aws

Cloud design patterns, architectures, and implementations

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Cloud design patterns, architectures, and implementations

Table of Contents

Cloud design patterns, architectures, and implementations

Anitha Deenadayalan, Amazon Web Services (AWS)

May 2024 (document history)

This guide provides guidance for implementing commonly used modernization design patterns by using AWS services. An increasing number of modern applications are designed by using microservices architectures to achieve scalability, improve release velocity, reduce the scope of impact for changes, and reduce regression. This leads to improved developer productivity and increased agility, better innovation, and an increased focus on business needs. Microservices architectures also support the use of the best technology for the service and the database, and promote polyglot code and polyglot persistence.

Traditionally, monolithic applications run in a single process, use one data store, and run on servers that scale vertically. In comparison, modern microservice applications are fine-grained, have independent fault domains, run as services across the network, and can use more than one data store depending on the use case. The services scale horizontally, and a single transaction might span multiple databases. Development teams must focus on network communication, polyglot persistence, horizontal scaling, eventual consistency, and transaction handling across the data stores when developing applications by using microservices architectures. Therefore, modernization patterns are critical for solving commonly occurring problems in modern application development, and they help accelerate software delivery.

This guide provides a technical reference for cloud architects, technical leads, application and business owners, and developers who want to choose the right cloud architecture for design patterns based on well-architected best practices. Each pattern discussed in this guide addresses one or more known scenarios in microservices architectures. The guide discusses the issues and considerations associated with each pattern, provides a high-level architectural implementation, and describes the AWS implementation for the pattern. Open source GitHub samples and workshop links are provided where available.

The guide covers the following patterns:

- Anti-corruption layer
- API routing patterns:

  - Hostname routing
  - Path routing
  - HTTP header routing
- Circuit breaker
- Event sourcing
- Hexagonal architecture
- Publish-subscribe
- Retry with backoff
- Saga patterns:
  - Saga choreography
  - Saga orchestration
- Scatter-gather
- Strangler fig
- Transactional outbox

Targeted business outcomes

By using the patterns discussed in this guide to modernize your applications, you can:

- Design and implement reliable, secure, operationally efficient architectures that are optimized for cost and performance.
- Reduce the cycle time for use cases that require these patterns, so you can focus on organization-specific challenges instead.
- Accelerate development by standardizing pattern implementations by using AWS services.
- Help your developers build modern applications without inheriting technical debt.

Anti-corruption layer pattern

Intent

The anti-corruption layer (ACL) pattern acts as a mediation layer that translates domain model semantics from one system to another system. It translates the model of the upstream bounded context (monolith) into a model that suits the downstream bounded context (microservice) before consuming the communication contract that's established by the upstream team. This pattern might be applicable when the downstream bounded context contains a core subdomain, or the upstream model is an unmodifiable legacy system. It also reduces transformation risk and business disruption by preventing changes to callers when their calls have to be redirected transparently to the target system.

Motivation

During the migration process, when a monolithic application is migrated into microservices, there might be changes in the domain model semantics of the newly migrated service. When the features within the monolith are required to call these microservices, the calls should be routed to the migrated service without requiring any changes to the calling services. The ACL pattern allows the monolith to call the microservices transparently by acting as an adapter or a facade layer that translates the calls into the newer semantics.

Applicability

Consider using this pattern when:

- Your existing monolithic application has to communicate with a function that has been migrated into a microservice, and the migrated service domain model and semantics differ from the original feature.
- Two systems have different semantics and need to exchange data, but it isn't practical to modify one system to be compatible with the other system.
- You want to use a quick and simplified approach to adapt one system to another with minimal impact.
- Your application is communicating with an external system.

Issues and considerations

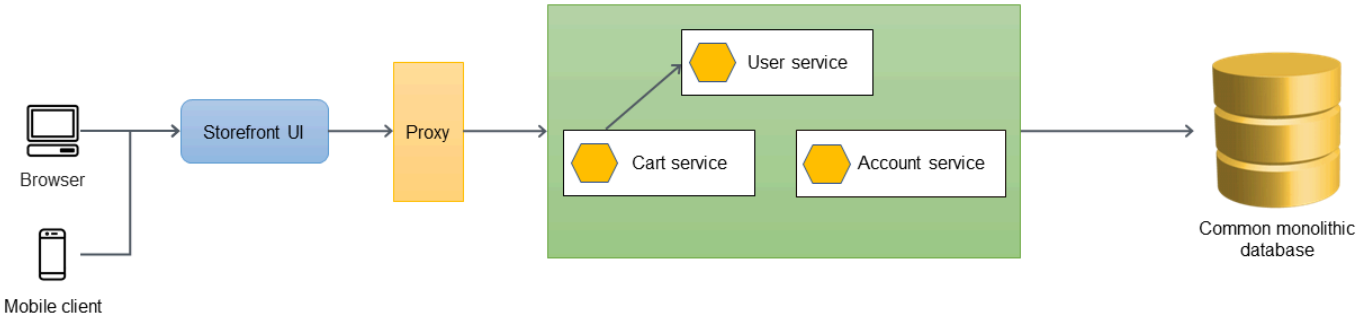
- **Team dependencies:** When different services in a system are owned by different teams, the new domain model semantics in the migrated services can lead to changes in the calling systems. However, teams might not be able to make these changes in a coordinated way, because they might have other priorities. ACL decouples the callees and translates the calls to match the semantics of the new services, thus avoiding the need for callers to make changes in the current system.
- **Operational overhead:** The ACL pattern requires additional effort to operate and maintain. This work includes integrating ACL with monitoring and alerting tools, the release process, and continuous integration and continuous delivery (CI/CD) processes.
- **Single point of failure:** Any failures in the ACL can make the target service unreachable, causing application issues. To mitigate this issue, you should build in retry capabilities and circuit breakers. See the retry with backoff and circuit breaker patterns to understand more about these options. Setting up appropriate alerts and logging will improve the mean time to resolution (MTTR).
- **Technical debt:** As part of your migration or modernization strategy, consider whether the ACL will be a transient or interim solution, or a long-term solution. If it's an interim solution, you should record the ACL as a technical debt and decommission it after all dependent callers have been migrated.
- **Latency:** The additional layer can introduce latency due to the conversion of requests from one interface to another. We recommend that you define and test performance tolerance in applications that are sensitive to response time before you deploy ACL into production environments.
- **Scaling bottleneck:** In high-load applications where services can scale to peak load, ACL can become a bottleneck and might cause scaling issues. If the target service scales on demand, you should design ACL to scale accordingly.
- **Service-specific or shared implementation:** You can design ACL as a shared object to convert and redirect calls to multiple services or service-specific classes. Take latency, scaling, and failure tolerance into account when you determine the implementation type for ACL.

