is an open source library for data scientists. With this SDK, you can create workflows that process and publish machine learning models using SageMaker and Step Functions. You can also create multi-step machine learning workflows in Python that orchestrate AWS infrastructure at scale, without having to provision and integrate the AWS services separately.

The AWS Step Functions Data Science SDK provides a Python API that can create and invoke Step Functions workflows. You can manage and execute these workflows directly in Python, as well as Jupyter notebooks.

In addition to creating production-ready workflows directly in Python, the AWS Step Functions Data Science SDK allows you to copy that workflow, experiment with new options, and then put the refined workflow into production.

For more information about the AWS Step Functions Data Science SDK, see the following:

- [Project on Github](#)
- [SDK documentation](#)

- The following [Example notebooks](#), which are available in Jupyter notebook instances in the [SageMaker console](#) and the related [GitHub project](#):
 - `hello_world_workflow.ipynb`
 - `machine_learning_workflow_abalone.ipynb`
 - `training_pipeline_pytorch_mnist.ipynb`

Deploying state machines using Terraform

[Terraform](#) by HashiCorp is a framework for building applications using infrastructure as code (IaC). With Terraform, you can create state machines and use features, such as previewing infrastructure deployments and creating reusable templates. Terraform templates help you maintain and reuse the code by breaking it down into smaller chunks.

If you're familiar with Terraform, you can follow the development lifecycle described in this topic as a model for creating and deploying your state machines in Terraform. If you aren't familiar with Terraform, we recommend that you first complete the workshop [Introduction to Terraform on AWS](#) for getting acquainted with Terraform.

Tip

To deploy an example of a state machine built using Terraform to your AWS account, see the module [Managing state machines with infrastructure as code](#) of *The AWS Step Functions Workshop*.

In this topic

- [Prerequisites](#)
- [State machine development lifecycle with Terraform](#)
- [IAM roles and policies for your state machine](#)

Prerequisites

Before you get started, make sure you complete the following prerequisites:

- Install Terraform on your machine. For information about installing Terraform, see [Install Terraform](#).

- Install Step Functions Local on your machine. We recommend that you install the Step Functions Local Docker image to use Step Functions Local. For more information, see [Testing state machines locally](#).
- Install AWS SAM CLI. For installation information, see [Installing the AWS SAM CLI](#) in the *AWS Serverless Application Model Developer Guide*.
- Install the AWS Toolkit for Visual Studio Code to view the workflow diagram of your state machines. For installation information, see [Installing the AWS Toolkit for Visual Studio Code](#) in the *AWS Toolkit for Visual Studio Code User Guide*.

State machine development lifecycle with Terraform

The following procedure explains how you can use a state machine prototype that you build using [Workflow Studio](#) in the Step Functions console as a starting point for local development with Terraform and the [AWS Toolkit for Visual Studio Code](#).

To view the complete example that discusses the state machine development with Terraform and presents the best practices in detail, see [Best practices for writing Step Functions Terraform projects](#).

To start the development lifecycle of a state machine with Terraform

1. Bootstrap a new Terraform project with the following command.

```
terraform init
```

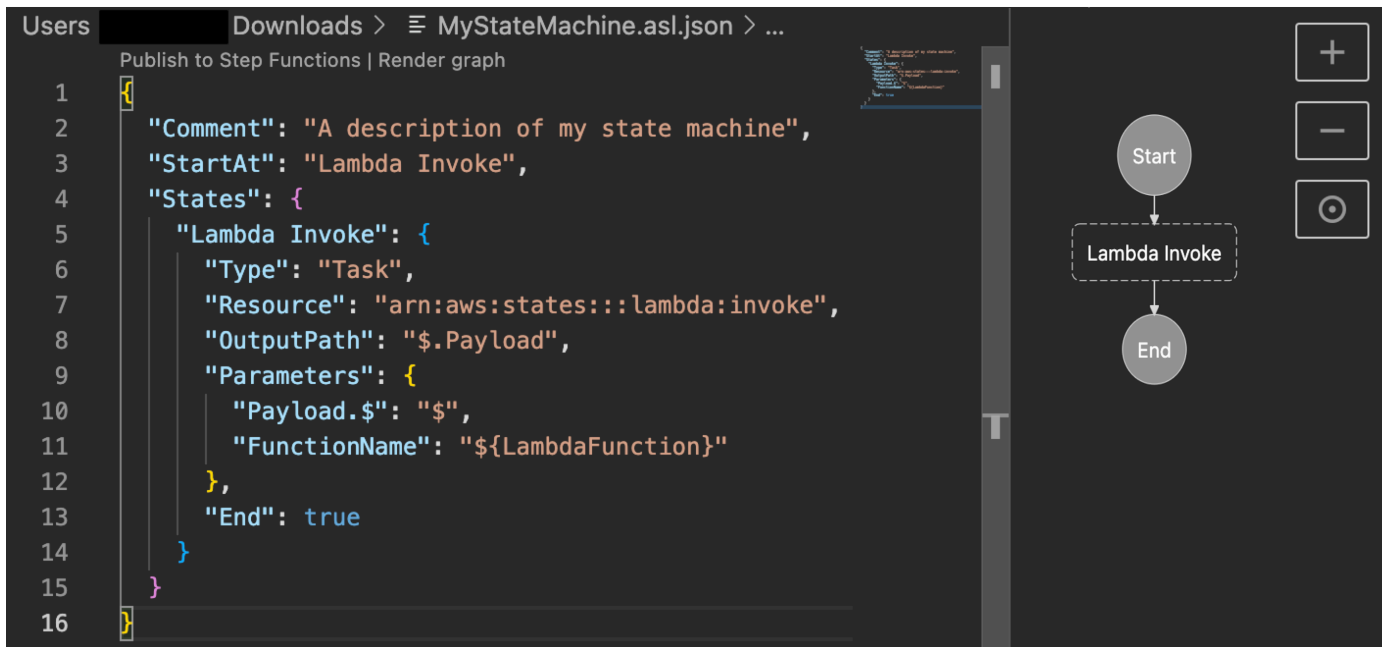
2. Open the [Step Functions console](#) to create a prototype for your state machine.
3. In Workflow Studio, do the following:
 - a. Create your workflow prototype.
 - b. Export the [Amazon States Language \(ASL\)](#) definition of your workflow. To do this, choose the **Import/Export** dropdownlist, and then select **Export JSON definition**.
4. Save the exported ASL definition within your project directory.

You pass the exported ASL definition as an input parameter to the [aws_sfn_state_machine](#) Terraform resource that uses the [templatefile](#) function. This function is used inside the definition field that passes the exported ASL definition and any variable substitutions.

Tip

Because the ASL definition file can contain lengthy blocks of text, we recommend you avoid the inline EOF method. This makes it easier to substitute parameters into your state machine definition.

- (Optional) Update the ASL definition within your IDE and visualize your changes using the AWS Toolkit for Visual Studio Code.



To avoid continuously exporting your definition and refactoring it into your project, we recommend that you make updates locally in your IDE and track these updates with [Git](#).

- Test your workflow using [Step Functions Local](#).

Tip

You can also locally test service integrations with Lambda functions and API Gateway APIs in your state machine using [AWS SAM CLI Local](#).

- Preview your state machine and other AWS resources before deploying the state machine. To do this, run the following command.

```
terraform plan
```

8. Deploy your state machine from your local environment or through [CI/CD pipelines](#) using the following command.

```
terraform apply
```

9. (Optional) Clean up your resources and delete the state machine using the following command.

```
terraform destroy
```

IAM roles and policies for your state machine

Use the [Terraform service integration policies](#) to add necessary IAM permissions to your state machine, for example, permission to invoke Lambda functions. You can also define explicit roles and policies and associate them with your state machine.

The following IAM policy example grants your state machine access to invoke a Lambda function named *myFunction*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:myFunction"
    }
  ]
}
```

We also recommend using the [aws_iam_policy_document](#) data source when defining IAM policies for your state machines in Terraform. This helps you check if your policy is malformed and substitute any resources with variables.

The following IAM policy example uses the `aws_iam_policy_document` data source and grants your state machine access to invoke a Lambda function named *myFunction*.

```
data "aws_iam_policy_document" "state_machine_role_policy" {
```

```
statement {
  effect = "Allow"

  actions = [
    "lambda:InvokeFunction"
  ]

  resources = ["${aws_lambda_function.["myFunction"].arn}:*"]
}
}
```

 Tip

To view more advanced AWS architectural patterns deployed with Terraform, see [Terraform examples at Serverless Land Workflows Collection](#).

Testing and debugging

Step Functions provides different ways to test and debug state machines. For example, you can [test and debug](#) your state machines in the console, use the [TestState](#) API to test an individual state, or use Step Functions Local to test state machines locally.

Using the [TestState](#) API, you provide the definition of a single state and execute it. You can test a single state without creating a state machine or updating an existing state machine.

Step Functions Local is a downloadable version of Step Functions that lets you develop and test applications using a version of Step Functions running in your own development environment. With Step Functions Local, you can run your state machines to test their input and output data flows, integrations with supported services, and more in your local development environment.

Topics

- [Using TestState API to test a state](#)
- [Testing state machines locally](#)

Using TestState API to test a state

The [TestState](#) API accepts the definition of a single state and executes it. You can test a state without creating a state machine or updating an existing state machine.

Using the [TestState](#) API, you can test the following:

- A state's [input and output processing data flow](#).
- An [AWS service integration](#) with other AWS services request and response
- An [HTTP Task](#) request and response

To test a state, you can also use the [Step Functions console](#), [AWS Command Line Interface \(AWS CLI\)](#), or the SDK.

The [TestState](#) API assumes an IAM role which must contain the required IAM permissions for the resources your state accesses. For information about the permissions a state might need, see [IAM permissions for using TestState API](#).

Topics

- [Considerations about using the TestState API](#)
- [Using inspection levels in TestState API](#)
- [IAM permissions for using TestState API](#)
- [Testing a state \(Console\)](#)
- [Testing a state using AWS CLI](#)
- [Testing and debugging input and output data flow](#)

Considerations about using the TestState API

Using the [TestState](#) API, you can test only one state at a time. The states that you can test include the following:

- All [Task types](#), except [Activities](#)
- [Pass](#)
- [Wait](#)
- [Choice](#)
- [Succeed](#)
- [Fail](#)

While using the TestState API, keep in mind the following considerations.

- The TestState API doesn't include support for the following:
 - [Task state](#) states that use the following resource types:
 - [Activity](#)
 - [Service integration patterns](#) of type `.sync` or `.waitForTaskToken`
 - [Parallel](#) state
 - [Map](#) state
- A test can run for up to five minutes. If a test exceeds this duration, it fails with the [States.Timeout](#) error.

Using inspection levels in TestState API

To test a state using the [TestState](#) API, you provide the definition of that state. The test then returns an output. For each state, you can specify the amount of detail you want to view in the test results. These details provide additional information about the state you're testing. For example, if you've used any input and output data processing filters, such as [InputPath](#) or [ResultPath](#) in a state, you can view the intermediate and final data processing results.

Step Functions provides the following levels to specify the details you want to view:

- [INFO](#)
- [DEBUG](#)
- [TRACE](#)

All these levels also return the `status` and `nextState` fields. `status` indicates the status of the state execution. For example, `SUCCEEDED`, `FAILED`, `RETRIABLE`, and `CAUGHT_ERROR`. `nextState` indicates the name of the next state to transition to. If you haven't defined a next state in your definition, this field returns an empty value.

For information about testing a state using these inspection levels in the Step Functions console and AWS CLI, see [Testing a state \(Console\)](#) and [Testing a state using AWS CLI](#).

INFO inspectionLevel

If the test succeeds, this level shows the state output. If the test fails, this level shows the error output. By default, Step Functions sets **Inspection level** to **INFO** if you don't specify a level.

Example of test with INFO level that succeeds

The following image shows a test for a Pass state that succeeds. The **Inspection level** for this state is set to **INFO** and the output for the state appears in the **Output** tab.

Test state ✕

Test a state in isolation using the TestState API to ensure that it works correctly. [Learn more](#)

✔

State Pass succeeded.

▶ Details

Test | State details

Execution role
Testing a state requires an execution role. Enter an IAM role ARN, select an existing IAM role from the list, or [learn how to create a new IAM role with permissions for a flow state](#)

↕
↻

State input - optional

```

1 {
2   "value1": 23,
3   "value2": 17
4 }
```

Must be in valid JSON format

Inspection level
Specifies the level of detail to return from this test. [Learn more](#)

INFO

▼

Return state output, status, error(s), and expected next step

Reveal secrets
Applies to HTTP tasks only. When combined with an inspection level of TRACE, will reveal any sensitive authorization data in the HTTP request and response. [Learn more](#)

Start test

Output | Input/output processing | HTTP request & response

▼ { 1 item
Collapse all

```

{
  "Sum" : 40
}
```

📄 Copy TestState API response

Done

Example of test with INFO level that fails

The following image shows a test that failed for a Task state when the **Inspection level** is set to **INFO**. The **Output** tab shows the error output that includes the error name and a detailed explanation of the cause for that error.

Test state ✕

Test a state in isolation using the TestState API to ensure that it works correctly. [Learn more](#)

✕ **Lambda.Unknown**
▶ Details

Test | State details

Execution role
 Testing a state requires an execution role. Enter an IAM role ARN, select an existing IAM role from the list, or [learn how to create a new IAM role with permissions for an optimized service integration](#)

myTaskStateRole

1 {
 "key": "value"
 }

Must be in valid JSON format

Inspection level
 Specifies the level of detail to return from this test. [Learn more](#)

INFO
 Return state output, status, error(s), and expected next step

Reveal secrets
 Applies to HTTP tasks only. When combined with an inspection level of TRACE, will reveal any sensitive authorization data in the HTTP request and response. [Learn more](#)

Start test

Output | Input/output processing | HTTP request & response

Expand all

```

{ 2 items
  "error" : "Lambda.Unknown"
  "cause" :
    "The cause could not be determined because Lambda did not return an error type. Returned payload: {"errorMessage":"2023-11-21T04:15:29.243Z c1abf98f-d3ef-4666-b0da-bc7c1a93b09a Task timed out after 3.01 seconds"}"
}
          
```

Copy TestState API response

Done

DEBUG inspectionLevel

If the test succeeds, this level shows the state output and the result of input and output data processing.

If the test fails, this level shows the error output. This level shows the intermediate data processing results up to the point of failure. For example, say that you tested a Task state that invokes a Lambda function. Imagine that you had applied the [InputPath](#), [Parameters](#), [ResultPath](#), and [OutputPath](#) filters to the Task state. Say that the invocation failed. In this case, the DEBUG level shows data processing results based on the application of the filters in the following order:

- `input` – Raw state input
- `afterInputPath` – Input after Step Functions applies the `InputPath` filter.
- `afterParameters` – The effective input after Step Functions applies the `Parameters` filter.

The diagnostic information available in this level can help you troubleshoot issues related to a [service integration](#) or [input and output data processing](#) flow that you might have defined.

Example of test with **DEBUG** level that succeeds

The following image shows a test for a Pass state that succeeds. The **Inspection level** for this state is set to **DEBUG**. The **Input/output processing** tab in the following image shows the result of the application of [Parameters](#) on the input provided for this state.

Test state ✕

Test a state in isolation using the TestState API to ensure that it works correctly. [Learn more](#)

✔ **State Pass succeeded.**
▶ Details

Test

State details

Execution role
Testing a state requires an execution role. Enter an IAM role ARN, select an existing IAM role from the list, or [learn how to create a new IAM role with permissions for a flow state](#)

myPassStateRole ↕ ↻

State input - optional

```

1 {
2   "inputArray": [
3     11,
4     12,
5     13
6   ]

```

Must be in valid JSON format

Inspection level
Specifies the level of detail to return from this test. [Learn more](#)

DEBUG ↕
Returns INFO-level detail + input/output processing

Reveal secrets
Applies to HTTP tasks only. When combined with an inspection level of TRACE, will reveal any sensitive authorization data in the HTTP request and response. [Learn more](#)

Start test

Output

Input/output processing

HTTP request & response

This tab shows the JSON data processing which occurred within the state when it executed. [Learn more](#)

```

{ 6 items
  "input" : {...} 1 item
  "afterInputPath" : {...} 1 item
  "afterParameters" : { 1 item
    "myArrayLength" : 3
  }
  "afterResultSelector" : {...} 1 item
  "afterResultPath" : {...} 1 item
  "output" : 3
}

```

Expand all

📄 Copy TestState API response

Done

Example of test with DEBUG level that fails

The following image shows a test that failed for a Task state when the **Inspection level** is set to **DEBUG**. The **Input/output processing** tab in the following image shows the input and output data processing result for the state up to the point of its failure.

Test state ✕

Test a state in isolation using the TestState API to ensure that it works correctly. [Learn more](#)

✕

States.Runtime

▶ Details

Test | State details

Execution role

Testing a state requires an execution role. Enter an IAM role ARN, select an existing IAM role from the list, or [learn how to create a new IAM role with permissions for an HTTP task](#)

AdminAllAccess
↕
↻

State input - optional

```

1 {
2   "object": "customer",
3   "address": null,
4   "balance": 0,
5   "created": 1699644289,
6   "currency": null

```

Must be in valid JSON format

Inspection level

Specifies the level of detail to return from this test. [Learn more](#)

DEBUG
↕

Reveal secrets

Applies to HTTP tasks only. When combined with an inspection level of TRACE, will reveal any sensitive authorization data in the HTTP request and response. [Learn more](#)

Start test

Output | **Input/output processing** | HTTP request & response

This tab shows the JSON data processing which occurred within the state when it executed. [Learn more](#)

Expand all

```

{ 2 items
  ▶ "input" : {...} 21 items
  ▶ "afterInputPath" : {...} 21 items
}

```

📄 Copy TestState API response

Done

TRACE inspectionLevel

Step Functions provides the **TRACE** level to test an [HTTP Task](#). This level returns information about the HTTP request that Step Functions makes and response that a third-party API returns. The response might contain information, such as headers and request body. In addition, you can view the state output and result of input and output data processing in this level.

If the test fails, this level shows the error output.

This level is only applicable for HTTP Task. Step Functions throws an error if you use this level for other state types.

When you set the **Inspection level** to **TRACE**, you can also view secrets included in the [EventBridge connection](#). To do this, you must set the `revealSecrets` parameter to `true` in the [TestState](#) API. In addition, you must make sure that the IAM user that calls the `TestState` API has permission for the `states:RevealSecrets` action. For an example of IAM policy that sets the `states:RevealSecrets` permission, see [IAM permissions for using TestState API](#). Without this permission, Step Functions throws an access denied error.

If you set the `revealSecrets` parameter to `false`, Step Functions omits all secrets in the HTTP request and response data.

Example of test with TRACE level that succeeds

The following image shows a test for an HTTP Task that succeeds. The **Inspection level** for this state is set to **TRACE**. The **HTTP request & response** tab in the following image shows the result of the third-party API call.

The following IAM policy example sets the `states:TestState`, `iam:PassRole`, and `states:RevealSecrets` permissions.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:TestState",
        "states:RevealSecrets",
        "iam:PassRole"
      ],
      "Resource": "*"
    }
  ]
}
```

Testing a state (Console)

You can test a [state](#) in the console and check the state output or input and output data processing flow. For an [HTTP Task](#), you can test the raw HTTP request and response.

To test a state

1. Open the [Step Functions console](#).
2. Choose **Create state machine** to start creating a state machine or choose an existing state machine.
3. In the [Design mode](#) of Workflow Studio, choose a state that you want to test.
4. Choose **Test state** in the [Inspector](#) panel of Workflow Studio.
5. In the **Test state** dialog box, do the following:
 - a. For **Execution role**, choose an execution role to test the state. Make sure that you have the required [IAM permissions](#) for the state that you want to test.
 - b. (Optional) Provide any JSON input that your selected state needs for the test.
 - c. For **Inspection level**, select one of the following options based on the values you want to view:

- [INFO](#) – Shows the state output in the **Output** tab if the test succeeds. If the test fails, **INFO** shows the error output, which includes the error name and a detailed explanation of the cause for that error. By default, Step Functions sets **Inspection level** to **INFO** if you don't select a level.
- [DEBUG](#) – Shows the state output and the result of input and output data processing if the test succeeds. If the test fails, **DEBUG** shows the error output, which includes the error name and a detailed explanation of the cause for that error.
- [TRACE](#) – Shows the raw HTTP request and response, and is useful for verifying headers, query parameters, and other API-specific details. This option is only available for the [HTTP Task](#).

Optionally, you can choose to select **Reveal secrets**. In combination with **TRACE**, this setting lets you see the sensitive data that the EventBridge connection inserts, such as API keys. The IAM user identity that you use to access the console must have permission to perform the `states:RevealSecrets` action. Without this permission, Step Functions throws an access denied error when you start the test. For an example of an IAM policy that sets the `states:RevealSecrets` permission, see [IAM permissions for using TestState API](#).

d. Choose **Start test**.

Testing a state using AWS CLI

You can test a [supported](#) state using the [TestState](#) API in the AWS CLI. This API accepts the definition of a state and executes it.

For each state, you can specify the amount of detail you want to view in the test results. These details provide additional information about the state's execution, including its input and output data processing result and HTTP request and response information. The following examples showcase the different inspection levels you can specify for the TestState API. Remember to replace the *italicized* text with your resource-specific information.

This section contains the following examples that describe how you can use the different inspection levels that Step Functions provides in the AWS CLI:

- [Using INFO inspectionLevel](#)
- [Using DEBUG inspectionLevel](#)

- [Using TRACE inspectionLevel](#)
- [Using jq utility in AWS CLI to filter and print the HTTP response that TestState API returns](#)

Example 1: Using INFO inspectionLevel to test a Choice state

To test a state using the INFO [inspectionLevel](#) in the AWS CLI, run the `test-state` command as shown in the following example.

```
aws stepfunctions test-state \  
  --definition '{"Type": "Choice", "Choices": [{"Variable": "$.number",  
"NumericEquals": 1, "Next": "Equals 1"}, {"Variable": "$.number", "NumericEquals": 2,  
"Next": "Equals 2"}], "Default": "No Match"}' \  
  --role-arn arn:aws:iam::123456789012:role/myRole \  
  --input '{"number": 2}'
```

This example uses a [Choice](#) state to determine the execution path for the state based on the numeric input you provide. By default, Step Functions sets the `inspectionLevel` to `INFO` if you don't set a level.

Step Functions returns the following output.

```
{  
  "output": "{\"number\": 2}",  
  "nextState": "Equals 2",  
  "status": "SUCCEEDED"  
}
```

Example 2: Using DEBUG inspectionLevel to debug input and output data processing in a Pass state

To test a state using the `DEBUG` [inspectionLevel](#) in the AWS CLI, run the `test-state` command as shown in the following example.

```
aws stepfunctions test-state \  
  --definition '{"Type": "Pass", "InputPath": "$.payload", "Parameters": {"data": 1},  
"ResultPath": "$.result", "OutputPath": "$.result.data", "Next": "Another State"}' \  
  --role-arn arn:aws:iam::123456789012:role/myRole \  
  --input '{"payload": {"foo": "bar"}}' \  
  --inspection-level DEBUG
```

This example uses a [Pass](#) state to showcase how Step Functions filters and manipulates input JSON data using the input and output data processing filters. This example uses these filters: [InputPath](#), [Parameters](#), [ResultPath](#), and [OutputPath](#).

Step Functions returns the following output.

```
{
  "output": "1",
  "inspectionData": {
    "input": "{\"payload\": {\"foo\": \"bar\"}}",
    "afterInputPath": "{\"foo\": \"bar\"}",
    "afterParameters": "{\"data\":1}",
    "afterResultSelector": "{\"data\":1}",
    "afterResultPath": "{\"payload\":{\"foo\": \"bar\"}, \"result\":{\"data\":1}}"
  },
  "nextState": "Another State",
  "status": "SUCCEEDED"
}
```

Example 3: Using TRACE inspectionLevel and revealSecrets to inspect the HTTP request sent to a third-party API

To test an [HTTP Task](#) using the TRACE [inspectionLevel](#) along with the [revealSecrets](#) parameter in the AWS CLI, run the `test-state` command as shown in the following example.

```
aws stepfunctions test-state \
  --definition '{"Type": "Task", "Resource": "arn:aws:states:::http:invoke",
  "Parameters": {"Method": "GET", "Authentication": {"ConnectionArn":
  "arn:aws:events:us-
  east-1:123456789012:connection/MyConnection/00000000-0000-0000-0000-000000000000"},
  "ApiEndpoint": "https://httpbin.org/get", "Headers": {"definitionHeader": "h1"},
  "RequestBody": {"message": "Hello from Step Functions!"}, "QueryParameters":
  {"queryParam": "q1"}}, "End": true}' \
  --role-arn arn:aws:iam:123456789012:role/myRole \
  --inspection-level TRACE \
  --reveal-secrets
```

This example tests if the HTTP Task calls the specified third-party API, `https://httpbin.org/`. It also shows the HTTP request and response data for the API call.

```
{
```

```

    "output": "{ \"Headers\": { \"date\": [ \"Tue, 21 Nov 2023 00:06:17 GMT\" ],
    \"access-control-allow-origin\": [ \"*\" ], \"content-length\": [ \"620\" ], \"server\":
    [ \"unicorn/19.9.0\" ], \"access-control-allow-credentials\": [ \"true\" ], \"content-
    type\": [ \"application/json\" ] }, \"ResponseBody\": { \"args\": { \"QueryParam1\":
    \"QueryParamValue1\", \"queryParams\": { \"q1\" }, \"headers\": { \"Authorization
    \": \"Basic XXXXXXXX\", \"Content-Type\": \"application/json; charset=UTF-8\",
    \"Customheader1\": \"CustomHeaderValue1\", \"Definitionheader\": \"h1\", \"Host\":
    \"httpbin.org\", \"Range\": \"bytes=0-262144\", \"Transfer-Encoding\": \"chunked\",
    \"User-Agent\": \"Amazon|StepFunctions|HttpInvoke|us-east-1\", \"X-Amzn-Trace-Id\":
    \"Root=1-00000000-0000-0000-0000-000000000000\", \"origin\": \"12.34.567.891\", \"url\":
    \"https://httpbin.org/get?queryParams=q1&QueryParam1=QueryParamValue1\" }, \"StatusCode
    \": 200, \"StatusText\": \"OK\" } }",
    "inspectionData": {
        "input": "{}",
        "afterInputPath": "{}",
        "afterParameters": "{ \"Method\": \"GET\", \"Authentication\": { \"ConnectionArn
        \": \"arn:aws:events:us-east-1:123456789012:connection/foo/a59c10f0-a315-4c1f-
        be6a-559b9a0c6250\" }, \"ApiEndpoint\": \"https://httpbin.org/get\", \"Headers\":
        { \"definitionHeader\": \"h1\" }, \"RequestBody\": { \"message\": \"Hello from Step Functions!
        \" }, \"QueryParameters\": { \"queryParams\": { \"q1\" } } }",
        "result": "{ \"Headers\": { \"date\": [ \"Tue, 21 Nov 2023 00:06:17 GMT\" ],
        \"access-control-allow-origin\": [ \"*\" ], \"content-length\": [ \"620\" ], \"server\":
        [ \"unicorn/19.9.0\" ], \"access-control-allow-credentials\": [ \"true\" ], \"content-
        type\": [ \"application/json\" ] }, \"ResponseBody\": { \"args\": { \"QueryParam1\":
        \"QueryParamValue1\", \"queryParams\": { \"q1\" }, \"headers\": { \"Authorization
        \": \"Basic XXXXXXXX\", \"Content-Type\": \"application/json; charset=UTF-8\",
        \"Customheader1\": \"CustomHeaderValue1\", \"Definitionheader\": \"h1\", \"Host\":
        \"httpbin.org\", \"Range\": \"bytes=0-262144\", \"Transfer-Encoding\": \"chunked\",
        \"User-Agent\": \"Amazon|StepFunctions|HttpInvoke|us-east-1\", \"X-Amzn-Trace-Id\":
        \"Root=1-00000000-0000-0000-0000-000000000000\", \"origin\": \"12.34.567.891\", \"url\":
        \"https://httpbin.org/get?queryParams=q1&QueryParam1=QueryParamValue1\" }, \"StatusCode
        \": 200, \"StatusText\": \"OK\" } }",
        "afterResultSelector": "{ \"Headers\": { \"date\": [ \"Tue, 21 Nov 2023
        00:06:17 GMT\" ], \"access-control-allow-origin\": [ \"*\" ], \"content-length\":
        [ \"620\" ], \"server\": [ \"unicorn/19.9.0\" ], \"access-control-allow-credentials
        \": [ \"true\" ], \"content-type\": [ \"application/json\" ] }, \"ResponseBody\": { \"args
        \": { \"QueryParam1\": \"QueryParamValue1\", \"queryParams\": { \"q1\" }, \"headers\":
        { \"Authorization\": \"Basic XXXXXXXX\", \"Content-Type\": \"application/json;
        charset=UTF-8\", \"Customheader1\": \"CustomHeaderValue1\", \"Definitionheader\": \"h1\",
        \"Host\": \"httpbin.org\", \"Range\": \"bytes=0-262144\", \"Transfer-Encoding\": \"chunked
        \", \"User-Agent\": \"Amazon|StepFunctions|HttpInvoke|us-east-1\", \"X-Amzn-Trace-Id\":
        \"Root=1-00000000-0000-0000-0000-000000000000\", \"origin\": \"12.34.567.891\", \"url\":
        \"https://httpbin.org/get?queryParams=q1&QueryParam1=QueryParamValue1\" }, \"StatusCode
        \": 200, \"StatusText\": \"OK\" } }",

```

```

    "afterResultPath": "{\\"Headers\\":{\\"date\\":[\\"Tue, 21 Nov 2023 00:06:17
    GMT\\"],\\"access-control-allow-origin\\":[\\"*\\"],\\"content-length\\":[\\"620\\"],
    \\"server\\":[\\"unicorn/19.9.0\\"],\\"access-control-allow-credentials\\":[\\"true\\"],
    \\"content-type\\":[\\"application/json\\"],\\"ResponseBody\\":{\\"args\\":{\\"QueryParam1\\":
    \\"QueryParamValue1\\",\\"queryParams\\":{\\"q1\\"},\\"headers\\":{\\"Authorization\\":
    \\"Basic XXXXXXXX\\",\\"Content-Type\\":{\\"application/json; charset=UTF-8\\"},
    \\"Customheader1\\":{\\"CustomHeaderValue1\\"},\\"Definitionheader\\":{\\"h1\\"},\\"Host\\":
    \\"httpbin.org\\"},\\"Range\\":{\\"bytes=0-262144\\"},\\"Transfer-Encoding\\":{\\"chunked\\"},
    \\"User-Agent\\":{\\"Amazon|StepFunctions|HttpInvoke|us-east-1\\"},\\"X-Amzn-Trace-Id\\":
    \\"Root=1-00000000-0000-0000-0000-000000000000\\"},\\"origin\\":{\\"12.34.567.891\\"},\\"url\\":
    \\"https://httpbin.org/get?queryParams=q1&QueryParam1=QueryParamValue1\\"},\\"StatusCode
    \":200,\\"StatusText\\":{\\"OK\\"}"}",
    "request": {
      "protocol": "https",
      "method": "GET",
      "url": "https://httpbin.org/get?
queryParams=q1&QueryParam1=QueryParamValue1",
      "headers": "[definitionHeader: h1, Authorization: Basic XXXXXXXX,
      CustomHeader1: CustomHeaderValue1, User-Agent: Amazon|StepFunctions|HttpInvoke|us-
      east-1, Range: bytes=0-262144]",
      "body": "{\\"message\\":{\\"Hello from Step Functions!\\"},\\"BodyKey1\\":
    \\"BodyValue1\\"}"
    },
    "response": {
      "protocol": "https",
      "statusCode": "200",
      "statusMessage": "OK",
      "headers": "[date: Tue, 21 Nov 2023 00:06:17 GMT, content-type:
      application/json, content-length: 620, server: unicorn/19.9.0, access-control-allow-
      origin: *, access-control-allow-credentials: true]",
      "body": "{\n  \\"args\\": {\n    \\"QueryParam1\\": \\"QueryParamValue1\\", \n
    \\"queryParams\\": \\"q1\\\"\n  }, \n  \\"headers\\": {\n    \\"Authorization\\": \\"Basic
    XXXXXXXX\\", \n    \\"Content-Type\\": \\"application/json; charset=UTF-8\\", \n
    \\"Customheader1\\": \\"CustomHeaderValue1\\", \n    \\"Definitionheader\\": \\"h1\\", \n
    \\"Host\\": \\"httpbin.org\\", \n    \\"Range\\": \\"bytes=0-262144\\", \n    \\"Transfer-
    Encoding\\": \\"chunked\\", \n    \\"User-Agent\\": \\"Amazon|StepFunctions|HttpInvoke|us-
    east-1\\", \n    \\"X-Amzn-Trace-Id\\": \\"Root=1-00000000-0000-0000-0000-000000000000\\\"\n
    }, \n  \\"origin\\": \\"12.34.567.891\\", \n  \\"url\\": \\"https://httpbin.org/get?
    queryParams=q1&QueryParam1=QueryParamValue1\\\"\n}\n"
    }
  },
  "status": "SUCCEEDED"
}

```

Example 4: Using jq utility to filter and print the response that TestState API returns

The TestState API returns JSON data as escaped strings in its response. The following AWS CLI example extends [Example 3](#) and uses the jq utility to filter and print the HTTP response that the TestState API returns in a human-readable format. For information about jq and its installation instructions, see [jq](#) on *GitHub*.

```
aws stepfunctions test-state \
  --definition '{"Type": "Task", "Resource": "arn:aws:states:::http:invoke",
  "Parameters": {"Method": "GET", "Authentication": {"ConnectionArn":
  "arn:aws:events:us-
east-1:123456789012:connection/MyConnection/0000000-0000-0000-0000-000000000000"}},
  "ApiEndpoint": "https://httpbin.org/get", "Headers": {"definitionHeader": "h1"},
  "RequestBody": {"message": "Hello from Step Functions!"}, "QueryParameters":
  {"queryParams": "q1"}}, "End": true}' \
  --role-arn arn:aws:iam::123456789012:role/myRole \
  --inspection-level TRACE \
  --reveal-secrets \
  | jq '.inspectionData.response.body | fromjson'
```

The following example shows the output returned in a human-readable format.

```
{
  "args": {
    "QueryParam1": "QueryParamValue1",
    "queryParams": "q1"
  },
  "headers": {
    "Authorization": "Basic XXXXXXXX",
    "Content-Type": "application/json; charset=UTF-8",
    "Customheader1": "CustomHeaderValue1",
    "Definitionheader": "h1",
    "Host": "httpbin.org",
    "Range": "bytes=0-262144",
    "Transfer-Encoding": "chunked",
    "User-Agent": "Amazon|StepFunctions|HttpInvoke|us-east-1",
    "X-Amzn-Trace-Id": "Root=1-0000000-0000-0000-0000-000000000000"
  },
  "origin": "12.34.567.891",
  "url": "https://httpbin.org/get?queryParams=q1&QueryParam1=QueryParamValue1"
```

```
}
```

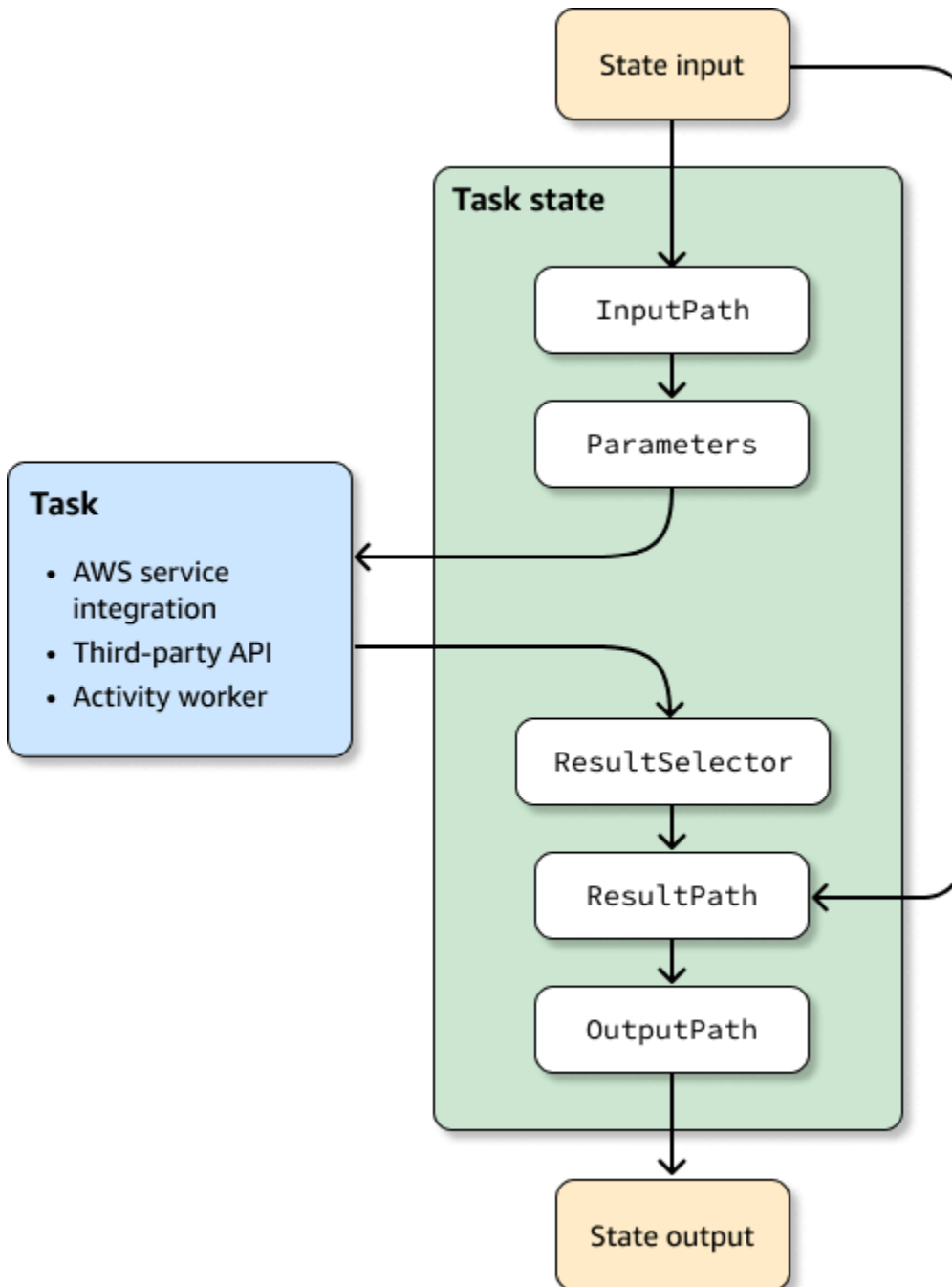
Testing and debugging input and output data flow

The `TestState` API is helpful for testing and debugging the data that flows through your workflow. This section provides some key concepts and explains how to use the `TestState` for this purpose.

Key concepts

In Step Functions, the process of filtering and manipulating JSON data as it passes through the states in your state machine is called *input and output processing*. For information about how this works, see [Input and Output Processing in Step Functions](#).

All the [state](#) types in the [Amazon States Language](#) (ASL) (Task, Parallel, Map, Pass, Wait, Choice, Succeed, and Fail) share a set of common fields for filtering and manipulating the JSON data that passes through them. These fields are: [InputPath](#), [Parameters](#), [ResultSelector](#), [ResultPath](#), and [OutputPath](#). Support for each field [varies across states](#). At runtime, Step Functions applies each field in a specific order. The following diagram shows the order in which these fields are applied to the data inside a Task state:



The following list describes the order of application of the input and output processing fields shown in the diagram.

1. *State input* is the JSON data passed to the current state from a previous state.
2. [InputPath](#) filters a portion of the raw state input.
3. [Parameters](#) configures the set of values to pass to the [Task](#).
4. The task performs work and returns a result.

5. [ResultSelector](#) selects a set of values to keep from the task result.
6. [ResultPath](#) combines the result with the raw state input, or replaces the result with it.
7. [OutputPath](#) filters a portion of the output to pass along to the next state.
8. *State output* is the JSON data passed from the current state to the next state.

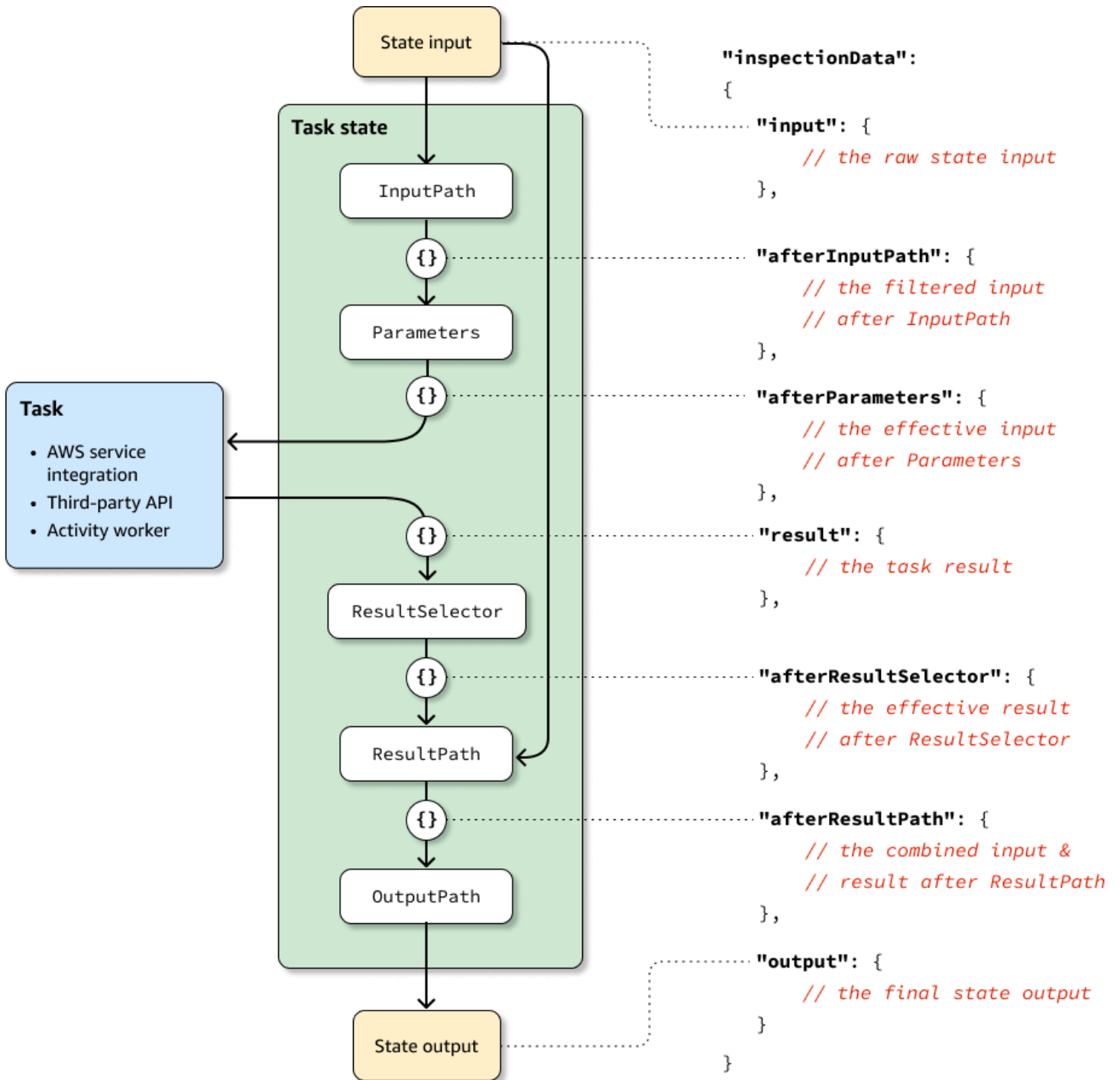
These input and output processing fields are optional. If you don't use any of these fields in your state definition, the task will consume the raw state input, and return the task result as the state output.

Using TestState to inspect input and output processing

When you call the TestState API and set the `inspectionLevel` parameter to `DEBUG`, the API response includes an object called `inspectionData`. This object contains fields to help you inspect how data was filtered or manipulated within the state when it was executed. The following example shows the `inspectionData` object for a Task state.

```
"inspectionData": {
  "input": string,
  "afterInputPath": string,
  "afterParameters": string,
  "result": string,
  "afterResultSelector": string,
  "afterResultPath": string,
  "output": string
}
```

In this example, each field that contains the `after` prefix, shows the data after a particular field was applied. For example, `afterInputPath` shows the effect of applying the `InputPath` field to filter the raw state input. The following diagram maps each [ASL definition](#) field to its corresponding field in the `inspectionData` object:



For examples of using the TestState API to debug input and output processing, see the following:

- [Testing a state using the DEBUG inspection level in the Step Functions console](#)
- [Testing a state using the DEBUG inspection level in the AWS CLI](#)

Testing state machines locally

AWS Step Functions Local is a downloadable version of Step Functions that lets you develop and test applications using a version of Step Functions running in your own development environment. The local version of Step Functions can invoke AWS Lambda functions, both in AWS and while running locally. You can also coordinate other [supported AWS services](#).

Note

Step Functions Local uses dummy accounts to work.

While running Step Functions Local, you can use one of the following ways to invoke service integrations:

- Configuring local endpoints for AWS Lambda and other services. For information about the supported endpoints, see [Setting Configuration Options for Step Functions Local](#).
- Making calls directly to an AWS service from Step Functions Local.
- Mocking the response from service integrations. For information about using mocked service integrations, see [Using Mocked Service Integrations](#).

AWS Step Functions Local is available as a JAR package or a self-contained Docker image that runs on Microsoft Windows, Linux, macOS, and other platforms that support Java or Docker.

Warning

The downloadable version of AWS Step Functions is intended to be used only for testing and should never be used to process sensitive information.

Tip

Make sure you're using Step Functions Local [version 1.12.0](#) or higher to be able to include all the [intrinsic functions](#) in your workflows.

The following topics describe how you can set up Step Functions Local using Docker and JAR file, and run Step Functions Local to work with AWS Lambda, AWS Serverless Application Model(AWS SAM) CLI Local, or other supported services.

Topics

- [Setting Up Step Functions Local \(Downloadable Version\) and Docker](#)
- [Setting Up Step Functions Local \(Downloadable Version\) - Java Version](#)
- [Setting Configuration Options for Step Functions Local](#)
- [Running Step Functions Local on Your Computer](#)
- [Testing Step Functions and AWS SAM CLI Local](#)
- [Using Mocked Service Integrations](#)

Setting Up Step Functions Local (Downloadable Version) and Docker

The Step Functions Local Docker image enables you to get started with Step Functions Local quickly by using a Docker image with all the needed dependencies. The Docker image enables you to include Step Functions Local in your containerized builds and as part of your continuous integration testing.

To get the Docker image for Step Functions Local, see <https://hub.docker.com/r/amazon/aws-stepfunctions-local>, or enter the following Docker pull command.

```
docker pull amazon/aws-stepfunctions-local
```

To start the downloadable version of Step Functions on Docker, run the following Docker run command

```
docker run -p 8083:8083 amazon/aws-stepfunctions-local
```

To interact with AWS Lambda or other supported services, you need to configure your credentials and other configuration options first. For more information, see the following topics:

- [Setting Configuration Options for Step Functions Local](#)
- [Credentials and configuration for Docker](#)

Setting Up Step Functions Local (Downloadable Version) - Java Version

The downloadable version of AWS Step Functions is provided as an executable JAR file and as a Docker image. The Java application runs on Windows, Linux, macOS, and other platforms that support Java. In addition to Java, you need to install the AWS Command Line Interface (AWS CLI). For information about installing and configuring the AWS CLI, see the [AWS Command Line Interface User Guide](#).

To set up and run Step Functions on your computer

1. Download Step Functions using the following links.

Download Links	Checksum
.tar.gz	.tar.gz.md5
.zip	.zip.md5

2. Extract the .zip file.
3. Test the download and view version information.

```
$ java -jar StepFunctionsLocal.jar -v
Step Function Local
Version: 1.0.0
Build: 2019-01-21
```

4. (Optional) View a listing of available commands.

```
$ java -jar StepFunctionsLocal.jar -h
```

5. To start Step Functions on your computer, open a command prompt, navigate to the directory where you extracted `StepFunctionsLocal.jar`, and enter the following command.

```
java -jar StepFunctionsLocal.jar
```

6. To access Step Functions running locally, use the `--endpoint-url` parameter. For example, using the AWS CLI, you would specify Step Functions commands as follows:

```
aws stepfunctions --endpoint-url http://localhost:8083 command
```

Note

By default, Step Functions Local uses a local test account and credentials, and the AWS Region is set to US East (N. Virginia). To use Step Functions Local with AWS Lambda, or other supported services, you must configure your credentials and Region.

If you use Express workflows with Step Functions Local, the execution history will be stored in a log file. It is not logged to CloudWatch Logs. The log file path will be based on the CloudWatch Logs log group ARN provided when you create the local state machine. The log file will be stored in `/aws/states/log-group-name/${execution_arn}.log` relative to the location where you are running Step Functions Local. For example, if the execution ARN is:

```
arn:aws:states:us-east-1:123456789012:express:test:example-ExpressLogGroup-wJalrXUtnFEMI
```

the log file will be:

```
aws/states/log-group-name/arn:aws:states:us-east-1:123456789012:express:test:example-ExpressLogGroup-wJalrXUtnFEMI.log
```

Setting Configuration Options for Step Functions Local

When you start AWS Step Functions Local by using the JAR file, you can set configuration options by using the AWS Command Line Interface (AWS CLI), or by including them in the system environment. For Docker, you must specify these options in a file that you reference when starting Step Functions Local.

Configuration Options

When you configure the Step Functions Local container to use an override endpoint such as Lambda Endpoint and Batch Endpoint, and make calls to that endpoint, Step Functions Local doesn't use the [credentials](#) you specify. Setting these endpoint overrides is optional.

Option	Command Line	Environment
Account	-account, --aws-account	AWS_ACCOUNT_ID

Option	Command Line	Environment
Region	-region, --aws-region	AWS_DEFAULT_REGION
Wait Time Scale	-waitTimeScale, --wait-time-scale	WAIT_TIME_SCALE
Lambda Endpoint	-lambdaEndpoint, --lambda-endpoint	LAMBDA_ENDPOINT
Batch Endpoint	-batchEndpoint, --batch-endpoint	BATCH_ENDPOINT
DynamoDB Endpoint	-dynamoDBEndpoint, --dynamodb-endpoint	DYNAMODB_ENDPOINT
ECS Endpoint	-ecsEndpoint, --ecs-endpoint	ECS_ENDPOINT
Glue Endpoint	-glueEndpoint, --glue-endpoint	GLUE_ENDPOINT
SageMaker Endpoint	-sageMakerEndpoint, --sagemaker-endpoint	SAGE_MAKER_ENDPOINT
SQS Endpoint	-sqsEndpoint, --sqs-endpoint	SQS_ENDPOINT
SNS Endpoint	-snsEndpoint, --sns-endpoint	SNS_ENDPOINT
Step Functions Endpoint	-stepFunctionsEndpoint, --step-functions-endpoint	STEP_FUNCTIONS_ENDPOINT

Credentials and configuration for Docker

To configure Step Functions Local for Docker, create the following file: `aws-stepfunctions-local-credentials.txt`.

This file contains your credentials and other configuration options. The following can be used as a template when creating the `aws-stepfunctions-local-credentials.txt` file.

```
AWS_DEFAULT_REGION=AWS_REGION_OF_YOUR_AWS_RESOURCES
```



```
AWS_ACCESS_KEY_ID=YOUR_AWS_ACCESS_KEY
AWS_SECRET_ACCESS_KEY=YOUR_AWS_SECRET_KEY
WAIT_TIME_SCALE=VALUE
LAMBDA_ENDPOINT=VALUE
BATCH_ENDPOINT=VALUE
DYNAMODB_ENDPOINT=VALUE
ECS_ENDPOINT=VALUE
GLUE_ENDPOINT=VALUE
SAGE_MAKER_ENDPOINT=VALUE
SQS_ENDPOINT=VALUE
SNS_ENDPOINT=VALUE
STEP_FUNCTIONS_ENDPOINT=VALUE
```

Once you have configured your credentials and configuration options in `aws-stepfunctions-local-credentials.txt`, start Step Functions with the following command.

```
docker run -p 8083:8083 --env-file aws-stepfunctions-local-credentials.txt amazon/aws-
stepfunctions-local
```

Note

It is recommended to use the special DNS name `host.docker.internal`, which resolves to the internal IP address that the host uses, such as `http://host.docker.internal:8000`. For more information, see Docker documentation for Mac and Windows at [Networking features in Docker Desktop for Mac](#) and [Networking features in Docker Desktop for Windows](#) respectively.

Running Step Functions Local on Your Computer

Use the local version of Step Functions to configure, develop and test state machines on your computer.

Run a HelloWorld state machine locally

After you run Step Functions locally with the AWS Command Line Interface (AWS CLI), you can start a state machine execution.

1. Create a state machine from the AWS CLI by escaping out the state machine definition.

```
aws stepfunctions --endpoint-url http://localhost:8083 create-state-machine --
definition "{\
  \"Comment\": \"A Hello World example of the Amazon States Language using a Pass
state\", \
  \"StartAt\": \"HelloWorld\", \
  \"States\": {\
    \"HelloWorld\": {\
      \"Type\": \"Pass\", \
      \"End\": true\
    }\
  }\
}" --name "HelloWorld" --role-arn "arn:aws:iam::012345678901:role/DummyRole"
```

Note

The `role-arn` is not used for Step Functions Local, but you must include it with the proper syntax. You can use the Amazon Resource Name (ARN) from the previous example.

If you successfully create the state machine, Step Functions responds with the creation date and the state machine ARN.

```
{
  "creationDate": 1548454198.202,
  "stateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:HelloWorld"
}
```

2. Start an execution using the ARN of the state machine you created.

```
aws stepfunctions --endpoint-url http://localhost:8083 start-execution --state-
machine-arn arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld
```

Step Functions Local with AWS SAM CLI Local

You can use the local version of Step Functions with a local version of AWS Lambda. To configure this, you must install and configure AWS SAM.

For information about configuring and running AWS SAM, see the following:

- [Set Up AWS SAM](#)
- [Start AWS SAM CLI Local](#)

When Lambda is running on your local system, you can start Step Functions Local. From the directory where you extracted your Step Functions local JAR files, start Step Functions Local and use the `--lambda-endpoint` parameter to configure the local Lambda endpoint.

```
java -jar StepFunctionsLocal.jar --lambda-endpoint http://127.0.0.1:3001 command
```

For more information about running Step Functions Local with AWS Lambda, see [Testing Step Functions and AWS SAM CLI Local](#).

Testing Step Functions and AWS SAM CLI Local

With both AWS Step Functions and AWS Lambda running on your local machine, you can test your state machine and Lambda functions without deploying your code to AWS.

For more information, see the following topics:

- [Testing state machines locally](#)
- [Set Up AWS SAM](#)

Topics

- [Step 1: Set Up AWS SAM](#)
- [Step 2: Test AWS SAM CLI Local](#)
- [Step 3: Start AWS SAM CLI Local](#)
- [Step 4: Start Step Functions Local](#)
- [Step 5: Create a State Machine That References Your AWS SAM CLI Local Function](#)
- [Step 6: Start an Execution of Your Local State Machine](#)

Step 1: Set Up AWS SAM

AWS Serverless Application Model (AWS SAM) CLI Local requires the AWS Command Line Interface, AWS SAM, and Docker to be installed.

1. [Install the AWS SAM CLI](#).

Note

Before installing the AWS SAM CLI, you need to install the AWS CLI and Docker. See the [Prerequisites](#) for installing the AWS SAM CLI.

2. Go through the [AWS SAM Quick Start](#) documentation. Be sure to follow the steps to do the following:
 1. [Initialize the Application](#)
 2. [Test the Application Locally](#)

This creates a `sam-app` directory, and builds an environment that includes a Python-based Hello World Lambda function.

Step 2: Test AWS SAM CLI Local

Now that you have installed AWS SAM and created the Hello World Lambda function, you can test the function. In the `sam-app` directory, enter the following command:

```
sam local start-api
```

This launches a local instance of your Lambda function. You should see output similar to the following:

```
2019-01-31 16:40:27 Found credentials in shared credentials file: ~/.aws/credentials
2019-01-31 16:40:27 Mounting HelloWorldFunction at http://127.0.0.1:3000/hello [GET]
2019-01-31 16:40:27 You can now browse to the above endpoints to invoke your functions.
  You do not need to restart/reload SAM CLI while working on your functions changes will
  be reflected instantly/automatically. You only need to restart SAM CLI if you update
  your AWS SAM template
2019-01-31 16:40:27 * Running on http://127.0.0.1:3000/ (Press CTRL+C to quit)
```

Open a browser and enter the following:

```
http://127.0.0.1:3000/hello
```

This will output a response similar to the following:

```
{"message": "hello world", "location": "72.21.198.66"}
```

Enter **CTRL+C** to end the Lambda API.

Step 3: Start AWS SAM CLI Local

Now that you've tested that the function works, start AWS SAM CLI Local. In the `sam-app` directory, enter the following command:

```
sam local start-lambda
```

This starts AWS SAM CLI Local and provides the endpoint to use, similar to the following output:

```
2019-01-29 15:33:32 Found credentials in shared credentials file: ~/.aws/credentials
2019-01-29 15:33:32 Starting the Local Lambda Service. You can now invoke your Lambda
  Functions defined in your template through the endpoint.
2019-01-29 15:33:32 * Running on http://127.0.0.1:3001/ (Press CTRL+C to quit)
```

Step 4: Start Step Functions Local

JAR File

If you're using the `.jar` file version of Step Functions Local, start Step Functions and specify the Lambda endpoint. In the directory where you extracted the `.jar` files, enter the following command:

```
java -jar StepFunctionsLocal.jar --lambda-endpoint http://localhost:3001
```

When Step Functions Local starts, it checks the environment, and then the credentials configured in your `~/.aws/credentials` file. By default, it starts using a fictitious user ID, and is listed as `region us-east-1`.

```
2019-01-29 15:38:06.324: Failed to load credentials from environment because Unable to
  load AWS credentials from environment variables (AWS_ACCESS_KEY_ID (or AWS_ACCESS_KEY)
  and AWS_SECRET_KEY (or AWS_SECRET_ACCESS_KEY))
2019-01-29 15:38:06.326: Loaded credentials from profile: default
2019-01-29 15:38:06.326: Starting server on port 8083 with account 123456789012, region
  us-east-1
```

Docker

If you're using the Docker version of Step Functions Local, launch Step Functions with the following command:

```
docker run -p 8083:8083 amazon/aws-stepfunctions-local
```

For information about installing the Docker version of Step Functions, see [Setting Up Step Functions Local \(Downloadable Version\) and Docker](#).

Note

You can specify the endpoint through the command line or by setting environment variables if you launch Step Functions from the `.jar` file. For the Docker version, you must specify the endpoints and credentials in a text file. See [Setting Configuration Options for Step Functions Local](#).

Step 5: Create a State Machine That References Your AWS SAM CLI Local Function

Once Step Functions Local is running, create a state machine that references the `HelloWorldFunction` that you initialized in [Step 1: Set Up AWS SAM](#).

```
aws stepfunctions --endpoint http://localhost:8083 create-state-machine --definition
"{\
  \"Comment\": \"A Hello World example of the Amazon States Language using an AWS
  Lambda Local function\", \
  \"StartAt\": \"HelloWorld\", \
  \"States\": {\
    \"HelloWorld\": {\
      \"Type\": \"Task\", \
      \"Resource\": \"arn:aws:lambda:us-east-1:123456789012:function:HelloWorldFunction
  \", \
      \"End\": true\
    }\
  }\
}" --name "HelloWorld" --role-arn "arn:aws:iam::012345678901:role/DummyRole"
```

This will create a state machine and provide an Amazon Resource Name (ARN) that you can use to start an execution.

```
{
  "creationDate": 1548805711.403,
  "stateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine>HelloWorld"
}
```

Step 6: Start an Execution of Your Local State Machine

Once you have created a state machine, start an execution. You'll need to reference the endpoint and state machine ARN when using the following `aws stepfunctions` command:

```
aws stepfunctions --endpoint http://localhost:8083 start-execution --state-machine
arn:aws:states:us-east-1:123456789012:stateMachine>HelloWorld --name test
```

This starts an execution named `test` of your `HelloWorld` state machine.

```
{
  "startDate": 1548810641.52,
  "executionArn": "arn:aws:states:us-east-1:123456789012:execution>HelloWorld:test"
}
```

Now that Step Functions is running locally, you can interact with it using the AWS CLI. For example, to get information about this execution, use the following command:

```
aws stepfunctions --endpoint http://localhost:8083 describe-execution --execution-arn
arn:aws:states:us-east-1:123456789012:execution>HelloWorld:test
```

Calling `describe-execution` for an execution provides more complete details, similar to the following output:

```
{
  "status": "SUCCEEDED",
  "startDate": 1549056334.073,
  "name": "test",
  "executionArn": "arn:aws:states:us-east-1:123456789012:execution>HelloWorld:test",
  "stateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine>HelloWorld",
  "stopDate": 1549056351.276,
  "output": "{\"statusCode\": 200, \"body\": \"{\\\\"message\\\\": \\\\"hello world\\\\"\",
\\\\"location\\\\": \\\\"72.21.198.64\\\\"}\"}",
  "input": "{}"
```

```
}
```

Using Mocked Service Integrations

In Step Functions Local, you can test the execution paths of your state machines without actually calling integrated services by using mocked service integrations. To configure your state machines to use mocked service integrations, you create a mock configuration file. In this file, you define the desired output of your service integrations as mocked responses and the executions which use your mocked responses to simulate an execution path as test cases.

By providing the mock configuration file to Step Functions Local, you can test service integration calls by running state machines that use the mocked responses specified in the test cases instead of making actual service integration calls.

Note

If you don't specify mocked service integration responses in the mock configuration file, Step Functions Local will invoke the AWS service integration using the endpoint you configured while setting up Step Functions Local. For information about configuring endpoints for Step Functions Local, see [Setting Configuration Options for Step Functions Local](#).

Topics

- [Key concepts in this topic](#)
- [Step 1: Specify Mocked Service Integrations in a Mock Configuration File](#)
- [Step 2: Provide the Mock Configuration File to Step Functions Local](#)
- [Step 3: Run Mocked Service Integration Tests](#)
- [Configuration File for Mocked Service Integrations](#)

Key concepts in this topic

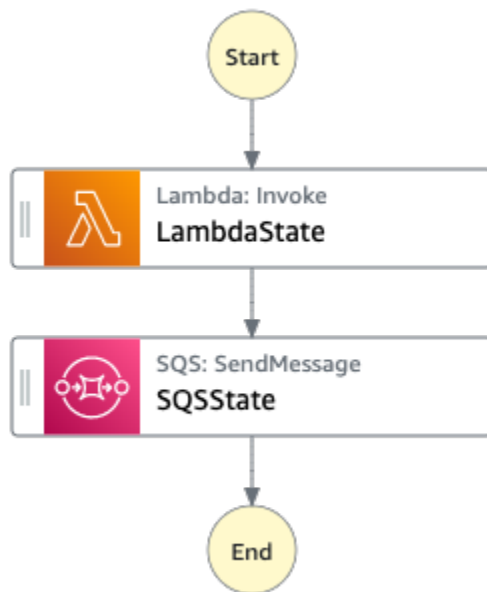
This topic uses several concepts which are defined in the following list:

- **Mocked Service Integrations** - Refers to Task states configured to use mocked responses instead of performing actual service calls.

- Mocked Responses - Refers to mock data that Task states can be configured to use.
- Test Cases - Refers to state machine executions configured to use mocked service integrations.
- Mock Configuration File - Refers to mock configuration file that contains JSON, which defines mocked service integrations, mocked responses, and test cases.

Step 1: Specify Mocked Service Integrations in a Mock Configuration File

You can test Step Functions AWS SDK and optimized service integrations using Step Functions Local. The following image shows the state machine defined in the State machine definition tab:



To do this, you must create a mock configuration file containing sections as defined in [Introducing structure of mock configuration](#).

1. Create a file named `MockConfigFile.json` to configure tests with mocked service integrations.

The following example shows a mock configuration file referencing a state machine with two defined states named `LambdaState` and `SQSState`.

Mock configuration file example

The following is an example of a mock configuration file which demonstrates how to mock responses from [invoking a Lambda function](#) and [sending a message to Amazon SQS](#). In this example, the [LambdaSQSIntegration](#) state machine contains three test

cases named HappyPath, RetryPath, and HybridPath which mock the Task states named LambdaState and SQSState. These states use the MockedLambdaSuccess, MockedSQSSuccess, and MockedLambdaRetry mocked service responses. These mocked service responses are defined in the MockedResponses section of the file.

```
{
  "StateMachines":{
    "LambdaSQSIntegration":{
      "TestCases":{
        "HappyPath":{
          "LambdaState":"MockedLambdaSuccess",
          "SQSState":"MockedSQSSuccess"
        },
        "RetryPath":{
          "LambdaState":"MockedLambdaRetry",
          "SQSState":"MockedSQSSuccess"
        },
        "HybridPath":{
          "LambdaState":"MockedLambdaSuccess"
        }
      }
    }
  },
  "MockedResponses":{
    "MockedLambdaSuccess":{
      "0":{
        "Return":{
          "StatusCode":200,
          "Payload":{
            "StatusCode":200,
            "body":"Hello from Lambda!"
          }
        }
      }
    },
    "LambdaMockedResourceNotReady":{
      "0":{
        "Throw":{
          "Error":"Lambda.ResourceNotReadyException",
          "Cause":"Lambda resource is not ready."
        }
      }
    }
  },
}
```



```

"States":{
  "LambdaState":{
    "Type":"Task",
    "Resource":"arn:aws:states:::lambda:invoke",
    "Parameters":{
      "Payload.$":"$",
      "FunctionName":"HelloWorldFunction"
    },
    "Retry":[
      {
        "ErrorEquals":[
          "States.ALL"
        ],
        "IntervalSeconds":2,
        "MaxAttempts":3,
        "BackoffRate":2
      }
    ],
    "Next":"SQSState"
  },
  "SQSState":{
    "Type":"Task",
    "Resource":"arn:aws:states:::sqs:sendMessage",
    "Parameters":{
      "QueueUrl":"https://sqs.us-east-1.amazonaws.com/123456789012/myQueue",
      "MessageBody.$":"$"
    },
    "End": true
  }
}
}

```

You can run the `LambdaSQSIntegration` state machine definition referenced in the mock configuration file using one of the following test cases:

- **HappyPath** - This test mocks the output of `LambdaState` and `SQSState` using `MockedLambdaSuccess` and `MockedSQSSuccess` respectively.
- The `LambdaState` will return the following value:

```

"0":{
  "Return":{

```

```

    "StatusCode":200,
    "Payload":{
      "StatusCode":200,
      "body":"Hello from Lambda!"
    }
  }
}

```

- The `SQSState` will return the following value:

```

"0":{
  "Return":{
    "MD5OfMessageBody":"3bcb6e8e-7h85-4375-b0bc-1a59812c6e51",
    "MessageId":"3bcb6e8e-8b51-4375-b0bc-1a59812c6e51"
  }
}

```

- `RetryPath` - This test mocks the output of `LambdaState` and `SQSState` using `MockedLambdaRetry` and `MockedSQSSuccess` respectively. In addition, `LambdaState` is configured to perform four retry attempts. The mocked responses for these attempts are defined and indexed in the `MockedLambdaRetry` state.
- The initial attempt ends with a task failure containing a cause and error message as shown in the following example:

```

"0":{
  "Throw": {
    "Error": "Lambda.ResourceNotReadyException",
    "Cause": "Lambda resource is not ready."
  }
}

```

- The first and second retry attempts end with a task failure containing a cause and error message as shown in the following example:

```

"1-2":{
  "Throw": {
    "Error": "Lambda.TimeoutException",
    "Cause": "Lambda timed out."
  }
}

```

- The third retry attempt ends with a task success containing state result from Payload section in the mocked Lambda response.

```
"3":{
  "Return": {
    "StatusCode": 200,
    "Payload": {
      "StatusCode": 200,
      "body": "Hello from Lambda!"
    }
  }
}
```

Note

- For states with a retry policy, Step Functions Local will exhaust the retry attempts set in the policy until it receives a success response. This means that you must denote mocks for retries with consecutive attempt numbers and should cover all the retry attempts before returning a success response.
- If you do not specify a mocked response for a specific retry attempt, for example, retry "3", the state machine execution will fail.

- HybridPath - This test mocks the output of LambdaState. After LambdaState runs successfully and receives mocked data as a response, SQSState performs an actual service call to the resource specified in production.

For information about how to start test executions with mocked service integrations, see [Step 3: Run Mocked Service Integration Tests](#).

2. Make sure that the mocked responses' structure conforms to the structure of actual service responses you receive when you make integrated service calls. For information about the structural requirements for mocked responses, see [Configuring mocked service integrations](#).

In the previous example mock configuration file, the mocked responses defined in MockedLambdaSuccess and MockedLambdaRetry conform to the structure of actual responses that are returned from calling HelloFromLambda.

⚠ Important

AWS service responses can vary in structure between different services. Step Functions Local doesn't validate if mocked response structures conform to actual service response structures. You must ensure that your mocked responses conform to actual responses before testing. To review the structure of service responses, you can either perform the actual service calls using Step Functions or view the documentation for those services.

Step 2: Provide the Mock Configuration File to Step Functions Local

You can provide the mock configuration file to Step Functions Local in one of the following ways:

Docker

📘 Note

If you're using the Docker version of Step Functions Local, you can provide the mock configuration file using an environment variable only. In addition, you must mount the mock configuration file onto the Step Functions Local container at the initial server boot-up.

Mount the mock configuration file onto any directory within the Step Functions Local container. Then, set an environment variable named `SFN MOCK CONFIG` that contains the path to the mock configuration file in the container. This method enables the mock configuration file to be named anything as long as the environment variable contains the file path and name.

The following command shows the format to start the Docker image.

```
docker run -p 8083:8083
--mount type=bind,readonly,source={absolute path to mock config file},destination=/
home/StepFunctionsLocal/MockConfigFile.json
-e SFN_MOCK_CONFIG="/home/StepFunctionsLocal/MockConfigFile.json" amazon/aws-
stepfunctions-local
```

The following example uses the command to start the Docker image.

```
docker run -p 8083:8083
--mount type=bind,readonly,source=/Users/admin/Desktop/workplace/
MockConfigFile.json,destination=/home/StepFunctionsLocal/MockConfigFile.json
-e SFN MOCK_CONFIG="/home/StepFunctionsLocal/MockConfigFile.json" amazon/aws-
stepfunctions-local
```

JAR File

Use one of the following ways to provide the mock configuration file to Step Functions Local:

- Place the mock configuration file in the same directory as `Step FunctionsLocal.jar`. When using this method, you must name the mock configuration file `MockConfigFile.json`.
- In the session running Step Functions Local, set an environment variable named `SFN MOCK_CONFIG`, to the full path of the mock configuration file. This method enables the mock configuration file to be named anything as long as the environment variable contains its file path and name. In the following example, the `SFN MOCK_CONFIG` variable is set to point at a mock configuration file named `EnvSpecifiedMockConfig.json`, located in the `/home/workspace` directory.

```
export SFN MOCK_CONFIG="/home/workspace/EnvSpecifiedMockConfig.json"
```

Note

- If you do not provide the environment variable `SFN MOCK_CONFIG` to Step Functions Local, by default, it will attempt to read a mock configuration file named `MockConfigFile.json` in the directory from which you launched Step Functions Local.
- If you place the mock configuration file in the same directory as `Step FunctionsLocal.jar` and set the environment variable `SFN MOCK_CONFIG`, Step Functions Local will read the file specified by the environment variable.

Step 3: Run Mocked Service Integration Tests

After you create and provide a mock configuration file to Step Functions Local, run the state machine configured in the mock configuration file using mocked service integrations. Then check the execution results using an API action.

1. Create a state machine based on the previously mentioned definition in the [mock configuration file](#).

```
aws stepfunctions create-state-machine \
  --endpoint http://localhost:8083 \
  --definition '{"Comment\":\"Thisstatemachineiscalled:LambdaSQSIntegration
\", \"StartAt\":\"LambdaState\", \"States\":{\"LambdaState\":{\"Type\":
\"Task\", \"Resource\":\"arn:aws:states:::lambda:invoke\", \"Parameters
\":{\"Payload.$\":\"$\", \"FunctionName\":\"arn:aws:lambda:us-
east-1:123456789012:function:HelloWorldFunction\"}, \"Retry\":[{\"ErrorEquals
\":[\"States.ALL\"], \"IntervalSeconds\":2, \"MaxAttempts\":3, \"BackoffRate
\":2}], \"Next\":\"SQSState\"}, \"SQSState\":{\"Type\":\"Task\", \"Resource\":
\"arn:aws:states:::sqs:sendMessage\", \"Parameters\":{\"QueueUrl\":\"https://
sqs.us-east-1.amazonaws.com/123456789012/myQueue\", \"MessageBody.$\":\"$\"}, \"End
\":true}}}' \
  --name "LambdaSQSIntegration" --role-arn "arn:aws:iam::123456789012:role/
service-role/LambdaSQSIntegration"
```

2. Run the state machine using mocked service integrations.

To use the mock configuration file, make a [StartExecution](#) API call on a state machine configured in the mock configuration file. To do this, append the suffix, *#test_name*, to the state machine ARN used by `StartExecution`. *test_name* is a test case, which is configured for the state machine in the same mock configuration file.

The following command is an example that uses the `LambdaSQSIntegration` state machine and mock configuration. In this example, the `LambdaSQSIntegration` state machine is executed using the `HappyPath` test defined in [Step 1: Specify Mocked Service Integrations in a Mock Configuration File](#). The `HappyPath` test contains the configuration for the execution to handle mock service integration calls that `LambdaState` and `SQSState` states make using the `MockedLambdaSuccess` and `MockedSQSSuccess` mocked service responses.

```
aws stepfunctions start-execution \
  --endpoint http://localhost:8083 \
  --name executionWithHappyPathMockedServices \
```

```
--state-machine arn:aws:states:us-
east-1:123456789012:stateMachine:LambdaSQSIntegration#HappyPath
```

3. View the state machine execution response.

The response to calling `StartExecution` using a mocked service integration test is same as the response to calling `StartExecution` normally, which returns the execution ARN and start date.

The following is an example response to calling `StartExecution` using the mocked service integration test:

```
{
  "startDate": "2022-01-28T15:03:16.981000-05:00",
  "executionArn": "arn:aws:states:us-
east-1:123456789012:execution:LambdaSQSIntegration:executionWithHappyPathMockedServices"
}
```

4. Check the execution's results by making a [ListExecutions](#), [DescribeExecution](#), or [GetExecutionHistory](#) API call.

```
aws stepfunctions get-execution-history \
  --endpoint http://localhost:8083 \
  --execution-arn arn:aws:states:us-
east-1:123456789012:execution:LambdaSQSIntegration:executionWithHappyPathMockedServices
```

The following example demonstrates parts of a response to calling `GetExecutionHistory` using the execution ARN from the example response shown in step 2. In this example, the output of `LambdaState` and `SQSState` is the mock data defined in `MockedLambdaSuccess` and `MockedSQSSuccess` in the [mock configuration file](#). In addition, the mocked data is used the same way that data returned by performing actual service integration calls would be used. Also, in this example, the output from `LambdaState` is passed onto `SQSState` as input.

```
{
  "events": [
    ...
    {
      "timestamp": "2021-12-02T19:39:48.988000+00:00",
      "type": "TaskStateEntered",
      "id": 2,
      "previousEventId": 0,
```

```

        "stateEnteredEventDetails": {
            "name": "LambdaState",
            "input": "{}",
            "inputDetails": {
                "truncated": false
            }
        }
    },
    ...
    {
        "timestamp": "2021-11-25T23:39:10.587000+00:00",
        "type": "LambdaFunctionSucceeded",
        "id": 5,
        "previousEventId": 4,
        "lambdaFunctionSucceededEventDetails": {
            "output": "{\"statusCode\":200,\"body\":\"Hello from Lambda!\"",
            "outputDetails": {
                "truncated": false
            }
        }
    },
    ...
    {
        "timestamp": "2021-12-02T19:39:49.464000+00:00",
        "type": "TaskStateEntered",
        "id": 7,
        "previousEventId": 6,
        "stateEnteredEventDetails": {
            "name": "SQSState",
            "input": "{\"statusCode\":200,\"body\":\"Hello from Lambda!\"",
            "inputDetails": {
                "truncated": false
            }
        }
    },
    ...
    {
        "timestamp": "2021-11-25T23:39:10.652000+00:00",
        "type": "TaskSucceeded",
        "id": 10,
        "previousEventId": 9,
        "taskSucceededEventDetails": {
            "resourceType": "sqs",

```

```
        "resource": "sendMessage",
        "output": "{\"MD50fMessageBody\": \"3bcb6e8e-7h85-4375-
b0bc-1a59812c6e51\", \"MessageId\": \"3bcb6e8e-8b51-4375-b0bc-1a59812c6e51\"}\",
        "outputDetails": {
            "truncated": false
        }
    },
    ...
]
```

Configuration File for Mocked Service Integrations

To use mocked service integrations, you must first create a mock configuration file named `MockConfigFile.json` containing your mock configurations. Then provide Step Functions Local with the mock configuration file. This configuration file defines test cases, which contain mock states that use mocked service integration responses. The following section contains information about the structure of mock configuration that includes the mock states and mocked responses:

Topics

- [Introducing structure of mock configuration](#)
- [Configuring mocked service integrations](#)

Introducing structure of mock configuration

A mock configuration is a JSON object containing the following top-level fields:

- `StateMachines` - The fields of this object represent state machines configured to use mocked service integrations.
- `MockedResponse` - The fields of this object represent mocked responses for service integration calls.

The following is an example of a mock configuration file which includes a `StateMachine` definition and `MockedResponse`.

```
{
  "StateMachines":{
```

```

    "LambdaSQSIntegration":{
      "TestCases":{
        "HappyPath":{
          "LambdaState":"MockedLambdaSuccess",
          "SQSState":"MockedSQSSuccess"
        },
        "RetryPath":{
          "LambdaState":"MockedLambdaRetry",
          "SQSState":"MockedSQSSuccess"
        },
        "HybridPath":{
          "LambdaState":"MockedLambdaSuccess"
        }
      }
    },
    "MockedResponses":{
      "MockedLambdaSuccess":{
        "0":{
          "Return":{
            "StatusCode":200,
            "Payload":{
              "StatusCode":200,
              "body":"Hello from Lambda!"
            }
          }
        }
      },
      "LambdaMockedResourceNotReady":{
        "0":{
          "Throw":{
            "Error":"Lambda.ResourceNotReadyException",
            "Cause":"Lambda resource is not ready."
          }
        }
      },
      "MockedSQSSuccess":{
        "0":{
          "Return":{
            "MD5OfMessageBody":"3bcb6e8e-7h85-4375-b0bc-1a59812c6e51",
            "MessageId":"3bcb6e8e-8b51-4375-b0bc-1a59812c6e51"
          }
        }
      }
    },
  },

```

```
"MockedLambdaRetry":{
  "0":{
    "Throw":{
      "Error":"Lambda.ResourceNotReadyException",
      "Cause":"Lambda resource is not ready."
    }
  },
  "1-2":{
    "Throw":{
      "Error":"Lambda.TimeoutException",
      "Cause":"Lambda timed out."
    }
  },
  "3":{
    "Return":{
      "StatusCode":200,
      "Payload":{
        "StatusCode":200,
        "body":"Hello from Lambda!"
      }
    }
  }
}
}
```

Mock configuration field reference

The following sections explain the top-level object fields that you must define in your mock configuration.

- [StateMachines](#)
- [MockedResponses](#)

StateMachines

The StateMachines object defines which state machines will use mocked service integrations. The configuration for each state machine is represented as a top-level field of StateMachines. The field name is the name of the state machine and value is an object containing a single field named TestCases, whose fields represent test cases of that state machine.

The following syntax shows a state machine with two test cases:

```
"MyStateMachine": {
  "TestCases": {
    "HappyPath": {
      ...
    },
    "SadPath": {
      ...
    }
  }
}
```

TestCases

The fields of `TestCases` represent individual test cases for the state machine. The name of each test case must be unique per state machine and the value of each test case is an object specifying a mocked response to use for Task states in the state machine.

The following example of a `TestCase` links two Task states to two `MockedResponses`:

```
"HappyPath": {
  "SomeTaskState": "SomeMockedResponse",
  "AnotherTaskState": "AnotherMockedResponse"
}
```

MockedResponses

`MockedResponses` is an object containing multiple mocked response objects with unique field names. A mocked response object defines the successful result or error output for each invocation of a mocked Task state. You specify the invocation number using individual integer strings, such as "0", "1", "2", and "3" or an inclusive range of integers, such as "0-1", "2-3".

When you mock a Task, you must specify a mocked response for every invocation. A response must contain a single field named `Return` or `Throw` whose value is the result or error output for the mocked Task invocation. If you do not specify a mocked response, the state machine execution will fail.

The following is an example of a `MockedResponse` with `Throw` and `Return` objects. In this example, the first three times the state machine is run, the response specified in "0-2" is returned, and the fourth time the state machine runs, the response specified in "3" is returned.

```
"SomeMockedResponse": {
```

```
"0-2": {
  "Throw": {
    ...
  }
},
"3": {
  "Return": {
    ...
  }
}
}
```

Note

If you are using a Map state, and want to ensure predictable responses for the Map state, set the value of `maxConcurrency` to 1. If you set a value greater than 1, Step Functions Local will run multiple iterations concurrently, which will cause the overall execution order of states across iterations to be unpredictable. This may further cause Step Functions Local to use different mocked responses for iteration states from one execution to the next.

Return

Return is represented as a field of the `MockedResponse` objects. It specifies the successful result of a mocked Task state.

The following is an example of a `Return` object that contains a mocked response for calling [Invoke](#) on a Lambda function:

```
"Return": {
  "StatusCode": 200,
  "Payload": {
    "StatusCode": 200,
    "body": "Hello from Lambda!"
  }
}
```

Throw

Throw is represented as a field of the `MockedResponse` objects. It specifies the [error output](#) of a failed Task. The value of `Throw` must be an object containing an `Error` and

Cause fields with string values. In addition, the string value you specify in `Error` field in the `MockConfigFile.json` must match the errors handled in the `Retry` and `Catch` sections of your state machine.

The following is an example of a `Throw` object that contains a mocked response for calling [Invoke](#) on a Lambda function:

```
"Throw": {
  "Error": "Lambda.TimeoutException",
  "Cause": "Lambda timed out."
}
```

Configuring mocked service integrations

You can mock any service integration using Step Functions Local. However, Step Functions Local doesn't enforce the mocks to be the same as the real APIs. A mocked Task will never call the service endpoint. If you do not specify a mocked response, a Task will attempt to call the service endpoints. In addition, Step Functions Local will automatically generate a task token when you mock a Task using the `.waitForTaskToken`.

Best practices for Step Functions

The following best practices for implementing AWS Step Functions workflows can help you optimize the performance of your implementations.

Topics

- [Use timeouts to avoid stuck executions](#)
- [Use Amazon S3 ARNs instead of passing large payloads](#)
- [Avoid reaching the history quota](#)
- [Handle Lambda service exceptions](#)
- [Avoid latency when polling for activity tasks](#)
- [Choosing Standard or Express Workflows](#)
- [Amazon CloudWatch Logs resource policy size restrictions](#)

Use timeouts to avoid stuck executions

By default, the Amazon States Language doesn't specify timeouts for state machine definitions. Without an explicit timeout, Step Functions often relies solely on a response from an activity worker to know that a task is complete. If something goes wrong and the `TimeoutSeconds` field isn't specified for an `Activity` or `Task` state, an execution is stuck waiting for a response that will never come.

To avoid this situation, specify a reasonable timeout when you create a `Task` in your state machine. For example:

```
"ActivityState": {
  "Type": "Task",
  "Resource": "arn:aws:states:us-east-1:123456789012:activity:HelloWorld",
  "TimeoutSeconds": 300,
  "Next": "NextState"
}
```

If you use a [callback with a task token \(`.waitForTaskToken`\)](#), we recommend that you use heartbeats and add the `HeartbeatSeconds` field in your `Task` state definition. You can set `HeartbeatSeconds` to be less than the task timeout so if your workflow fails with a heartbeat

error then you know it's because of the task failure instead of the task taking a long time to complete.

```
{
  "StartAt": "Push to SQS",
  "States": {
    "Push to SQS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
      "HeartbeatSeconds": 600,
      "Parameters": {
        "MessageBody": { "myTaskToken.$": "$$.Task.Token" },
        "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/push-based-queue"
      },
      "ResultPath": "$.SQS",
      "End": true
    }
  }
}
```

For more information, see [Task state](#) in the Amazon States Language documentation.

Note

You can set a timeout for your state machine using the `TimeoutSeconds` field in your Amazon States Language definition. For more information, see [State machine structure](#).

Use Amazon S3 ARNs instead of passing large payloads

Executions that pass large payloads of data between states can be terminated. If the data you are passing between states might grow to over 256 KB, use Amazon Simple Storage Service (Amazon S3) to store the data, and parse the Amazon Resource Name (ARN) of the bucket in the `Payload` parameter to get the bucket name and key value. Alternatively, adjust your implementation so that you pass smaller payloads in your executions.

In the following example, a state machine passes input to an AWS Lambda function, which processes a JSON file in an Amazon S3 bucket. After you run this state machine, the Lambda function reads the contents of the JSON file, and returns the file contents as output.

Create the Lambda function

The following Lambda function named *pass-large-payload* reads the contents of a JSON file stored in a specific Amazon S3 bucket.

Note

After you create this Lambda function, make sure you provide its IAM role the appropriate permission to read from an Amazon S3 bucket. For example, attach the **AmazonS3ReadOnlyAccess** permission to the Lambda function's role.

```
import json
import boto3
import io
import os

s3 = boto3.client('s3')

def lambda_handler(event, context):
    event = event['Input']
    final_json = str()

    s3 = boto3.resource('s3')
    bucket = event['bucket'].split(':')[1]
    filename = event['key']
    directory = "/tmp/{}".format(filename)

    s3.Bucket(bucket).download_file(filename, directory)

    with open(directory, "r") as jsonfile:

        final_json = json.load(jsonfile)

    os.popen("rm -rf /tmp")

    return final_json
```

Create the state machine

The following state machine invokes the Lambda function you previously created.

```
{
  "StartAt": "Invoke Lambda function",
```

```
"States":{
  "Invoke Lambda function":{
    "Type":"Task",
    "Resource":"arn:aws:states:::lambda:invoke",
    "Parameters":{
      "FunctionName":"arn:aws:lambda:us-east-2:123456789012:function:pass-large-payload",
      "Payload":{
        "Input.$":"$"
      }
    },
    "OutputPath": "$.Payload",
    "End":true
  }
}
```

Rather than pass a large amount of data in the input, you could save that data in an Amazon S3 bucket, and pass the Amazon Resource Name (ARN) of the bucket in the Payload parameter to get the bucket name and key value. Your Lambda function can then use that ARN to access the data directly. The following is example input for the state machine execution, where the data is stored in `data.json` in an Amazon S3 bucket named *large-payload-json*.

```
{
  "key": "data.json",
  "bucket": "arn:aws:s3:::large-payload-json"
}
```

Avoid reaching the history quota

AWS Step Functions has a hard quota of 25,000 entries in the execution event history. When an execution reaches 24,999 events, it waits for the next event to happen.

- If the event number 25,000 is `ExecutionSucceeded`, the execution finishes successfully.
- If the event number 25,000 isn't `ExecutionSucceeded`, the `ExecutionFailed` event is logged and the state machine execution fails because of reaching the history limit

To avoid reaching this quota for long-running executions, you can try one of the following workarounds:

- [Use the Map state in Distributed mode](#). In this mode, the Map state runs each iteration as a child workflow execution, which enables high concurrency of up to 10,000 parallel child workflow executions. Each child workflow execution has its own, separate execution history from that of the parent workflow.
- Start a new state machine execution directly from the Task state of a running execution. To start such nested workflow executions, use Step Functions' [StartExecution](#) API action in the parent state machine along with the necessary parameters. For more information about using nested workflows, see [Start Workflow Executions from a Task State](#) or [Using a Step Functions API action to continue a new execution](#) tutorial.

i Tip

To deploy an example of a nested workflow to your AWS account, see [Module 13 - Nested Express Workflows](#).

- Implement a pattern that uses an AWS Lambda function that can start a new execution of your state machine to split ongoing work across multiple workflow executions. For more information, see the [Using a Lambda function to continue a new execution](#) tutorial.

Handle Lambda service exceptions

AWS Lambda can occasionally experience transient service errors. In this case, invoking Lambda results in a 500 error, such as `ClientExecutionTimeoutException`, `ServiceException`, `AWSLambdaException`, or `SdkClientException`. As a best practice, proactively handle these exceptions in your state machine to Retry invoking your Lambda function, or to Catch the error.

Lambda errors are reported as `Lambda.ErrorMessage`. To retry a Lambda service exception error, you could use the following Retry code.

```
"Retry": [ {
  "ErrorEquals": [ "Lambda.ClientExecutionTimeoutException",
    "Lambda.ServiceException", "Lambda.AWSLambdaException", "Lambda.SdkClientException"],
  "IntervalSeconds": 2,
  "MaxAttempts": 6,
  "BackoffRate": 2
} ]
```

Note

Unhandled errors in Lambda are reported as `Lambda.Unknown` in the error output. These include out-of-memory errors and function timeouts. You can match on `Lambda.Unknown`, `States.ALL`, or `States.TaskFailed` to handle these errors. When Lambda hits the maximum number of invocations, the error is `Lambda.TooManyRequestsException`. For more information about Lambda function errors, see [Error handling and automatic retries](#) in the *AWS Lambda Developer Guide*.

For more information, see the following:

- [Retrying after an error](#)
- [Handling error conditions using a Step Functions state machine](#)
- [Lambda Invoke Errors](#)

Avoid latency when polling for activity tasks

The [GetActivityTask](#) API is designed to provide a [taskToken](#) *exactly once*. If a `taskToken` is dropped while communicating with an activity worker, a number of `GetActivityTask` requests can be blocked for 60 seconds waiting for a response until `GetActivityTask` times out.

If you only have a small number of polls waiting for a response, it's possible that all requests will queue up behind the blocked request and stop. However, if you have a large number of outstanding polls for each activity Amazon Resource Name (ARN), and some percentage of your requests are stuck waiting, there will be many more that can still get a `taskToken` and begin to process work.

For production systems, we recommend at least 100 open polls per activity ARN's at each point in time. If one poll gets blocked, and a portion of those polls queue up behind it, there are still many more requests that will receive a `taskToken` to process work while the `GetActivityTask` request is blocked.

To avoid these kinds of latency problems when polling for tasks:

- Implement your pollers as separate threads from the work in your activity worker implementation.

- Have at least 100 open polls per activity ARN at each point in time.

Note

Scaling to 100 open polls per ARN can be expensive. For example, 100 Lambda functions polling per ARN is 100 times more expensive than having a single Lambda function with 100 polling threads. To both reduce latency *and* minimize cost, use a language that has asynchronous I/O, and implement multiple polling threads per worker. For an example activity worker where the poller threads are separate from the work threads, see [Example Activity Worker in Ruby](#).

For more information on activities and activity workers see [Activities](#).

Choosing Standard or Express Workflows

AWS Step Functions offers Standard Workflows as the default workflow type, with the option to choose Express Workflows.

You can choose Standard Workflows when you need long-running, durable, and auditable workflows, or Express Workflows for high-volume, event processing workloads. Your state machine executions will behave differently, depending on which Type you select. The Type you choose cannot be changed after your state machine has been created.

- For detailed information about the differences between Standard and Express Workflows, see [Standard vs. Express Workflows](#).
- For information about optimizing cost while building serverless workflows using Step Functions, see [Cost-optimization using Express Workflows](#).

Amazon CloudWatch Logs resource policy size restrictions

CloudWatch Logs resource policies are limited to 5120 characters. When CloudWatch Logs detects that a policy approaches this size limit, it automatically enables log groups that start with `/aws/vendedlogs/`.

When you create a state machine with logging enabled, Step Functions must update your CloudWatch Logs resource policy with the log group you specify. To avoid reaching the CloudWatch

Logs resource policy size limit, prefix your CloudWatch Logs log group names with `/aws/vendedlogs/`. When you create a log group in the Step Functions console, the log group names are prefixed with `/aws/vendedlogs/states`. For more information, see [Enabling Logging from Certain AWS Services](#).

If you're unable to access the CloudWatch Logs, see [Unable to access the CloudWatch Logs](#) for information.

Using AWS Step Functions with other services

Learn about coordinating other AWS services or calling third-party APIs with AWS Step Functions.

Topics

- [Call other AWS services](#)
- [AWS SDK service integrations](#)
- [Optimized integrations for Step Functions](#)
- [Call third-party APIs](#)
- [Service integration patterns](#)
- [Pass parameters to a service API](#)
- [Change log for supported AWS SDK integrations](#)

Call other AWS services

With AWS service integrations, you can call API actions and coordinate executions directly from your workflow. You can use Step Functions' [AWS SDK integrations](#) to call any of the over two hundred AWS services directly from your state machine, giving you access to over nine thousand API actions. Or you can use [Step Functions' Optimized integrations](#), each of which has been customized to provide special functionality for your workflow. Some API actions are available in both types of integration. When possible, we recommend using the Optimized integration.

You coordinate these services directly from a Task state in the Amazon States Language. For example, using Step Functions, you can call other services to:

- Invoke an AWS Lambda function.
- Run an AWS Batch job and then perform different actions based on the results.
- Insert or get an item from Amazon DynamoDB.
- Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
- Publish to a topic in Amazon Simple Notification Service (Amazon SNS).
- Send a message in Amazon Simple Queue Service (Amazon SQS).
- Manage a job for AWS Glue or Amazon SageMaker.
- Build workflows for executing Amazon EMR jobs.
- Launch an AWS Step Functions workflow execution.

Optimized integrations

Optimized integrations have been customized by Step Functions to provide special functionality for a workflow context. For example, [Lambda Invoke](#) converts its API output from an escaped JSON to a JSON object. [AWS BatchSubmitJob](#) lets you pause execution until the job is complete. The first set of optimized integrations was released in 2018, and there are now over fifty APIs.

AWS SDK integrations

AWS SDK integrations work exactly like a standard API call using the AWS SDK. They provide the ability to call over nine thousand APIs across the more than two hundred AWS services directly from your state machine definition.

Integration pattern support

Standard Workflows and Express Workflows support the same integrations but not the same **integration patterns**.

- Optimized integrations pattern support is different for each integration.
- Express Workflows do **not** support Run a Job (.sync) or Wait for Callback (.waitForTaskToken).
- For more information, see [Standard vs. Express Workflows](#).

Standard Workflows

Supported service integrations

	Service	Request Response	Run a Job (.sync)	Wait for Callback (.waitForTaskToken)
Optimized integrations	Amazon API Gateway	✓		✓
	Amazon Athena	✓	✓	
	AWS Batch	✓	✓	
	Amazon Bedrock	✓	✓	✓

	Service	<u>Request Response</u>	<u>Run a Job (.sync)</u>	<u>Wait for Callback (.waitForTaskToken)</u>
	AWS CodeBuild	✓	✓	
	Amazon DynamoDB	✓		
	Amazon ECS/Fargate	✓	✓	✓
	Amazon EKS	✓	✓	✓
	Amazon EMR	✓	✓	
	Amazon EMR on EKS	✓	✓	
	Amazon EMR Serverless	✓	✓	
	Amazon EventBridge	✓		✓
	AWS Glue	✓	✓	
	AWS Glue DataBrew	✓	✓	
	AWS Lambda	✓		✓
	AWS Elemental MediaConvert	✓	✓	
	Amazon SageMaker	✓	✓	
	Amazon SNS	✓		✓
	Amazon SQS	✓		✓
	AWS Step Functions	✓	✓	✓
AWS SDK integrations	Over two hundred	✓		✓

Express Workflows

Supported service integrations

	Service	Request Response	Run a Job (.sync)	Wait for Callback (.waitForTaskToken)
Optimized integrations	Amazon API Gateway	✓		
	Amazon Athena	✓		
	AWS Batch	✓		
	Amazon Bedrock	✓		
	AWS CodeBuild	✓		
	Amazon DynamoDB	✓		
	Amazon ECS/Fargate	✓		
	Amazon EKS	✓		
	Amazon EMR	✓		
	Amazon EMR on EKS	✓		
	Amazon EMR Serverless	✓		
	Amazon EventBridge	✓		
	AWS Glue	✓		
	AWS Glue DataBrew	✓		
	AWS Lambda	✓		
	AWS Elemental MediaConvert	✓		
Amazon SageMaker	✓			

	Service	Request Response	Run a Job (.sync)	Wait for Callback (.waitForTaskToken)
	Amazon SNS	✓		
	Amazon SQS	✓		
	AWS Step Functions	✓		
AWS SDK integrations	Over two hundred	✓		

Cross-account access

Step Functions provides cross-account access to resources configured in different AWS accounts in your workflows. Using Step Functions service integrations, you can invoke any cross-account AWS resource even if that AWS service doesn't support resource-based policies or cross-account calls.

For more information, see [Accessing resources in other AWS accounts in your workflows](#).

AWS SDK service integrations

AWS Step Functions integrates with AWS services, letting you call each service's API actions from your workflow. You can use Step Functions' [AWS SDK integrations](#) to call almost any AWS service's API actions from your state machine. You can also use [Step Functions' Optimized integrations](#), each of which has been customized to provide special functionality for your workflow.

Some services or SDKs may not be available as AWS SDK integrations, either temporarily or permanently. Recently released services may not have SDK interactions available until a later update. Some services require customized configuration, such as specifying a customer-specific endpoint, which may be more suitable for an optimized integration. Other SDKs are unsuitable for use in a workflow, such as those for streaming audio or video. Finally, some services may be withheld until they pass certain internal validations performed by Step Functions.

Tip

To deploy an example of a workflow that uses AWS SDK integrations to your AWS account, see [Module 9 - AWS SDK service integrations](#) in *The AWS Step Functions Workshop*.

Topics

- [Using AWS SDK service integrations](#)
- [Supported AWS SDK service integrations](#)
- [Unsupported API actions for supported services](#)
- [Deprecated AWS SDK service integrations](#)

Using AWS SDK service integrations

To use AWS SDK integrations, you specify the service name and API call and, optionally, a [service integration pattern](#).

Note

- Parameters in Step Functions are expressed in PascalCase, even if the native service API is in camelCase. For example, you could use the Step Functions API action `startSyncExecution` and specify its parameter as `StateMachineArn`.
- For API actions that accept enumerated parameters, such as the [DescribeLaunchTemplateVersions](#) API action for Amazon EC2, specify a plural version of the parameter name. For example, specify `Filters` for the `Filter.N` parameter of the `DescribeLaunchTemplateVersions` API action.

You can call AWS SDK services directly from the Amazon States Language in the `Resource` field of a task state. To do this, use the following syntax:

```
arn:aws:states:::aws-sdk:serviceName:apiAction.[serviceIntegrationPattern]
```

For example, for Amazon EC2, you might use `arn:aws:states:::aws-sdk:ec2:describeInstances`. This would return output as defined for the [Amazon EC2 describeInstances](#) API call.

If an AWS SDK integration encounters an error, the resulting Error field will be composed of the service name and the error name, separated by a period character: *ServiceName.ErrorName*. Both the service name and error name are in Pascal case. You can also see the service name, in lowercase, in the Task state's Resource field. You can find the potential error names in the target service's API reference documentation.

For example, you might use the `arn:aws:states:::aws-sdk:acmpca:deleteCertificateAuthority` AWS SDK integration. The [AWS Private Certificate Authority API Reference](#) indicates that the `DeleteCertificateAuthority` API action can result in a `ResourceNotFoundException`, for example. To handle this error, you would thus specify the `ErrorAcmPca.ResourceNotFoundException` in your Task state's Retriers or Catchers. For more information on error handling in Step Functions, see [Error handling in Step Functions](#).

Step Functions can't autogenerate IAM policies for AWS SDK integrations. After you create your state machine, you will need to navigate to the IAM console and configure your role policies. See [IAM Policies for integrated services](#) for more information.

See the [Gather Amazon S3 bucket info using AWS SDK service integrations](#) tutorial for an example of how to use AWS SDK integrations.

Supported AWS SDK service integrations

The following table lists the AWS SDK service integrations supported by Step Functions. The *Task state resource* column lists the syntax to call a specific API action when using the integration for the service specified in the *Service name* column on the left. The *Date supported* column provides information about the dates on which the service integration was supported. In addition, the *Exception prefix* column on the right, lists the exception prefixes for each service integration. These exception prefixes are present in the exceptions that are generated when you erroneously perform an AWS SDK service integration with Step Functions.

Note

- Services marked with ******* have API actions that are not supported by Step Functions at this time. For information about the actions that aren't supported for a service, see the [Unsupported API actions for supported services](#) table.
- For information about the updates made with each launch to expand the support for AWS SDK integrations, see [Change log for supported AWS SDK integrations](#).

⚠ Important

API action support is released on a quarterly cadence. Updates to already supported actions, such as new parameters, may not be immediately available.

Service name	Task state resource	Date supported	Exception prefix
AWS AppFabric	arn:aws:states:::aws-sdk:appfabric: <i>[apiAction]</i>	January 18, 2024	AppFabric
B2B Data Interchange	arn:aws:states:::aws-sdk:b2bi: <i>[apiAction]</i>	January 18, 2024	B2Bi
AWS Data Exports	arn:aws:states:::aws-sdk:bcmdataexports: <i>[apiAction]</i>	January 18, 2024	BcmDataExports
Amazon Bedrock	arn:aws:states:::aws-sdk:bedrock: <i>[apiAction]</i>	January 18, 2024	Bedrock
Amazon Bedrock Agents	arn:aws:states:::aws-sdk:bedrockagent: <i>[apiAction]</i>	January 18, 2024	BedrockAgent
Amazon Bedrock Runtime Agents	arn:aws:states:::aws-sdk:bedrockagentruntime: <i>[apiAction]</i>	January 18, 2024	BedrockAgentRuntime
Amazon Bedrock Runtime	arn:aws:states:::aws-sdk:bedrockruntime: <i>[apiAction]</i>	January 18, 2024	BedrockRuntime

Service name	Task state resource	Date supported	Exception prefix
AWS Clean Rooms	arn:aws:states:::aws-sdk:cleanroomsml: <i>[apiAction]</i>	January 18, 2024	CleanRoomsML
Amazon CloudFront KeyValueCollection	arn:aws:states:::aws-sdk:cloudfrontkeyvaluestore: <i>[apiAction]</i>	January 18, 2024	CloudFrontKeyValueCollection
CodeGuru Security	arn:aws:states:::aws-sdk:codegurusecurity: <i>[apiAction]</i>	January 18, 2024	CodeGuruSecurity
AWS Cost Optimization Hub	arn:aws:states:::aws-sdk:costoptimizationhub: <i>[apiAction]</i>	January 18, 2024	CostOptimizationHub
Amazon DataZone	arn:aws:states:::aws-sdk:datazone: <i>[apiAction]</i>	January 18, 2024	DataZone
Amazon EKS Auth	arn:aws:states:::aws-sdk:eksauth: <i>[apiAction]</i>	January 18, 2024	EksAuth
AWS Entity Resolution	arn:aws:states:::aws-sdk:entityresolution: <i>[apiAction]</i>	January 18, 2024	EntityResolution
AWS Free Tier	arn:aws:states:::aws-sdk:free-tier: <i>[apiAction]</i>	January 18, 2024	FreeTier

Service name	Task state resource	Date supported	Exception prefix
Amazon Inspector Scan	arn:aws:states:::aws-sdk:inspectorscan: <i>[apiAction]</i>	January 18, 2024	InspectorScan
AWS Launch Wizard	arn:aws:states:::aws-sdk:launchwizard: <i>[apiAction]</i>	January 18, 2024	LaunchWizard
Amazon Managed Blockchain Query	arn:aws:states:::aws-sdk:managedblockchainquery: <i>[apiAction]</i>	January 18, 2024	ManagedBlockchainQuery
AWS Elemental MediaPackage V2	arn:aws:states:::aws-sdk:mediapackagev2: <i>[apiAction]</i>	January 18, 2024	MediaPackageV2
AWS HealthImaging	arn:aws:states:::aws-sdk:medicalimaging: <i>[apiAction]</i>	January 18, 2024	MedicalImaging
Network Manager	arn:aws:states:::aws-sdk:networkmanager: <i>[apiAction]</i>	January 18, 2024	NetworkManager
AWS Payment Cryptography	arn:aws:states:::aws-sdk:paymentcryptography: <i>[apiAction]</i>	January 18, 2024	PaymentCryptography
AWS Payment Cryptography Data	arn:aws:states:::aws-sdk:paymentcryptographydata: <i>[apiAction]</i>	January 18, 2024	PaymentCryptographyData

Service name	Task state resource	Date supported	Exception prefix
AWS Private CA Connector for Active Directory	arn:aws:states:::aws-sdk:pcaconnectorad: <i>[apiAction]</i>	January 18, 2024	PcaConnectorAd
Amazon Q Business	arn:aws:states:::aws-sdk:qbusiness: <i>[apiAction]</i>	January 18, 2024	QBusiness
Amazon Q Connect	arn:aws:states:::aws-sdk:qconnect: <i>[apiAction]</i>	January 18, 2024	QConnect
AWS re:Post	arn:aws:states:::aws-sdk:repostspace: <i>[apiAction]</i>	January 18, 2024	Repostspace
Amazon Timestream Query	arn:aws:states:::aws-sdk:timestreamquery: <i>[apiAction]</i>	January 18, 2024	TimestreamQuery
Amazon Timestream Write	arn:aws:states:::aws-sdk:timestreamwrite: <i>[apiAction]</i>	January 18, 2024	TimestreamWrite
Trusted Advisor	arn:aws:states:::aws-sdk:trustedadvisor: <i>[apiAction]</i>	January 18, 2024	TrustedAdvisor
Verified Permissions	arn:aws:states:::aws-sdk:verifiedpermissions: <i>[apiAction]</i>	January 18, 2024	VerifiedPermissions

Service name	Task state resource	Date supported	Exception prefix
Amazon WorkSpaces Thin Client	arn:aws:states:::aws-sdk:workspacesthinclient: <i>[apiAction]</i>	January 18, 2024	WorkspaceThinClient
AWS CloudTrail Data	arn:aws:states:::aws-sdk:cloudtraildata: <i>[apiAction]</i>	June 16, 2023	CloudTrailData
Amazon CloudWatch Internet Monitor	arn:aws:states:::aws-sdk:internetmonitor: <i>[apiAction]</i>	June 16, 2023	InternetMonitor
Amazon Interactive Video Service RealTime	arn:aws:states:::aws-sdk:ivsrealtime: <i>[apiAction]</i>	June 16, 2023	IvsRealTime
AWS IoT TwinMaker	arn:aws:states:::aws-sdk:iottwinmaker: <i>[apiAction]</i>	June 16, 2023	IoT TwinMaker
Amazon OpenSearch Ingestion	arn:aws:states:::aws-sdk:osis: <i>[apiAction]</i>	June 16, 2023	Osis
AWS Telco Network Builder	arn:aws:states:::aws-sdk:tnb: <i>[apiAction]</i>	June 16, 2023	Tnb
Amazon VPC Lattice	arn:aws:states:::aws-sdk:vpclattice: <i>[apiAction]</i>	June 16, 2023	VpcLattice

Service name	Task state resource	Date supported	Exception prefix
Amazon Chime Media Pipelines	arn:aws:states:::aws-sdk:chimesdkmediapipelines: <i>[apiAction]</i>	February 17, 2023	ChimeSdkMediaPipelines
Amazon Chime Voice	arn:aws:states:::aws-sdk:chimesdkvoice: <i>[apiAction]</i>	February 17, 2023	ChimeSdkVoice
Amazon CodeCatalyst	arn:aws:states:::aws-sdk:codecatalyst: <i>[apiAction]</i>	February 17, 2023	CodeCatalyst
Amazon Connect Cases	arn:aws:states:::aws-sdk:connectcases: <i>[apiAction]</i>	February 17, 2023	ConnectCases
Amazon DocumentDB Elastic Clusters	arn:aws:states:::aws-sdk:docdbelastic: <i>[apiAction]</i>	February 17, 2023	DocDbElastic
Amazon EMR Serverless	arn:aws:states:::aws-sdk:emrserverless: <i>[apiAction]</i>	February 17, 2023	EmrServerless
Amazon IVS Chat	arn:aws:states:::aws-sdk:ivs: <i>[apiAction]</i>	February 17, 2023	Ivs
Amazon Kendra Intelligent Ranking	arn:aws:states:::aws-sdk:kendraranking: <i>[apiAction]</i>	February 17, 2023	KendraRanking
AWS HealthOmics	arn:aws:states:::aws-sdk:omics: <i>[apiAction]</i>	February 17, 2023	Omics

Service name	Task state resource	Date supported	Exception prefix
Amazon Redshift Serverless	arn:aws:states:::aws-sdk:redshiftserverless: <i>[apiAction]</i>	February 17, 2023	RedshiftServerless
Amazon Security Lake	arn:aws:states:::aws-sdk:securitylake: <i>[apiAction]</i>	February 17, 2023	SecurityLake
AWS Backup Storage	arn:aws:states:::aws-sdk:backupstorage: <i>[apiAction]</i>	February 17, 2023	BackupStorage
AWS Clean Rooms	arn:aws:states:::aws-sdk:cleanrooms: <i>[apiAction]</i>	February 17, 2023	CleanRooms
AWS Control Tower	arn:aws:states:::aws-sdk:controltower: <i>[apiAction]</i>	February 17, 2023	ControlTower
AWS Health	arn:aws:states:::aws-sdk:health: <i>[apiAction]</i>	February 17, 2023	Health
AWS IoT FleetWise	arn:aws:states:::aws-sdk:iotfleetwise: <i>[apiAction]</i>	February 17, 2023	IoT FleetWise
AWS Mainframe Modernization	arn:aws:states:::aws-sdk:m2: <i>[apiAction]</i>	February 17, 2023	M2
AWS Migration Hub Orchestrator	arn:aws:states:::aws-sdk:migrationhuborchestrator: <i>[apiAction]</i>	February 17, 2023	Migration Hub Orchestrator

Service name	Task state resource	Date supported	Exception prefix
AWS Private 5G	arn:aws:states:::aws-sdk:privatenetworks: <i>[apiAction]</i>	February 17, 2023	PrivateNetworks
AWS Resource Explorer	arn:aws:states:::aws-sdk:resourceexplorer2: <i>[apiAction]</i>	February 17, 2023	ResourceExplorer2
AWS SimSpace Weaver	arn:aws:states:::aws-sdk:simspaceweaver: <i>[apiAction]</i>	February 17, 2023	SimSpaceWeaver
AWS Support App	arn:aws:states:::aws-sdk:supportapp: <i>[apiAction]</i>	February 17, 2023	SupportApp
CloudWatch Observability Access Manager	arn:aws:states:::aws-sdk:oam: <i>[apiAction]</i>	February 17, 2023	Oam
EventBridge Pipes	arn:aws:states:::aws-sdk:pipes: <i>[apiAction]</i>	February 17, 2023	Pipes
EventBridge Scheduler	arn:aws:states:::aws-sdk:scheduler: <i>[apiAction]</i>	February 17, 2023	Scheduler
IAM Roles Anywhere	arn:aws:states:::aws-sdk:rolesanywhere: <i>[apiAction]</i>	February 17, 2023	RolesAnywhere
Kinesis Video WebRTC Storage	arn:aws:states:::aws-sdk:kinesisvideowebrtcstorage: <i>[apiAction]</i>	February 17, 2023	KinesisVideoWebRtcStorage

Service name	Task state resource	Date supported	Exception prefix
License Manager Linux Subscriptions	arn:aws:states:::aws-sdk:licensemanagerlinuxsubscriptions: <i>[apiAction]</i>	February 17, 2023	LicenseManagerLinuxSubscriptions
License Manager User Subscriptions	arn:aws:states:::aws-sdk:licensemanagerusersubscriptions: <i>[apiAction]</i>	February 17, 2023	LicenseManagerUserSubscriptions
OpenSearch Serverless	arn:aws:states:::aws-sdk:opensearchserverless: <i>[apiAction]</i>	February 17, 2023	OpenSearchServerless
Route 53 ARC Zonal Shift	arn:aws:states:::aws-sdk:arczonalshift: <i>[apiAction]</i>	February 17, 2023	ArcZonalShift
SageMaker Geospatial	arn:aws:states:::aws-sdk:sagemakergeospatial: <i>[apiAction]</i>	February 17, 2023	SageMakerGeospatial
SageMaker Metrics	arn:aws:states:::aws-sdk:sagemakermetrics: <i>[apiAction]</i>	February 17, 2023	SageMakerMetrics
Systems Manager for SAP	arn:aws:states:::aws-sdk:ssmsap: <i>[apiAction]</i>	February 17, 2023	SsmSap
AWS Account Management	arn:aws:states:::aws-sdk:account: <i>[apiAction]</i>	April 19, 2022	Account

Service name	Task state resource	Date supported	Exception prefix
AWS Amplify	arn:aws:states:::aws-sdk:amplify: <i>[apiAction]</i>	September 30, 2021	Amplify
AWS App Mesh	arn:aws:states:::aws-sdk:apmesh: <i>[apiAction]</i>	September 30, 2021	AppMesh
AWS App Runner	arn:aws:states:::aws-sdk:aprunner: <i>[apiAction]</i>	September 30, 2021	AppRunner
AWS AppConfig	arn:aws:states:::aws-sdk:apconfig: <i>[apiAction]</i>	September 30, 2021	AppConfig
AWS AppConfig Data	arn:aws:states:::aws-sdk:appconfigdata: <i>[apiAction]</i>	April 19, 2022	AppConfigData
AWS AppSync	arn:aws:states:::aws-sdk:apsync: <i>[apiAction]</i>	September 30, 2021	AppSync
AWS Application Discovery Service	arn:aws:states:::aws-sdk:applicationdiscovery: <i>[apiAction]</i> ^{***} —	September 30, 2021	ApplicationDiscovery
AWS Application Migration Service	arn:aws:states:::aws-sdk:mgn: <i>[apiAction]</i>	September 30, 2021	Mgn
AWS Audit Manager	arn:aws:states:::aws-sdk:auditmanager: <i>[apiAction]</i>	September 30, 2021	AuditManager

Service name	Task state resource	Date supported	Exception prefix
AWS Auto Scaling Plans	arn:aws:states:::aws-sdk:autoscalingplans: <i>[apiAction]</i>	September 30, 2021	AutoScalingPlans
AWS Backup	arn:aws:states:::aws-sdk:backup: <i>[apiAction]</i>	September 30, 2021	Backup
AWS Backup gateway	arn:aws:states:::aws-sdk:backupgateway: <i>[apiAction]</i>	April 19, 2022	BackupGateway
AWS Batch	arn:aws:states:::aws-sdk:batch: <i>[apiAction]</i>	September 30, 2021	Batch
AWS Billing Conductor	arn:aws:states:::aws-sdk:billingconductor: <i>[apiAction]</i>	July 26, 2022	Billingconductor
AWS Budgets	arn:aws:states:::aws-sdk:budgets: <i>[apiAction]</i>	September 30, 2021	Budgets
AWS Certificate Manager	arn:aws:states:::aws-sdk:acm: <i>[apiAction]</i>	September 30, 2021	Acm
AWS Private Certificate Authority	arn:aws:states:::aws-sdk:acmpca: <i>[apiAction]</i>	September 30, 2021	AcmPca
AWS Cloud Map	arn:aws:states:::aws-sdk:servicediscovery: <i>[apiAction]</i>	September 30, 2021	ServiceDiscovery

Service name	Task state resource	Date supported	Exception prefix
AWS Cloud9	arn:aws:states:::aws-sdk:cloud9: <i>[apiAction]</i>	September 30, 2021	Cloud9
AWS CloudFormation	arn:aws:states:::aws-sdk:cloudformation: <i>[apiAction]</i>	September 30, 2021	CloudFormation
AWS CloudHSM	arn:aws:states:::aws-sdk:cloudhsm: <i>[apiAction]</i>	September 30, 2021	CloudHsm
AWS CloudHSM	arn:aws:states:::aws-sdk:cloudhsmv2: <i>[apiAction]</i>	September 30, 2021	CloudHsmV2
AWS CloudTrail	arn:aws:states:::aws-sdk:cloudtrail: <i>[apiAction]</i>	September 30, 2021	CloudTrail
AWS Cloud Control	arn:aws:states:::aws-sdk:cloudcontrol: <i>[apiAction]</i>	April 19, 2022	CloudControl
AWS CodeBuild	arn:aws:states:::aws-sdk:codebuild: <i>[apiAction]</i>	September 30, 2021	CodeBuild
AWS CodeCommit	arn:aws:states:::aws-sdk:codecommit: <i>[apiAction]</i>	September 30, 2021	CodeCommit
AWS CodeDeploy	arn:aws:states:::aws-sdk:codedeploy: <i>[apiAction]</i> ^{***} <u>—</u>	September 30, 2021	CodeDeploy

Service name	Task state resource	Date supported	Exception prefix
AWS CodePipeline	arn:aws:states:::aws-sdk:codepipeline: <i>[apiAction]</i>	September 30, 2021	CodePipeline
AWS CodeStar	arn:aws:states:::aws-sdk:codestar: <i>[apiAction]</i>	September 30, 2021	CodeStar
AWS CodeStar	arn:aws:states:::aws-sdk:codestarnotifications: <i>[apiAction]</i>	September 30, 2021	CodestarNotifications
AWS CodeStar	arn:aws:states:::aws-sdk:codestarconnections: <i>[apiAction]</i>	September 30, 2021	CodeStarConnections
AWS Compute Optimizer	arn:aws:states:::aws-sdk:computeoptimizer: <i>[apiAction]</i>	September 30, 2021	ComputeOptimizer
AWS Config	arn:aws:states:::aws-sdk:config: <i>[apiAction]</i>	September 30, 2021	Config
AWS Cost Explorer Service	arn:aws:states:::aws-sdk:costexplorer: <i>[apiAction]</i>	September 30, 2021	CostExplorer
AWS Cost and Usage Report	arn:aws:states:::aws-sdk:costandusageereport: <i>[apiAction]</i>	September 30, 2021	CostAndUsageReport

Service name	Task state resource	Date supported	Exception prefix
AWS Data Exchange	arn:aws:states:::aws-sdk:dataexchange: <i>[apiAction]</i>	September 30, 2021	DataExchange
AWS Data Pipeline	arn:aws:states:::aws-sdk:datapipeline: <i>[apiAction]</i>	September 30, 2021	DataPipeline
AWS DataSync	arn:aws:states:::aws-sdk:datasync: <i>[apiAction]</i>	September 30, 2021	DataSync
AWS Database Migration Service	arn:aws:states:::aws-sdk:databasemigration: <i>[apiAction]</i>	September 30, 2021	DatabaseMigration
AWS Device Farm	arn:aws:states:::aws-sdk:devicefarm: <i>[apiAction]</i>	September 30, 2021	DeviceFarm
AWS Direct Connect	arn:aws:states:::aws-sdk:directconnect: <i>[apiAction]</i> ^{***} —	September 30, 2021	DirectConnect
AWS Directory Service	arn:aws:states:::aws-sdk:directory: <i>[apiAction]</i>	September 30, 2021	Directory
AWS EC2 Instance Connect	arn:aws:states:::aws-sdk:ec2instanceconnect: <i>[apiAction]</i>	September 30, 2021	Ec2InstanceConnect
AWS Elastic Beanstalk	arn:aws:states:::aws-sdk:elasticbeanstalk: <i>[apiAction]</i>	September 30, 2021	ElasticBeanstalk

Service name	Task state resource	Date supported	Exception prefix
AWS Elemental MediaLive	arn:aws:states:::aws-sdk:medialive: <i>[apiAction]</i>	September 30, 2021	MediaLive
AWS Elemental MediaPackage	arn:aws:states:::aws-sdk:mediapackage: <i>[apiAction]</i> ^{***} <u>—</u>	September 30, 2021	MediaPackage
AWS Elemental MediaPackage VOD	arn:aws:states:::aws-sdk:mediapackagevod: <i>[apiAction]</i>	September 30, 2021	MediaPackageVod
AWS Elemental MediaStore	arn:aws:states:::aws-sdk:mediastore: <i>[apiAction]</i>	September 30, 2021	MediaStore
AWS Fault Injection Service	arn:aws:states:::aws-sdk:fis: <i>[apiAction]</i>	September 30, 2021	Fis
AWS Firewall Manager	arn:aws:states:::aws-sdk:fms: <i>[apiAction]</i>	September 30, 2021	Fms
AWS Glue	arn:aws:states:::aws-sdk:glue: <i>[apiAction]</i>	September 30, 2021	Glue
AWS Glue DataBrew	arn:aws:states:::aws-sdk:atabrew: <i>[apiAction]</i>	September 30, 2021	DataBrew
AWS IoT Greengrass	arn:aws:states:::aws-sdk:greengrass: <i>[apiAction]</i>	September 30, 2021	Greengrass

Service name	Task state resource	Date supported	Exception prefix
AWS Ground Station	arn:aws:states:::aws-sdk:groundstation: <i>[apiAction]</i>	September 30, 2021	GroundStation
AWS Identity and Access Management	arn:aws:states:::aws-sdk:iam: <i>[apiAction]</i>	September 30, 2021	iam
AWS IoT	arn:aws:states:::aws-sdk:iot: <i>[apiAction]</i> ^{***} <u>—</u>	September 30, 2021	iot
AWS IoT 1-Click	arn:aws:states:::aws-sdk:iot1clickprojects: <i>[apiAction]</i>	September 30, 2021	iot1ClickProjects
AWS IoT Analytics	arn:aws:states:::aws-sdk:iotanalytics: <i>[apiAction]</i>	September 30, 2021	IoTAnalytics
AWS IoT Core Device Advisor	arn:aws:states:::aws-sdk:iotdeviceadvisor: <i>[apiAction]</i> ^{***} <u>—</u>	September 30, 2021	iotDeviceAdvisor
AWS IoT Events	arn:aws:states:::aws-sdk:iotevents: <i>[apiAction]</i>	September 30, 2021	iotEvents
AWS IoT Events Data	arn:aws:states:::aws-sdk:ioteventsdata: <i>[apiAction]</i>	September 30, 2021	iotEventsData
AWS IoT Fleet Hub	arn:aws:states:::aws-sdk:iotfleethub: <i>[apiAction]</i>	September 30, 2021	IoT Fleet Hub

Service name	Task state resource	Date supported	Exception prefix
AWS IoT Greengrass Version 2	arn:aws:states:::aws-sdk:greengrassv2: <i>[apiAction]</i>	September 30, 2021	GreengrassV2
AWS IoT jobs data Plane	arn:aws:states:::aws-sdk:iotjobsdataplane: <i>[apiAction]</i>	September 30, 2021	lotJobsDataPlane
AWS IoT Secure Tunneling	arn:aws:states:::aws-sdk:iotsecuretunneling: <i>[apiAction]</i>	September 30, 2021	IoTSecure Tunneling
AWS IoT SiteWise	arn:aws:states:::aws-sdk:iotsitewise: <i>[apiAction]</i>	September 30, 2021	IoTSiteWise
AWS IoT Wireless	arn:aws:states:::aws-sdk:iotwireless: <i>[apiAction]</i>	September 30, 2021	lotWireless
AWS Key Management Service	arn:aws:states:::aws-sdk:kms: <i>[apiAction]</i>	September 30, 2021	Kms
AWS Lake Formation	arn:aws:states:::aws-sdk:lakeformation: <i>[apiAction]</i>	September 30, 2021	LakeFormation
AWS Lambda	arn:aws:states:::aws-sdk:lambda: <i>[apiAction]</i> ^{***} —	September 30, 2021	Lambda
AWS License Manager	arn:aws:states:::aws-sdk:licensemanager: <i>[apiAction]</i>	September 30, 2021	LicenseManager

Service name	Task state resource	Date supported	Exception prefix
AWS Marketplace	arn:aws:states:::aws-sdk:marketplacecatalog: <i>[apiAction]</i>	September 30, 2021	MarketplaceCatalog
AWS Marketplace Commerce Analytics	arn:aws:states:::aws-sdk:marketplacecommerceanalytics: <i>[apiAction]</i>	September 30, 2021	MarketplaceCommerceAnalytics
AWS Marketplace Entitlement Service	arn:aws:states:::aws-sdk:marketplaceentitlement: <i>[apiAction]</i>	September 30, 2021	MarketplaceEntitlement
AWS Elemental MediaTailor	arn:aws:states:::aws-sdk:mediatailor: <i>[apiAction]</i>	September 30, 2021	MediaTailor
AWS Migration Hub	arn:aws:states:::aws-sdk:migrationhub: <i>[apiAction]</i>	September 30, 2021	MigrationHub
AWS Migration Hub Config	arn:aws:states:::aws-sdk:migrationhubconfig: <i>[apiAction]</i>	September 30, 2021	MigrationHubConfig
AWS Migration Hub Strategy Recommendations	arn:aws:states:::aws-sdk:migrationhubstrategy: <i>[apiAction]</i>	April 19, 2022	MigrationHubStrategy
AWS Mobile	arn:aws:states:::aws-sdk:mobile: <i>[apiAction]</i>	September 30, 2021	

Service name	Task state resource	Date supported	Exception prefix
AWS Network Firewall	arn:aws:states:::aws-sdk:networkfirewall: <i>[apiAction]</i>	September 30, 2021	NetworkFirewall
AWS OpsWorks	arn:aws:states:::aws-sdk:opsworks: <i>[apiAction]</i>	September 30, 2021	OpsWorks
AWS OpsWorks CM	arn:aws:states:::aws-sdk:opsworkscm: <i>[apiAction]</i>	September 30, 2021	OpsWorksCm
AWS Organizations	arn:aws:states:::aws-sdk:organizations: <i>[apiAction]</i>	September 30, 2021	Organizations
AWS Outposts	arn:aws:states:::aws-sdk:outposts: <i>[apiAction]</i>	September 30, 2021	Outposts
AWS Panorama	arn:aws:states:::aws-sdk:panorama: <i>[apiAction]</i>	April 19, 2022	Panorama
Amazon Relational Database Service Performance Insights	arn:aws:states:::aws-sdk:pi: <i>[apiAction]</i>	September 30, 2021	Pi
AWS Price List	arn:aws:states:::aws-sdk:pricing: <i>[apiAction]</i>	September 30, 2021	Pricing
Amazon Relational Database Service	arn:aws:states:::aws-sdk:rdsdata: <i>[apiAction]</i> ^{***} —	September 30, 2021	RdsData

Service name	Task state resource	Date supported	Exception prefix
AWS Resilience Hub	arn:aws:states:::aws-sdk:resiliencehub: <i>[apiAction]</i>	April 19, 2022	Resiliencehub
AWS Resource Access Manager	arn:aws:states:::aws-sdk:ram: <i>[apiAction]</i>	September 30, 2021	Ram
AWS Resource Groups	arn:aws:states:::aws-sdk:resourcegroups: <i>[apiAction]</i>	September 30, 2021	ResourceGroups
AWS Resource Groups Tagging API	arn:aws:states:::aws-sdk:resourcegroupstaggingapi: <i>[apiAction]</i>	September 30, 2021	ResourceGroupsTaggingApi
AWS RoboMaker	arn:aws:states:::aws-sdk:robomaker: <i>[apiAction]</i>	September 30, 2021	RoboMaker
AWS IAM Identity Center	arn:aws:states:::aws-sdk:identitystore: <i>[apiAction]</i>	September 30, 2021	Identitystore
IAM Identity Center OIDC	arn:aws:states:::aws-sdk:ssoidc: <i>[apiAction]</i>	September 30, 2021	SsoOidc
AWS Secrets Manager	arn:aws:states:::aws-sdk:secretsmanager: <i>[apiAction]</i>	September 30, 2021	SecretsManager
AWS Security Token Service	arn:aws:states:::aws-sdk:sts: <i>[apiAction]</i> ^{***} —	September 30, 2021	Sts

Service name	Task state resource	Date supported	Exception prefix
AWS Security Hub	arn:aws:states:::aws-sdk:securityhub: : <i>[apiAction]</i>	September 30, 2021	SecurityHub
AWS Server Migration Service	arn:aws:states:::aws-sdk:sms: <i>[apiAction]</i>	September 30, 2021	Sms
AWS Service Catalog	arn:aws:states:::aws-sdk:servicecatalog: log: <i>[apiAction]</i>	September 30, 2021	ServiceCatalog
AWS Service Catalog AppRegistry	arn:aws:states:::aws-sdk:servicecatalogappregistry: istry: <i>[apiAction]</i>	September 30, 2021	ServiceCatalogAppRegistry
AWS Shield	arn:aws:states:::aws-sdk:shield: ield: <i>[apiAction]</i> ^{***}	September 30, 2021	Shield
AWS Signer	arn:aws:states:::aws-sdk:signer: gner: <i>[apiAction]</i>	September 30, 2021	Signer
IAM Identity Center	arn:aws:states:::aws-sdk:sso: : <i>[apiAction]</i>	September 30, 2021	Sso
IAM Identity Center Admin	arn:aws:states:::aws-sdk:ssoadmin: oadmin: <i>[apiAction]</i>	September 30, 2021	SsoAdmin
AWS Step Functions	arn:aws:states:::aws-sdk:sfn: <i>[apiAction]</i>	September 30, 2021	Sfn

Service name	Task state resource	Date supported	Exception prefix
AWS Storage Gateway	arn:aws:states:::aws-sdk:storagegateway: <i>[apiAction]</i>	September 30, 2021	StorageGateway
AWS Support	arn:aws:states:::aws-sdk:support: <i>[apiAction]</i>	September 30, 2021	Support
AWS Systems Manager Incident Manager	arn:aws:states:::aws-sdk:ssmincidents: <i>[apiAction]</i>		SsmIncidents
AWS Transfer Family	arn:aws:states:::aws-sdk:transfer: <i>[apiAction]</i>	September 30, 2021	Transfer
AWS WAF	arn:aws:states:::aws-sdk:waf: <i>[apiAction]</i>	September 30, 2021	Waf
AWS WAF Regional	arn:aws:states:::aws-sdk:wafregional: <i>[apiAction]</i>	September 30, 2021	WafRegional
AWS WAFV2	arn:aws:states:::aws-sdk:wafv2: <i>[apiAction]</i>	September 30, 2021	Wafv2
AWS Well-Architected Tool	arn:aws:states:::aws-sdk:wellarchitected: <i>[apiAction]</i>	September 30, 2021	WellArchitected
AWS X-Ray	arn:aws:states:::aws-sdk:xray: <i>[apiAction]</i>	September 30, 2021	XRay

Service name	Task state resource	Date supported	Exception prefix
AWS Marketplace Metering Service	arn:aws:states:::aws-sdk:marketplacemetering: <i>[apiAction]</i>	September 30, 2021	MarketplaceMetering
AWS Serverless Application Repository	arn:aws:states:::aws-sdk:serverlessapplicationrepository: <i>[apiAction]</i>	September 30, 2021	ServerlessApplicationRepository
AWS Identity and Access Management Access Analyzer	arn:aws:states:::aws-sdk:accessanalyzer: <i>[apiAction]</i>	September 30, 2021	AccessAnalyzer
Alexa for Business	arn:aws:states:::aws-sdk:alexaforbusiness: <i>[apiAction]</i>	September 30, 2021	AlexaForBusiness
Amazon API Gateway	arn:aws:states:::aws-sdk:apigateway: <i>[apiAction]</i>	September 30, 2021	ApiGateway
Amazon API Gateway	arn:aws:states:::aws-sdk:apigatewayv2: <i>[apiAction]</i>	September 30, 2021	ApiGatewayV2
Amazon AppIntegrations	arn:aws:states:::aws-sdk:appintegrations: <i>[apiAction]</i>	September 30, 2021	AppIntegrations
Amazon AppStream 2.0	arn:aws:states:::aws-sdk:appstream: <i>[apiAction]</i>	September 30, 2021	AppStream

Service name	Task state resource	Date supported	Exception prefix
Amazon AppFlow	arn:aws:states:::aws-sdk:appflow: <i>[apiAction]</i>	September 30, 2021	Appflow
Amazon Athena	arn:aws:states:::aws-sdk:athena: <i>[apiAction]</i>	September 30, 2021	Athena
Amazon Augmented AI	arn:aws:states:::aws-sdk:sagemakerairuntime: <i>[apiAction]</i>	September 30, 2021	SageMaker A2IRuntime
Amazon Braket	arn:aws:states:::aws-sdk:braket: <i>[apiAction]</i>	September 30, 2021	Braket
Amazon Chime	arn:aws:states:::aws-sdk:chime: <i>[apiAction]</i>	September 30, 2021	Chime
Amazon Chime Meetings	arn:aws:states:::aws-sdk:chimesdkmeetings: <i>[apiAction]</i>	April 19, 2022	ChimeSdkMeetings
Amazon Cloud Directory	arn:aws:states:::aws-sdk:clouddirectory: <i>[apiAction]</i>	September 30, 2021	CloudDirectory
Amazon CloudFront	arn:aws:states:::aws-sdk:cloudfront: <i>[apiAction]</i>	September 30, 2021	CloudFront
Amazon CloudSearch	arn:aws:states:::aws-sdk:cloudsearch: <i>[apiAction]</i>	September 30, 2021	CloudSearch

Service name	Task state resource	Date supported	Exception prefix
Amazon CloudWatch	arn:aws:states:::aws-sdk:cloudwatch: <i>[apiAction]</i>	September 30, 2021	CloudWatch
Amazon CloudWatch Application Insights	arn:aws:states:::aws-sdk:application insights: <i>[apiAction]</i>	September 30, 2021	ApplicationInsights
CloudWatch Evidently	arn:aws:states:::aws-sdk:evidently: <i>[apiAction]</i>	April 19, 2022	Evidently
Amazon CloudWatch Logs	arn:aws:states:::aws-sdk:cloudwatchlogs: <i>[apiAction]</i>	September 30, 2021	CloudWatchLogs
Amazon CloudWatch RUM	arn:aws:states:::aws-sdk:rum: <i>[apiAction]</i>	April 19, 2022	Rum
Amazon CloudWatch Synthetics	arn:aws:states:::aws-sdk:synthetics: <i>[apiAction]</i>	September 30, 2021	Synthetics
Amazon CodeGuru Profiler	arn:aws:states:::aws-sdk:codeguruprofiler: <i>[apiAction]</i>	September 30, 2021	CodeGuruProfiler
Amazon CodeGuru Reviewer	arn:aws:states:::aws-sdk:codegurureviewer: <i>[apiAction]</i>	September 30, 2021	CodeGuruReviewer
Amazon Cognito	arn:aws:states:::aws-sdk:cognitoidentity: <i>[apiAction]</i>	September 30, 2021	CognitoIdentity

Service name	Task state resource	Date supported	Exception prefix
Amazon Cognito Identity Provider	arn:aws:states:::aws-sdk:cognitoidentityprovider: <i>[apiAction]</i>	September 30, 2021	CognitoIdentityProvider
Amazon Cognito Sync	arn:aws:states:::aws-sdk:cognitosync: <i>[apiAction]</i>	September 30, 2021	CognitoSync
Amazon Comprehend	arn:aws:states:::aws-sdk:comprehend: <i>[apiAction]</i>	September 30, 2021	Comprehend
Amazon Comprehend Medical	arn:aws:states:::aws-sdk:comprehendmedical: <i>[apiAction]</i> ^{***} —	September 30, 2021	ComprehendMedical
Amazon Connect Contact Lens	arn:aws:states:::aws-sdk:connectcontactlens: <i>[apiAction]</i>	September 30, 2021	ConnectContactLens
Amazon Connect Participant Service	arn:aws:states:::aws-sdk:connectparticipant: <i>[apiAction]</i>	September 30, 2021	ConnectParticipant
Amazon Connect	arn:aws:states:::aws-sdk:connect: <i>[apiAction]</i>	September 30, 2021	Connect
Amazon Connect Voice ID	arn:aws:states:::aws-sdk:voiceid: <i>[apiAction]</i>	April 19, 2022	VoiceId
Amazon Connect Wisdom	arn:aws:states:::aws-sdk:wisdom: <i>[apiAction]</i>	April 19, 2022	Wisdom

Service name	Task state resource	Date supported	Exception prefix
Amazon Data Lifecycle Manager	arn:aws:states:::aws-sdk:dlm: <i>[apiAction]</i>	September 30, 2021	Dlm
Amazon Detective	arn:aws:states:::aws-sdk:detective: <i>[apiAction]</i>	September 30, 2021	Detective
Amazon DevOps Guru	arn:aws:states:::aws-sdk:devopsguru: <i>[apiAction]</i>	September 30, 2021	DevOpsGuru
Amazon DocumentDB (with MongoDB compatibility)	arn:aws:states:::aws-sdk:docdb: <i>[apiAction]</i>	September 30, 2021	DocDb
Amazon DynamoDB	arn:aws:states:::aws-sdk:dynamodb: <i>[apiAction]</i>	September 30, 2021	DynamoDb
Amazon DynamoDB Streams	arn:aws:states:::aws-sdk:dynamodbstreams: <i>[apiAction]</i>	September 30, 2021	DynamoDbStreams
Amazon EC2 Container Registry	arn:aws:states:::aws-sdk:ecr: <i>[apiAction]</i>	September 30, 2021	Ecr
Amazon EC2 Container Service	arn:aws:states:::aws-sdk:ecs: <i>[apiAction]</i>	September 30, 2021	Ecs
Amazon EC2 Systems Manager	arn:aws:states:::aws-sdk:ssm: <i>[apiAction]</i>	September 30, 2021	Ssm

Service name	Task state resource	Date supported	Exception prefix
Amazon EMR	arn:aws:states:::aws-sdk:emrcontainers: <i>[apiAction]</i> ^{***} <u>—</u>	September 30, 2021	EmrContainers
Amazon ElastiCache	arn:aws:states:::aws-sdk:elasticache: <i>[apiAction]</i>	September 30, 2021	ElastiCache
Amazon Elastic Inference	arn:aws:states:::aws-sdk:elasticinference: <i>[apiAction]</i>	September 30, 2021	ElasticInference
Amazon Elastic Block Store	arn:aws:states:::aws-sdk:ebs: <i>[apiAction]</i>	September 30, 2021	Ebs
Amazon Elastic Compute Cloud	arn:aws:states:::aws-sdk:ec2: <i>[apiAction]</i>	September 30, 2021	Ec2
Amazon Elastic Container Registry Public	arn:aws:states:::aws-sdk:ecrpublic: <i>[apiAction]</i>	September 30, 2021	EcrPublic
Amazon Elastic File System	arn:aws:states:::aws-sdk:efs: <i>[apiAction]</i> ^{***} <u>—</u>	September 30, 2021	Efs
Amazon Elastic Kubernetes Service	arn:aws:states:::aws-sdk:eks: <i>[apiAction]</i>	September 30, 2021	Eks
Amazon EMR	arn:aws:states:::aws-sdk:emr: <i>[apiAction]</i>	September 30, 2021	Emr

Service name	Task state resource	Date supported	Exception prefix
Amazon Elastic Transcoder	arn:aws:states:::aws-sdk:elastictranscoder: <i>[apiAction]</i> ^{***}	September 30, 2021	ElasticTranscoder
Amazon OpenSearch Service	arn:aws:states:::aws-sdk:elasticsearch: <i>[apiAction]</i>	September 30, 2021	Elasticsearch
Amazon OpenSearch Service	arn:aws:states:::aws-sdk:opensearch: <i>[apiAction]</i>	April 19, 2022	OpenSearch
Amazon EventBridge	arn:aws:states:::aws-sdk:eventbridge: <i>[apiAction]</i>	September 30, 2021	EventBridge
Amazon FSx	arn:aws:states:::aws-sdk:fsx: <i>[apiAction]</i>	September 30, 2021	FSx
Amazon Forecast Query	arn:aws:states:::aws-sdk:forecastquery: <i>[apiAction]</i>	September 30, 2021	Forecastquery
Amazon Forecast Service	arn:aws:states:::aws-sdk:forecast: <i>[apiAction]</i>	September 30, 2021	Forecast
Amazon Fraud Detector	arn:aws:states:::aws-sdk:frauddetector: <i>[apiAction]</i>	September 30, 2021	FraudDetector
Amazon GameLift	arn:aws:states:::aws-sdk:gamelift: <i>[apiAction]</i>	September 30, 2021	Amazon GameLift

Service name	Task state resource	Date supported	Exception prefix
Amazon GameSparks	arn:aws:states:::aws-sdk:gamesparks: <i>[apiAction]</i>	July 27, 2022	GameSparks
Amazon S3 Glacier	arn:aws:states:::aws-sdk:glacier: <i>[apiAction]</i>	September 30, 2021	Glacier
Amazon GuardDuty	arn:aws:states:::aws-sdk:guardduty: <i>[apiAction]</i>	September 30, 2021	GuardDuty
AWS HealthLake	arn:aws:states:::aws-sdk:healthlake: <i>[apiAction]</i>	September 30, 2021	HealthLake
Amazon Honeycode	arn:aws:states:::aws-sdk:honeycode: <i>[apiAction]</i>	September 30, 2021	Honeycode
Amazon Inspector	arn:aws:states:::aws-sdk:inspector: <i>[apiAction]</i>	September 30, 2021	Inspector
Amazon Inspector V2	arn:aws:states:::aws-sdk:inspector2: <i>[apiAction]</i>	April 19, 2022	Inspector2
Amazon Interactive Video Service	arn:aws:states:::aws-sdk:ivs: <i>[apiAction]</i>	September 30, 2021	Ivs
Amazon Kendra	arn:aws:states:::aws-sdk:kendra: <i>[apiAction]</i>	September 30, 2021	Kendra

Service name	Task state resource	Date supported	Exception prefix
Amazon Kinesis	arn:aws:states:::aws-sdk:kinesis: <i>[apiAction]</i> ^{***}	September 30, 2021	Kinesis
Amazon Kinesis Analytics	arn:aws:states:::aws-sdk:kinesisanalytics: <i>[apiAction]</i>	September 30, 2021	KinesisAnalytics
Amazon Kinesis Analytics V2	arn:aws:states:::aws-sdk:kinesisanalyticsv2: <i>[apiAction]</i>	September 30, 2021	KinesisAnalyticsV2
Amazon Kinesis Firehose	arn:aws:states:::aws-sdk:firehose: <i>[apiAction]</i>	September 30, 2021	Firehose
Amazon Kinesis Video Signaling Channels	arn:aws:states:::aws-sdk:kinesisvideosignaling: <i>[apiAction]</i>	September 30, 2021	KinesisVideoSignaling
Amazon Kinesis Video Streams	arn:aws:states:::aws-sdk:kinesisvideo: <i>[apiAction]</i>	September 30, 2021	KinesisVideo
Amazon Kinesis Video Streams Archived Media	arn:aws:states:::aws-sdk:kinesisvideoarchivedmedia: <i>[apiAction]</i>	September 30, 2021	KinesisVideoArchivedMedia
Amazon Kinesis video stream	arn:aws:states:::aws-sdk:kinesisvideomedia: <i>[apiAction]</i>	September 30, 2021	KinesisVideoMedia

Service name	Task state resource	Date supported	Exception prefix
Amazon Lex Model Building Service	arn:aws:states:::aws-sdk:lexmodelbuilding: <i>[apiAction]</i>	September 30, 2021	LexModelBuilding
Amazon Lex Model Building Service V2	arn:aws:states:::aws-sdk:lexmodelsv2: <i>[apiAction]</i>	September 30, 2021	LexModelsV2
Amazon Lex	arn:aws:states:::aws-sdk:lexruntime: <i>[apiAction]</i>	September 30, 2021	LexRuntime
Amazon Lex Runtime V2	arn:aws:states:::aws-sdk:lexruntimev2: <i>[apiAction]</i> ^{***} —	September 30, 2021	LexRuntimeV2
Amazon Lightsail	arn:aws:states:::aws-sdk:lightsail: <i>[apiAction]</i>	September 30, 2021	Lightsail
Amazon Location Service	arn:aws:states:::aws-sdk:location: <i>[apiAction]</i>	September 30, 2021	Location
Amazon Lookout for Equipment	arn:aws::states:::aws-sdk:lookoutequipment: <i>[apiAction]</i>	September 30, 2021	LookoutEquipment
Amazon Lookout for Metrics	arn:aws:states:::aws-sdk:lookoutmetrics: <i>[apiAction]</i>	September 30, 2021	LookoutMetrics
Amazon Lookout for Vision	arn:aws:states:::aws-sdk:lookoutvision: <i>[apiAction]</i>	September 30, 2021	LookoutVision

Service name	Task state resource	Date supported	Exception prefix
Amazon MQ	arn:aws:states:::aws-sdk:mq: <i>[apiAction]</i>	September 30, 2021	Mq
Amazon Macie	arn:aws:states:::aws-sdk:macie: <i>[apiAction]</i>	September 30, 2021	
Amazon Macie 2	arn:aws:states:::aws-sdk:macie2: <i>[apiAction]</i>	September 30, 2021	Macie2
Amazon Managed Blockchain	arn:aws:states:::aws-sdk:managedblockchain: <i>[apiAction]</i>	September 30, 2021	ManagedBlockchain
Amazon Managed Grafana	arn:aws:states:::aws-sdk:grafana: <i>[apiAction]</i>	April 19, 2022	Grafana
Amazon Managed Service for Prometheus	arn:aws:states:::aws-sdk:amp: <i>[apiAction]</i>	September 30, 2021	Amp
Amazon Managed Streaming for Apache Kafka	arn:aws:states:::aws-sdk:kafka: <i>[apiAction]</i>	September 30, 2021	Kafka
Amazon MSK Connect	arn:aws:states:::aws-sdk:kafkaconnect: <i>[apiAction]</i>	April 19, 2022	KafkaConnect
Amazon Managed Workflows for Apache Airflow	arn:aws:states:::aws-sdk:mwaa: <i>[apiAction]</i>	September 30, 2021	Mwaa

Service name	Task state resource	Date supported	Exception prefix
Amazon Mechanical Turk	arn:aws:states:::aws-sdk:mturk: <i>[apiAction]</i>	September 30, 2021	MTurk
Amazon MemoryDB for Redis	arn:aws:states:::aws-sdk:memorydb: <i>[apiAction]</i>	April 19, 2022	MemoryDB
Amazon Nimble Studio	arn:aws:states:::aws-sdk:nimble: <i>[apiAction]</i>	September 30, 2021	Nimble
Amazon Personalize	arn:aws:states:::aws-sdk:personalize: <i>[apiAction]</i>	September 30, 2021	Personalize
Amazon Personalize Events	arn:aws:states:::aws-sdk:personalizeevents: <i>[apiAction]</i>	September 30, 2021	PersonalizeEvents
Amazon Personalize Runtime	arn:aws:states:::aws-sdk:personalizeruntime: <i>[apiAction]</i>	September 30, 2021	PersonalizeRuntime
Amazon Pinpoint	arn:aws:states:::aws-sdk:pinpoint: <i>[apiAction]</i>	September 30, 2021	Pinpoint
Amazon Pinpoint Email Service	arn:aws:states:::aws-sdk:pinpointemail: <i>[apiAction]</i>	September 30, 2021	PinpointEmail
Amazon Pinpoint SMS and Voice Service	arn:aws:states:::aws-sdk:pinpointsmsvoice: <i>[apiAction]</i>	September 30, 2021	PinpointSmsVoice

Service name	Task state resource	Date supported	Exception prefix
Amazon Pinpoint SMS and Voice V2 Service	arn:aws:states:::aws-sdk:pinpointSMSvoicev2: <i>[apiAction]</i>	July 27, 2022	PinpointSMSVoiceV2
Amazon Polly	arn:aws:states:::aws-sdk:polly: <i>[apiAction]</i>	September 30, 2021	Polly
Amazon QLDB	arn:aws:states:::aws-sdk:qldb: <i>[apiAction]</i>	September 30, 2021	Qldb
Amazon QLDB Session	arn:aws:states:::aws-sdk:qldb-session: <i>[apiAction]</i>	September 30, 2021	QldbSession
Amazon QuickSight	arn:aws:states:::aws-sdk:quicksight: <i>[apiAction]</i>	September 30, 2021	QuickSight
Amazon Redshift	arn:aws:states:::aws-sdk:redshift: <i>[apiAction]</i>	September 30, 2021	Redshift
Amazon Redshift Data API	arn:aws:states:::aws-sdk:redshift-data: <i>[apiAction]</i>	September 30, 2021	RedshiftData
Amazon Rekognition	arn:aws:states:::aws-sdk:rekognition: <i>[apiAction]</i>	September 30, 2021	Rekognition
Amazon Relational Database Service	arn:aws:states:::aws-sdk:rds: <i>[apiAction]</i>	September 30, 2021	Rds

Service name	Task state resource	Date supported	Exception prefix
Amazon Route 53	arn:aws:states:::aws-sdk:route53: <i>[apiAction]</i>	September 30, 2021	Route53
Amazon Route 53 Recovery Control Config	arn:aws:states:::aws-sdk:route53recoverycontrolconfig: <i>[apiAction]</i>	September 30, 2021	Route53RecoveryControlConfig
Amazon Route 53 Domains	arn:aws:states:::aws-sdk:route53domains: <i>[apiAction]</i>	September 30, 2021	Route53Domains
Amazon Route 53 Resolver	arn:aws:states:::aws-sdk:route53resolver: <i>[apiAction]</i>	September 30, 2021	Route53Resolver
Amazon S3 on Outposts	arn:aws:states:::aws-sdk:s3outposts: <i>[apiAction]</i>	September 30, 2021	S3Outposts
Amazon SageMaker Runtime Feature Store Runtime	arn:aws:states:::aws-sdk:sagemakerfeaturestoreruntime: <i>[apiAction]</i>	September 30, 2021	SageMakerRuntimeFeatureStoreRuntime
Amazon SageMaker Runtime Runtime	arn:aws:states:::aws-sdk:sagemakerruntime: <i>[apiAction]</i>	September 30, 2021	SageMakerRuntimeRuntime
Amazon SageMaker	arn:aws:states:::aws-sdk:sagemaker: <i>[apiAction]</i>	September 30, 2021	SageMakerRuntime

Service name	Task state resource	Date supported	Exception prefix
Amazon SageMaker Edge Manager	arn:aws:states:::aws-sdk:sagemakeredge: <i>[apiAction]</i>	September 30, 2021	SagemakerEdge
Amazon Simple Email Service	arn:aws:states:::aws-sdk:ses: <i>[apiAction]</i>	September 30, 2021	Ses
Amazon Simple Email Service V2	arn:aws:states:::aws-sdk:sesv2: <i>[apiAction]</i>	September 30, 2021	SesV2
Amazon Simple Notification Service	arn:aws:states:::aws-sdk:sns: <i>[apiAction]</i>	September 30, 2021	Sns
Amazon Simple Queue Service	arn:aws:states:::aws-sdk:sqs: <i>[apiAction]</i>	September 30, 2021	Sqs
Amazon Simple Storage Service	arn:aws:states:::aws-sdk:s3: <i>[apiAction]</i> ^{***}	September 30, 2021	S3
Amazon Simple Workflow Service	arn:aws:states:::aws-sdk:swf: <i>[apiAction]</i>	September 30, 2021	Swf
Amazon Textract	arn:aws:states:::aws-sdk:textract: <i>[apiAction]</i>	September 30, 2021	Textract
Amazon Transcribe	arn:aws:states:::aws-sdk:transcribe: <i>[apiAction]</i>	September 30, 2021	Transcribe

Service name	Task state resource	Date supported	Exception prefix
Amazon Translate	arn:aws:states:::aws-sdk:translate: <i>[apiAction]</i>	September 30, 2021	Translate
Amazon WorkDocs	arn:aws:states:::aws-sdk:workdocs: <i>[apiAction]</i>	September 30, 2021	WorkDocs
Amazon WorkMail	arn:aws:states:::aws-sdk:workmail: <i>[apiAction]</i>	September 30, 2021	WorkMail
Amazon WorkMail Message Flow	arn:aws:states:::aws-sdk:workmailmessageflow: <i>[apiAction]</i>	September 30, 2021	WorkMailMessageFlow
Amazon WorkSpaces	arn:aws:states:::aws-sdk:workspaces: <i>[apiAction]</i>	September 30, 2021	WorkSpaces
Amazon WorkSpaces Web	arn:aws:states:::aws-sdk:workspacesweb: <i>[apiAction]</i>	April 19, 2022	WorkSpacesWeb
Amplify	arn:aws:states:::aws-sdk:amplifybackend: <i>[apiAction]</i>	September 30, 2021	AmplifyBackend
Amplify UI Builder	arn:aws:states:::aws-sdk:amplifyuibuilder: <i>[apiAction]</i>	April 19, 2022	AmplifyUiBuilder

Service name	Task state resource	Date supported	Exception prefix
Application Auto Scaling	arn:aws:states:::aws-sdk:applicationautoscaling: <i>[apiAction]</i>	September 30, 2021	ApplicationAutoScaling
Amazon EC2 Auto Scaling	arn:aws:states:::aws-sdk:autoscaling: <i>[apiAction]</i>	September 30, 2021	Auto Scaling
CodeArtifact	arn:aws:states:::aws-sdk:codeartifact: <i>[apiAction]</i>	September 30, 2021	Codeartifact
DynamoDB Accelerator	arn:aws:states:::aws-sdk:dax: <i>[apiAction]</i>	September 30, 2021	Dax
EC2 Image Builder	arn:aws:states:::aws-sdk:imagebuilder: <i>[apiAction]</i>	September 30, 2021	Imagebuilder
AWS Elastic Disaster Recovery	arn:aws:states:::aws-sdk:drs: <i>[apiAction]</i>	April 19, 2022	Drs
Elastic Load Balancing	arn:aws:states:::aws-sdk:elasticloadbalancing: <i>[apiAction]</i>	September 30, 2021	ElasticLoadBalancing
Elastic Load Balancing V2	arn:aws:states:::aws-sdk:elasticloadbalancingv2: <i>[apiAction]</i>	September 30, 2021	ElasticLoadBalancingV2

Service name	Task state resource	Date supported	Exception prefix
MediaConnect	arn:aws:states:::aws-sdk:mediacconnect: <i>[apiAction]</i>	September 30, 2021	MediaConnect
Amazon S3 Control	arn:aws:states:::aws-sdk:s3control: <i>[apiAction]</i> ***	September 30, 2021	S3Control
Recycle Bin for Amazon EBS	arn:aws:states:::aws-sdk:rb in: <i>[apiAction]</i>	April 19, 2022	Rbin
Savings Plans	arn:aws:states:::aws-sdk:savingsplans: <i>[apiAction]</i>	September 30, 2021	Savingsplans
Amazon EventBridge Schema Registry	arn:aws:states:::aws-sdk:schemas: <i>[apiAction]</i>	September 30, 2021	Schemas
Service Quotas	arn:aws:states:::aws-sdk:servicequotas: <i>[apiAction]</i>	September 30, 2021	ServiceQuotas
AWS Snowball	arn:aws:states:::aws-sdk:snowball: <i>[apiAction]</i>	September 30, 2021	Snowball

Unsupported API actions for supported services

The following table lists the unsupported API actions for AWS SDK service integrations. The right column contains the API actions that are currently not supported for the service listed in the left column.

Service name	Unsupported API action
AWS Application Discovery Service	<ul style="list-style-type: none"> • DescribeExportConfigurations • ExportConfigurations
Amazon Bedrock	<ul style="list-style-type: none"> • InvokeModelWithResponseStream
Agents for Amazon Bedrock Runtime	<ul style="list-style-type: none"> • InvokeAgent
AWS CodeDeploy	<ul style="list-style-type: none"> • BatchGetDeploymentInstances • GetDeploymentInstance • ListDeploymentInstances • SkipWaitTimeForInstanceTermination
Amazon Comprehend Medical	<ul style="list-style-type: none"> • DetectEntities
AWS Direct Connect	<ul style="list-style-type: none"> • AllocateConnectionOnInterconnect • DescribeConnectionLoa • DescribeConnectionsOnInterconnect • DescribeInterconnectLoa
Amazon Elastic File System	<ul style="list-style-type: none"> • CreateTags
Amazon Elastic Transcoder	<ul style="list-style-type: none"> • TestRole
Amazon EMR	<ul style="list-style-type: none"> • DescribeJobFlows
AWS IoT	<ul style="list-style-type: none"> • AttachPrincipalPolicy • ListPrincipalPolicies • DetachPrincipalPolicy • ListPolicyPrincipals • DetachPrincipalPolicy

Service name	Unsupported API action
AWS IoT Core Device Advisor	<ul style="list-style-type: none">ListTestCases
Amazon Kinesis	<ul style="list-style-type: none">SubscribeToShard
AWS Lambda	<ul style="list-style-type: none">InvokeAsyncInvokeWithResponseStream
Amazon Lex Runtime V2	<ul style="list-style-type: none">StartConversation
AWS Elemental MediaPackage	<ul style="list-style-type: none">RotateChannelCredentials
Amazon Relational Database Service	<ul style="list-style-type: none">ExecuteSql
Amazon Simple Storage Service	<ul style="list-style-type: none">SelectObjectContent
Amazon S3 Control	<ul style="list-style-type: none">SelectObjectContent
AWS Shield	<ul style="list-style-type: none">DeleteSubscription
AWS Security Token Service	<ul style="list-style-type: none">AssumeRoleAssumeRoleWithSAMLAssumeRoleWithWebIdentity

Deprecated AWS SDK service integrations

The following AWS SDK service integrations are now deprecated:

- AWS Mobile
- Amazon Macie
- AWS IoT RoboRunner

Optimized integrations for Step Functions

The following topics include the supported APIs, parameters, and request/response syntax in the Amazon States Language for coordinating other AWS services. The topics also provide example

code. You can call Optimized integrations services directly from the Amazon States Language in the Resource field of a Task state.

You can use three service integration patterns:

- [Request a Response \(default\)](#) - wait for HTTP response, then go to the next state
- [Run a Job \(.sync\)](#) - wait for the job to complete
- [Wait for Callback \(.waitForTaskToken\)](#) - pause a workflow until a task token is returned

Standard Workflows and Express Workflows support the same integrations but not the same **integration patterns**.

- Optimized integrations pattern support is different for each integration.
- Express Workflows do **not** support Run a Job (.sync) or Wait for Callback (.waitForTaskToken).
- For more information, see [Standard vs. Express Workflows](#).

Standard Workflows

Supported service integrations

	Service	Request Response	Run a Job (.sync)	Wait for Callback (.waitForTaskToken)
Optimized integrations	Amazon API Gateway	✓		✓
	Amazon Athena	✓	✓	
	AWS Batch	✓	✓	
	Amazon Bedrock	✓	✓	✓
	AWS CodeBuild	✓	✓	
	Amazon DynamoDB	✓		
	Amazon ECS/Fargate	✓	✓	✓

	Service	<u>Request Response</u>	<u>Run a Job (.sync)</u>	<u>Wait for Callback (.waitForTaskToken)</u>
	Amazon EKS	✓	✓	✓
	Amazon EMR	✓	✓	
	Amazon EMR on EKS	✓	✓	
	Amazon EMR Serverless	✓	✓	
	Amazon EventBridge	✓		✓
	AWS Glue	✓	✓	
	AWS Glue DataBrew	✓	✓	
	AWS Lambda	✓		✓
	AWS Elemental MediaConvert	✓	✓	
	Amazon SageMaker	✓	✓	
	Amazon SNS	✓		✓
	Amazon SQS	✓		✓
	AWS Step Functions	✓	✓	✓
AWS SDK integrations	Over two hundred	✓		✓

Express Workflows

Supported service integrations

	Service	<u>Request Response</u>	<u>Run a Job (.sync)</u>	<u>Wait for Callback (.waitForTaskToken)</u>
Optimized integrations	Amazon API Gateway	✓		
	Amazon Athena	✓		
	AWS Batch	✓		
	Amazon Bedrock	✓		
	AWS CodeBuild	✓		
	Amazon DynamoDB	✓		
	Amazon ECS/Fargate	✓		
	Amazon EKS	✓		
	Amazon EMR	✓		
	Amazon EMR on EKS	✓		
	Amazon EMR Serverless	✓		
	Amazon EventBridge	✓		
	AWS Glue	✓		
	AWS Glue DataBrew	✓		
	AWS Lambda	✓		
	AWS Elemental MediaConvert	✓		
Amazon SageMaker	✓			

	Service	Request Response	Run a Job (.sync)	Wait for Callback (.waitForTaskToken)
	Amazon SNS	✓		
	Amazon SQS	✓		
	AWS Step Functions	✓		
AWS SDK integrations	Over two hundred	✓		

Call API Gateway with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

How the Optimized API Gateway integration is different than the API Gateway AWS SDK integration

- `apigateway:invoke`: has no equivalent in the AWS SDK service integration. Instead, the Optimized API Gateway service calls your API Gateway endpoint directly.

You use Amazon API Gateway to create, publish, maintain, and monitor HTTP and REST APIs. To integrate with API Gateway, you define a Task state in Step Functions that directly calls an API Gateway HTTP or API Gateway REST endpoint, without writing code or relying on other infrastructure. A Task state definition includes all the necessary information for the API call. You can also select different authorization methods.

API Gateway feature support

The Step Functions API Gateway integration supports some, but not all API Gateway features. For a more detailed list of supported features, see the following.

- Supported by both the Step Functions API Gateway REST API and API Gateway HTTP API integrations:
 - **Authorizers:** IAM (using [Signature Version 4](#)), No Auth, Lambda Authorizers (request-parameter based and token-based with custom header)
 - **API types:** Regional
 - **API management:** API Gateway API domain names, API stage, Path, Query Parameters, Request Body
- Supported by the Step Functions API Gateway HTTP API integration. The Step Functions API Gateway REST API integration that provides the option for Edge-optimized APIs are not supported.
- Unsupported by the Step Functions API Gateway integration:
 - **Authorizers:** Amazon Cognito, Native Open ID Connect / OAuth 2.0, Authorization header for token-based Lambda authorizers
 - **API types:** Private
 - **API management:** Custom domain names

For more information about API Gateway and its HTTP and REST APIs, see the following.

- The [Amazon API Gateway concepts](#) page.
- [Choosing between HTTP APIs and REST APIs](#) in the API Gateway developer guide.

Request format

When you create your Task state definition, Step Functions validates the parameters, builds the necessary URL to perform the call, then calls the API. The response includes the HTTP status code, headers and response body. The request format has both required and optional parameters.

Required request parameters

- `ApiEndpoint`
 - Type: String
 - The hostname of an API Gateway URL. The format is `<API ID>.execute-api.<region>.amazonaws.com`.

The API ID can only contain a combination of the following alphanumeric characters:

0123456789abcdefghijklmnopqrstuvwxyz

- Method
 - Type: Enum
 - The HTTP method, which must be one of the following:
 - GET
 - POST
 - PUT
 - DELETE
 - PATCH
 - HEAD
 - OPTIONS

Optional request parameters

- Headers
 - Type: JSON
 - HTTP headers allow a list of values associated with the same key.
- Stage
 - Type: String
 - The name of the stage where the API is deployed to in API Gateway. It's optional for any HTTP API that uses the `$default` stage.
- Path
 - Type: String
 - Path parameters that are appended after the API endpoint.
- QueryParameters
 - Type: JSON
 - Query strings only allow a list of values associated with the same key.
- RequestBody
 - Type: JSON or String

- The HTTP Request body. Its type can be either a JSON object or `String`. `RequestBody` is only supported for PATCH, POST, and PUT HTTP methods.
- `AllowNullValues`
 - Type: `BOOLEAN` – default value: `false`
 - With the default setting, any **null** values in the request input state will **not** be sent to your API. In the following example, the `category` field will **not** be included in the request, unless `AllowNullValues` is set to `true` in your state machine definition.

```
{
  "NewPet": {
    "type": "turtle",
    "price": 123,
    "category": null
  }
}
```

Note

By default, fields with **null** values in the request input state will **not** be sent to your API. You can force null values to be sent to your API by setting `AllowNullValues` to `true` in your state machine definition.

- `AuthType`
 - Type: `JSON`
 - The authentication method. The default method is `NO_AUTH`. The allowed values are:
 - `NO_AUTH`
 - `IAM_ROLE`
 - `RESOURCE_POLICY`

See **Authentication and authorization** for more information.

Note

For security considerations, the following HTTP header keys are not currently permitted:

- Anything prefixed with `X-Forwarded`, `X-Amz` or `X-Amzn`.

- Authorization
- Connection
- Content-md5
- Expect
- Host
- Max-Forwards
- Proxy-Authenticate
- Server
- TE
- Transfer-Encoding
- Trailer
- Upgrade
- Via
- Www-Authenticate

The following code example shows how to invoke API Gateway using Step Functions.

```
{
  "Type": "Task",
  "Resource": "arn:aws:states:::apigateway:invoke",
  "Parameters": {
    "ApiEndpoint": "example.execute-api.us-east-1.amazonaws.com",
    "Method": "GET",
    "Headers": {
      "key": ["value1", "value2"]
    },
    "Stage": "prod",
    "Path": "bills",
    "QueryParameters": {
      "billId": ["123456"]
    },
    "RequestBody": {},
    "AuthType": "NO_AUTH"
  }
}
```

Authentication and authorization

You can use the following authentication methods:

- **No authorization:** Call the API directly with no authorization method.
- **IAM role:** With this method, Step Functions assumes the role of the state machine, signs the request with [Signature Version 4 \(SigV4\)](#), then calls the API.
- **Resource policy:** Step Functions authenticates the request, and then calls the API. You must attach a resource policy to the API which specifies the following:
 1. The state machine that will invoke API Gateway.

Important

You must specify your state machine to limit access to it. If you do not, then any state machine that authenticates its API Gateway request with **Resource policy** authentication to your API will be granted access.

2. That Step Functions is the service calling API Gateway: "Service":
"states.amazonaws.com".
3. The resource you want to access, including:
 - The *region*.
 - The *account-id* in the specified region.
 - The *api-id*.
 - The *stage-name*.
 - The *HTTP-VERB* (method).
 - The *resource-path-specifier*.

For an example resource policy, see [IAM policies for Step Functions and API Gateway](#).

For more information on the resource format, see [Resource format of permissions for executing API in API Gateway](#) in the API Gateway Developer Guide.

Note

Resource policies are only supported for the REST API.

Service integration patterns

The API Gateway integration supports two service integration patterns:

- [Request Response](#), which is the default integration pattern. It lets Step Functions progress to the next step immediately after it receives an HTTP response.
- [Wait for a Callback with the Task Token](#) (`.waitForTaskToken`), which waits until a task token is returned with a payload. To use the `.waitForTaskToken` pattern, append `.waitForTaskToken` to the end of the **Resource** field of your task definition as shown in the following example:

```
{
  "Type": "Task",
  "Resource": "arn:aws:states:::apigateway:invoke.waitForTaskToken",
  "Parameters": {
    "ApiEndpoint": "example.execute-api.us-east-1.amazonaws.com",
    "Method": "POST",
    "Headers": {
      "TaskToken.$": "States.Array($.Task.Token)"
    },
    "Stage": "prod",
    "Path": "bills/add",
    "QueryParameters": {},
    "RequestBody": {
      "billId": "my-new-bill"
    },
    "AuthType": "IAM_ROLE"
  }
}
```

Output format

The following output parameters are provided:

Name	Type	Description
ResponseBody	JSON or String	The response body of the API call.
Headers	JSON	The response headers.
StatusCode	Integer	The HTTP status code of the response.
StatusText	String	The status text of the response.

An example response:

```
{
  "ResponseBody": {
    "myBills": []
  },
  "Headers": {
    "key": ["value1", "value2"]
  },
  "StatusCode": 200,
  "StatusText": "OK"
}
```

Error handling

When an error occurs, an error and cause is returned as follows:

- If the HTTP status code is available, then the error will be returned in the format `ApiGateway.<HTTP Status Code>`.
- If the HTTP status code is not available, then the error will be returned in the format `ApiGateway.<Exception>`.

In both cases, the cause is returned as a string.

The following example shows a response where an error has occurred:

```
{
  "error": "ApiGateway.403",
  "cause": "{\"message\":\"Missing Authentication Token\"}"
}
```

Note

A status code of 2XX indicates success, and no error will be returned. All other status codes or thrown exceptions will result in an error.

For more information see:

- [Amazon API Gateway concepts](#) in the API Gateway Developer Guide.
- [IAM policies for Amazon API Gateway](#)
- A sample project that shows how to [Make a call to API Gateway](#)

[Amazon API Gateway concepts](#) in the API Gateway Developer Guide.

Call Athena with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

Note **How the Optimized Athena integration is different than the Athena AWS SDK integration**

- The [Run a Job \(.sync\)](#) integration pattern is supported.
- There are no optimizations for the [Request Response](#) integration pattern.
- The [Wait for a Callback with the Task Token](#) integration pattern is not supported.

The AWS Step Functions service integration with Amazon Athena enables you to use Step Functions to start and stop query execution, and get query results. Using Step Functions, you can run ad-hoc or scheduled data queries, and retrieve results targeting your S3 data lakes. Athena is serverless, so there is no infrastructure to set up or manage, and you pay only for the queries you run.

To integrate AWS Step Functions with Amazon Athena, you use the provided Athena service integration APIs.

The service integration APIs are the same as the corresponding Athena APIs. Not all APIs support all integration patterns, as shown in the following table.

API	Request Response	Run a Job (.sync)
StartQueryExecution	✓	✓
StopQueryExecution	✓	
GetQueryExecution	✓	
GetQueryResults	✓	

Supported Amazon Athena APIs:

Note

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 256 KB of data as a UTF-8 encoded string when you send to, or receive data from, another service. See [Quotas related to state machine executions](#).

- [StartQueryExecution](#)
 - [Request syntax](#)
 - Supported parameters:
 - [ClientRequestToken](#)
 - [ExecutionParameters](#)
 - [QueryExecutionContext](#)
 - [QueryString](#)
 - [ResultConfiguration](#)
 - [WorkGroup](#)
 - [Response syntax](#)
- [StopQueryExecution](#)

- [Request syntax](#)
- Supported parameters:
 - [QueryExecutionId](#)
- [GetQueryExecution](#)
 - [Request syntax](#)
 - Supported parameters:
 - [QueryExecutionId](#)
 - [Response syntax](#)
- [GetQueryResults](#)
 - [Request syntax](#)
 - Supported parameters:
 - [MaxResults](#)
 - [NextToken](#)
 - [QueryExecutionId](#)
 - [Response syntax](#)

The following includes a Task state that starts an Athena query.

```
"Start an Athena query": {
  "Type": "Task",
  "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
  "Parameters": {
    "QueryString": "SELECT * FROM \"myDatabase\".\"myTable\" limit 1",
    "WorkGroup": "primary",
    "ResultConfiguration": {
      "OutputLocation": "s3://athenaQueryResult"
    }
  },
  "Next": "Get results of the query"
}
```

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Manage AWS Batch with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

i How the Optimized AWS Batch integration is different than the AWS Batch AWS SDK integration

- The [Run a Job \(.sync\)](#) integration pattern is available.

Note that there are no optimizations for the [Request Response](#) or [Wait for a Callback with the Task Token](#) integration patterns.

Supported AWS Batch APIs:

- [SubmitJob](#)
 - [Request syntax](#)
 - Supported parameters:
 - [ArrayProperties](#)
 - [ContainerOverrides](#)
 - [DependsOn](#)
 - [JobDefinition](#)
 - [JobName](#)
 - [JobQueue](#)
 - [Parameters](#)
 - [RetryStrategy](#)
 - [Timeout](#)
 - [Tags](#)
 - [Response syntax](#)

Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

The following includes a Task state that submits an AWS Batch job and waits for it to complete.

```
{
  "StartAt": "BATCH_JOB",
  "States": {
    "BATCH_JOB": {
      "Type": "Task",
      "Resource": "arn:aws:states:::batch:submitJob.sync",
      "Parameters": {
        "JobDefinition": "preprocessing",
        "JobName": "PreprocessingBatchJob",
        "JobQueue": "SecondaryQueue",
        "Parameters.$": "$.batchjob.parameters",
        "ContainerOverrides": {
          "ResourceRequirements": [
            {
              "Type": "VCPU",
              "Value": "4"
            }
          ]
        }
      },
      "End": true
    }
  }
}
```

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Call Amazon Bedrock with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

Topics

- [Amazon Bedrock service integration APIs](#)
- [Task state definition for Amazon Bedrock integration](#)

Amazon Bedrock service integration APIs

To integrate AWS Step Functions with Amazon Bedrock, you can use the following APIs. These APIs are similar to the corresponding Amazon Bedrock APIs, with some differences in the request fields that are passed.

The following table describes the differences between each service integration API and its corresponding Amazon Bedrock API.

Amazon Bedrock service integration APIs and corresponding Amazon Bedrock APIs

Amazon Bedrock service integration API	Corresponding Amazon Bedrock API	Differences
<p><i>InvokeModel</i></p> <p>Invokes the specified Amazon Bedrock model to run inference using the input you provide in the request body. You use <code>InvokeModel</code> to run inference for text models, image models, and embedding models.</p>	<p>InvokeModel</p>	<p>The Amazon Bedrock service integration API request body includes the following additional parameters.</p> <ul style="list-style-type: none"> • Body – Specifies input data in the format specified in the content-type request header. Body contains parameters specific to the target model. <p>If you use the <code>InvokeModel</code> API, you must specify the <code>Body</code> parameter. Step Functions doesn't validate the input you provide in <code>Body</code>.</p> <p>When you specify <code>Body</code> using the Amazon Bedrock</p>

Amazon Bedrock service integration API	Corresponding Amazon Bedrock API	Differences
		<p>optimized integration, you can specify a payload of up to 256 KB. If your payload exceeds 256 KB, we recommend that you use Input.</p> <ul style="list-style-type: none"> • Input – Specifies the source to retrieve the input data from. This optional field is specific to Amazon Bedrock optimized integration with Step Functions. In this field, you can specify an <code>S3Uri</code>. <p>You can specify either <code>Body</code> in the <code>Parameters</code> or <code>Input</code>, but not both.</p> <p>When you specify <code>Input</code> without specifying <code>ContentType</code>, the content type of the input data source becomes the value for <code>ContentType</code>.</p> <ul style="list-style-type: none"> • Output – Specifies the destination where the API response is written. This optional field is specific to Amazon Bedrock optimized integration with Step Functions. In this field, you can specify an <code>S3Uri</code>.

Amazon Bedrock service integration API	Corresponding Amazon Bedrock API	Differences
		<p>If you specify this field, the API response body is replaced with a reference to the Amazon S3 location of the original output.</p> <p>The following example shows the syntax for InvokeModel API for Amazon Bedrock integration.</p> <pre data-bbox="1071 772 1507 1528"> { "ModelId": String, // required "Accept": String, // default: application/json "ContentType": String, // default: application/json "Input": { // not from Bedrock API "S3Uri": String }, "Output": { // not from Bedrock API "S3Uri": String } } </pre>
<p><i>CreateModelCustomizationJob</i></p> <p>Creates a fine-tuning job to customize a base model.</p>	<p>CreateModelCustomizationJob</p>	<p>None</p>

Amazon Bedrock service integration API	Corresponding Amazon Bedrock API	Differences
<p><i>CreateModelCustomizationJob</i>.sync</p> <p>Creates a fine-tuning job to customize a base model.</p>	<p>CreateModelCustomizationJob</p>	None

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Task state definition for Amazon Bedrock integration

The following Task state definition shows how you can integrate with Amazon Bedrock in your state machines. This example shows a Task state that extracts the full result of model invocation specified by the path, `result_one`. This is based on [Inference parameters for foundation models](#). This example uses the Cohere Command large language model (LLM).

```
{
  "Type": "Task",
  "Resource": "arn:aws:states:::bedrock:invokeModel",
  "Parameters": {
    "ModelId": "cohere.command-text-v14",
    "Body": {
      "prompt.$": "$.prompt_one",
      "max_tokens": 250
    },
    "ContentType": "application/json",
    "Accept": "*/*"
  },
  "ResultPath": "$.result_one",
  "ResultSelector": {
    "result_one.$": "$.Body.generations[0].text"
  },
  "End": true
}
```

Tip

To deploy an example of a state machine that integrates with Amazon Bedrock to your AWS account, see [Perform AI prompt-chaining with Amazon Bedrock](#).

Call AWS CodeBuild with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

How the Optimized CodeBuild integration is different than the CodeBuild AWS SDK integration

- The [Run a Job \(.sync\)](#) integration pattern is supported.
- After you call `StopBuild` or `StopBuildBatch`, the build or build batch is not immediately deletable until some internal work is completed within CodeBuild to finalize the state of the build or builds. If you attempt to use `BatchDeleteBuilds` or `DeleteBuildBatch` during this period, the build or build batch may not be deleted. The optimized service integrations for `BatchDeleteBuilds` and `DeleteBuildBatch` include an internal retry to simplify the use case of deleting immediately after stopping.

The AWS Step Functions service integration with AWS CodeBuild enables you to use Step Functions to trigger, stop, and manage builds, and to share build reports. Using Step Functions, you can design and run continuous integration pipelines for validating your software changes for applications.

Not all APIs support all integration patterns, as shown in the following table.

API	Request Response	Run a Job (.sync)
StartBuild	✓	✓
StopBuild	✓	
BatchDeleteBuilds	✓	

API	Request Response	Run a Job (.sync)
BatchGetReports	✓	
StartBuildBatch	✓	✓
StopBuildBatch	✓	
RetryBuildBatch	✓	✓
DeleteBuildBatch	✓	

ⓘ Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

Supported CodeBuild APIs and syntax:

- [StartBuild](#)
 - [Request syntax](#)
 - Supported parameters:
 - [ProjectName](#)
 - [ArtifactsOverride](#)
 - [BuildspecOverride](#)
 - [CacheOverride](#)
 - [CertificateOverride](#)
 - [ComputeTypeOverride](#)
 - [EncryptionKeyOverride](#)
 - [EnvironmentTypeOverride](#)
 - [EnvironmentVariablesOverride](#)
 - [GitCloneDepthOverride](#)
 - [GitSubmodulesConfigOverride](#)

- [IdempotencyToken](#)
- [ImageOverride](#)
- [ImagePullCredentialsTypeOverride](#)
- [InsecureSslOverride](#)
- [LogsConfigOverride](#)
- [PrivilegedModeOverride](#)
- [QueuedTimeoutInMinutesOverride](#)
- [RegistryCredentialOverride](#)
- [ReportBuildStatusOverride](#)
- [SecondaryArtifactsOverride](#)
- [SecondarySourcesOverride](#)
- [SecondarySourcesVersionOverride](#)
- [ServiceRoleOverride](#)
- [SourceAuthOverride](#)
- [SourceLocationOverride](#)
- [SourceTypeOverride](#)
- [SourceVersion](#)
- [TimeoutInMinutesOverride](#)
- [Response syntax](#)
- [StopBuild](#)
 - [Request syntax](#)
 - Supported parameters:
 - [Id](#)
 - [Response syntax](#)
- [BatchDeleteBuilds](#)
 - [Request syntax](#)
 - Supported parameters:
 - [Ids](#)
 - [Response syntax](#)
- [BatchGetReports](#)

- [Request syntax](#)
- Supported parameters:
 - [ReportArns](#)
- [Response syntax](#)
- [StartBuildBatch](#)
 - [Request syntax](#)
 - Supported parameters:
 - [ProjectName](#)
 - [ArtifactsOverride](#)
 - [BuildBatchConfigOverride](#)
 - [BuildspecOverride](#)
 - [BuildTimeoutInMinutesOverride](#)
 - [CacheOverride](#)
 - [CertificateOverride](#)
 - [ComputeTypeOverride](#)
 - [DebugSessionEnabled](#)
 - [EncryptionKeyOverride](#)
 - [EnvironmentTypeOverride](#)
 - [EnvironmentVariablesOverride](#)
 - [GitCloneDepthOverride](#)
 - [GitSubmodulesConfigOverride](#)
 - [IdempotencyToken](#)
 - [ImageOverride](#)
 - [ImagePullCredentialsTypeOverride](#)
 - [InsecureSslOverride](#)
 - [LogsConfigOverride](#)
 - [PrivilegedModeOverride](#)
 - [QueuedTimeoutInMinutesOverride](#)
 - [RegistryCredentialOverride](#)
 - [ReportBuildBatchStatusOverride](#)

- [SecondaryArtifactsOverride](#)
- [SecondarySourcesOverride](#)
- [SecondarySourcesVersionOverride](#)
- [ServiceRoleOverride](#)
- [SourceAuthOverride](#)
- [SourceLocationOverride](#)
- [SourceTypeOverride](#)
- [SourceVersion](#)
- [Response syntax](#)
- [StopBuildBatch](#)
 - [Request syntax](#)
 - Supported parameters:
 - [Id](#)
 - [Response syntax](#)
- [RetryBuildBatch](#)
 - [Request syntax](#)
 - Supported parameters:
 - [Id](#)
 - [IdempotencyToken](#)
 - [RetryType](#)
 - [Response syntax](#)
- [DeleteBuildBatch](#)
 - [Request syntax](#)
 - Supported parameters:
 - [Id](#)
 - [Response syntax](#)

Note

You can use the JSONPath recursive descent (`..`) operator for `BatchDeleteBuilds`. This returns an array, and enables you to turn the `Arn` field from `StartBuild` into a plural `Ids` parameter, as shown in the following example.

```
"BatchDeleteBuilds": {
  "Type": "Task",
  "Resource": "arn:aws:states:::codebuild:batchDeleteBuilds",
  "Parameters": {
    "Ids.$": "$.Build..Arn"
  },
  "Next": "MyNextState"
},
```

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Call DynamoDB APIs with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

Note

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 256 KB of data as a UTF-8 encoded string when you send to, or receive data from, another service. See [Quotas related to state machine executions](#).

Note **How the optimized DynamoDB integration is different than the DynamoDB AWS SDK integration**

- There is no optimization for the [Request Response](#) integration pattern.
- The [Wait for a Callback with the Task Token](#) integration pattern is not supported.

- Only [GetItem](#), [PutItem](#), [UpdateItem](#), and [DeleteItem](#) API actions are available through optimized integration. Other API actions, such as [CreateTable](#) are available using the DynamoDB AWS SDK integration.

Supported Amazon DynamoDB APIs and syntax:

- [GetItem](#)
 - [Request syntax](#)
 - Supported parameters:
 - [Key](#)
 - [TableName](#)
 - [AttributesToGet](#)
 - [ConsistentRead](#)
 - [ExpressionAttributeNames](#)
 - [ProjectionExpression](#)
 - [ReturnConsumedCapacity](#)
 - [Response syntax](#)
- [PutItem](#)
 - [Request syntax](#)
 - Supported parameters:
 - [Item](#)
 - [TableName](#)
 - [ConditionalOperator](#)
 - [ConditionExpression](#)
 - [Expected](#)
 - [ExpressionAttributeNames](#)
 - [ExpressionAttributeValue](#)
 - [ReturnConsumedCapacity](#)
 - [ReturnItemCollectionMetrics](#)
 - [ReturnValues](#)
 - [Response syntax](#)

- [DeleteItem](#)
 - [Request syntax](#)
 - Supported parameters:
 - [Key](#)
 - [TableName](#)
 - [ConditionalOperator](#)
 - [ConditionExpression](#)
 - [Expected](#)
 - [ExpressionAttributeNames](#)
 - [ExpressionAttributeValues](#)
 - [ReturnConsumedCapacity](#)
 - [ReturnItemCollectionMetrics](#)
 - [ReturnValues](#)
 - [Response syntax](#)
- [UpdateItem](#)
 - [Request syntax](#)
 - Supported parameters:
 - [Key](#)
 - [TableName](#)
 - [AttributeUpdates](#)
 - [ConditionalOperator](#)
 - [ConditionExpression](#)
 - [Expected](#)
 - [ExpressionAttributeNames](#)
 - [ExpressionAttributeValues](#)
 - [ReturnConsumedCapacity](#)
 - [ReturnItemCollectionMetrics](#)
 - [ReturnValues](#)
- [UpdateExpression](#)
 - [Response syntax](#)

Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

The following is a Task state that retrieves a message from DynamoDB.

```
"Read Next Message from DynamoDB": {
  "Type": "Task",
  "Resource": "arn:aws:states:::dynamodb:getItem",
  "Parameters": {
    "TableName": "TransferDataRecords-DDBTable-3I41R5L5EAGT",
    "Key": {
      "MessageId": {"S.$": "$.List[0]"}
    }
  },
  "ResultPath": "$.DynamoDB",
  "Next": "Send Message to SQS"
},
```

To see this state in a working example, see the [Transfer data records \(Lambda, DynamoDB, Amazon SQS\)](#) sample project.

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Manage Amazon ECS or Fargate Tasks with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

How the Optimized Amazon ECS/Fargate integration is different than the Amazon ECS or Fargate AWS SDK integration

- The [Run a Job \(.sync\)](#) integration pattern is supported.
- `ecs:runTask` can return an HTTP 200 response, but have a non-empty `Failures` field as follows:

- **Request Response:** Return the response and do not fail the task. This is the same as no optimization.
- **Run a Job or Task Token:** If a non-empty `Failures` field is encountered, the task is failed with an `AmazonECS.Unknown` error.

Supported Amazon ECS/Fargate APIs and syntax:

Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as `StateMachineArn`.

- [RunTask](#) starts a new task using the specified task definition.
 - [Request syntax](#)
 - Supported parameters:
 - [Cluster](#)
 - [Group](#)
 - [LaunchType](#)
 - [NetworkConfiguration](#)
 - [Overrides](#)
 - [PlacementConstraints](#)
 - [PlacementStrategy](#)
 - [PlatformVersion](#)
 - [PropagateTags](#)
 - [TaskDefinition](#)
 - [EnableExecuteCommand](#)
 - [Response syntax](#)

Passing Data to an Amazon ECS Task

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

You can use overrides to override the default command for a container, and pass input to your Amazon ECS tasks. See [ContainerOverride](#). In the example, we have used JsonPath to pass values to the Task from the input to the Task state.

The following includes a Task state that runs an Amazon ECS task and waits for it to complete.

```
{
  "StartAt": "Run an ECS Task and wait for it to complete",
  "States": {
    "Run an ECS Task and wait for it to complete": {
      "Type": "Task",
      "Resource": "arn:aws:states:::ecs:runTask.sync",
      "Parameters": {
        "Cluster": "cluster-arn",
        "TaskDefinition": "job-id",
        "Overrides": {
          "ContainerOverrides": [
            {
              "Name": "container-name",
              "Command.$": "$.commands"
            }
          ]
        }
      },
      "End": true
    }
  }
}
```

The "Command.\$": "\$.commands" line in ContainerOverrides passes the commands from the state input to the container.

For the previous example, each of the commands will be passed as a container override if the input to the execution is the following.

```
{
```

```

"commands": [
  "test command 1",
  "test command 2",
  "test command 3"
]
}

```

The following includes a Task state that runs an Amazon ECS task, and then waits for the task token to be returned. See [Wait for a Callback with the Task Token](#).

```

{
  "StartAt": "Manage ECS task",
  "States": {
    "Manage ECS task": {
      "Type": "Task",
      "Resource": "arn:aws:states:::ecs:runTask.waitForTaskToken",
      "Parameters": {
        "LaunchType": "FARGATE",
        "Cluster": "cluster-arn",
        "TaskDefinition": "job-id",
        "Overrides": {
          "ContainerOverrides": [
            {
              "Name": "container-name",
              "Environment": [
                {
                  "Name": "TASK_TOKEN_ENV_VARIABLE",
                  "Value.$": "$$.Task.Token"
                }
              ]
            }
          ]
        }
      }
    }
  }
},
  "End": true
}
}
}

```

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Call Amazon EKS with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

How the Optimized Amazon EKS integration is different than the Amazon EKS AWS SDK integration

- The [Run a Job \(.sync\)](#) integration pattern is supported.
- There are no optimizations for the [Request Response](#) integration pattern.
- The [Wait for a Callback with the Task Token](#) integration pattern is not supported.

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Step Functions provides two types of service integration APIs for integrating with Amazon Elastic Kubernetes Service. One lets you use the Amazon EKS APIs to create and manage an Amazon EKS cluster. The other lets you interact with your cluster using the Kubernetes API and run jobs as part of your application's workflow. You can use the Kubernetes API integrations with Amazon EKS clusters created using Step Functions, with Amazon EKS clusters created by the `eksctl` tool or the [Amazon EKS console](#), or similar methods. For more information, see [Creating an Amazon EKS cluster](#) in the Amazon EKS User Guide.

Note

The Step Functions EKS integration supports only Kubernetes APIs with public endpoint access. By default, EKS clusters API server endpoints have public access. For more information, see [Amazon EKS cluster endpoint access control](#) in the Amazon EKS User Guide.

Step Functions does not terminate an Amazon EKS cluster automatically if execution is stopped. If your state machine stops before your Amazon EKS cluster has terminated, your cluster may continue running indefinitely, and can accrue additional charges. To avoid this, ensure that any Amazon EKS cluster you create is terminated properly. For more information, see:

- [Deleting a cluster](#) in the Amazon EKS User Guide.

- [Run a Job \(.sync\)](#) in Service Integration Patterns.

Note

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 256 KB of data as a UTF-8 encoded string when you send to, or receive data from, another service. See [Quotas related to state machine executions](#).

Kubernetes API integrations

Step Functions supports the following Kubernetes APIs:

RunJob

The `eks:runJob` service integration allows you to run a job on your Amazon EKS cluster. The `eks:runJob.sync` variant allows you to wait for the job to complete, and, optionally retrieve logs.

Your Kubernetes API server must grant permissions to the IAM role used by your state machine. For more information, see [Permissions](#).

For the **Run a Job (.sync)** pattern, the status of the job is determined by polling. Step Functions initially polls at a rate of approximately 1 poll per minute. This rate eventually slows to approximately 1 poll every 5 minutes. If you require more frequent polling, or require more control over the polling strategy, you can use the `eks:call` integration to query the status of the job.

The `eks:runJob` integration is specific to `batch/v1` Kubernetes Jobs. For more information, see [Jobs](#) in the Kubernetes documentation. If you want to manage other Kubernetes resources, including custom resources, use the `eks:call` service integration. You can use Step Functions to build polling loops, as demonstrated in the [the section called “Poll for Job Status \(Lambda, AWS Batch\)”](#) sample project.

Supported parameters include:

- `ClusterName`: The name of the Amazon EKS cluster you want to call.
 - Type: `String`
 - Required: `yes`

- **CertificateAuthority:** The Base64-encoded certificate data required to communicate with your cluster. You can obtain this value from the [Amazon EKS console](#) or by using the Amazon EKS [DescribeCluster](#) API.
 - Type: String
 - Required: yes
- **Endpoint:** The endpoint URL for your Kubernetes API server. You can obtain this value from the [Amazon EKS console](#) or by using the Amazon EKS [DescribeCluster](#) API.
 - Type: String
 - Required: yes
- **Namespace:** The namespace in which to run the job. If not provided, the namespace default is used.
 - Type: String
 - Required: no
- **Job:** The definition of the Kubernetes Job. See [Jobs](#) in the Kubernetes documentation.
 - Type: JSON or String
 - Required: yes
- **LogOptions:** A set of options to control the optional retrieval of logs. Only applicable if the Run a Job (.sync) service integration pattern is used to wait for the completion of the job.
 - Type: JSON
 - Required: no
 - Logs are included in the response under the key `logs`. There may be multiple pods within the job, each with multiple containers.

```
{
  ...
  "logs": {
    "pods": {
      "pod1": {
        "containers": {
          "container1": {
            "log": <Log>
          },
          ...
        }
      },
      ...
    }
  },
  ...
}
```

```

    ...
  }
}

```

- Log retrieval is performed on a best-effort basis. If there is an error retrieving a log, in place of the log field there will be the fields `error` and `cause`.
- `LogOptions.RetrieveLogs`: Enable log retrieval after the job completes. By default, logs are not retrieved.
 - Type: Boolean
 - Required: no
- `LogOptions.RawLogs`: If `RawLogs` is set to true, logs will be returned as raw strings without attempting to parse them into JSON. By default, logs are deserialized into JSON if possible. In some cases such parsing can introduce unwanted changes, such as limiting the precision of numbers containing many digits.
 - Type: Boolean
 - Required: no
- `LogOptions.LogParameters`: The Kubernetes API's Read Log API supports query parameters to control log retrieval. For example, you can use `tailLines` or `limitBytes` to limit the size of retrieved logs and remain within the Step Functions data size quota. For more information, see the [Read Log](#) section of the Kubernetes API Reference.
 - Type: Map of String to List of Strings
 - Required: no
 - Example:

```

"LogParameters": {
  "tailLines": [ "6" ]
}

```

The following example includes a Task state that runs a job, waits for it to complete, then retrieves the job's logs:

```

{
  "StartAt": "Run a job on EKS",
  "States": {
    "Run a job on EKS": {
      "Type": "Task",

```

```
"Resource": "arn:aws:states:::eks:runJob.sync",
"Parameters": {
  "ClusterName": "MyCluster",
  "CertificateAuthority": "ANPAJ2UCCR6DPCEXAMPLE",
  "Endpoint": "https://AKIAIOSFODNN7EXAMPLE.y14.us-east-1.eks.amazonaws.com",
  "LogOptions": {
    "RetrieveLogs": true
  },
  "Job": {
    "apiVersion": "batch/v1",
    "kind": "Job",
    "metadata": {
      "name": "example-job"
    },
    "spec": {
      "backoffLimit": 0,
      "template": {
        "metadata": {
          "name": "example-job"
        },
        "spec": {
          "containers": [
            {
              "name": "pi-2000",
              "image": "perl",
              "command": [ "perl" ],
              "args": [
                "-Mbignum=bpi",
                "-wle",
                "print bpi(2000)"
              ]
            }
          ],
          "restartPolicy": "Never"
        }
      }
    }
  },
  "End": true
}
```

Call

The `eks:call` service integration allows you to use the Kubernetes API to read and write Kubernetes resource objects via a Kubernetes API endpoint.

Your Kubernetes API server must grant permissions to the IAM role used by your state machine. For more information, see [Permissions](#).

For more information about the available operations, see the [Kubernetes API Reference](#).

Supported parameters for `Call` include:

- **ClusterName:** The name of the Amazon EKS cluster you want to call.
 - Type: `String`
 - Required: Yes
- **CertificateAuthority:** The Base64-encoded certificate data required to communicate with your cluster. You can obtain this value from the [Amazon EKS console](#) or by using the Amazon EKS [DescribeCluster](#) API.
 - Type: `String`
 - Required: Yes
- **Endpoint:** The endpoint URL for your Kubernetes API server. You can find this value on the [Amazon EKS console](#) or by using Amazon EKS' `DescribeCluster` API.
 - Type: `String`
 - Required: Yes
- **Method:** The HTTP method of your request. One of: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, or `PATCH`.
 - Type: `String`
 - Required: Yes
- **Path:** The HTTP path of the Kubernetes REST API operation.
 - Type: `String`
 - Required: Yes
- **QueryParameters:** The HTTP query parameters of the Kubernetes REST API operation.
 - Type: `Map of String to List of Strings`
 - Required: No
 - **Example:**


```
"QueryParameters": {
  "labelSelector": [ "job-name=example-job" ]
}
```

- **RequestBody:** The HTTP message body of the Kubernetes REST API operation.
 - **Type:** JSON or String
 - **Required:** No

The following includes a Task state that uses `eks:call` to list the pods belonging to the job `example-job`.

```
{
  "StartAt": "Call EKS",
  "States": {
    "Call EKS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:call",
      "Parameters": {
        "ClusterName": "MyCluster",
        "CertificateAuthority": "ANPAJ2UCCR6DPCEXAMPLE",
        "Endpoint": "https://444455556666.y14.us-east-1.eks.amazonaws.com",
        "Method": "GET",
        "Path": "/api/v1/namespaces/default/pods",
        "QueryParameters": {
          "labelSelector": [
            "job-name=example-job"
          ]
        }
      },
      "End": true
    }
  }
}
```

The following includes a Task state that uses `eks:call` to delete the job `example-job`, and sets the `propagationPolicy` to ensure the job's pods are also deleted.

```
{
  "StartAt": "Call EKS",
  "States": {
```

```
"Call EKS": {
  "Type": "Task",
  "Resource": "arn:aws:states:::eks:call",
  "Parameters": {
    "ClusterName": "MyCluster",
    "CertificateAuthority": "ANPAJ2UCCR6DPCEXAMPLE",
    "Endpoint": "https://4444455556666.y14.us-east-1.eks.amazonaws.com",
    "Method": "DELETE",
    "Path": "/apis/batch/v1/namespaces/default/jobs/example-job",
    "QueryParameters": {
      "propagationPolicy": [
        "Foreground"
      ]
    }
  },
  "End": true
}
```

Supported Amazon EKS APIs

Supported Amazon EKS APIs and syntax include:

- [CreateCluster](#)
- [Request syntax](#)
- [Response syntax](#)

When an Amazon EKS cluster is created using the `eks:createCluster` service integration, the IAM role is added to the Kubernetes RBAC authorization table as the administrator (with `system:masters` permissions). Initially, only that IAM entity can make calls to the Kubernetes API server. For more information, see:

- [Managing users or IAM roles for your cluster](#) in the *Amazon EKS User Guide*
- The [Permissions](#) section

Amazon EKS uses service-linked roles which contain the permissions Amazon EKS requires to call other services on your behalf. If these service-linked roles do not exist in your account already, you must add the `iam:CreateServiceLinkedRole` permission to the IAM role used by Step Functions. For more information, see [Using Service-Linked Roles](#) in the *Amazon EKS User Guide*.

The IAM role used by Step Functions must have `iam:PassRole` permissions to pass the cluster IAM role to Amazon EKS. For more information, see [Amazon EKS cluster IAM role](#) in the *Amazon EKS User Guide*.

- [DeleteCluster](#)
 - [Request syntax](#)
 - [Response syntax](#)

You must delete any Fargate profiles or node groups before deleting a cluster.

- [CreateFargateProfile](#)
 - [Request syntax](#)
 - [Response syntax](#)

Amazon EKS uses service-linked roles which contain the permissions Amazon EKS requires to call other services on your behalf. If these service-linked roles do not exist in your account already, you must add the `iam:CreateServiceLinkedRole` permission to the IAM role used by Step Functions. For more information, see [Using Service-Linked Roles](#) in the *Amazon EKS User Guide*.

Amazon EKS on Fargate may not be available in all regions. For information on region availability, see the section on [Fargate](#) in the *Amazon EKS User Guide*.

The IAM role used by Step Functions must have `iam:PassRole` permissions to pass the pod execution IAM role to Amazon EKS. For more information, see [Pod execution role](#) in the *Amazon EKS User Guide*.

- [DeleteFargateProfile](#)
 - [Request syntax](#)
 - [Response syntax](#)
- [CreateNodegroup](#)
 - [Request syntax](#)
 - [Response syntax](#)

Amazon EKS uses service-linked role whichs contain the permissions Amazon EKS requires to call other services on your behalf. If these service-linked roles do not exist in your account already, you must add the `iam:CreateServiceLinkedRole` permission to the IAM role used

by Step Functions. For more information, see [Using Service-Linked Roles](#) in the *Amazon EKS User Guide*.