Implementation

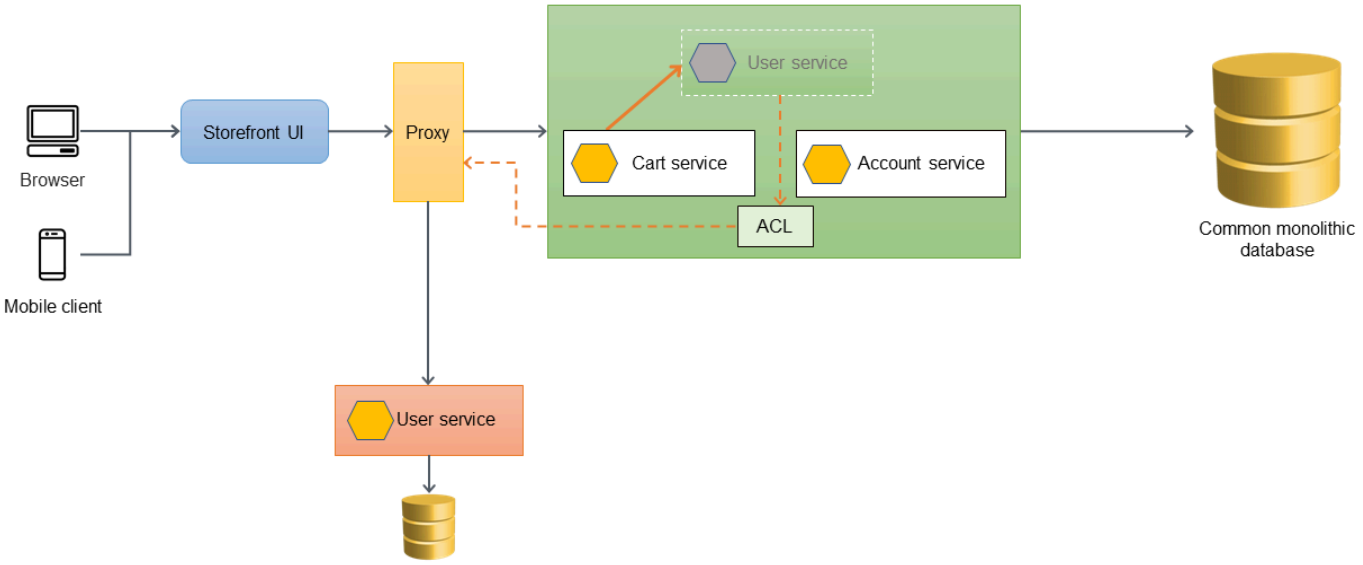
You can implement ACL inside your monolithic application as a class that's specific to the service that's being migrated, or as an independent service. The ACL must be decommissioned after all dependent services have been migrated into the microservices architecture.

High-level architecture

In the following example architecture, a monolithic application has three services: user service, cart service, and account service. The cart service is dependent on the user service, and the application uses a monolithic relational database.

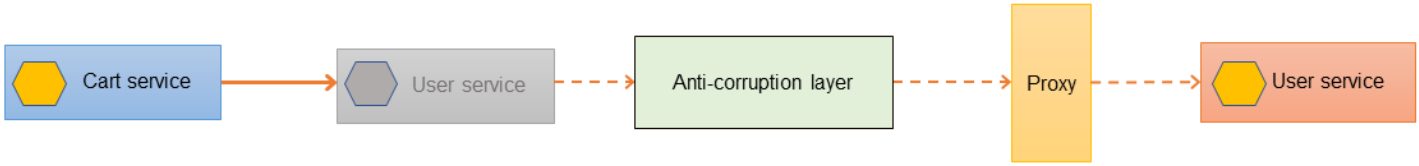


In the following architecture, the user service has been migrated to a new microservice. The cart service calls the user service, but the implementation is no longer available within the monolith. It's also likely that the interface of the newly migrated service won't match its previous interface, when it was inside the monolithic application.



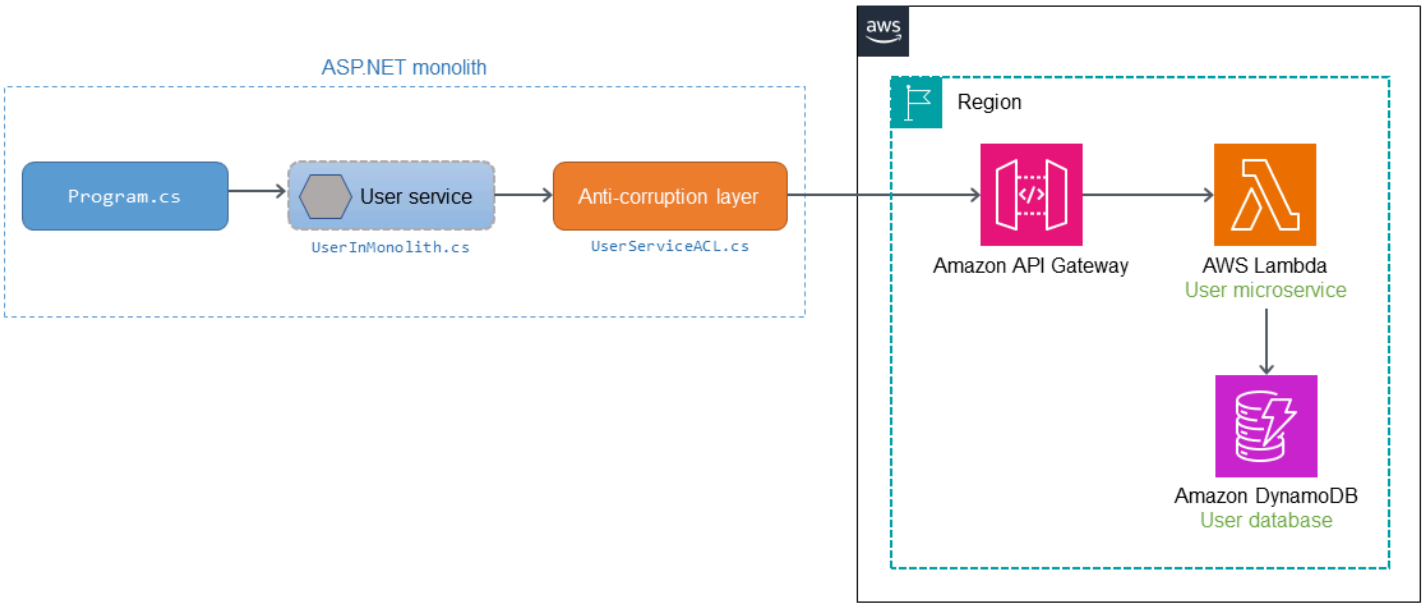
If the cart service has to call the newly migrated user service directly, this will require changes to the cart service and a thorough testing of the monolithic application. This can increase the transformation risk and business disruption. The goal should be to minimize the changes to the existing functionality of the monolithic application.