The IAM role used by Step Functions must have `iam:PassRole` permissions to pass the node IAM role to Amazon EKS. For more information, see [Using Service-Linked Roles](#) in the *Amazon EKS User Guide*.

- [DeleteNodegroup](#)
- [Request syntax](#)
- [Response syntax](#)

The following includes a Task that creates an Amazon EKS cluster.

```
{
  "StartAt": "CreateCluster.sync",
  "States": {
    "CreateCluster.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:createCluster.sync",
      "Parameters": {
        "Name": "MyCluster",
        "ResourcesVpcConfig": {
          "SubnetIds": [
            "subnet-053e7c47012341234",
            "subnet-027cfea4b12341234"
          ]
        },
        "RoleArn": "arn:aws:iam::123456789012:role/MyEKSClusterRole"
      },
      "End": true
    }
  }
}
```

The following includes a Task state that deletes an Amazon EKS cluster.

```
{
  "StartAt": "DeleteCluster.sync",
  "States": {
    "DeleteCluster.sync": {
      "Type": "Task",
```

```

    "Resource": "arn:aws:states:::eks:deleteCluster.sync",
    "Parameters": {
      "Name": "MyCluster"
    },
    "End": true
  }
}
}
}

```

The following includes a Task state that creates a Fargate profile.

```

{
  "StartAt": "CreateFargateProfile.sync",
  "States": {
    "CreateFargateProfile.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:createFargateProfile.sync",
      "Parameters": {
        "ClusterName": "MyCluster",
        "FargateProfileName": "MyFargateProfile",
        "PodExecutionRoleArn": "arn:aws:iam::123456789012:role/MyFargatePodExecutionRole",
        "Selectors": [{
          "Namespace": "my-namespace",
          "Labels": { "my-label": "my-value" }
        }]
      },
      "End": true
    }
  }
}
}
}

```

The following includes a Task state that deletes a Fargate profile.

```

{
  "StartAt": "DeleteFargateProfile.sync",
  "States": {
    "DeleteFargateProfile.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:deleteFargateProfile.sync",
      "Parameters": {
        "ClusterName": "MyCluster",
        "FargateProfileName": "MyFargateProfile"
      }
    }
  }
}
}
}

```

```
    },
    "End": true
  }
}
```

The following includes a Task state that creates a node group.

```
{
  "StartAt": "CreateNodegroup.sync",
  "States": {
    "CreateNodegroup.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:createNodegroup.sync",
      "Parameters": {
        "ClusterName": "MyCluster",
        "NodegroupName": "MyNodegroup",
        "NodeRole": "arn:aws:iam::123456789012:role/MyNodeInstanceRole",
        "Subnets": ["subnet-09fb51df01234", "subnet-027cfea4b1234"]
      },
      "End": true
    }
  }
}
```

The following includes a Task state that deletes a node group.

```
{
  "StartAt": "DeleteNodegroup.sync",
  "States": {
    "DeleteNodegroup.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:deleteNodegroup.sync",
      "Parameters": {
        "ClusterName": "MyCluster",
        "NodegroupName": "MyNodegroup"
      },
      "End": true
    }
  }
}
```

Permissions

When an Amazon EKS cluster is created using the `eks:createCluster` service integration, the IAM role is added to the Kubernetes RBAC authorization table as the administrator, with `system:masters` permissions. Initially, only that IAM entity can make calls to the Kubernetes API server. For example, you will not be able to use **kubectl** to interact with your Kubernetes API server, unless you assume the same role as your Step Functions state machine, or if you configure Kubernetes to grant permissions to additional IAM entities. For more information, see [Managing users or IAM roles for your cluster](#) in the *Amazon EKS User Guide*.

You can add permission for additional IAM entities, such as users or roles, by adding them to the `aws-auth` ConfigMap in the `kube-system` namespace. If you are creating your cluster from Step Functions, use the `eks:call` service integration.

The following includes a Task state that creates an `aws-auth` ConfigMap and grants `system:masters` permission to the user `arn:aws:iam::123456789012:user/my-user` and the IAM role `arn:aws:iam::123456789012:role/my-role`.

```
{
  "StartAt": "Add authorized user",
  "States": {
    "Add authorized user": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:call",
      "Parameters": {
        "ClusterName": "MyCluster",
        "CertificateAuthority": "LS0tLS1CRUd...UtLS0tLQo=",
        "Endpoint": "https://444455556666.yl4.us-east-1.eks.amazonaws.com",
        "Method": "POST",
        "Path": "/api/v1/namespaces/kube-system/configmaps",
        "RequestBody": {
          "apiVersion": "v1",
          "kind": "ConfigMap",
          "metadata": {
            "name": "aws-auth",
            "namespace": "kube-system"
          },
          "data": {
            "mapUsers": "[{ \"userarn\": \"arn:aws:iam::123456789012:user/my-user\",
            \"username\": \"my-user\", \"groups\": [ \"system:masters\" ] } ]",
            "mapRoles": "[{ \"rolearn\": \"arn:aws:iam::123456789012:role/my-role\",
            \"username\": \"my-role\", \"groups\": [ \"system:masters\" ] } ]"
```

```
    }
  }
},
"End": true
}
}
```

Note

You may see the ARN for an IAM role displayed in a format that includes the path **/service-role/**, such as `arn:aws:iam::123456789012:role/service-role/my-role`. This **service-role** path token should not be included when listing the role in `aws-auth`.

When your cluster is first created the `aws-auth` ConfigMap will not exist, but will be added automatically if you create a Fargate profile. You can retrieve the current value of `aws-auth`, add the additional permissions, and PUT a new version. It is usually easier to create `aws-auth` before the Fargate profile.

If your cluster was created outside of Step Functions, you can configure **kubectl** to communicate with your Kubernetes API server. Then, create a new `aws-auth` ConfigMap using `kubectl apply -f aws-auth.yaml` or edit one that already exists using `kubectl edit -n kube-system configmap/aws-auth`. For more information, see:

- [Create a kubeconfig for Amazon EKS](#) in the *Amazon EKS User Guide*.
- [Managing users or IAM roles for your cluster](#) in the *Amazon EKS User Guide*.

If your IAM role does not have sufficient permissions in Kubernetes, the `eks:call` or `eks:runJob` service integrations will fail with the following error:

```
Error:
EKS.401

Cause:
{
  "ResponseBody": {
    "kind": "Status",
    "apiVersion": "v1",
    "metadata": {}},
```



```
"status": "Failure",
"message": "Unauthorized",
"reason": "Unauthorized",
"code": 401
},
"StatusCode": 401,
"StatusText": "Unauthorized"
}
```

Call Amazon EMR with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

How the Optimized Amazon EMR integration is different than the Amazon EMR AWS SDK integration

The Optimized Amazon EMR service integration has a customized set of APIs that wrap the underlying Amazon EMR APIs, described below. Because of this, it differs significantly from the Amazon EMR AWS SDK service integration. In addition, the [Run a Job \(.sync\)](#) integration pattern is supported.

To integrate AWS Step Functions with Amazon EMR, you use the provided Amazon EMR service integration APIs. The service integration APIs are similar to the corresponding Amazon EMR APIs, with some differences in the fields that are passed and in the responses that are returned.

Step Functions does not terminate an Amazon EMR cluster automatically if execution is stopped. If your state machine stops before your Amazon EMR cluster has terminated, your cluster may continue running indefinitely, and can accrue additional charges. To avoid this, ensure that any Amazon EMR cluster you create is terminated properly. For more information, see:

- [Control Cluster Termination](#) in the Amazon EMR User Guide.
- The Service Integration Patterns [Run a Job \(.sync\)](#) section.

Note

As of `emr-5.28.0`, you can specify the parameter `StepConcurrencyLevel` when creating a cluster to allow multiple steps to run in parallel on a single cluster. You can use the Step Functions `Map` and `Parallel` states to submit work in parallel to the cluster.

The availability of Amazon EMR service integration is subject to the availability of Amazon EMR APIs. Please check the [Amazon EMR](#) documentation for limitations in special regions.

Note

For integration with Amazon EMR, Step Functions has a hard-coded 60 seconds job polling frequency for the first 10 minutes and 300 seconds after that.

The following table describes the differences between each service integration API and its corresponding Amazon EMR API.

Amazon EMR Service Integration APIs and Corresponding Amazon EMR APIs

Amazon EMR Service Integration API	Corresponding EMR API	Differences
<p><code>createCluster</code></p> <p>Creates and starts running a cluster (job flow).</p> <p>Amazon EMR is linked directly to a unique type of IAM role known as a service-linked role. For <code>createCluster</code> and <code>createCluster.sync</code> to work, you must have configured the necessary permissions to create the service-linked role <code>AWSServiceRoleForE</code></p>	<p>runJobFlow</p>	<p><code>createCluster</code> uses the same request syntax as runJobFlow, except for the following:</p> <ul style="list-style-type: none"> The field <code>Instances.KeepJobFlowAliveWhenNoSteps</code> is mandatory, and must have the Boolean value <code>TRUE</code>. The field <code>Steps</code> is not allowed. The field <code>Instances.InstanceFleets[in</code>

Amazon EMR Service Integration API	Corresponding EMR API	Differences
<p><code>MRCleanup</code> . For more information about this, including a statement you can add to your IAM permissions policy, see Using the Service-Linked Role for Amazon EMR.</p>		<p><code>dex].Name</code> should be provided and must be unique if the optional <code>modifyInstance FleetByName</code> connector API is used.</p> <ul style="list-style-type: none"> The field <code>Instances .InstanceGroups[index].Name</code> should be provided and must be unique if the optional <code>modifyInstanceGroupByName</code> API is used. <p>Response is this:</p> <pre>{ "ClusterId": "string" }</pre> <p>Amazon EMR uses this:</p> <pre>{ "JobFlowId": "string" }</pre>
<p><code>createCluster.sync</code></p> <p>Creates and starts running a cluster (job flow).</p>	<p>runJobFlow</p>	<p>The same as <code>createCluster</code> , but waits for the cluster to reach the <code>WAITING</code> state.</p>

Amazon EMR Service Integration API	Corresponding EMR API	Differences
<p><i>setClusterTerminationProtection</i></p> <p>Locks a cluster (job flow) so the EC2 instances in the cluster cannot be terminated by user intervention, an API call, or a job-flow error.</p>	<p><u>setTerminationProtection</u></p>	<p>Request uses this:</p> <pre data-bbox="1071 346 1507 506">{ "ClusterId": "string" }</pre> <p>Amazon EMR uses this:</p> <pre data-bbox="1071 611 1507 810">{ "JobFlowIds": ["string"] }</pre>
<p><i>terminateCluster</i></p> <p>Shuts down a cluster (job flow).</p>	<p><u>terminateJobFlows</u></p>	<p>Request uses this:</p> <pre data-bbox="1071 921 1507 1081">{ "ClusterId": "string" }</pre> <p>Amazon EMR uses this:</p> <pre data-bbox="1071 1186 1507 1386">{ "JobFlowIds": ["string"] }</pre>
<p><i>terminateCluster.sync</i></p> <p>Shuts down a cluster (job flow).</p>	<p><u>terminateJobFlows</u></p>	<p>The same as <code>terminateCluster</code>, but waits for the cluster to terminate.</p>

Amazon EMR Service Integration API	Corresponding EMR API	Differences
<p><i>addStep</i></p> <p>Adds a new step to a running cluster.</p> <p>Optionally, you can also specify the ExecutionRoleArn parameter while using this API.</p>	<p>addJobFlowSteps</p>	<p>Request uses the key "ClusterId" . Amazon EMR uses "JobFlowId" . Request uses a single step.</p> <pre data-bbox="1068 491 1507 688"> { "Step": <"StepConfig object"> } </pre> <p>Amazon EMR uses this:</p> <pre data-bbox="1068 793 1507 991"> { "Steps": [<StepConfig objects>] } </pre> <p>Response is this:</p> <pre data-bbox="1068 1096 1507 1255"> { "StepId": "string" } </pre> <p>Amazon EMR returns this:</p> <pre data-bbox="1068 1360 1507 1558"> { "StepIds": [<strings >] } </pre>

Amazon EMR Service Integration API	Corresponding EMR API	Differences
<p><i>addStep.sync</i></p> <p>Adds a new step to a running cluster.</p> <p>Optionally, you can also specify the ExecutionRoleArn parameter while using this API.</p>	<p>addJobFlowSteps</p>	<p>The same as <code>addStep</code>, but waits for the step to complete.</p>

Amazon EMR Service Integration API	Corresponding EMR API	Differences
<p><i>cancelStep</i></p> <p>Cancels a pending step in a running cluster.</p>	<p>cancelSteps</p>	<p>Request uses this:</p> <pre data-bbox="1073 348 1507 506">{ "StepId": "string" }</pre> <p>Amazon EMR uses this:</p> <pre data-bbox="1073 611 1507 810">{ "StepIds": [<strings >] }</pre> <p>Response is this:</p> <pre data-bbox="1073 915 1507 1150">{ "CancelStepsInfo": <CancelStepsInfo object> }</pre> <p>Amazon EMR uses this:</p> <pre data-bbox="1073 1255 1507 1493">{ "CancelStepsInfoLi st": [<CancelStepsInfo objects>] }</pre>

Amazon EMR Service Integration API	Corresponding EMR API	Differences
<p><i>modifyInstanceFleetByName</i></p> <p>Modifies the target On-Demand and target Spot capacities for the instance fleet with the specified <code>InstanceFleetName</code> .</p>	<p>modifyInstanceFleet</p>	<p>Request is the same as for <code>modifyInstanceFleet</code> , except for the following:</p> <ul style="list-style-type: none">• The field <code>InstanceFleetId</code> is not allowed.• At runtime the <code>InstanceFleetId</code> is determined automatically by the service integration by calling <code>ListInstanceFleets</code> and parsing the result.

Amazon EMR Service Integration API	Corresponding EMR API	Differences
<p><i>modifyInstanceGroupByName</i></p> <p>Modifies the number of nodes and configuration settings of an instance group.</p>	<p>modifyInstanceGroups</p>	<p>Request is this:</p> <pre data-bbox="1073 348 1507 663"> { "ClusterId": "string", "InstanceGroup": <InstanceGroupModifyConfig object> } </pre> <p>Amazon EMR uses a list:</p> <pre data-bbox="1073 768 1507 1083"> { "ClusterId": ["string"], "InstanceGroups": [<InstanceGroupModifyConfig objects>] } </pre> <p>Within the InstanceGroupModifyConfig object, the field InstanceGroupId is not allowed.</p> <p>A new field, InstanceGroupName, has been added. At runtime the InstanceGroupId is determined automatically by the service integration by calling ListInstanceGroups and parsing the result.</p>

The following includes a Task state that creates a cluster.

```
"Create_Cluster": {
  "Type": "Task",
  "Resource": "arn:aws:states:::elasticmapreduce:createCluster.sync",
  "Parameters": {
    "Name": "MyWorkflowCluster",
    "VisibleToAllUsers": true,
    "ReleaseLabel": "emr-5.28.0",
    "Applications": [
      {
        "Name": "Hive"
      }
    ],
    "ServiceRole": "EMR_DefaultRole",
    "JobFlowRole": "EMR_EC2_DefaultRole",
    "LogUri": "s3n://aws-logs-123456789012-us-east-1/elasticmapreduce/",
    "Instances": {
      "KeepJobFlowAliveWhenNoSteps": true,
      "InstanceFleets": [
        {
          "InstanceFleetType": "MASTER",
          "Name": "MASTER",
          "TargetOnDemandCapacity": 1,
          "InstanceTypeConfigs": [
            {
              "InstanceType": "m4.xlarge"
            }
          ]
        },
        {
          "InstanceFleetType": "CORE",
          "Name": "CORE",
          "TargetOnDemandCapacity": 1,
          "InstanceTypeConfigs": [
            {
              "InstanceType": "m4.xlarge"
            }
          ]
        }
      ]
    }
  },
  "End": true
}
```

The following includes a Task state that enables termination protection.

```
"Enable_Termination_Protection": {
  "Type": "Task",
  "Resource": "arn:aws:states:::elasticmapreduce:setClusterTerminationProtection",
  "Parameters": {
    "ClusterId.$": "$.ClusterId",
    "TerminationProtected": true
  },
  "End": true
}
```

The following includes a Task state that submits a step to a cluster.

```
"Step_One": {
  "Type": "Task",
  "Resource": "arn:aws:states:::elasticmapreduce:addStep.sync",
  "Parameters": {
    "ClusterId.$": "$.ClusterId",
    "ExecutionRoleArn": "arn:aws:iam::123456789012:role/myEMR-execution-role",
    "Step": {
      "Name": "The first step",
      "ActionOnFailure": "CONTINUE",
      "HadoopJarStep": {
        "Jar": "command-runner.jar",
        "Args": [
          "hive-script",
          "--run-hive-script",
          "--args",
          "-f",
          "s3://<region>.elasticmapreduce.samples/cloudfront/code/
Hive_CloudFront.q",
          "-d",
          "INPUT=s3://<region>.elasticmapreduce.samples",
          "-d",
          "OUTPUT=s3://<mybucket>/MyHiveQueryResults/"
        ]
      }
    }
  },
  "End": true
}
```

The following includes a Task state that cancels a step.

```
"Cancel_Step_One": {
  "Type": "Task",
  "Resource": "arn:aws:states:::elasticmapreduce:cancelStep",
  "Parameters": {
    "ClusterId.$": "$.ClusterId",
    "StepId.$": "$.AddStepsResult.StepId"
  },
  "End": true
}
```

The following includes a Task state that terminates a cluster.

```
"Terminate_Cluster": {
  "Type": "Task",
  "Resource": "arn:aws:states:::elasticmapreduce:terminateCluster.sync",
  "Parameters": {
    "ClusterId.$": "$.ClusterId"
  },
  "End": true
}
```

The following includes a Task state that scales a cluster up or down for an instance group.

```
"ModifyInstanceGroupByName": {
  "Type": "Task",
  "Resource": "arn:aws:states:::elasticmapreduce:modifyInstanceGroupByName",
  "Parameters": {
    "ClusterId": "j-1234567890123",
    "InstanceGroupName": "MyCoreGroup",
    "InstanceGroup": {
      "InstanceCount": 8
    }
  },
  "End": true
}
```

The following includes a Task state that scales a cluster up or down for an instance fleet.

```
"ModifyInstanceFleetByName": {
```

```
"Type": "Task",
"Resource": "arn:aws:states:::elasticmapreduce:modifyInstanceFleetByName",
"Parameters": {
  "ClusterId": "j-1234567890123",
  "InstanceFleetName": "MyCoreFleet",
  "InstanceFleet": {
    "TargetOnDemandCapacity": 8,
    "TargetSpotCapacity": 0
  }
},
"End": true
}
```

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Call Amazon EMR on EKS with AWS Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

How the Optimized Amazon EMR on EKS integration is different than the Amazon EMR on EKS AWS SDK integration

- The [Run a Job \(.sync\)](#) integration pattern is supported.
- There are no optimizations for the [Request Response](#) integration pattern.
- The [Wait for a Callback with the Task Token](#) integration pattern is not supported.


Note

For integration with Amazon EMR, Step Functions has a hard-coded 60 seconds job polling frequency for the first 10 minutes and 300 seconds after that.

To integrate AWS Step Functions with Amazon EMR on EKS, use the Amazon EMR on EKS service integration APIs. The service integration APIs are the same as the corresponding Amazon EMR on EKS APIs, but not all APIs support all integration patterns, as shown in the following table.

API	Request response	Run a job (.sync)
CreateVirtualCluster	✓	
DeleteVirtualCluster	✓	✓
StartJobRun	✓	✓

Supported Amazon EMR on EKS APIs:

 **Note**

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 256 KB of data as a UTF-8 encoded string when you send to, or receive data from, another service. See [Quotas related to state machine executions](#).

- [CreateVirtualCluster](#)
 - [Request syntax](#)
 - [Supported parameters](#)
 - [Response syntax](#)
- [DeleteVirtualCluster](#)
 - [Request syntax](#)
 - [Supported parameters](#)
 - [Response syntax](#)
- [StartJobRun](#)
 - [Request syntax](#)
 - [Supported parameters](#)
 - [Response syntax](#)

The following includes a Task state that creates a virtual cluster.

```
"Create_Virtual_Cluster": {
  "Type": "Task",
```

```

"Resource": "arn:aws:states:::emr-containers:createVirtualCluster",
"Parameters": {
  "Name": "MyVirtualCluster",
  "ContainerProvider": {
    "Id": "EKSClusterName",
    "Type": "EKS",
    "Info": {
      "EksInfo": {
        "Namespace": "Namespace"
      }
    }
  }
},
"End": true
}

```

The following includes a Task state that submits a job to a virtual cluster and waits for it to complete.

```

"Submit_Job": {
  "Type": "Task",
  "Resource": "arn:aws:states:::emr-containers:startJobRun.sync",
  "Parameters": {
    "Name": "MyJobName",
    "VirtualClusterId.$": "$.VirtualClusterId",
    "ExecutionRoleArn": "arn:aws:iam::<accountId>:role/job-execution-role",
    "ReleaseLabel": "emr-6.2.0-latest",
    "JobDriver": {
      "SparkSubmitJobDriver": {
        "EntryPoint": "s3://<mybucket>/jobs/trip-count.py",
        "EntryPointArguments": [
          "60"
        ],
        "SparkSubmitParameters": "--conf spark.driver.cores=2 --conf
spark.executor.instances=10 --conf spark.kubernetes.pyspark.pythonVersion=3 --conf
spark.executor.memory=10G --conf spark.driver.memory=10G --conf spark.executor.cores=1
--conf spark.dynamicAllocation.enabled=false"
      }
    },
    "ConfigurationOverrides": {
      "ApplicationConfiguration": [
        {
          "Classification": "spark-defaults",

```

```

        "Properties": {
            "spark.executor.instances": "2",
            "spark.executor.memory": "2G"
        }
    },
    ],
    "MonitoringConfiguration": {
        "PersistentAppUI": "ENABLED",
        "CloudWatchMonitoringConfiguration": {
            "LogGroupName": "MyLogGroupName",
            "LogStreamNamePrefix": "MyLogStreamNamePrefix"
        },
        "S3MonitoringConfiguration": {
            "LogUri": "s3://<mylogsbucket>"
        }
    },
    },
    "Tags": {
        "taskType": "jobName"
    },
    },
    "End": true
}

```

The following includes a Task state that deletes a virtual cluster and waits for the deletion to complete.

```

"Delete_Virtual_Cluster": {
    "Type": "Task",
    "Resource": "arn:aws:states:::emr-containers:deleteVirtualCluster.sync",
    "Parameters": {
        "Id.$": "$.VirtualClusterId"
    },
    "End": true
}

```

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Call Amazon EMR Serverless with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

i How the Optimized EMR Serverless integration is different than the EMR Serverless AWS SDK integration

- The Optimized EMR Serverless service integration has a customized set of [APIs](#) that wrap the underlying EMR Serverless APIs. Because of this customization, the optimized EMR Serverless integration differs significantly from the EMR Serverless AWS SDK service integration. In addition, the optimized EMR Serverless integration supports [Run a Job \(.sync\)](#) integration pattern.
- The [Wait for a Callback with the Task Token](#) integration pattern is not supported.

In this topic

- [EMR Serverless service integration APIs](#)
- [EMR Serverless integration use cases](#)

EMR Serverless service integration APIs

To integrate AWS Step Functions with EMR Serverless, you can use the following six EMR Serverless service integration APIs. These service integration APIs are similar to the corresponding EMR Serverless APIs, with some differences in the fields that are passed and in the responses that are returned.

The following table describes the differences between each service integration API and its corresponding EMR Serverless API.

EMR Serverless service integration APIs and corresponding EMR Serverless APIs

EMR Serverless service integration API	Corresponding EMR Serverless API	Differences
<i>createApplication</i> Creates an application.	CreateApplication	None

EMR Serverless service integration API	Corresponding EMR Serverless API	Differences
<p>EMR Serverless is linked to a unique type of IAM role known as a service-linked role. For <code>createApplication</code> and <code>createApplication.sync</code> to work, you must have configured the necessary permissions to create the service-linked role <code>AWSServiceRoleForAmazonEMRServerless</code>. For more information about this, including a statement you can add to your IAM permissions policy, see Using service-linked roles for EMR Serverless.</p>		
<p><code>createApplication.sync</code></p> <p>Creates an application.</p>	<p>CreateApplication</p>	<p>No differences between the requests and responses of the EMR Serverless API and EMR Serverless service integration API. However, <code>createApplication.sync</code> waits for the application to reach the CREATED state.</p>

EMR Serverless service integration API	Corresponding EMR Serverless API	Differences
<p><i>startApplication</i></p> <p>Starts a specified application and initializes the application's initial capacity if configured.</p>	<p>StartApplication</p>	<p>The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integration API response includes the following data.</p> <pre data-bbox="1068 583 1507 783">{ "ApplicationId": "string" }</pre>
<p><i>startApplication.sync</i></p> <p>Starts a specified application and initializes the initial capacity if configured.</p>	<p>StartApplication</p>	<p>The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integration API response includes the following data.</p> <pre data-bbox="1068 1136 1507 1335">{ "ApplicationId": "string" }</pre> <p>Also, <i>startApplication.sync</i> waits for the application to reach the STARTED state.</p>

EMR Serverless service integration API	Corresponding EMR Serverless API	Differences
<p><i>stopApplication</i></p> <p>Stops a specified application and releases initial capacity if configured. All scheduled and running jobs must be completed or cancelled before stopping an application.</p>	<p>StopApplication</p>	<p>The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integration API response includes the following data.</p> <pre data-bbox="1068 583 1507 783"> { "ApplicationId": "string" } </pre>
<p><i>stopApplication.sync</i></p> <p>Stops a specified application and releases initial capacity if configured. All scheduled and running jobs must be completed or cancelled before stopping an application.</p>	<p>StopApplication</p>	<p>The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integration API response includes the following data.</p> <pre data-bbox="1068 1136 1507 1335"> { "ApplicationId": "string" } </pre> <p>Also, <i>stopApplication.sync</i> waits for the application to reach the STOPPED state.</p>

EMR Serverless service integration API	Corresponding EMR Serverless API	Differences
<p><i>deleteApplication</i></p> <p>Deletes an application. An application must be in the STOPPED or CREATED state in order to be deleted.</p>	<p>DeleteApplication</p>	<p>The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integration API response includes the following data.</p> <pre data-bbox="1068 583 1507 783"> { "ApplicationId": "string" } </pre>
<p><i>deleteApplication.sync</i></p> <p>Deletes an application. An application must be in the STOPPED or CREATED state in order to be deleted.</p>	<p>DeleteApplication</p>	<p>The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integration API response includes the following data.</p> <pre data-bbox="1068 1136 1507 1335"> { "ApplicationId": "string" } </pre> <p>Also, <i>stopApplication.sync</i> waits for the application to reach the TERMINATED state.</p>
<p><i>startJobRun</i></p> <p>Starts a job run.</p>	<p>StartJobRun</p>	<p>None</p>

EMR Serverless service integration API	Corresponding EMR Serverless API	Differences
<p><i>startJobRun.sync</i></p> <p>Starts a job run.</p>	<p>StartJobRun</p>	<p>No differences between the requests and responses of the EMR Serverless API and EMR Serverless service integration API. However, <i>startJobRun.sync</i> waits for the application to reach the SUCCESS state.</p>
<p><i>cancelJobRun</i></p> <p>Cancels a job run.</p>	<p>CancelJobRun</p>	<p>None</p>
<p><i>cancelJobRun.sync</i></p> <p>Cancels a job run.</p>	<p>CancelJobRun</p>	<p>No differences between the requests and responses of the EMR Serverless API and EMR Serverless service integration API. However, <i>cancelJobRun.sync</i> waits for the application to reach the CANCELLED state.</p>

EMR Serverless integration use cases

For the Optimized EMR Serverless service integration, we recommend that you create a single application, and then use that application to run multiple jobs. For example, in a single state machine, you can include multiple [startJobRun](#) requests, all of which use the same application. The following [Task state](#) state examples show use cases to integrate EMR Serverless APIs with Step Functions. For information about other use cases of EMR Serverless, see [What is Amazon EMR Serverless](#).

Tip

To deploy an example of a state machine that integrates with EMR Serverless for running multiple jobs to your AWS account, see [Run an EMR Serverless job](#).

- [Create an application](#)
- [Start an application](#)
- [Stop an application](#)
- [Delete an application](#)
- [Start a job in an application](#)
- [Cancel a job in an application](#)

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

In the examples shown in the following use cases, replace the *italicized* text with your resource-specific information. For example, replace *yourApplicationId* with the ID of your EMR Serverless application, such as 00yv7iv71inak893.

Create an application

The following Task state example creates an application using the `createApplication.sync` service integration API.

```
"Create_Application": {
  "Type": "Task",
  "Resource": "arn:aws:states:::emr-serverless:createApplication.sync",
  "Parameters": {
    "Name": "MyApplication",
    "ReleaseLabel": "emr-6.9.0",
    "Type": "SPARK"
  },
  "End": true
}
```

Start an application

The following Task state example starts an application using the *startApplication.sync* service integration API.

```
"Start_Application": {
  "Type": "Task",
  "Resource": "arn:aws:states:::emr-serverless:startApplication.sync",
  "Parameters": {
    "ApplicationId": "yourApplicationId"
  },
  "End": true
}
```

Stop an application

The following Task state example stops an application using the *stopApplication.sync* service integration API.

```
"Stop_Application": {
  "Type": "Task",
  "Resource": "arn:aws:states:::emr-serverless:stopApplication.sync",
  "Parameters": {
    "ApplicationId": "yourApplicationId"
  },
  "End": true
}
```

Delete an application

The following Task state example deletes an application using the *deleteApplication.sync* service integration API.

```
"Delete_Application": {
  "Type": "Task",
  "Resource": "arn:aws:states:::emr-serverless:deleteApplication.sync",
  "Parameters": {
    "ApplicationId": "yourApplicationId"
  },
  "End": true
}
```


Start a job in an application

The following Task state example starts a job in an application using the *startJobRun.sync* service integration API.

```
"Start_Job": {
  "Type": "Task",
  "Resource": "arn:aws:states:::emr-serverless:startJobRun.sync",
  "Parameters": {
    "ApplicationId": "yourApplicationId",
    "ExecutionRoleArn": "arn:aws:iam::123456789012:role/myEMRServerless-execution-role",
    "JobDriver": {
      "SparkSubmit": {
        "EntryPoint": "s3://<mybucket>/sample.py",
        "EntryPointArguments": ["1"],
        "SparkSubmitParameters": "--conf spark.executor.cores=4 --conf
spark.executor.memory=4g --conf spark.driver.cores=2 --conf spark.driver.memory=4g --
conf spark.executor.instances=1"
      }
    }
  },
  "End": true
}
```

Cancel a job in an application

The following Task state example cancels a job in an application using the *cancelJobRun.sync* service integration API.

```
"Cancel_Job": {
  "Type": "Task",
  "Resource": "arn:aws:states:::emr-serverless:cancelJobRun.sync",
  "Parameters": {
    "ApplicationId.$": "$.ApplicationId",
    "JobRunId.$": "$.JobRunId"
  },
  "End": true
}
```

Call EventBridge with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

How the Optimized EventBridge integration is different than the EventBridge AWS SDK integration

- The execution ARN and the state machine ARN are automatically appended to the `Resources` field of each `PutEventsRequestEntry`.
- If the response from `PutEvents` contains a non-zero `FailedEntryCount` then the Task state fails with the error `EventBridge.FailedEntry`.

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Step Functions provides a service integration API for integrating with Amazon EventBridge. This lets you build event-driven applications by sending custom events directly from Step Functions workflows.

To use the `PutEvents` API, you will need to create an EventBridge rule in your account that matches the specific pattern of the events you will send. For example, you could:

- Create a Lambda function in your account that receives and prints an event that matches an EventBridge rule.
- Create an EventBridge rule in your account on the default event bus that matches a specific event pattern and targets the Lambda function.

For more information, see:

- [Adding Amazon EventBridge events with PutEvents](#) in the EventBridge User Guide.
- [Wait for a Callback with the Task Token](#) in Service Integration Patterns.

Note

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 256 KB of data as a UTF-8 encoded string when you send to, or receive data from, another service. See [Quotas related to state machine executions](#).

Supported EventBridge API

Supported EventBridge API and syntax include:

- [PutEvents](#)
 - [Request syntax](#)
 - Supported parameter:
 - [Entries](#)
 - [Response syntax](#)

The following includes a Task that sends a custom event:

```
{
  "Type": "Task",
  "Resource": "arn:aws:states:::events:putEvents",
  "Parameters": {
    "Entries": [
      {
        "Detail": {
          "Message": "MyMessage"
        },
        "DetailType": "MyDetailType",
        "EventBusName": "MyEventBus",
        "Source": "my.source"
      }
    ]
  },
  "End": true
}
```

Error handling

The `PutEvents` API accepts an array of entries as input, then returns an array of result entries. As long as the `PutEvents` action was successful, `PutEvents` will return an HTTP 200 response, even if one or more entries failed. `PutEvents` returns the number of failed entries in the `FailedEntryCount` field.

Step Functions checks whether the `FailedEntryCount` is greater than zero. If it is greater than zero, Step Functions fails the state with the error `EventBridge.FailedEntry`. This lets you use the built-in error handling of Step Functions on task states to catch or retry when there are failed entries, rather than needing to use an additional state to analyze the `FailedEntryCount` from the response.

Note

If you have implemented idempotency and can safely retry on all entries, you can use Step Functions' retry logic. Step Functions does not remove successful entries from the `PutEvents` input array before retrying. Instead, it retries with the original array of entries.

Manage AWS Glue Jobs with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

How the Optimized AWS Glue integration is different than the AWS GlueAWS SDK integration

- The [Run a Job \(.sync\)](#) integration pattern is available.
- The `JobName` field is extracted from the request and inserted into the response, which normally only contains `JobRunID`.

Supported AWS Glue API:

- [StartJobRun](#)

Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

The following includes a Task state that starts an AWS Glue job.

```
"Glue StartJobRun": {
  "Type": "Task",
  "Resource": "arn:aws:states:::glue:startJobRun.sync",
  "Parameters": {
    "JobName": "GlueJob-JTrR05198qMG"
  },
  "Next": "ValidateOutput"
},
```

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Manage AWS Glue DataBrew Jobs with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

You can use the DataBrew integration to add data cleaning and data normalization steps into your analytics and machine learning workflows.

Supported DataBrew API:

- [StartJobRun](#)

The following includes a Task state that starts a request-response DataBrew job.

```
"DataBrew StartJobRun": {
  "Type": "Task",
  "Resource": "arn:aws:states:::databrew:startJobRun",
  "Parameters": {
    "Name": "sample-proj-job-1"
  }
},
```

```
    },  
    "Next": "NEXT_STATE"  
  },
```

The following includes a Task state that starts a sync DataBrew job.

```
"DataBrew StartJobRun": {  
  "Type": "Task",  
  "Resource": "arn:aws:states:::databrew:startJobRun.sync",  
  "Parameters": {  
    "Name": "sample-proj-job-1"  
  },  
  "Next": "NEXT_STATE"  
},
```

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Invoke Lambda with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

How the Optimized Lambda integration is different than the Lambda AWS SDK integration

- The `Payload` field of the response is parsed from escaped Json to Json.
- If the response contains the field `FunctionError` or an exception is raised within the Lambda function, the task fails.

For more information about managing state input, output, and results, see [Input and Output Processing in Step Functions](#).

Supported AWS Lambda APIs:

- [Invoke](#)

- [Request Syntax](#)
- Supported Parameters
 - [ClientContext](#)
 - [FunctionName](#)
 - [InvocationType](#)
 - [Qualifier](#)
 - [Payload](#)
- [Response syntax](#)

ⓘ Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

The following includes a Task state that invokes a Lambda function.

```
{
  "StartAt":"CallLambda",
  "States":{
    "CallLambda":{
      "Type":"Task",
      "Resource":"arn:aws:states:::lambda:invoke",
      "Parameters":{
        "FunctionName":"arn:aws:lambda:us-east-1:123456789012:function:MyFunction"
      },
      "End":true
    }
  }
}
```

The following includes a Task state that implements the [callback](#) service integration pattern.

```
{
  "StartAt":"GetManualReview",
  "States":{
    "GetManualReview":{
```

```
    "Type": "Task",
    "Resource": "arn:aws:states:::lambda:invoke.waitForTaskToken",
    "Parameters": {
      "FunctionName": "arn:aws:lambda:us-east-1:123456789012:function:get-model-  
review-decision",
      "Payload": {
        "model.$": "$.new_model",
        "token.$": "$$.Task.Token"
      },
      "Qualifier": "prod-v1"
    },
    "End": true
  }
}
```

When you invoke a Lambda function, the execution will wait for the function to complete. If you invoke the Lambda function with a callback task, the heartbeat timeout doesn't start counting until after the Lambda function has completed executing and returned a result. As long as the Lambda function executes, the heartbeat timeout is not enforced.

It is also possible to call Lambda asynchronously using the `InvocationType` parameter, as seen in the following example:

Note

For asynchronous invocations of Lambda functions, the heartbeat timeout period starts immediately.

```
{
  "Comment": "A Hello World example of the Amazon States Language using Pass states",
  "StartAt": "Hello",
  "States": {
    "Hello": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Parameters": {
        "FunctionName": "arn:aws:lambda:us-east-1:123456789012:function:echo",
        "InvocationType": "Event"
      }
    }
  }
}
```



```

    },
    "End": true
  }
}
}

```

When the Task result is returned, the function output is nested inside a dictionary of metadata. For example:

```

{
  "ExecutedVersion": "$LATEST",
  "Payload": "FUNCTION OUTPUT",
  "SdkHttpMetadata": {
    "HttpHeaders": {
      "Connection": "keep-alive",
      "Content-Length": "4",
      "Content-Type": "application/json",
      "Date": "Fri, 26 Mar 2021 07:42:02 GMT",
      "X-Amz-Executed-Version": "$LATEST",
      "x-amzn-Remapped-Content-Length": "0",
      "x-amzn-RequestId": "0101aa0101-1111-111a-aa55-1010aaa1010",
      "X-Amzn-Trace-Id": "root=1-1a1a000a2a2-fe0101aa10ab;sampld=0"
    },
    "HttpStatusCode": 200
  },
  "SdkResponseMetadata": {
    "RequestId": "6b3bebdb-9251-453a-ae45-512d9e2bf4d3"
  },
  "StatusCode": 200
}

```

Alternatively, you can invoke a Lambda function by specifying a function ARN directly in the "Resource" field. When you invoke a Lambda function in this way, you can't specify `.waitForTaskToken`, and the task result contains only the function output.

```

{
  "StartAt": "CallFunction",
  "States": {
    "CallFunction": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloFunction",

```

```
        "End": true
      }
    }
  }
```

You can invoke a specific Lambda function version or alias by specifying those options in the ARN in the Resource field. See the following in the Lambda documentation:

- [AWS Lambda versioning](#)
- [AWS Lambda aliases](#)

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Manage AWS Elemental MediaConvert with Step Functions

Experiment with Step Functions and MediaConvert

Learn how to use the MediaConvert optimized integration in a workflow that detects and removes SMPTE color bars of unknown length from the beginning of a video clip. Read the blog post from Apr, 12, 2024: [Low code workflows with AWS Elemental MediaConvert](#)

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

How the optimized integration is different than standard AWS SDK integration

- The [Run a Job \(.sync\)](#) integration pattern is available.
- No optimizations for [Request Response](#) or [Wait for a Callback with the Task Token](#) integration patterns.

Supported MediaConvert APIs:

- [CreateJob](#)
- [Request syntax](#)

- Supported parameters:
 - [Role](#) (Required)
 - [Settings](#) (Required)
 - [CreateJobRequest](#) (Optional)
- [Response syntax](#) – see **CreateJobResponse** schema

The following includes a Task state that submits an MediaConvert job and waits for it to complete.

```
{
  "StartAt": "MediaConvert_CreateJob",
  "States": {
    "MediaConvert_CreateJob": {
      "Type": "Task",
      "Resource": "arn:aws:states:::mediaconvert:createJob.sync",
      "Parameters": {
        "Role": "arn:aws:iam::111122223333:role/Admin",
        "Settings": {
          "OutputGroups": [
            {
              "Outputs": [
                {
                  "ContainerSettings": {
                    "Container": "MP4"
                  },
                  "VideoDescription": {
                    "CodecSettings": {
                      "Codec": "H_264",
                      "H264Settings": {
                        "MaxBitrate": 1000,
                        "RateControlMode": "QVBR",
                        "SceneChangeDetect": "TRANSITION_DETECTION"
                      }
                    }
                  }
                }
              ],
              "AudioDescriptions": [
                {
                  "CodecSettings": {
                    "Codec": "AAC",
                    "AacSettings": {
                      "Bitrate": 96000,
                      "CodingMode": "CODING_MODE_2_0",

```

```

        "SampleRate": 48000
      }
    }
  ],
  "OutputGroupSettings": {
    "Type": "FILE_GROUP_SETTINGS",
    "FileGroupSettings": {
      "Destination": "s3://DOC-EXAMPLE-DESTINATION-BUCKET/"
    }
  }
},
"Inputs": [
  {
    "AudioSelectors": {
      "Audio Selector 1": {
        "DefaultSelection": "DEFAULT"
      }
    },
    "FileInput": "s3://DOC-EXAMPLE-SOURCE-BUCKET/DOC-EXAMPLE-SOURCE_FILE"
  }
]
},
"End": true
}
}

```

For information about how to configure IAM permissions when using Step Functions with MediaConvert, see [IAM policies for AWS Elemental MediaConvert](#).

❗ Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

Manage SageMaker with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

ⓘ How the Optimized SageMaker integration is different than the SageMaker AWS SDK integration

- The [Run a Job \(.sync\)](#) integration pattern is supported.
- There are no optimizations for the [Request Response](#) integration pattern.
- The [Wait for a Callback with the Task Token](#) integration pattern is not supported.

Supported SageMaker APIs and syntax:

- [CreateEndpoint](#)
 - [Request syntax](#)
 - Supported parameters:
 - [EndpointConfigName](#)
 - [EndpointName](#)
 - [Tags](#)
 - [Response syntax](#)
- [CreateEndpointConfig](#)
 - [Request syntax](#)
 - Supported parameters:
 - [EndpointConfigName](#)
 - [KmsKeyId](#)
 - [ProductionVariants](#)
 - [Tags](#)
 - [Response syntax](#)
- [CreateHyperParameterTuningJob](#)

Note

This API action supports the [.sync](#) integration pattern.

- [Request syntax](#)
- Supported parameters:
 - [HyperParameterTuningJobConfig](#)
 - [HyperParameterTuningJobName](#)
 - [Tags](#)
 - [TrainingJobDefinition](#)
 - [WarmStartConfig](#)
- [Response syntax](#)
- [CreateLabelingJob](#)

Note

This API action supports the [.sync](#) integration pattern.

- [Request syntax](#)
- Supported parameters:
 - [HumanTaskConfig](#)
 - [InputConfig](#)
 - [LabelAttributeName](#)
 - [LabelCategoryConfigS3Uri](#)
 - [LabelingJobAlgorithmsConfig](#)
 - [LabelingJobName](#)
 - [OutputConfig](#)
 - [RoleArn](#)
 - [StoppingConditions](#)

- [Response syntax](#)
- [CreateModel](#)
 - [Request syntax](#)
 - Supported parameters:
 - [Containers](#)
 - [EnableNetworkIsolation](#)
 - [ExecutionRoleArn](#)
 - [ModelName](#)
 - [PrimaryContainer](#)
 - [Tags](#)
 - [VpcConfig](#)
- [CreateProcessingJob](#)

Note

This API action supports the [.sync](#) integration pattern.

- [Request syntax](#)
- Supported parameters:
 - [AppSpecification](#)
 - [Environment](#)
 - [ExperimentConfig](#)
 - [NetworkConfig](#)
 - [ProcessingInputs](#)
 - [ProcessingJobName](#)
 - [ProcessingOutputConfig](#)
 - [ProcessingResources](#)
 - [RoleArn](#)
 - [StoppingCondition](#)
 - [Tags](#)
- [Response syntax](#)

- [CreateTrainingJob](#)

Note

This API action supports the [.sync](#) integration pattern.

- [Request syntax](#)
- Supported parameters:
 - [AlgorithmSpecification](#)
 - [HyperParameters](#)
 - [InputDataConfig](#)
 - [OutputDataConfig](#)
 - [ResourceConfig](#)
 - [RoleArn](#)
 - [StoppingCondition](#)
 - [Tags](#)
 - [TrainingJobName](#)
 - [VpcConfig](#)
- [Response syntax](#)
- [CreateTransformJob](#)

Note

This API action supports the [.sync](#) integration pattern.

Note

AWS Step Functions will not automatically create a policy for `CreateTransformJob`. You must attach an inline policy to the created role. For more information, see this example IAM policy: [CreateTrainingJob](#).

- [Request syntax](#)
- Supported parameters:
 - [BatchStrategy](#)
 - [Environment](#)
 - [MaxConcurrentTransforms](#)
 - [MaxPayloadInMB](#)
 - [ModelName](#)
 - [Tags](#)
 - [TransformInput](#)
 - [TransformJobName](#)
 - [TransformOutput](#)
 - [TransformResources](#)
- [Response syntax](#)
- [UpdateEndpoint](#)
 - [Request syntax](#)
 - Supported parameters:
 - [EndpointConfigName](#)
 - [EndpointName](#)
 - [Response syntax](#)

SageMaker Transform Job Example

The following includes a Task state that creates an Amazon SageMaker transform job, specifying the Amazon S3 location for DataSource and TransformOutput.

```
{
  "SageMaker CreateTransformJob": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sagemaker:createTransformJob.sync",
    "Parameters": {
      "ModelName": "SageMakerCreateTransformJobModel-9iFBKsYti9vr",
      "TransformInput": {
        "CompressionType": "None",
```

```

    "ContentType": "text/csv",
    "DataSource": {
      "S3DataSource": {
        "S3DataType": "S3Prefix",
        "S3Uri": "s3://my-s3bucket-example-1/TransformJobDataInput.txt"
      }
    },
    "TransformOutput": {
      "S3OutputPath": "s3://my-s3bucket-example-1/TransformJobOutputPath"
    },
    "TransformResources": {
      "InstanceCount": 1,
      "InstanceType": "ml.m4.xlarge"
    },
    "TransformJobName": "sfn-binary-classification-prediction"
  },
  "Next": "ValidateOutput"
},

```

SageMaker Training Job Example

The following includes a Task state that creates an Amazon SageMaker training job.

```

{
  "SageMaker CreateTrainingJob":{
    "Type":"Task",
    "Resource":"arn:aws:states:::sagemaker:createTrainingJob.sync",
    "Parameters":{
      "TrainingJobName":"search-model",
      "ResourceConfig":{
        "InstanceCount":4,
        "InstanceType":"ml.c4.8xlarge",
        "VolumeSizeInGB":20
      },
      "HyperParameters":{
        "mode":"batch_skipgram",
        "epochs":"5",
        "min_count":"5",
        "sampling_threshold":"0.0001",
        "learning_rate":"0.025",
        "window_size":"5",
        "vector_dim":"300",

```

```

        "negative_samples": "5",
        "batch_size": "11"
    },
    "AlgorithmSpecification": {
        "TrainingImage": "...",
        "TrainingInputMode": "File"
    },
    "OutputDataConfig": {
        "S3OutputPath": "s3://bucket-name/doc-search/model"
    },
    "StoppingCondition": {
        "MaxRuntimeInSeconds": 100000
    },
    "RoleArn": "arn:aws:iam::123456789012:role/docsearch-stepfunction-iam-role",
    "InputDataConfig": [
        {
            "ChannelName": "train",
            "DataSource": {
                "S3DataSource": {
                    "S3DataType": "S3Prefix",
                    "S3Uri": "s3://bucket-name/doc-search/interim-data/training-data/",
                    "S3DataDistributionType": "FullyReplicated"
                }
            }
        }
    ],
    "Retry": [
        {
            "ErrorEquals": [
                "SageMaker.AmazonSageMakerException"
            ],
            "IntervalSeconds": 1,
            "MaxAttempts": 100,
            "BackoffRate": 1.1
        },
        {
            "ErrorEquals": [
                "SageMaker.ResourceLimitExceededException"
            ],
            "IntervalSeconds": 60,
            "MaxAttempts": 5000,
            "BackoffRate": 1
        }
    ],

```

```

    {
      "ErrorEquals": [
        "States.Timeout"
      ],
      "IntervalSeconds": 1,
      "MaxAttempts": 5,
      "BackoffRate": 1
    }
  ],
  "Catch": [
    {
      "ErrorEquals": [
        "States.ALL"
      ],
      "ResultPath": "$.cause",
      "Next": "Sagemaker Training Job Error"
    }
  ],
  "Next": "Delete Interim Data Job"
}
}

```

SageMaker Labeling Job Example

The following includes a Task state that creates an Amazon SageMaker labeling job.

```

{
  "StartAt": "SageMaker CreateLabelingJob",
  "TimeoutSeconds": 3600,
  "States": {
    "SageMaker CreateLabelingJob": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sagemaker:createLabelingJob.sync",
      "Parameters": {
        "HumanTaskConfig": {
          "AnnotationConsolidationConfig": {
            "AnnotationConsolidationLambdaArn": "arn:aws:lambda:us-
west-2:123456789012:function:ACS-TextMultiClass"
          },
          "NumberOfHumanWorkersPerDataObject": 1,

```

```
    "PreHumanTaskLambdaArn": "arn:aws:lambda:us-west-2:123456789012:function:PRE-
TextMultiClass",
    "TaskDescription": "Classify the following text",
    "TaskKeywords": [
        "tc",
        "Labeling"
    ],
    "TaskTimeLimitInSeconds": 300,
    "TaskTitle": "Classify short bits of text",
    "UiConfig": {
        "UiTemplateS3Uri": "s3://s3bucket-example/TextClassification.template"
    },
    "WorkteamArn": "arn:aws:sagemaker:us-west-2:123456789012:workteam/private-
crowd/ExampleTesting"
},
"InputConfig": {
    "DataAttributes": {
        "ContentClassifiers": [
            "FreeOfPersonallyIdentifiableInformation",
            "FreeOfAdultContent"
        ]
    },
},
"DataSource": {
    "S3DataSource": {
        "ManifestS3Uri": "s3://s3bucket-example/manifest.json"
    }
}
},
"LabelAttributeName": "Categories",
"LabelCategoryConfigS3Uri": "s3://s3bucket-example/labelcategories.json",
"LabelingJobName": "example-job-name",
"OutputConfig": {
    "S3OutputPath": "s3://s3bucket-example/output"
},
"RoleArn": "arn:aws:iam::123456789012:role/service-role/AmazonSageMaker-
ExecutionRole",
"StoppingConditions": {
    "MaxHumanLabeledObjectCount": 10000,
    "MaxPercentageOfInputDatasetLabeled": 100
}
},
"Next": "ValidateOutput"
},
"ValidateOutput": {
```

```

    "Type": "Choice",
    "Choices": [
      {
        "Not": {
          "Variable": "$.LabelingJobArn",
          "StringEquals": ""
        },
        "Next": "Succeed"
      }
    ],
    "Default": "Fail"
  },
  "Succeed": {
    "Type": "Succeed"
  },
  "Fail": {
    "Type": "Fail",
    "Error": "InvalidOutput",
    "Cause": "Output is not what was expected. This could be due to a service outage
or a misconfigured service integration."
  }
}

```

SageMaker Processing Job Example

The following includes a Task state that creates an Amazon SageMaker processing job.

```

{
  "StartAt": "SageMaker CreateProcessingJob Sync",
  "TimeoutSeconds": 3600,
  "States": {
    "SageMaker CreateProcessingJob Sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sagemaker:createProcessingJob.sync",
      "Parameters": {
        "AppSpecification": {
          "ImageUri": "737474898029.dkr.ecr.sa-east-1.amazonaws.com/sagemaker-scikit-learn:0.20.0-cpu-py3"
        },
        "ProcessingResources": {
          "ClusterConfig": {
            "InstanceCount": 1,

```

```

        "InstanceType": "m1.t3.medium",
        "VolumeSizeInGB": 10
    }
},
    "RoleArn": "arn:aws:iam::123456789012:role/SM-003-
CreateProcessingJobAPIExecutionRole",
    "ProcessingJobName.$": "$.id"
},
    "Next": "ValidateOutput"
},
"ValidateOutput": {
    "Type": "Choice",
    "Choices": [
        {
            "Not": {
                "Variable": "$.ProcessingJobArn",
                "StringEquals": ""
            },
            "Next": "Succeed"
        }
    ],
    "Default": "Fail"
},
"Succeed": {
    "Type": "Succeed"
},
"Fail": {
    "Type": "Fail",
    "Error": "InvalidConnectorOutput",
    "Cause": "Connector output is not what was expected. This could be due to a
service outage or a misconfigured connector."
}
}
}

```

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Call Amazon SNS with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

i How the Optimized Amazon SNS integration is different than the Amazon SNS AWS SDK integration

There are no optimizations for the [Request Response](#) or [Wait for a Callback with the Task Token](#) integration patterns.

Supported Amazon SNS APIs:

i Note

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 256 KB of data as a UTF-8 encoded string when you send to, or receive data from, another service. See [Quotas related to state machine executions](#).

- [Publish](#)
 - [Request syntax](#)
 - Supported Parameters
 - [Message](#)
 - [MessageAttributes](#)
 - [MessageStructure](#)
 - [PhoneNumber](#)
 - [Subject](#)
 - [TargetArn](#)
 - [TopicArn](#)
 - [Response syntax](#)

i Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

The following includes a Task state that publishes to an Amazon Simple Notification Service (Amazon SNS) topic.

```
{
  "StartAt": "Publish to SNS",
  "States": {
    "Publish to SNS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "TopicArn": "arn:aws:sns:us-east-1:123456789012:myTopic",
        "Message.$": "$.input.message",
        "MessageAttributes": {
          "my_attribute_no_1": {
            "DataType": "String",
            "StringValue": "value of my_attribute_no_1"
          },
          "my_attribute_no_2": {
            "DataType": "String",
            "StringValue": "value of my_attribute_no_2"
          }
        }
      },
      "End": true
    }
  }
}
```

Passing dynamic values. You can modify the above example to dynamically pass an attribute from this JSON payload:

```
{
  "input": {
    "message": "Hello world"
  },
  "SNSDetails": {
    "attribute1": "some value",
    "attribute2": "some other value",
  }
}
```

Append the `.$` to the `StringValue` field:

```
"MessageAttributes": {
  "my_attribute_no_1": {
    "DataType": "String",
    "StringValue.$": "$.SNSDetails.attribute1"
  },
  "my_attribute_no_2": {
    "DataType": "String",
    "StringValue.$": "$.SNSDetails.attribute2"
  }
}
```

The following includes a Task state that publishes to an Amazon SNS topic, and then waits for the task token to be returned. See [Wait for a Callback with the Task Token](#).

```
{
  "StartAt": "Send message to SNS",
  "States": {
    "Send message to SNS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish.waitForTaskToken",
      "Parameters": {
        "TopicArn": "arn:aws:sns:us-east-1:123456789012:myTopic",
        "Message": {
          "Input.$": "$",
          "TaskToken.$": "$$.Task.Token"
        }
      },
      "End": true
    }
  }
}
```

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Call Amazon SQS with Step Functions

Step Functions can control certain AWS services directly from [Amazon States Language](#) (ASL). To learn more, see [Working with other services](#) and [Pass parameters to a service API](#).

i How the Optimized Amazon SQS integration is different than the Amazon SQS AWS SDK integration

There are no optimizations for the [Request Response](#) or [Wait for a Callback with the Task Token](#) integration patterns.

Supported Amazon SQS APIs:

i Note

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 256 KB of data as a UTF-8 encoded string when you send to, or receive data from, another service. See [Quotas related to state machine executions](#).

- [SendMessage](#)

Supported parameters:

- [DelaySeconds](#)
- [MessageAttribute](#)
- [MessageBody](#)
- [MessageDeduplicationId](#)
- [MessageGroupId](#)
- [QueueUrl](#)
- [Response syntax](#)

i Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

The following includes a Task state that sends an Amazon Simple Queue Service (Amazon SQS) message.

```

{
  "StartAt": "Send to SQS",
  "States": {
    "Send to SQS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage",
      "Parameters": {
        "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/myQueue",
        "MessageBody.$": "$.input.message",
        "MessageAttributes": {
          "my_attribute_no_1": {
            "DataType": "String",
            "StringValue": "attribute1"
          },
          "my_attribute_no_2": {
            "DataType": "String",
            "StringValue": "attribute2"
          }
        }
      }
    },
    "End": true
  }
}

```

The following includes a Task state that publishes to an Amazon SQS queue, and then waits for the task token to be returned. See [Wait for a Callback with the Task Token](#).

```

{
  "StartAt": "Send message to SQS",
  "States": {
    "Send message to SQS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
      "Parameters": {
        "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/myQueue",
        "MessageBody": {
          "Input.$": "$",
          "TaskToken.$": "$$.Task.Token"
        }
      }
    },
    "End": true
  }
}

```

```
}  
}
```

To learn more about receiving messages in Amazon SQS, see [Receive and Delete Your Message](#) in the *Amazon Simple Queue Service Developer Guide*.

For information about how to configure IAM permissions when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Manage AWS Step Functions Executions as an Integrated Service

Step Functions integrates with its own API as a service integration. This allows Step Functions to start a new execution of a state machine directly from the task state of a running execution. When building new workflows, use [nested workflow executions](#) to reduce the complexity of your main workflows and to reuse common processes.

How the Optimized Step Functions integration is different than the Step Functions AWS SDK integration

- The [Run a Job \(.sync\)](#) integration pattern is available.

Note that there are no optimizations for the [Request Response](#) or [Wait for a Callback with the Task Token](#) integration patterns.

For more information, see the following:

- [Start Executions from a Task](#)
- [Working with other services](#)
- [Pass parameters to a service API](#)

Supported Step Functions APIs and syntax:

- [StartExecution](#)
 - [Request Syntax](#)
 - Supported Parameters
 - [Input](#)

- [Name](#)
- [StateMachineArn](#)
- [Response syntax](#)

The following includes a Task state that starts an execution of another state machine and waits for it to complete.

```
{
  "Type": "Task",
  "Resource": "arn:aws:states:::states:startExecution.sync:2",
  "Parameters": {
    "Input": {
      "Comment": "Hello world!"
    },
    "StateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
    "Name": "ExecutionName"
  },
  "End": true
}
```

The following includes a Task state that starts an execution of another state machine.

```
{
  "Type": "Task",
  "Resource": "arn:aws:states:::states:startExecution",
  "Parameters": {
    "Input": {
      "Comment": "Hello world!"
    },
    "StateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
    "Name": "ExecutionName"
  },
  "End": true
}
```

The following includes a Task state that implements the [callback](#) service integration pattern.

```
{
```

```

    "Type": "Task",
    "Resource": "arn:aws:states:::states:startExecution.waitForTaskToken",
    "Parameters": {
      "Input": {
        "Comment": "Hello world!",
        "token.$": "$$.Task.Token"
      },
      "StateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
      "Name": "ExecutionName"
    },
    "End": true
  }
}

```

To associate a nested workflow execution with the parent execution that started it, pass a specially named parameter that includes the execution ID pulled from the [context object](#). When starting a nested execution, use a parameter named `AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID`. Pass the execution ID by appending `.$` to the parameter name, and referencing the ID in the context object with `$$.Execution.Id`. For more information, see [Accessing the Context Object](#).

```

{
  "Type": "Task",
  "Resource": "arn:aws:states:::states:startExecution.sync",
  "Parameters": {
    "Input": {
      "Comment": "Hello world!",
      "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
    },
    "StateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
    "Name": "ExecutionName"
  },
  "End": true
}

```

Nested state machines return the following:

Resource	Output
<code>startExecution.sync</code>	String

Resource	Output
startExecution.sync:2	JSON

Both will wait for the nested state machine to complete, but they return different Output formats. For example, if you create a Lambda function that returns the object { "MyKey": "MyValue" }, you would get the following responses:

For startExecution.sync:

```
{
  <other fields>
  "Output": "{ \"MyKey\": \"MyValue\" }"
}
```

For startExecution.sync:2:

```
{
  <other fields>
  "Output": {
    "MyKey": "MyValue"
  }
}
```

Configuring IAM permissions for nested state machines

A parent state machine determines if a child state machine has completed execution using polling and events. Polling requires permission for `states:DescribeExecution` while events sent through EventBridge to Step Functions require permissions for `events:PutTargets`, `events:PutRule`, and `events:DescribeRule`. If these permissions are missing from your IAM role, there may be a delay before a parent state machine becomes aware of the completion of the child state machine's execution.

For a state machine that calls `StartExecution` for a single nested workflow execution, use an IAM policy that limits permissions to that state machine.

For more information, see [IAM permissions for Step Functions](#).

Call third-party APIs

An HTTP Task is a type of [Task state](#) that lets you call any public, third-party API, such as Salesforce and Stripe, in your workflows. To call a third-party API, use the [Task](#) state with the `arn:aws:states:::http:invoke` resource. Then, provide the API endpoint configuration details, such as the API URL, method you want to use, and [authentication](#) details.

If you use [Workflow Studio](#) to build your state machine that contains an HTTP Task, Workflow Studio automatically generates an execution role with IAM policies for the HTTP Task. For more information, see [Role for testing HTTP Tasks in Workflow Studio](#).

Topics

- [HTTP Task definition](#)
- [HTTP Task fields](#)
- [Authentication for an HTTP Task](#)
- [Merging EventBridge connection and HTTP Task definition data](#)
- [Applying URL-encoding on request body](#)
- [IAM permissions to run an HTTP Task](#)
- [HTTP Task example](#)
- [Testing an HTTP Task](#)
- [Unsupported HTTP Task responses](#)

HTTP Task definition

The [ASL definition](#) represents an HTTP Task with `http:invoke` resource. The following HTTP Task definition invokes a Stripe API that returns a list of all customers.

```
"Call third-party API": {
  "Type": "Task",
  "Resource": "arn:aws:states:::http:invoke",
  "Parameters": {
    "ApiEndpoint": "https://api.stripe.com/v1/customers",
    "Authentication": {
      "ConnectionArn": "arn:aws:events:us-east-2:123456789012:connection/Stripe/81210c42-8af1-456b-9c4a-6ff02fc664ac"
    }
  },
}
```

```
    "Method": "GET"
  },
  "End": true
}
```

HTTP Task fields

An HTTP Task includes the following fields in its definition.

Resource (Required)

To specify a [task type](#), provide its ARN in the Resource field. For an HTTP Task, you specify the Resource field as follows.

```
"Resource": "arn:aws:states:::http:invoke"
```

Parameters (Required)

Contains the `ApiEndpoint`, `Method`, and `ConnectionArn` fields that provide information about the third-party API you want to call. Parameters also contains optional fields, such as `Headers` and `QueryParameters`.

You can specify a combination of static JSON and [JsonPath](#) syntax as Parameters in the Parameters field. For more information, see [Pass parameters to a service API](#).

ApiEndpoint (Required)

Specifies the URL of the third-party API you want to call. To append query parameters to the URL, use the [QueryParameters](#) field. The following example shows how you can call a Stripe API to fetch the list of all customers.

```
"ApiEndpoint": "https://api.stripe.com/v1/customers"
```

You can also specify a [reference path](#) using the [JsonPath](#) syntax to select the JSON node that contains the third-party API URL. For example, say you want to call one of Stripe's APIs using a specific customer ID. Imagine that you've provided the following state input.

```
{
  "customer_id": "1234567890",
  "name": "John Doe"
}
```

```
}
```

To retrieve the details of this customer ID using a Stripe API, specify the `ApiEndpoint` as shown in the following example. This example uses an [intrinsic function](#) and a reference path.

```
"ApiEndpoint.$": "States.Format('https://api.stripe.com/v1/customers/{}',  
$.customer_id)"
```

At runtime, Step Functions resolves the value of `ApiEndpoint` as follows.

```
https://api.stripe.com/v1/customers/1234567890
```

Method (Required)

Specifies the HTTP method you want to use for calling a third-party API. You can specify one of these methods in your HTTP Task: GET, POST, PUT, DELETE, PATCH, OPTIONS, or HEAD.

For example, to use the GET method, specify the `Method` field as follows.

```
"Method": "GET"
```

You can also use a [reference path](#) to specify the method at runtime. For example, `"Method.$": "$.myHTTPMethod"`.

Authentication (Required)

Contains the `ConnectionArn` field that specifies how to authenticate a third-party API call. Step Functions supports authentication for a specified `ApiEndpoint` using the connection resource of Amazon EventBridge.

ConnectionArn (Required)

Specifies the EventBridge connection ARN.

An HTTP Task requires an [EventBridge connection](#), which securely manages the authentication credentials of an API provider. A *connection* specifies the authorization type and credentials to use for authorizing a third-party API. Using a connection helps you avoid hard-coding secrets, such as API keys, into your state machine definition. In a connection, you can also specify [Headers](#), [QueryParameters](#), and [RequestBody](#) parameters.

When you create an EventBridge connection, you provide your authentication details. For more information about how authentication works for an HTTP Task, see [Authentication for an HTTP Task](#).

The following example shows how you can specify the `Authentication` field in your HTTP Task definition.

```
"Authentication": {
  "ConnectionArn": "arn:aws:events:us-east-2:123456789012:connection/
Stripe/81210c42-8af1-456b-9c4a-6ff02fc664ac"
}
```

Headers (Optional)

Provides additional context and metadata to the API endpoint. You can specify headers as a string or JSON array.

You can specify headers in the EventBridge connection and the `Headers` field in an HTTP Task. We recommend that you do not include authentication details to your API providers in the `Headers` field. We recommend that you include these details into your EventBridge connection.

Step Functions adds the headers that you specify in the EventBridge connection to the headers that you specify in the HTTP Task definition. If the same header keys are present in the definition and connection, Step Functions uses the corresponding values specified in the EventBridge connection for those headers. For more information about how Step Functions performs data merging, see [Merging EventBridge connection and HTTP Task definition data](#).

The following example specifies a header that will be included in a third-party API call: `content-type`.

```
"Headers": {
  "content-type": "application/json"
}
```

You can also use a [reference path](#) to specify the headers at runtime. For example, **"Headers.\$": "\$.myHTTPHeaders"**.

Step Functions sets the `User-Agent`, `Range`, and `Host` headers. Step Functions sets the value of the `Host` header based on the API you're calling. The following is an example of these headers.

```
User-Agent: Amazon|StepFunctions|HttpInvoke|us-east-1,  
Range: bytes=0-262144,  
Host: api.stripe.com
```

You can't use the following headers in your HTTP Task definition. If you use these headers, the HTTP Task fails with the [States.Runtime](#) error.

- A-IM
- Accept-Charset
- Accept-Datetime
- Accept-Encoding
- Cache-Control
- Connection
- Content-Encoding
- Content-MD5
- Date
- Expect
- Forwarded
- From
- Host
- HTTP2-Settings
- If-Match
- If-Modified-Since
- If-None-Match
- If-Range
- If-Unmodified-Since
- Max-Forwards
- Origin
- Pragma
- Proxy-Authorization
- Referer
- Server

- TE
- Trailer
- Transfer-Encoding
- Upgrade
- Via
- Warning
- x-forwarded-*
- x-amz-*
- x-amzn-*

QueryParameters (Optional)

Inserts key-value pairs at the end of an API URL. You can specify query parameters as a string, JSON array, or a JSON object. Step Functions automatically URL-encodes query parameters when it calls a third-party API.

For example, say that you want to call the Stripe API to search for customers that do their transactions in US dollars (USD). Imagine that you've provided the following `QueryParameters` as state input.

```
"QueryParameters": {  
  "currency": "usd"  
}
```

At runtime, Step Functions appends the `QueryParameters` to the API URL as follows.

```
https://api.stripe.com/v1/customers/search?currency=usd
```

You can also use a [reference path](#) to specify the query parameters at runtime. For example, **"QueryParameters.\$": "\$.myQueryParameters"**.

If you've specified query parameters in your EventBridge connection, Step Functions adds these query parameters to the query parameters that you specify in the HTTP Task definition. If the same query parameters keys are present in the definition and connection, Step Functions uses the corresponding values specified in the EventBridge connection for those headers. For more information about how Step Functions performs data merging, see [Merging EventBridge connection and HTTP Task definition data](#).

Transform (Optional)

Contains the `RequestBodyEncoding` and `RequestEncodingOptions` fields. By default, Step Functions sends the request body as JSON data to an API endpoint.

If your API provider accepts `form-urlencoded` request bodies, use the `Transform` field to specify URL-encoding for the request bodies. You must also specify the `content-type` header as `application/x-www-form-urlencoded`. Step Functions then automatically URL-encodes your request body.

RequestBodyEncoding

Specifies URL-encoding of your request body. You can specify one of these values: `NONE` or `URL_ENCODED`.

- `NONE` – The HTTP request body will be the serialized JSON of the `RequestBody` field. This is the default value.
- `URL_ENCODED` – The HTTP request body will be the URL-encoded form data of the `RequestBody` field.

RequestEncodingOptions

Determines the encoding option to use for arrays in your request body if you set `RequestBodyEncoding` to `URL_ENCODED`.

Step Functions supports the following array encoding options. For more information about these options and their examples, see [Applying URL-encoding on request body](#).

- `INDICES` – Encodes arrays using the index value of array elements. By default, Step Functions uses this encoding option.
- `REPEAT` – Repeats a key for each item in an array.
- `COMMAS` – Encodes all the values in a key as a comma-delimited list of values.
- `BRACKETS` – Repeats a key for each item in an array and appends a bracket, `[]`, to the key to indicate that it is an array.

The following example sets URL-encoding for the request body data. It also specifies to use the `COMMAS` encoding option for arrays in the request body.

```
"Transform": {
  "RequestBodyEncoding": "URL_ENCODED",
```

```
"RequestEncodingOptions": {
  "ArrayFormat": "COMMAS"
}
```

RequestBody (Optional)

Accepts JSON data that you provide in the state input. In `RequestBody`, you can specify a combination of static JSON and [JsonPath](#) syntax. For example, say that you provide the following state input:

```
{
  "CardNumber": "1234567890",
  "ExpiryDate": "09/25"
}
```

To use these values of `CardNumber` and `ExpiryDate` in your request body at runtime, you can specify the following JSON data in your request body.

```
"RequestBody": {
  "Card": {
    "Number.$": "$.CardNumber",
    "Expiry.$": "$.ExpiryDate",
    "Name": "John Doe",
    "Address": "123 Any Street, Any Town, USA"
  }
}
```

If the third-party API you want to call requires `form-urlencoded` request bodies, you must specify URL-encoding for your request body data. For more information, see [Applying URL-encoding on request body](#).

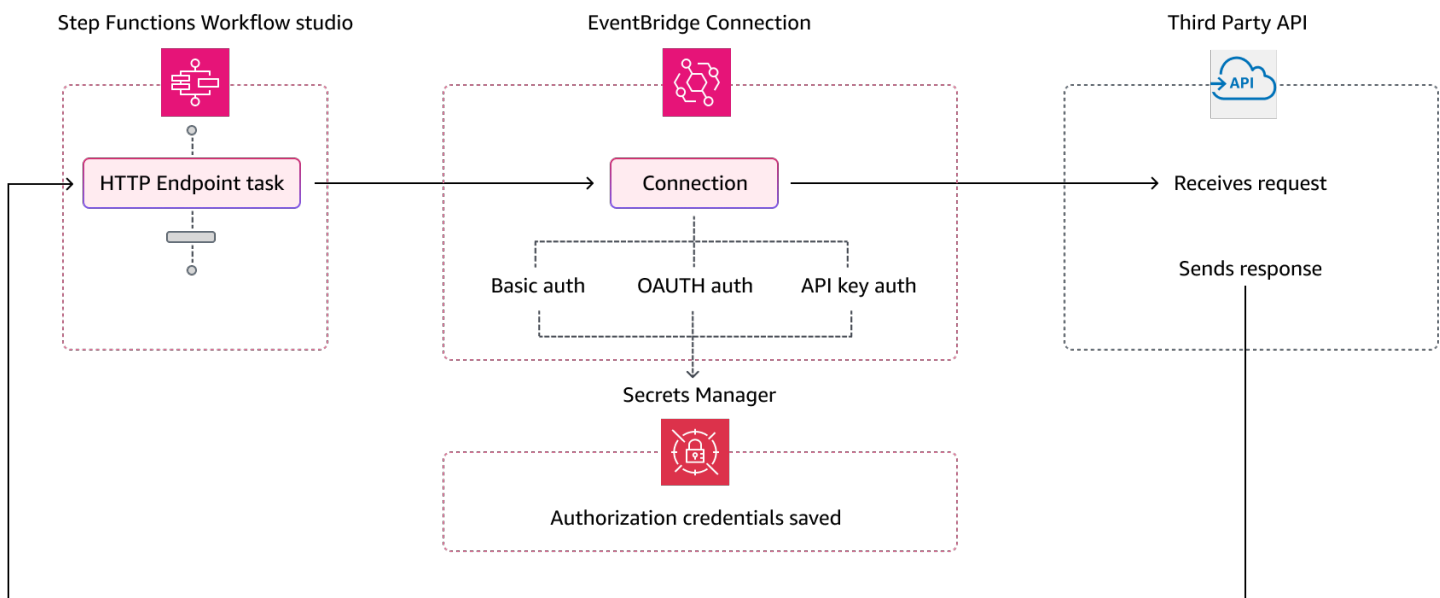
Authentication for an HTTP Task

An HTTP Task requires an [EventBridge connection](#), which securely manages the authentication credentials of an API provider. A *connection* specifies the authorization type and credentials to use for authorizing a third-party API. Using a connection helps you avoid hard-coding secrets, such as API keys, into your state machine definition. An EventBridge connection supports the Basic, OAuth, and API Key authorization schemes.

When you create an EventBridge connection, you provide your authentication details. You can also include the header, body, and query parameters that are required for authorization with an API. You must include the connection ARN in any HTTP Task that calls a third-party API.

When you create a connection and add authorization parameters, EventBridge creates a [secret](#) in AWS Secrets Manager. In this secret, EventBridge stores the connection and authorization parameters in an encrypted form. To successfully create or update a connection, you must use an AWS account that has permission to use Secrets Manager. For more information about the IAM permissions your state machine needs to access an EventBridge connection, see [IAM permissions to run an HTTP Task](#).

The following image shows how Step Functions handles authentication for third-party API calls using an EventBridge connection.



Merging EventBridge connection and HTTP Task definition data

When you invoke an HTTP Task, you can specify data in your EventBridge connection and your HTTP Task definition. This data includes [Headers](#), [QueryParameters](#), and [RequestBody](#) parameters. Before calling a third-party API, Step Functions merges the request body with the connection body parameters in all cases except if your request body is a string and the connection body parameters is non-empty. In this case, the HTTP Task fails with the [States.Runtime](#) error.

If there are any duplicate keys specified in the HTTP Task definition and the EventBridge connection, Step Functions overwrites the values in the HTTP Task with the values in the connection.

The following list describes how Step Functions merges data before calling a third-party API:

- **Headers** – Step Functions adds any headers you specified in the connection to the headers in the `Headers` field of the HTTP Task. If there is a conflict between the header keys, Step Functions uses the values specified in the connection for those headers. For example, if you specified the `content-type` header in both the HTTP Task definition and EventBridge connection, Step Functions uses the `content-type` header value specified in the connection.
- **Query parameters** – Step Functions adds any query parameters you specified in the connection to the query parameters in the `QueryParameters` field of the HTTP Task. If there is a conflict between the query parameter keys, Step Functions uses the values specified in the connection for those query parameters. For example, if you specified the `maxItems` query parameter in both the HTTP Task definition and EventBridge connection, Step Functions uses the `maxItems` query parameter value specified in the connection.
- **Body parameters**
 - Step Functions adds any request body values specified in the connection to the request body in the `RequestBody` field of the HTTP Task. If there is a conflict between the request body keys, Step Functions uses the values specified in the connection for the request body. For example, say that you specified a `Mode` field in the `RequestBody` of both the HTTP Task definition and EventBridge connection. Step Functions uses the `Mode` field value you specified in the connection.
 - If you specify the request body as a string instead of a JSON object, and the EventBridge connection also contains request body, Step Functions can't merge the request body specified in both these places. It fails the HTTP Task with the [States.Runtime](#) error.

Step Functions applies all transformations and serializes the request body after it completes the merging of the request body.

The following example sets the `Headers`, `QueryParameters`, and `RequestBody` fields in both the HTTP Task and EventBridge connection.

HTTP Task definition

```
{
  "Comment": "Data merging example for HTTP Task and EventBridge connection",
  "StartAt": "ListCustomers",
  "States": {
    "ListCustomers": {
```

```

    "Type": "Task",
    "Resource": "arn:aws:states:::http:invoke",
    "Parameters": {
      "Authentication": {
        "ConnectionArn": "arn:aws:events:us-
east-1:123456789012:connection/Example/81210c42-8af1-456b-9c4a-6ff02fc664ac"
      },
      "ApiEndpoint": "https://example.com/path",
      "Method": "GET",
      "Headers": {
        "Request-Id": "my_request_id",
        "Header-Param": "state_machine_header_param"
      },
      "RequestBody": {
        "Job": "Software Engineer",
        "Company": "AnyCompany",
        "BodyParam": "state_machine_body_param"
      },
      "QueryParameters": {
        "QueryParam": "state_machine_query_param"
      }
    }
  }
}
}
}
}

```

EventBridge connection

```

{
  "AuthorizationType": "API_KEY",
  "AuthParameters": {
    "ApiKeyAuthParameters": {
      "ApiKeyName": "ApiKey",
      "ApiKeyValue": "key_value"
    },
    "InvocationHttpParameters": {
      "BodyParameters": [
        {
          "Key": "BodyParam",
          "Value": "connection_body_param"
        }
      ],
      "HeaderParameters": [

```

```
{
  {
    "Key": "Header-Param",
    "Value": "connection_header_param"
  }
],
"QueryStringParameters": [
  {
    "Key": "QueryParam",
    "Value": "connection_query_param"
  }
]
}
}
```

In this example, duplicate keys are specified in the HTTP Task and EventBridge connection. Therefore, Step Functions overwrites the values in the HTTP Task with the values in the connection. The following code snippet shows the HTTP request that Step Functions sends to the third-party API.

```
POST /path?QueryParam=connection_query_param HTTP/1.1
Apikey: key_value
Content-Length: 79
Content-Type: application/json; charset=UTF-8
Header-Param: connection_header_param
Host: example.com
Range: bytes=0-262144
Request-Id: my_request_id
User-Agent: Amazon|StepFunctions|HttpInvoke|us-east-1

{"Job":"Software Engineer","Company":"AnyCompany","BodyParam":"connection_body_param"}
```

Applying URL-encoding on request body

By default, Step Functions sends the request body as JSON data to an API endpoint. If your third-party API provider requires `form-urlencoded` request bodies, you must specify URL-encoding for the request bodies. Step Functions then automatically URL-encodes the request body based on the URL-encoding option you select.

You specify URL-encoding using the [Transform](#) field. This field contains the [RequestBodyEncoding](#) field that specifies whether or not you want to apply URL-encoding for

your request bodies. When you specify the `RequestBodyEncoding` field, Step Functions converts your JSON request body to `form-urlencoded` request body before calling the third-party API. You must also specify the `content-type` header as `application/x-www-form-urlencoded` because APIs that accept URL-encoded data expect the `content-type` header.

To encode arrays in your request body, Step Functions provides the following array encoding options.

- **INDICES** – Repeats a key for each item in an array and appends a bracket, `[]`, to the key to indicate that it is an array. This bracket contains the index of the array element. Adding the index helps you specify the order of the array elements. By default, Step Functions uses this encoding option.

For example, if your request body contains the following array.

```
{"array": ["a", "b", "c", "d"]}
```

Step Functions encodes this array to the following string.

```
array[0]=a&array[1]=b&array[2]=c&array[3]=d
```

- **REPEAT** – Repeats a key for each item in an array.

For example, if your request body contains the following array.

```
{"array": ["a", "b", "c", "d"]}
```

Step Functions encodes this array to the following string.

```
array=a&array=b&array=c&array=d
```

- **COMMAS** – Encodes all the values in a key as a comma-delimited list of values.

For example, if your request body contains the following array.

```
{"array": ["a", "b", "c", "d"]}
```

Step Functions encodes this array to the following string.

```
array=a,b,c,d
```

- **BRACKETS** – Repeats a key for each item in an array and appends a bracket, [], to the key to indicate that it is an array.

For example, if your request body contains the following array.

```
{"array": ["a","b","c","d"]}
```

Step Functions encodes this array to the following string.

```
array[]=a&array[]=b&array[]=c&array[]=d
```

IAM permissions to run an HTTP Task

Your state machine execution role must have the `states:InvokeHTTPEndpoint`, `events:RetrieveConnectionCredentials`, `secretsmanager:GetSecretValue`, and `secretsmanager:DescribeSecret` permissions for an HTTP Task to call a third-party API. The following IAM policy example grants the least privileges required to your state machine role for calling Stripe APIs. This IAM policy also grants permission to the state machine role to access a specific EventBridge connection, including the secret for this connection that is stored in Secrets Manager.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Statement1",
      "Effect": "Allow",
      "Action": "states:InvokeHTTPEndpoint",
      "Resource": "arn:aws:states:us-east-2:123456789012:stateMachine:myStateMachine",
      "Condition": {
        "StringEquals": {
          "states:HTTPMethod": "GET"
        },
        "StringLike": {
          "states:HTTPEndpoint": "https://api.stripe.com/*"
        }
      }
    }
  ]
}
```

```

    }
  },
  {
    "Sid": "Statement2",
    "Effect": "Allow",
    "Action": [
      "events:RetrieveConnectionCredentials",
    ],
    "Resource": "arn:aws:events:us-
east-2:123456789012:connection/oauth_connection/aeabd89e-d39c-4181-9486-9fe03e6f286a"
  },
  {
    "Sid": "Statement3",
    "Effect": "Allow",
    "Action": [
      "secretsmanager:GetSecretValue",
      "secretsmanager:DescribeSecret"
    ],
    "Resource": "arn:aws:secretsmanager:*:*:secret:events!connection/*"
  }
]
}

```

HTTP Task example

The following state machine definition shows an HTTP Task that includes the [Headers](#), [QueryParameters](#), [Transform](#), and [RequestBody](#) parameters. The HTTP Task calls a Stripe API, <https://api.stripe.com/v1/invoices>, to generate an invoice. The HTTP Task also specifies URL-encoding for the request body using the INDICES encoding option.

Make sure that you've created an EventBridge connection. The following example shows a connection created using BASIC auth type.

```

{
  "Type": "BASIC",
  "AuthParameters": {
    "BasicAuthParameters": {
      "Password": "myPassword",
      "Username": "myUsername"
    }
  }
}

```

Remember to replace the *italicized* text with your resource-specific information.

```
{
  "Comment": "A state machine that uses HTTP Task",
  "StartAt": "CreateInvoiceAPI",
  "States": {
    "CreateInvoiceAPI": {
      "Type": "Task",
      "Resource": "arn:aws:states:::http:invoke",
      "Parameters": {
        "ApiEndpoint": "https://api.stripe.com/v1/invoices",
        "Method": "POST",
        "Authentication": {
          "ConnectionArn": "arn:aws:events:us-east-2:123456789012:connection/Stripe/81210c42-8af1-456b-9c4a-6ff02fc664ac"
        },
        "Headers": {
          "Content-Type": "application/x-www-form-urlencoded"
        },
        "RequestBody": {
          "customer.$": "$.customer_id",
          "description": "Monthly subscription",
          "metadata": {
            "order_details": "monthly report data"
          }
        },
        "Transform": {
          "RequestBodyEncoding": "URL_ENCODED",
          "RequestEncodingOptions": {
            "ArrayFormat": "INDICES"
          }
        }
      },
      "Retry": [
        {
          "ErrorEquals": [
            "States.Http.StatusCode.429",
            "States.Http.StatusCode.503",
            "States.Http.StatusCode.504",
            "States.Http.StatusCode.502"
          ],
          "BackoffRate": 2,
          "IntervalSeconds": 1,
          "MaxAttempts": 3,
        }
      ]
    }
  }
}
```



```
        "JitterStrategy": "FULL"
      }
    ],
    "Catch": [
      {
        "ErrorEquals": [
          "States.Http.StatusCode.404",
          "States.Http.StatusCode.400",
          "States.Http.StatusCode.401",
          "States.Http.StatusCode.409",
          "States.Http.StatusCode.500"
        ],
        "Comment": "Handle all non 200 ",
        "Next": "HandleInvoiceFailure"
      }
    ],
    "End": true
  }
}
```

To run this state machine, provide the customer ID as the input as shown in the following example:

```
{
  "customer_id": "1234567890"
}
```

The following example shows the HTTP request that Step Functions sends to the Stripe API.

```
POST /v1/invoices HTTP/1.1
Authorization: Basic <base64 of username and password>
Content-Type: application/x-www-form-urlencoded
Host: api.stripe.com
Range: bytes=0-262144
Transfer-Encoding: chunked
User-Agent: Amazon|StepFunctions|HttpInvoke|us-east-1

description=Monthly%20subscription&metadata%5Border_details%5D=monthly%20report
%20data&customer=1234567890
```

Testing an HTTP Task

You can use the [TestState](#) API through the console, SDK, or the AWS CLI to [test](#) an HTTP Task. The following procedure describes how to use the TestState API in the Step Functions console. You can iteratively test the API request, response, and authentication details until your HTTP Task is working as expected.

Test an HTTP Task state in Step Functions console

1. Open the [Step Functions console](#).
2. Choose **Create state machine** to start creating a state machine or choose an existing state machine that contains an HTTP Task.

Refer Step 4 if you're testing the task in an existing state machine.

3. In the [Design mode](#) of Workflow Studio, configure an HTTP Task visually. Or choose the Code mode to copy-paste the state machine definition from your local development environment.
4. In Design mode, choose **Test state** in the [Inspector](#) panel of Workflow Studio.
5. In the **Test state** dialog box, do the following:
 - a. For **Execution role**, choose an execution role to test the state. If you don't have a role with [sufficient permissions](#) for an HTTP Task, see [Role for testing HTTP Tasks in Workflow Studio](#) to create a role.
 - b. (Optional) Provide any JSON input that your selected state needs for the test.
 - c. For **Inspection level**, keep the default selection of **INFO**. This level shows you the status of the API call and the state output. This is useful for quickly checking the API response.
 - d. Choose **Start test**.
 - e. If the test succeeds, the state output appears on the right side of the **Test state** dialog box. If the test fails, an error appears.

In the **State details** tab of the dialog box, you can see the state definition and a link to your [EventBridge connection](#).

- f. Change the **Inspection level** to **TRACE**. This level shows you the raw HTTP request and response, and is useful for verifying headers, query parameters, and other API-specific details.
- g. Choose the **Reveal secrets** checkbox. In combination with **TRACE**, this setting lets you see the sensitive data that the EventBridge connection inserts, such as API keys. The IAM

user identity that you use to access the console must have permission to perform the `states:RevealSecrets` action. Without this permission, Step Functions throws an access denied error when you start the test. For an example of an IAM policy that sets the `states:RevealSecrets` permission, see [IAM permissions for using TestState API](#).

The following image shows a test for an HTTP Task that succeeds. The **Inspection level** for this state is set to **TRACE**. The **HTTP request & response** tab in the following image shows the result of the third-party API call.

Test state

Test a state in isolation using the TestState API to ensure that it works correctly. [Learn more](#)

State Call Stripe API succeeded.
▶ Details

Test | State details

Execution role
Testing a state requires an execution role. Enter an IAM role ARN, select an existing IAM role from the list, or [learn how to create a new IAM role with permissions for an HTTP task](#)

myHTTPTaskRole

State input - optional

```
1 {
2   "customer_id": "cus_OvaX00rSMf3NdJ"
3 }
```

Must be in valid JSON format

Inspection level
Specifies the level of detail to return from this test. [Learn more](#)

TRACE
Returns TRACE-level detail + HTTP request/response for HTTP tasks

Reveal secrets
Applies to HTTP tasks only. When combined with an inspection level of TRACE, will reveal any sensitive authorization data in the HTTP request and response. [Learn more](#)

Start test

Output | Input/output processing | **HTTP request & response**

Expand all

```
{ 2 items
  "request": { 4 items
    "headers":
      "[Authorization: Basic
      ,
      User-Agent: Amazon/StepFunctions/HttpInvoke/
      , Range: bytes=0-262144]"
    "method": "GET"
    "protocol": "https"
    "url":
      "https://api.stripe.com/v1/customers/cus_OvaX00rSMf3NdJ"
  }
  "response": { 5 items
    "body": { 22 items
      "id": "cus_OvaX00rSMf3NdJ"
      "object": "customer"
      "address": NULL
    }
  }
}
```

Copy TestState API response Done

- h. Choose **Start test**.
- i. If the test succeeds, you can see your HTTP details under the **HTTP request & response** tab.

Unsupported HTTP Task responses

An HTTP Task fails with the [States.Runtime](#) error if one of the following conditions is true for the response returned:

- The response contains a content-type header of `application/octet-stream`, `image/*`, `video/*`, or `audio/*`.
- The response can't be read as a valid string. For example, binary or image data.

Service integration patterns

AWS Step Functions integrates with services directly in the Amazon States Language. You can control these AWS services using three service integration patterns:

- Call a service and let Step Functions progress to the next state immediately after it gets an HTTP response.
- Call a service and have Step Functions wait for a job to complete.
- Call a service with a task token and have Step Functions wait until that token is returned with a payload.

Each of these service integration patterns is controlled by how you create a URI in the "Resource" field of your [task definition](#).

Ways to Call an Integrated Service

- [Request Response](#)
- [Run a Job \(.sync\)](#)
- [Wait for a Callback with the Task Token](#)

For information about configuring AWS Identity and Access Management (IAM) for integrated services, see [IAM Policies for integrated services](#).

Request Response

When you specify a service in the "Resource" string of your task state, and you *only* provide the resource, Step Functions will wait for an HTTP response and then progress to the next state. Step Functions will not wait for a job to complete.

The following example shows how you can publish an Amazon SNS topic.

```
"Send message to SNS":{
  "Type":"Task",
  "Resource":"arn:aws:states:::sns:publish",
  "Parameters":{
    "TopicArn":"arn:aws:sns:us-east-1:123456789012:myTopic",
    "Message":"Hello from Step Functions!"
  },
  "Next":"NEXT_STATE"
}
```

This example references the [Publish](#) API of Amazon SNS. The workflow progresses to the next state after calling the Publish API.

Tip

To deploy a sample workflow that uses the Request Response service integration pattern to your AWS account, see [Module 2 - Request Response](#) of *The AWS Step Functions Workshop*.

Run a Job (.sync)

For integrated services such as AWS Batch and Amazon ECS, Step Functions can wait for a request to complete before progressing to the next state. To have Step Functions wait, specify the "Resource" field in your task state definition with the `.sync` suffix appended after the resource URI.

For example, when submitting an AWS Batch job, use the "Resource" field in the state machine definition as shown in this example.

```
"Manage Batch task": {
  "Type": "Task",
  "Resource": "arn:aws:states:::batch:submitJob.sync",
  "Parameters": {
    "JobDefinition": "arn:aws:batch:us-east-2:123456789012:job-definition/
testJobDefinition",
    "JobName": "testJob",
    "JobQueue": "arn:aws:batch:us-east-2:123456789012:job-queue/testQueue"
  },
}
```

```
"Next": "NEXT_STATE"  
}
```

Having the `.sync` portion appended to the resource Amazon Resource Name (ARN) means that Step Functions waits for the job to complete. After calling `AWS Batch submitJob`, the workflow pauses. When the job is complete, Step Functions progresses to the next state. For more information, see the AWS Batch sample project: [Manage a batch job \(AWS Batch, Amazon SNS\)](#).

If a task using this (`.sync`) service integration pattern is aborted, and Step Functions is unable to cancel the task, you might incur additional charges from the integrated service. A task can be aborted if:

- The state machine execution is stopped.
- A different branch of a Parallel state fails with an uncaught error.
- An iteration of a Map state fails with an uncaught error.

Step Functions will make a best-effort attempt to cancel the task. For example, if a Step Functions `states:startExecution.sync` task is aborted, it will call the Step Functions `StopExecution` API action. However, it is possible that Step Functions will be unable to cancel the task. Reasons for this include, but are not limited to:

- Your IAM execution role lacks permission to make the corresponding API call.
- A temporary service outage occurred.

When you use the `.sync` service integration pattern, Step Functions uses polling that consumes your assigned quota and events to monitor a job's status. For `.sync` invocations within the same account, Step Functions uses EventBridge events and polls the APIs that you specify in the Task state. For [cross-account](#) `.sync` invocations, Step Functions only uses polling. For example, for `states:StartExecution.sync`, Step Functions performs polling on the [DescribeExecution](#) API and uses your assigned quota.

Tip

To deploy a sample workflow that uses the Run a Job (`.sync`) service integration pattern to your AWS account, see [Module 3 - Run a Job \(.sync\)](#) of *The AWS Step Functions Workshop*.

To see a list of what integrated services support waiting for a job to complete (`.sync`), see [Optimized integrations for Step Functions](#).

Note

Service integrations that use the `.sync` pattern require additional IAM permissions. For more information, see [IAM Policies for integrated services](#).

In some cases, you may want Step Functions to continue your workflow before the job is fully complete. You can achieve this in the same way as when using the [Wait for a Callback with the Task Token](#) service integration pattern. To do this, pass a task token to your job, then return it using a [SendTaskSuccess](#) or [SendTaskFailure](#) API call. Step Functions will use the data you provide in that call to complete the task, stop monitoring the job, and continue the workflow.

Wait for a Callback with the Task Token

Callback tasks provide a way to pause a workflow until a task token is returned. A task might need to wait for a human approval, integrate with a third party, or call legacy systems. For tasks like these, you can pause Step Functions until the workflow execution reaches the one year service quota (see, [Quotas related to state throttling](#)), and wait for an external process or workflow to complete. For these situations Step Functions allows you to pass a task token to the AWS SDK service integrations, and also to some Optimized service integrations. The task will pause until it receives that task token back with a [SendTaskSuccess](#) or [SendTaskFailure](#) call.

If a Task state using the callback task token times out, a new random token is generated. You can access the task tokens from the [context object](#).

Note

A task token must contain at least one character, and cannot exceed 1024 characters.

To use `.waitForTaskToken` with an AWS SDK integration, the API you use must have a parameter field in which to place the task token.

Note

You must pass task tokens from principals within the same AWS account. The tokens won't work if you send them from principals in a different AWS account.

Tip

To deploy a sample workflow that uses a callback task token service integration pattern to your AWS account, see [Module 4 - Wait for a Callback with the Task Token](#) of *The AWS Step Functions Workshop*.

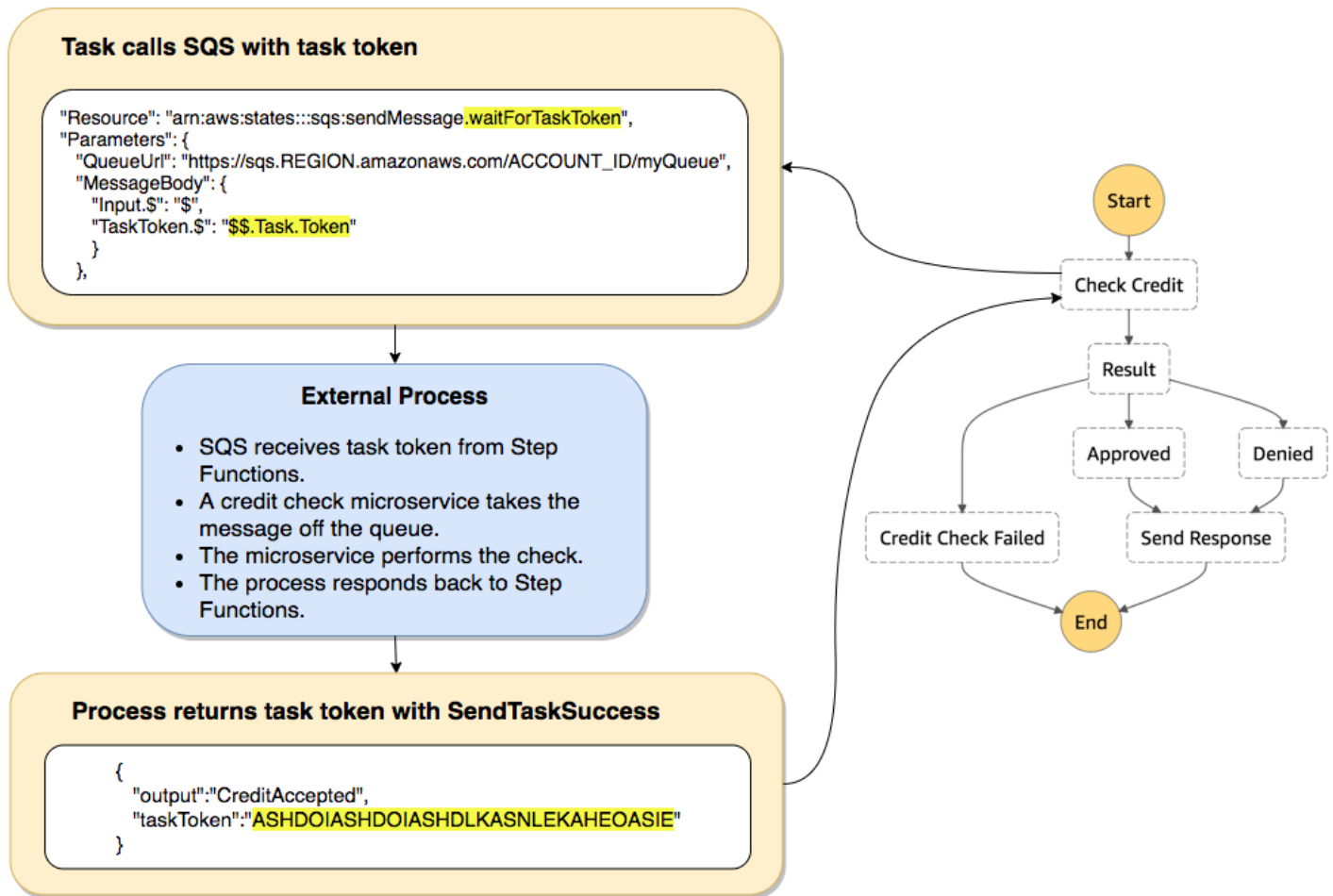
To see a list of what integrated services support waiting for a task token (`.waitForTaskToken`), see [Optimized integrations for Step Functions](#).

Topics

- [Task Token Example](#)
- [Get a Token from the Context Object](#)
- [Configure a Heartbeat Timeout for a Waiting Task](#)

Task Token Example

In this example, a Step Functions workflow needs to integrate with an external microservice to perform a credit check as a part of an approval workflow. Step Functions publishes an Amazon SQS message that includes a task token as a part of the message. An external system integrates with Amazon SQS, and pulls the message off the queue. When that's finished, it returns the result and the original task token. Step Functions then continues with its workflow.



The "Resource" field of the task definition that references Amazon SQS includes `.waitForTaskToken` appended to the end.

```

"Send message to SQS": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
  "Parameters": {
    "QueueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/myQueue",
    "MessageBody": {
      "Message": "Hello from Step Functions!",
      "TaskToken.$": "$$.Task.Token"
    }
  },
  "Next": "NEXT_STATE"
}

```

This tells Step Functions to pause and wait for the task token. When you specify a resource using `.waitForTaskToken`, the task token can be accessed in the "Parameters" field of your state definition with a special path designation (`$.Task.Token`). The initial `$.` designates that the path accesses the [context object](#), and gets the task token for the current task in a running execution.

When it's complete, the external service calls [SendTaskSuccess](#) or [SendTaskFailure](#) with the `taskToken` included. Only then does the workflow continue to the next state.

Note

To avoid waiting indefinitely if a process fails to send the task token with `SendTaskSuccess` or `SendTaskFailure`, see [Configure a Heartbeat Timeout for a Waiting Task](#).

Get a Token from the Context Object

The context object is an internal JSON object that contains information about your execution. Like state input, it can be accessed with a path from the "Parameters" field during an execution. When accessed from within a task definition, it includes information about the specific execution, including the task token.

```
{
  "Execution": {
    "Id": "arn:aws:states:us-
east-1:123456789012:execution:stateMachineName:executionName",
    "Input": {
      "key": "value"
    },
    "Name": "executionName",
    "RoleArn": "arn:aws:iam::123456789012:role...",
    "StartTime": "2019-03-26T20:14:13.192Z"
  },
  "State": {
    "EnteredTime": "2019-03-26T20:14:13.192Z",
    "Name": "Test",
    "RetryCount": 3
  },
  "StateMachine": {
    "Id": "arn:aws:states:us-east-1:123456789012:stateMachine:stateMachineName",
```

```

    "Name": "name"
  },
  "Task": {
    "Token": "h7XRiCdLtd/83p1E0dMccoxIzFhglSdKzpK9mBVkZsp7d9yrT1W"
  }
}

```

You can access the task token by using a special path from inside the "Parameters" field of your task definition. To access the input or the context object, you first specify that the parameter will be a path by appending a `.$` to the parameter name. The following specifies nodes from both the input and the context object in a "Parameters" specification.

```

"Parameters": {
  "Input.$": "$",
  "TaskToken.$": "$$.Task.Token"
},

```

In both cases, appending `.$` to the parameter name tells Step Functions to expect a path. In the first case, `"$"` is a path that includes the entire input. In the second case, `$$.` specifies that the path will access the context object, and `$$.Task .Token` sets the parameter to the value of the task token in the context object of a running execution.

In the Amazon SQS example, `.waitForTaskToken` in the "Resource" field tells Step Functions to wait for the task token to be returned. The `"TaskToken.$": "$$.Task.Token"` parameter passes that token as a part of the Amazon SQS message.

```

"Send message to SQS": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
  "Parameters": {
    "QueueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/myQueue",
    "MessageBody": {
      "Message": "Hello from Step Functions!",
      "TaskToken.$": "$$.Task.Token"
    }
  }
},
"Next": "NEXT_STATE"
}

```

For more information about the context object, see [Context object](#) in the [Input and Output Processing](#) section in this guide.

Configure a Heartbeat Timeout for a Waiting Task

A task that is waiting for a task token will wait until the execution reaches the one year service quota (see, [Quotas related to state throttling](#)). To avoid stuck executions you can configure a heartbeat timeout interval in your state machine definition. Use the [HeartbeatSeconds](#) field to specify the timeout interval.

```
{
  "StartAt": "Push to SQS",
  "States": {
    "Push to SQS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
      "HeartbeatSeconds": 600,
      "Parameters": {
        "MessageBody": { "myTaskToken.$": "$$.Task.Token" },
        "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/push-based-queue"
      },
      "ResultPath": "$.SQS",
      "End": true
    }
  }
}
```

In this state machine definition, a task pushes a message to Amazon SQS and waits for an external process to call back with the provided task token. The "HeartbeatSeconds": 600 field sets the heartbeat timeout interval to 10 minutes. The task will wait for the task token to be returned with one of these API actions:

- [SendTaskSuccess](#)
- [SendTaskFailure](#)
- [SendTaskHeartbeat](#)

If the waiting task doesn't receive a valid task token within that 10-minute period, the task fails with a `States.Timeout` error name.

For more information, see the callback task sample project [Callback Pattern Example \(Amazon SQS, Amazon SNS, Lambda\)](#).

Pass parameters to a service API

Use the `Parameters` field in a Task state to control what parameters are passed to a service API.

Inside the `Parameters` field, you must use the plural form of the array parameters in an API action. For example, if you use the [Filter](#) field of the `DescribeSnapshots` API action for integrating with Amazon EC2, you must define the field as `Filters`. If you don't use the plural form, Step Functions returns the following error:

```
The field Filter is not supported by Step Functions.
```

Pass static JSON as parameters

You can include a JSON object directly in your state machine definition to pass as a parameter to a resource.

For example, to set the `RetryStrategy` parameter for the `SubmitJob` API for AWS Batch, you could include the following in your parameters.

```
"RetryStrategy": {
  "attempts": 5
}
```

You can also pass multiple parameters with static JSON. As a more complete example, the following are the `Resource` and `Parameters` fields of the specification of a task that publishes to an Amazon SNS topic named *myTopic*.

```
"Resource": "arn:aws:states:::sns:publish",
"Parameters": {
  "TopicArn": "arn:aws:sns:us-east-2:123456789012:myTopic",
  "Message": "test message",
  "MessageAttributes": {
    "my attribute no 1": {
      "DataType": "String",
      "StringValue": "value of my attribute no 1"
    },
    "my attribute no 2": {
      "DataType": "String",
      "StringValue": "value of my attribute no 2"
    }
  }
}
```

```
},
```

Pass state input as parameters using Paths

You can pass portions of the state input as parameters by using [paths](#). A path is a string, beginning with \$, that's used to identify components within JSON text. Step Functions paths use [JsonPath](#) syntax.

To specify that a parameter use a path, end the parameter name with `.$`. For example, if your state input contains text within a node named `message`, you could pass that text as a parameter using a path.

Consider the following state input:

```
{
  "comment": "A message in the state input",
  "input": {
    "message": "foo",
    "otherInfo": "bar"
  },
  "data": "example"
}
```

To pass the value of the node named `message` as a parameter, specify the following syntax:

```
"Parameters": {"myMessage.$": "$.input.message"},
```

Step Functions then passes the value `foo` as a parameter.

For more information about using parameters in Step Functions, see the following:

- [Input and Output Processing](#)
- [InputPath, Parameters and ResultSelector](#)

Pass Context Object Nodes as Parameters

In addition to static content, and nodes from the state input, you can pass nodes from the context object as parameters. The context object is dynamic JSON data that exists during a state machine execution. It includes information about your state machine and the current execution. You can access the context object using a path in the `"Parameters"` field of a state definition.

For more information about the context object and how to access that data from a "Parameters" field, see the following:

- [Context object](#)
- [Accessing the Context Object](#)
- [Get a Token from the Context Object](#)

Change log for supported AWS SDK integrations

The following table summarizes when services are initially integrated with Step Functions and when their integration API was most recently updated. For details on using integrations, see [AWS SDK service integrations](#).

Important

API action support is released on a quarterly cadence. Updates to already supported actions, such as new parameters, may not be immediately available.

Service	Initial support	Updated
AWS AppFabric	January 18, 2024	
B2B Data Interchange	January 18, 2024	
AWS Data Exports	January 18, 2024	
Amazon Bedrock	January 18, 2024	
Amazon Bedrock Agents	January 18, 2024	
Amazon Bedrock Runtime Agents	January 18, 2024	
Amazon Bedrock Runtime	January 18, 2024	

Service	Initial support	Updated	
Amazon CloudFront KeyValueStore	January 18, 2024		
Amazon CodeGuru Security	January 18, 2024		
AWS Cost Optimizat ion Hub	January 18, 2024		
Amazon DataZone	January 18, 2024		
Amazon EKS Auth	January 18, 2024		
AWS Entity Resolutio n	January 18, 2024		
AWS Free Tier	January 18, 2024		
Amazon Inspector Scan	January 18, 2024		
AWS Launch Wizard	January 18, 2024		
Amazon Managed Blockchain Query	January 18, 2024		
AWS Elemental MediaPackage V2	January 18, 2024		
AWS HealthImaging	January 18, 2024		
Network Manager	January 18, 2024		
AWS Payment Cryptography	January 18, 2024		
AWS Payment Cryptography Data	January 18, 2024		

Service	Initial support	Updated	
AWS Private CA Connector for Active Directory	January 18, 2024		
Amazon Q Business	January 18, 2024		
Amazon Q Connect	January 18, 2024		
AWS re:Post	January 18, 2024		
Amazon Timestream Query	January 18, 2024		
Amazon Timestream Write	January 18, 2024		
Trusted Advisor	January 18, 2024		
Verified Permissions	January 18, 2024		
Amazon WorkSpaces Thin Client	January 18, 2024		
AWS CloudTrail Data	June 16, 2023		
Amazon CloudWatch Internet Monitor	June 16, 2023		
Amazon Interactive Video Service RealTime	June 16, 2023		
AWS IoT TwinMaker	June 16, 2023		
Amazon OpenSearch Ingestion	June 16, 2023		

Service	Initial support	Updated	
AWS Telco Network Builder	June 16, 2023		
Amazon VPC Lattice	June 16, 2023		
AWS Backup Storage	February 17, 2023		
Amazon Chime Media Pipelines	February 17, 2023		
Amazon Chime Voice	February 17, 2023		
AWS Clean Rooms	February 17, 2023	January 18, 2024	
Amazon CodeCatalyst	February 17, 2023		
Amazon Connect Cases	February 17, 2023		
AWS Control Tower	February 17, 2023		
Amazon DocumentDB Elastic Clusters	February 17, 2023		
Amazon EMR Serverless	February 17, 2023		
Amazon IVS Chat	February 17, 2023		
Amazon Kendra Intelligent Ranking	February 17, 2023		
AWS HealthOmics	February 17, 2023		
Amazon Redshift Serverless	February 17, 2023		
Amazon Security Lake	February 17, 2023		

Service	Initial support	Updated	
AWS Health	February 17, 2023		
AWS IoT FleetWise	February 17, 2023		
AWS IoT RoboRunner	February 17, 2023		
AWS Mainframe Modernization	February 17, 2023		
AWS Migration Hub Orchestrator	February 17, 2023		
AWS Private 5G	February 17, 2023		
AWS Resource Explorer	February 17, 2023		
AWS SimSpace Weaver	February 17, 2023		
AWS Support App	February 17, 2023		
CloudWatch Observability Access Manager	February 17, 2023		
EventBridge Pipes	February 17, 2023		
EventBridge Scheduler	February 17, 2023		
IAM Roles Anywhere	February 17, 2023		
Kinesis Video WebRTC Storage	February 17, 2023		
License Manager Linux Subscriptions	February 17, 2023		

Service	Initial support	Updated	
License Manager User Subscriptions	February 17, 2023		
OpenSearch Serverless	February 17, 2023		
Route 53 ARC Zonal Shift	February 17, 2023		
SageMaker Geospatial	February 17, 2023		
SageMaker Metrics	February 17, 2023		
Systems Manager for SAP	February 17, 2023		
AWS Account Management	April 19, 2022		
AWS Amplify	September 30, 2021		
AWS App Mesh	September 30, 2021		
AWS App Runner	September 30, 2021	February 17, 2023	
AWS AppConfig	September 30, 2021		
AWS AppConfig Data	April 19, 2022		
AWS AppSync	September 30, 2021	February 17, 2023	
AWS Application Discovery Service	September 30, 2021		
AWS Application Migration Service	September 30, 2021		
AWS Audit Manager	September 30, 2021		

Service	Initial support	Updated	
AWS Auto Scaling Plans	September 30, 2021		
AWS Backup	September 30, 2021	February 17, 2023	
AWS Backup gateway	April 19, 2022	February 17, 2023	
AWS Batch	September 30, 2021	February 17, 2023	
AWS Billing Conductor	July 26, 2022	February 17, 2023	
AWS Budgets	September 30, 2021		
AWS Certificate Manager	September 30, 2021		
AWS Private Certificate Authority	September 30, 2021		
AWS Cloud Map	September 30, 2021		
AWS Cloud9	September 30, 2021		
AWS CloudFormation	September 30, 2021	February 17, 2023	
AWS CloudHSM	September 30, 2021		
AWS CloudHSM	September 30, 2021		
AWS CloudTrail	September 30, 2021	February 17, 2023	
AWS Cloud Control	April 19, 2022		
AWS CodeBuild	September 30, 2021		
AWS CodeCommit	September 30, 2021	February 17, 2023	
AWS CodeDeploy	September 30, 2021		

Service	Initial support	Updated	
AWS CodePipeline	September 30, 2021		
AWS CodeStar	September 30, 2021		
AWS CodeStar	September 30, 2021		
AWS CodeStar	September 30, 2021		
AWS Compute Optimizer	September 30, 2021	February 17, 2023	
AWS Config	September 30, 2021	July 26, 2022	
AWS Cost Explorer Service	September 30, 2021	February 17, 2023	
AWS Cost and Usage Report	September 30, 2021		
AWS Data Exchange	September 30, 2021	July 26, 2022	
AWS Data Pipeline	September 30, 2021		
AWS DataSync	September 30, 2021	July 26, 2022	
AWS Database Migration Service	September 30, 2021		
AWS Device Farm	September 30, 2021		
AWS Direct Connect	September 30, 2021		
AWS Directory Service	September 30, 2021	February 17, 2023	
AWS EC2 Instance Connect	September 30, 2021		
AWS Elastic Beanstalk	September 30, 2021		

Service	Initial support	Updated	
AWS Elemental MediaLive	September 30, 2021		
AWS Elemental MediaPackage	September 30, 2021		
AWS Elemental MediaPackage VOD	September 30, 2021		
AWS Elemental MediaStore	September 30, 2021		
AWS Fault Injection Service	September 30, 2021		
AWS Firewall Manager	September 30, 2021	February 17, 2023	
AWS Glue	September 30, 2021	February 17, 2023	
AWS Glue DataBrew	September 30, 2021		
AWS IoT Greengrass	September 30, 2021		
AWS Ground Station	September 30, 2021	February 17, 2023	
AWS Identity and Access Management	September 30, 2021		
AWS IoT	September 30, 2021	February 17, 2023	
AWS IoT 1-Click	September 30, 2021		
AWS IoT Analytics	September 30, 2021		
AWS IoT Core Device Advisor	September 30, 2021		
AWS IoT Events	September 30, 2021		

Service	Initial support	Updated	
AWS IoT Events Data	September 30, 2021		
AWS IoT Fleet Hub	September 30, 2021		
AWS IoT Greengrass Version 2	September 30, 2021		
AWS IoT Jobs Data Plane	September 30, 2021		
AWS IoT Secure Tunneling	September 30, 2021		
AWS IoT SiteWise	September 30, 2021	February 17, 2023	
AWS IoT Things Graph	September 30, 2021		
AWS IoT Wireless	September 30, 2021	February 17, 2023	
AWS Key Managemen t Service	September 30, 2021	July 26, 2022	
AWS Lake Formation	September 30, 2021	February 17, 2023	
AWS Lambda	September 30, 2021	February 17, 2023	
AWS License Manager	September 30, 2021	February 17, 2023	
AWS Marketplace	September 30, 2021	February 17, 2023	
AWS Marketplace Commerce Analytics	September 30, 2021		
AWS Marketplace Entitlement Service	September 30, 2021		
AWS Elemental MediaTailor	September 30, 2021	July 26, 2022	

Service	Initial support	Updated	
AWS Migration Hub	September 30, 2021		
AWS Migration Hub Config	September 30, 2021		
AWS Migration Hub Strategy Recommendations	April 19, 2022	February 17, 2023	
AWS Mobile	September 30, 2021		
AWS Network Firewall	September 30, 2021		
AWS OpsWorks	September 30, 2021		
AWS OpsWorks CM	September 30, 2021		
AWS Organizations	September 30, 2021	February 17, 2023	
AWS Outposts	September 30, 2021		
AWS Panorama	April 19, 2022	February 17, 2023	
Amazon Relational Database Service Performance Insights	September 30, 2021		
AWS Price List	September 30, 2021		
Amazon Relational Database Service	September 30, 2021		
AWS Resilience Hub	April 19, 2022		
AWS Resource Access Manager	September 30, 2021		
AWS Resource Groups	September 30, 2021	February 17, 2023	

Service	Initial support	Updated	
AWS Resource Groups Tagging API	September 30, 2021		
AWS RoboMaker	September 30, 2021		
AWS IAM Identity Center	September 30, 2021	February 17, 2023	
AWS SSO OIDC	September 30, 2021		
AWS Secrets Manager	September 30, 2021		
AWS Security Token Service	September 30, 2021		
AWS Security Hub	September 30, 2021		
AWS Server Migration Service	September 30, 2021		
AWS Service Catalog	September 30, 2021		
AWS Service Catalog AppRegistry	September 30, 2021	February 17, 2023	
AWS Shield	September 30, 2021		
AWS Signer	September 30, 2021		
AWS IAM Identity Center	September 30, 2021		
AWS IAM Identity Center Admin	September 30, 2021		
AWS Step Functions	September 30, 2021	February 17, 2023	
AWS Storage Gateway	September 30, 2021		

Service	Initial support	Updated	
AWS Support	September 30, 2021		
AWS Transfer Family	September 30, 2021	February 17, 2023	
AWS WAF	September 30, 2021		
AWS WAF Regional	September 30, 2021		
AWS WAFV2	September 30, 2021		
AWS Well-Architected Tool	September 30, 2021	February 17, 2023	
AWS X-Ray	September 30, 2021	February 17, 2023	
AWS Marketplace Metering Service	September 30, 2021		
AWS Serverless Application Repository	September 30, 2021		
AWS Identity and Access Management Access Analyzer	September 30, 2021		
Alexa for Business	September 30, 2021		
Amazon API Gateway	September 30, 2021	February 17, 2023	
Amazon API Gateway	September 30, 2021		
Amazon AppIntegrations	September 30, 2021		
Amazon AppStream 2.0	September 30, 2021		
Amazon AppFlow	September 30, 2021	February 17, 2023	

Service	Initial support	Updated
Amazon Athena	September 30, 2021	February 17, 2023
Amazon Augmented AI	September 30, 2021	
Amazon Braket	September 30, 2021	
Amazon Chime	September 30, 2021	
Amazon Chime Meetings	April 19, 2022	February 17, 2023
Amazon Cloud Directory	September 30, 2021	
Amazon CloudFront	September 30, 2021	February 17, 2023
Amazon CloudSearch	September 30, 2021	
Amazon CloudWatch	September 30, 2021	February 17, 2023
Amazon CloudWatch Application Insights	September 30, 2021	
CloudWatch Evidently	April 19, 2022	
Amazon CloudWatch Logs	September 30, 2021	
Amazon CloudWatch RUM	April 19, 2022	February 17, 2023
Amazon CloudWatch Synthetics	September 30, 2021	
Amazon CodeGuru Profiler	September 30, 2021	

Service	Initial support	Updated	
Amazon CodeGuru Reviewer	September 30, 2021		
Amazon Cognito	September 30, 2021		
Amazon Cognito Identity Provider	September 30, 2021		
Amazon Cognito Sync	September 30, 2021		
Amazon Comprehend	September 30, 2021	February 17, 2023	
Amazon Comprehend Medical	September 30, 2021		
Amazon Connect Contact Lens	September 30, 2021		
Amazon Connect Participant Service	September 30, 2021		
Amazon Connect	September 30, 2021	February 17, 2023	
Amazon Connect Voice ID	April 19, 2022		
Amazon Connect Wisdom	April 19, 2022		
Amazon Data Lifecycle Manager	September 30, 2021		
Amazon Detective	September 30, 2021		
Amazon DevOps Guru	September 30, 2021	July 26, 2022	

Service	Initial support	Updated	
Amazon DocumentDB (with MongoDB compatibility)	September 30, 2021		
Amazon DynamoDB	September 30, 2021	February 17, 2023	
Amazon DynamoDB Streams	September 30, 2021		
Amazon EC2 Container Registry	September 30, 2021		
Amazon EC2 Container Service	September 30, 2021	February 17, 2023	
Amazon EC2 Systems Manager	September 30, 2021	February 17, 2023	
Amazon EMR	September 30, 2021	February 17, 2023	
Amazon ElastiCache	September 30, 2021		
Amazon Elastic Inference	September 30, 2021		
Amazon Elastic Block Store	September 30, 2021		
Amazon Elastic Compute Cloud	September 30, 2021	February 17, 2023	
Amazon Elastic Container Registry Public	September 30, 2021		
Amazon Elastic File System	September 30, 2021		

Service	Initial support	Updated	
Amazon Elastic Kubernetes Service	September 30, 2021	February 17, 2023	
Amazon EMR	September 30, 2021		
Amazon Elastic Transcoder	September 30, 2021		
Amazon OpenSearch Service	September 30, 2021	February 17, 2023	
Amazon OpenSearch Service	April 19, 2022	February 17, 2023	
Amazon EventBridge	September 30, 2021	February 17, 2023	
Amazon FSx	September 30, 2021	February 17, 2023	
Amazon Forecast Query	September 30, 2021	February 17, 2023	
Amazon Forecast Service	September 30, 2021	February 17, 2023	
Amazon Fraud Detector	September 30, 2021		
Amazon GameLift	September 30, 2021	February 17, 2023	
Amazon GameSparks	July 26, 2022		
Amazon S3 Glacier	September 30, 2021		
Amazon GuardDuty	September 30, 2021		
AWS HealthLake	September 30, 2021		
Amazon Honeycode	September 30, 2021		

Service	Initial support	Updated	
Amazon Inspector	September 30, 2021		
Amazon Inspector V2	April 19, 2022		
Amazon Interactive Video Service	September 30, 2021		
Amazon Kendra	September 30, 2021		
Amazon Kinesis	September 30, 2021		
Amazon Kinesis Analytics	September 30, 2021		
Amazon Kinesis Analytics V2	September 30, 2021		
Amazon Kinesis Firehose	September 30, 2021		
Amazon Kinesis Video Signaling Channels	September 30, 2021		
Amazon Kinesis Video Streams	September 30, 2021	February 17, 2023	
Amazon Kinesis Video Streams Archived Media	September 30, 2021		
Amazon Kinesis video stream	September 30, 2021		
Amazon Lex Model Building Service	September 30, 2021		
Amazon Lex Model Building Service V2	September 30, 2021	February 17, 2023	

Service	Initial support	Updated	
Amazon Lex	September 30, 2021		
Amazon Lex Runtime V2	September 30, 2021		
Amazon Lightsail	September 30, 2021	February 17, 2023	
Amazon Location Service	September 30, 2021	February 17, 2023	
Amazon Lookout for Equipment	September 30, 2021		
Amazon Lookout for Metrics	September 30, 2021	February 17, 2023	
Amazon Lookout for Vision	September 30, 2021		
Amazon MQ	September 30, 2021		
Amazon Macie	September 30, 2021		
Amazon Macie 2	September 30, 2021	February 17, 2023	
Amazon Managed Blockchain	September 30, 2021	February 17, 2023	
Amazon Managed Grafana	April 19, 2022	February 17, 2023	
Amazon Managed Service for Prometheus	September 30, 2021	February 17, 2023	
Amazon Managed Streaming for Apache Kafka	September 30, 2021	February 17, 2023	

Service	Initial support	Updated	
Amazon Managed Streaming for Apache Kafka	April 19, 2022		
Amazon Managed Workflows for Apache Airflow	September 30, 2021		
Amazon Mechanical Turk	September 30, 2021		
Amazon MemoryDB for Redis	April 19, 2022	February 17, 2023	
Amazon Nimble Studio	September 30, 2021		
Amazon Personalize	September 30, 2021	February 17, 2023	
Amazon Personalize Events	September 30, 2021		
Amazon Personalize Runtime	September 30, 2021		
Amazon Pinpoint	September 30, 2021		
Amazon Pinpoint Email Service	September 30, 2021		
Amazon Pinpoint SMS and Voice Service	September 30, 2021		
Amazon Pinpoint SMS and Voice V2 Service	July 26, 2022		

Service	Initial support	Updated	
Amazon Polly	September 30, 2021		
Amazon QLDB	September 30, 2021		
Amazon QLDB Session	September 30, 2021		
Amazon QuickSight	September 30, 2021	February 17, 2023	
Amazon Redshift	September 30, 2021		
Amazon Redshift Data API	September 30, 2021		
Amazon Rekognition	September 30, 2021	February 17, 2023	
Amazon Relational Database Service	September 30, 2021	February 17, 2023	
Amazon Route 53	September 30, 2021		
Amazon Route 53 Recovery Control Config	September 30, 2021	July 26, 2022	
Amazon Route 53 Domains	September 30, 2021	February 17, 2023	
Amazon Route 53 Resolver	September 30, 2021		
Amazon S3 on Outposts	September 30, 2021	July 26, 2022	
Amazon SageMaker Runtime Feature Store Runtime	September 30, 2021		

Service	Initial support	Updated	
Amazon SageMaker Runtime Runtime	September 30, 2021		
Amazon SageMaker	September 30, 2021	February 17, 2023	
Amazon SageMaker Edge Manager	September 30, 2021		
Amazon Simple Email Service	September 30, 2021		
Amazon Simple Email Service V2	September 30, 2021	February 17, 2023	
Amazon Simple Notification Service	September 30, 2021	February 17, 2023	
Amazon Simple Queue Service	September 30, 2021	February 17, 2023	
Amazon Simple Storage Service	September 30, 2021	February 17, 2023	
Amazon Simple Workflow Service	September 30, 2021		
Amazon Textract	September 30, 2021	February 17, 2023	
Amazon Transcribe	September 30, 2021		
Amazon Translate	September 30, 2021	February 17, 2023	
Amazon WorkDocs	September 30, 2021	February 17, 2023	
Amazon WorkMail	September 30, 2021	February 17, 2023	
Amazon WorkMail Message Flow	September 30, 2021		

Service	Initial support	Updated	
Amazon WorkSpaces	September 30, 2021	February 17, 2023	
Amazon WorkSpaces Web	April 19, 2022	February 17, 2023	
Amplify	September 30, 2021		
Amplify UI Builder	April 19, 2022	February 17, 2023	
Application Auto Scaling	September 30, 2021		
Amazon EC2 Auto Scaling	September 30, 2021	February 17, 2023	
CodeArtifact	September 30, 2021		
DynamoDB Accelerator	September 30, 2021		
EC2 Image Builder	September 30, 2021		
AWS Elastic Disaster Recovery	April 19, 2022	February 17, 2023	
Elastic Load Balancing	September 30, 2021		
Elastic Load Balancing V2	September 30, 2021		
MediaConnect	September 30, 2021		
Amazon S3 Control	September 30, 2021	February 17, 2023	
Recycle Bin for Amazon EBS	April 19, 2022	February 17, 2023	
Savings Plans	September 30, 2021		

Service	Initial support	Updated	
Amazon EventBridge Schema Registry	September 30, 2021		
Service Quotas	September 30, 2021		
AWS Snowball	September 30, 2021		

Starter templates for Step Functions

In the [AWS Step Functions console](#), you can choose one of the following starter templates to deploy state machines to your AWS accounts. These starter templates are ready-to-run sample projects that automatically create the workflow proptotype and definition, and all related AWS resources for the project.

You can use these sample projects to deploy and run them as is, or use the workflow prototypes to build on them. If you build upon these projects, Step Functions creates the workflow prototype, but doesn't deploy the resources listed in the workflow definition.

When you deploy the sample projects, they provision a fully functional state machine, and create the related resources for the state machine to run. When you create a sample project, Step Functions uses AWS CloudFormation to create the related resources referenced by the state machine.

Topics

- [Manage a batch job \(AWS Batch, Amazon SNS\)](#)
- [Manage a container task \(Amazon ECS, Amazon SNS\)](#)
- [Transfer data records \(Lambda, DynamoDB, Amazon SQS\)](#)
- [Poll for Job Status \(Lambda, AWS Batch\)](#)
- [Task Timer \(Lambda, Amazon SNS\)](#)
- [Callback Pattern Example \(Amazon SQS, Amazon SNS, Lambda\)](#)
- [Manage an Amazon EMR Job](#)
- [Run an EMR Serverless job](#)
- [Start a Workflow within a Workflow \(Step Functions, Lambda\)](#)
- [Dynamically process data with a Map state](#)
- [Process a CSV file with Distributed Map](#)
- [Process data in an Amazon S3 bucket with Distributed Map](#)
- [Train a Machine Learning Model](#)
- [Tune a Machine Learning Model](#)
- [Process High-Volume Messages from Amazon SQS \(Express Workflows\)](#)

- [Selective Checkpointing Example \(Express Workflows\)](#)
- [Build an AWS CodeBuild Project \(CodeBuild, Amazon SNS\)](#)
- [Preprocess data and train a machine learning model](#)
- [Lambda orchestration example](#)
- [Start an Athena query](#)
- [Execute multiple queries \(Amazon Athena, Amazon SNS\)](#)
- [Query large datasets \(Amazon Athena, Amazon S3, AWS Glue, Amazon SNS\)](#)
- [Keep data up to date \(Amazon Athena, Amazon S3, AWS Glue\)](#)
- [Manage an Amazon EKS cluster](#)
- [Make a call to API Gateway](#)
- [Call a microservice running on Fargate using API Gateway integration](#)
- [Send a custom event to EventBridge](#)
- [Invoke Synchronous Express Workflows](#)
- [Run ETL/ELT workflows using Amazon Redshift \(Lambda, Amazon Redshift Data API\)](#)
- [Use Step Functions and AWS Batch with error handling](#)
- [Fan out an AWS Batch job](#)
- [AWS Batch with Lambda](#)
- [Perform AI prompt-chaining with Amazon Bedrock](#)

Manage a batch job (AWS Batch, Amazon SNS)

This sample project demonstrates how to submit an AWS Batch job, and then send an Amazon SNS notification based on whether that job succeeds or fails. Deploying this sample project creates an AWS Step Functions state machine, an AWS Batch job, and an Amazon SNS topic.

In this project, Step Functions uses a state machine to call the AWS Batch job synchronously. It then waits for the job to succeed or fail, and it sends an Amazon SNS topic with a message about whether the job succeeded or failed.

Step 1: Create the state machine and provision resources

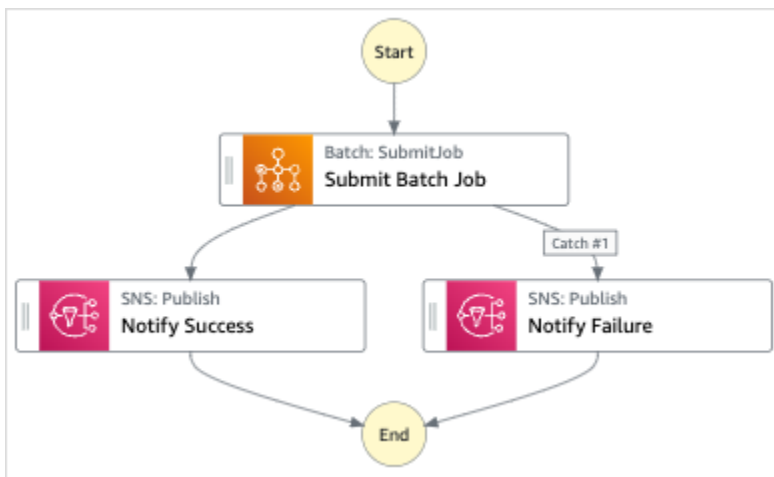
1. Open the [Step Functions console](#) and choose **Create state machine**.

2. Type **Manage a batch job** in the search box, and then choose **Manage a batch job** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- A AWS Batch job
- An Amazon SNS topic
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Manage a batch job** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#)

definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:

1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with AWS Batch and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls AWS Batch and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the `Resource` field, and by passing `Parameters` to the service API.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of the Amazon States Language for notification on an AWS Batch
job completion",
  "StartAt": "Submit Batch Job",
  "TimeoutSeconds": 3600,
  "States": {
    "Submit Batch Job": {
      "Type": "Task",
      "Resource": "arn:aws:states:::batch:submitJob.sync",
      "Parameters": {
        "JobName": "BatchJobNotification",
        "JobQueue": "arn:aws:batch:us-east-1:123456789012:job-queue/
BatchJobQueue-7049d367474b4dd",
        "JobDefinition": "arn:aws:batch:us-east-1:123456789012:job-definition/
BatchJobDefinition-74d55ec34c4643c:1"
      },
      "Next": "Notify Success",
      "Catch": [
        {
          "ErrorEquals": [ "States.ALL" ],
          "Next": "Notify Failure"
        }
      ]
    },
    "Notify Success": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "Message": "Batch job submitted through Step Functions succeeded",
        "TopicArn": "arn:aws:sns:us-east-1:123456789012:batchjobnotificatiointemplate-
SNSTopic-1J757CVBQ2KHM"
      },
      "End": true
    },
    "Notify Failure": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "Message": "Batch job submitted through Step Functions failed",
```

```
    "TopicArn": "arn:aws:sns:us-east-1:123456789012:batchjobnotificationtemplate-
SNSTopic-1J757CVBQ2KHM"
  },
  "End": true
}
}
```

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "sns:Publish"
      ],
      "Resource": [
        "arn:aws:sns:ap-northeast-1:123456789012:ManageBatchJob-SNSTopic-
JHLYYG7AZPZI"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "batch:SubmitJob",
        "batch:DescribeJobs",
        "batch:TerminateJob"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
```

```
        "arn:aws:events:ap-northeast-1:123456789012:rule/
StepFunctionsGetEventsForBatchJobsRule"
    ],
    "Effect": "Allow"
  }
]
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Manage a container task (Amazon ECS, Amazon SNS)

This sample project demonstrates how to run an AWS Fargate task, and then send an Amazon SNS notification based on whether that job succeeds or fails. Deploying this sample project will create an AWS Step Functions state machine, a Fargate cluster, and an Amazon SNS topic.

In this project, Step Functions uses a state machine to call the Fargate task synchronously. It then waits for the task to succeed or fail, and it sends an Amazon SNS topic with a message about whether the job succeeded or failed.

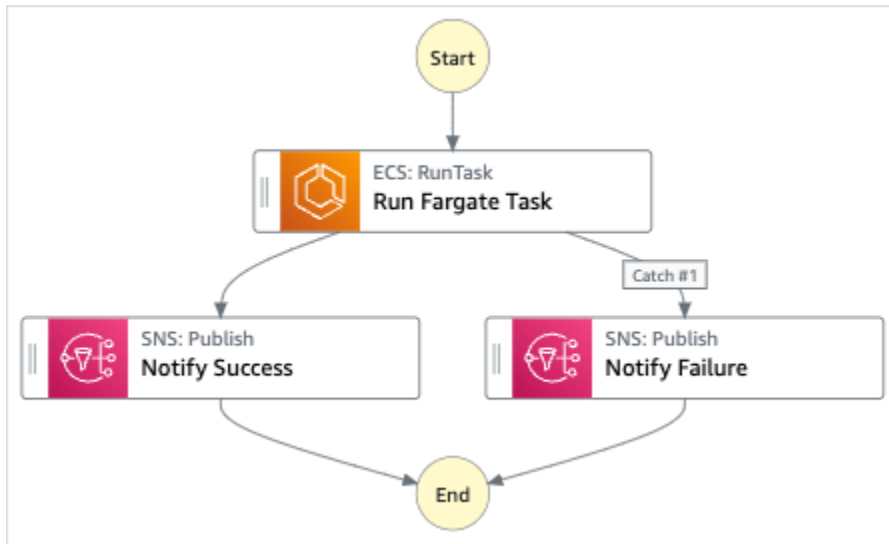
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Manage a container task** in the search box, and then choose **Manage a container task** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An AWS Fargate cluster
- An Amazon SNS topic
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Manage a container task** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

 **Note**

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with AWS Fargate and Amazon SNS by passing parameters directly to those resources. Browse through this example state machine to see how Step Functions uses a state machine to call the Fargate task synchronously, waits for the task to succeed or fail, and sends an Amazon SNS topic with a message about whether the job succeeded or failed.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of the Amazon States Language for notification on an AWS
Fargate task completion",
  "StartAt": "Run Fargate Task",
  "TimeoutSeconds": 3600,
  "States": {
    "Run Fargate Task": {
      "Type": "Task",
      "Resource": "arn:aws:states:::ecs:runTask.sync",
      "Parameters": {
        "LaunchType": "FARGATE",
```

```

    "Cluster": "arn:aws:ecs:ap-northeast-1:123456789012:cluster/
FargateTaskNotification-ECSCluster-VHLR20IF9IMP",
    "TaskDefinition": "arn:aws:ecs:ap-northeast-1:123456789012:task-definition/
FargateTaskNotification-ECSTaskDefinition-13Y0JT8Z2LY5Q:1",
    "NetworkConfiguration": {
      "AwsVpcConfiguration": {
        "Subnets": [
          "subnet-07e1ad3abcfce6758",
          "subnet-04782e7f34ae3efdb"
        ],
        "AssignPublicIp": "ENABLED"
      }
    },
    "Next": "Notify Success",
    "Catch": [
      {
        "ErrorEquals": [ "States.ALL" ],
        "Next": "Notify Failure"
      }
    ],
    "Notify Success": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "Message": "AWS Fargate Task started by Step Functions succeeded",
        "TopicArn": "arn:aws:sns:ap-northeast-1:123456789012:FargateTaskNotification-
SNSTopic-1XYW5YD5V0M7C"
      },
      "End": true
    },
    "Notify Failure": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "Message": "AWS Fargate Task started by Step Functions failed",
        "TopicArn": "arn:aws:sns:ap-northeast-1:123456789012:FargateTaskNotification-
SNSTopic-1XYW5YD5V0M7C"
      },
      "End": true
    }
  }
}

```

```
}
```

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. It's a best practice to include only those permissions that are necessary in your IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "sns:Publish"
      ],
      "Resource": [
        "arn:aws:sns:ap-northeast-1:123456789012:FargateTaskNotification-
        SNSTopic-1XYW5YD5V0M7C"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "ecs:RunTask"
      ],
      "Resource": [
        "arn:aws:ecs:ap-northeast-1:123456789012:task-definition/
        FargateTaskNotification-ECSTaskDefinition-13Y0JT8Z2LY5Q:1"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "ecs:StopTask",
        "ecs:DescribeTasks"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "events:PutTargets",
        "events:PutRule",

```

```
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:ap-northeast-1:123456789012:rule/
StepFunctionsGetEventsForECSTaskRule"
      ],
      "Effect": "Allow"
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Transfer data records (Lambda, DynamoDB, Amazon SQS)

This sample project demonstrates how to iteratively read items from an Amazon DynamoDB table and send these items to an Amazon SQS queue using a Step Functions state machine. Deploying this sample project will create a Step Functions state machine, a DynamoDB table, an AWS Lambda function, and an Amazon SQS queue.

In this project, Step Functions uses the Lambda function to populate the DynamoDB table. The state machine also uses a `for` loop to read each of the entries, and then sends each entry to an Amazon SQS queue.

Step 1: Create the state machine and provision resources

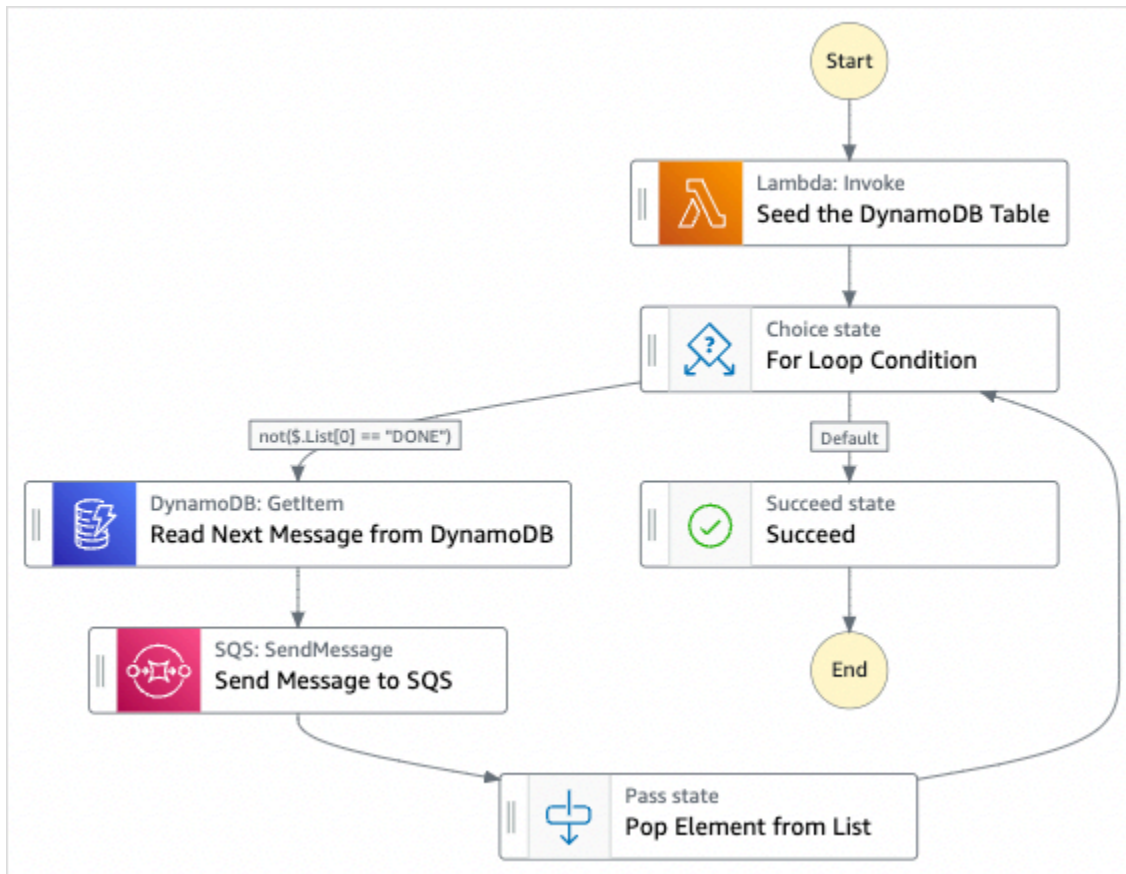
1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Transfer data records** in the search box, and then choose **Transfer data records** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- A Lambda function for seeding the DynamoDB table

- An Amazon SQS queue
- A DynamoDB table
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Transfer data records** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#)

definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:

1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with DynamoDB and Amazon SQS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls DynamoDB and Amazon SQS by connecting to the Amazon Resource Name (ARN) in the `Resource` field, and by passing `Parameters` to the service API.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment" : "An example of the Amazon States Language for reading messages from a
DynamoDB table and sending them to SQS",
  "StartAt": "Seed the DynamoDB Table",
  "TimeoutSeconds": 3600,
  "States": {
    "Seed the DynamoDB Table": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:sqsconnector-
SeedingFunction-T3U43VYDU50Q",
      "ResultPath": "$.List",
      "Next": "For Loop Condition"
    },
    "For Loop Condition": {
      "Type": "Choice",
      "Choices": [
        {
          "Not": {
            "Variable": "$.List[0]",
            "StringEquals": "DONE"
          },
          "Next": "Read Next Message from DynamoDB"
        }
      ],
      "Default": "Succeed"
    },
    "Read Next Message from DynamoDB": {
      "Type": "Task",
      "Resource": "arn:aws:states:::dynamodb:getItem",
      "Parameters": {
        "TableName": "sqsconnector-DDBTable-1CAFOJWP8QD6I",
        "Key": {
          "MessageId": {"S.$": "$.List[0]"}
        }
      },
      "ResultPath": "$.DynamoDB",
      "Next": "Send Message to SQS"
    },
    "Send Message to SQS": {
      "Type": "Task",
```



```

    "Resource": "arn:aws:states:::sqs:sendMessage",
    "Parameters": {
      "MessageBody.$": "$.DynamoDB.Item.Message.S",
      "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/sqsconnector-
SQSQueue-QVGQBW134PWK"
    },
    "ResultPath": "$.SQS",
    "Next": "Pop Element from List"
  },
  "Pop Element from List": {
    "Type": "Pass",
    "Parameters": {
      "List.$": "$.List[1:]"
    },
    "Next": "For Loop Condition"
  },
  "Succeed": {
    "Type": "Succeed"
  }
}
}

```

For more information about passing parameters and managing results, see the following:

- [Pass parameters to a service API](#)
- [ResultPath](#)

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. It's a best practice to include only those permissions that are necessary in your IAM policies.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:GetItem"
      ],
      "Resource": [

```

```

        "arn:aws:dynamodb:ap-northeast-1:123456789012:table/
TransferDataRecords-DDBTable-3I41R5L5EAGT"
    ],
    "Effect": "Allow"
  },
  {
    "Action": [
      "sqs:SendMessage"
    ],
    "Resource": [
      "arn:aws:sqs:ap-northeast-1:123456789012:TransferDataRecords-SQSQueue-
BKWXTS09LIW1"
    ],
    "Effect": "Allow"
  },
  {
    "Action": [
      "lambda:invokeFunction"
    ],
    "Resource": [
      "arn:aws:lambda:ap-
northeast-1:123456789012:function:TransferDataRecords-SeedingFunction-VN4KY2TPAZSR"
    ],
    "Effect": "Allow"
  }
]
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Poll for Job Status (Lambda, AWS Batch)

This sample project creates an AWS Batch job poller. It implements an AWS Step Functions state machine that uses AWS Lambda to create a Wait state loop that checks on an AWS Batch job.

This sample project creates and configures all resources so that your Step Functions workflow will submit an AWS Batch job, and will wait for that job to complete before ending successfully.

Note

You can also implement this pattern without using a Lambda function. For information about controlling AWS Batch directly, see [Using AWS Step Functions with other services](#).

This sample project creates the state machine, two Lambda functions, and an AWS Batch queue, and configures the related IAM permissions.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

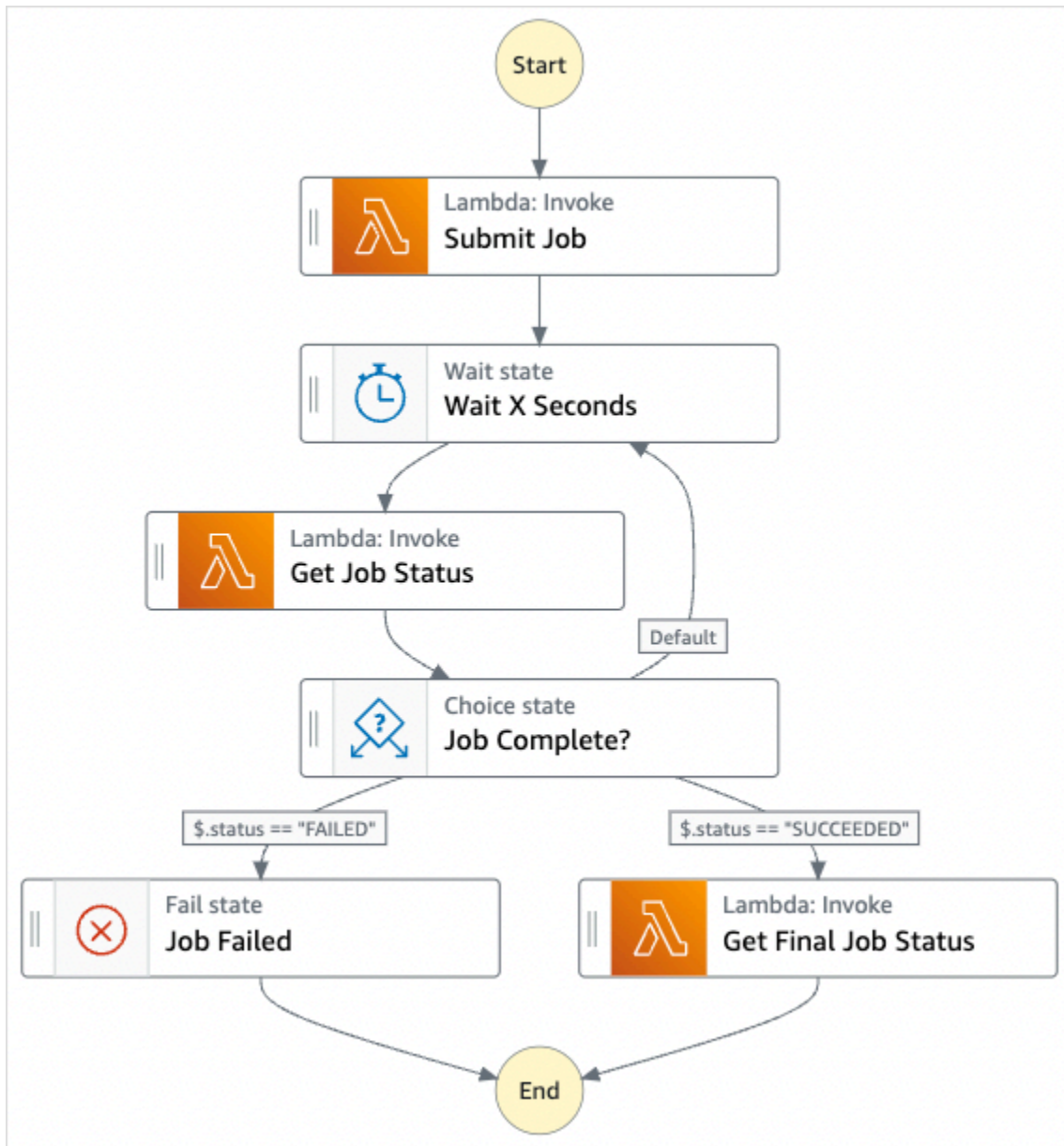
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Job Poller** in the search box, and then choose **Job Poller** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- Three Lambda functions to submit an AWS Batch job, get the current status of the submitted AWS Batch job, and the final job completion status.
- An AWS Batch job
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Job Poller** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#)

definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

After all the resources are provisioned and deployed, the **Start execution** dialog box is displayed with example input similar to the following.

```
{  
  "jobName": "my-job",
```

```
"jobDefinition": "arn:aws:batch:us-east-2:123456789012:job-definition/
SampleJobDefinition-343f54b445d5312:1",
  "jobQueue": "arn:aws:batch:us-east-2:123456789012:job-queue/
SampleJobQueue-4d9d696031e1449",
  "wait_time": 60
}
```

Note

`wait_time` instructs the `Wait` state to loop every 60 seconds.

- In the **Start execution** dialog box, do the following:

1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.

4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

For example, to view the changing status of your AWS Batch job and the looping results of your execution, choose the **Output** tab.

The following image shows the execution status graph in the **Graph view**. It also shows the execution output for the selected step in the **Output** tab.

The screenshot displays the AWS Step Functions console interface. On the left, the **Graph view** shows a state machine flowchart. The flow starts at a yellow circle labeled 'Start', followed by a green rectangle 'Submit Job' with a checkmark. This leads to another green rectangle 'Wait X Seconds' with a checkmark. From there, it goes to a green rectangle 'Get Job Status' with a checkmark, then to a green rectangle 'Job Complete?' with a checkmark. From 'Job Complete?', there are two paths: one leading to a dashed box labeled 'Job Failed' and another leading to a green rectangle 'Get Final Job Status' with a checkmark. Both paths converge at a yellow circle labeled 'End'. A legend at the bottom of the graph view identifies status icons: In progress (blue circle with dot), Failed (red square with X), Caught error (yellow triangle with exclamation mark), Canceled (gray circle with minus), and Succeeded (green circle with checkmark).

On the right, the **Get Final Job Status** state details are shown. It includes links for 'Lambda' and 'Log group'. Below this, there are tabs for 'Input', 'Output', 'Details', 'Definition', and 'Events'. The 'Output' tab is selected, showing a table with one row: '1' followed by the text '"SUCCEEDED"' in red. An 'Advanced view' toggle is also present.

Example State Machine Code

The state machine in this sample project integrates with AWS Lambda to submit an AWS Batch job. Browse through this example state machine to see how Step Functions controls Lambda and AWS Batch.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of the Amazon States Language that runs an AWS Batch job and
monitors the job until it completes.",
  "StartAt": "Submit Job",
  "States": {
    "Submit Job": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-
east-1:111122223333:function:StepFunctionsSample-JobStatusPol-SubmitJobFunction-
jDaYcl4cx55r",
      "ResultPath": "$.guid",
      "Next": "Wait X Seconds"
    },
    "Wait X Seconds": {
      "Type": "Wait",
      "SecondsPath": "$.wait_time",
      "Next": "Get Job Status"
    },
    "Get Job Status": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-
east-1:111122223333:function:StepFunctionsSample-JobStatusPoll-
CheckJobFunction-1JkJwY10vonI",
      "Next": "Job Complete?",
      "InputPath": "$.guid",
      "ResultPath": "$.status"
    },
    "Job Complete?": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.status",
          "StringEquals": "FAILED",
          "Next": "Job Failed"
        },
        {
          "Variable": "$.status",
          "StringEquals": "SUCCEEDED",
          "Next": "Get Final Job Status"
        }
      ],
      "Default": "Wait X Seconds"
    },
  },
}
```



```
"Job Failed": {
  "Type": "Fail",
  "Cause": "AWS Batch Job Failed",
  "Error": "DescribeJob returned FAILED"
},
"Get Final Job Status": {
  "Type": "Task",
  "Resource": "arn:aws::lambda:us-
east-1:111122223333:function:StepFunctionsSample-JobStatusPoll-
CheckJobFunction-1JkJwY10vonI",
  "InputPath": "$.guid",
  "End": true
}
}
```

Task Timer (Lambda, Amazon SNS)

This sample project creates a task timer. It implements an AWS Step Functions state machine that implements a `Wait` state, and uses an AWS Lambda function that sends an Amazon Simple Notification Service (Amazon SNS) notification. A [Wait](#) state is a state type that waits for a trigger to perform a single unit of work.

Note

This sample project implements an AWS Lambda function to send an Amazon Simple Notification Service (Amazon SNS) notification. You can also send an Amazon SNS notification directly from the Amazon States Language. See [Using AWS Step Functions with other services](#).

This sample project creates the state machine, a Lambda function, and an Amazon SNS topic, and configures the related AWS Identity and Access Management (IAM) permissions. For more information about the resources that are created with the **Task Timer** sample project, see the following:

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

- [AWS CloudFormation User Guide](#)

- [Amazon Simple Notification Service Developer Guide](#)
- [AWS Lambda Developer Guide](#)
- [IAM Getting Started Guide](#)

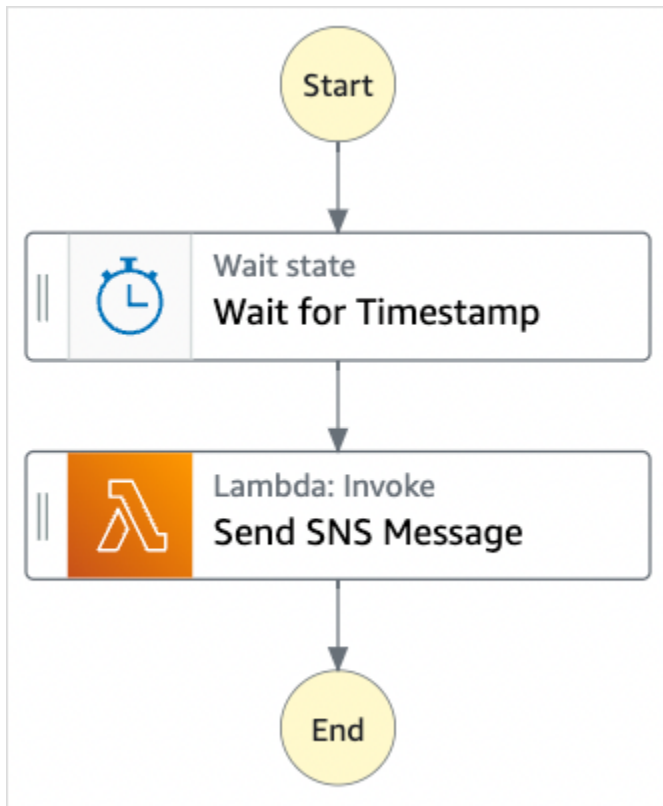
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Task Timer** in the search box, and then choose **Task Timer** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- a Lambda function that sends an Amazon SNS notification.
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Task Timer** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

After all the resources are provisioned and deployed, the **Start execution** dialog box is displayed with example input similar to the following.

```
{
  "jobName": "my-job", {
  "topic": "arn:aws:sns:us-east-2:123456789012:StepFunctionsSample-TaskTimercc68840e-
c3d3-42a8-911e-821b7ce248e5-SNSTopic-44UjcFxzhACT",
  "message": "HelloWorld",
  "timer_seconds": 10
}
```

- In the **Start execution** dialog box, do the following:

1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

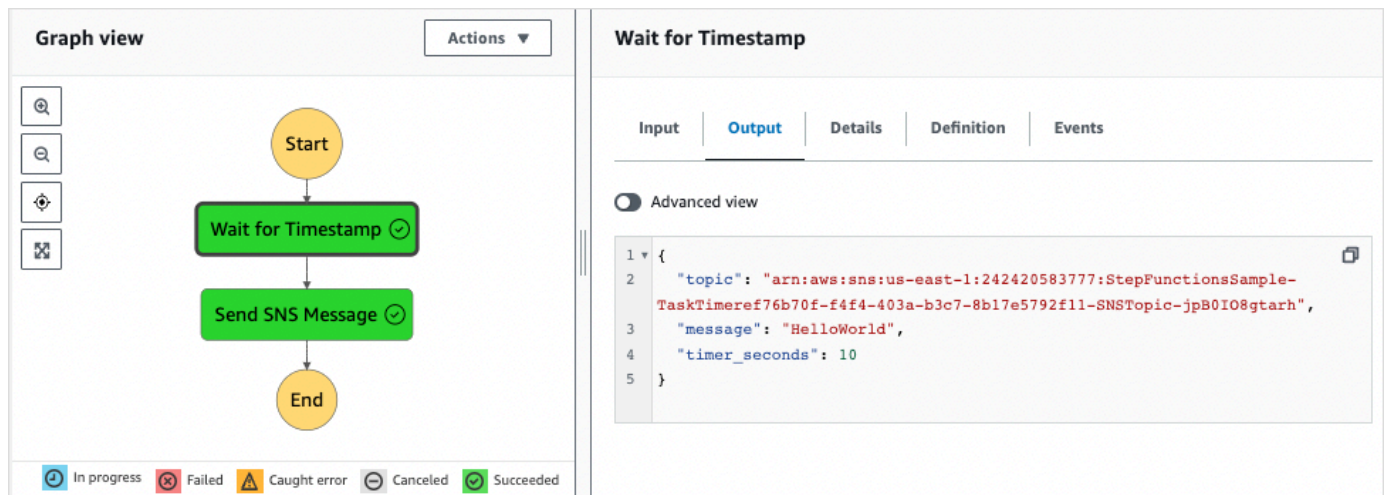
Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

For example, the following image shows the output of the selected step **Wait for Timestamp**. The output of this step is passed as input to the **Send SNS Message** step.



Callback Pattern Example (Amazon SQS, Amazon SNS, Lambda)

This sample project demonstrates how to have AWS Step Functions pause during a task, and wait for an external process to return a task token that was generated when the task started.

When this sample project is deployed and an execution is started, the following steps occur:

1. Step Functions passes a message that includes a task token to an Amazon Simple Queue Service (Amazon SQS) queue.
2. Step Functions then pauses, waiting for that token to be returned.
3. The Amazon SQS queue triggers an AWS Lambda function that calls [SendTaskSuccess](#) with that same task token.
4. When the task token is received, the workflow continues.
5. The "Notify Success" task publishes an Amazon Simple Notification Service (Amazon SNS) message that the callback was received.

To learn how to implement the callback pattern in Step Functions, see [Wait for a Callback with the Task Token](#).

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

Step 1: Create the state machine and provision resources

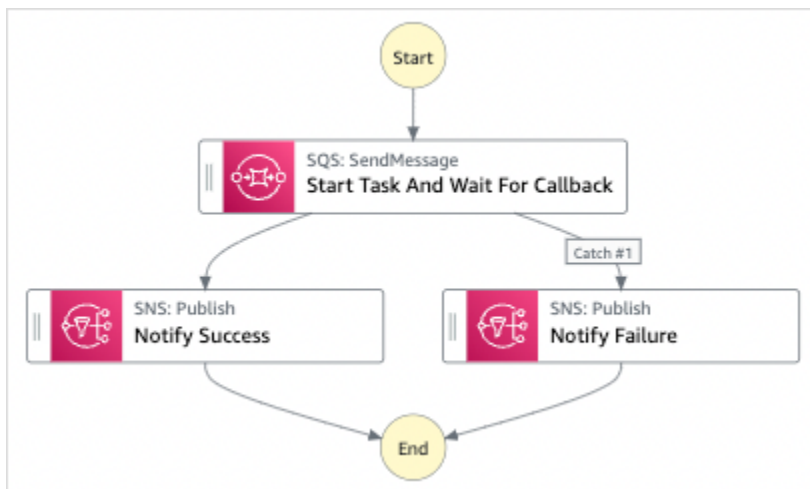
1. Open the [Step Functions console](#) and choose **Create state machine**.

2. Type **Callback pattern example** in the search box, and then choose **Callback pattern example** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon SQS message queue.
- A Lambda function that calls the Step Functions API action [SendTaskSuccess](#).
- An Amazon SNS topic to notify about the success or failure of a task indicating whether or not the workflow can continue.
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Callback pattern example** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.

2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

For example, to review how Step Functions progressed through the workflow and received a callback from Amazon SQS, review the entries in the **Events** table. The following image shows the execution output for the **Notify Success** step. It also shows the first five events from the execution event history. Expand each event to view more details about that event.

Graph view

```

graph TD
    Start((Start)) --> Task[Start Task And Wait For Callback]
    Task --> NotifySuccess[Notify Success]
    Task --> NotifyFailure[Notify Failure]
    NotifySuccess --> End((End))
    NotifyFailure --> End
  
```

Legend: ● In progress ⊗ Failed ⚠ Caught error ⊖ Canceled ✔ Succeeded

Notify Success

sns:publish

Input | **Output** | Details | Definition | Events

Advanced view

```

1 {
2   "MessageId": "f86995a8-9531-5391-ab76-c8f43e6c3bf1",
3   "SdkHttpMetadata": {
4     "AllHttpHeaders": {
5       "x-amzn-RequestId": [
6         "e3307ad6-f75d-526d-908a-278a5c007a0d"
7       ],
8     },
9     "Content-Length": [
10      "294"
11    ]
12  }
  
```

Events (13)

Filter by properties or search by keyword Filter by a date and time range

ID	Type	Step	Resource	Started After	Timestamp
▶ 1	ExecutionStarted			0	Aug 20, 2023, 17:00:27.681 (UTC-07:00)
▶ 2	TaskStateEntered	Start Task And Wait For Callback		00:00:00.031	Aug 20, 2023, 17:00:27.712 (UTC-07:00)
▶ 3	TaskScheduled	Start Task And Wait For Callback	sqs:sendMessage	00:00:00.031	Aug 20, 2023, 17:00:27.712 (UTC-07:00)
▶ 4	TaskStarted	Start Task And Wait For Callback	sqs:sendMessage	00:00:00.116	Aug 20, 2023, 17:00:27.797 (UTC-07:00)
▶ 5	TaskSubmitted	Start Task And Wait For Callback	sqs:sendMessage	00:00:00.208	Aug 20, 2023, 17:00:27.889 (UTC-07:00)

Lambda Callback Example

To see how the components of this sample project work together, see the resources that were deployed in your AWS account. For example, here is the Lambda function that calls Step Functions with the task token.

```

console.log('Loading function');
const aws = require('aws-sdk');

exports.lambda_handler = (event, context, callback) => {
  const stepfunctions = new aws.StepFunctions();

  for (const record of event.Records) {
    const messageBody = JSON.parse(record.body);
    const taskToken = messageBody.TaskToken;

    const params = {
      output: "\"Callback task completed successfully.\\"",
      taskToken: taskToken
    };
  }
};
  
```

```
    console.log(`Calling Step Functions to complete callback task with params
${JSON.stringify(params)}`);

    stepfunctions.sendTaskSuccess(params, (err, data) => {
        if (err) {
            console.error(err.message);
            callback(err.message);
            return;
        }
        console.log(data);
        callback(null);
    });
}
```

Manage an Amazon EMR Job

This sample project demonstrates Amazon EMR and AWS Step Functions integration.

It shows how to create an Amazon EMR cluster, add multiple steps and run them, and then terminate the cluster.

Important

Amazon EMR does not have a free pricing tier. Running the sample project will incur costs. You can find pricing information on the [Amazon EMR pricing](#) page. The availability of Amazon EMR service integration is subject to the availability of Amazon EMR APIs. Because of this, this sample project might not work correctly in some AWS Regions. See the [Amazon EMR](#) documentation for limitations in special Regions.

Step 1: Create the State Machine and Provision Resources

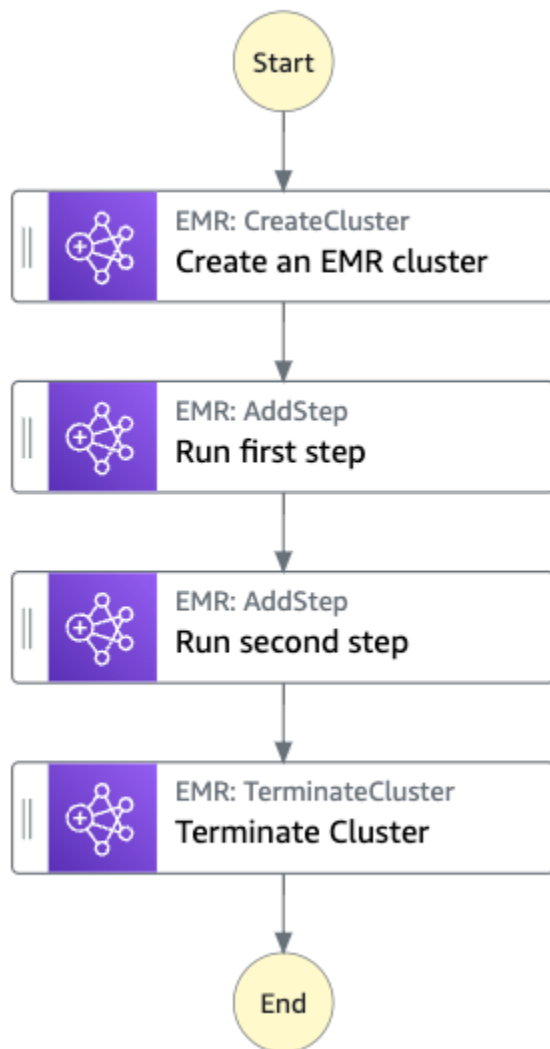
1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Manage an EMR job** in the search box, and then choose **Manage an EMR job** from the search results that are returned.
3. Choose **Next** to continue.

- Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon S3 bucket
- An Amazon EMR cluster
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Manage an EMR job** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

ℹ Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

ℹ Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including

input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with Amazon EMR by passing parameters directly to those resources. Browse through this example state machine to see how Step Functions uses a state machine to call the Amazon EMR task synchronously, waits for the task to succeed or fail, and terminates the cluster.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of the Amazon States Language for running jobs on Amazon EMR",
  "StartAt": "Create an EMR cluster",
  "States": {
    "Create an EMR cluster": {
      "Type": "Task",
      "Resource": "arn:<PARTITION>:states:::elasticmapreduce:createCluster.sync",
      "Parameters": {
        "Name": "ExampleCluster",
        "VisibleToAllUsers": true,
        "ReleaseLabel": "emr-5.26.0",
        "Applications": [
          { "Name": "Hive" }
        ],
        "ServiceRole": "<EMR_SERVICE_ROLE>",
        "JobFlowRole": "<EMR_EC2_INSTANCE_PROFILE>",
        "LogUri": "s3://<EMR_LOG_S3_BUCKET>/logs/",
        "Instances": {
          "KeepJobFlowAliveWhenNoSteps": true,
          "InstanceFleets": [
            {
              "Name": "MyMasterFleet",
              "InstanceFleetType": "MASTER",
              "TargetOnDemandCapacity": 1,
              "InstanceTypeConfigs": [
                {
                  "InstanceType": "m5.xlarge"
                }
              ]
            }
          ]
        }
      }
    }
  }
}
```

```

    },
    {
      "Name": "MyCoreFleet",
      "InstanceFleetType": "CORE",
      "TargetOnDemandCapacity": 1,
      "InstanceTypeConfigs": [
        {
          "InstanceType": "m5.xlarge"
        }
      ]
    }
  ]
}
},
"ResultPath": "$.cluster",
"Next": "Run first step"
},
"Run first step": {
  "Type": "Task",
  "Resource": "arn:<PARTITION>:states:::elasticmapreduce:addStep.sync",
  "Parameters": {
    "ClusterId.$": "$.cluster.ClusterId",
    "Step": {
      "Name": "My first EMR step",
      "ActionOnFailure": "CONTINUE",
      "HadoopJarStep": {
        "Jar": "command-runner.jar",
        "Args": ["<COMMAND_ARGUMENTS>"]
      }
    }
  }
},
"Retry" : [
  {
    "ErrorEquals": [ "States.ALL" ],
    "IntervalSeconds": 1,
    "MaxAttempts": 3,
    "BackoffRate": 2.0
  }
],
"ResultPath": "$.firstStep",
"Next": "Run second step"
},
"Run second step": {
  "Type": "Task",

```



```

"Resource": "arn:<PARTITION>:states:::elasticmapreduce:addStep.sync",
"Parameters": {
  "ClusterId.$": "$.cluster.ClusterId",
  "Step": {
    "Name": "My second EMR step",
    "ActionOnFailure": "CONTINUE",
    "HadoopJarStep": {
      "Jar": "command-runner.jar",
      "Args": ["<COMMAND_ARGUMENTS>"]
    }
  }
},
"Retry" : [
  {
    "ErrorEquals": [ "States.ALL" ],
    "IntervalSeconds": 1,
    "MaxAttempts": 3,
    "BackoffRate": 2.0
  }
],
"ResultPath": "$.secondStep",
"Next": "Terminate Cluster"
},
"Terminate Cluster": {
  "Type": "Task",
  "Resource": "arn:<PARTITION>:states:::elasticmapreduce:terminateCluster",
  "Parameters": {
    "ClusterId.$": "$.cluster.ClusterId"
  },
  "End": true
}
}
}

```

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. It's a best practice to include only those permissions that are necessary in your IAM policies.

```

{
  "Version": "2012-10-17",
  "Statement": [

```

```

    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:RunJobFlow",
        "elasticmapreduce:DescribeCluster",
        "elasticmapreduce:TerminateJobFlows"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": [
        "arn:aws:iam::123456789012:role/StepFunctionsSample-EMRJobManagement-EMRServiceRole-ANPAJ2UCCR6DPCEXAMPLE",
        "arn:aws:iam::123456789012:role/StepFunctionsSample-EMRJobManagementWJALRXUTNFEMI-ANPAJ2UCCR6DPCEXAMPLE-EMRec2InstanceProfile-1ANPAJ2UCCR6DPCEXAMPLE"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:sa-east-1:123456789012:rule/StepFunctionsGetEventForEMRRunJobFlowRule"
      ]
    }
  ]
}

```

The following policy ensures that `addStep` has sufficient permissions.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [

```

```

        "elasticmapreduce:AddJobFlowSteps",
        "elasticmapreduce:DescribeStep",
        "elasticmapreduce:CancelSteps"
    ],
    "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
},
{
    "Effect": "Allow",
    "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
    ],
    "Resource": [
        "arn:aws:events:sa-east-1:123456789012:rule/
StepFunctionsGetEventForEMRAddJobFlowStepsRule"
    ]
}
]
}
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Run an EMR Serverless job

This sample project demonstrates how you can create and start an EMR Serverless application. This project also shows how you can run multiple jobs within that application.

This sample project creates the state machine, the supporting AWS resources, and configures the related IAM permissions. Explore this sample project to learn about running EMR Serverless jobs using Step Functions state machines, or use it as a starting point for your own projects.

Important

EMR Serverless does not have a free pricing tier. Running the sample project will incur costs. You can find pricing information on the [Amazon EMR Serverless pricing](#) page. In addition, the availability of EMR Serverless service integration is subject to the availability of EMR Serverless APIs. Because of this, this sample project might not work

correctly or be available in some AWS Regions. See the [Other considerations](#) topic for information about availability of EMR Serverless in AWS Regions.

AWS CloudFormation template and additional resources

You use a CloudFormation template to deploy this sample project. This template creates the following resources in your AWS account:

- A Step Functions state machine.
- Execution role for the state machine. This role grants the permissions that your state machine needs to access other AWS services and resources such as the EMR Serverless [CreateApplication](#) action.
- Job execution role for EMR Serverless. This role grants the permissions that an EMR Serverless job run can assume when it calls other services on your behalf.

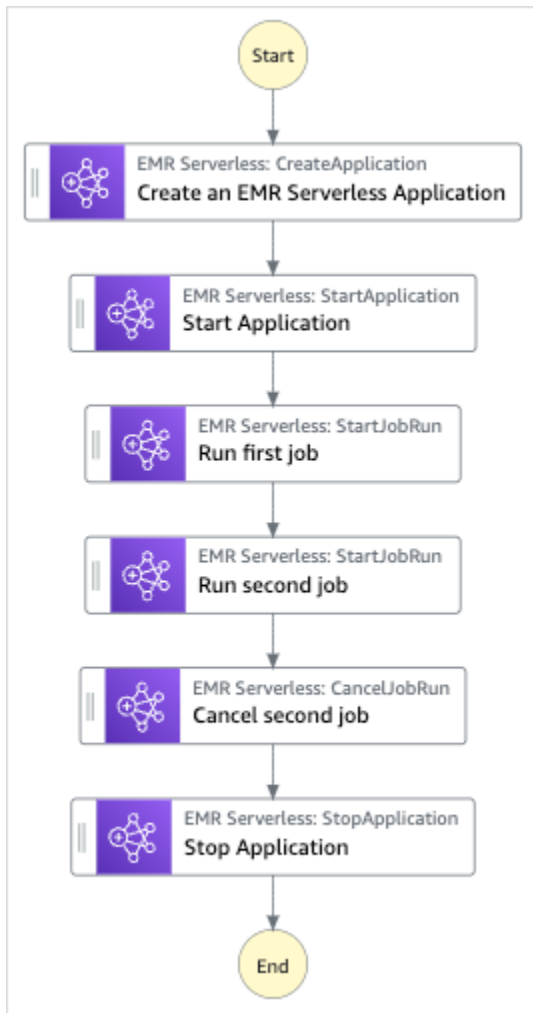
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **EMR Serverless** in the search box, and then choose **Run an EMR Serverless job** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- A Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Run an EMR Serverless job** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions automatically generates a unique execution name.

Note

Step Functions allows you to create names for state machines, executions, activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

3. Choose **Start execution**.

4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Start a Workflow within a Workflow (Step Functions, Lambda)

This sample project demonstrates how to use an AWS Step Functions state machine to start other state machine executions. For information about starting state machine executions from another state machine, see [Start Workflow Executions from a Task State](#).

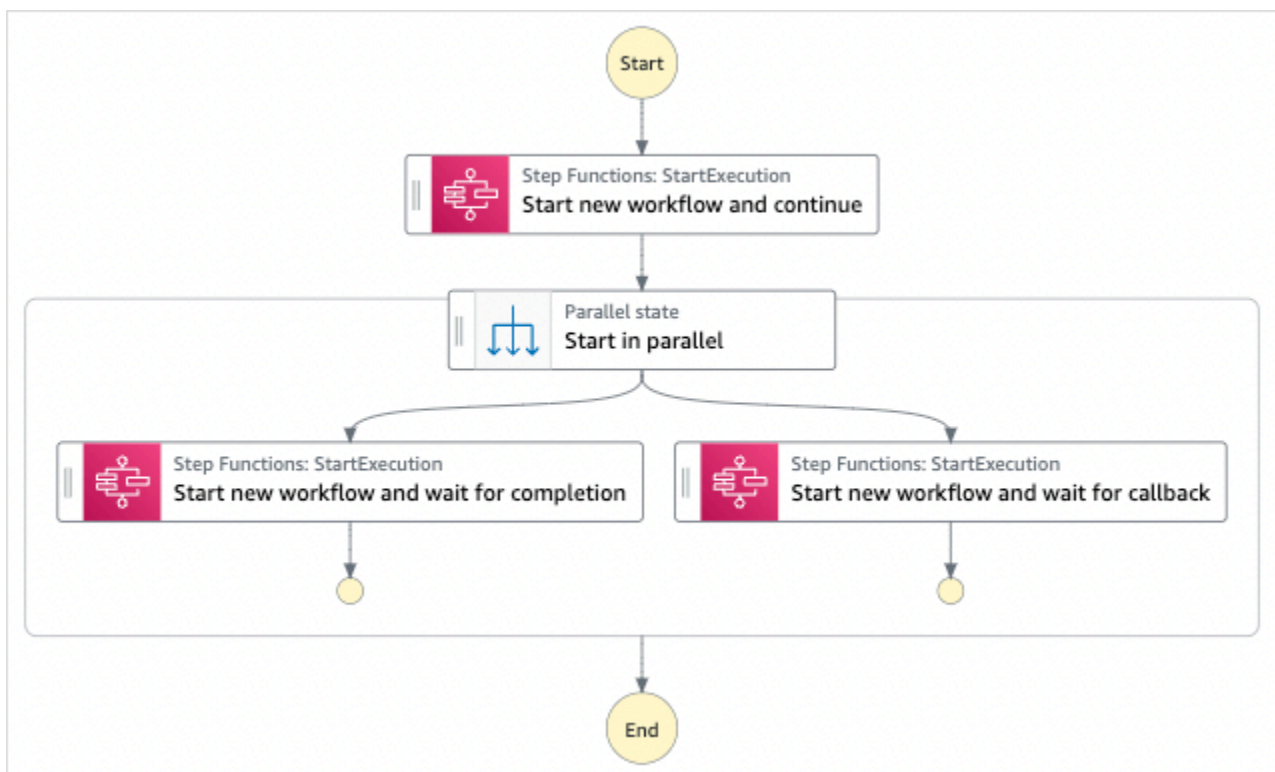
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Start a workflow within a workflow** in the search box, and then choose **Start a workflow within a workflow** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An additional state machine. The execution of this state machine is started by the state machine that you run.
- A callback Lambda function. This function is used in the additional state machine to implement the callback mechanism.
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Start a workflow within a workflow** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.


 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.


2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

 **Note**

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

 **Note**

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates another state machine and AWS Lambda by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions calls the [StartExecution](#) API action for the other state machine. It launches two instances of the other state machine in parallel: one using the [Run a Job \(.sync\)](#) pattern and one using the [Wait for a Callback with the Task Token](#) pattern.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of combining workflows using a Step Functions StartExecution
task state with various integration patterns.",
  "StartAt": "Start new workflow and continue",
  "States": {
    "Start new workflow and continue": {
      "Comment": "Start an execution of another Step Functions state machine and
continue",
      "Type": "Task",
      "Resource": "arn:aws:states:::states:startExecution",
      "Parameters": {
        "StateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:NestingPatternAnotherStateMachine-HZ9gtgspmdun",
        "Input": {
          "NeedCallback": false,
          "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
        }
      },
      "Next": "Start in parallel"
    },
    "Start in parallel": {
      "Comment": "Start two executions of the same state machine in parallel",
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "Start new workflow and wait for completion",
          "States": {
            "Start new workflow and wait for completion": {
              "Comment": "Start an execution of the same
'NestingPatternAnotherStateMachine' and wait for its completion",
              "Type": "Task",
              "Resource": "arn:aws:states:::states:startExecution.sync",
              "Parameters": {
```


Dynamically process data with a Map state

This sample project demonstrates dynamic parallelism using a [Map](#) state.

In this project, Step Functions uses an AWS Lambda function to pull messages off an Amazon SQS queue, and pass a JSON array of those message to a Map state. For each message in the queue, the state machine writes the message to DynamoDB, invokes the other Lambda function to remove the message from Amazon SQS, and then publishes the message to the Amazon SNS topic.

For more information on Map states and Step Functions service integrations, see the following:

- [Map](#)
- [Using AWS Step Functions with other services](#)

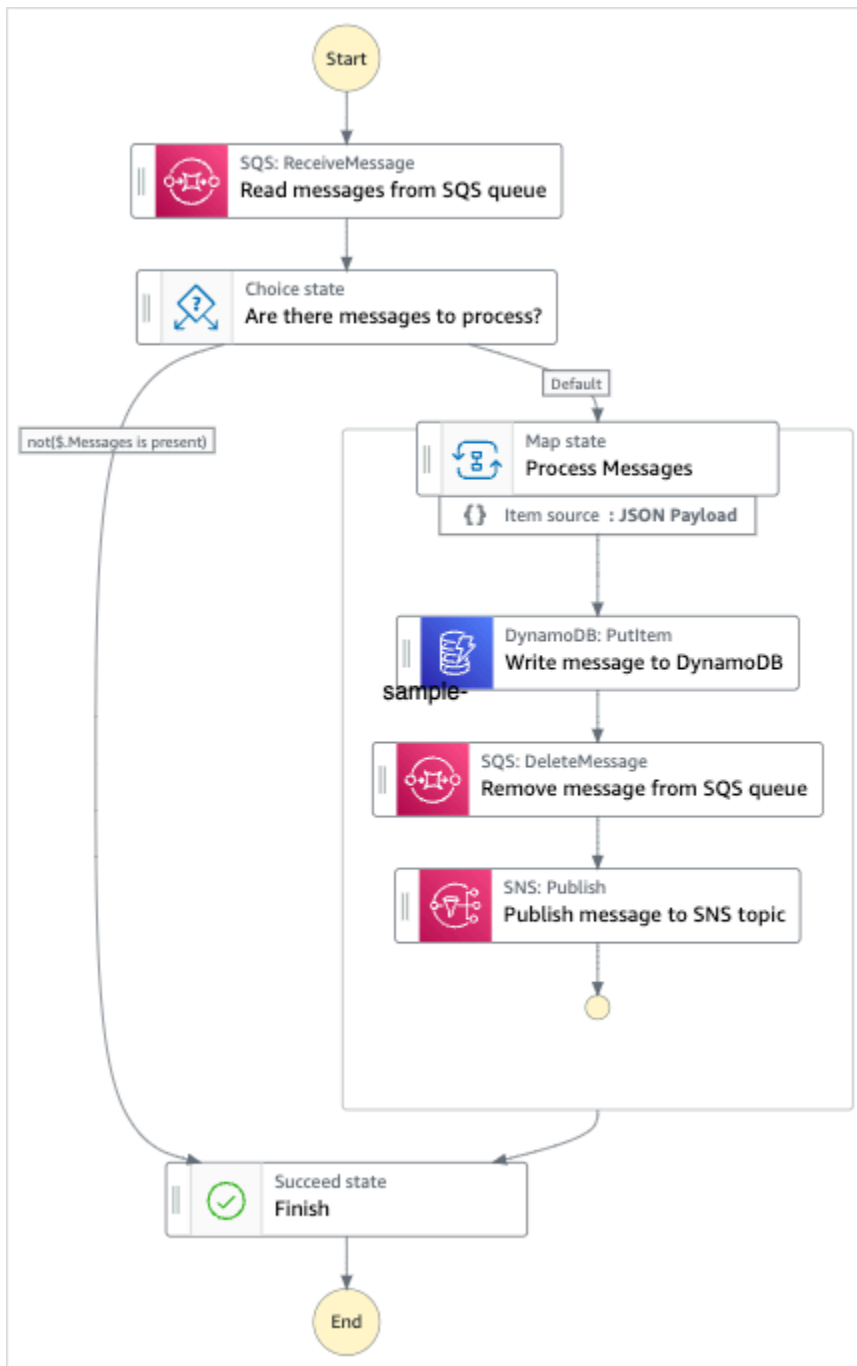
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Dynamically process data with a Map state** in the search box, and then choose **Dynamically process data with a Map state** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon SQS queue from which the Map state reads and removes messages iteratively.
- A DynamoDB table to which the Map state writes messages iteratively.
- An Amazon SNS topic to which Step Functions publishes the messages it reads from the Amazon SQS queue.
- Two AWS Lambda functions
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Dynamically process data with a Map state** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:

- If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

After the resources of the sample project are deployed, you must add items to the Amazon SQS queue and subscribe to the Amazon SNS topic before you run the state machine.

Step 2: Subscribe to the Amazon SNS topic

1. Open the [Amazon SNS console](#).
2. Choose **Topics** and choose the topic that was created by the Map state sample project.

The name will be similar to **MapSampleProj-SNSTopic-1CQO4HQ3IR1KN**.

3. Choose **Create subscription**.

The **Create subscription** page is displayed, listing the **Topic ARN** for the topic.

4. Under **Protocol**, choose **Email**.
5. Under **Endpoint**, enter an email address to subscribe to the topic.
6. Choose **Create subscription**.

Note

You must confirm the subscription in your email before it is active.

7. Open the **Subscription Confirmation** email in the related account and open the **Confirm subscription** URL.

The **Subscription confirmed!** page is displayed.

Step 3: Add messages to the Amazon SQS queue

1. Open the [Amazon SQS console](#).
2. Choose the queue that was created by the Map state sample project.

The name will be similar to **MapSampleProj-SQSQueue-1UDIC9VZDORN7**.

3. Choose **Send and receive messages**.
4. On the **Send and receive messages** page, enter a message and choose **Send message**.
5. Entering another message and choose **Send message**. Continue entering more messages until you have several in the Amazon SQS queue.

Step 4: Run the state machine

Note

Queues in Amazon SNS are eventually consistent. For best results, wait a few minutes between populating your queue and running an execution of your state machine.

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including

input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example state machine code

The state machine in this sample project integrates with Amazon SQS, Amazon SNS, and Lambda by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls Lambda, DynamoDB, Amazon SNS by connecting to the Amazon Resource Name (ARN) in the `Resource` field, and by passing `Parameters` to the service API.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of the Amazon States Language for reading messages from an SQS
  queue and iteratively processing each message.",
  "StartAt": "Read messages from SQS Queue",
  "States": {
    "Read messages from SQS Queue": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "OutputPath": "$.Payload",
      "Parameters": {
        "FunctionName": "MapSampleProj-ReadFromSQSQueueLambda-1MY3M63RMJVA9"
      },
      "Next": "Are there messages to process?"
    },
    "Are there messages to process?": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$",
          "StringEquals": "No messages",
          "Next": "Finish"
        }
      ],
      "Default": "Process messages"
    },
    "Process messages": {
      "Type": "Map",
```

```
"Next": "Finish",
"ItemsPath": "$",
"Parameters": {
  "MessageNumber.$": "$$.Map.Item.Index",
  "MessageDetails.$": "$$.Map.Item.Value"
},
"Iterator": {
  "StartAt": "Write message to DynamoDB",
  "States": {
    "Write message to DynamoDB": {
      "Type": "Task",
      "Resource": "arn:aws:states:::dynamodb:putItem",
      "ResultPath": null,
      "Parameters": {
        "TableName": "MapSampleProj-DDBTable-YJDJ1MKIN6C5",
        "ReturnConsumedCapacity": "TOTAL",
        "Item": {
          "MessageId": {
            "S.$": "$.MessageDetails.MessageId"
          },
          "Body": {
            "S.$": "$.MessageDetails.Body"
          }
        }
      }
    },
    "Next": "Remove message from SQS queue"
  },
  "Remove message from SQS queue": {
    "Type": "Task",
    "Resource": "arn:aws:states:::lambda:invoke",
    "InputPath": "$.MessageDetails",
    "ResultPath": null,
    "Parameters": {
      "FunctionName": "MapSampleProj-DeleteFromSQSQueueLambda-198J2839Z05K2",
      "Payload": {
        "ReceiptHandle.$": "$.ReceiptHandle"
      }
    },
    "Next": "Publish message to SNS topic"
  },
  "Publish message to SNS topic": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish",
    "InputPath": "$.MessageDetails",
```

```

        "Parameters": {
            "Subject": "Message from Step Functions!",
            "Message.$": "$.Body",
            "TopicArn": "arn:aws:sns:us-east-1:012345678910:MapSampleProj-
SNSTopic-1CQ04HQ3IR1KN"
        },
        "End": true
    }
}
},
"Finish": {
    "Type": "Succeed"
}
}
}

```

IAM example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "arn:aws:lambda:us-east-1:012345678901:function:MapSampleProj-
ReadFromSQSQueueLambda-1MY3M63RMJVA9",
                "arn:aws:lambda:us-east-1:012345678901:function:MapSampleProj-
DeleteFromSQSQueueLambda-198J2839Z05K2"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "dynamodb:PutItem"
            ],
            "Resource": [

```

```
        "arn:aws:dynamodb:us-east-1:012345678901:table/MapSampleProj-DDBTable-
YJDJ1MKIN6C5"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "sns:Publish"
      ],
      "Resource": [
        "arn:aws:sns:us-east-1:012345678901:MapSampleProj-
SNSTopic-1CQ04HQ3IR1KN"
      ],
      "Effect": "Allow"
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Process a CSV file with Distributed Map

This sample project demonstrates how you can use the [Distributed Map state](#) to iterate over 10,000 rows of a CSV file that is generated using a Lambda function. The CSV file contains shipping information of customer orders and is stored in an Amazon S3 bucket. The Distributed Map iterates over a batch of 10 rows in the CSV file for data analysis.

The Distributed Map contains a Lambda function to detect any delayed orders. The Distributed Map also contains an [Inline Map](#) to process the delayed orders in a batch and returns these delayed orders in an array. For each delayed order, the Inline Map sends a message to an Amazon SQS queue. Finally, this sample project stores the [Map Run](#) results to another Amazon S3 bucket in your AWS account.

With Distributed Map, you can run up to 10,000 parallel child workflow executions at a time. In this sample project, the maximum concurrency of Distributed Map is set at 1000 that limits it to 1000 parallel child workflow executions.

This sample project creates the state machine, the supporting AWS resources, and configures the related IAM permissions. Explore this sample project to learn about using the Distributed Map for orchestrating large-scale, parallel workloads, or use it as a starting point for your own projects.

AWS CloudFormation template and additional resources

You use a CloudFormation template to deploy this sample project. This template creates the following resources in your AWS account:

- A Step Functions state machine.
- Execution role for the state machine. This role grants the permissions that your state machine needs to access other AWS services and resources such as the Lambda function's [Invoke](#) action.
- A Lambda function named `CSVGeneratorFunction` that generates a CSV file which contains the customer order details.
- Execution role for the CSV generator Lambda function. This role grants the function permission to access other AWS services.
- An Amazon S3 input bucket to store the generated CSV file.
- A delayed order detection Lambda function that analyzes the CSV file data and detects any delayed orders.
- Execution role for the delayed order Lambda function. This role grants the function permission to access other AWS services.
- An Amazon S3 output bucket to store the analysis results of the customer orders.
- An Amazon SQS queue to which Step Functions sends messages for every delayed order. These messages contain the IDs of the customers and their orders.
- A CloudWatch log group that stores information related to the state machine's execution history.

Important

Standard charges apply for each service.

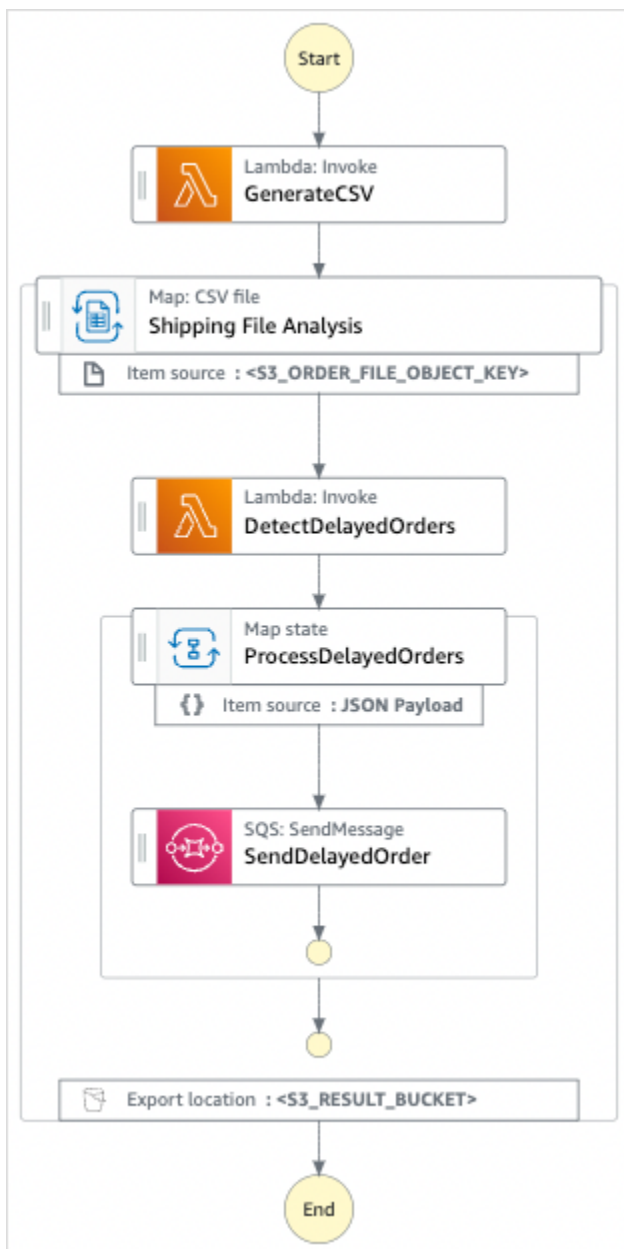
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Distributed Map to process a CSV file in S3** in the search box, and then choose **Distributed Map to process a CSV file in S3** from the search results that are returned.
3. Choose **Next** to continue.

- Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

For information about the resources that will be created for this sample project, see [AWS CloudFormation template and additional resources](#).

The following image shows the workflow graph for the **Distributed Map to process a CSV file in S3** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

After all the resources are provisioned and deployed, you can run the state machine.

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 - a. (Optional) Enter input values in JSON format to run your sample project.

If you chose to **Run a demo**, you need not provide any execution input.

ℹ Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

- b. Choose **Start execution**.
- c. (Optional) The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

After the execution is complete, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively.

- For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).
- For more information about viewing a *Distributed Map state*'s execution in the console, see [Examining Map Run](#).

- d. (Optional) Review the execution results exported to the Amazon S3 bucket. These results include data, such as execution input and output, ARN, and execution status. For more information, see [ResultWriter](#).

Process data in an Amazon S3 bucket with Distributed Map

This sample project demonstrates how you can use the [Distributed Map state](#) to process large-scale data, for example, analyze historical weather data and identify the weather station that has the highest average temperature on the planet each month. The weather data is recorded in over 12,000 CSV files, which in turn are stored in an Amazon S3 bucket.

This sample project includes two *Distributed Map states* named **Distributed S3 copy NOA Data** and **ProcessNOAaData**. **Distributed S3 copy NOA Data** iterates over the CSV files in a public Amazon S3 bucket named **noaa-gsod-pds** and copies them to an Amazon S3 bucket in your AWS account. **ProcessNOAaData** iterates over the copied files and includes a Lambda function that performs the temperature analysis.

The sample project first checks the contents of the Amazon S3 bucket with a call to the [ListObjectsV2](#) API action. Based on the number of [keys](#) returned in response to this call, the sample project takes one of the following decisions:

- If the key count is more than or equal to 1, the project transitions to the **ProcessNOAaData** state. This *Distributed Map state* includes a Lambda function named **TemperatureFunction** that finds the weather station that had the highest average temperature each month. This function returns a dictionary with year-month as the key and a dictionary that contains information about the weather station as the value.
- If the returned key count doesn't exceed 1, the **Distributed S3 copy NOA Data** state lists all objects from the public bucket **noaa-gsod-pds** and iteratively copies the individual objects to another bucket in your account in batches of 100. An [Inline Map](#) performs the iterative copying of the objects.

After all objects are copied, the project transitions to the **ProcessNOAaData** state for processing the weather data.

The sample project finally transitions to a reducer Lambda function that performs a final aggregation of the results returned by the **TemperatureFunction** function and writes the results to an Amazon DynamoDB table.

With Distributed Map, you can run up to 10,000 parallel child workflow executions at a time. In this sample project, the maximum concurrency of **ProcessNOAAData** Distributed Map is set at 3000 that limits it to 3000 parallel child workflow executions.

This sample project creates the state machine, the supporting AWS resources, and configures the related IAM permissions. Explore this sample project to learn about using the Distributed Map for orchestrating large-scale, parallel workloads, or use it as a starting point for your own projects.

Important

This sample project is only available in the US East (N. Virginia) Region.

AWS CloudFormation template and additional resources

You use a CloudFormation template to deploy this sample project. This template creates the following resources in your AWS account:

- A Step Functions state machine.
- Execution role for the state machine. This role grants the permissions that your state machine needs to access other AWS services and resources such as the Lambda function's [Invoke](#) action.
- An Amazon S3 bucket named `NOAADataBucket`. This bucket contains the CSV files with weather data.
- A Lambda function named `ReducerFunction` that performs a final aggregation of the weather data and writes the results to an Amazon DynamoDB table.
- Execution role for the reducer Lambda function. This role grants the function permission to access other AWS services.
- An Amazon S3 output bucket named `ResultsBucket` to store the weather analysis results.
- A DynamoDB table named `ResultsDynamoDBTable` that contains the results returned by the `ReducerFunction`.
- A Lambda function named `TemperatureFunction` that finds the highest monthly average temperature.
- Execution role for the Lambda function. This role grants the function permission to access other AWS services.
- A CloudWatch log group that stores information related to the state machine's execution history.

⚠ Important

Standard charges apply for each service.

Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Distributed Map to process files in S3** in the search box, and then choose **Distributed Map to process files in S3** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

For information about the resources that will be created for this sample project, see [AWS CloudFormation template and additional resources](#).

The following image shows the workflow graph for the **Distributed Map to process files in S3** sample project:

- If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.


 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

After all the resources are provisioned and deployed, you can run the state machine.

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 - a. (Optional) Enter input values in JSON format to run your sample project.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

- b. Choose **Start execution**.
- c. (Optional) The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

After the execution is complete, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively.

- For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).
 - For more information about viewing a *Distributed Map state*'s execution in the console, see [Examining Map Run](#).
- d. (Optional) Review the execution results exported to the Amazon S3 bucket. These results include data, such as execution input and output, ARN, and execution status. For more information, see [ResultWriter](#).

Train a Machine Learning Model

This sample project demonstrates how to use SageMaker and AWS Step Functions to train a machine learning model and how to batch transform a test dataset.

In this project, Step Functions uses a Lambda function to seed an Amazon S3 bucket with a test dataset. It then trains a machine learning model and performs a batch transform, using the [SageMaker service integration](#).

For more information about SageMaker and Step Functions service integrations, see the following:

- [Using AWS Step Functions with other services](#)
- [Manage SageMaker with Step Functions](#)

Note

This sample project may incur charges.

For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see [SageMaker Pricing](#).

Step 1: Create the state machine and provision resources

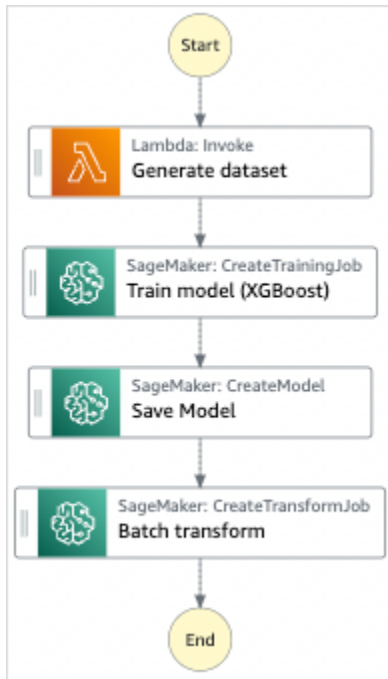
1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Train a machine learning model** in the search box, and then choose **Train a machine learning model** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An AWS Lambda function
- An Amazon Simple Storage Service (Amazon S3) bucket

- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Train a machine learning model** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with SageMaker and AWS Lambda by passing parameters directly to those resources, and uses an Amazon S3 bucket for the training data source and output.