In this case, we recommend that you introduce an ACL between the old user service and the newly migrated user service. The ACL works as an adapter or a facade that converts the calls into the newer interface. ACL can be implemented inside the monolithic application as a class (for example, `UserServiceFacade` or `UserServiceAdapter`) that's specific to the service that was migrated. The anti-corruption layer must be decommissioned after all dependent services have been migrated into the microservices architecture.



Implementation using AWS services

The following diagram shows how you can implement this ACL example by using AWS services.



The user microservice is migrated out of the ASP.NET monolithic application and deployed as an AWS Lambda function on AWS. Calls to the Lambda function are routed through Amazon API Gateway. ACL is deployed in the monolith to translate the call to adapt to the semantics of the user microservice.

When `Program.cs` calls the user service (`UserInMonolith.cs`) inside the monolith, the call is routed to the ACL (`UserServiceACL.cs`). The ACL translates the call to the new semantics and interface, and calls the microservice through the API Gateway endpoint. The caller (`Program.cs`) isn't aware of the translation and routing that take place in the user service and ACL. Because the caller isn't aware of the code changes, there is less business disruption and reduced transformation risk.

Sample code

The following code snippet provides the changes to the original service and the implementation of `UserServiceACL.cs`. When a request is received, the original user service calls the ACL. The ACL converts the source object to match the interface of the newly migrated service, calls the service, and returns the response to the caller.

```
public class UserInMonolith: IUserInMonolith
{
    private readonly IACL _userServiceACL;
    public UserInMonolith(IACL userServiceACL) => (_userServiceACL) = (userServiceACL);
    public async Task<HttpStatusCode> UpdateAddress(UserDetails userDetails)
    {
        //Wrap the original object in the derived class
        var destUserDetails = new UserDetailsWrapped("user", userDetails);
        //Logic for updating address has been moved to a microservice
        return await _userServiceACL.CallMicroservice(destUserDetails);
    }
}

public class UserServiceACL: IACL
{
    static HttpClient _client = new HttpClient();
    private static string _apiGatewayDev = string.Empty;

    public UserServiceACL()
    {
        IConfiguration config = new
 ConfigurationBuilder().AddJsonFile(AppContext.BaseDirectory + "../../../
config.json").Build();
        _apiGatewayDev = config["APIGatewayURL:Dev"];
        _client.DefaultRequestHeaders.Accept.Add(new
 MediaTypeWithQualityHeaderValue("application/json"));
    }
    public async Task<HttpStatusCode> CallMicroservice(ISourceObject details)
```

```
    {
        _apiGatewayDev +=  "/" + details.ServiceName;
        Console.WriteLine(_apiGatewayDev);

        var userDetails = details as UserDetails;
        var userMicroserviceModel = new UserMicroserviceModel();
        userMicroserviceModel.UserId = userDetails.UserId;
        userMicroserviceModel.Address = userDetails.AddressLine1 + ", " +
 userDetails.AddressLine2;
        userMicroserviceModel.City = userDetails.City;
        userMicroserviceModel.State = userDetails.State;
        userMicroserviceModel.Country = userDetails.Country;

        if (Int32.TryParse(userDetails.ZipCode, out int zipCode))
        {
            userMicroserviceModel.ZipCode = zipCode;
            Console.WriteLine("Updated zip code");
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
            return HttpStatusCode.BadRequest;
        }

        var jsonString =
 JsonSerializer.Serialize<UserMicroserviceModel>(userMicroserviceModel);
        var payload = JsonSerializer.Serialize(userMicroserviceModel);
        var content = new StringContent(payload, Encoding.UTF8, "application/json");

        var response = await _client.PostAsync(_apiGatewayDev, content);
        return response.StatusCode;
    }
}
```

GitHub repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/anti-corruption-layer-pattern.

Related content

- Strangler fig pattern

- Circuit breaker pattern
- Retry with backoff pattern

API routing patterns

In agile development environments, autonomous teams (for example squads and tribes) own one or more services that include many microservices. The teams expose these services as APIs to allow their consumers to interact with their group of services and actions.

There are three major methods for exposing HTTP APIs to upstream consumers by using hostnames and paths:

Method	Description	Example
Hostname routing	Expose each service as a hostname.	`billing.api.example.com`
Path routing	Expose each service as a path.	`api.example.com/billing`
Header-based routing	Expose each service as an HTTP header.	`x-example-action: something`

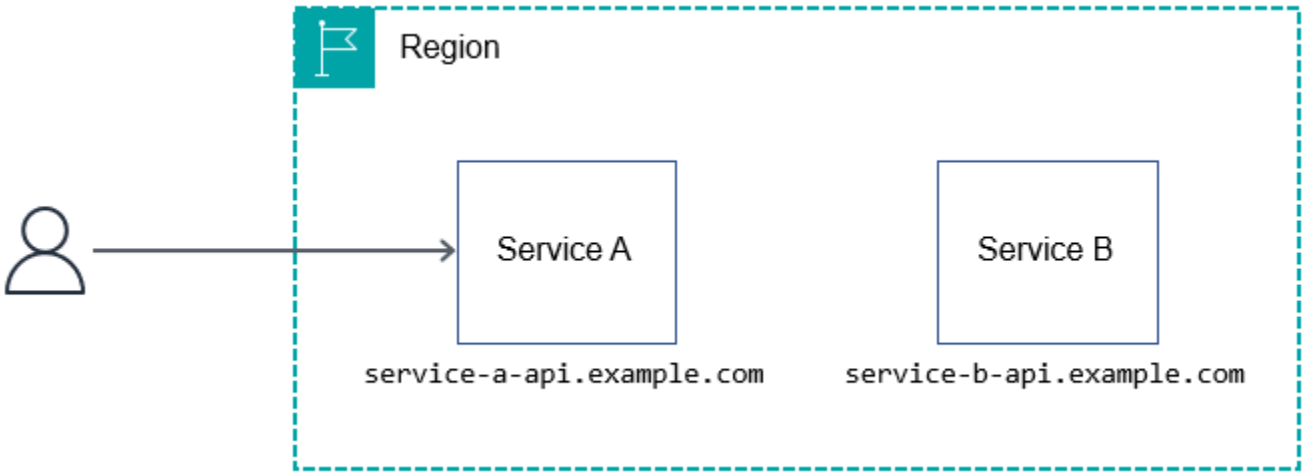
This section outlines typical use cases for these three routing methods and their trade-offs to help you decide which method best fits your requirements and organizational structure.