Browse through this example state machine to see how Step Functions controls Lambda and SageMaker.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```

{
  "StartAt": "Generate dataset",
  "States": {
    "Generate dataset": {
      "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:TrainAndBatchTransform-SeedingFunction-17RNS0TG97HPV",
      "Type": "Task",
      "Next": "Train model (XGBoost)"
    },
    "Train model (XGBoost)": {
      "Resource": "arn:aws:states:::sagemaker:createTrainingJob.sync",
      "Parameters": {
        "AlgorithmSpecification": {
          "TrainingImage": "433757028032.dkr.ecr.us-west-2.amazonaws.com/
xgboost:latest",
          "TrainingInputMode": "File"
        },
        "OutputDataConfig": {
          "S3OutputPath": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/models"
        },
        "StoppingCondition": {
          "MaxRuntimeInSeconds": 86400
        },
        "ResourceConfig": {
          "InstanceCount": 1,
          "InstanceType": "ml.m4.xlarge",
          "VolumeSizeInGB": 30
        },
        "RoleArn": "arn:aws:iam::123456789012:role/TrainAndBatchTransform-
SageMakerAPIExecutionRole-Y9IX3DLF6EU0",
        "InputDataConfig": [
          {
            "DataSource": {
              "S3DataSource": {
                "S3DataDistributionType": "ShardedByS3Key",
                "S3DataType": "S3Prefix",
                "S3Uri": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/csv/
train.csv"
              }
            },
            "ChannelName": "train",
            "ContentType": "text/csv"
          }
        ]
      }
    }
  }
}

```

```

    ],
    "HyperParameters": {
      "objective": "reg:logistic",
      "eval_metric": "rmse",
      "num_round": "5"
    },
    "TrainingJobName.$": "$$.Execution.Name"
  },
  "Type": "Task",
  "Next": "Save Model"
},
"Save Model": {
  "Parameters": {
    "PrimaryContainer": {
      "Image": "433757028032.dkr.ecr.us-west-2.amazonaws.com/xgboost:latest",
      "Environment": {},
      "ModelDataUrl.$": "$$.ModelArtifacts.S3ModelArtifacts"
    },
    "ExecutionRoleArn": "arn:aws:iam::123456789012:role/TrainAndBatchTransform-
SageMakerAPIExecutionRole-Y9IX3DLF6EU0",
    "ModelName.$": "$$.TrainingJobName"
  },
  "Resource": "arn:aws:states:::sagemaker:createModel",
  "Type": "Task",
  "Next": "Batch transform"
},
"Batch transform": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sagemaker:createTransformJob.sync",
  "Parameters": {
    "ModelName.$": "$$.Execution.Name",
    "TransformInput": {
      "CompressionType": "None",
      "ContentType": "text/csv",
      "DataSource": {
        "S3DataSource": {
          "S3DataType": "S3Prefix",
          "S3Uri": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/csv/
test.csv"
        }
      }
    }
  },
  "TransformOutput": {
    "S3OutputPath": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/output"
  }
}

```

```
    },
    "TransformResources": {
      "InstanceCount": 1,
      "InstanceType": "ml.m4.xlarge"
    },
    "TransformJobName.$": "$$.Execution.Name"
  },
  "End": true
}
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "cloudwatch:PutMetricData",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:DescribeLogStreams",
        "s3:GetObject",
        "s3:PutObject",
        "s3:ListBucket",
        "ecr:GetAuthorizationToken",
        "ecr:BatchCheckLayerAvailability",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage"
      ],
      "Resource": "*",
      "Effect": "Allow"
    }
  ]
}
```

```
]
}
```

The following policy allows the Lambda function to seed the Amazon S3 bucket with sample data.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:PutObject"
      ],
      "Resource": "arn:aws:s3:::trainandbatchtransform-s3bucket-1jn11e6gadwfz/*",
      "Effect": "Allow"
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Tune a Machine Learning Model

This sample project demonstrates using SageMaker to tune the hyperparameters of a machine learning model, and to batch transform a test dataset.

In this project, Step Functions uses a Lambda function to seed an Amazon S3 bucket with a test dataset. It then creates a hyperparameter tuning job using the [SageMaker service integration](#). It then uses a Lambda function to extract the data path, saves the tuning model, extracts the model name, and then runs a batch transform job to perform inference in SageMaker.

For more information about SageMaker and Step Functions service integrations, see the following:

- [Using AWS Step Functions with other services](#)
- [Manage SageMaker with Step Functions](#)

Note

This sample project may incur charges.

For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see [SageMaker Pricing](#).

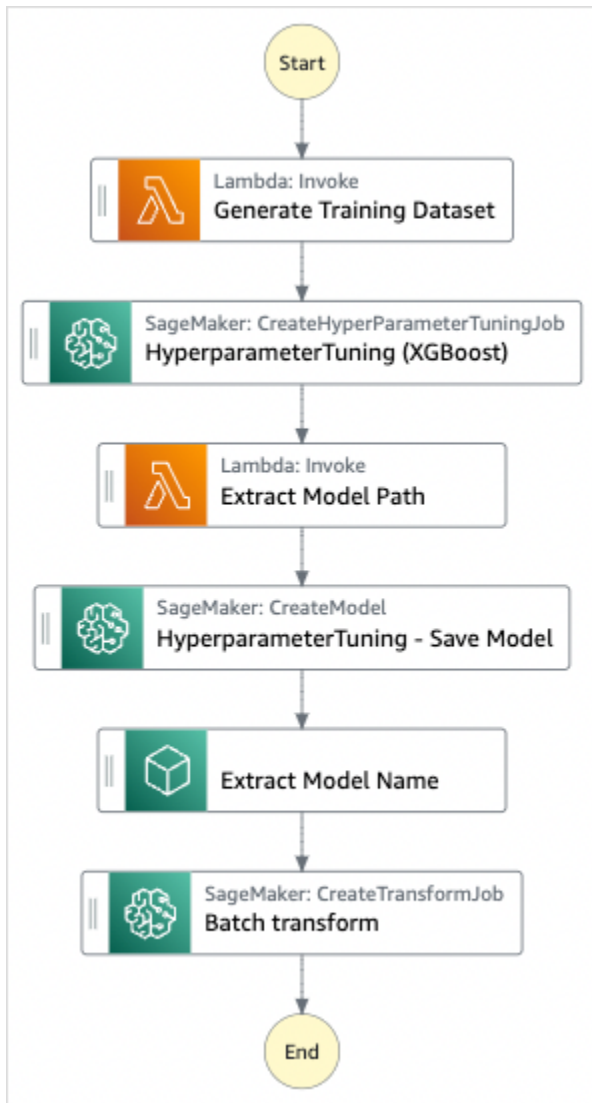
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Tune a machine learning model** in the search box, and then choose **Tune a machine learning model** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- Three AWS Lambda functions
- An Amazon Simple Storage Service (Amazon S3) bucket
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Tune a machine learning model** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with SageMaker and AWS Lambda by passing parameters directly to those resources, and uses an Amazon S3 bucket for the training data source and output.

Browse through this example state machine to see how Step Functions controls Lambda and SageMaker.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```

{
  "StartAt": "Generate Training Dataset",
  "States": {
    "Generate Training Dataset": {
      "Resource": "arn:aws:lambda:us-
west-2:012345678912:function:StepFunctionsSample-SageMa-
LambdaForDataGeneration-1TF67BUE5A12U",
      "Type": "Task",
      "Next": "HyperparameterTuning (XGBoost)"
    },
    "HyperparameterTuning (XGBoost)": {
      "Resource":
"arn:aws:states:::sagemaker:createHyperParameterTuningJob.sync",
      "Parameters": {
        "HyperParameterTuningJobName.$": "$.body.jobName",
        "HyperParameterTuningJobConfig": {
          "Strategy": "Bayesian",
          "HyperParameterTuningJobObjective": {
            "Type": "Minimize",
            "MetricName": "validation:rmse"
          },
          "ResourceLimits": {
            "MaxNumberOfTrainingJobs": 2,
            "MaxParallelTrainingJobs": 2
          },
          "ParameterRanges": {
            "ContinuousParameterRanges": [{
              "Name": "alpha",
              "MinValue": "0",
              "MaxValue": "1000",
              "ScalingType": "Auto"
            },
            {
              "Name": "gamma",
              "MinValue": "0",
              "MaxValue": "5",
              "ScalingType": "Auto"
            }
          ],
          "IntegerParameterRanges": [{
            "Name": "max_delta_step",
            "MinValue": "0",
            "MaxValue": "10",

```

```

        "ScalingType": "Auto"
    },
    {
        "Name": "max_depth",
        "MinValue": "0",
        "MaxValue": "10",
        "ScalingType": "Auto"
    }
]
}
},
"TrainingJobDefinition": {
    "AlgorithmSpecification": {
        "TrainingImage": "433757028032.dkr.ecr.us-west-2.amazonaws.com/
xgboost:latest",
        "TrainingInputMode": "File"
    },
    "OutputDataConfig": {
        "S3OutputPath": "s3://stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/models"
    },
    "StoppingCondition": {
        "MaxRuntimeInSeconds": 86400
    },
    "ResourceConfig": {
        "InstanceCount": 1,
        "InstanceType": "ml.m4.xlarge",
        "VolumeSizeInGB": 30
    },
    "RoleArn": "arn:aws:iam::012345678912:role/StepFunctionsSample-
SageM-SageMakerAPIExecutionRol-1MNH1VS5CGGOG",
    "InputDataConfig": [{
        "DataSource": {
            "S3DataSource": {
                "S3DataDistributionType": "FullyReplicated",
                "S3DataType": "S3Prefix",
                "S3Uri": "s3://stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/csv/train.csv"
            }
        },
        "ChannelName": "train",
        "ContentType": "text/csv"
    }],
}
{

```

```

        "DataSource": {
            "S3DataSource": {
                "S3DataDistributionType": "FullyReplicated",
                "S3DataType": "S3Prefix",
                "S3Uri": "s3://stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/csv/validation.csv"
            }
        },
        "ChannelName": "validation",
        "ContentType": "text/csv"
    ]],
    "StaticHyperParameters": {
        "precision_dtype": "float32",
        "num_round": "2"
    }
}
},
"Type": "Task",
"Next": "Extract Model Path"
},
"Extract Model Path": {
    "Resource": "arn:aws:lambda:us-
west-2:012345678912:function:StepFunctionsSample-SageM-LambdaToExtractModelPath-
V0R37CVARUS9",
    "Type": "Task",
    "Next": "HyperparameterTuning - Save Model"
},
"HyperparameterTuning - Save Model": {
    "Parameters": {
        "PrimaryContainer": {
            "Image": "433757028032.dkr.ecr.us-west-2.amazonaws.com/
xgboost:latest",
            "Environment": {},
            "ModelDataUrl.$": "$.body.modelDataUrl"
        },
        "ExecutionRoleArn": "arn:aws:iam::012345678912:role/
StepFunctionsSample-SageM-SageMakerAPIExecutionRol-1MNH1VS5CGG0G",
        "ModelName.$": "$.body.bestTrainingJobName"
    },
    "Resource": "arn:aws:states:::sagemaker:createModel",
    "Type": "Task",
    "Next": "Extract Model Name"
},
"Extract Model Name": {

```

```

        "Resource": "arn:aws:lambda:us-
west-2:012345678912:function:StepFunctionsSample-SageM-
LambdaToExtractModelName-8FU0B30SM5EM",
        "Type": "Task",
        "Next": "Batch transform"
    },
    "Batch transform": {
        "Type": "Task",
        "Resource": "arn:aws:states:::sagemaker:createTransformJob.sync",
        "Parameters": {
            "ModelName.$": "$.body.jobName",
            "TransformInput": {
                "CompressionType": "None",
                "ContentType": "text/csv",
                "DataSource": {
                    "S3DataSource": {
                        "S3DataType": "S3Prefix",
                        "S3Uri": "s3://stepfunctionssample-sagemak-
bucketformodelanddata-80fb1mdlcs9f/csv/test.csv"
                    }
                }
            },
            "TransformOutput": {
                "S3OutputPath": "s3://stepfunctionssample-sagemak-
bucketformodelanddata-80fb1mdlcs9f/output"
            },
            "TransformResources": {
                "InstanceCount": 1,
                "InstanceType": "ml.m4.xlarge"
            },
            "TransformJobName.$": "$.body.jobName"
        },
        "End": true
    }
}
}
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Examples

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

The following IAM policy is attached to the state machine, and allows the state machine execution to access necessary SageMaker, Lambda, and Amazon S3 resources.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "sagemaker:CreateHyperParameterTuningJob",
        "sagemaker:DescribeHyperParameterTuningJob",
        "sagemaker:StopHyperParameterTuningJob",
        "sagemaker:ListTags",
        "sagemaker:CreateModel",
        "sagemaker:CreateTransformJob",
        "iam:PassRole"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:us-west-2:012345678912:function:StepFunctionsSample-
        SageMa-LambdaForDataGeneration-1TF67BUE5A12U",
        "arn:aws:lambda:us-west-2:012345678912:function:StepFunctionsSample-
        SageM-LambdaToExtractModelPath-V0R37CVARUS9",
        "arn:aws:lambda:us-west-2:012345678912:function:StepFunctionsSample-
        SageM-LambdaToExtractModelName-8FU0B30SM5EM"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "events:PutTargets",
        "events:PutRule",

```



```

        "events:DescribeRule"
    ],
    "Resource": [
        "arn:aws:events:*:*:rule/
StepFunctionsGetEventsForSageMakerTrainingJobsRule",
        "arn:aws:events:*:*:rule/
StepFunctionsGetEventsForSageMakerTransformJobsRule",
        "arn:aws:events:*:*:rule/
StepFunctionsGetEventsForSageMakerTuningJobsRule"
    ],
    "Effect": "Allow"
}
]
}

```

The following IAM policy is referenced in the `TrainingJobDefinition` and `HyperparameterTuning` fields of the `HyperparameterTuning` state.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "cloudwatch:PutMetricData",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:DescribeLogStreams",
        "ecr:GetAuthorizationToken",
        "ecr:BatchCheckLayerAvailability",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "sagemaker:DescribeHyperParameterTuningJob",
        "sagemaker:StopHyperParameterTuningJob",
        "sagemaker:ListTags"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "s3:GetObject",
        "s3:PutObject"
      ],

```

```

    ],
    "Resource": "arn:aws:s3:::stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/*",
    "Effect": "Allow"
  },
  {
    "Action": [
      "s3:ListBucket"
    ],
    "Resource": "arn:aws:s3:::stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f",
    "Effect": "Allow"
  }
]
}

```

The following IAM policy allows the Lambda function to seed the Amazon S3 bucket with sample data.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:PutObject"
      ],
      "Resource": "arn:aws:s3:::stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/*",
      "Effect": "Allow"
    }
  ]
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Process High-Volume Messages from Amazon SQS (Express Workflows)

This sample project demonstrates how to use an AWS Step Functions Express Workflow to process messages or data from a high-volume event source, such as Amazon Simple Queue Service

(Amazon SQS). Because Express Workflows can be started at a very high rate, they are ideal for high-volume event processing or streaming data workloads.

Here are two commonly used methods to execute your state machine from an event source:

- **Configure an Amazon CloudWatch Events rule to start a state machine execution whenever the event source emits an event.** For more information, see [Creating a CloudWatch Events Rule That Triggers on an Event](#).
- **Map the event source to a Lambda function, and write function code to execute your state machine.** The AWS Lambda function is invoked each time your event source emits an event, in turn starting a state machine execution. For more information see [Using AWS Lambda with Amazon SQS](#).

This sample project uses the second method to start an execution each time the Amazon SQS queue sends a message. You can use a similar configuration to trigger Express Workflows execution from other event sources, such as Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and Amazon Kinesis.

For more information about Express Workflows and Step Functions service integrations, see the following:

- [Standard vs. Express Workflows](#)
- [Using AWS Step Functions with other services](#)
- [Quotas](#)

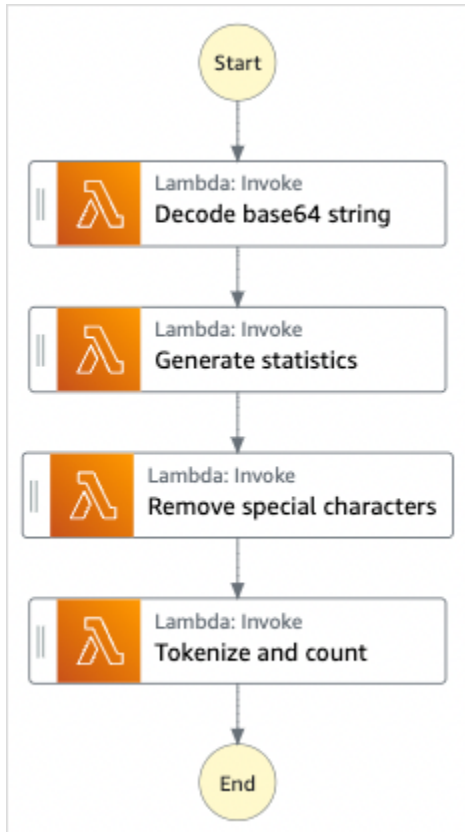
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Process high-volume messages from SQS** in the search box, and then choose **Process high-volume messages from SQS** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- Four Lambda function
- An Amazon SQS queue
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Process high-volume messages from SQS** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL)

definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Trigger the state machine execution

1. Open the [Amazon SQS console](#).
2. Select the queue that was created by the sample project.

The name will be similar to **Example-SQSQueue-wJalrXUtnFEMI**.


```
def lambda_handler(event, context):
    message_body = event['Records'][0]['body']
    client = boto3.client('stepfunctions')
    response = client.start_execution(
        stateMachineArn='${ExpressStateMachineArn}',
        input=message_body
    )
```

Example State Machine Code

The Express Workflow in this sample project consists of a set of Lambda functions for text processing.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of using Express workflows to run text processing for each message sent from an SQS queue.",
  "StartAt": "Decode base64 string",
  "States": {
    "Decode base64 string": {
      "Type": "Task",
      "Resource": "arn:<PARTITION>:states:::lambda:invoke",
      "OutputPath": "$.Payload",
      "Parameters": {
        "FunctionName": "<BASE64_DECODER_LAMBDA_FUNCTION_NAME>",
        "Payload.$": "$"
      },
      "Next": "Generate statistics"
    },
    "Generate statistics": {
      "Type": "Task",
      "Resource": "arn:<PARTITION>:states:::lambda:invoke",
      "OutputPath": "$.Payload",
      "Parameters": {
        "FunctionName": "<TEXT_STATS_GENERATING_LAMBDA_FUNCTION_NAME>",
        "Payload.$": "$"
      },
      "Next": "Remove special characters"
    },
    "Remove special characters": {
```

```

    "Type": "Task",
    "Resource": "arn:<PARTITION>:states:::lambda:invoke",
    "OutputPath": "$.Payload",
    "Parameters": {
      "FunctionName": "<STRING_CLEANING_LAMBDA_FUNCTION_NAME>",
      "Payload.$": "$"
    },
    "Next": "Tokenize and count"
  },
  "Tokenize and count": {
    "Type": "Task",
    "Resource": "arn:<PARTITION>:states:::lambda:invoke",
    "OutputPath": "$.Payload",
    "Parameters": {
      "FunctionName": "<TOKENIZING_AND_WORD_COUNTING_LAMBDA_FUNCTION_NAME>",
      "Payload.$": "$"
    },
    "End": true
  }
}
}
}

```

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:us-east-1:123456789012:function:example-Base64DecodeLambda-wJalrXUtnFEMI",
        "arn:aws:lambda:us-east-1:123456789012:function:example-StringCleanerLambda-je7MtGbClwBF",
        "arn:aws:lambda:us-east-1:123456789012:function:example-TokenizerCounterLambda-wJalrXUtnFEMI",

```



```

        "arn:aws:lambda:us-east-1:123456789012:function:example-
GenerateStatsLambda-je7MtGbClwBF"
    ],
    "Effect": "Allow"
}
]
}

```

The following policy ensures that there are sufficient permissions for CloudWatch Logs.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "logs:CreateLogDelivery",
        "logs:GetLogDelivery",
        "logs:UpdateLogDelivery",
        "logs>DeleteLogDelivery",
        "logs:ListLogDeliveries",
        "logs:PutResourcePolicy",
        "logs:DescribeResourcePolicies",
        "logs:DescribeLogGroups"
      ],
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Selective Checkpointing Example (Express Workflows)

This sample project demonstrates how to combine Standard and Express Workflows by running a mock e-commerce workflow that does selective checkpointing. Deploying this sample project creates a Standard workflows state machine, a nested Express Workflows state machine, an AWS

Lambda function, an Amazon Simple Queue Service (Amazon SQS) queue, and an Amazon Simple Notification Service (Amazon SNS) topic.

For more information about Express Workflows, nested workflows, and Step Functions service integrations, see the following:

- [Standard vs. Express Workflows](#)
- [Start Workflow Executions from a Task State](#)
- [Using AWS Step Functions with other services](#)

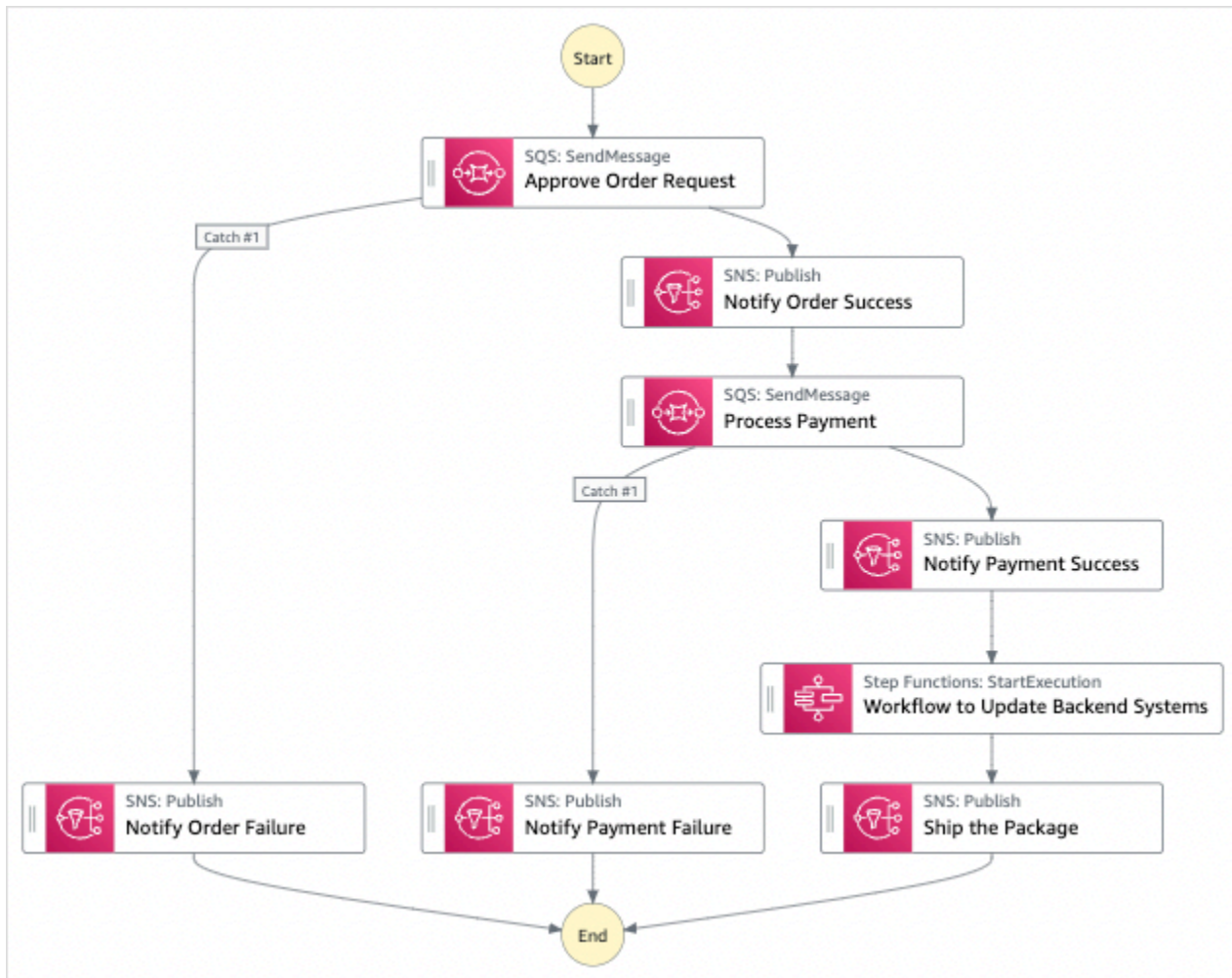
Step 1: Create the State Machine and Provision Resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Selective checkpointing example** in the search box, and then choose **Selective checkpointing example** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An AWS Lambda function
- An Amazon SQS queue
- An Amazon SNS topic
- An AWS Step Functions state machine of type Standard
- A Step Functions state machine of type Express
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Selective checkpointing example** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

After the resources of the sample project are deployed do the following.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:

1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

4. Go to your [CloudWatch Logs log group](#) and inspect the logs. The name of the log group will look like **example-ExpressLogGroup-wJalrXUtnFEMI**.

Example State Machine Code for the Parent (Standard Workflows)

The state machine in this sample project integrates with Amazon SQS , Amazon SNS, and Step Functions Express Workflows.

Browse through this example state machine to see how Step Functions processes input from Amazon SQS and Amazon SNS, and then uses a nested Express Workflows state machine to update backend systems.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of combining standard and express workflows to run a mock e-commerce workflow that does selective checkpointing.",
  "StartAt": "Approve Order Request",
  "States": {
    "Approve Order Request": {
      "Type": "Task",
      "Resource": "arn:<PARTITION>:states:::sqs:sendMessage.waitForTaskToken",
      "Parameters": {
        "QueueUrl": "<SQS_QUEUE_URL>",
        "MessageBody": {
          "MessageTitle": "Order Request received. Pausing workflow to wait for manual approval. ",
          "TaskToken.$": "$$.Task.Token"
        }
      },
      "Next": "Notify Order Success",
      "Catch": [
        {
          "ErrorEquals": [
            "States.ALL"
          ],
          "Next": "Notify Order Failure"
        }
      ]
    },
    "Notify Order Success": {
      "Type": "Task",
      "Resource": "arn:<PARTITION>:states:::sns:publish",
      "Parameters": {
        "Message": "Order has been approved. Resuming workflow.",
        "TopicArn": "<SNS_ARN>"
      },
      "Next": "Process Payment"
    },
    "Notify Order Failure": {
```

```

    "Type": "Task",
    "Resource": "arn:<PARTITION>:states:::sns:publish",
    "Parameters": {
      "Message": "Order not approved. Order failed.",
      "TopicArn": "<SNS_ARN>"
    },
    "End": true
  },
  "Process Payment": {
    "Type": "Task",
    "Resource": "arn:<PARTITION>:states:::sqs:sendMessage.waitForTaskToken",
    "Parameters": {
      "QueueUrl": "<SQS_QUEUE_URL>",
      "MessageBody": {
        "MessageTitle": "Payment sent to third-party for processing.
Pausing workflow to wait for response.",
        "TaskToken.$": "$$.Task.Token"
      }
    },
    "Next": "Notify Payment Success",
    "Catch": [
      {
        "ErrorEquals": [
          "States.ALL"
        ],
        "Next": "Notify Payment Failure"
      }
    ]
  },
  "Notify Payment Success": {
    "Type": "Task",
    "Resource": "arn:<PARTITION>:states:::sns:publish",
    "Parameters": {
      "Message": "Payment processing succeeded. Resuming workflow.",
      "TopicArn": "<SNS_ARN>"
    },
    "Next": "Workflow to Update Backend Systems"
  },
  "Notify Payment Failure": {
    "Type": "Task",
    "Resource": "arn:<PARTITION>:states:::sns:publish",
    "Parameters": {
      "Message": "Payment processing failed.",
      "TopicArn": "<SNS_ARN>"
    }
  }
}

```

```

    },
    "End": true
  },
  "Workflow to Update Backend Systems": {
    "Comment": "Starting an execution of an Express workflow to handle backend
updates. Express workflows are fast and cost-effective for steps where checkpointing
isn't required.",
    "Type": "Task",
    "Resource": "arn:<PARTITION>:states:::states:startExecution.sync",
    "Parameters": {
      "StateMachineArn": "<UPDATE_DATABASE_EXPRESS_STATE_MACHINE_ARN>",
      "Input": {
        "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
      }
    },
    "Next": "Ship the Package"
  },
  "Ship the Package": {
    "Type": "Task",
    "Resource": "arn:<PARTITION>:states:::sns:publish",
    "Parameters": {
      "Message": "Order and payment received, database is updated and the
package is ready to ship.",
      "TopicArn": "<SNS_ARN>"
    },
    "End": true
  }
}
}

```

Example IAM Role for the Parent State Machine

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

Amazon SNS policy:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [

```



```

        "sns:Publish"
    ],
    "Resource": "arn:aws:sns:us-east-1:123456789012:Checkpoint-SNSTopic-
wJalrXUtnFEMI",
    "Effect": "Allow"
}
]
}

```

Amazon SQS policy:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "sqs:SendMessage"
      ],
      "Resource": "arn:aws:sqs:us-east-1:123456789012:Checkpoint-SQSQueue-
je7MtGbClwBF",
      "Effect": "Allow"
    }
  ]
}

```

States execution policy:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "states:StartExecution",
        "states:DescribeExecution",
        "states:StopExecution"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "events:PutTargets",
        "events:PutRule",

```

```
        "events:DescribeRule"
      ],
      "Resource": "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventsForStepFunctionsExecutionRule",
      "Effect": "Allow"
    }
  ]
}
```

Example State Machine Code for the Nested State Machine (Express Workflows)

The state machine in this sample project updates backend information when called by the parent state machine.

Browse through this example state machine to see how Step Functions updates the different components of the mock e-commerce backend systems.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).



```
{
```

```
"StartAt": "Update Order History",
"States": {
  "Update Order History": {
    "Type": "Task",
    "Resource": "arn:aws:states:::lambda:invoke",
    "Parameters": {
      "FunctionName": "Checkpoint-UpdateDatabaseLambdaFunction-wJalrXUtnFEMI",
      "Payload": {
        "Message": "Update order history."
      }
    }
  },
  "Next": "Update Data Warehouse"
},
"Update Data Warehouse": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "Parameters": {
    "FunctionName": "Checkpoint-UpdateDatabaseLambdaFunction-wJalrXUtnFEMI",
    "Payload": {
      "Message": "Update data warehouse."
    }
  }
},
"Next": "Update Customer Profile"
},
"Update Customer Profile": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "Parameters": {
    "FunctionName": "Checkpoint-UpdateDatabaseLambdaFunction-wJalrXUtnFEMI",
    "Payload": {
      "Message": "Update customer profile."
    }
  }
},
"Next": "Update Inventory"
},
"Update Inventory": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "Parameters": {
    "FunctionName": "Checkpoint-UpdateDatabaseLambdaFunction-wJalrXUtnFEMI",
    "Payload": {
      "Message": "Update inventory."
    }
  }
},
}
```

```
    "End": true
  }
}
```

Example IAM Role for Child State Machine

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:us-east-1:123456789012:function:Example-UpdateDatabaseLambdaFunction-wJalrXUtnFEMI"
      ],
      "Effect": "Allow"
    }
  ]
}
```

The following policy ensures that there are sufficient permissions for CloudWatch Logs.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "logs:CreateLogDelivery",
        "logs:GetLogDelivery",
        "logs:UpdateLogDelivery",
        "logs>DeleteLogDelivery",
        "logs>ListLogDeliveries",
        "logs:PutResourcePolicy",
        "logs:DescribeResourcePolicies",
        "logs:DescribeLogGroups"
      ]
    }
  ]
}
```

```
    ],
    "Resource": [
        "*"
    ],
    "Effect": "Allow"
}
]
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Build an AWS CodeBuild Project (CodeBuild, Amazon SNS)

This sample project demonstrates how to use AWS Step Functions to build an AWS CodeBuild project, run tests, and then send an Amazon SNS notification.

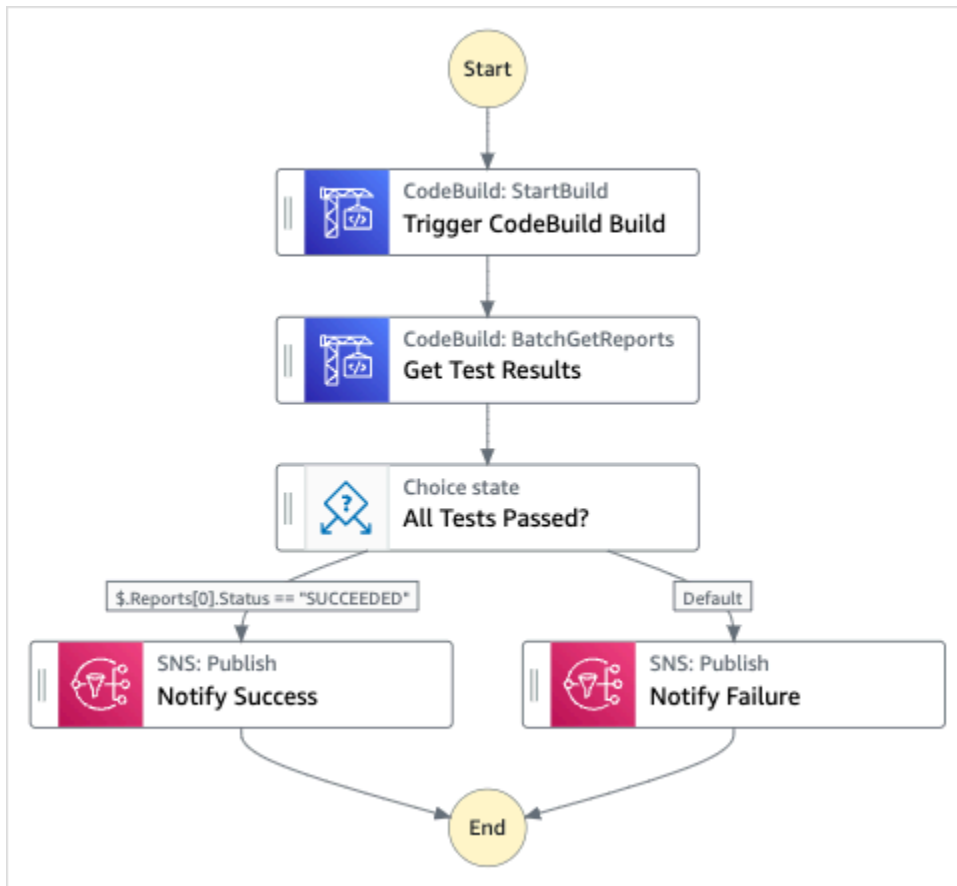
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Start a CodeBuild build** in the search box, and then choose **Start a CodeBuild build** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An AWS CodeBuild build
- An Amazon SNS topic
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Start a CodeBuild build** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.


 Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names

don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

 **Note**

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with CodeBuild and Amazon SNS.

Browse through this example state machine to see how Step Functions uses a state machine to build a CodeBuild project, and then sends an Amazon SNS topic with a message about whether the job succeeded or failed.

For more information about how Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of using CodeBuild to run tests, get test results and send a notification.",
  "StartAt": "Trigger CodeBuild Build",
  "States": {
    "Trigger CodeBuild Build": {
```



```
    "Type": "Task",
    "Resource": "arn:aws:states:::codebuild:startBuild.sync",
    "Parameters": {
      "ProjectName": "CodeBuildProject-Dtw1jBhEYGdf"
    },
    "Next": "Get Test Results"
  },
  "Get Test Results": {
    "Type": "Task",
    "Resource": "arn:aws:states:::codebuild:batchGetReports",
    "Parameters": {
      "ReportArns.$": "$.Build.ReportArns"
    },
    "Next": "All Tests Passed?"
  },
  "All Tests Passed?": {
    "Type": "Choice",
    "Choices": [
      {
        "Variable": "$.Reports[0].Status",
        "StringEquals": "SUCCEEDED",
        "Next": "Notify Success"
      }
    ],
    "Default": "Notify Failure"
  },
  "Notify Success": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish",
    "Parameters": {
      "Message": "CodeBuild build tests succeeded",
      "TopicArn": "arn:aws:sns:sa-east-1:123456789012:StepFunctionsSample-CodeBuildExecution3da9ead6-bc1f-4441-99ac-591c140019c4-SNSTopic-EVYLVNGW85JP"
    },
    "End": true
  },
  "Notify Failure": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish",
    "Parameters": {
      "Message": "CodeBuild build tests failed",
      "TopicArn": "arn:aws:sns:sa-east-1:123456789012:StepFunctionsSample-CodeBuildExecution3da9ead6-bc1f-4441-99ac-591c140019c4-SNSTopic-EVYLVNGW85JP"
    },
  },
```

```
        "End": true
    }
}
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Preprocess data and train a machine learning model

This sample project demonstrates how to use SageMaker and AWS Step Functions to preprocess data and train a machine learning model.

In this project, Step Functions uses a Lambda function to seed an Amazon S3 bucket with a test dataset and a Python script for data processing. It then trains a machine learning model and performs a batch transform, using the [SageMaker service integration](#).

For more information about SageMaker and Step Functions service integrations, see the following:

- [Using AWS Step Functions with other services](#)
- [Manage SageMaker with Step Functions](#)

Note

This sample project may incur charges.

For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see [SageMaker Pricing](#).

Step 1: Create the state machine and provision resources

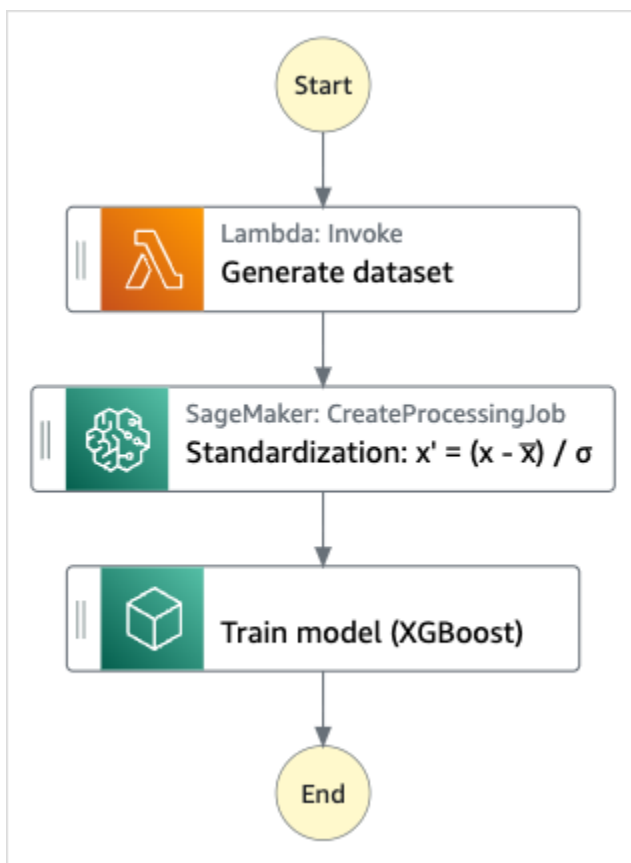
1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Preprocess data and train a machine learning model** in the search box, and then choose **Preprocess data and train a machine learning model** from the search results that are returned.
3. Choose **Next** to continue.

- Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An AWS Lambda function
- An Amazon S3 bucket
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Preprocess data and train a machine learning model** sample project:



- Choose **Use template** to continue with your selection.
- Do one of the following:

- If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with SageMaker and AWS Lambda by passing parameters directly to those resources, and uses an Amazon S3 bucket for the training data source and output.

Browse through this example state machine to see how Step Functions controls Lambda and SageMaker.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "StartAt": "Generate dataset",
  "States": {
    "Generate dataset": {
      "Resource": "arn:aws:lambda:sa-east-1:1234567890:function:FeatureTransform-
LambdaForDataGeneration-17M8LX7I09LUW",
      "Type": "Task",
      "Next": "Standardization:  $x' = (x - \bar{x}) / \sigma$ "
    },
    "Standardization:  $x' = (x - \bar{x}) / \sigma$ ": {
      "Resource": "arn:aws:states:::sagemaker:createProcessingJob.sync",
      "Parameters": {
        "ProcessingResources": {
          "ClusterConfig": {
            "InstanceCount": 1,
            "InstanceType": "ml.m5.xlarge",
            "VolumeSizeInGB": 10
          }
        }
      },
      "ProcessingInputs": [
        {
          "InputName": "input-1",
          "S3Input": {
            "S3Uri": "s3://featuretransform-bucketforcodeanddata-1jn1le6gadwfz/
input/raw.csv",
            "LocalPath": "/opt/ml/processing/input",
            "S3DataType": "S3Prefix",
            "S3InputMode": "File",
            "S3DataDistributionType": "FullyReplicated",
            "S3CompressionType": "None"
          }
        }
      ]
    }
  }
}
```

```

    },
    {
      "InputName": "code",
      "S3Input": {
        "S3Uri": "s3://featuretransform-bucketforcodeanddata-1jn1le6gadwfz/
code/transform.py",
        "LocalPath": "/opt/ml/processing/input/code",
        "S3DataType": "S3Prefix",
        "S3InputMode": "File",
        "S3DataDistributionType": "FullyReplicated",
        "S3CompressionType": "None"
      }
    }
  ],
  "ProcessingOutputConfig": {
    "Outputs": [
      {
        "OutputName": "train_data",
        "S3Output": {
          "S3Uri": "s3://featuretransform-
bucketforcodeanddata-1jn1le6gadwfz/train",
          "LocalPath": "/opt/ml/processing/output/train",
          "S3UploadMode": "EndOfJob"
        }
      }
    ]
  },
  "AppSpecification": {
    "ImageUri": "737474898029.dkr.ecr.sa-east-1.amazonaws.com/sagemaker-scikit-
learn:0.20.0-cpu-py3",
    "ContainerEntrypoint": [
      "python3",
      "/opt/ml/processing/input/code/transform.py"
    ]
  },
  "StoppingCondition": {
    "MaxRuntimeInSeconds": 300
  },
  "RoleArn": "arn:aws:iam::1234567890:role/SageMakerAPIExecutionRole-
AIDACKCEVSQ6C2EXAMPLE",
  "ProcessingJobName.$": "$$.Execution.Name"
},
"Type": "Task",
"Next": "Train model (XGBoost)"

```

```

    },
    "Train model (XGBoost)": {
      "Resource": "arn:aws:states:::sagemaker:createTrainingJob.sync",
      "Parameters": {
        "AlgorithmSpecification": {
          "TrainingImage": "855470959533.dkr.ecr.sa-east-1.amazonaws.com/
xgboost:latest",
          "TrainingInputMode": "File"
        },
        "OutputDataConfig": {
          "S3OutputPath": "s3://featuretransform-bucketforcodeanddata-1jn1le6gadwfz/
models"
        },
        "StoppingCondition": {
          "MaxRuntimeInSeconds": 86400
        },
        "ResourceConfig": {
          "InstanceCount": 1,
          "InstanceType": "ml.m5.xlarge",
          "VolumeSizeInGB": 30
        },
        "RoleArn": "arn:aws:iam::1234567890:role/SageMakerAPIExecutionRole-
AIDACKCEVSQ6C2EXAMPLE",
        "InputDataConfig": [
          {
            "DataSource": {
              "S3DataSource": {
                "S3DataDistributionType": "ShardedByS3Key",
                "S3DataType": "S3Prefix",
                "S3Uri": "s3://featuretransform-bucketforcodeanddata-1jn1le6gadwfz"
              }
            },
            "ChannelName": "train",
            "ContentType": "text/csv"
          }
        ],
        "HyperParameters": {
          "objective": "reg:logistic",
          "eval_metric": "rmse",
          "num_round": "5"
        },
        "TrainingJobName.$": "$$.Execution.Name"
      },
      "Type": "Task",

```



```
    "End": true
  }
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "cloudwatch:PutMetricData",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:DescribeLogStreams",
        "s3:GetObject",
        "s3:PutObject",
        "s3:ListBucket",
        "ecr:GetAuthorizationToken",
        "ecr:BatchCheckLayerAvailability",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage"
      ],
      "Resource": "*",
      "Effect": "Allow"
    }
  ]
}
```

The following policy allows the Lambda function to seed the Amazon S3 bucket with sample data.

```
{
  "Version": "2012-10-17",
```

```
"Statement": [
  {
    "Action": [
      "s3:PutObject"
    ],
    "Resource": "arn:aws:s3:::featuretransform-
bucketforcodeanddata-1jn11e6gadwfz/*",
    "Effect": "Allow"
  }
]
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Lambda orchestration example

This sample project demonstrates how to integrate AWS Lambda functions in Step Functions state machines.

In this project, Step Functions uses Lambda functions to check a stock price and determine a buy or sell trading recommendation. The user is then provided this recommendation and can choose whether to buy or sell the stock. The result of the trade is returned using an SNS topic.

For more information about Step Functions service integrations, see the following:

- [Using AWS Step Functions with other services](#)
- IAM policies for:
 - [IAM policies for AWS Lambda](#)
 - [IAM policies for Amazon SQS](#)
 - [IAM policies for Amazon SNS](#)

Note

This sample project may incur charges.

For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the free tier, see [Pricing](#).

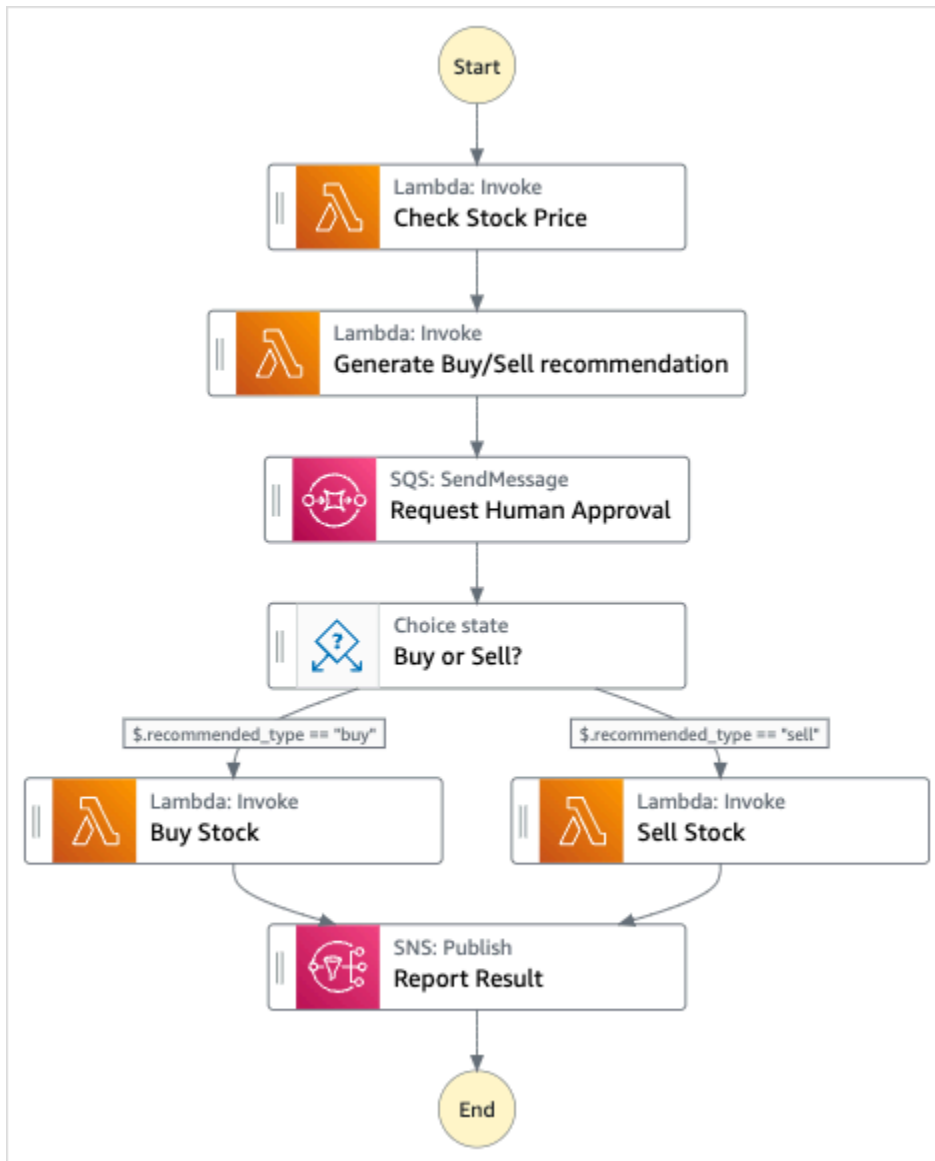
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Orchestrate Lambda functions** in the search box, and then choose **Orchestrate Lambda functions** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- Five Lambda functions
- An Amazon Simple Queue Service queue
- An Amazon Simple Notification Service topic
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Orchestrate Lambda functions** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

After all the resources are provisioned and deployed, the **Start execution** dialog box is displayed.

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:

1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

About the state machine and its execution

The state machine in this sample project integrates with AWS Lambda by passing parameters directly to those resources, uses an Amazon SQS queue to manage the request for human approval, and uses an Amazon SNS topic to return the results of the query.

A Step Functions execution receives a JSON text as input and passes that input to the first state in the workflow. Individual states receive JSON data as input and usually pass JSON data as output

to the next state. In this sample project, the output of each step is passed as input to the next step in the workflow. For example, the **Generate Buy/Sell recommendation** step receives the output of the **Check Stock Price** step as input. Further, the output of the **Generate Buy/Sell recommendation** step is passed as input to the next step, **Request Human Approval**, which mimics a human approval step.

Note

To view the output returned by a step and the input passed on to a step, open the *Execution Details* page for your workflow execution. In the [Step details](#) section, view the input and output for each step you select in the [View mode](#).

To implement a human approval step, you typically pause the workflow execution until a task token is returned. In this sample project, a message is passed to an Amazon SQS queue, which is used as a trigger to the Lambda function defined to handle callback functionality. The message contains a task token and the output returned by the preceding step. The Lambda function is invoked with the payload of the message. The workflow execution is paused until it receives the task token back with a [SendTaskSuccess](#) API call. For more information about task tokens, see [Wait for a Callback with the Task Token](#).

The following code for the `StepFunctionsSample-HelloLambda-ApproveSqsLambda` function shows how it is defined to automatically approve any tasks submitted by the Amazon SQS queue through the Step Functions state machine.

Sample Lambda function code to handle callback functionality and return the task token

```
exports.lambdaHandler = (event, context, callback) => {
  const stepfunctions = new aws.StepFunctions();

  // For every record in sqs queue
  for (const record of event.Records) {
    const messageBody = JSON.parse(record.body);
    const taskToken = messageBody.TaskToken;

    const params = {
      output: "\"approved\"",
      taskToken: taskToken
    };
  };
};
```

```
    console.log(`Calling Step Functions to complete callback task with params
    ${JSON.stringify(params)}`);

    // Approve
    stepfunctions.sendTaskSuccess(params, (err, data) => {
      if (err) {
        console.error(err.message);
        callback(err.message);
        return;
      }
      console.log(data);
      callback(null);
    });
  }
};
```

Browse through this example state machine to see how Step Functions controls Lambda and Amazon SQS.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "StartAt": "Check Stock Price",
  "States": {
    "Check Stock Price": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-
west-1:111122223333:function:StepFunctionsSample-HelloLam-
CheckStockPriceLambda-444455556666",
      "Next": "Generate Buy/Sell recommendation"
    },
    "Generate Buy/Sell recommendation": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-
west-1:111122223333:function:StepFunctionsSample-Hello-
GenerateBuySellRecommend-123456789012",
      "ResultPath": "$.recommended_type",
      "Next": "Request Human Approval"
    },
    "Request Human Approval": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
```



```
    "Parameters": {
      "QueueUrl": "https://sqs.us-west-1.amazonaws.com/111122223333/
StepFunctionsSample-HelloLambda4444-5555-6666-RequestHumanApprovalSqs-777788889999",
      "MessageBody": {
        "Input.$": "$",
        "TaskToken.$": "$$.Task.Token"
      }
    },
    "ResultPath": null,
    "Next": "Buy or Sell?"
  },
  "Buy or Sell?": {
    "Type": "Choice",
    "Choices": [
      {
        "Variable": "$.recommended_type",
        "StringEquals": "buy",
        "Next": "Buy Stock"
      },
      {
        "Variable": "$.recommended_type",
        "StringEquals": "sell",
        "Next": "Sell Stock"
      }
    ]
  },
  "Buy Stock": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-
west-1:111122223333:function:StepFunctionsSample-HelloLambda-
BuyStockLambda-000000000000",
    "Next": "Report Result"
  },
  "Sell Stock": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-
west-1:111122223333:function:StepFunctionsSample-HelloLambda-
SellStockLambda-111111111111",
    "Next": "Report Result"
  },
  "Report Result": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish",
    "Parameters": {
```

```

        "TopicArn": "arn:aws:sns:us-west-1:111122223333:StepFunctionsSample-
HelloLambda1111-2222-3333-ReportResultSnsTopic-222222222222",
        "Message": {
            "Input.$": "$"
        }
    },
    "End": true
}
}
}
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Examples

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```

{
  "Statement": [
    {
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": "arn:aws:lambda:us-
west-1:111122223333:function:StepFunctionsSample-HelloLam-
CheckStockPriceLambda-444455556666",
      "Effect": "Allow"
    }
  ]
}

```

```

{
  "Statement": [
    {
      "Action": [
        "lambda:InvokeFunction"
      ],

```

```

        "Resource": "arn:aws:lambda:us-
west-1:111122223333:function:StepFunctionsSample-Hello-
GenerateBuySellRecommend-123456789012",
        "Effect": "Allow"
    }
]
}

```

```

{
  "Statement": [
    {
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": "arn:aws:lambda:us-
west-1:111122223333:function:StepFunctionsSample-HelloLambda-
BuyStockLambda-777788889999",
      "Effect": "Allow"
    }
  ]
}

```

```

{
  "Statement": [
    {
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": "arn:aws:lambda:us-
west-1:111122223333:function:StepFunctionsSample-HelloLambda-
SellStockLambda-000000000000",
      "Effect": "Allow"
    }
  ]
}

```

```

{
  "Statement": [
    {
      "Action": [
        "sqs:SendMessage*"
      ],

```

```
        "Resource": "arn:aws:sqs:us-west-1:111122223333:StepFunctionsSample-HelloLambda4444-5555-6666-RequestHumanApprovalSqs-11111111111",
        "Effect": "Allow"
    }
]
```

```
{
  "Statement": [
    {
      "Action": [
        "sns:Publish"
      ],
      "Resource": "arn:aws:sns:us-west-1:111122223333:StepFunctionsSample-HelloLambda1111-2222-3333-ReportResultSnsTopic-22222222222",
      "Effect": "Allow"
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Start an Athena query

This sample project, which is based on standard workflows, demonstrates how to use Step Functions and Amazon Athena to start an Athena query and send a notification with query results.

In this project, Step Functions uses Lambda functions and an AWS Glue crawler to generate a set of example data. It then performs a query using the [Athena service integration](#) and returns the results using an SNS topic.

For more information about Athena and Step Functions service integrations, see the following:

- [Using AWS Step Functions with other services](#)
- [Call Athena with Step Functions](#)

Note

This sample project may incur charges.

For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see [Athena Pricing](#).

Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Start an Athena query** in the search box, and then choose **Start an Athena query** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon Athena query
- An AWS Glue crawler
- An Amazon SNS topic
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Start an Athena query** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with Athena and AWS Lambda by passing parameters directly to those resources, and uses an SNS topic to return the results of the query.

Browse through this example state machine to see how Step Functions controls Lambda and Athena.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
```



```
"StartAt": "Generate example log",
"States": {
  "Generate example log": {
    "Resource": "arn:aws:lambda:us-east-1:111122223333:function:StepFunctionsSample-
Athena-LambdaForDataGeneration-AKIAIOSFODNN7EXAMPLE",
    "Type": "Task",
    "Next": "Run Glue crawler"
  },
  "Run Glue crawler": {
    "Resource": "arn:aws:lambda:us-east-1:111122223333:function:StepFunctionsSample-
Athen-LambdaForInvokingCrawler-AKIAI44QH8DHBEXAMPLE",
    "Type": "Task",
    "Next": "Start an Athena query"
  },
  "Start an Athena query": {
    "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
    "Parameters": {
      "QueryString": "SELECT * FROM \"athena-sample-project-db-wJalrXUtnFEMI\".\"log
\" limit 1",
      "WorkGroup": "stepfunctions-athena-sample-project-workgroup-wJalrXUtnFEMI"
    },
    "Type": "Task",
    "Next": "Get query results"
  },
  "Get query results": {
    "Resource": "arn:aws:states:::athena:getQueryResults",
    "Parameters": {
      "QueryExecutionId.$": "$.QueryExecution.QueryExecutionId"
    },
    "Type": "Task",
    "Next": "Send query results"
  },
  "Send query results": {
    "Resource": "arn:aws:states:::sns:publish",
    "Parameters": {
      "TopicArn": "arn:aws:sns:us-east-1:111122223333:StepFunctionsSample-
AthenaDataQueryd1111-2222-3333-777788889999-SNSTopic-ANPAJ2UCCR6DPCEXAMPLE",
      "Message": {
        "Input.$": "$.ResultSet.Rows"
      }
    },
    "Type": "Task",
    "End": true
  }
}
```

```
}  
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Action": [  
        "lambda:InvokeFunction"  
      ],  
      "Resource": [  
        "arn:aws:lambda:us-east-1:111122223333:function:StepFunctionsSample-  
Athena-LambdaForDataGeneration-AKIAIOSFODNN7EXAMPLE",  
        "arn:aws:lambda:us-east-1:111122223333:function:StepFunctionsSample-  
Athen-LambdaForInvokingCrawler-AKIAI44QH8DHBEXAMPLE"  
      ],  
      "Effect": "Allow"  
    },  
    {  
      "Action": [  
        "sns:Publish"  
      ],  
      "Resource": [  
        "arn:aws:sns:us-east-1:111122223333:StepFunctionsSample-  
AthenaDataQueryd1111-2222-3333-777788889999-SNSTopic-ANPAJ2UCCR6DPCEXAMPLE"  
      ],  
      "Effect": "Allow"  
    },  
    {  
      "Action": [  
        "athena:getQueryResults",  
        "athena:startQueryExecution",  
        "athena:stopQueryExecution",  
        "athena:getQueryExecution",  
      ]  
    }  
  ]  
}
```

```

        "athena:getDataCatalog"
    ],
    "Resource": [
        "arn:aws:athena:us-east-1:111122223333:workgroup/stepfunctions-athena-
sample-project-workgroup-wJalrXUtnFEMI",
        "arn:aws:athena:us-east-1:111122223333:datacatalog/*"
    ],
    "Effect": "Allow"
},
{
    "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:ListBucketMultipartUploads",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload",
        "s3:CreateBucket",
        "s3:PutObject"
    ],
    "Resource": "arn:aws:s3:::*",
    "Effect": "Allow"
},
{
    "Action": [
        "glue:CreateDatabase",
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:UpdateDatabase",
        "glue>DeleteDatabase",
        "glue:CreateTable",
        "glue:UpdateTable",
        "glue:GetTable",
        "glue:GetTables",
        "glue>DeleteTable",
        "glue:BatchDeleteTable",
        "glue:BatchCreatePartition",
        "glue:CreatePartition",
        "glue:UpdatePartition",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:BatchGetPartition",
        "glue>DeletePartition",
        "glue:BatchDeletePartition"
    ]
}

```

```
    ],
    "Resource": [
      "arn:aws:glue:us-east-1:111122223333:database/*",
      "arn:aws:glue:us-east-1:111122223333:table/*",
      "arn:aws:glue:us-east-1:111122223333:catalog"
    ],
    "Effect": "Allow"
  }
]
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Execute multiple queries (Amazon Athena, Amazon SNS)

This sample project demonstrates how to run Athena queries in succession and then in parallel, handle errors and then send an Amazon SNS notification based on whether the queries succeed or fail.

In this project, Step Functions uses a state machine to run Athena queries synchronously. After the query results are returned, enter parallel state with two Athena queries executing in parallel. It then waits for the job to succeed or fail, and it sends an Amazon SNS topic with a message about whether the job succeeded or failed.

Step 1: Create the state machine and provision resources

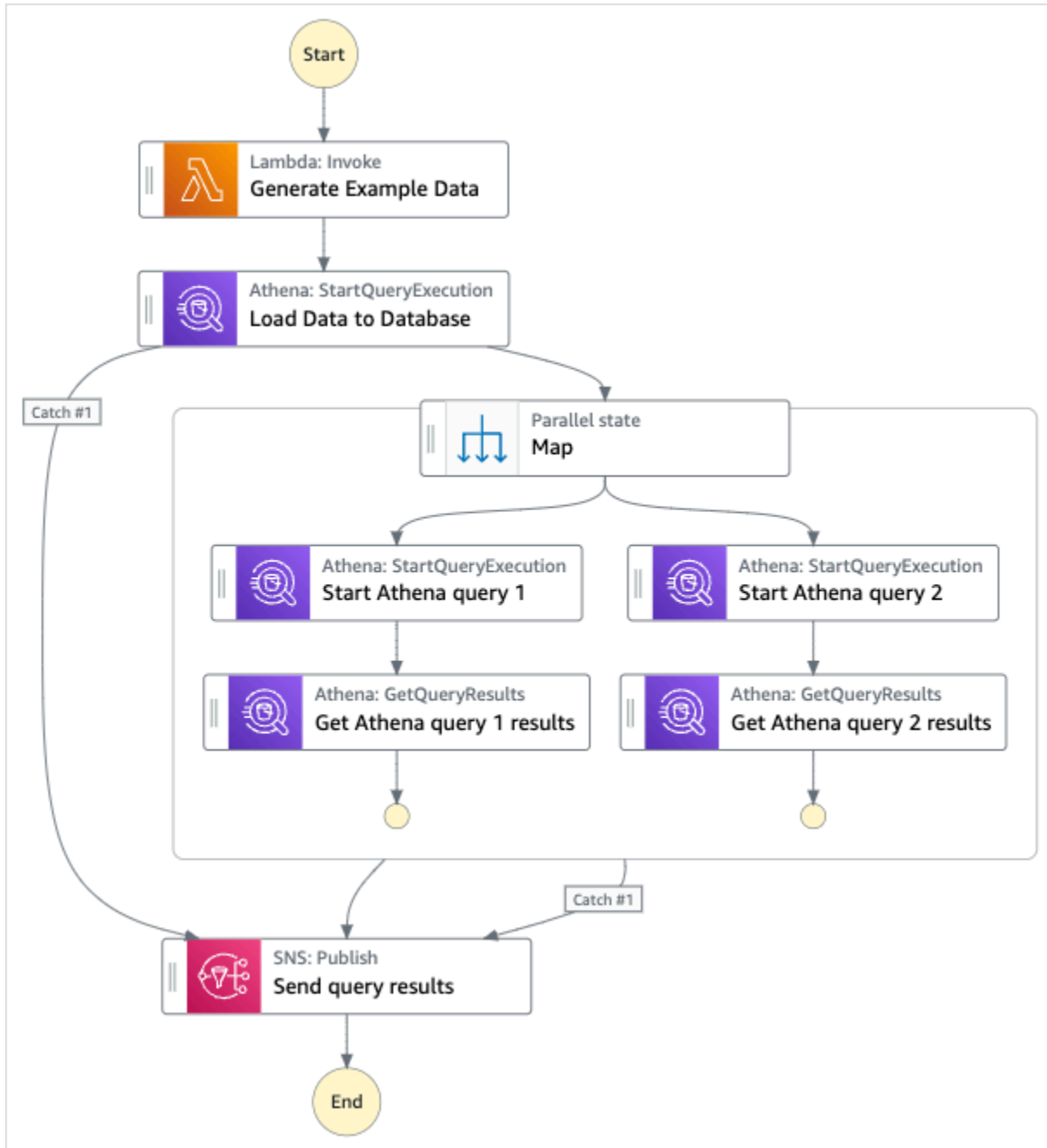
1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Execute multiple queries** in the search box, and then choose **Execute multiple queries** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- Amazon Athena queries

- An Amazon SNS topic
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Execute multiple queries** sample project:



5. Choose **Use template** to continue with your selection.

6. Do one of the following:

- If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions automatically generates a unique execution name.

ℹ Note

Step Functions allows you to create names for state machines, executions, activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with Amazon Athena and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls Amazon Athena and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the Resource field, and by passing Parameters to the service API.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of using Athena to execute queries in sequence and parallel,
with error handling and notifications.",
  "StartAt": "Generate Example Data",
  "States": {
    "Generate Example Data": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "OutputPath": "$.Payload",
      "Parameters": {
        "FunctionName": "<ATHENA_FUNCTION_NAME>"
      },
      "Next": "Load Data to Database"
    },
    "Load Data to Database": {
      "Type": "Task",
      "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
      "Parameters": {
        "QueryString": "<ATHENA_QUERYSTRING>",
        "WorkGroup": "<ATHENA_WORKGROUP>"
      },
      "Catch": [
        {
          "ErrorEquals": [
            "States.ALL"
          ],
          "Next": "Send query results"
        }
      ],
      "Next": "Map"
    }
  }
},
```



```
"Map": {
  "Type": "Parallel",
  "ResultSelector": {
    "Query1Result.$": "$[0].ResultSet.Rows",
    "Query2Result.$": "$[1].ResultSet.Rows"
  },
  "Catch": [
    {
      "ErrorEquals": [
        "States.ALL"
      ],
      "Next": "Send query results"
    }
  ],
  "Branches": [
    {
      "StartAt": "Start Athena query 1",
      "States": {
        "Start Athena query 1": {
          "Type": "Task",
          "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
          "Parameters": {
            "QueryString": "<ATHENA_QUERYSTRING>",
            "WorkGroup": "<ATHENA_WORKGROUP>"
          },
          "Next": "Get Athena query 1 results"
        },
        "Get Athena query 1 results": {
          "Type": "Task",
          "Resource": "arn:aws:states:::athena:getQueryResults",
          "Parameters": {
            "QueryExecutionId.$": "$.QueryExecution.QueryExecutionId"
          },
          "End": true
        }
      }
    },
    {
      "StartAt": "Start Athena query 2",
      "States": {
        "Start Athena query 2": {
          "Type": "Task",
          "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
          "Parameters": {
```

```

        "QueryString": "<ATHENA_QUERYSTRING>",
        "WorkGroup": "<ATHENA_WORKGROUP>"
    },
    "Next": "Get Athena query 2 results"
},
"Get Athena query 2 results": {
    "Type": "Task",
    "Resource": "arn:aws:states:::athena:getQueryResults",
    "Parameters": {
        "QueryExecutionId.$": "$.QueryExecution.QueryExecutionId"
    },
    "End": true
}
}
],
"Next": "Send query results"
},
"Send query results": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish",
    "Parameters": {
        "Message.$": "$",
        "TopicArn": "<SNS_TOPIC_ARN>"
    },
    "End": true
}
}
}

```

IAM Examples

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

AthenaStartQueryExecution

```

{
    "Version": "2012-10-17",
    "Statement": [
        {

```

```

    "Effect": "Allow",
    "Action": [
        "athena:startQueryExecution",
        "athena:stopQueryExecution",
        "athena:getQueryExecution",
        "athena:getDataCatalog"
    ],
    "Resource": [
        "arn:aws:athena:us-east-2:123456789012:workgroup/stepfunctions-athena-
sample-project-workgroup-ztuvu9yuix",
        "arn:aws:athena:us-east-2:123456789012:datacatalog/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:ListBucketMultipartUploads",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload",
        "s3:CreateBucket",
        "s3:PutObject"
    ],
    "Resource": [
        "arn:aws:s3:::*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "glue:CreateDatabase",
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:UpdateDatabase",
        "glue>DeleteDatabase",
        "glue:CreateTable",
        "glue:UpdateTable",
        "glue:GetTable",
        "glue:GetTables",
        "glue>DeleteTable",
        "glue:BatchDeleteTable",
        "glue:BatchCreatePartition",

```

```

        "glue:CreatePartition",
        "glue:UpdatePartition",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:BatchGetPartition",
        "glue>DeletePartition",
        "glue:BatchDeletePartition"
    ],
    "Resource": [
        "arn:aws:glue:us-east-2:123456789012:catalog",
        "arn:aws:glue:us-east-2:123456789012:database/*",
        "arn:aws:glue:us-east-2:123456789012:table/*",
        "arn:aws:glue:us-east-2:123456789012:userDefinedFunction/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "lakeformation:GetDataAccess"
    ],
    "Resource": [
        "*"
    ]
}
]
}

```

AthenaGetQueryResults

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:getQueryResults"
            ],
            "Resource": [
                "arn:aws:us-east-2:123456789012:workgroup/*"
            ]
        },
        {

```

```

        "Effect": "Allow",
        "Action": [
            "s3:GetObject"
        ],
        "Resource": [
            "arn:aws:s3:::*"
        ]
    }
]
}

```

SNSPublish

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sns:Publish"
      ],
      "Resource": [
        "arn:aws:sns:us-east-2:123456789012:StepFunctionsSample-
AthenaMultipleQueriesec229b-5cbe-4754-a8a8-078474bac878-SNSTopic-9AID0HEJT7TH"
      ]
    }
  ]
}

```

LambdaInvokeFunction

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:us-east-2:123456789012:function:StepFunctionsSample-
Athen-LambdaForStringGeneratio-GQFQjN7mE9gl:*"
      ]
    }
  ]
}

```

```
    ],
  },
  {
    "Effect": "Allow",
    "Action": [
      "lambda:InvokeFunction"
    ],
    "Resource": [
      "arn:aws:lambda:us-east-2:123456789012:function:StepFunctionsSample-
Athen-LambdaForStringGeneratio-GQFQjN7mE9g1"
    ]
  }
]
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Query large datasets (Amazon Athena, Amazon S3, AWS Glue, Amazon SNS)

This sample project demonstrates how to ingest a large data set in Amazon S3 and partition it through AWS Glue Crawlers, then execute Amazon Athena queries against that partition.

In this project, the Step Functions state machine invokes an AWS Glue crawler that partitions a large dataset in Amazon S3. Once the AWS Glue crawler returns a success message, the workflow executes Athena queries against that partition. Once query execution is successfully complete, an Amazon SNS notification is sent to an Amazon SNS topic.

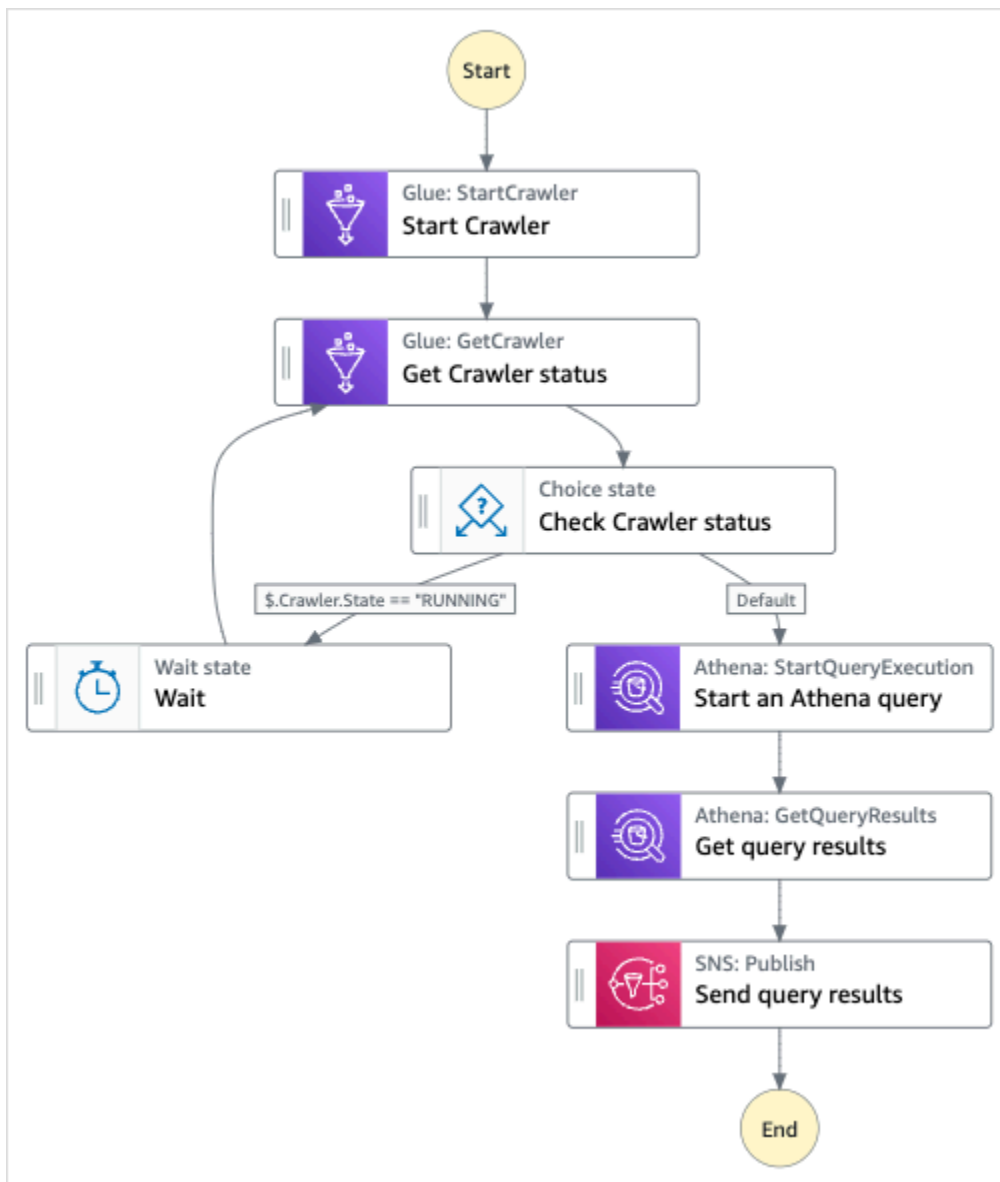
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Query large datasets** in the search box, and then choose **Query large datasets** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon S3 bucket
- An AWS Glue crawler
- An Amazon SNS topic
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Query large datasets** sample project:



5. Choose **Use template** to continue with your selection.

6. Do one of the following:

- If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions automatically generates a unique execution name.

ℹ Note

Step Functions allows you to create names for state machines, executions, activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with Amazon S3, AWS Glue, Amazon Athena and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls Amazon S3, AWS Glue, Amazon Athena and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the Resource field, and by passing Parameters to the service API.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example demonstrates how to ingest a large data set in Amazon S3 and
partition it through aws Glue Crawlers, then execute Amazon Athena queries against
that partition.",
  "StartAt": "Start Crawler",
  "States": {
    "Start Crawler": {
      "Type": "Task",
      "Next": "Get Crawler status",
      "Parameters": {
        "Name": "<GLUE_CRAWLER_NAME>"
      },
      "Resource": "arn:aws:states:::aws-sdk:glue:startCrawler"
    },
    "Get Crawler status": {
      "Type": "Task",
      "Parameters": {
        "Name": "<GLUE_CRAWLER_NAME>"
      },
      "Resource": "arn:aws:arn:aws:states:::aws-sdk:glue:getCrawler",
      "Next": "Check Crawler status"
    },
    "Check Crawler status": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.Crawler.State",
          "StringEquals": "RUNNING",
          "Next": "Wait"
        }
      ]
    }
  }
}
```

```

    "Default": "Start an Athena query"
  },
  "Wait": {
    "Type": "Wait",
    "Seconds": 30,
    "Next": "Get Crawler status"
  },
  "Start an Athena query": {
    "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
    "Parameters": {
      "QueryString": "<ATHENA_QUERYSTRING>",
      "WorkGroup": "<ATHENA_WORKGROUP>"
    },
    "Type": "Task",
    "Next": "Get query results"
  },
  "Get query results": {
    "Resource": "arn:aws:states:::athena:getQueryResults",
    "Parameters": {
      "QueryExecutionId.$": "$.QueryExecution.QueryExecutionId"
    },
    "Type": "Task",
    "Next": "Send query results"
  },
  "Send query results": {
    "Resource": "arn:aws:states:::sns:publish",
    "Parameters": {
      "TopicArn": "<SNS_TOPIC_ARN>",
      "Message": {
        "Input.$": "$.ResultSet.Rows"
      }
    },
    "Type": "Task",
    "End": true
  }
}
}
}

```

IAM Examples

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

AthenaGetQueryResults

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:getQueryResults"
      ],
      "Resource": [
        "arn:aws:athena:us-east-2:123456789012:workgroup/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::*"
      ]
    }
  ]
}
```

AthenaStartQueryExecution

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:startQueryExecution",
        "athena:stopQueryExecution",
        "athena:getQueryExecution",
        "athena:getDataCatalog"
      ],
      "Resource": [
        "arn:aws:athena:us-east-2:123456789012:workgroup/stepfunctions-athena-sample-project-workgroup-8v7bshiv70",

```

```
        "arn:aws:athena:us-east-2:123456789012:datacatalog/*"
    ],
},
{
    "Effect": "Allow",
    "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:ListBucketMultipartUploads",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload",
        "s3:CreateBucket",
        "s3:PutObject"
    ],
    "Resource": [
        "arn:aws:s3::*:*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "glue:CreateDatabase",
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:UpdateDatabase",
        "glue>DeleteDatabase",
        "glue:CreateTable",
        "glue:UpdateTable",
        "glue:GetTable",
        "glue:GetTables",
        "glue>DeleteTable",
        "glue:BatchDeleteTable",
        "glue:BatchCreatePartition",
        "glue:CreatePartition",
        "glue:UpdatePartition",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:BatchGetPartition",
        "glue>DeletePartition",
        "glue:BatchDeletePartition"
    ],
    "Resource": [
        "arn:aws:glue:us-east-2:123456789012:catalog",
```

```

        "arn:aws:glue:us-east-2:123456789012:database/*",
        "arn:aws:glue:us-east-2:123456789012:table/*",
        "arn:aws:glue:us-east-2:123456789012:userDefinedFunction/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "lakeformation:GetDataAccess"
    ],
    "Resource": [
        "*"
    ]
}
]
}

```

SNSPublish

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": [
                "arn:aws:sns:us-east-2:123456789012:StepFunctionsSample-
                AthenaIngestLargeDataset92bc4949-abf8-4a1e-9236-5b7c81b3efa3-SNSTopic-8Y5ZLI5AASXV"
            ]
        }
    ]
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Keep data up to date (Amazon Athena, Amazon S3, AWS Glue)

This sample project demonstrates how to query a target table to get current data with AWS Glue Catalog, then update it with new data from other sources using Amazon Athena.

In this project, the Step Functions state machine calls AWS Glue Catalog to verify if a target table exists in an Amazon S3 Bucket. If no table is found one, it will create a new table. Then, Step Functions runs an Athena query to add rows to the target table from a different data source: first querying the target table to get the most recent date, then querying the source table for more recent data and inserting it into the target table.

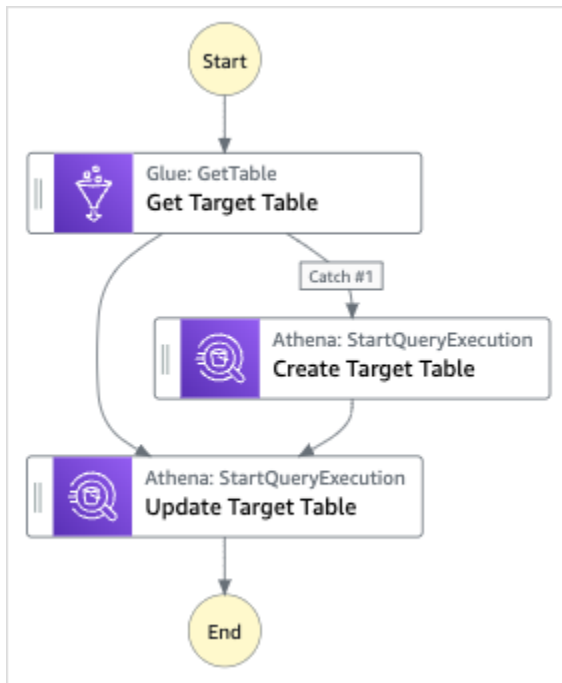
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Keep data up to date** in the search box, and then choose **Keep data up to date** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon S3 bucket
- Amazon Athena queries
- An AWS Glue Data Catalog call
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Keep data up to date** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions automatically generates a unique execution name.

Note

Step Functions allows you to create names for state machines, executions, activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

3. Choose **Start execution**.

4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with Amazon S3, AWS Glue, and Amazon Athena by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls Amazon S3, AWS Glue, and Amazon Athena by connecting to the Amazon Resource Name (ARN) in the `Resource` field, and by passing `Parameters` to the service API.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example demonstrates how to use Athena to query a target table to
get current data, then update it with new data from other sources.",
  "StartAt": "Get Target Table",
  "States": {
    "Get Target Table": {
      "Type": "Task",
      "Parameters": {
        "DatabaseName": "<GLUE_DATABASE_NAME>",
        "Name": "target"
      },
      "Catch": [
        {
          "ErrorEquals": [
            "Glue.EntityNotFoundException"
          ]
        }
      ]
    }
  }
}
```

```

        "Next": "Create Target Table"
    }
  ],
  "Resource": "arn:aws:states:::aws-sdk:glue:getTable",
  "Next": "Update Target Table"
},
"Create Target Table": {
  "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
  "Parameters": {
    "QueryString": "<ATHENA_QUERYSTRING>",
    "WorkGroup": "<ATHENA_WORKGROUP>"
  },
  "Type": "Task",
  "Next": "Update Target Table"
},
"Update Target Table": {
  "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
  "Parameters": {
    "QueryString": "<ATHENA_QUERYSTRING>",
    "WorkGroup": "<ATHENA_WORKGROUP>"
  },
  "Type": "Task",
  "End": true
}
}
}
}

```

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

AthenaStartQueryExecution

```

"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "athena:startQueryExecution",

```

```
        "athena:stopQueryExecution",
        "athena:getQueryExecution",
        "athena:getDataCatalog"
    ],
    "Resource": [
        "arn:aws:athena:us-east-2:123456789012:workgroup/stepfunctions-athena-
sample-project-workgroup-26ujlyawxg",
        "arn:aws:athena:us-east-2:123456789012:datacatalog/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:ListBucketMultipartUploads",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload",
        "s3:CreateBucket",
        "s3:PutObject"
    ],
    "Resource": [
        "arn:aws:s3::*:*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "glue:CreateDatabase",
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:UpdateDatabase",
        "glue>DeleteDatabase",
        "glue:CreateTable",
        "glue:UpdateTable",
        "glue:GetTable",
        "glue:GetTables",
        "glue>DeleteTable",
        "glue:BatchDeleteTable",
        "glue:BatchCreatePartition",
        "glue:CreatePartition",
        "glue:UpdatePartition",
        "glue:GetPartition",
```

```
        "glue:GetPartitions",
        "glue:BatchGetPartition",
        "glue>DeletePartition",
        "glue:BatchDeletePartition"
    ],
    "Resource": [
        "arn:aws::glue:us-east-2:123456789012:catalog",
        "arn:aws::glue:us-east-2:123456789012:database/*",
        "arn:aws::glue:us-east-2:123456789012:table/*",
        "arn:aws::glue:us-east-2:123456789012:userDefinedFunction/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "lakeformation:GetDataAccess"
    ],
    "Resource": [
        "*"
    ]
}
]
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Manage an Amazon EKS cluster

This sample project demonstrates how to use Step Functions and Amazon Elastic Kubernetes Service to create an Amazon EKS cluster with a node group, run a job on Amazon EKS, then examine the output. When finished, it removes the node groups and Amazon EKS cluster.

For more information about Step Functions and Step Functions service integrations, see the following:

- [Using AWS Step Functions with other services](#)
- [Call Amazon EKS with Step Functions](#)

Note

This sample project may incur charges.

For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see [Amazon EKS Pricing](#).

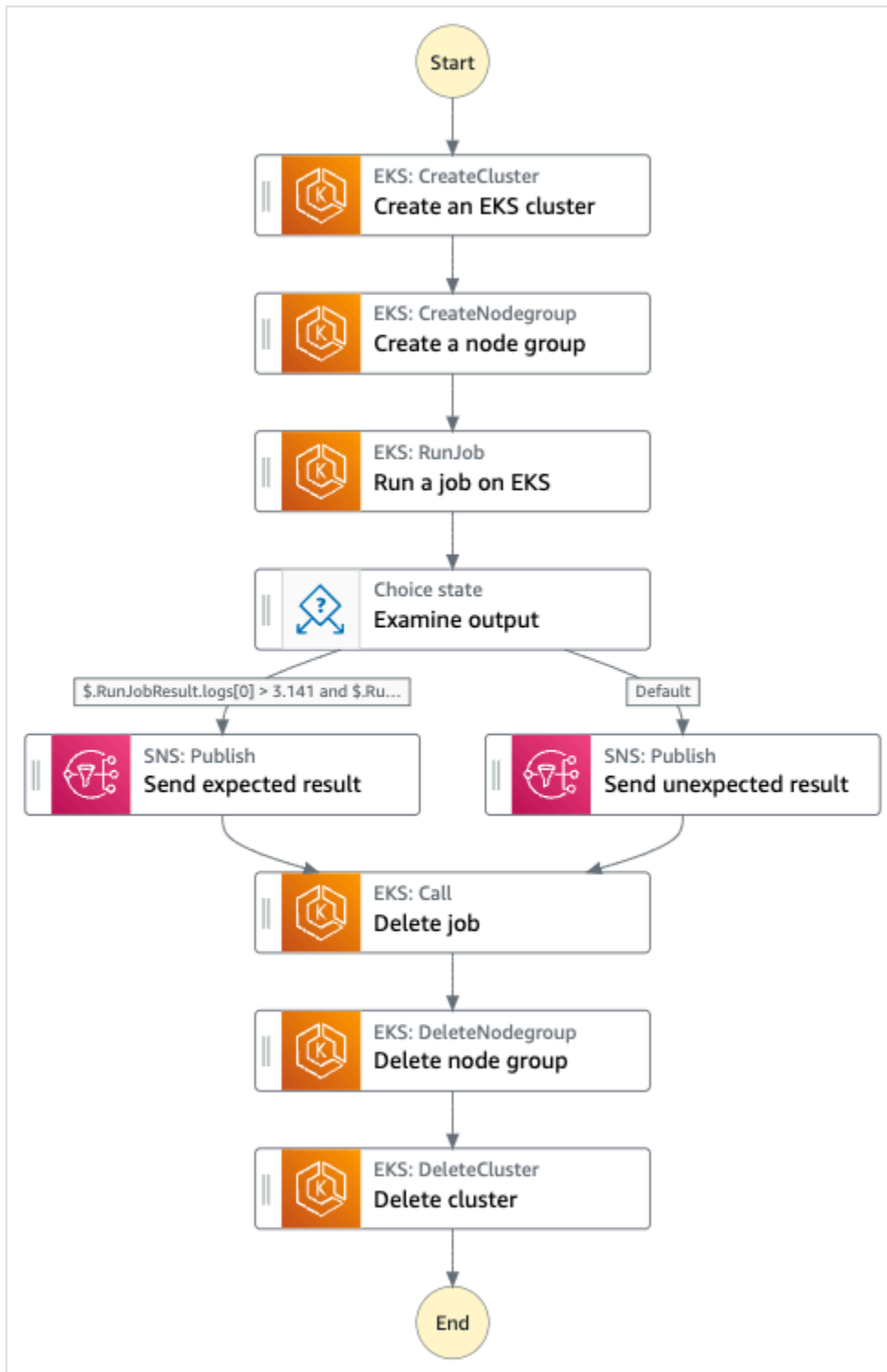
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Manage an EKS cluster** in the search box, and then choose **Manage an EKS cluster** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon Elastic Kubernetes Service cluster
- An Amazon SNS topic
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Manage an EKS cluster** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:

- If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with Amazon EKS by creating an Amazon EKS cluster and node group, and uses an SNS topic to return results.

Browse through this example state machine to see how Step Functions manages Amazon EKS clusters and node groups.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of the Amazon States Language for running Amazon EKS Cluster",
  "StartAt": "Create an EKS cluster",
  "States": {
    "Create an EKS cluster": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:createCluster.sync",
      "Parameters": {
        "Name": "ExampleCluster",
        "ResourcesVpcConfig": {
          "SubnetIds": [
            "subnet-0aacf887d9f00e6a7",
            "subnet-0e5fc41e7507194ab"
          ]
        },
        "RoleArn": "arn:aws:iam::111122223333:role/StepFunctionsSample-EKSClusterManag-
EKSServiceRole-ANPAJ2UCCR6DPCEXAMPLE"
      },
      "Retry": [{
        "ErrorEquals": [ "States.ALL" ],
        "IntervalSeconds": 30,
        "MaxAttempts": 2,
        "BackoffRate": 2
      }],
      "ResultPath": "$.eks",
      "Next": "Create a node group"
    },
    "Create a node group": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:createNodegroup.sync",
      "Parameters": {
        "ClusterName": "ExampleCluster",
```

```

    "NodegroupName": "ExampleNodegroup",
    "NodeRole": "arn:aws:iam::111122223333:role/StepFunctionsSample-EKSClusterMan-
NodeInstanceRole-ANPAJ2UCCR6DPCEXAMPLE",
    "Subnets": [
      "subnet-0aacf887d9f00e6a7",
      "subnet-0e5fc41e7507194ab"]
  },
  "Retry": [{
    "ErrorEquals": [ "States.ALL" ],
    "IntervalSeconds": 30,
    "MaxAttempts": 2,
    "BackoffRate": 2
  }],
  "ResultPath": "$.nodegroup",
  "Next": "Run a job on EKS"
},
"Run a job on EKS": {
  "Type": "Task",
  "Resource": "arn:aws:states:::eks:runJob.sync",
  "Parameters": {
    "ClusterName": "ExampleCluster",
    "CertificateAuthority.$": "$.eks.Cluster.CertificateAuthority.Data",
    "Endpoint.$": "$.eks.Cluster.Endpoint",
    "LogOptions": {
      "RetrieveLogs": true
    }
  },
  "Job": {
    "apiVersion": "batch/v1",
    "kind": "Job",
    "metadata": {
      "name": "example-job"
    }
  },
  "spec": {
    "backoffLimit": 0,
    "template": {
      "metadata": {
        "name": "example-job"
      }
    },
    "spec": {
      "containers": [
        {
          "name": "pi-20",
          "image": "perl",
          "command": [

```

```

        "perl"
      ],
      "args": [
        "-Mbignum=bpi",
        "-wle",
        "print '{ ' . '\"pi\": ' . bpi(20) . ' }';"
      ]
    }
  ],
  "restartPolicy": "Never"
}
}
}
},
"ResultSelector": {
  "status.$": "$.status",
  "logs.$": "$.logs..pi"
},
"ResultPath": "$.RunJobResult",
"Next": "Examine output"
},
"Examine output": {
  "Type": "Choice",
  "Choices": [
    {
      "And": [
        {
          "Variable": "$.RunJobResult.logs[0]",
          "NumericGreaterThan": 3.141
        },
        {
          "Variable": "$.RunJobResult.logs[0]",
          "NumericLessThan": 3.142
        }
      ]
    },
    {
      "Next": "Send expected result"
    }
  ],
  "Default": "Send unexpected result"
},
"Send expected result": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sns:publish",

```

```
    "Parameters": {
      "TopicArn": "arn:aws:sns:sa-east-1:111122223333:StepFunctionsSample-
EKSClusterManagement123456789012-SNSTopic-ANPAJ2UCCR6DPCEXAMPLE",
      "Message": {
        "Input.$": "States.Format('Saw expected value for pi: {}',
$.RunJobResult.logs[0])"
      }
    },
    "ResultPath": "$.SNSResult",
    "Next": "Delete job"
  },
  "Send unexpected result": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish",
    "Parameters": {
      "TopicArn": "arn:aws:sns:sa-east-1:111122223333:StepFunctionsSample-
EKSClusterManagement123456789012-SNSTopic-ANPAJ2UCCR6DPCEXAMPLE",
      "Message": {
        "Input.$": "States.Format('Saw unexpected value for pi: {}',
$.RunJobResult.logs[0])"
      }
    },
    "ResultPath": "$.SNSResult",
    "Next": "Delete job"
  },
  "Delete job": {
    "Type": "Task",
    "Resource": "arn:aws:states:::eks:call",
    "Parameters": {
      "ClusterName": "ExampleCluster",
      "CertificateAuthority.$": "$.eks.Cluster.CertificateAuthority.Data",
      "Endpoint.$": "$.eks.Cluster.Endpoint",
      "Method": "DELETE",
      "Path": "/apis/batch/v1/namespaces/default/jobs/example-job"
    },
    "ResultSelector": {
      "status.$": "$.ResponseBody.status"
    },
    "ResultPath": "$.DeleteJobResult",
    "Next": "Delete node group"
  },
  "Delete node group": {
    "Type": "Task",
    "Resource": "arn:aws:states:::eks:deleteNodegroup.sync",
```

```
    "Parameters": {
      "ClusterName": "ExampleCluster",
      "NodegroupName": "ExampleNodegroup"
    },
    "Next": "Delete cluster"
  },
  "Delete cluster": {
    "Type": "Task",
    "Resource": "arn:aws:states:::eks:deleteCluster.sync",
    "Parameters": {
      "Name": "ExampleCluster"
    },
    "End": true
  }
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "eks:CreateCluster"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "eks:DescribeCluster",
        "eks>DeleteCluster"
      ],
    }
  ]
}
```

```

        "Resource": "arn:aws:eks:sa-east-1:111122223333:cluster/*"
    },
    {
        "Effect": "Allow",
        "Action": "iam:PassRole",
        "Resource": [
            "arn:aws:iam::111122223333:role/StepFunctionsSample-EKSClusterManag-
EKSServiceRole-ANPAJ2UCCR6DPCEXAMPLE"
        ],
        "Condition": {
            "StringEquals": {
                "iam:PassedToService": "eks.amazonaws.com"
            }
        }
    }
]
}

```

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": [
                "arn:aws:sns:sa-east-1:111122223333:StepFunctionsSample-
EKSClusterManagement123456789012-SNSTopic-ANPAJ2UCCR6DPCEXAMPLE"
            ]
        }
    ]
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Make a call to API Gateway

This sample project demonstrates how to use Step Functions to make a call to API Gateway and checks whether the call succeeded.

For more information about API Gateway and Step Functions service integrations, see the following:

- [Using AWS Step Functions with other services](#)
- [Call API Gateway with Step Functions](#)

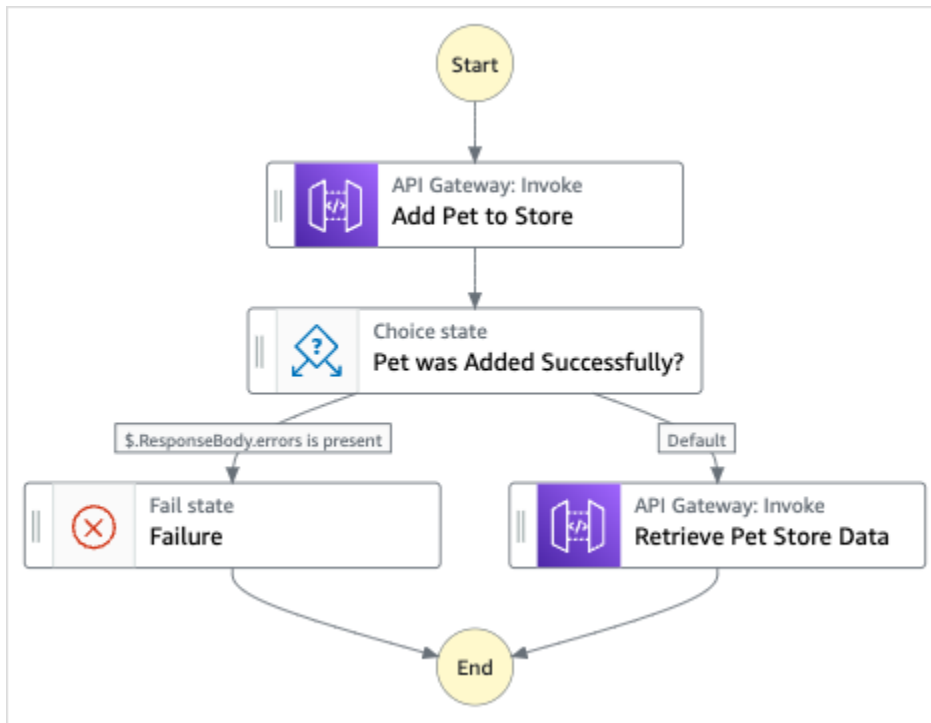
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Make a call to API Gateway** in the search box, and then choose **Make a call to API Gateway** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon API Gateway REST API that is called by the state machine.
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Make a call to API Gateway** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with API Gateway by calling the API Gateway REST API and passing any necessary parameters.

Browse through this example state machine to see how Step Functions interacts with API Gateway.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "Calling APIGW REST Endpoint",
  "StartAt": "Add Pet to Store",
  "States": {
    "Add Pet to Store": {
      "Type": "Task",
      "Resource": "arn:aws:states:::apigateway:invoke",
      "Parameters": {
        "ApiEndpoint": "<POST_PETS_API_ENDPOINT>",
        "Method": "POST",
        "Stage": "default",
        "Path": "pets",
```

```

    "RequestBody.$": "$.NewPet",
    "AuthType": "IAM_ROLE"
  },
  "ResultSelector": {
    "ResponseBody.$": "$.ResponseBody"
  },
  "Next": "Pet was Added Successfully?"
},
"Pet was Added Successfully?": {
  "Type": "Choice",
  "Choices": [
    {
      "Variable": "$.ResponseBody.errors",
      "IsPresent": true,
      "Next": "Failure"
    }
  ],
  "Default": "Retrieve Pet Store Data"
},
"Failure": {
  "Type": "Fail"
},
"Retrieve Pet Store Data": {
  "Type": "Task",
  "Resource": "arn:aws:states:::apigateway:invoke",
  "Parameters": {
    "ApiEndpoint": "<GET_PETS_API_ENDPOINT>",
    "Method": "GET",
    "Stage": "default",
    "Path": "pets",
    "AuthType": "IAM_ROLE"
  },
  "ResultSelector": {
    "Pets.$": "$.ResponseBody"
  },
  "ResultPath": "$.ExistingPets",
  "End": true
}
}
}
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-west-1:111122223333:c8hqe4kdg5/default/GET/pets",
        "arn:aws:execute-api:us-west-1:111122223333:c8hqe4kdg5/default/POST/pets"
      ],
      "Effect": "Allow"
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Call a microservice running on Fargate using API Gateway integration

This sample project demonstrates how to use Step Functions to make a call to API Gateway in order to interact with a service on AWS Fargate, and also to check whether the call succeeded.

For more information about API Gateway and Step Functions service integrations, see the following:

- [Using AWS Step Functions with other services](#)
- [Call API Gateway with Step Functions](#)

Note

This sample project may incur charges. For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see [Pricing](#).

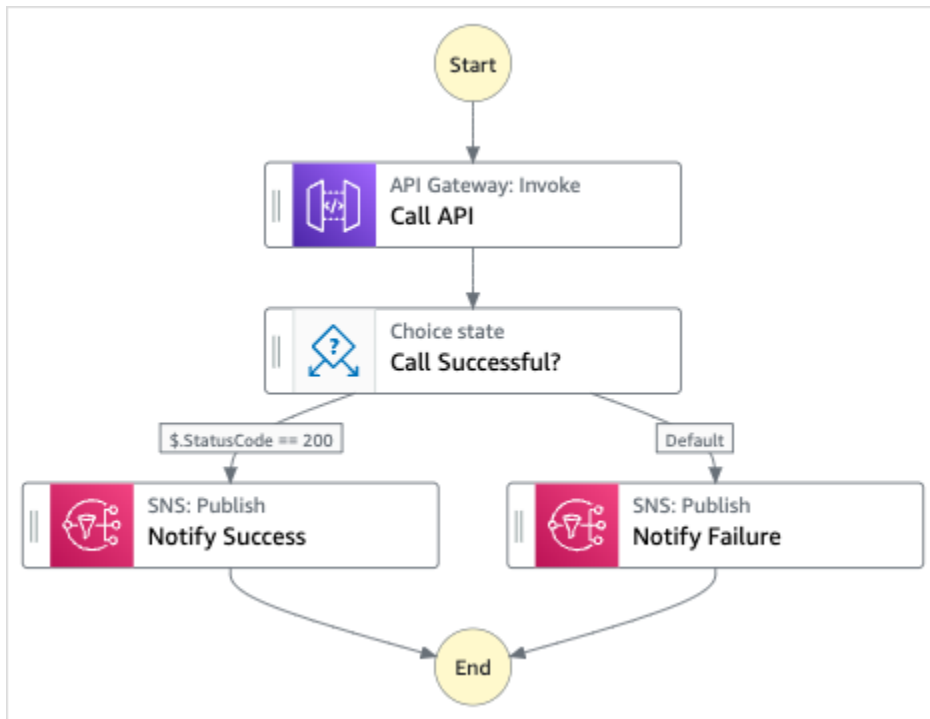
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Call a microservice with API Gateway** in the search box, and then choose **Call a microservice with API Gateway** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon API Gateway HTTP API that is called by the state machine.
- An Amazon API Gateway Amazon VPC Link.
- An Amazon Virtual Private Cloud.
- An Application Load Balancer.
- A Fargate cluster.
- An Amazon SNS topic
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles
- Several additional services that are required to enable these resources to work together.

The following image shows the workflow graph for the **Call a microservice with API Gateway** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

 **Note**

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with API Gateway by calling an API Gateway HTTP API that is connected to a service on Fargate. This is hosted on a private subnet, and accessed through a private application load balancer.

Browse through this example state machine to see how Step Functions interacts with API Gateway and returns results.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "Calling APIGW HTTP Endpoint",
  "StartAt": "Call API",
  "States": {
    "Call API": {
      "Type": "Task",
      "Resource": "arn:<PARTITION>:states:::apigateway:invoke",
      "Parameters": {
        "ApiEndpoint": "<API_ENDPOINT>",
        "Method": "GET",
```

```
    "AuthType": "IAM_ROLE"
  },
  "Next": "Call Successful?"
},
"Call Successful?": {
  "Type": "Choice",
  "Choices": [
    {
      "Variable": "$.StatusCode",
      "NumericEquals": 200,
      "Next": "Notify Success"
    }
  ],
  "Default": "Notify Failure"
},
"Notify Success": {
  "Type": "Task",
  "Resource": "arn:<PARTITION>:states:::sns:publish",
  "Parameters": {
    "Message": "Call was successful",
    "TopicArn": "<SNS_TOPIC_ARN>"
  },
  "End": true
},
"Notify Failure": {
  "Type": "Task",
  "Resource": "arn:<PARTITION>:states:::sns:publish",
  "Parameters": {
    "Message": "Call was not successful",
    "TopicArn": "<SNS_TOPIC_ARN>"
  },
  "End": true
}
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "sns:Publish"
      ],
      "Resource": [
        "arn:aws:sns:us-east-1:111122223333:apigw-ecs-sample-2000-
        SNSTopic-444455556666"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:111122223333:444444444444/*/*/*/*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

```
{
  "Statement": [
    {
      "Action": [
        "ec2:AttachNetworkInterface",
        "ec2:CreateNetworkInterface",
        "ec2:CreateNetworkInterfacePermission",
        "ec2>DeleteNetworkInterface",
        "ec2>DeleteNetworkInterfacePermission",
        "ec2:Describe*",
        "ec2:DetachNetworkInterface",
```

```

        "elasticloadbalancing:DeregisterInstancesFromLoadBalancer",
        "elasticloadbalancing:DeregisterTargets",
        "elasticloadbalancing:Describe*",
        "elasticloadbalancing:RegisterInstancesWithLoadBalancer",
        "elasticloadbalancing:RegisterTargets"
    ],
    "Resource": "*",
    "Effect": "Allow"
}
]
}

```

```

{
  "Statement": [
    {
      "Action": [
        "ecr:GetAuthorizationToken",
        "ecr:BatchCheckLayerAvailability",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*",
      "Effect": "Allow"
    }
  ]
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Send a custom event to EventBridge

This sample project demonstrates how to use Step Functions to send a custom event to an event bus that matches a rule with multiple targets (Amazon EventBridge, AWS Lambda, Amazon Simple Notification Service, Amazon Simple Queue Service).

For more information about Step Functions and Step Functions service integrations, see the following:

- [Using AWS Step Functions with other services](#)

- [Call EventBridge with Step Functions](#)

Note

This sample project may incur charges.

For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see [EventBridge Pricing](#).

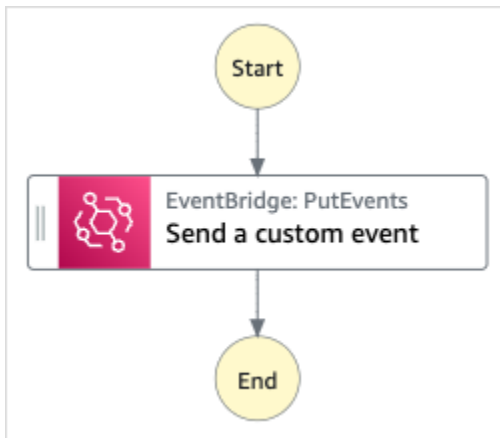
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Send a custom event to EventBridge** in the search box, and then choose **Send a custom event to EventBridge** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon EventBridge event
- An Amazon SNS topic
- An Amazon SQS queue
- A Lambda function
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Send a custom event to EventBridge** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

 **Note**

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with EventBridge by sending a custom event to an EventBridge event bus. The event sent to the event bus matches an EventBridge rule that triggers a Lambda function that sends messages to an Amazon SNS topic and an Amazon SQS queue.

Browse through this example state machine to see how Step Functions manages EventBridge.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of the Amazon States Language for sending a custom event to
Amazon EventBridge",
  "StartAt": "Send a custom event",
  "States": {
    "Send a custom event": {
      "Resource": "arn:<PARTITION>:states:::events:putEvents",
      "Type": "Task",
      "Parameters": {
        "Entries": [{
          "Detail": {
            "Message": "Hello from Step Functions!"
          }
        }
      ]
    }
  }
}
```



```
    },
    "DetailType": "MessageFromStepFunctions",
    "EventBusName": "<EVENT_BUS_NAME>",
    "Source": "my.statemachine"
  ]
},
"End": true
}
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "events:PutEvents"
      ],
      "Resource": [
        "arn:aws:events:us-east-1:1234567890:event-bus/stepfunctions-
sampleproject-eventbus"
      ],
      "Effect": "Allow"
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Invoke Synchronous Express Workflows

This sample project demonstrates how to invoke Synchronous Express Workflows through Amazon API Gateway to manage an employee database.

In this project, Step Functions uses API Gateway endpoints to start Step Functions Synchronous Express Workflows. These then use DynamoDB to search for, add, and remove employees in an employee database.

For more information about Step Functions Synchronous Express Workflows, see [Synchronous and Asynchronous Express Workflows](#).

Note

This sample project may incur charges.

For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see [Step Functions Pricing](#).

Step 1: Create the state machine and provision resources

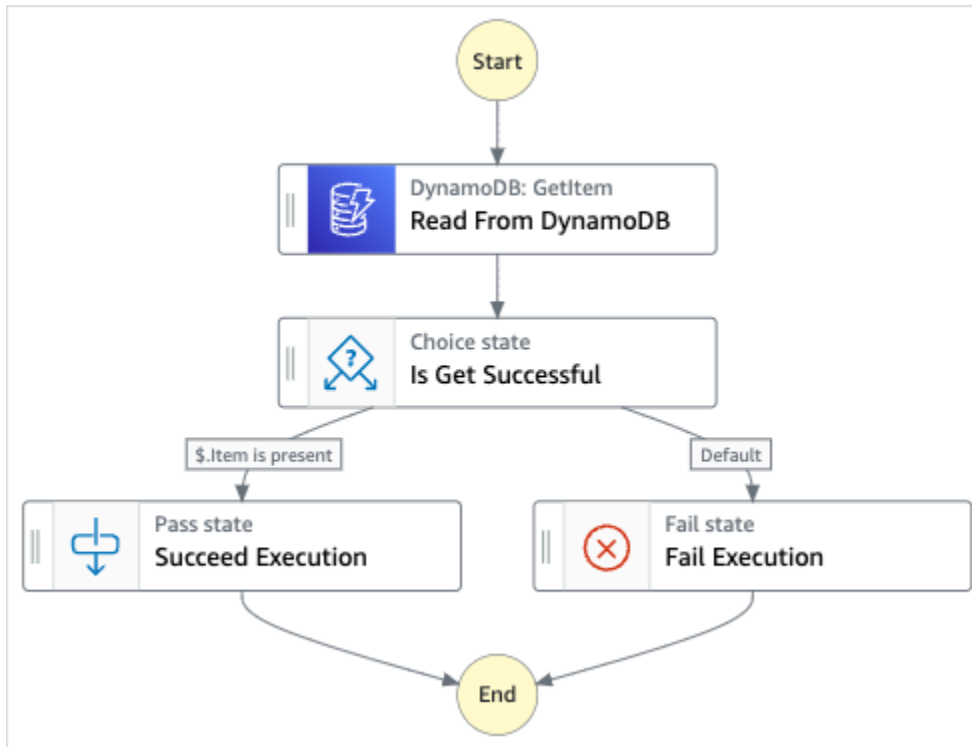
1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Invoke Synchronous Express Workflows through API Gateway** in the search box, and then choose **Invoke Synchronous Express Workflows through API Gateway** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon API Gateway HTTPS API that is called by a state machine.
- An Amazon DynamoDB table.
- Three AWS Step Functions state machines.

- Related AWS Identity and Access Management (IAM) roles.

The following image shows the workflow graph for the **Invoke Synchronous Express Workflows through API Gateway** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with API Gateway and DynamoDB by using API Gateway to invoke a Synchronous Express Workflow, which then updates or reads from the employee database using DynamoDB.

Browse through this example state machine to see how Step Functions reads from DynamoDB to retrieve employee information.

To understand more about how to invoke Step Functions using API Gateway, see the following.

- [Call API Gateway with Step Functions](#)

- [How to invoke a private Gateway](#) in the API Gateway Developer Guide.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "This state machine returns an employee entry from DynamoDB",
  "StartAt": "Read From DynamoDB",
  "States": {
    "Read From DynamoDB": {
      "Type": "Task",
      "Resource": "arn:aws:states:::dynamodb:getItem",
      "Parameters": {
        "TableName": "StepFunctionsSample-
SynchronousExpressWorkflowAKIAIOSFODNN7EXAMPLE-DynamoDBTable-ANPAJ2UCCR6DPCEXAMPLE",
        "Key": {
          "EmployeeId": {"S.$": "$.employee"}
        }
      },
      "Retry": [
        {
          "ErrorEquals": [
            "DynamoDB.AmazonDynamoDBException"
          ],
          "IntervalSeconds": 3,
          "MaxAttempts": 2,
          "BackoffRate": 1.5
        }
      ],
      "Next": "Is Get Successful"
    },
    "Is Get Successful": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.Item",
          "IsPresent": true,
          "Next": "Succeed Execution"
        }
      ],
      "Default": "Fail Execution"
    },
    "Succeed Execution": {
```

```
    "Type": "Pass",
    "Parameters" : {
      "employee.$": "$.Item.EmployeeId.S",
      "jobTitle.$": "$.Item.JobTitle.S"
    },
    "End": true
  },
  "Fail Execution": {
    "Type": "Fail",
    "Error": "EmployeeDoesNotExist"
  }
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Examples

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogDelivery",
        "logs:GetLogDelivery",
        "logs:UpdateLogDelivery",
        "logs>DeleteLogDelivery",
        "logs>ListLogDeliveries",
        "logs:PutResourcePolicy",
        "logs:DescribeResourcePolicies",
        "logs:DescribeLogGroups"
      ],
      "Resource": "*"
    }
  ]
}
```

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb>DeleteItem"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:111122223333:table/Write"
      ]
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Run ETL/ELT workflows using Amazon Redshift (Lambda, Amazon Redshift Data API)

This sample project demonstrates how to use Step Functions and the Amazon Redshift Data API to run an ETL/ELT workflow that loads data into the Amazon Redshift data warehouse.

In this project, Step Functions uses an AWS Lambda function and the Amazon Redshift Data API to create the required database objects and to generate a set of example data, then executes two jobs in parallel that perform loading dimension tables, followed by a fact table. Once both dimension load jobs end successfully, Step Functions executes the load job for the fact table, runs the validation job, then pauses the Amazon Redshift cluster.

Note

You can modify the ETL logic to receive data from other sources such as Amazon S3, which can use the [COPY](#) command to copy data from Amazon S3 to an Amazon Redshift table.

For more information about Amazon Redshift and Step Functions service integrations, see the following:

- [Using AWS Step Functions with other services](#)
- [Using the Amazon Redshift Data API](#)
- [Amazon Redshift Data API service](#)
- [Creating a Step Functions state machine that uses Lambda](#)
- IAM policies for:
 - [IAM policies for AWS Lambda](#)
 - [Authorizing access to the Amazon Redshift Data API](#)

Note

This sample project may incur charges.

For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see [AWS Step Functions pricing](#).

Step 1: Create the state machine and provision resources

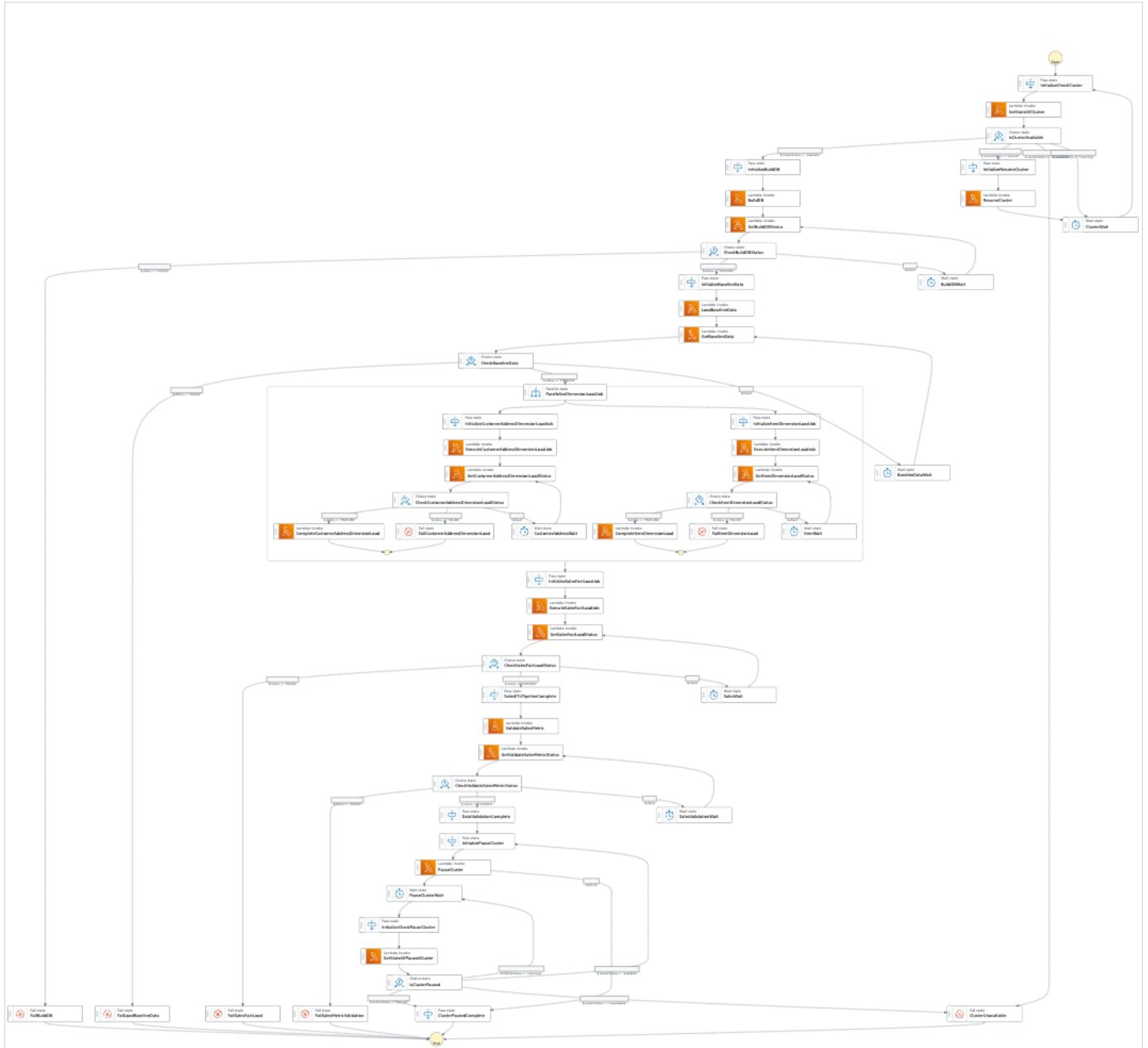
1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **ETL job in Amazon Redshift** in the search box, and then choose **ETL job in Amazon Redshift** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An Amazon Redshift cluster
- Two Lambda functions
- An Amazon Redshift schema

- Five Amazon Redshift tables
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles.

The following image shows the workflow graph for the **ETL job in Amazon Redshift** sample project:



5. Choose **Use template** to continue with your selection.

6. Do one of the following:

- If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

 **Important**

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

 **Tip**

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with AWS Lambda by passing the ETL logic as the `InputPath` directly to those resources and being executed asynchronously using Amazon Redshift Data API.

Browse through this example state machine to see how Step Functions controls AWS Lambda and the Amazon Redshift Data API.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "A simple ETL workflow for loading dimension and fact tables",
  "StartAt": "InitializeCheckCluster",
  "States": {
    "InitializeCheckCluster": {
      "Type": "Pass",
      "Next": "GetStateOfCluster",
      "Result": {
        "input": {
          "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
          "operation": "status"
        }
      }
    },
    "GetStateOfCluster": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftOperations-AKIAIOSFODNN7EXAMPLE",
      "TimeoutSeconds": 180,
      "HeartbeatSeconds": 60,
      "Next": "IsClusterAvailable",
      "InputPath": "$",
      "ResultPath": "$.clusterStatus"
    },
    "IsClusterAvailable": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.clusterStatus",
          "StringEquals": "available",
          "Next": "InitializeBuildDB"
        }
      ]
    }
  }
}
```

```

    },
    {
      "Variable": "$.clusterStatus",
      "StringEquals": "paused",
      "Next": "InitializeResumeCluster"
    },
    {
      "Variable": "$.clusterStatus",
      "StringEquals": "unavailable",
      "Next": "ClusterUnavailable"
    },
    {
      "Variable": "$.clusterStatus",
      "StringEquals": "resuming",
      "Next": "ClusterWait"
    }
  ]
},
"ClusterWait": {
  "Type": "Wait",
  "Seconds": 720,
  "Next": "InitializeCheckCluster"
},
"InitializeResumeCluster": {
  "Type": "Pass",
  "Next": "ResumeCluster",
  "Result": {
    "input": {
      "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
      "operation": "resume"
    }
  }
},
"ResumeCluster": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftOperations-AKIAIOSFODNN7EXAMPLE",
  "TimeoutSeconds": 180,
  "HeartbeatSeconds": 60,
  "Next": "ClusterWait",
  "InputPath": "$",
  "ResultPath": "$"
},
"InitializeBuildDB": {

```

```
"Type": "Pass",
"Next": "BuildDB",
"Result": {
  "input": {
    "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
    "redshift_database": "dev",
    "redshift_user": "awsuser",
    "redshift_schema": "tpcds",
    "action": "build_database",
    "sql_statement": [
      "create schema if not exists {0} authorization {1};",
      "create table if not exists {0}.customer",
      "(c_customer_sk          int4          not null encode az64",
      ",c_customer_id         char(16) not null encode zstd",
      ",c_current_addr_sk      int4                               encode az64",
      ",c_first_name           char(20)           encode zstd",
      ",c_last_name             char(30)           encode zstd",
      ",primary key (c_customer_sk)",
      ") distkey(c_customer_sk);",
      "--",
      "create table if not exists {0}.customer_address",
      "(ca_address_sk         int4          not null encode az64",
      ",ca_address_id         char(16) not null encode zstd",
      ",ca_state              char(2)           encode zstd",
      ",ca_zip                char(10)          encode zstd",
      ",ca_country            varchar(20)       encode zstd",
      ",primary key (ca_address_sk)",
      ") distkey(ca_address_sk);",
      "--",
      "create table if not exists {0}.date_dim",
      "(d_date_sk             integer not null encode az64",
      ",d_date_id            char(16) not null encode zstd",
      ",d_date               date              encode az64",
      ",d_day_name           char(9)           encode zstd",
      ",primary key (d_date_sk)",
      ") diststyle all;",
      "--",
      "create table if not exists {0}.item",
      "(i_item_sk            int4          not null encode az64",
      ",i_item_id           char(16) not null encode zstd",
      ",i_rec_start_date    date              encode az64",
      ",i_rec_end_date      date              encode az64",
      ",i_current_price     numeric(7,2)      encode az64",
      ",i_category          char(50)         encode zstd",
```

```

        ",i_product_name  char(50)          encode zstd",
        ",primary key (i_item_sk)",
        ") distkey(i_item_sk) sortkey(i_category);",
        "--",
        "create table if not exists {0}.store_sales",
        "(ss_sold_date_sk      int4",
        ",ss_item_sk           int4 not null encode az64",
        ",ss_customer_sk        int4          encode az64",
        ",ss_addr_sk             int4          encode az64",
        ",ss_store_sk           int4          encode az64",
        ",ss_ticket_number       int8 not null encode az64",
        ",ss_quantity            int4          encode az64",
        ",ss_net_paid             numeric(7,2) encode az64",
        ",ss_net_profit          numeric(7,2) encode az64",
        ",primary key (ss_item_sk, ss_ticket_number)",
        ") distkey(ss_item_sk) sortkey(ss_sold_date_sk);"
    ]
}
},
"BuildDB": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-
RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
    "TimeoutSeconds": 180,
    "HeartbeatSeconds": 60,
    "Next": "GetBuildDBStatus",
    "InputPath": "$",
    "ResultPath": "$"
},
"GetBuildDBStatus": {
    "Type": "Task",
    "Next": "CheckBuildDBStatus",
    "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-
RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
    "TimeoutSeconds": 180,
    "HeartbeatSeconds": 60,
    "InputPath": "$",
    "ResultPath": "$.status"
},
"CheckBuildDBStatus": {
    "Type": "Choice",
    "Choices": [
        {

```



```

        "Variable": "$.status",
        "StringEquals": "FAILED",
        "Next": "FailBuildDB"
    },
    {
        "Variable": "$.status",
        "StringEquals": "FINISHED",
        "Next": "InitializeBaselineData"
    }
],
"Default": "BuildDBWait"
},
"BuildDBWait": {
    "Type": "Wait",
    "Seconds": 15,
    "Next": "GetBuildDBStatus"
},
"FailBuildDB": {
    "Type": "Fail",
    "Cause": "Database Build Failed",
    "Error": "Error"
},
"InitializeBaselineData": {
    "Type": "Pass",
    "Next": "LoadBaselineData",
    "Result": {
        "input": {
            "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
            "redshift_database": "dev",
            "redshift_user": "awsuser",
            "redshift_schema": "tpcds",
            "action": "load_baseline_data",
            "sql_statement": [
                "begin transaction;",
                "truncate table {0}.customer;",
                "insert into {0}.customer
(c_customer_sk,c_customer_id,c_current_addr_sk,c_first_name,c_last_name)",
                "values",
                "(7550,'AAAAAAAAOHNBAAAA',9264662,'Michelle','Deaton'),",
                "(37079,'AAAAAAAAAHNAJAAAA',13971208,'Michael','Simms'),",
                "(40626,'AAAAAAACLOJAAAA',1959255,'Susan','Ryder'),",
                "(2142876,'AAAAAAAAMJCLACAA',7644556,'Justin','Brown');",
                "analyze {0}.customer;",
                "--",

```

```

        "truncate table {0}.customer_address;",
        "insert into {0}.customer_address
(ca_address_sk,ca_address_id,ca_state,ca_zip,ca_country)",
        "values",
        "(13971208,'AAAAAAAAAIAPCFNAA','NE','63451','United States'),",
        "(7644556,'AAAAAAAAAMIFKEHAA','SD','58883','United States'),",
        "(9264662,'AAAAAAAAGBOFNIAA','CA','99310','United States');",
        "analyze {0}.customer_address;",
        "--",
        "truncate table {0}.item;",
        "insert into {0}.item
(i_item_sk,i_item_id,i_rec_start_date,i_rec_end_date,i_current_price,i_category,i_product_name)",
        "values",

"(3417,'AAAAAAAIFNAAAA','1997-10-27',NULL,14.29,'Electronics','ationoughtesepri
'),",
        "(9615,'AAAAAAA0IFCAAAA','1997-10-27',NULL,9.68,'Home','antioughtcallyn
st'),",
        "(3693,'AAAAAAAAMGOAAAA','2001-03-12',NULL,2.10,'Men','prin
stcallypri'),",

"(3630,'AAAAAAAAMCOAAAA','2001-10-27',NULL,2.95,'Electronics','barpricallypri'),",

"(16506,'AAAAAAAIIHAEAAAA','2001-10-27',NULL,3.85,'Home','callybaranticallyought'),",

"(7866,'AAAAAAAIILOBAAAA','2001-10-27',NULL,12.60,'Jewelry','callycallyeingation');",
        "--",
        "analyze {0}.item;",
        "truncate table {0}.date_dim;",
        "insert into {0}.date_dim (d_date_sk,d_date_id,d_date,d_day_name)",
        "values",
        "(2450521,'AAAAAAAJFEGFCAA','1997-03-13','Thursday'),",
        "(2450749,'AAAAAAAANDFGCAA','1997-10-27','Monday'),",
        "(2451251,'AAAAAAAADHGFCAA','1999-03-13','Saturday'),",
        "(2451252,'AAAAAAAEDHGFCAA','1999-03-14','Sunday'),",
        "(2451981,'AAAAAAAANAKGFCAA','2001-03-12','Monday'),",
        "(2451982,'AAAAAAA0AKGFCAA','2001-03-13','Tuesday'),",
        "(2452210,'AAAAAAAACPFGCAA','2001-10-27','Saturday'),",
        "(2452641,'AAAAAAAABKMGFCAA','2003-01-01','Wednesday'),",
        "(2452642,'AAAAAAAACKMGFCAA','2003-01-02','Thursday');",
        "--",
        "analyze {0}.date_dim;",
        "-- commit and End transaction",
        "commit;"

```

```
        "end transaction;"
    ]
}
},
"LoadBaselineData": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-
RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
    "TimeoutSeconds": 180,
    "HeartbeatSeconds": 60,
    "Next": "GetBaselineData",
    "InputPath": "$",
    "ResultPath": "$"
},
"GetBaselineData": {
    "Type": "Task",
    "Next": "CheckBaselineData",
    "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-
RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
    "TimeoutSeconds": 180,
    "HeartbeatSeconds": 60,
    "InputPath": "$",
    "ResultPath": "$.status"
},
"CheckBaselineData": {
    "Type": "Choice",
    "Choices": [
        {
            "Variable": "$.status",
            "StringEquals": "FAILED",
            "Next": "FailLoadBaselineData"
        },
        {
            "Variable": "$.status",
            "StringEquals": "FINISHED",
            "Next": "ParallelizeDimensionLoadJob"
        }
    ],
    "Default": "BaselineDataWait"
},
"BaselineDataWait": {
    "Type": "Wait",
    "Seconds": 20,
```

```

    "Next": "GetBaselineData"
  },
  "FailLoadBaselineData": {
    "Type": "Fail",
    "Cause": "Load Baseline Data Failed",
    "Error": "Error"
  },
  "ParallelizeDimensionLoadJob": {
    "Type": "Parallel",
    "Next": "InitializeSalesFactLoadJob",
    "ResultPath": "$.status",
    "Branches": [
      {
        "StartAt": "InitializeCustomerAddressDimensionLoadJob",
        "States": {
          "InitializeCustomerAddressDimensionLoadJob": {
            "Type": "Pass",
            "Next": "ExecuteCustomerAddressDimensionLoadJob",
            "Result": {
              "input": {
                "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
                "redshift_database": "dev",
                "redshift_user": "awsuser",
                "redshift_schema": "tpcds",
                "action": "load_customer_address",
                "sql_statement": [
                  "begin transaction;",
                  "/* Create a staging table to hold the input data. Staging table is
created with BACKUP NO option for faster inserts and also data temporary */",
                  "drop table if exists {0}.stg_customer_address;",
                  "create table if not exists {0}.stg_customer_address",
                  "(ca_address_id    varchar(16)  encode zstd",
                  ",ca_state        varchar(2)   encode zstd",
                  ",ca_zip            varchar(10) encode zstd",
                  ",ca_country       varchar(20)  encode zstd",
                  ")\"",
                  "backup no",
                  "diststyle even;",
                  "/* Ingest data from source */",
                  "insert into {0}.stg_customer_address
(ca_address_id,ca_state,ca_zip,ca_country)",
                  "values",
                  "('AAAAAAACFBBAAAA','NE','','United States'),",
                  "('AAAAAAAAGAEFAAAA','NE','61749','United States'),",

```



```

        "TimeoutSeconds": 180,
        "HeartbeatSeconds": 60,
        "Next": "GetCustomerAddressDimensionLoadStatus",
        "InputPath": "$",
        "ResultPath": "$"
    },
    "GetCustomerAddressDimensionLoadStatus": {
        "Type": "Task",
        "Next": "CheckCustomerAddressDimensionLoadStatus",
        "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-
RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
        "TimeoutSeconds": 180,
        "HeartbeatSeconds": 60,
        "InputPath": "$",
        "ResultPath": "$.status"
    },
    "CheckCustomerAddressDimensionLoadStatus": {
        "Type": "Choice",
        "Choices": [
            {
                "Variable": "$.status",
                "StringEquals": "FAILED",
                "Next": "FailCustomerAddressDimensionLoad"
            },
            {
                "Variable": "$.status",
                "StringEquals": "FINISHED",
                "Next": "CompleteCustomerAddressDimensionLoad"
            }
        ],
        "Default": "CustomerAddressWait"
    },
    "CustomerAddressWait": {
        "Type": "Wait",
        "Seconds": 5,
        "Next": "GetCustomerAddressDimensionLoadStatus"
    },
    "CompleteCustomerAddressDimensionLoad": {
        "Type": "Task",
        "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-
RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
        "TimeoutSeconds": 180,
        "HeartbeatSeconds": 60,
        "End": true
    }

```

```

    },
    "FailCustomerAddressDimensionLoad": {
      "Type": "Fail",
      "Cause": "ETL Workflow Failed",
      "Error": "Error"
    }
  }
},
{
  "StartAt": "InitializeItemDimensionLoadJob",
  "States": {
    "InitializeItemDimensionLoadJob": {
      "Type": "Pass",
      "Next": "ExecuteItemDimensionLoadJob",
      "Result": {
        "input": {
          "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
          "redshift_database": "dev",
          "redshift_user": "awsuser",
          "redshift_schema": "tpcds",
          "action": "load_item",
          "sql_statement": [
            "begin transaction;",
            "/* Create a staging table to hold the input data. Staging table is
created with BACKUP NO option for faster inserts and also data temporary */",
            "drop table if exists {0}.stg_item;",
            "create table if not exists {0}.stg_item",
            "(i_item_id          varchar(16) encode zstd",
            ",i_rec_start_date  date encode zstd",
            ",i_rec_end_date      date encode zstd",
            ",i_current_price      numeric(7,2) encode zstd",
            ",i_category           varchar(50) encode zstd",
            ",i_product_name       varchar(50) encode zstd",
            ")",
            "backup no",
            "diststyle even;",
            "/* Ingest data from source */",
            "insert into {0}.stg_item",

            "(i_item_id,i_rec_start_date,i_rec_end_date,i_current_price,i_category,i_product_name)",
            "values",

            "('AAAAAAAAABJBAAAA', '2000-10-27', NULL, 4.10, 'Books', 'ationoughtesecally')",

```

```

                "('AAAAAAAAAOPKBAAAA', '2001-10-27', NULL, 4.22, 'Books', 'ableoughtn
stcally'),",
                "('AAAAAAAAAHGPAAAAA', '1997-10-27', NULL, 29.30, 'Books', 'priesen
stpri'),",

                "('AAAAAAAAICMAAAAA', '2001-10-27', NULL, 1.93, 'Books', 'eseoughtoughtpri'),",

                "('AAAAAAAAAGPGBAAAA', '2001-10-27', NULL, 9.96, 'Books', 'bareingeinganti'),",
                "('AAAAAAAAANBEBAAAA', '1997-10-27', NULL, 2.25, 'Music', 'n
steseoughtanti'),",

                "('AAAAAAAAACLAAAAAA', '2001-10-27', NULL, 1.71, 'Home', 'bareingought'),",

                "('AAAAAAAAOBBDAAAA', '2001-10-27', NULL, 5.55, 'Books', 'callyationantiablight');",
                "/"
*****
                "** Type 2 is maintained for i_current_price column.",
                "** Update all attributes for the item when the price is not
changed",

                "** Sunset existing active item record with current i_rec_end_date
and insert a new record when the price does not match",

*****
                "update {0}.item",
                "  set i_category = stg_item.i_category,",
                "      i_product_name = stg_item.i_product_name",
                "  from {0}.stg_item",
                "  where item.i_item_id = stg_item.i_item_id",
                "    and item.i_rec_end_date is null",
                "    and item.i_current_price = stg_item.i_current_price;",
                "insert into {0}.item",
                "(i_item_sk",
                ",i_item_id",
                ",i_rec_start_date",
                ",i_rec_end_date",
                ",i_current_price",
                ",i_category",
                ",i_product_name",
                ")",
                "with max_item_sk as",
                "(select max(i_item_sk) max_item_sk",
                "  from {0}.item)",
                "select row_number() over (order by stg_item.i_item_id) +
max_item_sk as i_item_sk",

```



```

        "      ,stg_item.i_item_id",
        "      ,trunc(sysdate) as i_rec_start_date",
        "      ,null as i_rec_end_date",
        "      ,stg_item.i_current_price",
        "      ,stg_item.i_category",
        "      ,stg_item.i_product_name",
        "    from {0}.stg_item, {0}.item, max_item_sk",
        "    where item.i_item_id = stg_item.i_item_id",
        "      and item.i_rec_end_date is null",
        "      and item.i_current_price <> stg_item.i_current_price;",
        "/* Sunset penultimate records that were inserted as type 2 */",
        "update {0}.item",
        "  set i_rec_end_date = trunc(sysdate)",
        "  from {0}.stg_item",
        "  where item.i_item_id = stg_item.i_item_id",
        "    and item.i_rec_end_date is null",
        "    and item.i_current_price <> stg_item.i_current_price;",
        "/* Commit and End transaction */",
        "commit;",
        "end transaction;"
    ]
  }
},
"ExecuteItemDimensionLoadJob": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
  "TimeoutSeconds": 180,
  "HeartbeatSeconds": 60,
  "Next": "GetItemDimensionLoadStatus",
  "InputPath": "$",
  "ResultPath": "$"
},
"GetItemDimensionLoadStatus": {
  "Type": "Task",
  "Next": "CheckItemDimensionLoadStatus",
  "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
  "TimeoutSeconds": 180,
  "HeartbeatSeconds": 60,
  "InputPath": "$",
  "ResultPath": "$.status"
},

```

```

    "CheckItemDimensionLoadStatus": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.status",
          "StringEquals": "FAILED",
          "Next": "FailItemDimensionLoad"
        },
        {
          "Variable": "$.status",
          "StringEquals": "FINISHED",
          "Next": "CompleteItemDimensionLoad"
        }
      ],
      "Default": "ItemWait"
    },
    "ItemWait": {
      "Type": "Wait",
      "Seconds": 5,
      "Next": "GetItemDimensionLoadStatus"
    },
    "CompleteItemDimensionLoad": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
      "TimeoutSeconds": 180,
      "HeartbeatSeconds": 60,
      "End": true
    },
    "FailItemDimensionLoad": {
      "Type": "Fail",
      "Cause": "ETL Workflow Failed",
      "Error": "Error"
    }
  }
}
],
},
"InitializeSalesFactLoadJob": {
  "Type": "Pass",
  "Next": "ExecuteSalesFactLoadJob",
  "Result": {
    "input": {
      "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",

```

```

"redshift_database": "dev",
"redshift_user": "awsuser",
"redshift_schema": "tpcds",
"snapshot_date": "2003-01-02",
"action": "load_sales_fact",
"sql_statement": [
  "begin transaction;",
  "/* Create a stg_store_sales staging table */",
  "drop table if exists {0}.stg_store_sales;",
  "create table {0}.stg_store_sales",
  "(sold_date          date encode zstd",
  ",i_item_id          varchar(16) encode zstd",
  ",c_customer_id      varchar(16) encode zstd",
  ",ca_address_id      varchar(16) encode zstd",
  ",ss_ticket_number   integer encode zstd",
  ",ss_quantity        integer encode zstd",
  ",ss_net_paid         numeric(7,2) encode zstd",
  ",ss_net_profit       numeric(7,2) encode zstd",
  ")",
  "backup no",
  "diststyle even;",
  "/* Ingest data from source */",
  "insert into {0}.stg_store_sales",

"(sold_date,i_item_id,c_customer_id,ca_address_id,ss_ticket_number,ss_quantity,ss_net_paid,ss_
  "values",

"('2003-01-02','AAAAAAAAIFNAAAAA','AAAAAAAAOHNBAAAA','AAAAAAAAAGBOFNIAA',1403191,13,5046.37,150
"('2003-01-02','AAAAAAAAIFNAAAAA','AAAAAAAAOHNBAAAA','AAAAAAAAAGBOFNIAA',1403191,13,2103.72,-12
"('2003-01-02','AAAAAAAIILOBAAAA','AAAAAAAAOHNBAAAA','AAAAAAAAAGBOFNIAA',1403191,13,959.10,-130
"('2003-01-02','AAAAAAAIILOBAAAA','AAAAAAAAHNAJAAAA','AAAAAAAIIAPCFNAA',1403191,13,962.65,-475
"('2003-01-02','AAAAAAAAMCOAAAAA','AAAAAAAAHNAJAAAA','AAAAAAAIIAPCFNAA',1201746,17,111.60,-241
"('2003-01-02','AAAAAAAAMCOAAAAA','AAAAAAAAHNAJAAAA','AAAAAAAIIAPCFNAA',1201746,17,4013.02,-11
"('2003-01-02','AAAAAAAAMCOAAAAA','AAAAAAAAMJCLACAA','AAAAAAAAMIFKEHAA',1201746,17,2689.12,-55
"('2003-01-02','AAAAAAAAMGOAAAAA','AAAAAAAAMJCLACAA','AAAAAAAAMIFKEHAA',193971,18,1876.89,-556
  "/* Delete any rows from target store_sales for the input date for
  idempotency */",

```

```

        "delete from {0}.store_sales where ss_sold_date_sk in (select d_date_sk
from {0}.date_dim where d_date='{1}');"
        "/* Insert data from staging table to the target table */",
        "insert into {0}.store_sales",
        "(ss_sold_date_sk",
        ",ss_item_sk",
        ",ss_customer_sk",
        ",ss_addr_sk",
        ",ss_ticket_number",
        ",ss_quantity",
        ",ss_net_paid",
        ",ss_net_profit",
        ")",
        "select date_dim.d_date_sk ss_sold_date_sk",
        "      ,item.i_item_sk ss_item_sk",
        "      ,customer.c_customer_sk ss_customer_sk",
        "      ,customer_address.ca_address_sk ss_addr_sk",
        "      ,ss_ticket_number",
        "      ,ss_quantity",
        "      ,ss_net_paid",
        "      ,ss_net_profit",
        " from {0}.stg_store_sales as store_sales",
        " inner join {0}.date_dim on store_sales.sold_date = date_dim.d_date",
        " left join {0}.item on store_sales.i_item_id = item.i_item_id and
item.i_rec_end_date is null",
        " left join {0}.customer on store_sales.c_customer_id =
customer.c_customer_id",
        " left join {0}.customer_address on store_sales.ca_address_id =
customer_address.ca_address_id;",
        "/* Drop staging table */",
        "drop table if exists {0}.stg_store_sales;",
        "/* Commit and End transaction */",
        "commit;",
        "end transaction;"
    ]
}
},
"ExecuteSalesFactLoadJob": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-
RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
    "TimeoutSeconds": 180,
    "HeartbeatSeconds": 60,

```

```
    "Next": "GetSalesFactLoadStatus",
    "InputPath": "$",
    "ResultPath": "$"
  },
  "GetSalesFactLoadStatus": {
    "Type": "Task",
    "Next": "CheckSalesFactLoadStatus",
    "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-
RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
    "TimeoutSeconds": 180,
    "HeartbeatSeconds": 60,
    "InputPath": "$",
    "ResultPath": "$.status"
  },
  "CheckSalesFactLoadStatus": {
    "Type": "Choice",
    "Choices": [
      {
        "Variable": "$.status",
        "StringEquals": "FAILED",
        "Next": "FailSalesFactLoad"
      },
      {
        "Variable": "$.status",
        "StringEquals": "FINISHED",
        "Next": "SalesETLPipelineComplete"
      }
    ],
    "Default": "SalesWait"
  },
  "SalesWait": {
    "Type": "Wait",
    "Seconds": 5,
    "Next": "GetSalesFactLoadStatus"
  },
  "FailSalesFactLoad": {
    "Type": "Fail",
    "Cause": "ETL Workflow Failed",
    "Error": "Error"
  },
  "ClusterUnavailable": {
    "Type": "Fail",
    "Cause": "Redshift cluster is not available",
    "Error": "Error"
  }
}
```

```

    },
    "SalesETLPipelineComplete": {
      "Type": "Pass",
      "Next": "ValidateSalesMetric",
      "Result": {
        "input": {
          "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
          "redshift_database": "dev",
          "redshift_user": "awsuser",
          "redshift_schema": "tpcds",
          "snapshot_date": "2003-01-02",
          "action": "validate_sales_metric",
          "sql_statement": [
            "select 1/count(1) from {0}.store_sales where ss_sold_date_sk in (select",
            "d_date_sk from {0}.date_dim where d_date='{1}')"
          ]
        }
      }
    },
    "ValidateSalesMetric": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
      "TimeoutSeconds": 180,
      "HeartbeatSeconds": 60,
      "Next": "GetValidateSalesMetricStatus",
      "InputPath": "$",
      "ResultPath": "$"
    },
    "GetValidateSalesMetricStatus": {
      "Type": "Task",
      "Next": "CheckValidateSalesMetricStatus",
      "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
      "TimeoutSeconds": 180,
      "HeartbeatSeconds": 60,
      "InputPath": "$",
      "ResultPath": "$.status"
    },
    "CheckValidateSalesMetricStatus": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.status",

```

```

        "StringEquals": "FAILED",
        "Next": "FailSalesMetricValidation"
    },
    {
        "Variable": "$.status",
        "StringEquals": "FINISHED",
        "Next": "DataValidationComplete"
    }
],
"Default": "SalesValidationWait"
},
"SalesValidationWait": {
    "Type": "Wait",
    "Seconds": 5,
    "Next": "GetValidateSalesMetricStatus"
},
"FailSalesMetricValidation": {
    "Type": "Fail",
    "Cause": "Data Validation Failed",
    "Error": "Error"
},
"DataValidationComplete": {
    "Type": "Pass",
    "Next": "InitializePauseCluster"
},
"InitializePauseCluster": {
    "Type": "Pass",
    "Next": "PauseCluster",
    "Result": {
        "input": {
            "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
            "operation": "pause"
        }
    }
},
"PauseCluster": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftOperations-AKIAIOSFODNN7EXAMPLE",
    "TimeoutSeconds": 180,
    "HeartbeatSeconds": 60,
    "Next": "PauseClusterWait",
    "InputPath": "$",
    "ResultPath": "$.clusterStatus",

```

```

    "Catch": [
      {
        "ErrorEquals": [
          "States.ALL"
        ],
        "Next": "ClusterPausedComplete"
      }
    ]
  },
  "InitializeCheckPauseCluster": {
    "Type": "Pass",
    "Next": "GetStateOfPausedCluster",
    "Result": {
      "input": {
        "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
        "operation": "status"
      }
    }
  },
  "GetStateOfPausedCluster": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftOperations-AKIAIOSFODNN7EXAMPLE",
    "TimeoutSeconds": 180,
    "HeartbeatSeconds": 60,
    "Next": "IsClusterPaused",
    "InputPath": "$",
    "ResultPath": "$.clusterStatus"
  },
  "IsClusterPaused": {
    "Type": "Choice",
    "Choices": [
      {
        "Variable": "$.clusterStatus",
        "StringEquals": "available",
        "Next": "InitializePauseCluster"
      },
      {
        "Variable": "$.clusterStatus",
        "StringEquals": "paused",
        "Next": "ClusterPausedComplete"
      },
      {
        "Variable": "$.clusterStatus",

```



```

        "StringEquals": "unavailable",
        "Next": "ClusterUnavailable"
    },
    {
        "Variable": "$.clusterStatus",
        "StringEquals": "resuming",
        "Next": "PauseClusterWait"
    }
]
},
"PauseClusterWait": {
    "Type": "Wait",
    "Seconds": 720,
    "Next": "InitializeCheckPauseCluster"
},
"ClusterPausedComplete": {
    "Type": "Pass",
    "End": true
}
}
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftDataApi-AIDACKCEVSQ6C2EXAMPLE",
                "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftOperations-AKIAIOSFODNN7EXAMPLE"
            ]
        }
    ]
}

```

```
    ],  
    "Effect": "Allow"  
  }  
]  
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Use Step Functions and AWS Batch with error handling

This sample project demonstrates how to use Step Functions to run an AWS Batch job using a state machine with error-handling capabilities.

In this project, Step Functions uses a state machine to call the AWS Batch job synchronously. It then waits for the job to succeed or fail, retries and catches errors when a job fails, then sends an Amazon SNS topic with a message about whether the job succeeded or failed.

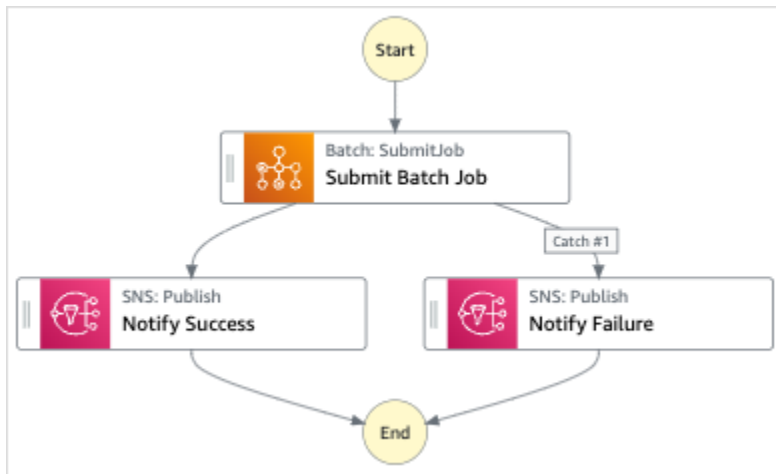
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Manage a batch job** in the search box, and then choose **Manage a batch job** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- A AWS Batch job
- An Amazon SNS topic
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Manage a batch job** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language](#) (ASL) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.


After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

 **Important**

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

 **Note**

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with AWS Batch and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls AWS Batch and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the `Resource` field, and by passing `Parameters` to the service API.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of the Amazon States Language for notification on an AWS Batch
job completion",
  "StartAt": "Submit Batch Job",
  "TimeoutSeconds": 3600,
  "States": {
    "Submit Batch Job": {
      "Type": "Task",
      "Resource": "arn:aws:states:::batch:submitJob.sync",
      "Parameters": {
        "JobName": "BatchJobNotification",
```

```
    "JobQueue": "arn:aws:batch:us-west-2:123456789012:job-queue/  
BatchJobQueue-123456789abcdef",  
    "JobDefinition": "arn:aws:batch:us-west-2:123456789012:job-definition/  
BatchJobDefinition-123456789abcdef:1"  
  },  
  "Next": "Notify Success",  
  "Retry": [  
    {  
      "ErrorEquals": [  
        "States.ALL"  
      ],  
      "IntervalSeconds": 30,  
      "MaxAttempts": 2,  
      "BackoffRate": 1.5  
    }  
  ],  
  "Catch": [  
    {  
      "ErrorEquals": [ "States.ALL" ],  
      "Next": "Notify Failure"  
    }  
  ]  
},  
"Notify Success": {  
  "Type": "Task",  
  "Resource": "arn:aws:states:::sns:publish",  
  "Parameters": {  
    "Message": "Batch job submitted through Step Functions succeeded",  
    "TopicArn": "arn:aws:sns:us-west-2:123456789012:StepFunctionsSample-  
BatchJobManagement12345678-9abc-def0-1234-567890abcdef-SNSTopic-A2B3C4D5E6F7G"  
  },  
  "End": true  
},  
"Notify Failure": {  
  "Type": "Task",  
  "Resource": "arn:aws:states:::sns:publish",  
  "Parameters": {  
    "Message": "Batch job submitted through Step Functions failed",  
    "TopicArn": "arn:aws:sns:us-west-2:123456789012:StepFunctionsSample-  
BatchJobManagement12345678-9abc-def0-1234-567890abcdef-SNSTopic-A2B3C4D5E6F7G"  
  },  
  "End": true  
}  
}
```

```
}
```

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

Example BatchJobNotificationAccessPolicy

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "sns:Publish"
      ],
      "Resource": [
        "arn:aws:sns:us-west-2:123456789012:StepFunctionsSample-
BatchJobManagement12345678-9abc-def0-1234-567890abcdef-SNSTopic-A2B3C4D5E6F7G"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "batch:SubmitJob",
        "batch:DescribeJobs",
        "batch:TerminateJob"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:us-west-2:123456789012:rule/
StepFunctionsGetEventsForBatchJobsRule"
      ],
      "Effect": "Allow"
    }
  ]
}
```

```
    }  
  ]  
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Fan out an AWS Batch job

This sample project demonstrates how to use Step Functions's [Map](#) state to fan out AWS Batch jobs.

In this project, Step Functions uses a state machine to invoke a Lambda function to do simple pre-processing, then invokes multiple AWS Batch jobs in parallel using the [Map](#) state.

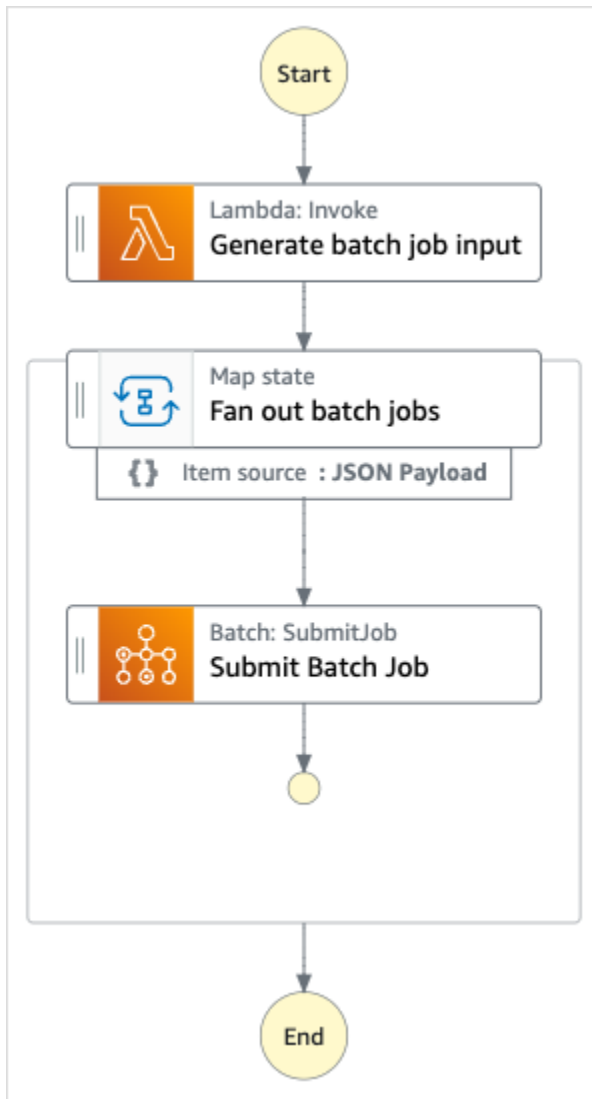
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Fan out a batch job** in the search box, and then choose **Fan out a batch job** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- A Lambda function
- An AWS Batch job queue
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Fan out a batch job** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

Note

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with AWS Batch and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls AWS Batch and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the `Resource` field, and by passing `Parameters` to the service API.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of the Amazon States Language for fanning out AWS Batch job",
  "StartAt": "Generate batch job input",
  "TimeoutSeconds": 3600,
  "States": {
    "Generate batch job input": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "OutputPath": "$.Payload",
      "Parameters": {
        "FunctionName": "<GENERATE_BATCH_JOB_INPUT_LAMBDA_FUNCTION_NAME>"
      },
      "Next": "Fan out batch jobs"
    },
    "Fan out batch jobs": {
      "Comment": "Start multiple executions of batch job depending on pre-processed data",
      "Type": "Map",
      "End": true,
      "ItemsPath": "$",
      "Parameters": {
        "BatchNumber.$": "$$.Map.Item.Value"
      },
      "Iterator": {
        "StartAt": "Submit Batch Job",
        "States": {
          "Submit Batch Job": {
            "Type": "Task",
            "Resource": "arn:aws:states:::batch:submitJob.sync",
            "Parameters": {
              "JobName": "BatchJobFanOut",
              "JobQueue": "<BATCH_QUEUE_ARN>",
              "JobDefinition": "<BATCH_JOB_DEFINITION_ARN>"
            },
            "End": true
          }
        }
      }
    }
  }
}
```

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

Example BatchJobFanOutAccessPolicy

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "batch:SubmitJob",
        "batch:DescribeJobs",
        "batch:TerminateJob"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:us-west-2:123456789012:rule/StepFunctionsGetEventsForBatchJobsRule"
      ],
      "Effect": "Allow"
    }
  ]
}
```

Example InvokeGenerateBatchJobMapLambdaPolicy

```
{
  "Statement": [
    {
      "Action": [
        "lambda:InvokeFunction"
      ]
    }
  ]
}
```

```
    ],
    "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:StepFunctionsSample-BatchJobFa-
GenerateBatchJobMap-444455556666",
    "Effect": "Allow"
  }
]
```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

AWS Batch with Lambda

This sample project demonstrates how to use Step Functions to pre-process data with AWS Lambda functions and then orchestrate AWS Batch jobs.

In this project, Step Functions uses a state machine to invoke a Lambda function to do simple pre-processing before an AWS Batch job is submitted. Multiple jobs may be invoked depending on the result or success of the previous one.

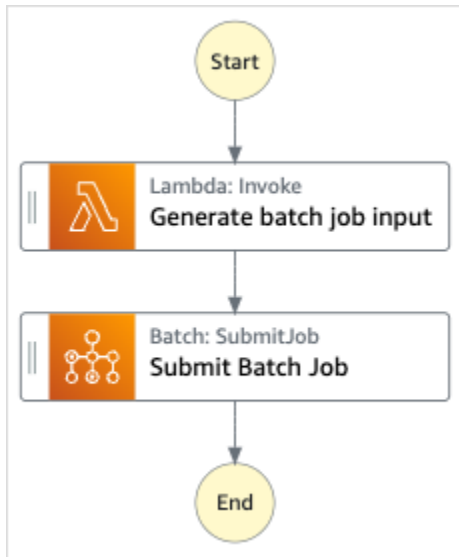
Step 1: Create the State Machine and Provision Resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **Batch job with Lambda** in the search box, and then choose **Batch job with Lambda** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- A Lambda function
- An AWS Batch job
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Batch job with Lambda** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

Note

Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

 **Note**

If the demo project you deployed contains prepopulated execution input data, use that input to run the state machine.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Example State Machine Code

The state machine in this sample project integrates with AWS Batch and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls AWS Batch and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the `Resource` field, and by passing `Parameters` to the service API.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```
{
  "Comment": "An example of the Amazon States Language for using batch job with pre-
processing lambda",
  "StartAt": "Generate batch job input",
  "TimeoutSeconds": 3600,
  "States": {
    "Generate batch job input": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "OutputPath": "$.batch_input",
```

```

    "Parameters": {
      "FunctionName": "<GENERATE_BATCH_JOB_INPUT_LAMBDA_FUNCTION_NAME>"
    },
    "Next": "Submit Batch Job"
  },
  "Submit Batch Job": {
    "Type": "Task",
    "Resource": "arn:aws:states:::batch:submitJob.sync",
    "Parameters": {
      "JobName": "BatchJobFanOut",
      "JobQueue": "<BATCH_QUEUE_ARN>",
      "JobDefinition": "<BATCH_JOB_DEFINITION_ARN>",
      "Parameters.$": "$.batch_input"
    },
    "End": true
  }
}
}
}

```

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

Example BatchJobWithLambdaAccessPolicy

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "sns:Publish"
      ],
      "Resource": [
        "arn:aws:sns:us-west-2:123456789012:ManageBatchJob-SNSTopic-
        JHLYYG7AZPZI"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "batch:SubmitJob",

```

```

        "batch:DescribeJobs",
        "batch:TerminateJob"
    ],
    "Resource": "*",
    "Effect": "Allow"
},
{
    "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
    ],
    "Resource": [
        "arn:aws:events:us-west-2:123456789012:rule/
StepFunctionsGetEventsForBatchJobsRule"
    ],
    "Effect": "Allow"
}
]
}

```

Example InvokeGenerateBatchJobMapLambdaPolicy

```

{
    "Statement": [
        {
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:StepFunctionsSample-BatchWithL-
GenerateBatchJobMap-444455556666",
            "Effect": "Allow"
        }
    ]
}

```

For information about how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](#).

Perform AI prompt-chaining with Amazon Bedrock

This sample project demonstrates how you can integrate with Amazon Bedrock to perform AI prompt-chaining. This sample project shows how you can build high-quality chatbots using Amazon Bedrock. The project chains together some prompts and resolves them in the sequence in which they're provided. Chaining of these prompts augments the ability of the language model being used to deliver a highly-curated response.

This sample project creates the state machine, the supporting AWS resources, and configures the related IAM permissions. Explore this sample project to learn about using Amazon Bedrock optimized service integration with Step Functions state machines, or use it as a starting point for your own projects.

Topics

- [AWS CloudFormation template and additional resources](#)
- [Prerequisites](#)
- [Step 1: Create the state machine and provision resources](#)
- [Step 2: Run the state machine](#)

AWS CloudFormation template and additional resources

You use a CloudFormation template to deploy this sample project. This template creates the following resources in your AWS account:

- A Step Functions state machine.
- Execution role for the state machine. This role grants the permissions that your state machine needs to access other AWS services and resources such as the Amazon Bedrock [InvokeModel](#) action.

Prerequisites

This sample project uses the Cohere Command large language model (LLM). To successfully run this sample project, you must add access to this LLM from the Amazon Bedrock console. To add the model access, do the following:

1. Open the [Amazon Bedrock console](#).

2. On the navigation pane, choose **Model access**.
3. Choose **Manage model access**.
4. Select the check box next to **Cohere**.
5. Choose **Request access**. The **Access status** for **Cohere** model shows as **Access granted**.

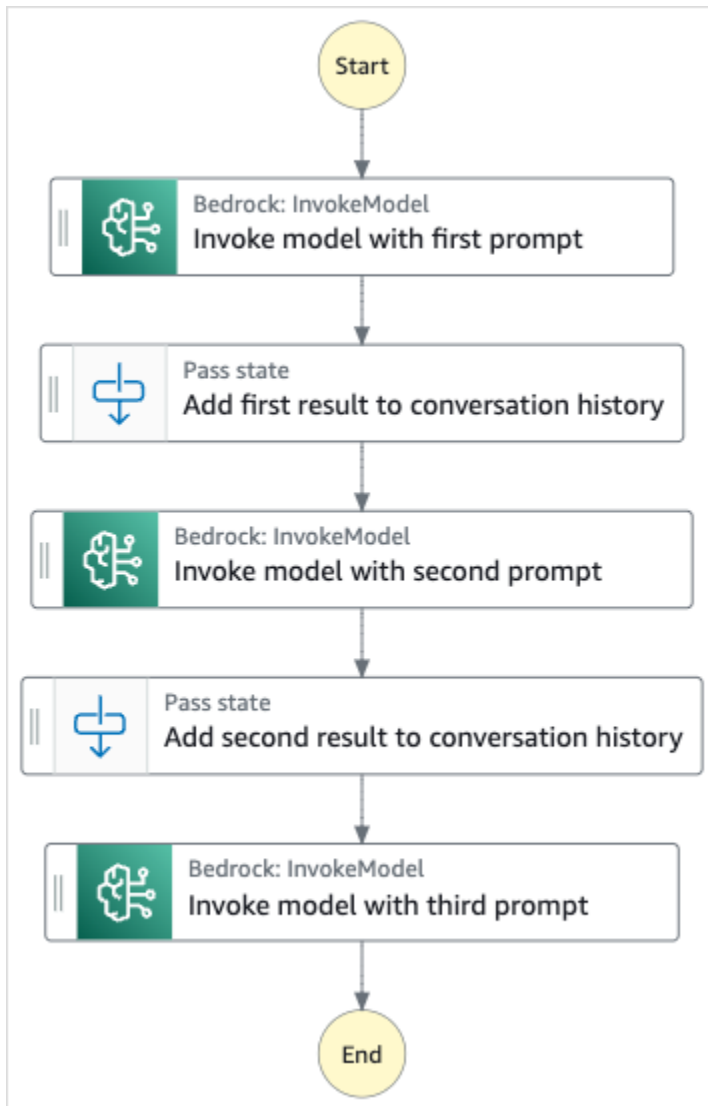
Step 1: Create the state machine and provision resources

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Type **bedrock** in the search box, and then choose **Perform AI prompt-chaining with Bedrock** from the search results that are returned.
3. Choose **Next** to continue.
4. Step Functions lists the AWS services used in the sample project you selected. It also shows a workflow graph for the sample project. Deploy this project to your AWS account or use it as a starting point for building your own projects. Based on how you want to proceed, choose **Run a demo** or **Build on it**.

This sample project deploys the following resources:

- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

The following image shows the workflow graph for the **Perform AI prompt-chaining with Bedrock** sample project:



5. Choose **Use template** to continue with your selection.
6. Do one of the following:
 - If you selected **Build on it**, Step Functions creates the workflow prototype for the sample project you selected. Step Functions doesn't deploy the resources listed in the workflow definition.

In Workflow Studio's [Design mode](#), drag and drop states from the [States browser](#) to continue building your workflow prototype. Or switch to the [Code mode](#) that provides an integrated code editor similar to VS Code for updating the [Amazon States Language \(ASL\)](#) definition of your state machine within the Step Functions console. For more information about using Workflow Studio to build your state machines, see [Using Workflow Studio](#).

⚠ Important

Remember to update the placeholder Amazon Resource Name (ARN) for the resources used in the sample project before you [run your workflow](#).

- If you selected **Run a demo**, Step Functions creates a read-only sample project which uses an AWS CloudFormation template to deploy the AWS resources listed in that template to your AWS account.

ℹ Tip

To view the state machine definition of the sample project, choose **Code**.

When you're ready, choose **Deploy and run** to deploy the sample project and create the resources.

It can take up to 10 minutes for these resources and related IAM permissions to be created. While your resources are being deployed, you can open the CloudFormation Stack ID link to see which resources are being provisioned.

After all the resources in the sample project are created, you can see the new sample project listed on the **State machines** page.

⚠ Important

Standard charges may apply for each service used in the CloudFormation template.

Step 2: Run the state machine

1. On the **State machines** page, choose your sample project.
2. On the sample project page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:
 1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions automatically generates a unique execution name.

Note

Step Functions allows you to create names for state machines, executions, activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

If you chose to **Run a demo**, you need not provide any execution input.

3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see [Execution Details page – Interface overview](#).

Quotas

AWS Step Functions places quotas on the sizes of certain state machine parameters, such as the number of API actions during a certain time period or the number of state machines that you can define. Although these quotas are designed to prevent a misconfigured state machine from consuming all of the resources of the system, many aren't hard quotas.

To request a service quota increase, you can do one of the following:

- Use the Service Quotas console at <https://console.aws.amazon.com/servicequotas/home>. For information about requesting a quota increase using the Service Quotas console, see [Requesting a quota increase](#) in the *Service Quotas User Guide*.
- Use the **Support Center** page in the AWS Management Console to request a quota increase for resources provided by AWS Step Functions on a per-Region basis. For more information, see [AWS service quotas](#) in the *AWS General Reference*.

Note

If a particular stage of your state machine execution or activity execution takes too long, you can configure a state machine timeout to cause a timeout event.

Topics

- [General quotas](#)
- [Quotas related to accounts](#)
- [Quotas related to HTTP Task](#)
- [Quotas related to state throttling](#)
- [Quotas related to API action throttling](#)
- [Quotas related to state machine executions](#)
- [Quotas related to task executions](#)
- [Quotas related to versions and aliases](#)
- [Restrictions related to tagging](#)

General quotas

Quota	Description
Names in Step Functions	<p>Names of state machines, executions, and activity tasks must not exceed 80 characters in length. These names must be unique for your account and AWS Region, and must not contain any of the following:</p> <ul style="list-style-type: none">• Whitespace• Wildcard characters (? *)• Bracket characters (< > { } [])• Special characters (" # % \ ^ ~ ` \$ & , ; : /)• Control characters (\\u0000 - \\u001f or \\u007f - \\u009f). <p>If your state machine is of type Express, you can provide the same name to multiple executions of the state machine. Step Functions generates a unique execution ARN for each Express state machine execution, even if multiple executions have the same name.</p> <p>Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.</p>

Quotas related to accounts

Resource	Default quota	Can be increased to
Maximum number of registered state machines	10,000	25,000
Maximum number of registered activities	10,000	15,000
Maximum request size	1 MB per request. This is the total data size per Step Functions API request, including the request header and all other associated request data.	Hard quota
Maximum open executions per account	1,000,000 executions for each AWS account in each AWS Region. Exceeding this will cause an <code>ExecutionLimitExceeded</code> error. This doesn't apply to Express Workflows.	Millions
Maximum number of open Map Runs	1000	Hard quota
An open Map Run is a Map Run that has started, but hasn't yet completed. Scheduled Map Runs wait at the MapRunStarted event until the total number of open Map Runs is less than the default quota of 1000.	This quota is applicable for Distributed Map state .	

Resource	Default quota	Can be increased to
Maximum redrives of a Map Run.	1000 This quota is applicable for <i>Distributed Map state</i> .	Hard quota
Maximum number of parallel Map Run child executions	10,000	Hard quota

Quotas related to HTTP Task

HTTP Tasks are throttled using a token bucket scheme to maintain the Step Functions service bandwidth.

Resource	Bucket size	Refill rate per second
HTTP Task	300	300

The following table lists the quota for an HTTP Task duration.

Resource	Default quota
HTTP Task duration	60 seconds
An HTTP Task duration refers to the time taken by an HTTP Task to send an HTTP request and receive a response.	This is a hard quota that can't be changed.

Quotas related to state throttling

Step Functions state transitions are throttled using a token bucket scheme to maintain service bandwidth. Standard Workflows and Express Workflows have different state transition throttling. Standard Workflows quotas are soft quotas and can be increased.

Note

Throttling on the StateTransition service metric is reported as ExecutionThrottled in Amazon CloudWatch. For more information, see the [ExecutionThrottled CloudWatch metric](#).

Service metric	Standard		Express	
	Bucket size	Refill rate per second	Bucket size	Refill rate per second
StateTransition — <i>In US East (N. Virginia), US West (Oregon), and Europe (Ireland)</i>	5,000	5,000	Unlimited	Unlimited
StateTransition — <i>All other regions</i>	800	800	Unlimited	Unlimited

Quotas related to API action throttling

Some Step Functions API actions are throttled using a token bucket scheme to maintain service bandwidth. These quotas are soft quotas and can be increased.

Note

Throttling quotas are per account, per AWS Region.
AWS Step Functions may increase both the bucket size and refill rate at any time.

	Standard		Express	
API name	Bucket size	Refill rate per second	Bucket size	Refill rate per second
StartExecution — <i>In US East (N. Virginia), US West (Oregon), and Europe (Ireland)</i>	1,300	300	6,000	6,000
StartExecution — <i>All other regions</i>	800	150	6,000	6,000

Quota related to TestState API

API name	Quota	Can be increased to
TestState	1 transaction per second (TPS)	Hard quota

Other quotas

These quotas are soft quotas and can be increased.

	In US East (N. Virginia), US West (Oregon), and Europe (Ireland)		All other regions	
API name	Bucket size	Refill rate per second	Bucket size	Refill rate per second
CreateActivity	100	1	100	1

API name	In US East (N. Virginia), US West (Oregon), and Europe (Ireland)		All other regions	
	Bucket size	Refill rate per second	Bucket size	Refill rate per second
CreateStateMachine	100	1	100	1
DeleteActivity	100	1	100	1
DeleteStateMachine	100	1	100	1
DescribeActivity	200	1	200	1
DescribeExecution	300	15	250	10
DescribeStateMachine	200	20	200	20
DescribeStateMachineForExecution	200	1	200	1
GetActivityTask	3,000	500	1,500	300
GetExecutionHistory	400	20	400	20
ListActivities	100	10	100	5

API name	In US East (N. Virginia), US West (Oregon), and Europe (Ireland)		All other regions	
	Bucket size	Refill rate per second	Bucket size	Refill rate per second
ListExecutions	200	5	100	2
ListStateMachines	100	5	100	5
ListTagsForResource	100	1	100	1
SendTaskFailure	3,000	500	1,500	300
SendTaskHeartbeat	3,000	500	1,500	300
SendTaskSuccess	3,000	500	1,500	300
StartSyncExecution	Synchronous Express execution API calls don't contribute to existing account capacity limits. Step Functions provides capacity on demand and automatically scales with sustained workload. Surges in workload may be throttled until capacity is available.			
	If you experience throttling, try again after some time. For information about Synchronous Express workflows, see Synchronous and Asynchronous Express Workflows .			
StopExecution	1,000	200	500	25
TagResource	200	1	200	1
UntagResource	200	1	200	1

API name	In US East (N. Virginia), US West (Oregon), and Europe (Ireland)		All other regions	
	Bucket size	Refill rate per second	Bucket size	Refill rate per second
UpdateStateMachine	100	1	100	1

Quotas related to state machine executions

The following table describes quotas related to state machine executions. State machine execution quotas are hard quotas that can't be changed, except for the *Execution history retention time* quota.

Quota	Standard	Express
Maximum execution time	1 year. If an execution runs for more than the 1-year maximum, it will fail with a <code>States.Timeout</code> error and emit a <code>Execution sTimedOut</code> CloudWatch metric.	5 minutes. If an execution runs for more than the 5-minute maximum, it will fail with a <code>States.Timeout</code> error and emit a <code>Execution sTimedOut</code> CloudWatch metric.
Maximum execution history size	25,000 events in a single state machine execution history. If the execution history reaches this quota, the execution will fail. To avoid this, see Avoid reaching the history quota .	Unlimited.
Maximum execution idle time	1 year (constrained by the maximum execution time).	5 minutes (constrained by the maximum execution time).
Execution history retention time	90 days after an execution is closed. After this time, you	To see execution history, Amazon CloudWatch Logs

Quota	Standard	Express
	<p>can no longer retrieve or view the execution history. There is no further quota for the number of closed executions that Step Functions retains.</p> <p>To meet compliance, organizational, or regulatory requirements, you can reduce the execution history retention period to 30 days by sending a quota request. To do this, use the AWS Support Center Console and create a new case.</p> <p>The change to reduce the retention period to 30 days is applicable for each account in a Region.</p>	<p>logging must be configured. For more information, see Logging using CloudWatch Logs.</p>
<p>Execution redrivable period</p> <p>Redrivable period refers to the time during which you can redrive a given Standard Workflow execution. This period starts from the day a state machine completes its execution.</p>	<p>14 days.</p> <p>This hard quota is applicable for Distributed Map state.</p>	<p>Redrive is currently not supported for Express workflows.</p>

Quotas related to task executions

The following table describes quotas related to task executions. These are all hard quotas that cannot be changed.

Quota	Standard	Express
Maximum task execution time	1 year (constrained by the maximum execution time)	5 minutes (constrained by the maximum execution time)
Maximum time Step Functions keeps a task in the queue	1 year (constrained by the maximum execution time)	5 minutes (constrained by the maximum execution time)
Maximum activity pollers per Amazon Resource Name (ARN)	1,000 pollers calling <code>GetActivityTask</code> per ARN. Exceeding this quota results in this error: <i>"The maximum number of workers concurrently polling for activity tasks has been reached."</i>	Does not apply to Express Workflows.
Maximum input or output size for a task, state, or execution	256 KB of data as a UTF-8 encoded string. This quota affects tasks (activity, Lambda function, or integrated service), state or execution output, and input data when scheduling a task, entering a state, or starting an execution	256 KB of data as a UTF-8 encoded string. This quota affects tasks (activity, Lambda function, or integrated service), state or execution output, and input data when scheduling a task, entering a state, or starting an execution

Quotas related to versions and aliases

Resource	Default quota
Maximum number of published state machine versions	1000 for each state machine.

Resource	Default quota
	To request an increase to this soft limit, use the Support Center page in the AWS Management Console .
Maximum number of state machine aliases	100 for each state machine. To request an increase to this soft limit, use the Support Center page in the AWS Management Console .

Restrictions related to tagging

Be aware of these restrictions when tagging Step Functions resources.

Note

Tagging restrictions cannot be increased like other quotas.

Restriction	Description
Maximum number of tags per resource	50
Maximum key length	128 Unicode characters in UTF-8
Maximum value length	256 Unicode characters in UTF-8
Prefix restriction	Do not use the <code>aws:</code> prefix in your tag names or values because it is reserved for AWS use. You can't edit or delete tag names or values with this prefix. Tags with this prefix do not count against your tags per resource quota.
Character restrictions	Tags may only contain Unicode letters, digits, whitespace, or these symbols: <code>_ . : / = + - @</code>

Logging and monitoring in AWS Step Functions

Logging and monitoring are important for maintaining the reliability, availability, and performance of Step Functions and your AWS solutions. There are several tools available to use with Step Functions:

Tip

To deploy a sample workflow to your AWS account and learn how to monitor metrics, logs, and traces of the workflow execution, see [Module 12 - Observability](#) of *The AWS Step Functions Workshop*.

Topics

- [Monitoring Step Functions Using CloudWatch](#)
- [EventBridge \(CloudWatch Events\) for Step Functions execution status changes](#)
- [Recording API calls with AWS CloudTrail](#)
- [Logging using CloudWatch Logs](#)
- [AWS X-Ray and Step Functions](#)
- [Using AWS User Notifications with AWS Step Functions](#)

Monitoring Step Functions Using CloudWatch

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS Step Functions and your AWS solutions. You should collect as much monitoring data from the AWS services that you use so that you can debug multi-point failures. Before you start monitoring Step Functions, you should create a monitoring plan that answers the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?
- Who will perform the monitoring tasks?

- Who should be notified when something goes wrong?

The next step is to establish a baseline for normal performance in your environment. To do this, measure performance at various times and under different load conditions. As you monitor Step Functions, consider storing historical monitoring data. Such data can give you a baseline to compare against current performance data, to identify normal performance patterns and performance anomalies, and to devise ways to address issues.

For example, with Step Functions, you can monitor how many activities or AWS Lambda tasks fail due to a heartbeat timeout. When performance falls outside your established baseline, you might have to change your heartbeat interval.

To establish a baseline you should, at a minimum, monitor the following metrics:

- `ActivitiesStarted`
- `ActivitiesTimedOut`
- `ExecutionsStarted`
- `ExecutionsTimedOut`
- `LambdaFunctionsStarted`
- `LambdaFunctionsTimedOut`

The following sections describe metrics that Step Functions provides to Amazon CloudWatch. You can use these metrics to track your state machines and activities and to set alarms on threshold values. You can view metrics using the AWS Management Console.

Metrics that report a time interval

Some of the Step Functions CloudWatch metrics are *time intervals*, always measured in milliseconds. These metrics generally correspond to stages of your execution for which you can set state machine, activity, and Lambda function timeouts, with descriptive names.

For example, the `ActivityRunTime` metric measures the time it takes for an activity to complete after it begins to execute. You can set a timeout value for the same time period.

In the CloudWatch console, you can get the best results if you choose **average** as the display statistic for time interval metrics.

Metrics that report a count

Some of the Step Functions CloudWatch metrics report results as a *count*. For example, `ExecutionsFailed` records the number of failed state machine executions.

Step Functions emits two `ExecutionsStarted` metrics for every state machine execution. This causes the [SampleCount](#) statistic for the `ExecutionsStarted` metric to show the value of 2 for every state machine execution. The `SampleCount` statistic shows `ExecutionStarted=1` and `ExecutionStarted=0` when the execution completes.

Tip

We recommend selecting **Sum** as the display statistic for metrics that report a count in the CloudWatch console.

Execution metrics

The `AWS/States` namespace includes the following metrics for all Step Functions executions. These are dimensionless metrics that apply across your account in a region.

Metric	Description
<code>OpenExecutionCount</code>	<p>Approximate number of currently <i>open executions</i>—workflows that are currently in progress in your account.</p> <p>The intent is to provide insight into when your workflows are approaching the maximum execution limit, to avoid Execution LimitExceeded errors when calling <code>StartExecution</code> or <code>RedriveExecution</code> for Standard Workflows.</p> <p>The metric depends on active workflow state transitions, so at low levels, the estimate may not align with observed running workflow count.</p>
<code>OpenExecutionLimit</code>	<p>Maximum number of open executions. For more information, see Quotas related to accounts.</p> <p><i>This limit does not apply to Express Workflows.</i></p>

Execution metrics for state machine with version or alias

When you run a state machine execution with a [version](#) or an [alias](#), Step Functions emits the following metrics. The `ExecutionThrottled` metric will only be emitted in the case of throttled execution. These metrics will include a `StateMachineArn` to identify a specific state machine.

Metric	Description
<code>ExecutionTime</code>	Interval, in milliseconds, between the time the execution starts and the time it closes.
<code>ExecutionThrottled</code>	Number of <code>StateEntered</code> events and retries that have been throttled. This is related to <code>StateTransition</code> throttling. For more information, see Quotas related to state throttling .
<code>ExecutionsAborted</code>	Number of aborted or terminated executions.
<code>ExecutionsFailed</code>	Number of failed executions.
<code>ExecutionsStarted</code>	Number of started executions.
<code>ExecutionsSucceeded</code>	Number of successfully completed executions.
<code>ExecutionsTimedOut</code>	Number of executions that time out for any reason.

Execution metrics for Express Workflows

The `AWS/States` namespace includes the following metrics for Step Functions Express Workflows' executions.

Metric	Description
<code>ExpressExecutionMemory</code>	The total memory consumed by an Express Workflow.
<code>ExpressExecutionBilledDuration</code>	The duration for which an Express Workflow is charged.

Metric	Description
ExpressExecutionBilledMemory	The amount of consumed memory for which an Express Workflow is charged.

Redrive execution metrics for Standard Workflows

When you [redrive](#) a state machine execution, Step Functions emits the following metrics.

For all redriven executions, the `Executions*` metric is emitted. For example, say a redriven execution aborts. This execution will emit non-zero datapoints for both `RedrivenExecutionsAborted` and `ExecutionsAborted`.

Metric	Description
ExecutionsRedriven	Number of redriven executions.
RedrivenExecutionsAborted	Number of redriven executions that are canceled or terminated.
RedrivenExecutionsTimedOut	Number of redriven executions that time out for any reason.
RedrivenExecutionsSucceeded	Number of redriven executions that completed successfully.
RedrivenExecutionsFailed	Number of redriven executions that failed.

Dimension for Step Functions execution metrics

Dimension	Description
StateMachineArn	The Amazon Resource Name (ARN) of the state machine for the execution in question.

Dimensions for executions with version

Dimension	Description
StateMachineArn	The Amazon Resource Name (ARN) of the state machine whose execution was started by a version .
Version	State machine version used to start the execution.

Dimensions for executions with an alias

Dimension	Description
StateMachineArn	The Amazon Resource Name (ARN) of the state machine whose execution was started by an alias .
Alias	State machine alias used to start the execution.

Resource count metrics for versions and aliases

The AWS/States namespace includes the following metrics for the count of versions and aliases of a state machine.

Metric	Description
AliasCount	Number of aliases created for the state machine. You can create up to 100 aliases for each state machine.
VersionCount	Number of versions published for the state machine. You can publish up to 1000 versions of a state machine.

Dimension for resource count metrics for versions and aliases

Dimension	Description
ResourceArn	The Amazon Resource Name (ARN) of the state machine with a version or an alias.

Activity Metrics

The AWS/States namespace includes the following metrics for Step Functions activities.

Metric	Description
ActivityRunTime	Interval, in milliseconds, between the time the activity starts and the time it closes.
ActivityScheduleTime	Interval, in milliseconds, for which the activity stays in the schedule state.
ActivityTime	Interval, in milliseconds, between the time the activity is scheduled and the time it closes.
ActivitiesFailed	Number of failed activities.
ActivitiesHeartbeatTimedOut	Number of activities that time out due to a heartbeat timeout.
ActivitiesScheduled	Number of scheduled activities.
ActivitiesStarted	Number of started activities.
ActivitiesSucceeded	Number of successfully completed activities.
ActivitiesTimedOut	Number of activities that time out on close.

Dimension for Step Functions Activity Metrics

Dimension	Description
ActivityArn	The ARN of the activity.

Lambda Function Metrics

The AWS/States namespace includes the following metrics for Step Functions Lambda functions.

Metric	Description
LambdaFunctionRunTime	Interval, in milliseconds, between the time the Lambda function starts and the time it closes.
LambdaFunctionScheduleTime	Interval, in milliseconds, for which the Lambda function stays in the schedule state.
LambdaFunctionTime	Interval, in milliseconds, between the time the Lambda function is scheduled and the time it closes.
LambdaFunctionsFailed	Number of failed Lambda functions.
LambdaFunctionsScheduled	Number of scheduled Lambda functions.
LambdaFunctionsStarted	Number of started Lambda functions.
LambdaFunctionsSucceeded	Number of successfully completed Lambda functions.
LambdaFunctionsTimedOut	Number of Lambda functions that time out on close.

Dimension for Step Functions Lambda Function Metrics

Dimension	Description
LambdaFunctionArn	The ARN of the Lambda function.

Note

Lambda Function Metrics are emitted for Task states that specify the Lambda function ARN in the `Resource` field. Task states that use `"Resource": "arn:aws:states:::lambda:invoke"` emit Service Integration Metrics instead. For more information, see [Invoke Lambda with Step Functions](#).

Service Integration Metrics

The AWS/States namespace includes the following metrics for Step Functions service integrations. For more information, see [Using AWS Step Functions with other services](#).

Metric	Description
ServiceIntegrationRunTime	Interval, in milliseconds, between the time the Service Task starts and the time it closes.
ServiceIntegrationScheduleTime	Interval, in milliseconds, for which the Service Task stays in the schedule state.
ServiceIntegrationTime	Interval, in milliseconds, between the time the Service Task is scheduled and the time it closes.
ServiceIntegrationFailed	Number of failed Service Tasks.
ServiceIntegrationScheduled	Number of scheduled Service Tasks.

Metric	Description
ServiceIntegration sStarted	Number of started Service Tasks.
ServiceIntegration sSucceeded	Number of successfully completed Service Tasks.
ServiceIntegration sTimedOut	Number of Service Tasks that time out on close.

Dimension for Step Functions Service Integration Metrics

Dimension	Description
ServiceIntegration ResourceArn	The resource ARN of the integrated service.

Service Metrics

The AWS/States namespace includes the following metrics for the Step Functions service.

Metric	Description
ThrottledEvents	Count of requests that have been throttled.
ProvisionedBuckets Size	Count of available requests per second.
ProvisionedRefillR ate	Count of requests per second that are allowed into the bucket.
ConsumedCapacity	Count of requests per second.

Dimension for Step Functions Service Metrics

Dimension	Description
ServiceMetric	Filters data to show State Transitions metrics.

API Metrics

The AWS/States namespace includes the following metrics for the Step Functions API.

Metric	Description
ThrottledEvents	Count of requests that have been throttled.
ProvisionedBucketsSize	Count of available requests per second.
ProvisionedRefillRate	Count of requests per second that are allowed into the bucket.
ConsumedCapacity	Count of requests per second.

Dimension for Step Functions API Metrics

Dimension	Description
APIName	Filters data to an API of the specified API name.

Best-effort CloudWatch metrics delivery

CloudWatch metrics are delivered on a best-effort basis.

The completeness and timeliness of metrics are not guaranteed. The data point for a particular request might be returned with a timestamp that is later than when the request was actually processed. The data point for a minute might be delayed before being available through

CloudWatch, or it might not be delivered at all. CloudWatch request metrics give you an idea of the state machine executions in near-real time. It is not meant to be a complete accounting of all execution-related metrics.

It follows from the best-effort nature of this feature that the reports available at the [Billing & Cost Management Dashboard](#) might include one or more access requests that do not appear in the execution metrics.

Viewing Metrics for Step Functions

1. Sign in to the AWS Management Console and open the CloudWatch console.
2. Choose **Metrics**, and on the **All Metrics** tab, choose **States**.

The screenshot shows the AWS CloudWatch console interface. On the left sidebar, the 'Metrics' menu item is highlighted with a red box. The main content area displays an empty graph titled 'Untitled graph' with a message: 'Your CloudWatch graph is empty. Select some metrics to appear here.' Below the graph, there are tabs for 'All metrics', 'Graphed metrics', and 'Graph options'. A search bar is present with the placeholder text 'Search for any metric, dimension or resource id'. Underneath, a list of metrics is shown, with 'States' (36 Metrics) highlighted in a red box.

If you ran any executions recently, you will see up to four types of metrics:

- **Execution Metrics**
 - **Activity Function Metrics**
 - **Lambda Function Metrics**
 - **Service Integration Metrics**
3. Choose a metric type to see a list of metrics.

All metrics		Graphed metrics	Graph options
All	> States	> Execution Metrics	Search for any metric, dimension or resource id
<input type="checkbox"/>	StateMachineArn (18)		Metric Name
<input type="checkbox"/>	arn:aws:states:us-east-1:123456789012:stateMachin		ExecutionTime
<input type="checkbox"/>	arn:aws:states:us-east-1:123456789012:stateMachin		ExecutionsAborted
<input type="checkbox"/>	arn:aws:states:us-east-1:123456789012:stateMachin		ExecutionsTimedOut
<input type="checkbox"/>	arn:aws:states:us-east-1:123456789012:stateMachin		ExecutionsStarted
<input type="checkbox"/>	arn:aws:states:us-east-1:123456789012:stateMachin		ExecutionsSucceeded
<input type="checkbox"/>	arn:aws:states:us-east-1:123456789012:stateMachin		ExecutionsFailed
<input type="checkbox"/>	arn:aws:states:us-east-1:123456789012:stateMachin		ExecutionsSucceeded

- To sort your metrics by **Metric Name** or **StateMachineArn**, use the column headings.
- To view graphs for a metric, choose the box next to the metric on the list. You can change the graph parameters using the time range controls above the graph view.

You can choose custom time ranges using relative or absolute values (specific days and times). You can also use the dropdown list to display values as lines, stacked areas, or numbers (values).

- To view the details about a graph, hover over the metric color code that appears below the graph.

■ ExecutionsAborted ■ ExecutionsStarted ■ ExecutionsSucceeded ■ ExecutionsTimedOut

The metric's details are displayed.

■ States ExecutionsStarted

StateMachineArn: arn:aws:states:us-east-1:123456789012:stateMachine:MyStateMachine-U3WWRPGROPE5

Region: us-east-1

Period: 5 Minutes

Statistic: Sum

Unit: Count

Hold Shift to hide

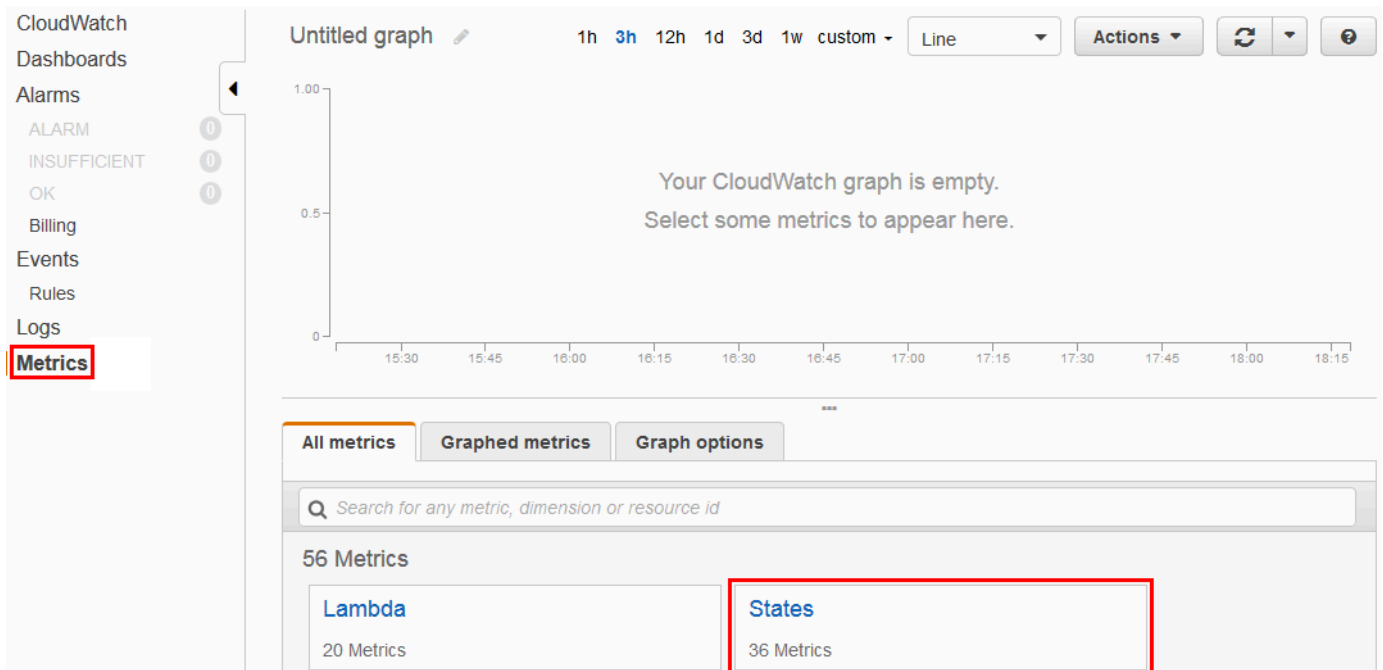
For more information about working with CloudWatch metrics, see [Using Amazon CloudWatch Metrics](#) in the *Amazon CloudWatch User Guide*.

Setting Alarms for Step Functions

You can use Amazon CloudWatch alarms to perform actions. For example, if you want to know when an alarm threshold is reached, you can set an alarm to send a notification to an Amazon SNS topic or to send an email when the `StateMachinesFailed` metric rises above a certain threshold.

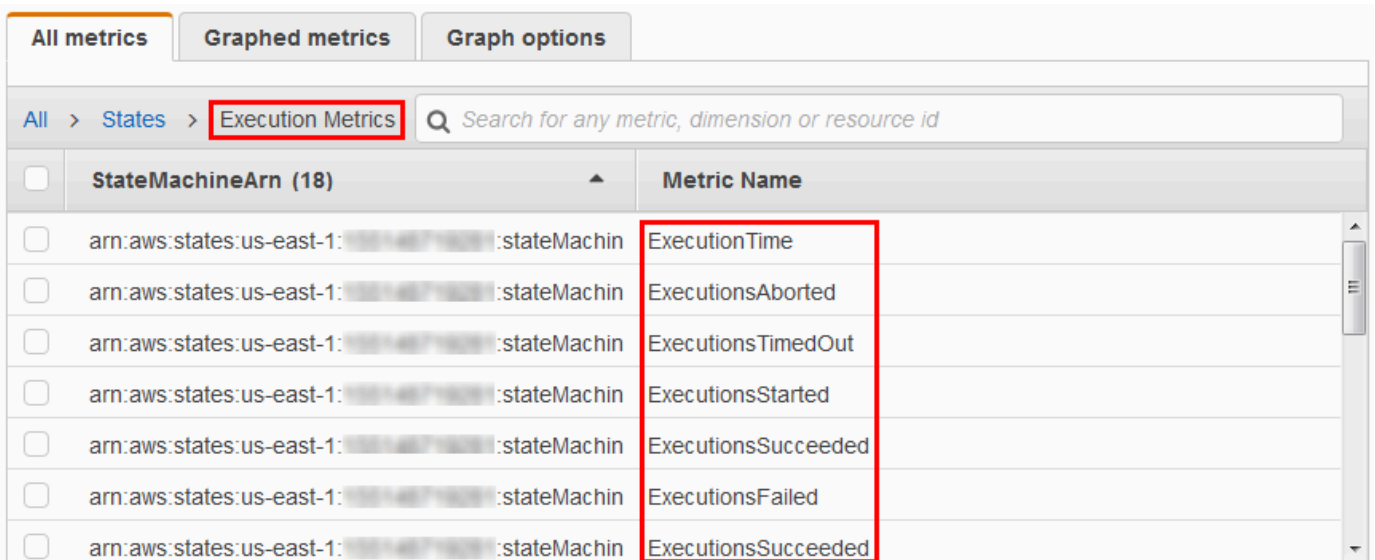
To set an alarm on a metric

1. Sign in to the AWS Management Console and open the CloudWatch console.
2. Choose **Metrics**, and on the **All Metrics** tab, choose **States**.



If you ran any executions recently, you will see up to four types of metrics:

- **Execution Metrics**
 - **Activity Function Metrics**
 - **Lambda Function Metrics**
 - **Service Integration Metrics**
3. Choose a metric type to see a list of metrics.

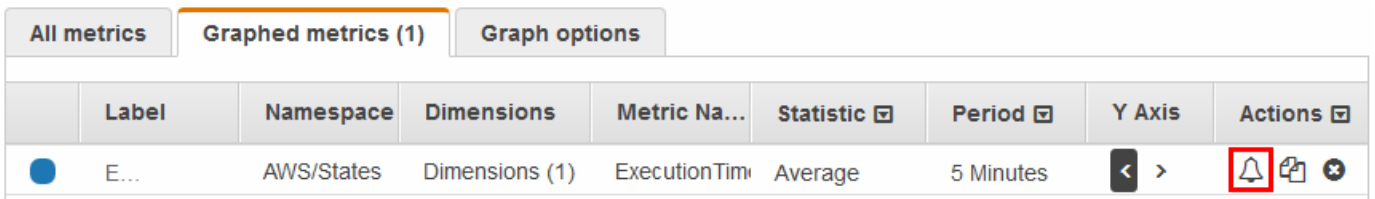


4. Choose a metric, and then choose **Graphed metrics**.

5. Choose



next to a metric on the list.



The **Create Alarm** page is displayed.

Create Alarm ✕

1. Select Metric **2. Define Alarm**

Alarm Threshold

Provide the details and threshold for your alarm. Use the graph on the right to help set the appropriate threshold.

Name:

Description:

Whenever: ExecutionTime

is:

for: consecutive period(s)

Actions

Define what actions are taken when your alarm changes state.

Notification Delete

Whenever this alarm:

Send notification to: New list Enter list ⓘ

Alarm Preview

This alarm will trigger when the blue line goes up to or above the red line for a duration of 5 minutes

ExecutionTime >= 0

Namespace: AWS/States

StateMachine-
Arn:

Metric Name:

Period:

Statistic: Standard Custom

6. Enter the values for the **Alarm threshold** and **Actions**, and then choose **Create Alarm**.

For more information about setting and using CloudWatch alarms, see [Creating Amazon CloudWatch Alarms](#) in the *Amazon CloudWatch User Guide*.

EventBridge (CloudWatch Events) for Step Functions execution status changes

Amazon EventBridge is an AWS service that enables you to respond to state changes in an AWS resource. For example, you can respond to the execution status changes of a Step Functions Standard Workflow with EventBridge using the following two ways:

- You can configure EventBridge rules to react to events that are emitted when the execution status of a Step Functions state machine changes. This enables you to monitor your workflows without having to constantly poll using the [DescribeExecution](#) API. Based on changes in state machine executions, you can use an EventBridge target to start new state machine executions, call AWS Lambda functions, publish messages to Amazon Simple Notification Service (Amazon SNS) topics, and more.
- You can also configure a Step Functions state machine as a target in EventBridge. This enables you to trigger an execution of a Step Functions workflow in response to an event from another AWS service.

For more information, see the [Amazon EventBridge User Guide](#).

Express Workflows, however, do not emit events to EventBridge. To monitor the execution of an Express Workflow, you can use CloudWatch Logs. To do this, on the state machine [Execution Details](#) page, choose the **Monitoring** and **Logging** tabs. On the **Monitoring** tab, you can view the CloudWatch metrics for events, such as **Execution Duration**, **Execution Errors**, and **Billed Memory**. On the **Logging** tab, you can view recent logs and the logging configuration.

Tip

To deploy an example of an Express Workflow to your AWS account and learn how to monitor Express Workflows, see the [Monitoring Express Workflows](#) module of *The AWS Step Functions Workshop*.

EventBridge payloads

An EventBridge event can contain an input property in its definition. For some events, an EventBridge event can also contain an output property in its definition.

- If the combined escaped input and escaped output sent to EventBridge exceeds 248KB, then the input will be excluded. Similarly, if the escaped output exceeds 248KB, then the output will be excluded. This is a result of the EventBridge events quotas.
- You can determine whether a payload has been truncated with the `inputDetails` and `outputDetails` properties. For more information, see the [CloudWatchEventsExecutionDataDetails Data Type](#).
- For Standard Workflows, you can see the full input and output by using [DescribeExecution](#).

- DescribeExecution is not available for Express Workflows. If you want to see the full input/output, you can wrap your Express Workflow with a Standard Workflow. Another option is to use Amazon S3 ARNs. For information about using ARNs, see [the section called “Use Amazon S3 ARNs instead of passing large payloads”](#).

Topics

- [Step Functions event examples](#)
- [Routing a Step Functions event to EventBridge in the EventBridge console](#)

Step Functions event examples

The following are examples of Step Functions sending events to EventBridge:

Topics

- [Execution started](#)
- [Execution succeeded](#)
- [Execution failed](#)
- [Execution timed out](#)
- [Execution aborted](#)

In each case, the detail section in the event data provides the same information as the [DescribeExecution](#) API. The status field indicates the status of the execution at the time the event was sent, one of RUNNING, SUCCEEDED, FAILED, TIMED_OUT, or ABORTED depending on the event emitted.