Hostname routing pattern

Routing by hostname is a mechanism for isolating API services by giving each API its own hostname; for example, `service-a.api.example.com` or `service-a.example.com`.

Typical use case

Routing by using hostnames reduces the amount of friction in releases, because nothing is shared between service teams. Teams are responsible for managing everything from DNS entries to service operations in production.



Pros

Hostname routing is by far the most straightforward and scalable method for HTTP API routing. You can use any relevant AWS service to build an architecture that follows this method—you can create an architecture with Amazon API Gateway, AWS AppSync, Application Load Balancers and Amazon Elastic Compute Cloud (Amazon EC2), or any other HTTP-compliant service.

Teams can use hostname routing to fully own their subdomain. It also makes it easier to isolate, test, and orchestrate deployments for specific AWS Regions or versions; for example, `region.service-a.api.example.com` or `dev.region.service-a.api.example.com`.

Cons

When you use hostname routing, your consumers have to remember different hostnames to interact with each API that you expose. You can mitigate this issue by providing a client SDK. However, client SDKs come with their own set of challenges. For example, they have to support rolling updates, multiple languages, versioning, communicating breaking changes caused by security issues or bug fixes, documentation, and so on.

When you use hostname routing, you also need to register the subdomain or domain every time you create a new service.

Path routing pattern

Routing by paths is the mechanism of grouping multiple or all APIs under the same hostname, and using a request URI to isolate services; for example, `api.example.com/service-a` or `api.example.com/service-b`.

Typical use case

Most teams opt for this method because they want a simple architecture—a developer has to remember only one URL such as `api.example.com` to interact with the HTTP API. API documentation is often easier to digest because it is often kept together instead of being split across different portals or PDFs.

Path-based routing is considered a simple mechanism for sharing an HTTP API. However, it involves operational overhead such as configuration, authorization, integrations, and additional latency due to multiple hops. It also requires mature change management processes to ensure that a misconfiguration doesn't disrupt all services.

On AWS, there are multiple ways to share an API and route effectively to the correct service. The following sections discuss three approaches: HTTP service reverse proxy, API Gateway, and Amazon CloudFront. None of the suggested approaches for unifying API services relies on the downstream services running on AWS. The services could run anywhere without issue or on any technology, as long as they're HTTP-compatible.

HTTP service reverse proxy

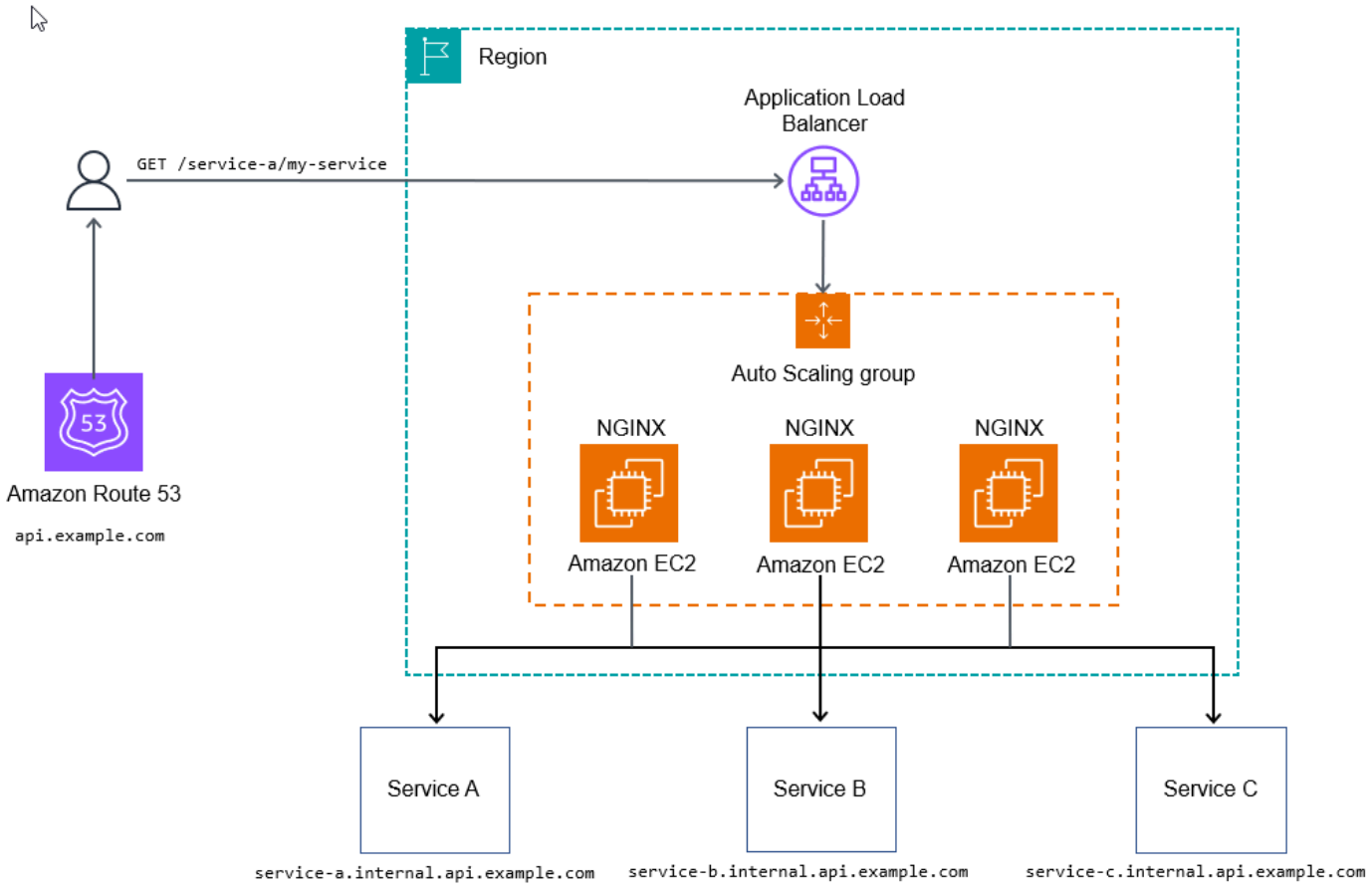
You can use an HTTP server such as NGINX to create dynamic routing configurations. In a Kubernetes architecture, you can also create an ingress rule to match a path to a service. (This guide doesn't cover Kubernetes ingress; see the Kubernetes documentation for more information.)

The following configuration for NGINX dynamically maps an HTTP request of `api.example.com/my-service/` to `my-service.internal.api.example.com`.

```
server {
    listen  80;

    location (^/[\w-]+)/(.*) {
        proxy_pass $scheme://$1.internal.api.example.com/$2;
    }
}
```

The following diagram illustrates the HTTP service reverse proxy method.