Execution started

```
{
  "version": "0",
  "id": "315c1398-40ff-a850-213b-158f73e60175",
  "detail-type": "Step Functions Execution Status Change",
  "source": "aws.states",
  "account": "123456789012",
  "time": "2019-02-26T19:42:21Z",
  "region": "us-east-2",
  "resources": [
```

```

    "arn:aws:states:us-east-2:123456789012:execution:state-machine-name:execution-
name"
  ],
  "detail": {
    "executionArn": "arn:aws:states:us-east-2:123456789012:execution:state-machine-
name:execution-name",
    "stateMachineArn": "arn:aws::states:us-east-2:123456789012:stateMachine:state-
machine",
    "name": "execution-name",
    "status": "RUNNING",
    "startDate": 1551225271984,
    "stopDate": null,
    "input": "{}",
    "inputDetails": {
      "included": true
    },
    "output": null,
    "outputDetails": null
  }
}

```

Execution succeeded

```

{
  "version": "0",
  "id": "315c1398-40ff-a850-213b-158f73e60175",
  "detail-type": "Step Functions Execution Status Change",
  "source": "aws.states",
  "account": "123456789012",
  "time": "2019-02-26T19:42:21Z",
  "region": "us-east-2",
  "resources": [
    "arn:aws:states:us-east-2:123456789012:execution:state-machine-name:execution-
name"
  ],
  "detail": {
    "executionArn": "arn:aws:states:us-east-2:123456789012:execution:state-machine-
name:execution-name",
    "stateMachineArn": "arn:aws:states:us-east-2:123456789012:stateMachine:state-
machine",
    "name": "execution-name",
    "status": "SUCCEEDED",
    "startDate": 1547148840101,

```

```
    "stopDate": 1547148840122,
    "input": "{}",
    "inputDetails": {
      "included": true
    },
    "output": "\"Hello World!\"",
    "outputDetails": {
      "included": true
    }
  }
}
```

Execution failed

```
{
  "version": "0",
  "id": "315c1398-40ff-a850-213b-158f73e60175",
  "detail-type": "Step Functions Execution Status Change",
  "source": "aws.states",
  "account": "123456789012",
  "time": "2019-02-26T19:42:21Z",
  "region": "us-east-2",
  "resources": [
    "arn:aws:states:us-east-2:123456789012:execution:state-machine-name:execution-name"
  ],
  "detail": {
    "executionArn": "arn:aws:states:us-east-2:123456789012:execution:state-machine-name:execution-name",
    "stateMachineArn": "arn:aws:states:us-east-2:123456789012:stateMachine:state-machine",
    "name": "execution-name",
    "status": "FAILED",
    "startDate": 1551225146847,
    "stopDate": 1551225151881,
    "input": "{}",
    "inputDetails": {
      "included": true
    },
    "output": null,
    "outputDetails": null
  }
}
```


Execution timed out

```
{
  "version": "0",
  "id": "315c1398-40ff-a850-213b-158f73e60175",
  "detail-type": "Step Functions Execution Status Change",
  "source": "aws.states",
  "account": "123456789012",
  "time": "2019-02-26T19:42:21Z",
  "region": "us-east-2",
  "resources": [
    "arn:aws:states:us-east-2:123456789012:execution:state-machine-name:execution-name"
  ],
  "detail": {
    "executionArn": "arn:aws:states:us-east-2:123456789012:execution:state-machine-name:execution-name",
    "stateMachineArn": "arn:aws:states:us-east-2:123456789012:stateMachine:state-machine",
    "name": "execution-name",
    "status": "TIMED_OUT",
    "startDate": 1551224926156,
    "stopDate": 1551224927157,
    "input": "{}",
    "inputDetails": {
      "included": true
    },
    "output": null,
    "outputDetails": null
  }
}
```

Execution aborted

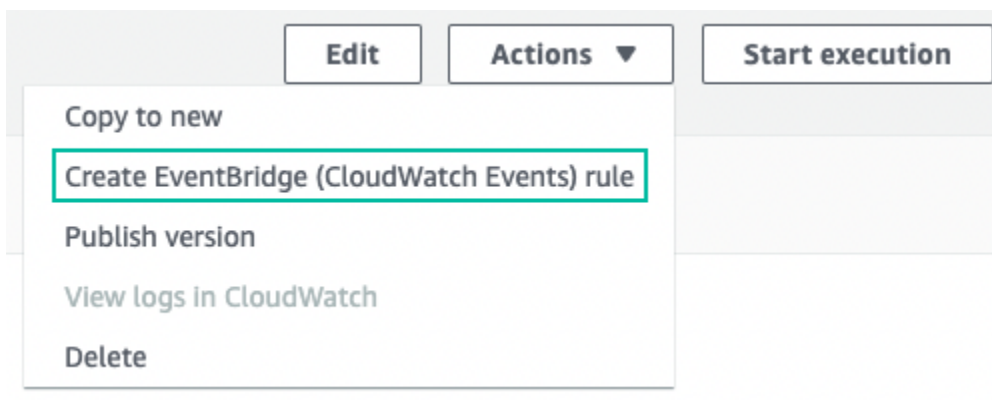
```
{
  "version": "0",
  "id": "315c1398-40ff-a850-213b-158f73e60175",
  "detail-type": "Step Functions Execution Status Change",
  "source": "aws.states",
  "account": "123456789012",
  "time": "2019-02-26T19:42:21Z",
  "region": "us-east-2",
```

```
"resources": [  
  "arn:aws:states:us-east-2:123456789012:execution:state-machine-name:execution-name"  
],  
"detail": {  
  "executionArn": "arn:aws:states:us-east-2:123456789012:execution:state-machine-name:execution-name",  
  "stateMachineArn": "arn:aws:states:us-east-2:123456789012:stateMachine:state-machine",  
  "name": "execution-name",  
  "status": "ABORTED",  
  "startDate": 1551225014968,  
  "stopDate": 1551225017576,  
  "input": "{}",  
  "inputDetails": {  
    "included": true  
  },  
  "output": null,  
  "outputDetails": null  
}  
}
```

Routing a Step Functions event to EventBridge in the EventBridge console

Use the following instructions to learn how to trigger the execution of a Step Functions state machine whenever a specific Step Functions state machine completes running successfully. You use the Amazon EventBridge console to specify the state machine whose execution you want to trigger.

1. On the **Details** page of a state machine, choose **Actions**, and then choose **Create EventBridge (CloudWatch Events) rule**.



- Alternatively, open the EventBridge console at <https://console.aws.amazon.com/events/>. In the navigation pane, choose **Rules** under **Buses**.
2. Choose **Create rule**. This opens the **Define rule detail** page.
 3. Enter a **Name** for your rule (for example, *StepFunctionsEventRule*) and optionally enter a **Description** for the rule.
 4. For **Event bus** and **Rule type**, keep the default selections.
 5. Choose **Next**. This opens the **Build event pattern** page.
 6. Under **Event Source**, keep the default selection of **AWS events or EventBridge partner events**.
 7. Keep the default selections for the **Sample event** and **Creation method** sections.
 8. Under **Event pattern**, do the following:
 - a. In the **Event source** dropdown list, keep the default selection of **AWS services**.
 - b. From the **AWS service** dropdown list, choose **Step Functions**.
 - c. From the **Event type** dropdown list, select **Step Functions Execution Status Change**.
 - d. (Optional) Configure a specific status, state machine Amazon Resource Name (ARN), or execution ARN. For this procedure, choose **Specific status(es)**, and then choose **SUCCEEDED** from the dropdown list.
 9. Choose **Next**. This opens the **Select target(s)** page.
 10. Under **Target types**, keep the default selection of **AWS service**.
 11. From the **Select a target** dropdown list, choose an AWS service. For example, you could launch a Lambda function, or run a Step Functions state machine. For this procedure, choose **Step Functions state machine**.
 12. From the **State machine** dropdown list, choose a state machine.
 13. Under **Execution role**, keep the default selection of **Create a new role for this specific resource**.
 14. Choose **Next**. This opens the **Configure tags** page.
 15. choose **Next** again. This opens the **Review and create** page.
 16. Review the details of the rule and choose **Create rule**.

The rule is created and the **Rules** page is displayed, listing all your Amazon EventBridge rules.

Recording API calls with AWS CloudTrail

AWS Step Functions is integrated with [AWS CloudTrail](#), a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures all API calls for Step Functions as events. The calls captured include calls from the Step Functions console and code calls to the Step Functions API operations. Using the information collected by CloudTrail, you can determine the request that was made to Step Functions, the IP address from which the request was made, when it was made, and additional details.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root user or user credentials.
- Whether the request was made on behalf of an IAM Identity Center user.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

CloudTrail is active in your AWS account when you create the account and you automatically have access to the CloudTrail **Event history**. The CloudTrail **Event history** provides a viewable, searchable, downloadable, and immutable record of the past 90 days of recorded management events in an AWS Region. For more information, see [Working with CloudTrail Event history](#) in the *AWS CloudTrail User Guide*. There are no CloudTrail charges for viewing the **Event history**.

For an ongoing record of events in your AWS account past 90 days, create a trail or a [CloudTrail Lake](#) event data store.

CloudTrail trails

A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. All trails created using the AWS Management Console are multi-Region. You can create a single-Region or a multi-Region trail by using the AWS CLI. Creating a multi-Region trail is recommended because you capture activity in all AWS Regions in your account. If you create a single-Region trail, you can view only the events logged in the trail's AWS Region. For more information about trails, see [Creating a trail for your AWS account](#) and [Creating a trail for an organization](#) in the *AWS CloudTrail User Guide*.

You can deliver one copy of your ongoing management events to your Amazon S3 bucket at no charge from CloudTrail by creating a trail, however, there are Amazon S3 storage charges. For

more information about CloudTrail pricing, see [AWS CloudTrail Pricing](#). For information about Amazon S3 pricing, see [Amazon S3 Pricing](#).

CloudTrail Lake event data stores

CloudTrail Lake lets you run SQL-based queries on your events. CloudTrail Lake converts existing events in row-based JSON format to [Apache ORC](#) format. ORC is a columnar storage format that is optimized for fast retrieval of data. Events are aggregated into *event data stores*, which are immutable collections of events based on criteria that you select by applying [advanced event selectors](#). The selectors that you apply to an event data store control which events persist and are available for you to query. For more information about CloudTrail Lake, see [Working with AWS CloudTrail Lake](#) in the *AWS CloudTrail User Guide*.

CloudTrail Lake event data stores and queries incur costs. When you create an event data store, you choose the [pricing option](#) you want to use for the event data store. The pricing option determines the cost for ingesting and storing events, and the default and maximum retention period for the event data store. For more information about CloudTrail pricing, see [AWS CloudTrail Pricing](#).

Data events in CloudTrail

[Data events](#) provide information about the resource operations performed on or in a resource (for example, reading or writing to an Amazon S3 object). These are also known as data plane operations. Data events are often high-volume activities. By default, CloudTrail doesn't log data events. The CloudTrail **Event history** doesn't record data events.

Additional charges apply for data events. For more information about CloudTrail pricing, see [AWS CloudTrail Pricing](#).

You can log data events for the Step Functions resource types by using the CloudTrail console, AWS CLI, or CloudTrail API operations. For more information about how to log data events, see [Logging data events with the AWS Management Console](#) and [Logging data events with the AWS Command Line Interface](#) in the *AWS CloudTrail User Guide*.

The following table lists the Step Functions resource types for which you can log data events. The **Data event type** column shows the value to choose from the **Data event type** list on the CloudTrail console. The **resources.type value** column shows the `resources.type` value, which you would specify when configuring advanced event selectors using the AWS CLI or CloudTrail APIs. The **Data APIs logged to CloudTrail** column shows the API calls logged to CloudTrail for the resource type.

You can configure advanced event selectors to filter on the `eventName`, `readOnly`, and `resources.ARN` fields to log only those events that are important to you. For more information about these fields, see [AdvancedFieldSelector](#) in the *AWS CloudTrail API Reference*.

Data event type	resources.type value	Data APIs logged to CloudTrail
Step Functions state machine	AWS::StepFunctions::StateMachine	<ul style="list-style-type: none">InvokeHTTPEndpoint

Management events in CloudTrail

[Management events](#) provide information about management operations that are performed on resources in your AWS account. These are also known as control plane operations. By default, CloudTrail logs management events.

State Machine

- [CreateStateMachine](#)
- [ListStateMachines](#)
- [DescribeStateMachine](#)
- [UpdateStateMachine](#)
- [DeleteStateMachine](#)
- [ValidateStateMachineDefinition](#)
- [TestState](#)

State Machine Alias

- [CreateStateMachineAlias](#)
- [ListStateMachineAliases](#)
- [DescribeStateMachineAlias](#)
- [UpdateStateMachineAlias](#)
- [DeleteStateMachineAlias](#)

State Machine Version

- [ListStateMachineVersions](#)
- [PublishStateMachineVersion](#)
- [DeleteStateMachineVersion](#)

Executions

- [StartExecution](#)
- [StartSyncExecution](#)
- [RedriveExecution](#)
- [ListExecutions](#)
- [DescribeExecution](#)
- [GetExecutionHistory](#)
- [DescribeStateMachineForExecution](#)
- [StopExecution](#)

Activity

- [CreateActivity](#)
- [ListActivities](#)
- [DescribeActivity](#)
- [DeleteActivity](#)
- [GetActivityTask](#)

Task Token

- [SendTaskSuccess](#)
- [SendTaskHeartbeat](#)
- [SendTaskFailure](#)

MapRun

- [ListMapRuns](#)

- [DescribeMapRun](#)
- [UpdateMapRun](#)

Tags

- [ListTagsForResource](#)
- [TagResource](#)
- [UntagResource](#)

Event examples

An event represents a single request from any source and includes information about the requested API operation, the date and time of the operation, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so events don't appear in any specific order.

The following example shows a CloudTrail **data event** that demonstrates `InvokeHTTPEndpoint`.

```
{
  "eventVersion": "1.09",
  "userIdentity": {
    "accountId": "123456789012",
    "invokedBy": "states.amazonaws.com"
  },
  "eventTime": "2024-05-01T01:23:45Z",
  "eventSource": "states.amazonaws.com",
  "eventName": "InvokeHTTPEndpoint",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "states.amazonaws.com",
  "userAgent": "states.amazonaws.com",
  "requestParameters": null,
  "responseElements": null,
  "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEeaaaa",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::StepFunctions::StateMachine",
      "ARN": "arn:aws:states:us-
east-1:123456789012:stateMachine:ExampleStateMachine"
```



```
    }
  ],
  "eventType": "AwsServiceEvent",
  "managementEvent": false,
  "recipientAccountId": "123456789012",
  "serviceEventDetails": {
    "httpMethod": "GET",
    "httpEndpoint": "https://example.com"
  },
  "eventCategory": "Data"
}
```

The following example shows a CloudTrail **management event** that demonstrates the `CreateStateMachine` operation.

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDAJYDLDBVBI4EXAMPLE",
    "arn": "arn:aws:iam::123456789012:user/test-user",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "userName": "test-user"
  },
  "eventTime": "2024-05-01T01:23:45Z",
  "eventSource": "states.amazonaws.com",
  "eventName": "CreateStateMachine",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "AWS Internal",
  "userAgent": "AWS Internal",
  "requestParameters": {
    "name": "MyStateMachine",
    "definition": "HIDDEN_DUE_TO_SECURITY_REASONS",
    "roleArn": "arn:aws:iam::123456789012:role/MyStateMachineRole",
    "type": "STANDARD",
    "loggingConfiguration": {
      "level": "OFF",
      "includeExecutionData": false
    },
    "tags": [],
    "tracingConfiguration": {
      "enabled": false
    }
  }
}
```

```
    },
    "publish": false
  },
  "responseElements": {
    "stateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:MyStateMachine",
    "creationDate": "May 1, 2024 1:23:45 AM"
  },
  "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
  "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
  "readOnly": false,
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "123456789012",
  "eventCategory": "Management"
}
```

For information about CloudTrail record contents, see [CloudTrail record contents](#) in the *AWS CloudTrail User Guide*.

Logging using CloudWatch Logs

Standard Workflows record execution history in AWS Step Functions, although you can optionally configure logging to Amazon CloudWatch Logs.

Unlike Standard Workflows, Express Workflows don't record execution history in AWS Step Functions. To see execution history and results for an Express Workflow, you must configure logging to Amazon CloudWatch Logs. Publishing logs doesn't block or slow down executions.

Note

When you configure logging, [CloudWatch Logs charges](#) will apply and you will be billed at the vended logs rate. For more information, see **Vended Logs** under the **Logs** tab on the CloudWatch Pricing page.

Configure logging

When you create a Standard Workflow using the Step Functions console, it will not be configured to enable logging to CloudWatch Logs. An Express Workflow created using the Step Functions console will by default be configured to enable logging to CloudWatch Logs.

For Express workflows, Step Functions can create a role with the necessary AWS Identity and Access Management (IAM) policy for CloudWatch Logs. If you create a Standard Workflow, or an Express Workflow using the API, CLI, or AWS CloudFormation, Step Functions will not enable logging by default, and you will need ensure your role has the necessary permissions.

For each execution started from the console, Step Functions provides a link to CloudWatch Logs, configured with the correct filter to fetch log events specific for that execution.

To configure logging, you can pass the [LoggingConfiguration](#) parameter when using [CreateStateMachine](#) or [UpdateStateMachine](#). You can further analyze your data in CloudWatch Logs by using CloudWatch Logs Insights. For more information see [Analyzing Log Data with CloudWatch Logs Insights](#).

CloudWatch Logs payloads

Execution history events may contain either input or output properties in their definitions. If escaped input or escaped output sent to CloudWatch Logs exceeds 248KB, it will be truncated as a result of CloudWatch Logs quotas.

- You can determine whether a payload has been truncated by reviewing the `inputDetails` and `outputDetails` properties. For more information, see the [HistoryEventExecutionDataDetails Data Type](#).
- For Standard Workflows, you can see the full execution history by using [GetExecutionHistory](#).
- `GetExecutionHistory` is not available for Express Workflows. If you want to see the full input and output, you can use Amazon S3 ARNs. For more information, see [the section called "Use Amazon S3 ARNs instead of passing large payloads"](#).

IAM Policies for logging to CloudWatch Logs

You will also need to configure your state machine's execution IAM role to have the proper permission to log to CloudWatch Logs as shown in the following example.

IAM policy example

The following is an example policy you can use to configure your permissions. As shown in the following example, you need to specify `*` in the `Resource` field because CloudWatch API actions, such as `CreateLogDelivery` and `DescribeLogGroups`, don't support [Resource types defined by](#)

[Amazon CloudWatch Logs](#). For more information, see [Actions defined by Amazon CloudWatch Logs](#).

- For information about CloudWatch resources, see [CloudWatch Logs resources and operations](#) in the *Amazon CloudWatch User Guide*.
- For information about the permissions you need to set up sending logs to CloudWatch Logs, see [User permissions](#) in the section titled *Logs sent to CloudWatch Logs*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogDelivery",
        "logs:CreateLogStream",
        "logs:GetLogDelivery",
        "logs:UpdateLogDelivery",
        "logs>DeleteLogDelivery",
        "logs:ListLogDeliveries",
        "logs:PutLogEvents",
        "logs:PutResourcePolicy",
        "logs:DescribeResourcePolicies",
        "logs:DescribeLogGroups"
      ],
      "Resource": "*"
    }
  ]
}
```

Unable to access the CloudWatch Logs

If you're unable to access the CloudWatch Logs, make sure you've done the following:

1. Configured your state machine's execution IAM role to have the proper permission to log to CloudWatch Logs.

If you're using the [CreateStateMachine](#) or [UpdateStateMachine](#) requests, make sure you've specified the IAM role in the `roleArn` parameter containing the permissions as shown in the [preceding example](#).

2. Checked the CloudWatch Logs resource policy doesn't exceed the 5120 character limit for CloudWatch Logs resource policies.

If you've exceeded the character limit, remove unnecessary permissions from the CloudWatch Logs resource policy, or prefix the log group name with `/aws/vendedlogs`, which will grant permissions to the log group without appending more characters to the resource policy. When you create a log group in the Step Functions console, the log group names are prefixed with `/aws/vendedlogs/states`. For more information, see [Amazon CloudWatch Logs resource policy size restrictions](#).

Log levels

You can choose from OFF, ALL, ERROR, or FATAL. No event types log when set to OFF and all event types do when set to ALL. For ERROR and FATAL, see the following table.

For more information about the execution data displayed for Express Workflow executions based on these **Log levels**, see [Standard and Express Workflow executions in the console](#).

Event Type	ALL	ERROR	FATAL	OFF
ChoiceStateEntered	✓			
ChoiceStateExited	✓			
ExecutionAborted	✓	✓	✓	
ExecutionFailed	✓	✓	✓	
ExecutionStarted	✓			
ExecutionSucceeded	✓			
ExecutionTimedOut	✓	✓	✓	

Event Type	ALL	ERROR	FATAL	OFF
FailStateEntered	✓	✓		
LambdaFunctionFailed	✓	✓		
LambdaFunctionScheduled	✓			
LambdaFunctionScheduleFailed	✓	✓		
LambdaFunctionStarted	✓			
LambdaFunctionStartFailed	✓	✓		
LambdaFunctionSucceeded	✓			
LambdaFunctionTimedOut	✓	✓		
MapIterationAborted	✓	✓		
MapIterationFailed	✓	✓		
MapIterationStarted	✓			
MapIterationSucceeded	✓			
MapRunAborted	✓	✓		

Event Type	ALL	ERROR	FATAL	OFF
MapRunFailed	✓	✓		
MapStateAborted	✓	✓		
MapStateEntered	✓			
MapStateExited	✓			
MapStateFailed	✓	✓		
MapStateStarted	✓			
MapStateSucceeded	✓			
ParallelStateAborted	✓	✓		
ParallelStateEntered	✓			
ParallelStateExited	✓			
ParallelStateFailed	✓	✓		
ParallelStateStarted	✓			
ParallelStateSucceeded	✓			
PassStateEntered	✓			

Event Type	ALL	ERROR	FATAL	OFF
PassStateExited	✓			
SucceedStateEntered	✓			
SucceedStateExited	✓			
TaskFailed	✓	✓		
TaskScheduled	✓			
TaskStarted	✓			
TaskStartFailed	✓	✓		
TaskStateAborted	✓	✓		
TaskStateEntered	✓			
TaskStateExited	✓			
TaskSubmitFailed	✓	✓		
TaskSubmitted	✓			
TaskSucceeded	✓			
TaskTimedOut	✓	✓		
WaitStateAborted	✓	✓		
WaitStateEntered	✓			

Event Type	ALL	ERROR	FATAL	OFF
WaitStateExited	✓			

AWS X-Ray and Step Functions

You can use [AWS X-Ray](#) to visualize the components of your state machine, identify performance bottlenecks, and troubleshoot requests that resulted in an error. Your state machine sends trace data to X-Ray, and X-Ray processes the data to generate a service map and searchable trace summaries.

With X-Ray enabled for your state machine, you can trace requests as they are executed in Step Functions, in all AWS Regions where X-Ray is available. This gives you a detailed overview of an entire Step Functions request. Step Functions will send traces to X-Ray for state machine executions, even when a trace ID is not passed by an upstream service. You can use an X-Ray service map to view the latency of a request, including any AWS services that are integrated with X-Ray. You can also configure sampling rules to tell X-Ray which requests to record, and at what sampling rates, according to criteria that you specify.

When X-Ray is not enabled for your state machine, and an upstream service does not pass a trace ID, Step Functions will not send traces to X-Ray for state machine executions. However, if a trace ID is passed by an upstream service, Step Functions will then send traces to X-Ray for state machine executions.

You can use AWS X-Ray with Step Functions in regions where both are supported. See the [Step Functions](#) and [X-Ray](#) endpoints and quotas pages for information on region support for X-Ray and Step Functions.

X-Ray and Step Functions Combined Quotas

You can add data to a trace for up to seven days, and query trace data going back thirty days, the length of time that X-Ray stores trace data. Your traces will be subject to X-Ray quotas. In addition to other quotas, X-Ray provides a minimum guaranteed trace size of 100KB for Step Functions state machines. If more than 100KB of trace data is provided to X-Ray, this may result in a frozen trace. See the service quotas section of the [X-Ray endpoints and quotas](#) page for more information on other quotas for X-Ray.

Important

Step Functions doesn't support X-Ray tracing for the child workflow executions started by a [Distributed Map state](#) because it's easy to exceed the [Trace document size limit](#) for such executions.

Topics

- [Setup and configuration](#)
- [Concepts](#)
- [Step Functions service integrations and X-Ray](#)
- [Viewing the X-Ray console](#)
- [Viewing X-Ray tracing information for Step Functions](#)
- [Traces](#)
- [Service map](#)
- [Segments and subsegments](#)
- [Analytics](#)
- [Configuration](#)
- [What if there is no data in the trace map or service map?](#)

Setup and configuration

Enable X-Ray tracing when creating a state machine


You can enable X-Ray tracing when creating a new state machine by selecting **Enable X-Ray tracing** on the **Specify details** page.

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. On the **Choose authoring method** page, choose an appropriate option to create your state machine. If you choose **Run a sample project**, you cannot enable X-Ray tracing during the state machine creation, and you will need to enable X-Ray tracing after your state machine has been created. For more information about enabling X-Ray in an existing state machine, see [Enable X-Ray in an existing state machine](#).

Choose **Next**.

3. On the **Specify details** page, configure your state machine.
4. Choose **Enable X-Ray tracing**.

Tracing

You can enable AWS X-Ray tracing on your state machine for end-to-end application debugging, performance profiling, and error analysis. Standard X-Ray charges apply. [Learn more](#) 

Enable X-Ray tracing
Step Functions will send traces to AWS X-Ray for state machine executions, even when a trace ID is not passed by an upstream service.

Your Step Functions state machine will now send traces to X-Ray for state machine executions.

Note


If you choose to use an existing IAM role, you should ensure that X-Ray writes are allowed. For more information about the permissions that you need, see [IAM policies for X-Ray](#).

Enable X-Ray in an existing state machine

To enable X-Ray in an existing state machine:


1. In the [Step Functions console](#), select the state machine for which you want to enable tracing.
2. Choose **Edit**.
3. Choose **Enable X-Ray tracing**.

Tracing

You can enable AWS X-Ray tracing on your state machine for end-to-end application debugging, performance profiling, and error analysis. Standard X-Ray charges apply. [Learn more](#) 

Enable X-Ray tracing
Step Functions will send traces to AWS X-Ray for state machine executions, even when a trace ID is not passed by an upstream service.

You will see a notification telling you that you that you may need to make additional changes.

 **Note**

When you enable X-Ray for an existing state machine, you must ensure that you have an IAM policy that grants sufficient permissions for X-Ray to perform traces. You can either add one manually, or generate one. For more information, see the IAM policy section for [IAM policies for AWS X-Ray](#).

4. (Optional) Auto-generate a new role for your state machine to include X-Ray permissions.
5. Choose **Save**.

Configure X-Ray tracing for Step Functions

When you first run a state machine with X-Ray tracing enabled, it will use the default configuration values for X-Ray tracing. AWS X-Ray does not collect data for every request that is sent to an application. Instead, it collects data for a statistically significant number of requests. The default is to record the first request each second, and five percent of any additional requests. One request per second is the *reservoir*. This ensures that at least one trace is recorded each second as long as the service is serving requests. Five percent is the *rate* at which additional requests beyond the reservoir size are sampled.

To avoid incurring service charges when you are getting started, the default sampling rate is conservative. You can configure X-Ray to modify the default sampling rule and configure additional rules that apply sampling based on properties of the service or request.

For example, you might want to disable sampling and trace all requests for calls that modify state or handle AWS accounts or transactions. For high-volume read-only calls, like background polling, health checks, or connection maintenance, you can sample at a low rate and still get enough data to observe issues that occur.

To configure a sampling rule for your state machine:

1. Go to the [X-Ray console](#).
2. Choose **Sampling**.
3. To create a rule, choose **Create sampling rule**.

To edit a rule, choose a rule's name.

To delete a rule, choose a rule and use the **Actions** menu to delete it.

Some parts of existing sampling rules, such as the name and priority, cannot be changed. Instead, add or clone an existing rule, make the changes you want, then use the new rule.

For detailed information on X-Ray sampling rules and how to configure the various parameters, see [Configuring sampling rules in the X-Ray console](#).

Integrate upstream services

To integrate the execution of Step Functions workflows, such as Express, Synchronous, and Standard workflows, with an upstream service you need to set the `traceHeader`. This is automatically done for you if you are using a HTTP API in API Gateway. However, if you're using a Lambda function and/or an SDK, you need to set the `traceHeader` on the [StartExecution](#) or [StartSyncExecution](#) API calls yourself.

You must specify the `traceHeader` format as `\p{ASCII}#`. Additionally, to let Step Functions use the same trace ID, you must specify the format as `Root={TRACE_ID};Sampled={1 or 0}`. If you're using a Lambda function, replace the `TRACE_ID` with the trace ID in your current segment and set the `Sampled` field as `1` if your sampling mode is true and `0` if your sampling mode is false. Providing the trace ID in this format ensures that you'll get a complete trace.

The following is an example written in Python to showcase how to specify the `traceHeader`.

```
state_machine = config.get_string_parameter("STATE_MACHINE_ARN")
if (xray_recorder.current_subsegment() is not None and
    xray_recorder.current_subsegment().sampled) :
    trace_id = "Root={};Sampled=1".format(
        xray_recorder.current_subsegment().trace_id
    )
else:
    trace_id = "Root=not enabled;Sampled=0"
LOGGER.info("trace %s", trace_id)

# execute it
response = states.start_sync_execution(
    stateMachineArn=state_machine,
    input=event['body'],
    name=context.aws_request_id,
    traceHeader=trace_id
```

```
)  
LOGGER.info(response)
```

Concepts

The X-Ray console

The AWS X-Ray console enables you to view service maps and traces for requests that your applications serve. You can access the console to view detailed information collected by X-Ray when it's enabled for your state machine.

See [Viewing the X-Ray console](#) for information on how to access the X-Ray console for your state machine executions.

For detailed information about the X-Ray console, see the [X-Ray console documentation](#).

Segments, subsegments, and traces

A **segment** records information about a request to your state machine. It contains information such as the work that your state machine performs, and may also contain **subsegments** with information about downstream calls.

A **trace** collects all the segments generated by a single request.

Sampling

To ensure efficient tracing and provide a representative sample of the requests that your application serves, X-Ray applies a **sampling** algorithm to determine which requests get traced. This can be changed by editing the sampling rules.

Metrics

For your state machine, X-Ray will meter invocation time, state transition time, the overall execution time of Step Functions, and variances in this execution time. This information can be accessed through the X-Ray console.

Analytics

The AWS X-Ray Analytics console is an interactive tool for interpreting trace data. You can refine the active dataset with increasingly granular filters by clicking the graphs and the panels of metrics

and fields that are associated with the current trace set. This lets you analyze how your state machine is performing, and quickly locate and identify performance issues.

For detailed information about X-Ray analytics, see [Interacting with the AWS X-Ray Analytics console](#)

Step Functions service integrations and X-Ray

Some of the AWS services that integrate with Step Functions provide integration with AWS X-Ray by adding a tracing header to requests, running the X-Ray daemon, or making sampling decisions and uploading trace data to X-Ray. Others must be instrumented using the AWS X-Ray SDK. A few do not yet support X-Ray integration. X-Ray integration is necessary to provide complete trace data when using a service integration with Step Functions

Native X-Ray support

Service integrations with native X-Ray support include:

- [Amazon Simple Notification Service](#)
- [Amazon Simple Queue Service](#)
- [AWS Lambda](#)
- AWS Step Functions

Instrumentation required

Service integrations that require [X-Ray instrumentation](#):

- Amazon Elastic Container Service
- AWS Batch
- AWS Fargate

Client-side trace only

Other service integrations do not support X-Ray traces. However, client side traces can still be collected:

- Amazon DynamoDB
- Amazon EMR

Name	Res.	Duration	Status
WaitForCallbackStateMachine	-	635 ms	⚠️
Start Task And Wait For Callback	-	321 ms	✅
SQS	200	42.0 ms	✅
Notify Success	-	192 ms	⚠️
SNS	403	150 ms	❌
▼ SQS AWS::SQS::Queue (Client Response)			
WaitForCallbackStateMachine	200	42.0 ms	✅
QueueTime	-	42.0 ms	✅
▼ SNS AWS::SNS (Client Response)			
WaitForCallbackStateMachine	403	150 ms	⚠️

Service map

The service map in the X-Ray console helps you to identify services where errors are occurring, where there are connections with high latency, or see traces for requests that were unsuccessful.

Name	Res.	Duration	Status
WaitForCallbackStateMachine	-	635 ms	⚠️
Start Task And Wait For Callback	-	321 ms	✅
SQS	200	42.0 ms	✅
Notify Success	-	192 ms	⚠️
SNS	403	150 ms	❌
▼ SQS AWS::SQS::Queue (Client Response)			
WaitForCallbackStateMachine	200	42.0 ms	✅
QueueTime	-	42.0 ms	✅
▼ SNS AWS::SNS (Client Response)			
WaitForCallbackStateMachine	403	150 ms	⚠️

On the trace map, you can choose a service node to view requests for that node, or an edge between two nodes to view requests that traveled that connection. Here, the `WaitForCallback`

node has been selected, and you can view additional information about its execution and response status.

Segment - WaitForCallbackStateMachine-0T4bSbt6zLcp

Overview

Resources

Annotations

Metadata

Exceptions

Segment ID 0T4bSbt6zLcp
 Parent ID 0T4bSbt6zLcp
 Name WaitForCallbackStateMachine-0T4bSbt6zLcp
 Origin AWS::StepFunctions::StateMachine

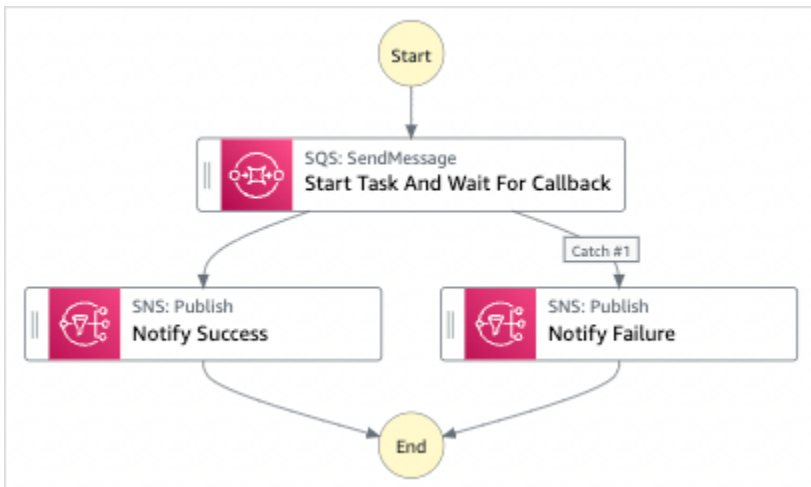
Time

Start time 2020-06-29 20:03:04.379 (UTC)
 End time 2020-06-29 20:03:05.014 (UTC)
 Duration 635 ms
 In progress false

Errors & Faults

Error true
 Fault false

You can see how the X-Ray service map correlates to the state machine. There is a service map node for each service integration that is called by Step Functions, provided it supports X-Ray.



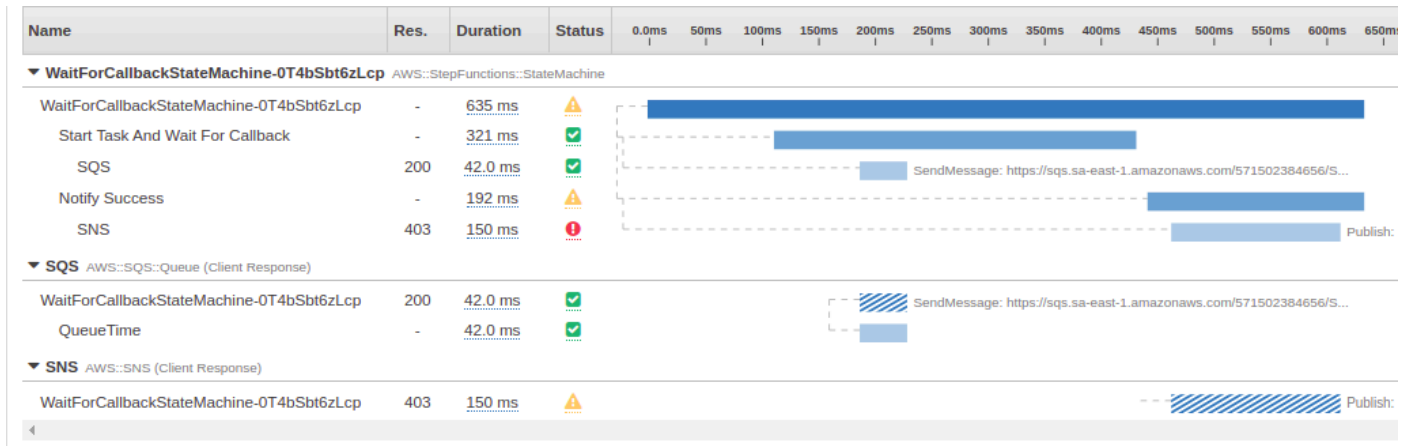
Segments and subsegments

A **trace** is a collection of **segments** generated by a single request. Each segment provides the resource's name, details about the request, and details about the work done. On the **Traces** page, you can see the segments and, if expanded, its corresponding subsegments. You can choose a segment or subsegment to view detailed information about it.

Choose each of the tabs to see how information for segments and subsegments is displayed.

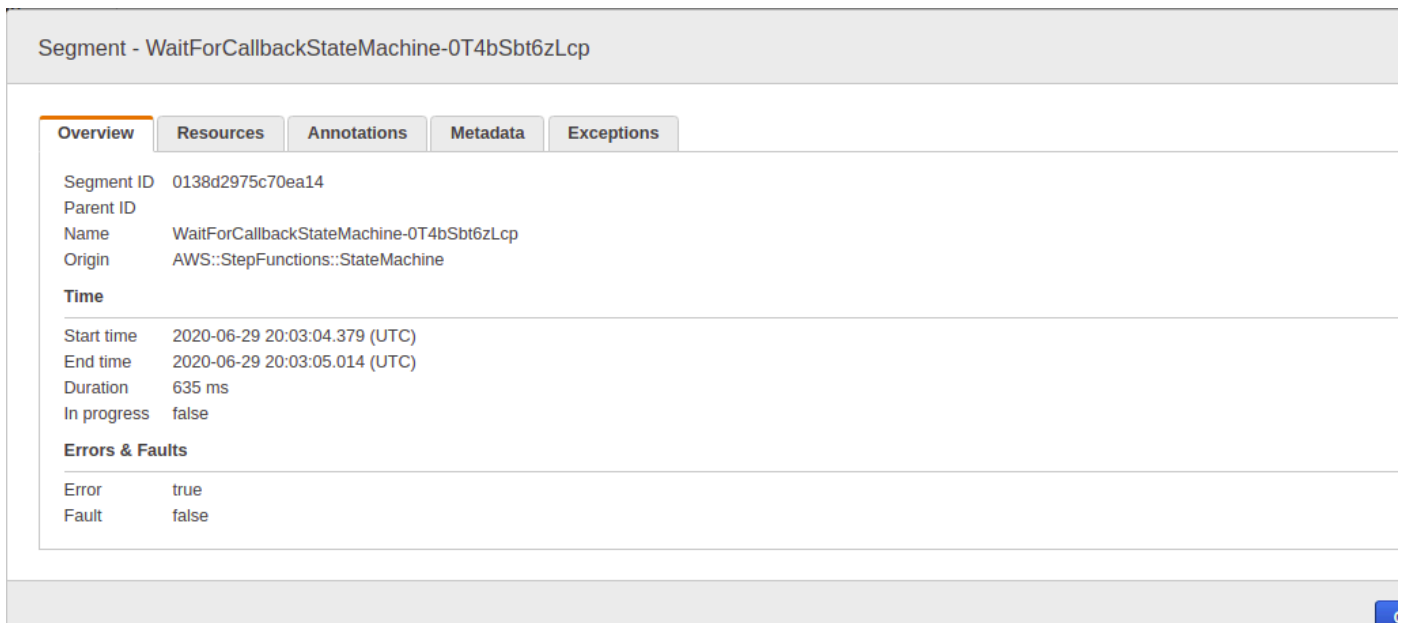
Overview of Segments

An overview of segments and subsegments for this state machine. There is a different segment for each node on the service map.



View segment detail

Choosing a segment provides the resource's name, details about the request, and details about the work done.



View subsegment detail

A segment can break down the data about the work done into subsegments. Choosing a subsegment lets you view more granular timing information and details. A subsegment can

contain additional details about a call to an AWS service, an external HTTP API, or an SQL database.

Subsegment - Start Task And Wait For Callback

OverviewResourcesAnnotationsMetadataExceptions

Subsegment ID	f3ac5e5d10e60d1b
Parent ID	0018287c7c7e9a11
Name	Start Task And Wait For Callback
Time	
Start time	2020-06-29 20:03:04.491 (UTC)
End time	2020-06-29 20:03:04.812 (UTC)
Duration	321 ms
In progress	false
Errors & Faults	
Error	false
Fault	false

Analytics

The AWS X-Ray **Analytics** console is an interactive tool for interpreting trace data. You can use this to more easily understand how your state machine is performing. The console enables you to explore, analyze, and visualize traces through interactive response time and time-series graphs. This can help you quickly locate performance and latency issues.

You can refine the active dataset with increasingly granular filters by clicking the graphs and the panels of metrics and fields that are associated with the current trace set.

All traces in the group ⓘ 1 traces in the group. [Show in charts](#) ⓘ
Complete 100% scanned (found 1 traces)

Retrieved traces ⓘ

1 traces

Filtered trace set A ⓘ

To add a filter, click and drag one of the charts below or click one of the table rows.

+ Compare

(Copy filter trace set A)

Response time distribution ⓘ

Click and drag to filter the traces by response time.

Response time distribution Duration distribution

Time series activity ⓘ

Click and drag to filter the traces by time.

ⓘ Select rows from the following tables to filter traces. Choose the cog icon to explore table configuration options. ⚙️

USER	COUNT	%
-	1	100.00%

HTTP STATUS CODE	COUNT	%
-	1	100.00%

Configuration

You can configure sampling and encryption options from the X-Ray console.

Sampling

Choose **Sampling** to view details about the sampling rate and configuration. You can change the sampling rules to control the amount of data that you record, and modify sampling behavior to suit your specific requirements.

Sampling rules

Customize the default sampling strategy to control cost or filter out unwanted requests by applying sampling rules. By default, you can create up to 25 sampling rules in addition to the default rule. If you'd like to create more than 25 sampling rules, please contact customer support to get the limit increased. [Learn more](#)

Create sampling rule
Actions v ↻

	Priority	Rule	Trend 📈
<input type="checkbox"/>	10000	<p><u>Default</u></p> <ul style="list-style-type: none"> ▪ Service name matches * ▪ Service type matches * ▪ Host matches * ▪ Resource ARN matches * ▪ HTTP method matches * ▪ URL path matches * <p style="border: 1px dashed green; padding: 2px; display: inline-block;">Limit to 1 r/sec, then 5% fixed rate</p>	<p>0 r/sec (0%)</p> <p>1 r/sec</p> <p>0.5 r/sec</p> <p>-5m 0</p>

Encryption

Choose **Encryption** to modify the encryption settings. You can use the default setting, where X-Ray encrypts traces and data at rest, or, if needed, you can choose a customer master key. Standard [AWS KMS](#) charges apply in the latter case.

AWS X-Ray

- Getting started
- Service map
- Traces
- Analytics
- Configuration
- Sampling
- Encryption

Encryption configuration

By default, X-Ray encrypts traces and related data at rest. If you need to encrypt data at rest with a key that you can audit or disable, choose a customer master key from the following list. Standard AWS Key Management Service charges apply. [Learn more](#)

Use default encryption

 Use a customer master key

KMS master key Select a key ↻

Description -
 Account -
 Key ARN -

Cancel Apply changes

What if there is no data in the trace map or service map?

If you have enabled X-Ray, but can't see any data in the X-Ray console, check that:

- Your IAM roles are set up correctly to allow writing to X-Ray.
- Sampling rules allow sampling of data.
- Since there can be a short delay before newly created or modified IAM roles are applied, check the trace or service maps again after a few minutes.

- If you see **Data Not Found** in the X-Ray Traces panel, check your [IAM account settings](#) and ensure that AWS Security Token Service is enabled for the intended region. For more information, see [Activating and deactivating AWS STS in an AWS Region](#) in the *IAM User Guide*.

Using AWS User Notifications with AWS Step Functions

You can use [AWS User Notifications](#) to set up delivery channels to get notified about AWS Step Functions events. You receive a notification when an event matches a rule that you specify. You can receive notifications for events through multiple channels, including email, [AWS Chatbot](#) chat notifications, or [AWS Console Mobile Application](#) push notifications. You can also see notifications in the [Console Notifications Center](#). User Notifications supports aggregation, which can reduce the number of notifications you receive during specific events.

Security in AWS Step Functions

This section provides information about AWS Step Functions security and authentication.

Step Functions uses IAM to control access to other AWS services and resources. For an overview of how IAM works, see [Overview of Access Management](#) in the *IAM User Guide*. For an overview of security credentials, see [AWS Security Credentials](#) in the *Amazon Web Services General Reference*.

Data protection in AWS Step Functions

The AWS [shared responsibility model](#) applies to data protection in AWS Step Functions. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the [Data Privacy FAQ](#). For information about data protection in Europe, see the [AWS Shared Responsibility Model and GDPR](#) blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see [Federal Information Processing Standard \(FIPS\) 140-2](#).

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Step Functions or other AWS services using the console, API, AWS CLI, or AWS

SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Encryption in AWS Step Functions

Encryption at Rest

Step Functions always encrypts your data at rest. Data in AWS Step Functions is encrypted at rest using transparent server-side encryption. This helps reduce the operational burden and complexity involved in protecting sensitive data. With encryption at rest, you can build security-sensitive applications that meet encryption compliance and regulatory requirements

Encryption in transit

Step Functions encrypts data in transit between the service and other integrated AWS services (see [Using AWS Step Functions with other services](#)). All data that passes between Step Functions and integrated services is encrypted using Transport Layer Security (TLS).

Identity and Access Management in AWS Step Functions

Access to AWS Step Functions requires credentials that AWS can use to authenticate your requests. These credentials must have permissions to access AWS resources, such as retrieving event data from other AWS resources. The following sections provide details on how you can use [AWS Identity and Access Management \(IAM\)](#) and Step Functions to help secure your resources by controlling who can access them.

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Step Functions resources. IAM is an AWS service that you can use with no additional charge.

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Step Functions.

Service user – If you use the Step Functions service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Step Functions

features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Step Functions, see [Troubleshooting AWS Step Functions identity and access](#).

Service administrator – If you're in charge of Step Functions resources at your company, you probably have full access to Step Functions. It's your job to determine which Step Functions features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Step Functions, see [How AWS Step Functions works with IAM](#).

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Step Functions. To view example Step Functions identity-based policies that you can use in IAM, see [Identity-based policy examples for AWS Step Functions](#).

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see [How to sign in to your AWS account](#) in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see [Signing AWS API requests](#) in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Multi-factor authentication](#) in the

AWS IAM Identity Center User Guide and [Using multi-factor authentication \(MFA\) in AWS](#) in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see [Tasks that require root user credentials](#) in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see [What is IAM Identity Center?](#) in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An [IAM user](#) is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see [Rotate access keys regularly for use cases that require long-term credentials](#) in the *IAM User Guide*.

An [IAM group](#) is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier

to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see [When to create an IAM user \(instead of a role\)](#) in the *IAM User Guide*.

IAM roles

An [IAM role](#) is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by [switching roles](#). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see [Using IAM roles](#) in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see [Creating a role for a third-party Identity Provider](#) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permission sets, see [Permission sets](#) in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see [Cross account resource access in IAM](#) in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.

- **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see [Forward access sessions](#).
- **Service role** – A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.
- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Using an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see [When to create an IAM role \(instead of a user\)](#) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see [Overview of JSON policies](#) in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies](#) in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see [Choosing between managed policies and inline policies](#) in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list \(ACL\) overview](#) in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [How SCPs work](#) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](#) in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see [Policy evaluation logic](#) in the *IAM User Guide*.

Access Control

You can have valid credentials to authenticate your requests, but unless you have permissions you cannot create or access Step Functions resources. For example, you must have permissions to invoke AWS Lambda, Amazon Simple Notification Service (Amazon SNS), and Amazon Simple Queue Service (Amazon SQS) targets associated with your Step Functions rules.

The following sections describe how to manage permissions for Step Functions.

- [Creating an IAM role for your state machine](#)
- [Creating Granular IAM Permissions for Non-Admin Users](#)
- [Amazon VPC Endpoints for Step Functions](#)
- [IAM Policies for integrated services](#)
- [IAM policies for using Distributed Map state](#)

Policy actions for Step Functions

Supports policy actions

Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Step Functions actions, see [Resources Defined by AWS Step Functions](#) in the *Service Authorization Reference*.

Policy actions in Step Functions use the following prefix before the action:

```
states
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [  
  "states:action1",  
  "states:action2"  
]
```

To view examples of Step Functions identity-based policies, see [Identity-based policy examples for AWS Step Functions](#).

Policy resources for Step Functions

Supports policy resources	Yes
---------------------------	-----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Resource JSON policy element specifies the object or objects to which the action applies. Statements must include either a Resource or a NotResource element. As a best practice, specify a resource using its [Amazon Resource Name \(ARN\)](#). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Step Functions resource types and their ARNs, see [Actions Defined by AWS Step Functions](#) in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see [Resources Defined by AWS Step Functions](#).

To view examples of Step Functions identity-based policies, see [Identity-based policy examples for AWS Step Functions](#).

Policy condition keys for Step Functions

Supports service-specific policy condition keys	Yes
---	-----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Condition element (or Condition *block*) lets you specify conditions in which a statement is in effect. The Condition element is optional. You can create conditional expressions that use [condition operators](#), such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple Condition elements in a statement, or multiple keys in a single Condition element, AWS evaluates them using a logical AND operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical OR operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see [IAM policy elements: variables and tags](#) in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see [AWS global condition context keys](#) in the *IAM User Guide*.

To see a list of Step Functions condition keys, see [Condition Keys for AWS Step Functions](#) in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see [Resources Defined by AWS Step Functions](#).

To view examples of Step Functions identity-based policies, see [Identity-based policy examples for AWS Step Functions](#).

ACLs in Step Functions

Supports ACLs	No
---------------	----

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with Step Functions

Supports ABAC (tags in policies)

Partial

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the [condition element](#) of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see [What is ABAC?](#) in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see [Use attribute-based access control \(ABAC\)](#) in the *IAM User Guide*.

Using temporary credentials with Step Functions

Supports temporary credentials

Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see [AWS services that work with IAM](#) in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see [Switching to a role \(console\)](#) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see [Temporary security credentials in IAM](#).

Cross-service principal permissions for Step Functions

Supports forward access sessions (FAS)	Yes
--	-----

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see [Forward access sessions](#).

Service roles for Step Functions

Supports service roles	Yes
------------------------	-----

A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.

⚠ Warning

Changing the permissions for a service role might break Step Functions functionality. Edit service roles only when Step Functions provides guidance to do so.

Service-linked roles for Step Functions

Supports service-linked roles	No
-------------------------------	----

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see [AWS services that work with IAM](#). Find a service in the table that includes a Yes in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

How AWS Step Functions works with IAM

Before you use IAM to manage access to Step Functions, learn what IAM features are available to use with Step Functions.

IAM features you can use with AWS Step Functions

IAM feature	Step Functions support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys (service-specific)	Yes
ACLs	No

IAM feature	Step Functions support
ABAC (tags in policies)	Partial
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	No

To get a high-level view of how Step Functions and other AWS services work with most IAM features, see [AWS services that work with IAM](#) in the *IAM User Guide*.

Identity-based policy examples for AWS Step Functions

By default, users and roles don't have permission to create or modify Step Functions resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see [Creating IAM policies](#) in the *IAM User Guide*.

For details about actions and resource types defined by Step Functions, including the format of the ARNs for each of the resource types, see [Actions, Resources, and Condition Keys for AWS Step Functions](#) in the *Service Authorization Reference*.

Topics

- [Policy best practices](#)
- [Using the Step Functions console](#)
- [Allow users to view their own permissions](#)

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Step Functions resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see [AWS managed policies](#) or [AWS managed policies for job functions](#) in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see [Policies and permissions in IAM](#) in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see [IAM JSON policy elements: Condition](#) in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see [IAM Access Analyzer policy validation](#) in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see [Configuring MFA-protected API access](#) in the *IAM User Guide*.

For more information about best practices in IAM, see [Security best practices in IAM](#) in the *IAM User Guide*.

Using the Step Functions console

To access the AWS Step Functions console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Step Functions resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the Step Functions console, also attach the Step Functions *ConsoleAccess* or *ReadOnly* AWS managed policy to the entities. For more information, see [Adding permissions to a user](#) in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ViewOwnUserInfo",
      "Effect": "Allow",
      "Action": [
        "iam:GetUserPolicy",
        "iam:ListGroupsWithUser",
        "iam:ListAttachedUserPolicies",
        "iam:ListUserPolicies",
        "iam:GetUser"
      ],
      "Resource": ["arn:aws:iam::*:user/${aws:username}"]
    },
    {
      "Sid": "NavigateInConsole",
      "Effect": "Allow",
      "Action": [
        "iam:GetGroupPolicy",
        "iam:GetPolicyVersion",
```



```
        "iam:GetPolicy",
        "iam:ListAttachedGroupPolicies",
        "iam:ListGroupPolicies",
        "iam:ListPolicyVersions",
        "iam:ListPolicies",
        "iam:ListUsers"
    ],
    "Resource": "*"
}
]
```

Identity-based policies for Step Functions

Supports identity-based policies	Yes
----------------------------------	-----

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies](#) in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see [IAM JSON policy elements reference](#) in the *IAM User Guide*.

Identity-based policy examples for Step Functions

To view examples of Step Functions identity-based policies, see [Identity-based policy examples for AWS Step Functions](#).

Resource-based policies within Step Functions

Supports resource-based policies	No
----------------------------------	----

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that

support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see [Cross account resource access in IAM](#) in the *IAM User Guide*.

AWS managed policies for AWS Step Functions

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining [customer managed policies](#) that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see [AWS managed policies](#) in the *IAM User Guide*.

AWS managed policy: AWSStepFunctionsConsoleFullAccess

You can attach the [AWSStepFunctionsConsoleFullAccess](#) policy to your IAM identities.

This policy grants *administrator* permissions that allow a user access to use the Step Functions console. For a full console experience, a user may also need `iam:PassRole` permission on other IAM roles that can be assumed by the service.

AWS managed policy: AWSStepFunctionsReadOnlyAccess

You can attach the [AWSStepFunctionsReadOnlyAccess](#) policy to your IAM identities.

This policy grants *read-only* permissions that allow a user or role to list and describe state machines, activities, executions, activities, tags, MapRuns, and state machine alias and versions. This policy also grants permission to check the syntax of state machine definitions that you provide.

AWS managed policy: AWSStepFunctionsFullAccess

You can attach the [AWSStepFunctionsFullAccess](#) policy to your IAM identities.

This policy grants *full* permissions to a user or role to use the Step Functions API. For full access, a user must have *iam:PassRole* permission on at least one IAM role that can be assumed by the service.

Step Functions updates to AWS managed policies

View details about updates to AWS managed policies for Step Functions since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Step Functions [Document history](#) page.

Change	Description	Date
AWSStepFunctionsReadOnlyAccess – Update to an existing policy	Step Functions added new permissions to allow calling <code>states:ValidateStateMachineDefinition</code> API action to check the syntax	April 25, 2024

Change	Description	Date
	of state machine definitions that you provide.	
AWSStepFunctionsReadOnlyAccess – Update to an existing policy	Step Functions added new permissions to allow listing and reading data related to: Tags (ListTagsForResource), Distributed Map (ListMapRuns, DescribeMapRun), Versions and Aliases (DescribeStateMachineAlias, ListStateMachineAliases, ListStateMachineVersions).	April 02, 2024
Step Functions started tracking changes	Step Functions started tracking changes for its AWS managed policies.	April 02, 2024

Creating an IAM role for your state machine

AWS Step Functions can execute code and access AWS resources (such as invoking an AWS Lambda function). To maintain security, you must grant Step Functions access to those resources by using an IAM role.

The [Tutorials for Step Functions](#) in this guide enable you to take advantage of automatically generated IAM roles that are valid for the AWS Region in which you create the state machine. However, you can create your own IAM role for a state machine.

When creating an IAM policy for your state machines to use, the policy should include the permissions that you would like the state machines to assume. You can use an existing AWS managed policy as an example or you can create a custom policy from scratch that meets your specific needs. For more information, see [Creating IAM policies](#) in the *IAM User Guide*

To create your own IAM role for a state machine, follow the steps in this section.

In this example, you create an IAM role with permission to invoke a Lambda function.

Create a role for Step Functions

1. Sign in to the [IAM console](#), and then choose **Roles, Create role**.
2. On the **Select trusted entity** page, under **AWS service**, select **Step Functions** from the list, and then choose **Next: Permissions**.
3. On the **Attached permissions policy** page, choose **Next: Review**.
4. On the **Review** page, enter `StepFunctionsLambdaRole` for **Role Name**, and then choose **Create role**.

The IAM role appears in the list of roles.

For more information about IAM permissions and policies, see [Access Management](#) in the *IAM User Guide*.

Prevent cross-service confused deputy issue

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). This type of impersonation can happen cross-account and cross-service. The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access.

To prevent confused deputies, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account. This section focuses on cross-service confused deputy prevention specific to AWS Step Functions; however, you can learn more about this topic in the [confused deputy problem](#) section of the *IAM User Guide*.

We recommend using the [aws:SourceArn](#) and [aws:SourceAccount](#) global condition context keys in resource policies to limit the permissions that Step Functions gives another service to access your resources. Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource, or if you're specifying multiple resources, use the `aws:SourceArn`

global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, `arn:aws:states:*:111122223333:*`.

Here's an example of a *trusted policy* that shows how you can use `aws:SourceArn` and `aws:SourceAccount` with Step Functions to prevent the confused deputy issue.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "states.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:states:us-east-1:111122223333:stateMachine:*"
        },
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        }
      }
    }
  ]
}
```

Attach an Inline Policy

Step Functions can control other services directly in a Task state. Attach inline policies to allow Step Functions to access the API actions of the services you need to control.

1. Open the [IAM console](#), choose **Roles**, search for your Step Functions role, and select that role.
2. Select **Add inline policy**.
3. Use the **Visual editor** or the **JSON** tab to create policies for your role.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

Note

For examples of IAM policies created by the Step Functions console, see [IAM Policies for integrated services](#).

Creating Granular IAM Permissions for Non-Admin Users

The default managed policies in IAM, such as `ReadOnly`, don't fully cover all types of AWS Step Functions permissions. This section describes these different types of permissions and provides some example configurations.

Step Functions has four categories of permissions. Depending on what access you want to provide to a user, you can control access by using permissions in these categories.

[Service-Level Permissions](#)

Apply to components of the API that do **not** act on a specific resource.

[State Machine-Level Permissions](#)

Apply to all API components that act on a specific state machine.

[Execution-Level Permissions](#)

Apply to all API components that act on a specific execution.

[Activity-Level Permissions](#)

Apply to all API components that act on a specific activity or on a particular instance of an activity.

Service-Level Permissions

This permission level applies to all API actions that do **not** act on a specific resource. These include [CreateStateMachine](#), [CreateActivity](#), [ListStateMachines](#), [ListActivities](#), and [ValidationStateMachineDefinition](#).

```
{
  "Version": "2012-10-17",
```

```

"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "states:ListStateMachines",
      "states:ListActivities",
      "states:CreateStateMachine",
      "states:CreateActivity",
      "states:ValidationStateMachineDefinition",
    ],
    "Resource": [
      "arn:aws:states:*:*:*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "iam:PassRole"
    ],
    "Resource": [
      "arn:aws:iam:::role/my-execution-role"
    ]
  }
]
}

```

State Machine-Level Permissions

This permission level applies to all API actions that act on a specific state machine. These API operations require the Amazon Resource Name (ARN) of the state machine as part of the request, such as [DeleteStateMachine](#), [DescribeStateMachine](#), [StartExecution](#), and [ListExecutions](#).

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeStateMachine",
        "states:StartExecution",
        "states>DeleteStateMachine",
        "states:ListExecutions",
      ]
    }
  ]
}

```



```

        "states:UpdateStateMachine",
        "states:TestState",
        "states:RevealSecrets"
    ],
    "Resource": [
        "arn:aws:states:*:*:stateMachine:StateMachinePrefix*"
    ]
}
]
}

```

Execution-Level Permissions

This permission level applies to all the API actions that act on a specific execution. These API operations require the ARN of the execution as part of the request, such as [DescribeExecution](#), [GetExecutionHistory](#), and [StopExecution](#).

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeExecution",
        "states:DescribeStateMachineForExecution",
        "states:GetExecutionHistory",
        "states:StopExecution"
      ],
      "Resource": [
        "arn:aws:states:*:*:execution:*:ExecutionPrefix*"
      ]
    }
  ]
}

```

Activity-Level Permissions

This permission level applies to all the API actions that act on a specific activity or on a particular instance of it. These API operations require the ARN of the activity or the token of the instance as part of the request, such as [DeleteActivity](#), [DescribeActivity](#), [GetActivityTask](#), and [SendTaskHeartbeat](#).

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeActivity",
        "states>DeleteActivity",
        "states:GetActivityTask",
        "states:SendTaskHeartbeat"
      ],
      "Resource": [
        "arn:aws:states:*:*:activity:ActivityPrefix*"
      ]
    }
  ]
}
```

Accessing resources in other AWS accounts in your workflows

Step Functions provides cross-account access to resources configured in different AWS accounts in your workflows. Using Step Functions service integrations, you can invoke any cross-account AWS resource even if that AWS service doesn't support resource-based policies or cross-account calls.

For example, assume you own two AWS accounts, called Development and Testing, in the same AWS Region. Using cross-account access, your workflow in the Development account can access resources, such as Amazon S3 buckets, Amazon DynamoDB tables, and Lambda functions that are available in the Testing account.

Important

IAM roles and resource-based policies delegate access across accounts only within a single partition. For example, assume that you have an account in US West (N. California) in the standard `aws` partition. You also have an account in China (Beijing) in the `aws-cn` partition. You can't use an Amazon S3 resource-based policy in your account in China (Beijing) to allow access for users in your standard `aws` account.

For more information about cross-account access, see [Cross-account policy evaluation logic](#) in the *IAM User Guide*.

Although each AWS account maintains complete control over its own resources, with Step Functions, you can reorganize, swap, add, or remove steps in your workflows without the need to customize any code. You can do this even as the processes change or applications evolve.

You can also invoke executions of nested state machines so they're available across different accounts. Doing so efficiently separates and isolates your workflows. When you use the [.sync](#) service integration pattern in your workflows that access another Step Functions workflow in a different account, Step Functions uses polling that consumes your assigned quota. For more information, see [Run a Job \(.sync\)](#).

Note

Currently, cross-Region AWS SDK integration and cross-Region AWS resource access aren't available in Step Functions.

Contents

- [Key concepts in this topic](#)
- [Invoking cross-account resources](#)
- [Tutorial: Accessing cross-account AWS resources](#)
- [Cross-account access for .sync integration pattern](#)

Key concepts in this topic

[Execution role](#)

An IAM role that Step Functions uses to run code and access AWS resources, such as the AWS Lambda function's Invoke action.

[Service integration](#)

The AWS SDK integration API actions that can be called from within a Task state in your workflows.

source account

An AWS account that owns the state machine and has started its execution.

target account

An AWS account to which you make cross-account calls.

target role

An IAM role in the target account that the state machine assumes for making calls to resources that the target account owns.

[Run a Job \(.sync\)](#)

A service integration pattern used to call services, such as AWS Batch. It also makes a Step Functions state machine wait for a job to complete before progressing to the next state. To indicate that Step Functions should wait, append the `.sync` suffix in the `Resource` field in your Task state definition.

Invoking cross-account resources

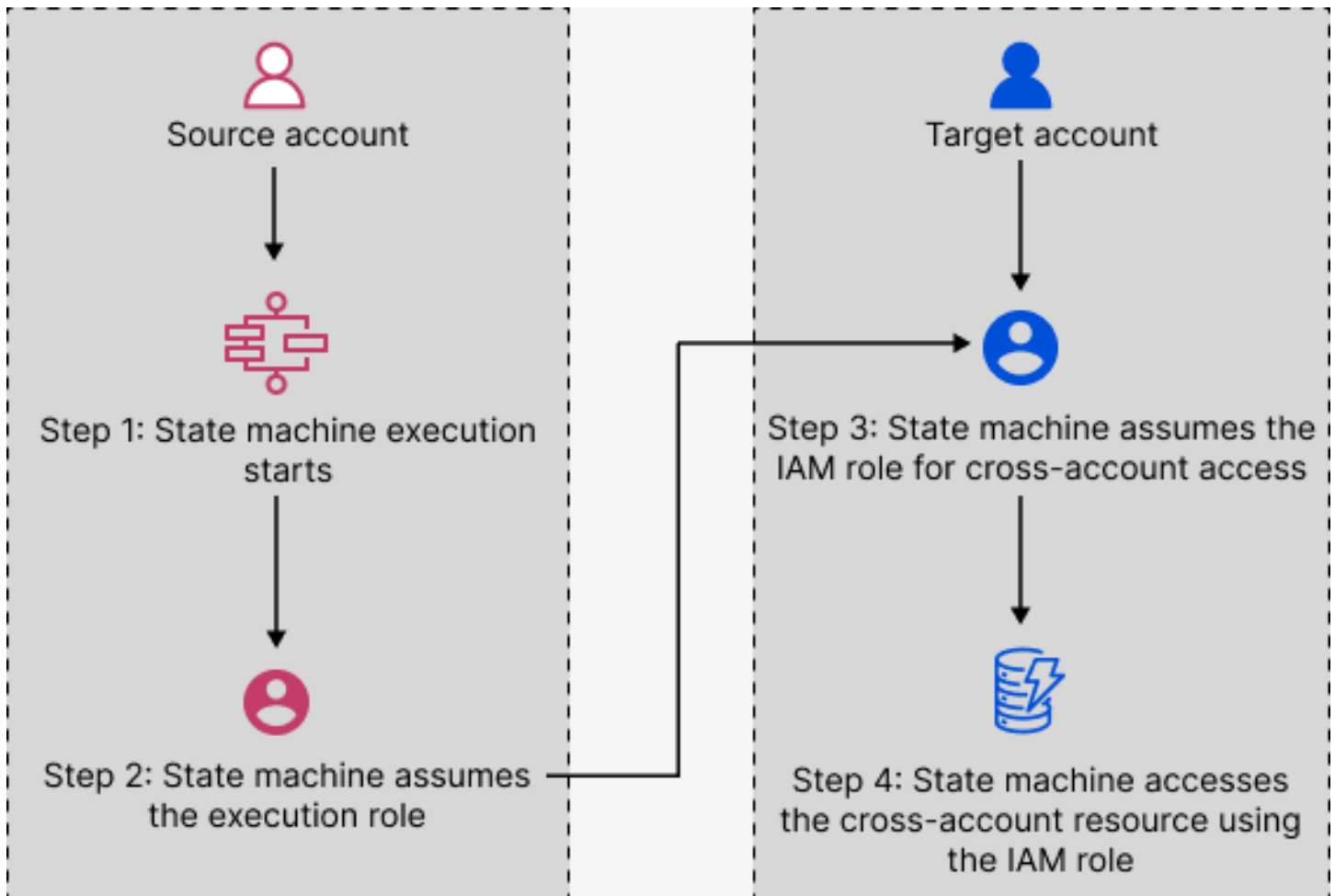
To invoke a cross-account resource in your workflows, do the following:

1. Create an IAM role in the target account that contains the resource. This role grants the source account, containing the state machine, permissions to access the target account's resources.
2. In the Task state's definition, specify the target IAM role to be assumed by the state machine before invoking the cross-account resource.
3. Modify the trust policy in the target IAM role to allow the source account to assume this role temporarily. The trust policy must include the Amazon Resource Name (ARN) of the state machine defined in the source account. Also, define the appropriate permissions in the target IAM role to call the AWS resource.
4. Update the source account's execution role to include the required permission for assuming the target IAM role.

For an example, see [Tutorial: Accessing cross-account AWS resources](#).

Note

You can configure your state machine to assume an IAM role for accessing resources from multiple AWS accounts. However, a state machine can assume only one IAM role at a given time.



Tutorial: Accessing cross-account AWS resources

With the cross-account access support in Step Functions, you can share resources configured in different AWS accounts. In this tutorial, we walk you through the process of accessing a cross-account Lambda function defined in an account called **Production**. This function is invoked from a state machine in an account called **Development**. In this tutorial, the **Development** account is referred to as the *source account* and the **Production** account is the *target account* containing the target IAM role.

To start, in your Task state's definition, you specify the target IAM role the state machine must assume before invoking the cross-account Lambda function. Then, modify the trust policy in the target IAM role to allow the source account to assume the target role temporarily. Also, to call the AWS resource, define the appropriate permissions in the target IAM role. Finally, update the source account's execution role to specify the required permission to assume the target role.

You can configure your state machine to assume an IAM role for accessing resources from multiple AWS accounts. However, a state machine can assume only one IAM role at a given time based on the Task state's definition.

Note

Currently, cross-Region AWS SDK integration and cross-Region AWS resource access aren't available in Step Functions.

Contents

- [Prerequisites](#)
- [Step 1: Update the Task state definition to specify the target role](#)
- [Step 2: Update the target role's trust policy](#)
- [Step 3: Add the required permission in the target role](#)
- [Step 4: Add permission in execution role to assume the target role](#)

Prerequisites

- This tutorial uses the example of a Lambda function for demonstrating how to set up cross-account access. You can use any other AWS resource, but make sure you've configured the resource in a different account.

Important

IAM roles and resource-based policies delegate access across accounts only within a single partition. For example, assume that you have an account in US West (N. California) in the standard `aws` partition. You also have an account in China (Beijing) in the `aws-cn` partition. You can't use an Amazon S3 resource-based policy in your account in China (Beijing) to allow access for users in your standard `aws` account.

- Make a note of the cross-account resource's Amazon Resource Name (ARN) in a text file. Later in this tutorial, you'll provide this ARN in your state machine's Task state definition. The following is an example of a Lambda function ARN:

```
arn:aws:lambda:us-east-2:123456789012:function:functionName
```

- Make sure you've created the target IAM role that the state machine needs to assume.

Step 1: Update the Task state definition to specify the target role

In the Task state of your workflow, add a `Credentials` field containing the identity the state machine must assume before invoking the cross-account Lambda function.

The following procedure demonstrates how to access a cross-account Lambda function called `Echo`. You can call any AWS resource by following these steps.

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. On the **Choose authoring method** page, choose **Design your workflow visually** and keep all the default selections.
3. To open Workflow Studio, choose **Next**.
4. On the **Actions** tab, drag and drop a Task state on the canvas. This invokes the cross-account Lambda function that's using this Task state.
5. On the **Configuration** tab, do the following:
 - a. Rename the state to **Cross-account call**.
 - b. For **Function name**, choose **Enter function name**, and then enter the Lambda function ARN in the box. For example, `arn:aws:lambda:us-east-2:111122223333:function:Echo`.
 - c. For **Provide IAM role ARN**, specify the target IAM role ARN. For example, `arn:aws:iam::111122223333:role/LambdaRole`.

Tip

Alternatively, you can also specify a [reference path](#) to an existing key-value pair in the state's JSON input that contains the IAM role ARN. To do this, choose **Get IAM role ARN at runtime from state input**. For an example of specifying a value by using a reference path, see [Specifying JSONPath as IAM role ARN](#).

6. Choose **Next**.
7. On the **Review generated code** page, choose **Next**.
8. On the **Specify state machine settings** page, specify details for the new state machine, such as a name, permissions, and logging level.

9. Choose **Create state machine**.
10. Make a note of the state machine's IAM role ARN and the state machine ARN in a text file. You'll need to provide these ARNs in the target account's trust policy.

Your Task state definition should now look similar to the following definition.

```
{
  "StartAt": "Cross-account call",
  "States": {
    "Cross-account call": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Credentials": {
        "RoleArn": "arn:aws:iam::111122223333:role/LambdaRole"
      },
      "Parameters": {
        "FunctionName": "arn:aws:lambda:us-east-2:111122223333:function:Echo",
      },
      "End": true
    }
  }
}
```

Step 2: Update the target role's trust policy

The IAM role must exist in the target account and you must modify its trust policy to allow the source account to assume this role temporarily. Additionally, you can control who can assume the target IAM role.

After you create the trust relationship, a user from the source account can use the AWS Security Token Service (AWS STS) [AssumeRole](#) API operation. This operation provides temporary security credentials that enable access to AWS resources in a target account.

1. Open the IAM console at <https://console.aws.amazon.com/iam/>.
2. On the navigation pane of the console, choose **Roles** and then use the Search box to search for the target IAM role. For example, *LambdaRole*.
3. Choose the **Trust relationships** tab.
4. Choose **Edit trust policy** and paste the following trust policy. Make sure to replace the AWS account number and IAM role ARN. The `sts:ExternalId` field further controls who can

assume the role. The state machine's name must include only characters that the AWS Security Token Service AssumeRole API supports. For more information, see [AssumeRole](#) in the *AWS Security Token Service API Reference*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:role/ExecutionRole" // The source
        account's state machine execution role ARN
      },
      "Condition": { // Control which account and state machine can assume the
        target IAM role

        "StringEquals": {
          "sts:ExternalId": "arn:aws:states:us-
east-1:123456789012:stateMachine:testCrossAccount" // ARN of the state machine
          that will assume the role.
        }
      }
    }
  ]
}
```

5. Keep this window open and proceed to the next step for further actions.

Step 3: Add the required permission in the target role

Permissions in the IAM policies determine whether a specific request is allowed or denied. The target IAM role must have the correct permission to invoke the Lambda function.

1. Choose the **Permissions** tab.
2. Choose **Add permissions** and then choose **Create inline policy**.
3. Choose the **JSON** tab and replace the existing content with the following permission. Make sure to replace your Lambda function ARN.

```
{
  "Version": "2012-10-17",
```

```
"Statement": [  
  {  
    "Effect": "Allow",  
    "Action": "lambda:InvokeFunction",  
    "Resource": "arn:aws:lambda:us-east-2:111122223333:function:Echo" // The  
cross-account AWS resource being accessed  
  }  
]
```

4. Choose **Review policy**.
5. On the **Review policy** page, enter a name for the permission, and then choose **Create policy**.

Step 4: Add permission in execution role to assume the target role

Step Functions doesn't automatically generate the [AssumeRole](#) policy for all cross-account service integrations. You must add the required permission in the state machine's execution role to allow it to assume a target IAM role in one or more AWS accounts.

1. Open your state machine's execution role in the IAM console at <https://console.aws.amazon.com/iam/>. To do this:
 - a. Open the state machine that you created in [Step 1 in the source account](#).
 - b. On the **State machine detail** page, choose **IAM role ARN**.
2. On the **Permissions** tab, choose **Add permissions** and then choose **Create inline policy**.
3. Choose the **JSON** tab and replace the existing content with the following permission. Make sure to replace your Lambda function ARN.

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": "sts:AssumeRole",  
      "Resource": "arn:aws:iam::111122223333:role/LambdaRole" // The target role  
to be assumed  
    }  
  ]  
}
```

4. Choose **Review policy**.
5. On the **Review policy** page, enter a name for the permission, and then choose **Create policy**.

Cross-account access for `.sync` integration pattern

When you use the [.sync](#) service integration patterns in your workflows, Step Functions polls the invoked cross-account resource to confirm the task is complete. This causes a slight delay between the actual task completion time and the time when Step Functions recognizes the task as complete. The target IAM role needs the required permissions for a `.sync` invocation to complete this polling loop. To do this, the target IAM role must have a trust policy that allows the source account to assume it. Additionally, the target IAM role needs the required permissions to complete the polling loop.

Note

For nested Express Workflows, `arn:aws:states:::states:startExecution.sync` isn't currently supported. Use `arn:aws:states:::aws-sdk:sfn:startSyncExecution` instead.

Trust policy update for `.sync` calls

Update the trust policy of your target IAM role as shown in the following example. The `sts:ExternalId` field further controls who can assume the role. The state machine's name must include only characters that the AWS Security Token Service `AssumeRole` API supports. For more information, see [AssumeRole](#) in the *AWS Security Token Service API Reference*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Principal": {
        "AWS": "arn:aws:iam::sourceAccountID:role/InvokeRole",
      },
      "Condition": {
        "StringEquals": {
```

```

        "sts:ExternalId": "arn:aws:states:us-
east-2:sourceAccountID:stateMachine:stateMachineName"
    }
}
]
}

```

Permissions required for .sync calls

To grant the permissions required for your state machine, update the required permissions for the target IAM role. For more information, see [the section called “IAM Policies for integrated services”](#). The Amazon EventBridge permissions from the example policies aren't required. For example, to start a state machine, add the following permissions.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": [
        "arn:aws:states:region:accountID:stateMachine:stateMachineName"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeExecution",
        "states:StopExecution"
      ],
      "Resource": [
        "arn:aws:states:region:accountID:execution:stateMachineName:*"
      ]
    }
  ]
}

```

Amazon VPC Endpoints for Step Functions

If you use Amazon Virtual Private Cloud (Amazon VPC) to host your AWS resources, you can establish a connection between your Amazon VPC and AWS Step Functions workflows. You can use this connection with your Step Functions workflows without crossing the public internet. Amazon VPC endpoints are supported by Standard Workflows, Express Workflows, and Synchronous Express Workflows.

Amazon VPC lets you launch AWS resources in a custom virtual network. You can use a VPC to control your network settings, such as the IP address range, subnets, route tables, and network gateways. For more information about VPCs, see the [Amazon VPC User Guide](#).

To connect your Amazon VPC to Step Functions, you must first define an *interface VPC endpoint*, which lets you connect your VPC to other AWS services. The endpoint provides reliable, scalable connectivity, without requiring an internet gateway, network address translation (NAT) instance, or VPN connection. For more information, see [Interface VPC Endpoints \(AWS PrivateLink\)](#) in the *Amazon VPC User Guide*.

Creating the Endpoint

You can create an AWS Step Functions endpoint in your VPC using the AWS Management Console, the AWS Command Line Interface (AWS CLI), an AWS SDK, the AWS Step Functions API, or AWS CloudFormation.

For information about creating and configuring an endpoint using the Amazon VPC console or the AWS CLI, see [Creating an Interface Endpoint](#) in the *Amazon VPC User Guide*.

Note

When you create an endpoint, specify Step Functions as the service that you want your VPC to connect to. In the Amazon VPC console, service names vary based on the AWS Region. For example, if you choose US East (N. Virginia), the service name for Standard Workflows and Express Workflows is **com.amazonaws.us-east-1.states**, and the service name for Synchronous Express Workflows is **com.amazonaws.us-east-1.sync-states**.

Note

It's possible to use VPC Endpoints without overriding the endpoint in the SDK through [Private DNS](#). However, if you want to override the endpoint in the SDK for Synchronous Express Workflows, you need to set `DisableHostPrefixInjection` configuration to `true`. Example (Java SDK V2):

```
SfnClient.builder()
    .endpointOverride(URI.create("https://vpce-{vpceId}.sync-states.us-
east-1.vpce.amazonaws.com"))
    .overrideConfiguration(ClientOverrideConfiguration.builder()

    .advancedOptions(ImmutableMap.of(SdkAdvancedClientOption.DISABLE_HOST_PREFIX_INJECTION,
true))
    .build())
    .build();
```

For information about creating and configuring an endpoint using AWS CloudFormation, see the [AWS::EC2::VPCEndpoint](#) resource in the *AWS CloudFormation User Guide*.

Amazon VPC Endpoint Policies

To control connectivity access to Step Functions you can attach an AWS Identity and Access Management (IAM) endpoint policy while creating an Amazon VPC endpoint. You can create complex IAM rules by attaching multiple endpoint policies. For more information, see:

- [Amazon Virtual Private Cloud Endpoint Policies for Step Functions](#)
- [Creating Granular IAM Permissions for Non-Admin Users](#)
- [Controlling Access to Services with VPC Endpoints](#)

Amazon Virtual Private Cloud Endpoint Policies for Step Functions

You can create an Amazon VPC endpoint policy for Step Functions in which you specify the following:

- The principal that can perform actions.
- The actions that can be performed.

- The resources on which the actions can be performed.

The following example shows an Amazon VPC endpoint policy that allows one user to create state machines, and denies all other users permission to delete state machines. The example policy also grants all users execution permission.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "*Execution",
      "Resource": "*",
      "Effect": "Allow",
      "Principal": "*"
    },
    {
      "Action": "states:CreateStateMachine",
      "Resource": "*",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:user/MyUser"
      }
    },
    {
      "Action": "states>DeleteStateMachine",
      "Resource": "*",
      "Effect": "Deny",
      "Principal": "*"
    }
  ]
}
```

For more information about creating endpoint policies, see the following:

- [Creating Granular IAM Permissions for Non-Admin Users](#)
- [Controlling Access to Services with VPC Endpoints](#)

IAM Policies for integrated services

When you create a state machine in the AWS Step Functions console, Step Functions produces an AWS Identity and Access Management (IAM) policy based on the resources used in your state machine definition as follows:

- If your state machine uses one of the Optimized integrations, Step Functions will create a policy with the necessary permissions and roles for your state machine. (Exception: MediaConvert integration requires you to manually set up permissions – see [IAM policies for AWS Elemental MediaConvert](#).)
- If your state machine uses one of the AWS SDK integrations, an IAM role with partial permissions will be created. Afterwards, you can use the IAM console to add any missing role policies.

The following examples show how Step Functions generates an IAM policy based on your state machine definition. Items in the example code such as `[[resourceName]]` are replaced with the static resources listed in your state machine definition. If you have multiple static resources, there will be an entry for each in the IAM role.

Dynamic vs. Static Resources

Static resources are defined directly in the task state of your state machine. When you include the information about the resources you want to call directly in your task states, Step Functions creates an IAM role for only those resources.

Dynamic resources are those that are passed in to your state input, and accessed using a Path (see [Paths](#)). If you are passing dynamic resources to your task, Step Functions will create a more privileged policy that specifies: "Resource": "*".

Additional permissions for tasks using the Run a Job pattern

For tasks that use the [Run a Job](#) pattern (those ending in `.sync`), additional permissions are needed to monitor and receive a response from the API actions of connected services. The related policies include more permissions than for tasks that use the Request Response or Wait for Callback patterns. See [Service integration patterns](#) for information about synchronous tasks.

Note

You need to provide additional permissions for service integrations that support the Run a Job (.sync) pattern.

Step Functions uses two methods to monitor a job's status when a job is run on a connected service, polling and events.

Polling requires permission for Describe or Get API actions, such as `ecs:DescribeTasks` or `glue:GetJobRun`. If these permissions are missing from your role, then Step Functions may be unable to determine the status of your job. This is because some Run a Job (.sync) service integrations do not support EventBridge events, and some services only send events on a best-effort basis.

Events sent from AWS services to Amazon EventBridge are directed to Step Functions using a managed rule, and require permissions for `events:PutTargets`, `events:PutRule`, and `events:DescribeRule`. If these permissions are missing from your role, there may be a delay before Step Functions becomes aware of the completion of your job. For more information about EventBridge events, see [Events from AWS services](#).

Note

For Run a Job (.sync) tasks that support both polling and events, your task may still complete properly using events. This can occur even if your role lacks the required permissions for polling. In this case, you may not immediately notice that the polling permissions are incorrect or missing. In the rare instance that the event fails to be delivered to or processed by Step Functions, your execution could become stuck. To verify that your polling permissions are configured correctly, you can run an execution in an environment without EventBridge events in the following ways:

- Delete the managed rule from EventBridge, which is responsible for forwarding events to Step Functions. This managed rule is shared by all state machines in your account, so you should perform this action only in a test or development account to avoid any unintentional impact on other state machines. You can identify the specific managed rule to delete by inspecting the `Resource` field used for `events:PutRule` in the policy template for the target service. The managed rule will be recreated the next time you

create or update a state machine that uses that service integration. For more information on deleting EventBridge rules, see [Disabling or deleting a rule](#).

- Test with Step Functions Local, which does not support the use of events to complete Run a Job (.sync) tasks. To use Step Functions Local, assume the IAM role used by your state machine. You may need to edit the Trust Relationship. Set the `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN` environment variables to the assumed role's values, then launch Step Functions Local using `java -jar StepFunctionsLocal.jar`. Last, use the AWS CLI with the `--endpoint-url` parameter to create a state machine, start an execution, and get the execution history. For more information, see [Testing state machines locally](#).

If a task that uses the Run a Job (.sync) pattern is stopped, Step Functions will make a best-effort attempt to cancel the task. This requires permission to `Cancel`, `Stop`, `Terminate`, or `Delete` API actions, such as `batch:TerminateJob` or `eks>DeleteCluster`. If these permissions are missing from your role, Step Functions will be unable to cancel your task and you may accrue additional charges while it continues to run. For more information on stopping tasks, see [Run a Job](#).

Policy templates used to create IAM roles

The following topics include the policy templates used when you choose to have Step Functions create a new role for you.

Note

Review these templates to understand how Step Functions creates your IAM policies, and as an example of how to manually create IAM policies for Step Functions when working with other AWS services. For more information about Step Functions service integrations, see [Using AWS Step Functions with other services](#).

Topics

- [IAM policies for Amazon API Gateway](#)
- [IAM policies for Amazon Athena](#)
- [IAM policies for AWS Batch](#)
- [IAM policies for Amazon Bedrock](#)

- [IAM policies for AWS CodeBuild](#)
- [IAM policies for Amazon DynamoDB](#)
- [IAM policies for Amazon ECS/AWS Fargate](#)
- [IAM policies for Amazon EKS](#)
- [IAM policies for Amazon EMR](#)
- [IAM policies for Amazon EMR on EKS](#)
- [IAM policies for Amazon EMR Serverless](#)
- [IAM policies for Amazon EventBridge](#)
- [IAM policies for AWS Lambda](#)
- [IAM policies for AWS Elemental MediaConvert](#)
- [IAM policies for AWS Glue](#)
- [IAM policies for AWS Glue DataBrew](#)
- [IAM policies for Amazon SageMaker](#)
- [IAM policies for Amazon SNS](#)
- [IAM policies for Amazon SQS](#)
- [IAM policies for AWS Step Functions](#)
- [IAM policies for AWS X-Ray](#)
- [Activities or No Tasks](#)

IAM policies for Amazon API Gateway

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

Resources:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
```

```

        "execute-api:Invoke"
    ],
    "Resource": [
        "arn:[[region]]:[[accountId]]:*"
    ]
}
]
}

```

The following code example shows a resource policy for calling API Gateway.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "states.amazonaws.com"
      },
      "Action": "execute-api:Invoke",
      "Resource": "arn:aws:execute-api:<region>:<account-id>:<api-id>/<stage-name>/<HTTP-VERB>/<resource-path-specifier>",
      "Condition": {
        "StringEquals": {
          "aws:SourceArn": [
            "<SourceStateMachineArn>"
          ]
        }
      }
    }
  ]
}

```

IAM policies for Amazon Athena

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

StartQueryExecution

Static resources

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:startQueryExecution",
        "athena:stopQueryExecution",
        "athena:getQueryExecution",
        "athena:getDataCatalog"
      ],
      "Resource": [
        "arn:aws:athena:{{region}}:{{accountId}}:workgroup/[[workGroup]]",
        "arn:aws:athena:{{region}}:{{accountId}}:datacatalog/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:ListBucketMultipartUploads",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload",
        "s3:CreateBucket",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "glue:CreateDatabase",
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:UpdateDatabase",
        "glue>DeleteDatabase",
        "glue:CreateTable",
        "glue:UpdateTable",

```

```

        "glue:GetTable",
        "glue:GetTables",
        "glue>DeleteTable",
        "glue:BatchDeleteTable",
        "glue:BatchCreatePartition",
        "glue:CreatePartition",
        "glue:UpdatePartition",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:BatchGetPartition",
        "glue>DeletePartition",
        "glue:BatchDeletePartition"
    ],
    "Resource": [
        "arn:aws:glue:{{region}}:{{accountId}}:catalog",
        "arn:aws:glue:{{region}}:{{accountId}}:database/*",
        "arn:aws:glue:{{region}}:{{accountId}}:table/*",
        "arn:aws:glue:{{region}}:{{accountId}}:userDefinedFunction/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "lakeformation:GetDataAccess"
    ],
    "Resource": [
        "*"
    ]
}
]
}

```

Request Response

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:startQueryExecution",
                "athena:getDataCatalog"
            ],
        }
    ],
}

```

```

    "Resource": [
      "arn:aws:athena:{{region}}:{{accountId}}:workgroup/[workGroup]",
      "arn:aws:athena:{{region}}:{{accountId}}:datacatalog/*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "s3:GetBucketLocation",
      "s3:GetObject",
      "s3:ListBucket",
      "s3:ListBucketMultipartUploads",
      "s3:ListMultipartUploadParts",
      "s3:AbortMultipartUpload",
      "s3:CreateBucket",
      "s3:PutObject"
    ],
    "Resource": [
      "arn:aws:s3::*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "glue:CreateDatabase",
      "glue:GetDatabase",
      "glue:GetDatabases",
      "glue:UpdateDatabase",
      "glue>DeleteDatabase",
      "glue:CreateTable",
      "glue:UpdateTable",
      "glue:GetTable",
      "glue:GetTables",
      "glue>DeleteTable",
      "glue:BatchDeleteTable",
      "glue:BatchCreatePartition",
      "glue:CreatePartition",
      "glue:UpdatePartition",
      "glue:GetPartition",
      "glue:GetPartitions",
      "glue:BatchGetPartition",
      "glue>DeletePartition",
      "glue:BatchDeletePartition"
    ],
  },

```

```

    "Resource": [
      "arn:aws:glue:{{region}}:{{accountId}}:catalog",
      "arn:aws:glue:{{region}}:{{accountId}}:database/*",
      "arn:aws:glue:{{region}}:{{accountId}}:table/*",
      "arn:aws:glue:{{region}}:{{accountId}}:userDefinedFunction/*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "lakeformation:GetDataAccess"
    ],
    "Resource": [
      "*"
    ]
  }
]
}

```

Dynamic resources

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:startQueryExecution",
        "athena:stopQueryExecution",
        "athena:getQueryExecution",
        "athena:getDataCatalog"
      ],
      "Resource": [
        "arn:aws:athena:{{region}}:{{accountId}}:workgroup/*",
        "arn:aws:athena:{{region}}:{{accountId}}:datacatalog/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [

```



```

        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:ListBucketMultipartUploads",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload",
        "s3:CreateBucket",
        "s3:PutObject"
    ],
    "Resource": [
        "arn:aws:s3:::*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "glue:CreateDatabase",
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:UpdateDatabase",
        "glue>DeleteDatabase",
        "glue:CreateTable",
        "glue:UpdateTable",
        "glue:GetTable",
        "glue:GetTables",
        "glue>DeleteTable",
        "glue:BatchDeleteTable",
        "glue:BatchCreatePartition",
        "glue:CreatePartition",
        "glue:UpdatePartition",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:BatchGetPartition",
        "glue>DeletePartition",
        "glue:BatchDeletePartition"
    ],
    "Resource": [
        "arn:aws:glue:{{region}}:{{accountId}}:catalog",
        "arn:aws:glue:{{region}}:{{accountId}}:database/*",
        "arn:aws:glue:{{region}}:{{accountId}}:table/*",
        "arn:aws:glue:{{region}}:{{accountId}}:userDefinedFunction/*"
    ]
},
{

```

```

    "Effect": "Allow",
    "Action": [
        "lakeformation:GetDataAccess"
    ],
    "Resource": [
        "*"
    ]
}
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:startQueryExecution",
        "athena:getDataCatalog"
      ],
      "Resource": [
        "arn:aws:athena:{{region}}:{{accountId}}:workgroup/*",
        "arn:aws:athena:{{region}}:{{accountId}}:datacatalog/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:ListBucketMultipartUploads",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload",
        "s3:CreateBucket",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::*"
      ]
    }
  ],
}

```

```

{
  "Effect": "Allow",
  "Action": [
    "glue:CreateDatabase",
    "glue:GetDatabase",
    "glue:GetDatabases",
    "glue:UpdateDatabase",
    "glue>DeleteDatabase",
    "glue:CreateTable",
    "glue:UpdateTable",
    "glue:GetTable",
    "glue:GetTables",
    "glue>DeleteTable",
    "glue:BatchDeleteTable",
    "glue:BatchCreatePartition",
    "glue:CreatePartition",
    "glue:UpdatePartition",
    "glue:GetPartition",
    "glue:GetPartitions",
    "glue:BatchGetPartition",
    "glue>DeletePartition",
    "glue:BatchDeletePartition"
  ],
  "Resource": [
    "arn:aws:glue:{{region}}:{{accountId}}:catalog",
    "arn:aws:glue:{{region}}:{{accountId}}:database/*",
    "arn:aws:glue:{{region}}:{{accountId}}:table/*",
    "arn:aws:glue:{{region}}:{{accountId}}:userDefinedFunction/*"
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "lakeformation:GetDataAccess"
  ],
  "Resource": [
    "*"
  ]
}
]
}

```

StopQueryExecution

Resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:stopQueryExecution"
      ],
      "Resource": [
        "arn:aws:athena:{{region}}:{{accountId}}:workgroup/*"
      ]
    }
  ]
}
```

GetQueryExecution

Resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:getQueryExecution"
      ],
      "Resource": [
        "arn:aws:athena:{{region}}:{{accountId}}:workgroup/*"
      ]
    }
  ]
}
```

GetQueryResults

Resources

```
{
```

```

"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "athena:getQueryResults"
    ],
    "Resource": [
      "arn:aws:athena:{{region}}:{{accountId}}:workgroup/*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "s3:GetObject"
    ],
    "Resource": [
      "arn:aws:s3:::*"
    ]
  }
]
}

```

IAM policies for AWS Batch

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

Because AWS Batch provides partial support for resource-level access control, you must use `"Resource": "*"` .

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "batch:SubmitJob",
        "batch:DescribeJobs",
        "batch:TerminateJob"
      ]
    }
  ]
}

```

```
    ],
    "Resource": "*"
  },
  {
    "Effect": "Allow",
    "Action": [
      "events:PutTargets",
      "events:PutRule",
      "events:DescribeRule"
    ],
    "Resource": [
      "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForBatchJobsRule"
    ]
  }
]
```

Request Response

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "batch:SubmitJob"
      ],
      "Resource": "*"
    }
  ]
}
```

IAM policies for Amazon Bedrock

When you create a state machine using the console, Step Functions automatically creates an execution role for your state machine with the least privileges required. These automatically generated IAM roles are valid for the AWS Region in which you create the state machine.

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

We recommend that when you create IAM policies, do not include wildcards in the policies. As a security best practice, you should scope your policies down as much as possible. You should use dynamic policies only when certain input parameters are not known during runtime.

In this topic

- [IAM policy examples for Amazon Bedrock integration with Step Functions](#)

IAM policy examples for Amazon Bedrock integration with Step Functions

The following section describes the IAM permissions you need based on the Amazon Bedrock API that you use for a specific foundation or provisioned model. This section also contains examples of policies that grant full access.

Remember to replace the *italicized* text with your resource-specific information.

- [IAM policy example to access a specific foundation model using InvokeModel](#)
- [IAM policy example to access a specific provisioned model using InvokeModel](#)
- [Full access IAM policy example to use InvokeModel](#)
- [IAM policy example to access a specific foundation model as a base model](#)
- [IAM policy example to access a specific custom model as a base model](#)
- [Full access IAM policy example to use CreateModelCustomizationJob.sync](#)
- [IAM policy example to access a specific foundation model using CreateModelCustomizationJob.sync](#)
- [IAM policy example to access a custom model using CreateModelCustomizationJob.sync](#)
- [Full access IAM policy example to use CreateModelCustomizationJob.sync](#)

IAM policy example to access a specific foundation model using InvokeModel

The following is an IAM policy example for a state machine that accesses a specific foundation model named `amazon.titan-text-express-v1` using the [InvokeModel](#) API action.

```
{
  "Version": "2012-10-17",
```

```

    "Statement": [
      {
        "Effect": "Allow",
        "Sid": "InvokeModel1",
        "Action": [
          "bedrock:InvokeModel"
        ],
        "Resource": [
          "arn:aws:bedrock:us-east-2::foundation-model/amazon.titan-text-express-
v1"
        ]
      }
    ]
  }

```

IAM policy example to access a specific provisioned model using InvokeModel

The following is an IAM policy example for a state machine that accesses a specific provisioned model named `c2oi931u1ksx` using the [InvokeModel](#) API action.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Sid": "InvokeModel1",
      "Action": [
        "bedrock:InvokeModel"
      ],
      "Resource": [
        "arn:aws:bedrock:us-east-2:123456789012:provisioned-model/c2oi931u1ksx"
      ]
    }
  ]
}

```

Full access IAM policy example to use InvokeModel

The following is an IAM policy example for a state machine that provides full access when you use the [InvokeModel](#) API action.

```

{
  "Version": "2012-10-17",

```



```

    "Statement": [
      {
        "Effect": "Allow",
        "Sid": "InvokeModel1",
        "Action": [
          "bedrock:InvokeModel"
        ],
        "Resource": [
          "arn:aws:bedrock:us-east-2::foundation-model/*",
          "arn:aws:bedrock:us-east-2:123456789012:provisioned-model/*"
        ]
      }
    ]
  }
}

```

IAM policy example to access a specific foundation model as a base model

The following is an IAM policy example for a state machine to access a specific foundation model named `amazon.titan-text-express-v1` as a base model using the [CreateModelCustomizationJob](#) API action.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Sid": "CreateModelCustomizationJob1",
      "Action": [
        "bedrock:CreateModelCustomizationJob"
      ],
      "Resource": [
        "arn:aws:bedrock:us-east-2::foundation-model/amazon.titan-text-express-  
v1",
        "arn:aws:bedrock:us-east-2:123456789012:custom-model/*",
        "arn:aws:bedrock:us-east-2:123456789012:model-customization-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Sid": "CreateModelCustomizationJob2",
      "Action": [
        "iam:PassRole"
      ],
    }
  ]
}

```

```

        "Resource": [
            "arn:aws:iam::123456789012:role/myRole"
        ]
    }
]
}

```

IAM policy example to access a specific custom model as a base model

The following is an IAM policy example for a state machine to access a specific custom model as a base model using the [CreateModelCustomizationJob](#) API action.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Sid": "CreateModelCustomizationJob1",
      "Action": [
        "bedrock:CreateModelCustomizationJob"
      ],
      "Resource": [
        "arn:aws:bedrock:us-east-2:123456789012:custom-model/*",
        "arn:aws:bedrock:us-east-2:123456789012:model-customization-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Sid": "CreateModelCustomizationJob2",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/[roleName]"
      ]
    }
  ]
}

```

Full access IAM policy example to use CreateModelCustomizationJob.sync

The following is an IAM policy example for a state machine that provides full access when you use the [CreateModelCustomizationJob](#) API action.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Sid": "CreateModelCustomizationJob1",
      "Action": [
        "bedrock:CreateModelCustomizationJob"
      ],
      "Resource": [
        "arn:aws:bedrock:us-east-2::foundation-model/*",
        "arn:aws:bedrock:us-east-2:123456789012:custom-model/*",
        "arn:aws:bedrock:us-east-2:123456789012:model-customization-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Sid": "CreateModelCustomizationJob2",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/myRole"
      ]
    }
  ]
}

```

IAM policy example to access a specific foundation model using `CreateModelCustomizationJob.sync`

The following is an IAM policy example for a state machine to access a specific foundation model named `amazon.titan-text-express-v1` using the [CreateModelCustomizationJob.sync](#) API action.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Sid": "CreateModelCustomizationJob1",
      "Action": [

```

```

        "bedrock:CreateModelCustomizationJob"
    ],
    "Resource": [
        "arn:aws:bedrock:us-east-2::foundation-model/amazon.titan-text-express-
v1",
        "arn:aws:bedrock:us-east-2:123456789012:custom-model/*",
        "arn:aws:bedrock:us-east-2:123456789012:model-customization-job/*"
    ]
},
{
    "Effect": "Allow",
    "Sid": "CreateModelCustomizationJob2",
    "Action": [
        "bedrock:GetModelCustomizationJob",
        "bedrock:StopModelCustomizationJob"
    ],
    "Resource": [
        "arn:aws:bedrock:us-east-2:123456789012:model-customization-job/*"
    ]
},
{
    "Effect": "Allow",
    "Sid": "CreateModelCustomizationJob3",
    "Action": [
        "iam:PassRole"
    ],
    "Resource": [
        "arn:aws:iam::123456789012:role/myRole"
    ]
}
]
}

```

IAM policy example to access a custom model using `CreateModelCustomizationJob.sync`

The following is an IAM policy example for a state machine to access a custom model using the [CreateModelCustomizationJob.sync](#) API action.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",

```

```

    "Sid": "CreateModelCustomizationJob1",
    "Action": [
      "bedrock:CreateModelCustomizationJob"
    ],
    "Resource": [
      "arn:aws:bedrock:us-east-2:123456789012:custom-model/*",
      "arn:aws:bedrock:us-east-2:123456789012:model-customization-job/*"
    ]
  },
  {
    "Effect": "Allow",
    "Sid": "CreateModelCustomizationJob2",
    "Action": [
      "bedrock:GetModelCustomizationJob",
      "bedrock:StopModelCustomizationJob"
    ],
    "Resource": [
      "arn:aws:bedrock:us-east-2:123456789012:model-customization-job/*"
    ]
  },
  {
    "Effect": "Allow",
    "Sid": "CreateModelCustomizationJob3",
    "Action": [
      "iam:PassRole"
    ],
    "Resource": [
      "arn:aws:iam::123456789012:role/myRole"
    ]
  }
]
}

```

Full access IAM policy example to use `CreateModelCustomizationJob.sync`

The following is an IAM policy example for a state machine that provides full access when you use the [CreateModelCustomizationJob.sync](#) API action.

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",

```

```

    "Sid": "CreateModelCustomizationJob1",
    "Action": [
      "bedrock:CreateModelCustomizationJob"
    ],
    "Resource": [
      "arn:aws:bedrock:us-east-2::foundation-model/*",
      "arn:aws:bedrock:us-east-2:123456789012:custom-model/*",
      "arn:aws:bedrock:us-east-2:123456789012:model-customization-job/*"
    ]
  },
  {
    "Effect": "Allow",
    "Sid": "CreateModelCustomizationJob2",
    "Action": [
      "bedrock:GetModelCustomizationJob",
      "bedrock:StopModelCustomizationJob"
    ],
    "Resource": [
      "arn:aws:bedrock:us-east-2:123456789012:model-customization-job/*"
    ]
  },
  {
    "Effect": "Allow",
    "Sid": "CreateModelCustomizationJob3",
    "Action": [
      "iam:PassRole"
    ],
    "Resource": [
      "arn:aws:iam::123456789012:role/myRole"
    ]
  }
]
}

```

IAM policies for AWS CodeBuild

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

Resources:

```
{
```

```

"Version": "2012-10-17",
"Statement": [
  {
    "Action": [
      "sns:Publish"
    ],
    "Resource": [
      "arn:aws:sns:sa-east-1:123456789012:StepFunctionsSample-
CodeBuildExecution1111-2222-3333-wJalrXUtnFEMI-SNSTopic-bPxRfiCYEXAMPLEKEY"
    ],
    "Effect": "Allow"
  },
  {
    "Action": [
      "codebuild:StartBuild",
      "codebuild:StopBuild",
      "codebuild:BatchGetBuilds",
      "codebuild:BatchGetReports"
    ],
    "Resource": "*",
    "Effect": "Allow"
  },
  {
    "Action": [
      "events:PutTargets",
      "events:PutRule",
      "events:DescribeRule"
    ],
    "Resource": [
      "arn:aws:events:sa-east-1:123456789012:rule/
StepFunctionsGetEventForCodeBuildStartBuildRule"
    ],
    "Effect": "Allow"
  }
]
}

```

StartBuild

Static resources

Run a Job (.sync)

```
{
```

```

"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "codebuild:StartBuild",
      "codebuild:StopBuild",
      "codebuild:BatchGetBuilds"
    ],
    "Resource": [
      "arn:aws:codebuild:[[region]]:[[accountId]]:project/[[projectName]]"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "events:PutTargets",
      "events:PutRule",
      "events:DescribeRule"
    ],
    "Resource": [
      "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventForCodeBuildStartBuildRule"
    ]
  }
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuild"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:project/[[projectName]]"
      ]
    }
  ]
}

```



```
]
}
```

Dynamic resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuild",
        "codebuild:StopBuild",
        "codebuild:BatchGetBuilds"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:*:project/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventForCodeBuildStartBuildRule"
      ]
    }
  ]
}
```

Request Response

```
{
  "Version": "2012-10-17",
```

```

    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "codebuild:StartBuild"
        ],
        "Resource": [
          "arn:aws:codebuild:[[region]]:*:project/*"
        ]
      }
    ]
  }
}

```

StopBuild

Static resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StopBuild"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:project/[[projectName]]"
      ]
    }
  ]
}

```

Dynamic resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StopBuild"
      ],

```

```
    "Resource": [
      "arn:aws:codebuild:[[region]]:*:project/*"
    ]
  }
]
}
```

BatchDeleteBuilds

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:BatchDeleteBuilds"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:project/[[projectName]]"
      ]
    }
  ]
}
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:BatchDeleteBuilds"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:*:project/*"
      ]
    }
  ]
}
```

BatchGetReports

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:BatchGetReports"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:report-group/[[reportName]]"
      ]
    }
  ]
}
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:BatchGetReports"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:*:report-group/*"
      ]
    }
  ]
}
```

StartBuildBatch

Static resources

Run a Job (.sync)

```
{
```

```

"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "codebuild:StartBuildBatch",
      "codebuild:StopBuildBatch",
      "codebuild:BatchGetBuildBatches"
    ],
    "Resource": [
      "arn:aws:codebuild:[[region]]:[[accountId]]:project/[[projectName]]"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "events:PutTargets",
      "events:PutRule",
      "events:DescribeRule"
    ],
    "Resource": [
      "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventForCodeBuildStartBuildBatchRule"
    ]
  }
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuildBatch"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:project/[[projectName]]"
      ]
    }
  ]
}

```

```
    ]
  }
}
```

Dynamic resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuildBatch",
        "codebuild:StopBuildBatch",
        "codebuild:BatchGetBuildBatches"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:project/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventForCodeBuildStartBuildBatchRule"
      ]
    }
  ]
}
```

Request Response

```
{
  "Version": "2012-10-17",
```

```

    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "codebuild:StartBuildBatch"
        ],
        "Resource": [
          "arn:aws:codebuild:[[region]]:[[accountId]]:project/*"
        ]
      }
    ]
  }
}

```

StopBuildBatch

Static resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StopBuildBatch"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:project/[[projectName]]"
      ]
    }
  ]
}

```

Dynamic resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StopBuildBatch"
      ],

```

```

        "Resource": [
            "arn:aws:codebuild:[[region]]:[[accountId]]:project/*"
        ]
    }
]
}

```

RetryBuildBatch

Static resources

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:RetryBuildBatch",
        "codebuild:StopBuildBatch",
        "codebuild:BatchGetBuildBatches"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:project/[[projectName]]"
      ]
    }
  ]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:RetryBuildBatch"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:project/[[projectName]]"
      ]
    }
  ]
}

```



```

    ]
  }
]
}

```

Dynamic resources

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:RetryBuildBatch",
        "codebuild:StopBuildBatch",
        "codebuild:BatchGetBuildBatches"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:project/*"
      ]
    }
  ]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:RetryBuildBatch"
      ],
      "Resource": [
        "arn:aws:codebuild:[[region]]:[[accountId]]:project/*"
      ]
    }
  ]
}

```

```
    ]  
  }  
}
```

DeleteBuildBatch

Static resources

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "codebuild:DeleteBuildBatch"  
      ],  
      "Resource": [  
        "arn:aws:codebuild:{{region}}:{{accountId}}:project/{{projectName}}"  
      ]  
    }  
  ]  
}
```

Dynamic resources

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "codebuild:DeleteBuildBatch"  
      ],  
      "Resource": [  
        "arn:aws:codebuild:{{region}}:{{accountId}}:project/*"  
      ]  
    }  
  ]  
}
```

IAM policies for Amazon DynamoDB

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb>DeleteItem"
      ],
      "Resource": [
        "arn:aws:dynamodb:{{region}}:{{accountId}}:table/{{tableName}}"
      ]
    }
  ]
}
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb>DeleteItem"
      ],
      "Resource": "*"
    }
  ]
}
```

```
}

```

For more information about the IAM policies for all DynamoDB API actions, see [IAM policies with DynamoDB](#) in the *Amazon DynamoDB Developer Guide*. Additionally, for information about the IAM policies for PartiQL for DynamoDB, see [IAM policies with PartiQL for DynamoDB](#) in the *Amazon DynamoDB Developer Guide*.

IAM policies for Amazon ECS/AWS Fargate

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

Because the value for TaskId is not known until the task is submitted, Step Functions creates a more privileged "Resource": "*" policy.

Note

You can only stop Amazon Elastic Container Service (Amazon ECS) tasks that were started by Step Functions, despite the "*" IAM policy.

Run a Job (.sync)

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecs:RunTask"
      ],
      "Resource": [
        "arn:aws:ecs:[region]:
[[accountId]]:task-definition/[taskDefinition]:[revisionNumber]"
      ]
    },
    {
      "Effect": "Allow",
```

```

        "Action": [
            "ecs:StopTask",
            "ecs:DescribeTasks"
        ],
        "Resource": "*"
    },
    {
        "Effect": "Allow",
        "Action": [
            "events:PutTargets",
            "events:PutRule",
            "events:DescribeRule"
        ],
        "Resource": [
            "arn:aws:events:[[region]]:
[[accountId]]:rule/StepFunctionsGetEventsForECSTaskRule"
        ]
    }
]
}

```

Dynamic resources

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask",
                "ecs:StopTask",
                "ecs:DescribeTasks"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:[[region]]:
[[accountId]]:rule/StepFunctionsGetEventsForECSTaskRule"
            ]
        }
    ]
}

```

```

    ],
    "Resource": [
      "arn:aws:events:[region]:
[[accountId]]:rule/StepFunctionsGetEventsForECSTaskRule"
    ]
  }
]
}

```

Request Response and Callback (.waitForTaskToken)

Static resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecs:RunTask"
      ],
      "Resource": [
        "arn:aws:ecs:[region]:
[[accountId]]:task-definition/[taskDefinition]:[revisionNumber]"
      ]
    }
  ]
}

```

Dynamic resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecs:RunTask"
      ],
      "Resource": "*"
    }
  ]
}

```

```
}
```

If your scheduled Amazon ECS tasks require the use of a task execution role, a task role, or a task role override, then you must add `iam:PassRole` permissions for each task execution role, task role, or task role override to the CloudWatch Events IAM role of the calling entity, which in this case is Step Functions.

IAM policies for Amazon EKS

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

CreateCluster

Resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "eks:CreateCluster"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "eks:DescribeCluster",
        "eks>DeleteCluster"
      ],
      "Resource": "arn:aws:eks:sa-east-1:444455556666:cluster/*"
    },
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": [
        "arn:aws:iam::444455556666:role/StepFunctionsSample-EKSClusterManag-
        EKSServiceRole-ANPAJ2UCCR6DPCEXAMPLE"
      ],
    }
  ]
}
```

```

        "Condition": {
            "StringEquals": {
                "iam:PassedToService": "eks.amazonaws.com"
            }
        }
    ]
}

```

CreateNodeGroup

Resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:DescribeSubnets",
        "eks:CreateNodegroup"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "eks:DescribeNodegroup",
        "eks>DeleteNodegroup"
      ],
      "Resource": "arn:aws:eks:sa-east-1:444455556666:nodegroup/*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:GetRole",
        "iam:ListAttachedRolePolicies"
      ],
      "Resource": "arn:aws:iam::444455556666:role/*"
    },
    {

```



```

    "Effect": "Allow",
    "Action": "iam:PassRole",
    "Resource": [
        "arn:aws:iam::444455556666:role/StepFunctionsSample-EKSClusterMan-
NodeInstanceRole-ANPAJ2UCCR6DPCEXAMPLE"
    ],
    "Condition": {
        "StringEquals": {
            "iam:PassedToService": "eks.amazonaws.com"
        }
    }
}
]
}

```

DeleteCluster

Resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "eks:DeleteCluster",
        "eks:DescribeCluster"
      ],
      "Resource": [
        "arn:aws:eks:sa-east-1:444455556666:cluster/ExampleCluster"
      ]
    }
  ]
}

```

DeleteNodegroup

Resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {

```

```

        "Effect": "Allow",
        "Action": [
            "eks:DeleteNodegroup",
            "eks:DescribeNodegroup"
        ],
        "Resource": [
            "arn:aws:eks:sa-east-1:444455556666:nodegroup/ExampleCluster/
ExampleNodegroup/*"
        ]
    }
]
}

```

For more information about using Amazon EKS with Step Functions, see [Call Amazon EKS with Step Functions](#).

IAM policies for Amazon EMR

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

addStep

Static resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:AddJobFlowSteps",
        "elasticmapreduce:DescribeStep",
        "elasticmapreduce:CancelSteps"
      ],
      "Resource": [
        "arn:aws:elasticmapreduce:[region]:[accountId]:cluster/[clusterId]"
      ]
    }
  ]
}

```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:AddJobFlowSteps",
        "elasticmapreduce:DescribeStep",
        "elasticmapreduce:CancelSteps"
      ],
      "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
    }
  ]
}
```

cancelStep

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "elasticmapreduce:CancelSteps",
      "Resource": [
        "arn:aws:elasticmapreduce:{{region}}:{{accountId}}:cluster/{{clusterId}}"
      ]
    }
  ]
}
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```

        "Action": "elasticmapreduce:CancelSteps",
        "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
    }
]
}

```

createCluster

Static resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:RunJobFlow",
        "elasticmapreduce:DescribeCluster",
        "elasticmapreduce:TerminateJobFlows"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": [
        "arn:aws:iam::{{account}}:role/[[roleName]]"
      ]
    }
  ]
}

```

setClusterTerminationProtection

Static resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "elasticmapreduce:SetTerminationProtection",
      "Resource": [

```

```

    "arn:aws:elasticmapreduce:[[region]]:[[accountId]]:cluster/[[clusterId]]"
  ]
}

```

Dynamic resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "elasticmapreduce:SetTerminationProtection",
      "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
    }
  ]
}

```

modifyInstanceFleetByName

Static resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:ModifyInstanceFleet",
        "elasticmapreduce:ListInstanceFleets"
      ],
      "Resource": [
        "arn:aws:elasticmapreduce:[[region]]:[[accountId]]:cluster/[[clusterId]]"
      ]
    }
  ]
}

```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:ModifyInstanceFleet",
        "elasticmapreduce:ListInstanceFleets"
      ],
      "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
    }
  ]
}
```

modifyInstanceGroupByName

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:ModifyInstanceGroups",
        "elasticmapreduce:ListInstanceGroups"
      ],
      "Resource": [
        "arn:aws:elasticmapreduce:{{region}}:{{accountId}}:cluster/{{clusterId}}"
      ]
    }
  ]
}
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
```

```

        "elasticmapreduce:ModifyInstanceGroups",
        "elasticmapreduce:ListInstanceGroups"
    ],
    "Resource": "*"
}
]
}

```

terminateCluster

Static resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:TerminateJobFlows",
        "elasticmapreduce:DescribeCluster"
      ],
      "Resource": [
        "arn:aws:elasticmapreduce:{{region}}:{{accountId}}:cluster/{{clusterId}}"
      ]
    }
  ]
}

```

Dynamic resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:TerminateJobFlows",
        "elasticmapreduce:DescribeCluster"
      ],
      "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
    }
  ]
}

```

IAM policies for Amazon EMR on EKS

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

CreateVirtualCluster

Resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-containers:CreateVirtualCluster"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "iam:CreateServiceLinkedRole",
      "Resource": "arn:aws:iam:{{accountId}}:role/aws-service-role/emr-containers.amazonaws.com/AnAWSServiceRoleForAmazonEMRContainers",
      "Condition": {
        "StringLike": {
          "iam:AWSServiceName": "emr-containers.amazonaws.com"
        }
      }
    }
  ]
}
```

DeleteVirtualCluster

Static resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
```



```

{
  "Effect": "Allow",
  "Action": [
    "emr-containers:DeleteVirtualCluster",
    "emr-containers:DescribeVirtualCluster"
  ],
  "Resource": [
    "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/
[[virtualClusterId]]"
  ]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-containers:DeleteVirtualCluster"
      ],
      "Resource": [
        "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/
[[virtualClusterId]]"
      ]
    }
  ]
}

```

Dynamic resources

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [

```

```

        "emr-containers:DeleteVirtualCluster",
        "emr-containers:DescribeVirtualCluster"
    ],
    "Resource": [
        "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/*"
    ]
}
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-containers:DeleteVirtualCluster"
      ],
      "Resource": [
        "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/*"
      ]
    }
  ]
}

```

StartJobRun

Static resources

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "emr-containers:StartJobRun",
      "Resource": [
        "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/
[[virtualClusterId]]"
      ]
    }
  ]
}

```

```

    ],
    "Condition": {
      "StringEquals": {
        "emr-containers:ExecutionRoleArn": [
          "[[executionRoleArn]]"
        ]
      }
    }
  },
  {
    "Effect": "Allow",
    "Action": [
      "emr-containers:DescribeJobRun",
      "emr-containers:CancelJobRun"
    ],
    "Resource": [
      "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/[[virtualClusterId]]/jobruns/*"
    ]
  }
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "emr-containers:StartJobRun",
      "Resource": [
        "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/[[virtualClusterId]]"
      ],
      "Condition": {
        "StringEquals": {
          "emr-containers:ExecutionRoleArn": [
            "[[executionRoleArn]]"
          ]
        }
      }
    }
  ]
}

```

```
]
}
```

Dynamic resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "emr-containers:StartJobRun",
      "Resource": [
        "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/*"
      ],
      "Condition": {
        "StringEquals": {
          "emr-containers:ExecutionRoleArn": [
            "[[executionRoleArn]]"
          ]
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "emr-containers:DescribeJobRun",
        "emr-containers:CancelJobRun"
      ],
      "Resource": [
        "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/*"
      ]
    }
  ]
}
```

Request Response

```
{
  "Version": "2012-10-17",
  "Statement": [
```

```
{
  "Effect": "Allow",
  "Action": "emr-containers:StartJobRun",
  "Resource": [
    "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/*"
  ],
  "Condition": {
    "StringEquals": {
      "emr-containers:ExecutionRoleArn": [
        "[[executionRoleArn]]"
      ]
    }
  }
}
```

IAM policies for Amazon EMR Serverless

When you create a state machine using the console, Step Functions automatically creates an execution role for your state machine with the least privileges required. These automatically generated IAM roles are valid for the AWS Region in which you create the state machine.

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

We recommend that when you create IAM policies, do not include wildcards in the policies. As a security best practice, you should scope your policies down as much as possible. You should use dynamic policies only when certain input parameters are not known during runtime.

Further, administrator users should be careful when granting non-administrator users execution roles for running the state machines. We recommend that you include `passRole` policies in the execution roles if you're creating policies on your own. We also recommend that you add the `aws:SourceARN` and `aws:SourceAccount` context keys in the execution roles.

In this topic

- [IAM policy examples for EMR Serverless integration with Step Functions](#)

IAM policy examples for EMR Serverless integration with Step Functions

- [IAM policy example for CreateApplication](#)
- [IAM policy example for StartApplication](#)
- [IAM policy example for StopApplication](#)
- [IAM policy example for DeleteApplication](#)
- [IAM policy example for StartJobRun](#)
- [IAM policy example for CancelJobRun](#)

IAM policy example for CreateApplication

The following is an IAM policy example for a state machine with a CreateApplication [Task state](#) state.

Note

You need to specify the CreateServiceLinkedRole permissions in your IAM policies during the creation of the first ever application in your account. Thereafter, you need not add this permission. For information about CreateServiceLinkedRole, see [CreateServiceLinkedRole](#) in the <https://docs.aws.amazon.com/IAM/latest/APIReference/>.

Static and dynamic resources for the following policies are the same.

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:CreateApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/*"
      ]
    },
    {
```

```

    "Effect": "Allow",
    "Action": [
        "emr-serverless:GetApplication",
        "emr-serverless:DeleteApplication"
    ],
    "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
    ],
    "Resource": [
        "arn:aws:events:{{region}}:{{accountId}}:rule/
StepFunctionsGetEventsForEMRServerlessApplicationRule"
    ]
},
{
    "Effect": "Allow",
    "Action": "iam:CreateServiceLinkedRole",
    "Resource": "arn:aws:iam::{{accountId}}:role/aws-service-role/ops.emr-
serverless.amazonaws.com/AWSServiceRoleForAmazonEMRServerless",
    "Condition": {
        "StringLike": {
            "iam:AWSServiceName": "ops.emr-serverless.amazonaws.com"
        }
    }
}
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",

```

```

    "Action": [
      "emr-serverless:CreateApplication"
    ],
    "Resource": [
      "arn:aws:emr-serverless:{{region}}:{{accountId}}:/*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": "iam:CreateServiceLinkedRole",
    "Resource": "arn:aws:iam:{{accountId}}:role/aws-service-role/ops.emr-serverless.amazonaws.com/AWSServiceRoleForAmazonEMRServerless*",
    "Condition": {
      "StringLike": {
        "iam:AWSServiceName": "ops.emr-serverless.amazonaws.com"
      }
    }
  }
]
}

```

IAM policy example for StartApplication

Static resources

The following are IAM policy examples for static resources when you use a state machine with a StartApplication [Task state](#) state.

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StartApplication",
        "emr-serverless:GetApplication",
        "emr-serverless:StopApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/[[applicationId]]"
      ]
    }
  ]
}

```



```

    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "events:PutTargets",
      "events:PutRule",
      "events:DescribeRule"
    ],
    "Resource": [
      "arn:aws:events:{{region}}:
{{accountId}}:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
    ]
  }
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StartApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/
[[applicationId]]"
      ]
    }
  ]
}

```

Dynamic resources

The following are IAM policy examples for dynamic resources when you use a state machine with a StartApplication [Task state](#) state.

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StartApplication",
        "emr-serverless:GetApplication",
        "emr-serverless:StopApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:{{region}}:
        {{accountId}}:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
      ]
    }
  ]
}
```

Request Response

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StartApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
      ]
    }
  ]
}
```

```

    ]
  }
]
}

```

IAM policy example for StopApplication

Static resources

The following are IAM policy examples for static resources when you use a state machine with a StopApplication [Task state](#) state.

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StopApplication",
        "emr-serverless:GetApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/
[[applicationId]]"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:{{region}}:
{{accountId}}:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
      ]
    }
  ]
}

```

Request Response

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StopApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/
[[applicationId]]"
      ]
    }
  ]
}
```

Dynamic resources

The following are IAM policy examples for dynamic resources when you use a state machine with a StopApplication [Task state](#) state.

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StopApplication",
        "emr-serverless:GetApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",

```

```

        "events:PutRule",
        "events:DescribeRule"
    ],
    "Resource": [
        "arn:aws:events:{{region}}:
{{accountId}}:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
    ]
}
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StopApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
      ]
    }
  ]
}

```

IAM policy example for DeleteApplication

Static resources

The following are IAM policy examples for static resources when you use a state machine with a DeleteApplication [Task state](#) state.

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",

```

```

    "Action": [
      "emr-serverless:DeleteApplication",
      "emr-serverless:GetApplication"
    ],
    "Resource": [
      "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/
[[applicationId]]"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "events:PutTargets",
      "events:PutRule",
      "events:DescribeRule"
    ],
    "Resource": [
      "arn:aws:events:{{region}}:
{{accountId}}:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
    ]
  }
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:DeleteApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/
[[applicationId]]"
      ]
    }
  ]
}

```

Dynamic resources

The following are IAM policy examples for dynamic resources when you use a state machine with a `DeleteApplication` [Task state](#) state.

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:DeleteApplication",
        "emr-serverless:GetApplication"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:{{region}}:
        {{accountId}}:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
      ]
    }
  ]
}
```

Request Response

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
```

```

        "emr-serverless:DeleteApplication"
    ],
    "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
    ]
}
]
}

```

IAM policy example for StartJobRun

Static resources

The following are IAM policy examples for static resources when you use a state machine with a StartJobRun [Task state](#) state.

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StartJobRun"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/
[[applicationId]]"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": [
        "[[jobExecutionRoleArn]]"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "emr-serverless.amazonaws.com"
        }
      }
    }
  ],
}

```



```

    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:GetJobRun",
        "emr-serverless:CancelJobRun"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/
[[applicationId]]/jobruns/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:{{region}}:
{{accountId}}:rule/StepFunctionsGetEventsForEMRServerlessJobRule"
      ]
    }
  ]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StartJobRun"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/
[[applicationId]]"
      ]
    },
    {
      "Effect": "Allow",

```

```

        "Action": "iam:PassRole",
        "Resource": [
            "[[jobExecutionRoleArn]]"
        ],
        "Condition": {
            "StringEquals": {
                "iam:PassedToService": "emr-serverless.amazonaws.com"
            }
        }
    }
]
}

```

Dynamic resources

The following are IAM policy examples for dynamic resources when you use a state machine with a `StartJobRun` [Task state](#) state.

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:StartJobRun",
        "emr-serverless:GetJobRun",
        "emr-serverless:CancelJobRun"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": [
        "[[jobExecutionRoleArn]]"
      ],
      "Condition": {
        "StringEquals": {

```

```

        "iam:PassedToService": "emr-serverless.amazonaws.com"
    }
}
},
{
    "Effect": "Allow",
    "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
    ],
    "Resource": [
        "arn:aws:events:{{region}}:
{{accountId}}:rule/StepFunctionsGetEventsForEMRServerlessJobRule"
    ]
}
]
}

```

Request Response

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "emr-serverless:StartJobRun"
            ],
            "Resource": [
                "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": [
                "[[jobExecutionRoleArn]]"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "emr-serverless.amazonaws.com"
                }
            }
        }
    ]
}

```

```

    }
  }
]
}

```

IAM policy example for CancelJobRun

Static resources

The following are IAM policy examples for static resources when you use a state machine with a CancelJobRun [Task state](#) state.

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:CancelJobRun",
        "emr-serverless:GetJobRun"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/
[[applicationId]]/jobruns/[[jobRunId]]"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:{{region}}:
{{accountId}}:rule/StepFunctionsGetEventsForEMRServerlessJobRule"
      ]
    }
  ]
}

```

Request Response

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:CancelJobRun"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/
[[applicationId]]/jobruns/[[jobRunId]]"
      ]
    }
  ]
}
```

Dynamic resources

The following are IAM policy examples for dynamic resources when you use a state machine with a `CancelJobRun` [Task state](#) state.

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:CancelJobRun",
        "emr-serverless:GetJobRun"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",

```

```

        "events:PutRule",
        "events:DescribeRule"
    ],
    "Resource": [
        "arn:aws:events:{{region}}:
{{accountId}}:rule/StepFunctionsGetEventsForEMRServerlessJobRule"
    ]
}
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:CancelJobRun"
      ],
      "Resource": [
        "arn:aws:emr-serverless:{{region}}:{{accountId}}:/applications/*"
      ]
    }
  ]
}

```

IAM policies for Amazon EventBridge

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

PutEvents

Static resources

```

{
  "Version": "2012-10-17",
  "Statement": [

```

```

    {
      "Action": [
        "events:PutEvents"
      ],
      "Resource": [
        "arn:aws:events:us-east-1:123456789012:event-bus/stepfunctions-
sampleproject-eventbus"
      ],
      "Effect": "Allow"
    }
  ]
}

```

Dynamic resources

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "events:PutEvents"
      ],
      "Resource": "arn:aws:events:*:*:event-bus/*"
    }
  ]
}

```

For more information about using EventBridge with Step Functions, see [Call EventBridge with Step Functions](#).

IAM policies for AWS Lambda

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

AWS Step Functions generates an IAM policy based on your state machine definition. For a state machine with two AWS Lambda task states that call `function1` and `function2`, a policy with `lambda:Invoke` permissions for the two functions must be used.

This is shown in the following example.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:[[region]]:[[accountId]]:function:[[function1]]",
        "arn:aws:lambda:[[region]]:[[accountId]]:function:[[function2]]"
      ]
    }
  ]
}
```

IAM policies for AWS Elemental MediaConvert

The following example templates show how AWS Step Functions requires you to set up your IAM policies based on the resources in your state machine definition. You can use the IAM console to add any missing role policies. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

Because MediaConvert provides partial support for resource-level access control, you must use "Resource": "*".

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "mediaconvert:CreateJob",

```



```

        "mediaconvert:GetJob",
        "mediaconvert:CancelJob"
    ],
    "Resource": "*"
  },
  {
    "Effect": "Allow",
    "Action": [
      "events:PutTargets",
      "events:PutRule",
      "events:DescribeRule"
    ],
    "Resource": [
      "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForMediaConvertJobRule"
    ]
  }
]
}

```

Request Response

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "mediaconvert:CreateJob"
      ],
      "Resource": "*"
    }
  ]
}

```

IAM policies for AWS Glue

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

AWS Glue does not have resource-based control.

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "glue:StartJobRun",
        "glue:GetJobRun",
        "glue:GetJobRuns",
        "glue:BatchStopJobRun"
      ],
      "Resource": "*"
    }
  ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "glue:StartJobRun"
      ],
      "Resource": "*"
    }
  ]
}
```

IAM policies for AWS Glue DataBrew

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "databrew:startJobRun",
        "databrew:listJobRuns",
        "databrew:stopJobRun"
      ],
      "Resource": [
        "arn:aws:databrew:{{region}}:{{accountId}}:job/*"
      ]
    }
  ]
}
```

Request Response

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "databrew:startJobRun"
      ],
      "Resource": [
        "arn:aws:databrew:{{region}}:{{accountId}}:job/*"
      ]
    }
  ]
}
```

IAM policies for Amazon SageMaker

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

Note

For these examples, `[[roleArn]]` refers to the Amazon Resource Name (ARN) of the IAM role that SageMaker uses to access model artifacts and docker images for deployment on ML compute instances, or for batch transform jobs. For more information, see [Amazon SageMaker Roles](#).

CreateTrainingJob

Static resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",
        "sagemaker:DescribeTrainingJob",
        "sagemaker:StopTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:training-
job/[[trainingJobName]]*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

```

    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "iam:PassRole"
    ],
    "Resource": [
      "[[roleArn]]"
    ],
    "Condition": {
      "StringEquals": {
        "iam:PassedToService": "sagemaker.amazonaws.com"
      }
    }
  },
  {
    "Effect": "Allow",
    "Action": [
      "events:PutTargets",
      "events:PutRule",
      "events:DescribeRule"
    ],
    "Resource": [
      "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForSageMakerTrainingJobsRule"
    ]
  }
]
}

```

Request Response and Callback (.waitForTaskToken)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob"
      ],
      "Resource": [

```

```

    "arn:aws:sagemaker:[[region]]:[[accountId]]:training-
job/[[trainingJobName]]*"
  ],
  {
    "Effect": "Allow",
    "Action": [
      "sagemaker:ListTags"
    ],
    "Resource": [
      "*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "iam:PassRole"
    ],
    "Resource": [
      "[[roleArn]]"
    ],
    "Condition": {
      "StringEquals": {
        "iam:PassedToService": "sagemaker.amazonaws.com"
      }
    }
  }
]
}

```

Dynamic resources

.sync or .waitForTaskToken

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",

```

```
    "sagemaker:DescribeTrainingJob",
    "sagemaker:StopTrainingJob"
  ],
  "Resource": [
    "arn:aws:sagemaker:[[region]]:[[accountId]]:training-job/*"
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "sagemaker:ListTags"
  ],
  "Resource": [
    "*"
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "iam:PassRole"
  ],
  "Resource": [
    "[[roleArn]]"
  ],
  "Condition": {
    "StringEquals": {
      "iam:PassedToService": "sagemaker.amazonaws.com"
    }
  }
},
{
  "Effect": "Allow",
  "Action": [
    "events:PutTargets",
    "events:PutRule",
    "events:DescribeRule"
  ],
  "Resource": [
    "arn:aws:events:[[region]]:[[accountId]]:rule/StepFunctionsGetEventsForSageMakerTrainingJobsRule"
  ]
}
]
```

```
}
```

Request Response and Callback (.waitForTaskToken)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:training-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "[[roleArn]]"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}
```


CreateTransformJob

Note

AWS Step Functions will not automatically create a policy for `CreateTransformJob` when you create a state machine that integrates with SageMaker. You must attach an inline policy to the created role based on one of the following IAM examples.

Static resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTransformJob",
        "sagemaker:DescribeTransformJob",
        "sagemaker:StopTransformJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:transform-
job/[[transformJobName]]*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
    }
  ]
}
```

```

    "Resource": [
      "[[roleArn]]"
    ],
    "Condition": {
      "StringEquals": {
        "iam:PassedToService": "sagemaker.amazonaws.com"
      }
    }
  },
  {
    "Effect": "Allow",
    "Action": [
      "events:PutTargets",
      "events:PutRule",
      "events:DescribeRule"
    ],
    "Resource": [
      "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForSageMakerTransformJobsRule"
    ]
  }
]
}

```

Request Response and Callback (.waitForTaskToken)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTransformJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:transform-
job/[[[transformJobName]]*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [

```

```

    "sagemaker:ListTags"
  ],
  "Resource": [
    "*"
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "iam:PassRole"
  ],
  "Resource": [
    "[[roleArn]]"
  ],
  "Condition": {
    "StringEquals": {
      "iam:PassedToService": "sagemaker.amazonaws.com"
    }
  }
}
]
}

```

Dynamic resources

Run a Job (.sync)

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTransformJob",
        "sagemaker:DescribeTransformJob",
        "sagemaker:StopTransformJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:transform-job/*"
      ]
    }
  ],
}

```

```

{
  "Effect": "Allow",
  "Action": [
    "sagemaker:ListTags"
  ],
  "Resource": [
    "*"
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "iam:PassRole"
  ],
  "Resource": [
    "[[roleArn]]"
  ],
  "Condition": {
    "StringEquals": {
      "iam:PassedToService": "sagemaker.amazonaws.com"
    }
  }
},
{
  "Effect": "Allow",
  "Action": [
    "events:PutTargets",
    "events:PutRule",
    "events:DescribeRule"
  ],
  "Resource": [
    "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForSageMakerTransformJobsRule"
  ]
}
]
}

```

Request Response and Callback (.waitForTaskToken)

```

{
  "Version": "2012-10-17",

```

```
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "sagemaker:CreateTransformJob"
    ],
    "Resource": [
      "arn:aws:sagemaker:[[region]]:[[accountId]]:transform-job/*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "sagemaker:ListTags"
    ],
    "Resource": [
      "*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "iam:PassRole"
    ],
    "Resource": [
      "[[roleArn]]"
    ],
    "Condition": {
      "StringEquals": {
        "iam:PassedToService": "sagemaker.amazonaws.com"
      }
    }
  }
]
```

IAM policies for Amazon SNS

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sns:Publish"
      ],
      "Resource": [
        "arn:aws:sns:[[region]]:[[accountId]]:[[topicName]]"
      ]
    }
  ]
}
```

Resources based on a Path, or publishing to TargetArn or PhoneNumber

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sns:Publish"
      ],
      "Resource": "*"
    }
  ]
}
```

IAM policies for Amazon SQS

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

Static resources

```
{
```

```
"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "sqs:SendMessage"
    ],
    "Resource": [
      "arn:aws:sqs:[[region]]:[[accountId]]:[[queueName]]"
    ]
  }
]
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sqs:SendMessage"
      ],
      "Resource": "*"
    }
  ]
}
```

IAM policies for AWS Step Functions

For a state machine that calls `StartExecution` for a single nested workflow execution, use an IAM policy that limits permissions to that state machine.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ]
    }
  ]
}
```

```

    ],
    "Resource": [
      "arn:aws:states:[[region]]:[[accountId]]:stateMachine:[[stateMachineName]]"
    ]
  }
]
}

```

For more information, see the following:

- [Using AWS Step Functions with other services](#)
- [Pass parameters to a service API](#)
- [Manage AWS Step Functions Executions as an Integrated Service](#)

Synchronous

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": [
        "arn:aws:states:[[region]]:[[accountId]]:stateMachine:
[[stateMachineName]]"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeExecution",
        "states:StopExecution"
      ],
      "Resource": [
        "arn:aws:states:[[region]]:[[accountId]]:execution:[[stateMachineName]]:*"
      ]
    }
  ],
}

```



```

    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForStepFunctionsExecutionRule"
      ]
    }
  ]
}

```

Asynchronous

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": [
        "arn:aws:states:[[region]]:[[accountId]]:stateMachine:[[stateMachineName]]"
      ]
    }
  ]
}

```

For more information about nested workflow executions, see [Start Workflow Executions from a Task State](#).

IAM policies for AWS X-Ray

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see [IAM Policies for integrated services](#) and [Service integration patterns](#).

To enable X-Ray tracing, you will need an IAM policy with suitable permissions to allow tracing. If your state machine uses other integrated services, you may need additional IAM policies. See the IAM policies for your specific service integrations.

When you create a state machine with X-Ray tracing enabled, an IAM policy is automatically created.

Note

If you enable X-Ray tracing for an existing state machine you must ensure that you add a policy with sufficient permissions to enable X-Ray traces.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "xray:PutTraceSegments",
        "xray:PutTelemetryRecords",
        "xray:GetSamplingRules",
        "xray:GetSamplingTargets"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

For more information about using X-Ray with Step Functions, see [AWS X-Ray and Step Functions](#).

Activities or No Tasks

For a state machine that has only Activity tasks, or no tasks at all, use an IAM policy that denies access to all actions and resources.

```
{
  "Version": "2012-10-17",
```

```
"Statement": [  
  {  
    "Effect": "Deny",  
    "Action": "*",  
    "Resource": "*"  }  
]
```

For more information about using `Activity` tasks, see [Activities](#).

IAM policies for using Distributed Map state

When you create workflows with the Step Functions console, Step Functions can automatically generate IAM policies based on the resources in your workflow definition. These policies include the least privileges necessary to allow the state machine role to invoke the [StartExecution](#) API action for the *Distributed Map state*. These policies also include the least privileges necessary for Step Functions to access AWS resources, such as Amazon S3 buckets and objects and Lambda functions. We highly recommend that you include only those permissions that are necessary in your IAM policies. For example, if your workflow includes a Map state in Distributed mode, scope your policies down to the specific Amazon S3 bucket and folder that contains your dataset.

Important

If you specify an Amazon S3 bucket and object, or prefix, with a [reference path](#) to an existing key-value pair in your *Distributed Map state* input, make sure that you update the IAM policies for your workflow. Scope the policies down to the bucket and object names the path resolves to at runtime.

In this topic:

- [Example of IAM policy for running a Distributed Map state](#)
- [Example of IAM policy for redriving a Distributed Map](#)
- [Examples of IAM policies for reading data from Amazon S3 datasets](#)
- [Example of IAM policy for writing data to an Amazon S3 bucket](#)

Example of IAM policy for running a Distributed Map state

When you include a *Distributed Map state* in your workflows, Step Functions needs appropriate permissions to allow the state machine role to invoke the [StartExecution](#) API action for the *Distributed Map state*.

The following IAM policy example grants the least privileges required to your state machine role for running the *Distributed Map state*.

Note

Make sure that you replace *stateMachineName* with the name of the state machine in which you're using the *Distributed Map state*. For example, `arn:aws:states:us-east-2:123456789012:stateMachine:mystateMachine`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": [
        "arn:aws:states:region:accountID:stateMachine:stateMachineName"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeExecution",
        "states:StopExecution"
      ],
      "Resource": "arn:aws:states:region:accountID:execution:stateMachineName:*"
    }
  ]
}
```

Example of IAM policy for redriving a Distributed Map

You can restart unsuccessful child workflow executions in a Map Run by [redriving](#) your [parent workflow](#). A redriven parent workflow redrives all the unsuccessful states, including Distributed Map. Make sure that your execution role has the least privileges necessary to allow it to invoke the [RedriveExecution](#) API action on the parent workflow.

The following IAM policy example grants the least privileges required to your state machine role for redriving a *Distributed Map state*.

Note

Make sure that you replace *stateMachineName* with the name of the state machine in which you're using the *Distributed Map state*. For example, `arn:aws:states:us-east-2:123456789012:stateMachine:mystateMachine`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:RedriveExecution"
      ],
      "Resource": "arn:aws:states:us-east-2:123456789012:execution:myStateMachine/myMapRunLabel:*"
    }
  ]
}
```

Examples of IAM policies for reading data from Amazon S3 datasets

The following IAM policy examples grant the least privileges required to access your Amazon S3 datasets using the [ListObjectsV2](#) and [GetObject](#) API actions.

Example IAM policy for Amazon S3 object as dataset

The following example shows an IAM policy that grants the least privileges to access the objects organized within *processImages* in an Amazon S3 bucket named *myBucket*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::myBucket"
      ],
      "Condition": {
        "StringLike": {
          "s3:prefix": [
            "processImages"
          ]
        }
      }
    }
  ]
}
```

Example IAM policy for a CSV file as dataset

The following example shows an IAM policy that grants least privileges to access a CSV file named *ratings.csv*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::myBucket/csvDataset/ratings.csv"
      ]
    }
  ]
}
```

Example IAM policy for an Amazon S3 inventory as dataset

The following example shows an IAM policy that grants least privileges to access an Amazon S3 inventory report.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::destination-prefix/source-bucket/config-ID/YYYY-MM-DDTHH-MMZ/manifest.json",
        "arn:aws:s3:::destination-prefix/source-bucket/config-ID/data/*"
      ]
    }
  ]
}
```

Example of IAM policy for writing data to an Amazon S3 bucket

The following IAM policy example grants the least privileges required to write your child workflow execution results to a folder named *csvJobs* in an Amazon S3 bucket using the [PutObject](#) API action.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload"
      ],
      "Resource": [
        "arn:aws:s3:::resultBucket/csvJobs/*"
      ]
    }
  ]
}
```

```
]
}
```

IAM permissions for AWS KMS key encrypted Amazon S3 bucket

Distributed Map state uses multipart uploads to write the child workflow execution results to an Amazon S3 bucket. If the bucket is encrypted using an AWS Key Management Service (AWS KMS) key, you must also include permissions in your IAM policy to perform the `kms:Decrypt`, `kms:Encrypt`, and `kms:GenerateDataKey` actions on the key. These permissions are required because Amazon S3 must decrypt and read data from the encrypted file parts before it completes the multipart upload.

The following IAM policy example grants permission to the `kms:Decrypt`, `kms:Encrypt`, and `kms:GenerateDataKey` actions on the key used to encrypt your Amazon S3 bucket.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": [
      "kms:Decrypt",
      "kms:Encrypt",
      "kms:GenerateDataKey"
    ],
    "Resource": [
      "arn:aws:kms:us-east-1:123456789012:key/111aa2bb-333c-4d44-5555-a111bb2c33dd"
    ]
  }
}
```

For more information, see [Uploading a large file to Amazon S3 with encryption using an AWS KMS key](#) in the *AWS Knowledge Center*.

If your IAM user or role is in the same AWS account as the KMS key, then you must have these permissions on the key policy. If your IAM user or role belongs to a different account than the KMS key, then you must have the permissions on both the key policy and your IAM user or role.

Tag-based Policies

Step Functions supports policies based on tags. For example, you could restrict access to all Step Functions resources that include a tag with the key `environment` and the value `production`.


```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "states:TagResource",
        "states:UntagResource",
        "states>DeleteActivity",
        "states>DeleteStateMachine",
        "states:StopExecution"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {"aws:ResourceTag/environment": "production"}
      }
    }
  ]
}
```

This policy will Deny the ability to delete state machines or activities, stop executions, and add or delete new tags for all resources that have been tagged as environment/production.

For tag-based authorization, state machine execution resources as shown in the following example inherit the tags associated with a state machine.

```
arn:<partition>:states:<Region>:<account-id>:execution:<StateMachineName>:<ExecutionId>
```

When you call [DescribeExecution](#) or other APIs in which you specify the execution resource ARN, Step Functions uses tags associated with the state machine to accept or deny the request while performing tag-based authorization. This helps you allow or deny access to state machine executions at the state machine level.

For more information about tagging, see the following:

- [Tagging in Step Functions](#)
- [Controlling Access Using IAM Tags](#)

Troubleshooting AWS Step Functions identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Step Functions and IAM.

Topics

- [I am not authorized to perform an action in Step Functions](#)
- [I am not authorized to perform iam:PassRole](#)
- [I want to allow people outside of my AWS account to access my Step Functions resources](#)

I am not authorized to perform an action in Step Functions

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` user tries to use the console to view details about a fictional `my-example-widget` resource but does not have the fictional `states:GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
states:GetWidget on resource: my-example-widget
```

In this case, Mateo's policy must be updated to allow him to access the `my-example-widget` resource using the `states:GetWidget` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Step Functions.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Step Functions. However, the action requires the service to have

permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Step Functions resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Step Functions supports these features, see [How AWS Step Functions works with IAM](#).
- To learn how to provide access to your resources across AWS accounts that you own, see [Providing access to an IAM user in another AWS account that you own](#) in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see [Providing access to AWS accounts owned by third parties](#) in the *IAM User Guide*.
- To learn how to provide access through identity federation, see [Providing access to externally authenticated users \(identity federation\)](#) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see [Cross account resource access in IAM](#) in the *IAM User Guide*.

Logging and Monitoring

For information about logging and monitoring in AWS Step Functions, see the [Logging and monitoring](#) section.

Compliance Validation for AWS Step Functions

Third-party auditors assess the security and compliance of AWS Step Functions as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others.

For a list of AWS services in scope of specific compliance programs, see [AWS Services in Scope by Compliance Program](#). For general information, see [AWS Compliance Programs](#).

You can download third-party audit reports using AWS Artifact. For more information, see [Downloading Reports in AWS Artifact](#).

Your compliance responsibility when using Step Functions is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- [Security and Compliance Quick Start Guides](#) – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- [Architecting for HIPAA Security and Compliance on Amazon Web Services](#) – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.
- [AWS Compliance Resources](#) – This collection of workbooks and guides might apply to your industry and location.
- [Evaluating Resources with Rules](#) in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- [AWS Security Hub](#) – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience in AWS Step Functions

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see [AWS Global Infrastructure](#).

In addition to the AWS global infrastructure, Step Functions offers several features to help support your data resiliency and backup needs.

Infrastructure Security in AWS Step Functions

As a managed service, AWS Step Functions is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see [AWS Cloud Security](#). To design your AWS environment using the best practices for infrastructure security, see [Infrastructure Protection](#) in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access Step Functions through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

You can call the AWS API operations from any network location, but Step Functions doesn't support resource-based access policies, which can include restrictions based on the source IP address. You can also use Step Functions policies to control access from specific Amazon Virtual Private Cloud (Amazon VPC) endpoints or specific VPCs. Effectively, this isolates network access to a given Step Functions resource from only the specific VPC within the AWS network.

Configuration and Vulnerability Analysis in AWS Step Functions

Configuration and IT controls are a shared responsibility between AWS and you, our customer. For more information, see the AWS [shared responsibility model](#).

Troubleshooting

If you encounter difficulties when working with Step Functions, use the following troubleshooting resources.

Topics

- [General troubleshooting](#)
- [Troubleshooting service integrations](#)
- [Troubleshooting activities](#)
- [Troubleshooting Express Workflows](#)

General troubleshooting

I'm unable to create a state machine.

The IAM role associated with the state machine might not have [sufficient permissions](#). Check the IAM role's permissions, including for AWS service integration tasks, X-Ray, and CloudWatch logging. Additional permissions are required for `.sync` task states.

I'm unable to use a JsonPath to reference the previous task's output.

For a JsonPath, a JSON key must end with `.$`. This means a JsonPath can only be used in a key-value pair. If you want to use a JsonPath other places, such as an array, you can use [intrinsic functions](#). For example, you could use something similar to the following:

Task A output:

```
{
  "sample": "test"
}
```

Task B:

```
{
  "JsonPathSample.{$": "$.sample"
}
```

Tip

Use the [data flow simulator in the Step Functions console](#) to test JSON path syntax, to better understand how data is manipulated within a state, and to see how data is passed between states.

There was a delay in state transitions.

For standard workflows, there is a limit on the number of state transitions. When you exceed the state transition limit, Step Functions delays state transitions until the bucket for the quota is filled. State transition limit throttling can be monitored by reviewing the `ExecutionThrottled` metric in the [Execution metrics](#) section of the CloudWatch Metrics page.

When I start new Standard Workflow executions, they fail with the `ExecutionLimitExceeded` error.

Step Functions has a limit of 1,000,000 open executions for each AWS account in each AWS Region. If you exceed this limit, Step Functions throws an `ExecutionLimitExceeded` error. This limit does not apply to Express Workflows. You can use the following [CloudWatch Metrics math](#) in the *Amazon CloudWatch User Guide* to approximate the number of open executions: `ExecutionsStarted - (ExecutionsSucceeded + ExecutionsTimedOut + ExecutionsFailed + ExecutionsAborted)`.

A failure on one branch in a parallel state causes the whole execution to fail.

This is an expected behavior. To avoid encountering failures when using a parallel state, configure Step Functions to [catch errors](#) thrown from each branch.

Troubleshooting service integrations

My job is complete in the downstream service, but in Step Functions the task state remains "In progress" or its completion is delayed.

For `.sync` service integration patterns, Step Functions uses EventBridge rules, downstream APIs, or a combination of both to detect the downstream job status. For some services, Step Functions

does not create EventBridge rules to monitor. For example, for the AWS Glue service integration, instead of using EventBridge rules, Step Functions makes a `glue:GetJobRun` call. Because of the frequency of API calls, there is a difference between the downstream task completion and the Step Functions task completion time. Step Functions requires IAM permissions to manage the EventBridge rules and to make calls to the downstream service. For more details about how insufficient permissions on your execution role can affect the completion of tasks, see [Additional permissions for tasks using the Run a Job pattern](#).

I want to return a JSON output from a nested state machine execution.

There are two Step Functions synchronous service integrations for Step Functions: `startExecution.sync` and `startExecution.sync:2`. Both wait for the nested state machine to complete, but they return different Output formats. You can use `startExecution.sync:2` to return a JSON output under Output.

I can't invoke a Lambda function from another account.

Accessing the Lambda function with cross-account support

If [cross-account access](#) of AWS resources is available in your Region, use the following method to invoke a Lambda function from another account.

To invoke a cross-account resource in your workflows, do the following:

1. Create an IAM role in the target account that contains the resource. This role grants the source account, containing the state machine, permissions to access the target account's resources.
2. In the Task state's definition, specify the target IAM role to be assumed by the state machine before invoking the cross-account resource.
3. Modify the trust policy in the target IAM role to allow the source account to assume this role temporarily. The trust policy must include the Amazon Resource Name (ARN) of the state machine defined in the source account. Also, define the appropriate permissions in the target IAM role to call the AWS resource.
4. Update the source account's execution role to include the required permission for assuming the target IAM role.

For an example, see [Tutorial: Accessing cross-account AWS resources](#).

Note

You can configure your state machine to assume an IAM role for accessing resources from multiple AWS accounts. However, a state machine can assume only one IAM role at a given time.

For an example of a Task state definition that specifies a cross-account resource, see [Task state's Credentials field examples](#).

Accessing the Lambda function without cross-account support

If cross-account access of AWS resources is unavailable in your Region, use the following method to invoke a Lambda function from another account.

In the Task state's Resource field, use `arn:aws:states:::lambda:invoke` and pass the FunctionArn in parameters. The IAM role that is associated with the state machine must have the right permissions to invoke cross-account Lambda functions: `lambda:invokeFunction`.

```
{
  "StartAt": "CallLambda",
  "States": {
    "CallLambda": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Parameters": {
        "FunctionName": "arn:aws:lambda:us-west-2:123456789012:function:my-function"
      },
      "End": true
    }
  }
}
```

I'm unable to see task tokens passed from `.waitForTaskToken` states.

In the Task state's Parameters field, you must pass a task token. For example, you could use something similar to the following code.

```
{
  "StartAt": "taskToken",
```

```
"States":{
  "taskToken":{
    "Type":"Task",
    "Resource":"arn:aws:states:::lambda:invoke.waitForTaskToken",
    "Parameters":{
      "FunctionName":"get-model-review-decision",
      "Payload":{
        "token.$":"$$$.Task.Token"
      },
    },
    "End":true
  }
}
```

Note

You can try to use `.waitForTaskToken` with any API action. However, some APIs don't have any suitable parameters.

Troubleshooting activities

My state machine execution is stuck at an activity state.

An activity task state doesn't start until you poll a task token by using the [GetActivityTask](#) API action. As a best practice, add a task level timeout in order to avoid a stuck execution. For more information, see [Use timeouts to avoid stuck executions](#).

If your state machine is stuck in the [ActivityScheduled](#) event, it indicates that your activity worker fleet has issues or is underscaled. You should monitor the [ActivityScheduleTime](#) CloudWatch metric and set an alarm when that time increases. However, to time out any stuck state machine executions in which the Activity state doesn't transition to the ActivityStarted state, define a timeout at state machine-level. To do this, specify a `TimeoutSeconds` field at the beginning of the state machine definition, outside of the `States` field.

My activity worker times out while waiting for a task token.

Workers use the [GetActivityTask](#) API action to retrieve a task with the specified activity ARN that is scheduled for execution by a running state machine. `GetActivityTask` starts a long poll, so

the service holds the HTTP connection open and responds as soon as a task becomes available. The maximum time the service hold the request before responding is 60 seconds. If no task is available within 60 seconds, the poll returns a taskToken with a null string. To avoid this timeout, configure a client side socket [with a timeout of at least 65](#) seconds in the AWS SDK or in the client you are using to make the API call.

Troubleshooting Express Workflows

My application times out before receiving a response from a [StartSyncExecution](#) API call.

Configure a client side socket timeout in the AWS SDK or client you use to make the API call. To receive a response, the timeout must have a value higher than the duration of the Express Workflow executions.

I'm unable to see the execution history in order to troubleshoot Express Workflow failures.

Express Workflows don't record execution history in AWS Step Functions. Instead, you must turn on CloudWatch logging. Once logging is turned on, you can use CloudWatch Logs Insights queries to review your Express Workflow executions. You can also view execution history for Express Workflow executions on the Step Functions console if you choose the **Enable** button in the **Executions** tab. For more information, see [Viewing and debugging executions on the Step Functions console](#).

To list executions based on duration:

```
fields ispresent(execution_arn) as exec_arn
| filter exec_arn
| filter type in ["ExecutionStarted", "ExecutionSucceeded", "ExecutionFailed",
"ExecutionAborted", "ExecutionTimedOut"]
| stats latest(type) as status,
  tomillis(earliest(event_timestamp)) as UTC_starttime,
  tomillis(latest(event_timestamp)) as UTC_endtime,
  latest(event_timestamp) - earliest(event_timestamp) as duration_in_ms by
  execution_arn
| sort duration desc
```

To list failed and cancelled executions:

```
fields ispresent(execution_arn) as isRes | filter type in ["ExecutionFailed",  
"ExecutionAborted", "ExecutionTimedOut"]
```

Related information

The following table lists related resources that you might find useful as you work with this service.

Resource	Description
AWS Step Functions API Reference	Descriptions of API actions, parameters, and data types and a list of errors that the service returns.
AWS Step Functions Command Line Reference	Descriptions of the AWS CLI commands that you can use to work with AWS Step Functions.
Product information for Step Functions	The primary webpage for information about Step Functions.
Discussion Forums	A community-based forum for developers to discuss technical questions related to Step Functions and other AWS services.
AWS Support Information	The primary webpage for information about AWS Support, a one-on-one, fast-response support channel to help you build and run applications on AWS infrastructure services.

Recent feature launches

The following table lists the Regions in which new Step Functions features are available.

Launch date	Feature name	Regions available
November 26, 2023	Invoke public HTTPS endpoints and test individual states	<ul style="list-style-type: none"> • US East (N. Virginia) – us-east-1 • US West (Oregon) – us-west-2 • US East (Ohio) – us-east-2 • Europe (Ireland) – eu-west-1 • Europe (Frankfurt) – eu-central-1 • Europe (Stockholm) – eu-north-1 • Asia Pacific (Sydney) – ap-southeast-2 • Asia Pacific (Tokyo) – ap-northeast-1 • Asia Pacific (Singapore) – ap-southeast-1
November 15, 2023	Redrive executions	For a complete list of the AWS Regions in which this feature is available, see the options in the Region dropdown list on the page titled AWS services by Region .
October 12, 2023	Optimized integration for Amazon EMR Serverless	For a complete list of the AWS Regions in which this feature is available, see the options in the Region dropdown list on

Launch date	Feature name	Regions available
		the page titled AWS services by Region .
September 07, 2023	Enhanced error handling	For a complete list of the AWS Regions in which this feature is available, see the options in the Region dropdown list on the page titled AWS services by Region .
August 31, 2023	Workflow Studio enhancements for a streamlined authoring experience	For a complete list of the AWS Regions in which this feature is available, see the options in the Region dropdown list on the page titled AWS services by Region .
June 22, 2023	Versions and aliases	For a complete list of the AWS Regions in which this feature is available, see the options in the Region dropdown list on the page titled AWS services by Region .
June 16, 2023	New AWS SDK integrations	For a complete list of the AWS Regions in which this feature is available, see the options in the Region dropdown list on the page titled AWS services by Region .

Launch date	Feature name	Regions available
December 01, 2022	Orchestrate large-scale parallel workflows for data processing with <i>Distributed Map state</i>	For a complete list of the AWS Regions in which this feature is available, see the options in the Region dropdown list on the page titled AWS services by Region .

Document history

This section lists major changes to the *AWS Step Functions Developer Guide*.

Change	Description	Date changed
Updates	<p>AWS managed policy updates - new permission: <code>states:ValidateStateMachineDefinition</code></p> <p>Added information about new permission to check the syntax of a state machine that you provide. To learn more, see AWS managed policies for AWS Step Functions.</p>	April 29, 2024
New feature	<p>Step Functions adds optimized integration for AWS Elemental MediaConvert</p> <p>AWS Elemental MediaConvert provides broadcast-grade video and audio file transcoding, which customers can automate with code to suit their media workflows. With the optimized integration for AWS Step Functions in MediaConvert, it is now possible to orchestrate using the low-code visual tool Workflow Studio. To learn more, see the documentation to Manage AWS Elemental MediaConvert with Step Functions.</p>	April 12, 2024
Updates	<p>AWS managed policy updates - Update to an existing policy: <code>AWSStepFunctionsReadOnlyAccess</code></p> <p>Added information about new read-only permissions for tags, distributed maps, and versions and aliases. To learn more, see AWS managed policies for AWS Step Functions.</p>	April 02, 2024
Updates	<p>Step Functions adds support for Open Workflow metrics</p> <p>With open workflow metrics, you now have account-level visibility into the number of standard workflows in progress as well as your open workflow limit. You</p>	February 29, 2024

Change	Description	Date changed
	<p>can manage workloads across all workflows, regardless of how they're started, to ensure smooth workflow operations. You can set CloudWatch alarms to monitor your workflows and proactively receive alerts as you approach your limits. Once alerted, you can effectively manage your workflows by taking actions such as stopping specific workflows or requesting a limit increase.</p> <p>Open workflow metrics is available to use in CloudWatch for standard workflows with no additional configuration required. To learn more, see Execution metrics.</p>	
Updates	Service integration additions and updates. For the list of new and updated AWS SDK integrations, see Change log for supported AWS SDK integrations . For the full list of services, see Supported AWS SDK service integrations .	January 18, 2024
New feature	Use Workflow Studio in Application Composer to build serverless workflows using AWS CloudFormation templates. For more information, see Using Workflow Studio in Application Composer .	November 27, 2023
New feature	<p>Step Functions now lets you directly invoke public HTTPS endpoints and test individual states using a new Test State API. For more information, see:</p> <ul style="list-style-type: none">• Call third-party APIs• Using TestState API to test a state	November 26, 2023

Change	Description	Date changed
New feature	Step Functions now integrates with Amazon Bedrock. For more information, see the following topics: <ul style="list-style-type: none">• Call Amazon Bedrock with Step Functions• IAM permissions for Amazon Bedrock• Perform AI prompt-chaining with Amazon Bedrock• Using AWS Step Functions with other services	November 26, 2023
New feature	Step Functions now lets you redrive workflow executions of type Standard from their point of failure. For more information, see Redriving executions and Redriving Map Runs .	November 15, 2023
Documentation-only update	Published a new topic that explains how to run state machines on a schedule using Amazon EventBridge Scheduler. For more information, see Using Amazon EventBridge Scheduler with AWS Step Functions .	October 16, 2023
New feature	Step Functions now integrates with Amazon EMR Serverless. For more information, see the following topics: <ul style="list-style-type: none">• Call Amazon EMR Serverless with Step Functions• Run an EMR Serverless job• Optimized integrations for Step Functions• Using AWS Step Functions with other services	October 12, 2023
Documentation-only update	Added information about running state machines on a schedule using Amazon EventBridge Scheduler. For more information, see Using EventBridge Scheduler .	October 05, 2023

Change	Description	Date changed
Update	Reorganized and updated the <i>Distributed Map state</i> topics for clarity, brevity, and establishing a clear journey map for new users. For more information, see Using Map state in Distributed mode to orchestrate large-scale parallel workloads .	October 6, 2023
Fixes	Fixed code samples in a tutorial to use AWS CDK v2. For more information, see Creating a Lambda state machine for Step Functions using AWS CDK .	September 19, 2023
Update	Added information about the enhanced error handling capabilities that Step Functions has introduced to identify errors clearly and implement retries with greater control. For more information, see Fail and Retrying after an error .	September 07, 2023
Update	Step Functions has added enhancements to Workflow Studio for streamlining workflow authoring experience. For more information, see AWS Step Functions Workflow Studio .	August 31, 2023
Documentation-only update	Added information about twice the actual metric count reported for the <code>ExecutionsStarted</code> metric. For more information, see Metrics that report a count .	July 25, 2023
Documentation-only update	Step Functions has added two new sample projects that demonstrate the following common use cases for the <i>Distributed Map state</i> : <ul style="list-style-type: none">• Processing a CSV file• Processing data in an Amazon S3 bucket	July 17, 2023
Documentation-only update	Published a new topic about deploying state machines using Terraform. For more information, see Deploying state machines using Terraform .	July 5, 2023

Change	Description	Date changed
Documentation-only update	<p>Updated the following procedures to match changes to the Amazon EventBridge interface.</p> <ul style="list-style-type: none">• Routing a Step Functions event to EventBridge• Starting a State Machine Execution in Response to Amazon S3 Events	June 26, 2023
New feature	<p>Step Functions now provides the ability to create multiple state machine versions and aliases for improved resiliency while deploying serverless workflows. For more information, see Manage continuous deployments with versions and aliases.</p>	June 22, 2023
Documentation-only update	<p>Improved the description of <code>TimeoutSeconds</code> and <code>HeartbeatSeconds</code> fields to describe how they're different from each other. For more information, see Task state fields.</p>	June 22, 2023
Documentation-only update	<p>Published a new section that describes how to flatten an array of arrays typically returned as result for Parallel and Map states. For more information, see Flattening an array of arrays.</p>	June 20, 2023
Update	<p>Step Functions has expanded support for AWS SDK integrations by adding seven AWS services and 468 new API actions. For more information, see Supported AWS SDK service integrations and Change log for supported AWS SDK integrations.</p>	June 16, 2023
Documentation-only update	<p>Published a new topic that lists the AWS Regions in which recently launched Step Functions features are available. For more information, see Recent feature launches.</p>	June 16, 2023

Change	Description	Date changed
Documentation-only update	Step Functions now includes a section about AWS User Notifications, an AWS service that acts as a central location for your AWS notifications in the AWS Management Console. For more information, see Using AWS User Notifications with Step Functions .	May 4, 2023
Documentation-only update	Added a new section that explains about the permissions needed to write child workflow execution results to an Amazon S3 bucket encrypted with an AWS Key Management Service (AWS KMS) key. For more information, see IAM permissions for AWS KMS key encrypted Amazon S3 bucket .	April 29, 2023
Documentation-only update	Added a new topic that explains about the Data flow simulator feature. For more information, see Data flow simulator .	April 14, 2023
Quota update	Added information about default quota of 1000 for open Map Runs in each account. For more information, see Quotas related to accounts .	April 05, 2023
Documentation-only update	Added a Note about unavailability of X-Ray tracing for the Distributed Map state . For more information, see AWS X-Ray and Step Functions .	March 21, 2023
Documentation-only update	Added information about how Step Functions handles tag-based authorization. For more information, see Tagging in Step Functions and Tag-based Policies .	March 15, 2023
Documentation-only update	Added information about how Step Functions parses CSV files used as input in Distributed Map state . For more information, see CSV file in an Amazon S3 bucket .	March 14, 2023

Change	Description	Date changed
Documentation-only update	Added information about how Step Functions handles cross-account invocations for the Run a Job (.sync) pattern. For more information, see Run a Job (.sync) .	March 01, 2023
Documentation-only update	Reduce the history retention period of your completed workflow executions from 90 days to 30 days. For more information about adjusting the retention period, see Execution guarantees and Quotas related to state machine executions .	February 21, 2023
Update	Step Functions has expanded support for AWS SDK integrations by adding 35 AWS services and 1100 new API actions. For more information, see Supported AWS SDK service integrations and Change log for supported AWS SDK integrations .	February 17, 2023
Documentation-only update	Published a Getting Started tutorial series that walks you through the process of creating a workflow for credit card application using Step Functions. For more information, see Getting started with AWS Step Functions .	December 30, 2022
New feature	Step Functions adds support to orchestrate large-scale parallel workflows for data processing using a new Distributed mode for Map state. For more information, see Using Map state in Distributed mode to orchestrate large-scale parallel workloads .	December 01, 2022
New feature	<p>Step Functions now supports access to cross-account AWS resources configured in other accounts. For more information, see</p> <ul style="list-style-type: none"> • Accessing resources in other AWS accounts in your workflows • Tutorial: Accessing cross-account AWS resources • Task state 	November 18, 2022

Change	Description	Date changed
Update	<p>Step Functions now provides a new console experience for viewing and debugging Express workflow executions. For more information see:</p> <ul style="list-style-type: none">• Standard and Express Workflow executions in the console• Viewing and debugging executions on the Step Functions console	October 18, 2022
Update	<p>Added support to optionally specify the <code>ExecutionRoleArn</code> parameter while using the <code>addStep</code> and <code>addStep.sync</code> APIs for the Amazon EMR optimized service integration. For more information, see Call Amazon EMR with Step Functions.</p>	September 20, 2022
Documentation-only update	<p>Added a new topic that provides recommendations about optimizing cost while building serverless workflows using Step Functions. For more information, see Cost-optimization using Express Workflows.</p>	September 15, 2022

Change	Description	Date changed
Update	<p>Step Functions adds support for 14 new intrinsic functions for performing data processing tasks, such as array manipulations, data encoding and decoding, hash calculations, JSON data manipulation, math function operations, and unique identifier generation.</p> <p>Documentation-only update:</p> <p>Grouped all the existing and newly introduced intrinsic functions into the following categories based on the type of data processing task they help you perform:</p> <ul style="list-style-type: none"> • Intrinsics for arrays • Intrinsics for data encoding and decoding • Intrinsic for hash calculation • Intrinsics for JSON data manipulation • Intrinsics for Math operations • Intrinsic for String operation • Intrinsic for unique identifier generation • Intrinsic for generic operation <p>For more information, see Intrinsic functions.</p>	August 31, 2022
Update	<p>Step Functions has expanded support for AWS SDK integrations by adding three more AWS services – AWS Billing Conductor, Amazon GameSparks, and Amazon Pinpoint SMS and Voice V2. For more information, see Change log for supported AWS SDK integrations.</p>	July 26, 2022
Documentation-only update	<p>Added a new topic to include a summary of all the updates made to AWS SDK integrations supported by Step Functions. For more information, see Change log for supported AWS SDK integrations</p>	July 26, 2022

Change	Description	Date changed
Documentation-only update	<i>AWS Step Functions Developer Guide</i> now includes details about the execution metrics that are emitted specifically for Express Workflows. For more information, see Execution metrics for Express Workflows .	June 09, 2022

Change	Description	Date changed
Update	<p data-bbox="477 275 1049 310">Step Functions console enhancements</p> <p data-bbox="477 354 1305 438">The console now features a redesigned Execution Details page that includes the following enhancements:</p> <ul data-bbox="477 483 1321 1367" style="list-style-type: none"><li data-bbox="477 483 1321 564">• Ability to identify the reason for a failed execution at a glance.<li data-bbox="477 588 1321 814">• Two new modes of visualizations for your state machine – Table view and Event view. These views also provide you the ability to apply filters to only view the information of interest. In addition, you can sort the Event view contents based on the event timestamps.<li data-bbox="477 837 1321 961">• Switch between the different iterations of Map state in the Graph view mode using a dropdown list or in the Table view mode's tree view for Map states.<li data-bbox="477 984 1321 1163">• View in-depth information about each state in the workflow, including the complete input and output data transfer path and retry attempts for Task or Parallel states.<li data-bbox="477 1186 1321 1367">• Miscellaneous enhancements including the option to copy the state machine's execution Amazon Resource Name, view the count of total state machine transitions, and export the execution details in JSON format. <p data-bbox="477 1444 915 1480">Documentation-only updates</p> <p data-bbox="477 1524 1321 1703">Added a new topic to explain the various types of information displayed in the Execution Details page. Also, added a tutorial to show how to examine this information. For more information, see:</p> <ul data-bbox="477 1747 1187 1831" style="list-style-type: none"><li data-bbox="477 1747 1187 1831">• Viewing and debugging executions on the Step Functions console	May 09, 2022

Change	Description	Date changed
	<ul style="list-style-type: none">• Tutorial: Examining state machine executions using the Step Functions console	
Update	<p>Step Functions now provides a workaround to prevent the confused deputy security issue, which arises when an entity (a service or an account) is coerced by a different entity to perform an action. For more information, see:</p> <ul style="list-style-type: none">• Prevent cross-service confused deputy issue	May 02, 2022
Update	<ul style="list-style-type: none">• Step Functions has expanded support for AWS SDK integrations by adding 21 more AWS services. For more information, see: Supported AWS SDK service integrations.• Documentation-only updates:<ul style="list-style-type: none">• Added a list of all the exception prefixes present in the exceptions that are generated when you erroneously perform an AWS SDK service integration with Step Functions. For more information, see: Supported AWS SDK service integrations.• Added a list of all the unsupported API actions for supported AWS SDK integrations. For more information, see: Unsupported API actions for supported services.• Added a list of all the supported AWS SDK integrations that are now deprecated. For more information, see: Deprecated AWS SDK service integrations.	April 19, 2022
New feature	<p>Step Functions Local now supports AWS SDK integration and mocking of service integrations. For more information, see:</p> <ul style="list-style-type: none">• Using Mocked Service Integrations	January 28, 2022

Change	Description	Date changed
New feature	<p>AWS Step Functions now supports creating an Amazon API Gateway REST API with synchronous express state machine as backend integration using the AWS Cloud Development Kit (AWS CDK). For more information, see:</p> <ul style="list-style-type: none">• Creating an API Gateway REST API with Synchronous Express State Machine Using the AWS CDK	December 10, 2021
Update	<p>Step Functions has added three new sample projects that demonstrate the integration of Step Functions and Amazon Athena's upgraded console. For more information, see:</p> <ul style="list-style-type: none">• Execute multiple queries (Amazon Athena, Amazon SNS)• Query large datasets (Amazon Athena, Amazon S3, AWS Glue, Amazon SNS)• Keep data up to date (Amazon Athena, Amazon S3, AWS Glue)	November 22, 2021
New feature	<p>Step Functions has added Amazon VPC endpoints support for Synchronous Express Workflows. For more information, see:</p> <ul style="list-style-type: none">• Amazon VPC Endpoints for Step Functions	November 15, 2021
Update	<p>AWS Step Functions has added three new sample projects that demonstrate how to use the Step Functions AWS Batch integration. For more information, see:</p> <ul style="list-style-type: none">• Fan out an AWS Batch job• AWS Batch with Lambda• Use Step Functions and AWS Batch with error handling	October 14, 2021

Change	Description	Date changed
New feature	<p>AWS Step Functions has added AWS SDK integrations, letting you use the API actions for all of the more than two hundred AWS services. For more information, see:</p> <ul style="list-style-type: none">• AWS SDK service integrations• Gather Amazon S3 bucket info using AWS SDK service integrations	September 30, 2021
New feature	<p>AWS Step Functions has added a visual workflow designer, the AWS Step Functions Workflow Studio. For more information, see:</p> <ul style="list-style-type: none">• AWS Step Functions Workflow Studio• Learn to use the AWS Step Functions Workflow Studio	June 17, 2021
Update	<p>AWS Step Functions has added four new APIs, <code>StartBuildBatch</code>, <code>StopBuildBatch</code>, <code>RetryBuildBatch</code> and <code>DeleteBuildBatch</code>, to the CodeBuild integration. For more information, see:</p> <ul style="list-style-type: none">• Call AWS CodeBuild with Step Functions	June 4, 2021
New feature	<p>AWS Step Functions now integrates with Amazon EventBridge. For more information, see:</p> <ul style="list-style-type: none">• Call EventBridge with Step Functions• IAM policies for Step Functions and IAM policies for Amazon EventBridge• A sample project that shows how to Send a custom event to EventBridge	May 14, 2021

Change	Description	Date changed
Update	<p>AWS Step Functions has added a new sample project that shows how to use Step Functions and the Amazon Redshift Data API to run an ETL/ELT workflow. For more information, see:</p> <ul style="list-style-type: none">• Run ETL/ELT workflows using Amazon Redshift (Lambda, Amazon Redshift Data API)	April 16, 2021
New feature	<p>AWS Step Functions has a new data flow simulator in the console. For more information, see:</p> <ul style="list-style-type: none">• Step Functions console	April 8, 2021
New feature	<p>AWS Step Functions now integrates with Amazon EMR on EKS. For more information, see:</p> <ul style="list-style-type: none">• Call Amazon EMR on EKS with AWS Step Functions	March 29, 2021
Update	<p>YAML support for state machine definitions has been added to AWS Toolkit for Visual Studio Code and AWS CloudFormation. For more information, see:</p> <ul style="list-style-type: none">• Definition format support• AWS Toolkit for Visual Studio Code	March 4, 2021
New feature	<p>AWS Step Functions now integrates with AWS Glue DataBrew. For more information, see:</p> <ul style="list-style-type: none">• Manage AWS Glue DataBrew Jobs with Step Functions• What is AWS Glue DataBrew? in the DataBrew developer guide.	January 6, 2021

Change	Description	Date changed
New feature	<p>AWS Step Functions Synchronous Express Workflows are now available, giving you an easy way to orchestrate microservices. For more information, see:</p> <ul style="list-style-type: none">• Synchronous and Asynchronous Express Workflows• A sample project that shows how to Invoke Synchronous Express Workflows• The StartSyncExecution API documentation.	November 24, 2020
New feature	<p>AWS Step Functions now integrates with Amazon API Gateway. For more information, see:</p> <ul style="list-style-type: none">• Call API Gateway with Step Functions• IAM policies for Step Functions and IAM policies for Amazon API Gateway• A sample project that shows how to Make a call to API Gateway	November 17, 2020
New feature	<p>AWS Step Functions now integrates with Amazon Elastic Kubernetes Service. For more information, see:</p> <ul style="list-style-type: none">• Call Amazon EKS with Step Functions• IAM policies for Step Functions and IAM policies for Amazon EKS• A sample project that shows how to Manage an Amazon EKS cluster	November 16, 2020

Change	Description	Date changed
New feature	<p>AWS Step Functions now integrates with Amazon Athena. For more information, see:</p> <ul style="list-style-type: none">• Call Athena with Step Functions• IAM policies for Step Functions and IAM policies for Amazon Athena• A sample project that shows how to Start an Athena query	October 22, 2020
New feature	<p>AWS Step Functions now supports tracing end-to-end workflows with AWS X-Ray, giving you full visibility across state machine executions and making it easier to analyze and debug your distributed applications. For more information, see:</p> <ul style="list-style-type: none">• AWS X-Ray and Step Functions• IAM policies for Step Functions and IAM policies for AWS X-Ray• AWS Step Functions API Reference• TracingConfiguration	September 14, 2020

Change	Description	Date changed
Update	<p>AWS Step Functions now supports payload sizes up to 256 KB of data as a UTF-8 encoded string. This lets you process larger payloads in both Standard and Express workflows.</p> <p>Your existing state machines do not need to be changed in order to use the larger payloads. However, you will need to update to the latest versions of the Step Functions SDK and Local Runner to use the updated APIs. For more information, see:</p> <ul style="list-style-type: none"> • Quotas • the section called “Use Amazon S3 ARNs instead of passing large payloads” • States.DataLimitExceeded • the section called “CloudWatch Logs payloads” • the section called “EventBridge payloads” • AWS Step Functions API Reference <ul style="list-style-type: none"> • CloudWatchEventsExecutionDataDetails • HistoryEventExecutionDataDetails • GetExecutionHistory • ActivityScheduledEventDetails • ActivitySucceededEventDetails • CloudWatchEventsExecutionDataDetails • ExecutionSucceededEventDetails • LambdaFunctionScheduledEventDetails • ExecutionSucceededEventDetails • StateEnteredEventDetails • StateExitedEventDetails • TaskSubmittedEventDetails 	September 3, 2020

Change	Description	Date changed
	<ul style="list-style-type: none"><li data-bbox="509 260 938 296">• TaskSucceededEventDetails	

Change	Description	Date changed
Update	<p>The Amazon States Language has been updated as follows:</p> <ul style="list-style-type: none">• Choice Rules has added<ul style="list-style-type: none">• A null comparison operator, <code>IsNull</code>. <code>IsNull</code> tests against the JSON null value, and can be used to detect if the output of a previous state is null or not.• Four other new operators have been added, <code>IsBoolean</code>, <code>IsNumeric</code>, <code>IsString</code> and <code>IsTimestamp</code>.• A test for the existence or non-existence of a field using the <code>IsPresent</code> operator. <code>IsPresent</code> can be used to prevent <code>States.Runtime</code> errors when there is an attempt to access a non-existent key.• Wildcard pattern matching to support string comparison against patterns with one or more wildcards.• Comparison between two variables for supported comparison operators.• Timeout and heartbeat values in a Task state can now be provided dynamically from the state input instead of a fixed value using the <code>TimeoutSecondsPath</code> and <code>HeartbeatSecondsPath</code> fields. See the Task state state for more information.• The new ResultSelector field provides a way to manipulate a state's result before <code>ResultPath</code> is applied. The <code>ResultSelector</code> field is an optional field in the Map, Parallel, and Task state states.• Intrinsic functions have been added to allow basic operations without Task states. Intrinsic functions can be used within the <code>Parameters</code> and <code>ResultSelector</code> fields.	August 13, 2020

Change	Description	Date changed
Update	<p>AWS Step Functions now supports the Amazon SageMaker <code>CreateProcessingJob</code> API call. For more information, see:</p> <ul style="list-style-type: none"> • Manage SageMaker with Step Functions • Preprocess data and train a machine learning model, a sample project that demonstrates <code>CreateProcessingJob</code> . 	August 4, 2020
New feature	<p>AWS Step Functions is now supported by AWS Serverless Application Model, making it easier to integrate workflow orchestration into your serverless applications. For more information, see:</p> <ul style="list-style-type: none"> • AWS Step Functions and AWS SAM • AWS::Serverless::StateMachine • AWS SAM Policy Templates 	May 27, 2020
New feature	<p>AWS Step Functions has introduced a new synchronous invocation for nesting Step Functions executions. The new invocation, <code>arn:aws:states:::states:startExecution.sync:2</code> , returns a JSON object. The original invocation, <code>arn:aws:states:::states:startExecution.sync</code> , continues to be supported, and returns a JSON-escaped string. For more information, see:</p> <ul style="list-style-type: none"> • Manage AWS Step Functions Executions as an Integrated Service 	May 19, 2020

Change	Description	Date changed
New feature	<p>AWS Step Functions now integrates with AWS CodeBuild. For more information, see:</p> <ul style="list-style-type: none">• Using AWS Step Functions with other services• Call AWS CodeBuild with Step Functions• Optimized integrations for Step Functions	May 5, 2020
New feature	<p>Step Functions is now supported in AWS Toolkit for Visual Studio Code, making it easier to create and visualize state machine based workflows without leaving your code editor.</p>	March 31, 2020
Update	<p>You can now configure logging to Amazon CloudWatch Logs for Standard workflows. For more information, see:</p> <ul style="list-style-type: none">• Logging using CloudWatch Logs	February 25, 2020
New feature	<p>AWS Step Functions can now be accessed without requiring a public IP address, directly from Amazon Virtual Private Cloud (VPC). For more information, see:</p> <ul style="list-style-type: none">• Amazon VPC Endpoints for Step Functions	December 23, 2019

Change	Description	Date changed
New feature	<p>Express Workflows are a new workflow type, suitable for high-volume event processing workloads such as IoT data ingestion, streaming data processing and transformation, and mobile application backends.</p> <p>For more information, review the following new and updated topics.</p> <ul style="list-style-type: none">• Standard vs. Express Workflows<ul style="list-style-type: none">• Execution guarantees• Using AWS Step Functions with other services<ul style="list-style-type: none">• Optimized integrations for Step Functions• Process High-Volume Messages from Amazon SQS (Express Workflows)• Selective Checkpointing Example (Express Workflows)• Quotas<ul style="list-style-type: none">• Quotas• Logging using CloudWatch Logs• AWS Step Functions API Reference<ul style="list-style-type: none">• CreateStateMachine• UpdateStateMachine• DescribeStateMachine• DescribeStateMachineForExecution• StopExecution• DescribeExecution• GetExecutionHistory• ListExecutions• ListStateMachines• StartExecution• CloudWatchLogsLogGroup	December 3, 2019

Change	Description	Date changed
	<ul style="list-style-type: none">• LogDestination• LoggingConfiguration	
New feature	<p>AWS Step Functions now integrates with Amazon EMR. For more information, see:</p> <ul style="list-style-type: none">• Using AWS Step Functions with other services• Call Amazon EMR with Step Functions• Optimized integrations for Step Functions	November 19, 2019
Update	<p>AWS Step Functions has released the AWS Step Functions Data Science SDK. For more information, see the following.</p> <ul style="list-style-type: none">• Project on Github• SDK Documentation• The following Example Notebooks, which are available in the SageMaker console and the related GitHub project.<ul style="list-style-type: none">• <code>hello_world_workflow.ipynb</code>• <code>machine_learning_workflow_abalone.ipynb</code>• <code>training_pipeline_pytorch_mnist.ipynb</code>	November 7, 2019


Change	Description	Date changed
Update	<p>Step Functions now supports more API actions for Amazon SageMaker, and includes two new sample projects to demonstrate the functionality. For more information, see the following.</p> <ul style="list-style-type: none">• Manage SageMaker with Step Functions• Using AWS Step Functions with other services• Train a Machine Learning Model• Tune a Machine Learning Model	October 3, 2019
New feature	<p>Step Functions supports starting new workflow executions by calling <code>StartExecution</code> as an integrated service API. See:</p> <ul style="list-style-type: none">• Start Workflow Executions from a Task State• Manage AWS Step Functions Executions as an Integrated Service• Using AWS Step Functions with other services• IAM Policies for Starting Step Functions Workflow Executions	August 12, 2019

Change	Description	Date changed
New feature	<p>Step Functions includes the ability to pass a task token to integrated services, and pause the execution until that task token is returned with <code>SendTaskSuccess</code> or <code>SendTaskFailure</code>. See:</p> <ul style="list-style-type: none">• Service integration patterns• Wait for a Callback with the Task Token• Callback Pattern Example (Amazon SQS, Amazon SNS, Lambda)• Optimized integrations for Step Functions• Deploying an Example Human Approval Project• Service Integration Metrics <p>Step Functions now provides a way to access dynamic information about your current execution directly in the "Parameters" field of a state definition. See:</p> <ul style="list-style-type: none">• Context object• Pass Context Object Nodes as Parameters	May 23, 2019
New feature	<p>Step Functions supports CloudWatch Events for execution status changes, see:</p> <ul style="list-style-type: none">• EventBridge (CloudWatch Events) for Step Functions execution status changes• Amazon CloudWatch Events User Guide	May 8, 2019
New feature	<p>Step Functions supports IAM permissions using tags. For more information, see:</p> <ul style="list-style-type: none">• Tagging in Step Functions• Tag-based Policies	March 5, 2019

Change	Description	Date changed
New feature	Step Functions Local is now available. You can run Step Functions on your local machine for testing and development. Step Functions Local is available for download as either a Java application, or as a Docker image. See Testing state machines locally .	February 4, 2019
New feature	AWS Step Functions is now available in the Beijing and Ningxia regions. See Supported regions .	January 15, 2018
New feature	Step Functions supports resource tagging to help track your cost allocation. You can tag state machines on the Details page, or through API actions. See Tagging in Step Functions .	January 7, 2019
New feature	AWS Step Functions is now available in the Europe (Paris), and South America (São Paulo) regions. See Supported regions .	December 13, 2018
New feature	AWS Step Functions is now available the Europe (Stockholm) region. See Supported regions for a list of supported regions.	December 12, 2018
New feature	Step Functions now integrates with some AWS services. You can now directly call and pass parameters to the API of these integrated services from a task state in the Amazon States Language. For more information, see: <ul style="list-style-type: none">• Using AWS Step Functions with other services• Pass parameters to a service API• Optimized integrations for Step Functions	November 29, 2018
Update	Improved the description of <code>TimeoutSeconds</code> and <code>HeartbeatSeconds</code> in the documentation for task states. See Task state .	October 24, 2018

Change	Description	Date changed
Update	Improved the description for the <i>Maximum execution history size</i> limit and provided a link to the related best practices topic. <ul style="list-style-type: none">• Quotas related to state machine executions• Avoid reaching the history quota	October 17, 2018
Update	Added a new tutorial to the AWS Step Functions documentation: See Starting a State Machine Execution in Response to Amazon S3 Events .	September 25, 2018
Update	Removed the entry <i>Maximum executions displayed in Step Functions console</i> from the limits documentation. See Quotas .	September 13, 2018
Update	Added a best practices topic to the AWS Step Functions documentation on improving latency when polling for activity tasks. See Avoid latency when polling for activity tasks .	August 30, 2018
Update	Improved the AWS Step Functions topic on activities and activity workers. See Activities .	August 29, 2018
Update	Improved the AWS Step Functions topic on CloudTrail integration. See Recording API calls with AWS CloudTrail .	August 7, 2018
Update	Added JSON examples to AWS CloudFormation tutorial. See Creating a Lambda state machine for Step Functions using AWS CloudFormation .	June 23, 2018
Update	Added a new topic on handling Lambda service errors. See Handle Lambda service exceptions .	June 20, 2018
New feature	AWS Step Functions is now available the Asia Pacific (Mumbai) region. See Supported regions for a list of supported regions.	June 28, 2018


Change	Description	Date changed
New feature	AWS Step Functions is now available the AWS GovCloud (US-West) region. See Supported regions for a list of supported regions. For information about using Step Functions in the AWS GovCloud (US-West) Region, see AWS GovCloud (US) .	June 28, 2018
Update	Improved documentation on error handling for Parallel states. See Error Handling .	June 20, 2018
Update	Improved documentation about Input and Output processing in Step Functions. Learn how to use <code>InputPath</code> , <code>ResultPath</code> , and <code>OutputPath</code> to control the flow of JSON through your workflows, states, and tasks. See: <ul style="list-style-type: none">• Input and Output Processing in Step Functions• ResultPath	June 7, 2018
Update	Improved code examples for parallel states. See Parallel .	June 4, 2018
New feature	You can now monitor API and Service metrics in CloudWatch. See Monitoring Step Functions Using CloudWatch .	May 25, 2018
Update	<code>StartExecution</code> , <code>StopExecution</code> , and <code>StateTransition</code> now have increased throttling limits in the following regions: <ul style="list-style-type: none">• US East (N. Virginia)• US West (Oregon)• Europe (Ireland) For more information see Quotas .	May 16, 2018


Change	Description	Date changed
New feature	AWS Step Functions is now available the US West (N. California) and Asia Pacific (Seoul) regions. See Supported regions for a list of supported regions.	May 5, 2018
Update	Updated procedures and images to match changes to the interface.	April 25, 2018
Update	Added a new tutorial that shows how to start a new execution to continue your work. See Continuing Long-running Workflow Executions as a New Execution . This tutorial describes a design pattern that can help avoid some service limitations. See Avoid reaching the history quota .	April 19, 2018
Update	Improved introduction to states documentation by adding conceptual information about state machines. See States .	March 9, 2018
Update	<p>In addition to HTML, PDF, and Kindle, the AWS Step Functions Developer Guide is available on GitHub. To leave feedback, choose the GitHub icon in the upper right-hand corner.</p> 	March 2, 2018
Update	<p>Added a topic describing other resources relating to Step Functions.</p> <p>See Related information.</p>	February 20, 2018

Change	Description	Date changed
New feature	<ul style="list-style-type: none">• When you create a new state machine, you must acknowledge that AWS Step Functions will create an IAM role which allows access to your Lambda functions.• Updated the following tutorials to reflect the minor changes in the state machine creation workflow:<ul style="list-style-type: none">• Creating a Step Functions state machine that uses Lambda• Creating an Activity state machine using Step Functions• Handling error conditions using a Step Functions state machine• Iterate a loop with Lambda	February 19, 2018
Update	<p>Added a topic that describes an example activity worker written in Ruby. This implementation can be used to create a Ruby activity worker directly, or as a design pattern for creating an activity worker in another language.</p> <p>See Example Activity Worker in Ruby.</p>	February 6, 2018
Update	<p>Added a new tutorial describing a design pattern that uses a Lambda function to iterate a count.</p> <p>See Creating a Step Functions state machine that uses Lambda.</p>	January 31, 2018
Update	<p>Updated content on IAM permissions to include <code>DescribeStateMachineForExecution</code> and <code>UpdateStateMachine</code> APIs.</p> <p>See Creating Granular IAM Permissions for Non-Admin Users.</p>	January 26, 2018

Change	Description	Date changed
Update	Added newly available regions: Canada (Central), Asia Pacific (Singapore). See Supported regions .	January 25, 2018
Update	Updated tutorials and procedures to reflect that IAM allows you to select <i>Step Functions</i> as a role.	January 24, 2018
Update	Added a new <i>Best Practices</i> topic that suggests not passing large payloads between states. See Use Amazon S3 ARNs instead of passing large payloads .	January 23, 2018
Update	Corrected procedures to match updated interface for creating a state machine: <ul style="list-style-type: none">• Creating a Step Functions state machine that uses Lambda• Creating an Activity state machine using Step Functions• Handling error conditions using a Step Functions state machine	January 17, 2018

Change	Description	Date changed
New Feature	<p>You can use <i>Sample Projects</i> to quickly provision state machines and all related AWS resources. See Starter templates for Step Functions,</p> <p>Available sample projects include:</p> <ul style="list-style-type: none">• Poll for Job Status (Lambda, AWS Batch)• Task Timer (Lambda, Amazon SNS) <div data-bbox="477 695 1321 961"><p>Note</p><p>These sample projects and related documentation replace tutorials that described implementing the same functionality.</p></div>	January 11, 2018
Update	<p>Added a <i>Best Practices</i> section that includes information on avoiding stuck executions. See Best practices for Step Functions.</p>	January 5, 2018
Update	<p>Added a note on how retries can affect pricing:</p> <div data-bbox="477 1255 1321 1522"><p>Note</p><p>Retries are treated as state transitions. For information about how state transitions affect billing, see Step Functions Pricing.</p></div>	December 8, 2017

Change	Description	Date changed
Update	<p>Added information related to resource names:</p> <div data-bbox="477 348 1321 808" style="border: 1px solid #ccc; border-radius: 10px; padding: 10px;"><p> Note</p><p>Step Functions allows you to create names for state machines, executions, and activities, and labels that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.</p></div>	December 6, 2017
Update	Improved security overview information and added a topic on granular IAM permissions. See Security in AWS Step Functions and Creating Granular IAM Permissions for Non-Admin Users .	November 27, 2017
New Feature	You can update an existing state machine. See Update your state machine .	November 15, 2017

Change	Description	Date changed
Update	<p>Added a note to clarify <code>Lambda.Unknown</code> errors and linked to the Lambda documentation in the following sections:</p> <ul style="list-style-type: none"> • Error names • Step 3: Create a state machine with a Catch field <div data-bbox="477 617 1321 1268" style="border: 1px solid #add8e6; border-radius: 10px; padding: 10px; margin-top: 10px;"> <p> Note</p> <p>Unhandled errors in Lambda are reported as <code>Lambda.Unknown</code> in the error output. These include out-of-memory errors and function timeouts. You can match on <code>Lambda.Unknown</code> , <code>States.ALL</code> , or <code>States.TaskFailed</code> to handle these errors. When Lambda hits the maximum number of invocations, the error is <code>Lambda.TooManyRequestsException</code> . For more information about Lambda function errors, see Error handling and automatic retries in the <i>AWS Lambda Developer Guide</i>.</p> </div>	October 17, 2017
Update	Corrected and clarified IAM instructions and updated the screenshots in all tutorials .	October 11, 2017

Change	Description	Date changed
Update	<ul style="list-style-type: none">• Added new screenshots for state machine execution results to reflect changes in the Step Functions console. Rewrote the Lambda instructions in the following tutorials to reflect changes in the Lambda console:<ul style="list-style-type: none">• Creating a Step Functions state machine that uses Lambda• Creating a Job Status Poller• Creating a Task Timer• Handling error conditions using a Step Functions state machine• Corrected and clarified information about creating state machines in the following sections:<ul style="list-style-type: none">• Creating an Activity state machine using Step Functions	October 6, 2017
Update	<p>Rewrote the IAM instructions in the following sections to reflect changes in the IAM console:</p> <ul style="list-style-type: none">• Creating an IAM role for your state machine• Creating a Step Functions state machine that uses Lambda• Creating a Job Status Poller• Creating a Task Timer• Handling error conditions using a Step Functions state machine• Creating a Step Functions API using API Gateway	October 5, 2017
Update	Rewrote the State Machine Data section.	September 28, 2017
New feature	The limits related to API action throttling are increased for all regions where Step Functions is available.	September 18, 2017

Change	Description	Date changed
Update	<ul style="list-style-type: none">• Corrected and clarified information about starting new executions in all tutorials.• Corrected and clarified information in the Quotas related to accounts section.	September 14, 2017
Update	<p>Rewrote the following tutorials to reflect changes in the Lambda console:</p> <ul style="list-style-type: none">• Creating a Step Functions state machine that uses Lambda• Handling error conditions using a Step Functions state machine• Creating a Job Status Poller	August 28, 2017
New feature	Step Functions is available in Europe (London).	August 23, 2017
New feature	The visual workflows of state machines let you zoom in, zoom out, and center the graph.	August 21, 2017
New feature	<div style="border: 1px solid #f08080; border-radius: 10px; padding: 10px; margin-bottom: 10px;"><p>⚠ Important</p><p>An execution can't use the name of another execution for 90 days.</p></div> <p>When you make multiple <code>StartExecution</code> calls with the same name, the new execution doesn't run.</p> <p>For more information, see the name request parameter of the <code>StartExecution</code> API action in the <i>AWS Step Functions API Reference</i>.</p>	August 18, 2017

Change	Description	Date changed
Update	Added information about an alternative way of passing the state machine ARN to the Creating a Step Functions API using API Gateway tutorial.	August 17, 2017
Update	Added the new <i>Creating a Job Status Poller</i> tutorial.	August 10, 2017
New feature	<ul style="list-style-type: none">• Step Functions emits the ExecutionThrottled CloudWatch metric. For more information, see Monitoring Step Functions Using CloudWatch.• Added the Quotas related to state throttling section.	August 3, 2017
Update	Updated the instructions in the Step 1: Create an IAM Role for API Gateway section.	July 18, 2017
Update	Corrected and clarified information in the Choice section.	June 23, 2017
Update	Added information about using resources under other AWS accounts to the following tutorials: <ul style="list-style-type: none">• Creating a Step Functions state machine that uses Lambda• Creating a Lambda state machine for Step Functions using AWS CloudFormation• Creating an Activity state machine using Step Functions• Handling error conditions using a Step Functions state machine	June 22, 2017

Change	Description	Date changed
Update	Corrected and clarified information in the following sections: <ul style="list-style-type: none">• Handling error conditions using a Step Functions state machine• States• Error handling in Step Functions	June 21, 2017
Update	Rewrote all tutorials to match the Step Functions console refresh.	June 12, 2017
New feature	Step Functions is available in Asia Pacific (Sydney).	June 8, 2017
Update	Restructured the Amazon States Language section.	June 7, 2017
Update	Corrected and clarified information in the Creating an Activity state machine using Step Functions section.	June 6, 2017
Update	Corrected the code examples in the State machine examples using Retry and using Catch section.	June 5, 2017
Update	Restructured this guide using AWS documentation standards.	May 31, 2017
Update	Corrected and clarified information in the Parallel section.	May 25, 2017
Update	Merged the Paths and Filters sections into the Input and Output Processing in Step Functions section.	May 24, 2017
Update	Corrected and clarified information in the Monitoring Step Functions Using CloudWatch section.	May 15, 2017

Change	Description	Date changed
Update	Updated the GreeterActivities.java worker code in the Creating an Activity state machine using Step Functions tutorial.	May 9, 2017
Update	Added an introductory video to the What is AWS Step Functions? section.	April 19, 2017
Update	Corrected and clarified information in the following tutorials: <ul style="list-style-type: none"> • Creating a Step Functions state machine that uses Lambda • Creating an Activity state machine using Step Functions • Handling error conditions using a Step Functions state machine 	April 19, 2017
Update	Added information about Lambda templates to the Creating a Step Functions state machine that uses Lambda and Handling error conditions using a Step Functions state machine tutorials.	April 6, 2017
Update	Changed the "Maximum input or result data size" limit to "Maximum input or result data size for a task, state, or execution" (32,768 characters). For more information, see Quotas related to task executions .	March 31, 2017
New feature	<ul style="list-style-type: none"> • Step Functions supports executing state machines by setting Step Functions as Amazon CloudWatch Events targets. 	March 21, 2017
New feature	<ul style="list-style-type: none"> • Step Functions allows Lambda function error handling as the preferred error handling method. • Updated the Handling error conditions using a Step Functions state machine tutorial and the Error handling in Step Functions section. 	March 16, 2017

Change	Description	Date changed
New feature	Step Functions is available in Europe (Frankfurt).	March 7, 2017
Update	<p>Reorganized the topics in the table of contents and updated the following tutorials:</p> <ul style="list-style-type: none"> • Creating a Step Functions state machine that uses Lambda • Creating an Activity state machine using Step Functions • Handling error conditions using a Step Functions state machine 	February 23, 2017
New feature	<ul style="list-style-type: none"> • The State Machines page of the Step Functions console includes the Copy to New and Delete buttons. • Updated the screenshots to match the console changes. 	February 23, 2017
New feature	<ul style="list-style-type: none"> • Step Functions supports creating APIs using API Gateway. • Added the Creating a Step Functions API using API Gateway tutorial. 	February 14, 2017
New feature	<ul style="list-style-type: none"> • Step Functions supports integration with AWS CloudFormation. • Added the Creating a Lambda state machine for Step Functions using AWS CloudFormation tutorial. 	February 10, 2017
Update	Clarified the current behavior of the <code>ResultPath</code> and <code>OutputPath</code> fields in relation to <code>Parallel</code> states.	February 6, 2017
Update	<ul style="list-style-type: none"> • Clarified state machine naming restrictions in tutorials. • Corrected some code examples. 	January 5, 2017
Update	Updated Lambda function examples to use the latest programming model.	December 9, 2016

Change	Description	Date changed
Initial release	Initial release of AWS Step Functions.	December 1, 2016