This approach might be sufficient for some use cases that don't use additional configurations to start processing requests, allowing for the downstream API to collect metrics and logs.

To get ready for operational production readiness, you will want to be able to add observability to every level of your stack, add additional configuration, or add scripts to customize your API ingress point to allow for more advanced features such as rate limiting or usage tokens.

Pros

The ultimate aim of the HTTP service reverse proxy method is to create a scalable and manageable approach to unifying APIs into a single domain so it appears coherent to any API consumer. This approach also enables your service teams to deploy and manage their own APIs, with minimal overhead after deployment. AWS managed services for tracing, such as AWS X-Ray or AWS WAF, are still applicable here.

Cons

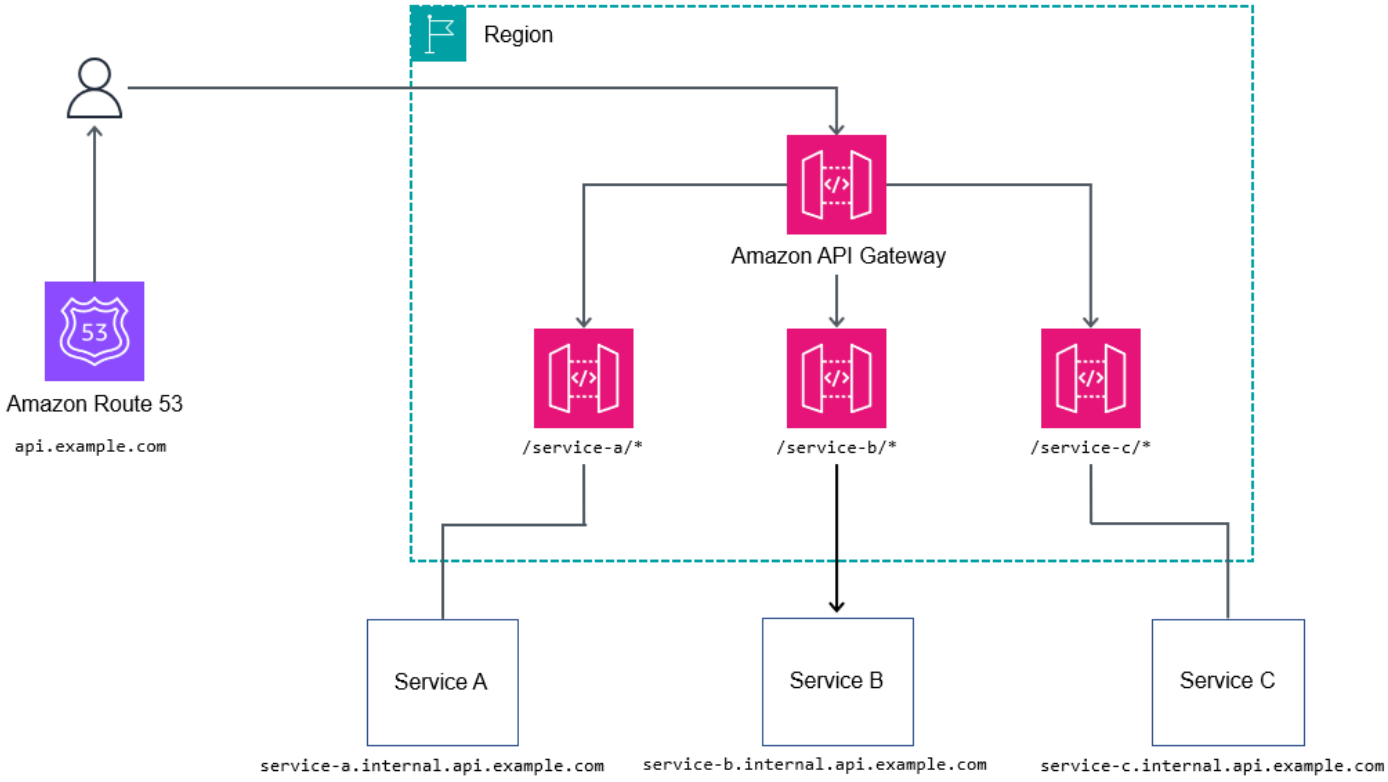
The major downside of this approach is the extensive testing and management of infrastructure components that are required, although this might not be an issue if you have site reliability engineering (SRE) teams in place.

There is a cost tipping point with this method. At low to medium volumes, it is more expensive than some of the other methods discussed in this guide. At high volumes, it is very cost-effective (around 100K transactions per second or better).

API Gateway

The Amazon API Gateway service (REST APIs and HTTP APIs) can route traffic in a way that's similar to the HTTP service reverse proxy method. Using an API gateway in HTTP proxy mode provides a simple way to wrap many services into an entry point to the top-level subdomain `api.example.com`, and then proxy requests to the nested service; for example, `billing.internal.api.example.com`.

You probably don't want to get too granular by mapping every path in every service in the root or core API gateway. Instead, opt for wildcard paths such as `/billing/*` to forward requests to the billing service. By not mapping every path in the root or core API gateway, you gain more flexibility over your APIs, because you don't have to update the root API gateway with every API change.



Pros

For control over more complex workflows, such as changing request attributes, REST APIs expose the Apache Velocity Template Language (VTL) to allow you to modify the request and response. REST APIs can provide additional benefits such as these:

- Auth N/Z with AWS Identity and Access Management (IAM), Amazon Cognito, or AWS Lambda authorizers
- AWS X-Ray for tracing
- Integration with AWS WAF
- Basic rate limiting
- Usage tokens for bucketing consumers into different tiers (see Throttle API requests for better throughput in the API Gateway documentation)

Cons

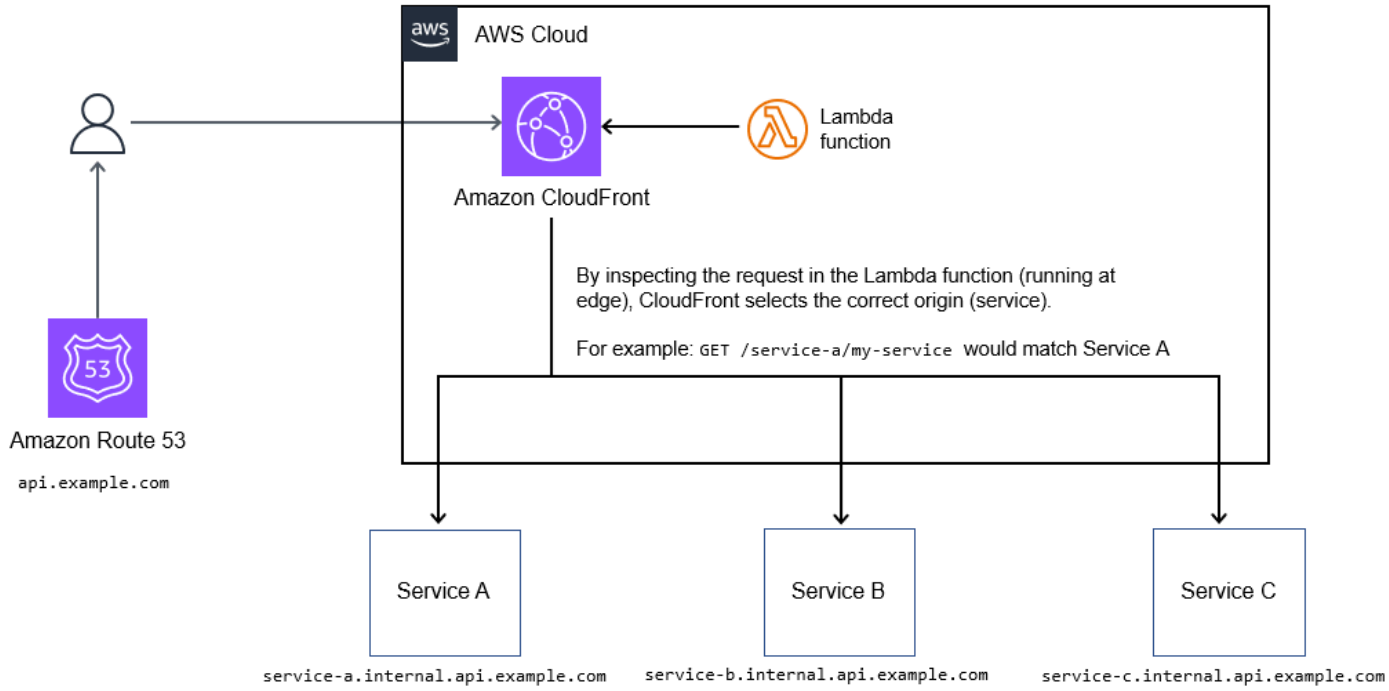
At high volumes, cost might be an issue for some users.

CloudFront

You can use the dynamic origin selection feature in Amazon CloudFront to conditionally select an origin (a service) to forward the request. You can use this feature to route a number of services through a single hostname such as `api.example.com`.

Typical use case

The routing logic lives as code within the Lambda@Edge function, so it supports highly customizable routing mechanisms such as A/B testing, canary releases, feature flagging, and path rewriting. This is illustrated in the following diagram.



Pros

If you require caching API responses, this method is good way to unify a collection of services behind a single endpoint. It is a cost-effective method to unify collections of APIs.

Also, CloudFront supports field-level encryption as well as integration with AWS WAF for basic rate limiting and basic ACLs.

Cons

This method supports a maximum of 250 origins (services) that can be unified. This limit is sufficient for most deployments, but it might cause issues with a large number of APIs as you grow your portfolio of services.

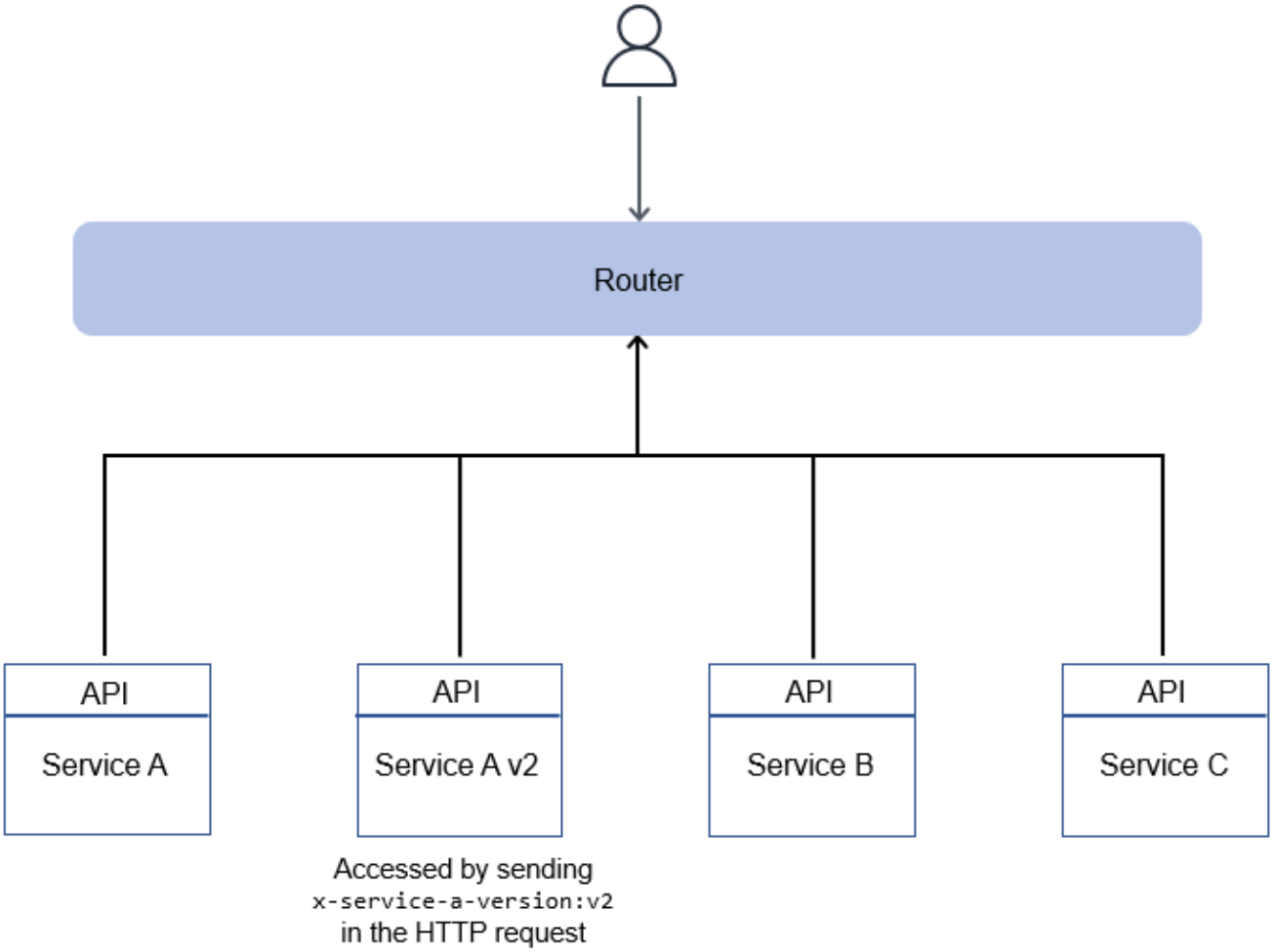
Updating Lambda@Edge functions currently takes a few minutes. CloudFront also takes up to 30 minutes to complete propagating changes to all points of presence. This ultimately blocks further updates until they complete.

HTTP header routing pattern

Header-based routing enables you to target the correct service for each request by specifying an HTTP header in the HTTP request. For example, sending the header `x-service-a-action: get-thing` would enable you to `get thing` from `Service A`. The path of the request is still important, because it offers guidance on which resource you're trying to work on.

In addition to using HTTP header routing for actions, you can use it as a mechanism for version routing, enabling feature flags, A/B tests, or similar needs. In reality, you will likely use header routing with one of the other routing methods to create robust APIs.

The architecture for HTTP header routing typically has a thin routing layer in front of microservices that routes to the correct service and returns a response, as illustrated in the following diagram. This routing layer could cover all services or just a few services to enable an operation such as version-based routing.



Pros

Configuration changes require minimal effort and can be automated easily. This method is also flexible and supports creative ways to expose only specific operations you would want from a service.

Cons

As with the hostname routing method, HTTP header routing assumes that you have full control over the client and can manipulate custom HTTP headers. Proxies, content delivery networks (CDNs), and load balancers can limit the header size. Although this is unlikely to be a concern, it could be an issue depending on how many headers and cookies you add.

Circuit breaker pattern

Intent

The circuit breaker pattern can prevent a caller service from retrying a call to another service (*callee*) when the call has previously caused repeated timeouts or failures. The pattern is also used to detect when the callee service is functional again.

Motivation

When multiple microservices collaborate to handle requests, one or more services might become unavailable or exhibit high latency. When complex applications use microservices, an outage in one microservice can lead to application failure. Microservices communicate through remote procedure calls, and transient errors could occur in network connectivity, causing failures. (The transient errors can be handled by using the retry with backoff pattern.) During synchronous execution, the cascading of timeouts or failures can cause a poor user experience.

However, in some situations, the failures could take longer to resolve—for example, when the callee service is down or a database contention results in timeouts. In such cases, if the calling service retries the calls repeatedly, these retries might result in network contention and database thread pool consumption. Additionally, if multiple users are retrying the application repeatedly, this will make the problem worse and can cause performance degradation in the entire application.

The circuit breaker pattern was popularized by Michael Nygard in his book, *Release It* (Nygard 2018). This design pattern can prevent a caller service from retrying a service call that has previously caused repeated timeouts or failures. It can also detect when the callee service is functional again.

Circuit breaker objects work like electrical circuit breakers that automatically interrupt the current when there is an abnormality in the circuit. Electrical circuit breakers shut off, or trip, the flow of the current when there is a fault. Similarly, the circuit breaker object is situated between the caller and the callee service, and trips if the callee is unavailable.

The fallacies of distributed computing are a set of assertions made by Peter Deutsch and others at Sun Microsystems. They say that programmers who are new to distributed applications invariably make false assumptions. The network reliability, zero-latency expectations, and bandwidth limitations result in software applications written with minimal error handling for network errors.

During a network outage, applications might indefinitely wait for a reply and continually consume application resources. Failure to retry the operations when the network becomes available can also lead to application degradation. If API calls to a database or an external service time out because of network issues, repeated calls with no circuit breaker can affect cost and performance.

Applicability

Use this pattern when:

- The caller service makes a call that is most likely going to fail.
- A high latency exhibited by the callee service (for example, when database connections are slow) causes timeouts to the callee service.
- The caller service makes a synchronous call, but the callee service isn't available or exhibits high latency.

Issues and considerations

- **Service agnostic implementation:** To prevent code bloat, we recommend that you implement the circuit breaker object in a microservice-agnostic and API-driven way.
- **Circuit closure by callee:** When the callee recovers from the performance issue or failure, they can update the circuit status to `CLOSED`. This is an extension of the circuit breaker pattern and can be implemented if your recovery time objective (RTO) requires it.
- **Multithreaded calls:** The expiration timeout value is defined as the period of time the circuit remains tripped before calls are routed again to check for service availability. When the callee service is called in multiple threads, the first call that failed defines the expiration timeout value. Your implementation should ensure that subsequent calls do not move the expiration timeout endlessly.
- **Force open or close the circuit:** System administrators should have the ability to open or close a circuit. This can be done by updating the expiration timeout value in the database table.
- **Observability:** The application should have logging set up to identify the calls that fail when the circuit breaker is open.

Implementation

High-level architecture

In the following example, the caller is the order service and the callee is the payment service.

When there are no failures, the order service routes all calls to the payment service by the circuit breaker, as the following diagram shows.

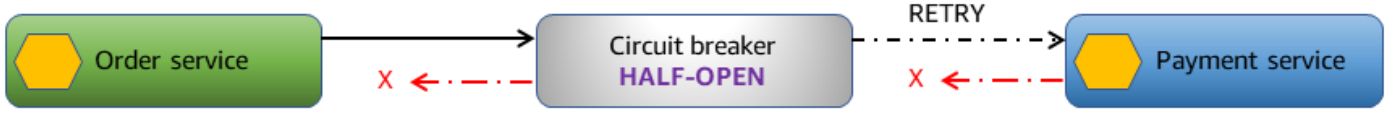
If the payment service times out, the circuit breaker can detect the timeout and track the failure.

Circuit breaker with payment service failure

If the timeouts exceed a specified threshold, the application opens the circuit. When the circuit is open, the circuit breaker object doesn't route the calls to the payment service. It returns an immediate failure when the order service calls the payment service.

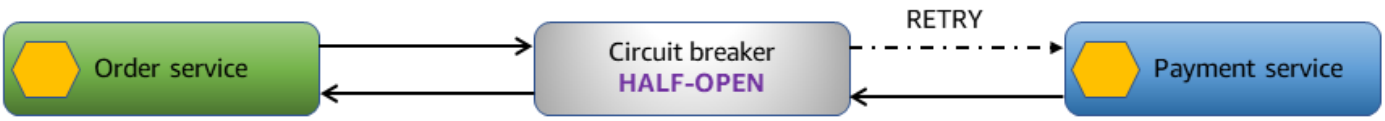
Circuit breaker stops routing to payment service

The circuit breaker object periodically tries to see if the calls to the payment service are successful.



Circuit breaker periodically retries payment service

When the call to payment service succeeds, the circuit is closed, and all further calls are routed to the payment service again.



Circuit breaker with working payment service

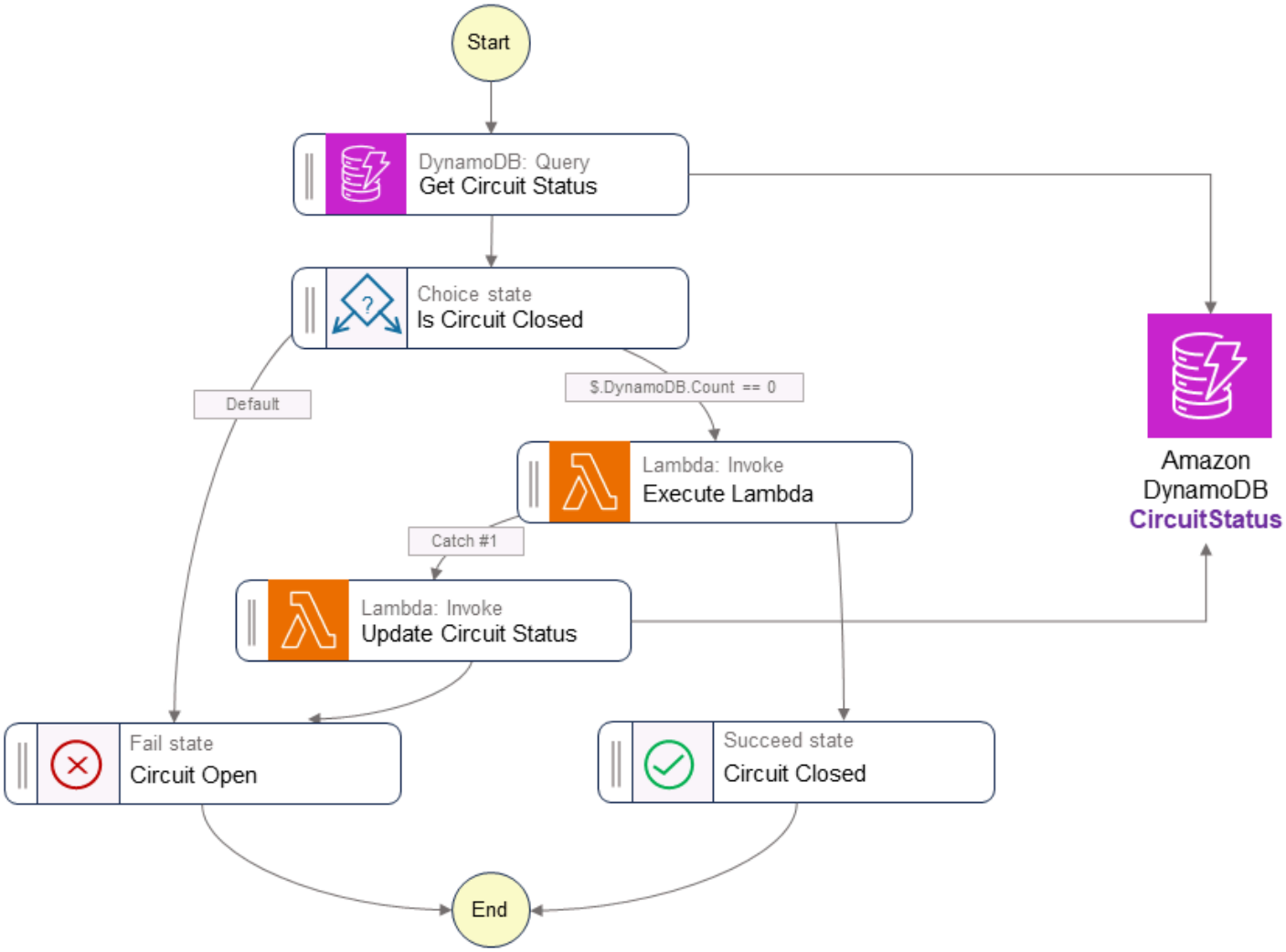
Implementation using AWS services

The sample solution uses express workflows in AWS Step Functions to implement the circuit breaker pattern. The Step Functions state machine lets you configure the retry capabilities and decision-based control flow required for the pattern implementation.

The solution also uses an Amazon DynamoDB table as the data store to track the circuit status. This can be replaced with an in-memory datastore such as Amazon ElastiCache for Redis for better performance.

When a service wants to call another service, it starts the workflow with the name of the callee service. The workflow gets the circuit breaker status from the DynamoDB `CircuitStatus` table, which stores the currently degraded services. If `CircuitStatus` contains an unexpired record for the callee, the circuit is open. The Step Functions workflow returns an immediate failure and exits with a `FAIL` state.

If the `CircuitStatus` table doesn't contain a record for the callee or contains an expired record, the service is operational. The `ExecuteLambda` step in the state machine definition calls the Lambda function that's sent through a parameter value. If the call succeeds, the Step Functions workflow exits with a SUCCESS state.



If the service call fails or a timeout occurs, the application retries with exponential backoff for a defined number of times. If the service call fails after the retries, the workflow inserts a record in the `CircuitStatus` table for the service with the an `ExpiryTimeStamp`, and the workflow exits with a `FAIL` state. Subsequent calls to the same service return an immediate failure as long as the circuit breaker is open. The `Get Circuit Status` step in the state machine definition checks the service availability based on the `ExpiryTimeStamp` value. Expired items are deleted from the `CircuitStatus` table by using the DynamoDB time to live (TTL) feature.

Sample code

The following code uses the `GetCircuitStatus` Lambda function to check the circuit breaker status.

```
var serviceDetails = _dbContext.QueryAsync<CircuitBreaker>(serviceName,
 QueryOperator.GreaterThan,
                new List<object>
                    {currentTimeStamp}).GetRemainingAsync();

if (serviceDetails.Result.Count > 0)
{
    functionData.CircuitStatus = serviceDetails.Result[0].CircuitStatus;
}
else
{
    functionData.CircuitStatus = "";
}
```

The following code shows the Amazon States Language statements in the Step Functions workflow.

```
"Is Circuit Closed": {
    "Type": "Choice",
    "Choices": [
    {
        "Variable": "$.CircuitStatus",
        "StringEquals": "OPEN",
        "Next": "Circuit Open"
    },
    {
        "Variable": "$.CircuitStatus",
        "StringEquals": "",
        "Next": "Execute Lambda"
    }
    ]
},
"Circuit Open": {
    "Type": "Fail"
}
```

GitHub repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/circuit-breaker-netcore-blog.

Blog references

- Using the circuit breaker pattern with AWS Step Functions and Amazon DynamoDB

Related content

- Strangler fig pattern
- Retry with backoff pattern
- AWS App Mesh circuit breaker capabilities

Event sourcing pattern

Intent

In event-driven architectures, the event sourcing pattern stores the events that result in a state change in a data store. This helps to capture and maintain a complete history of state changes, and promotes auditability, traceability, and the ability to analyze past states.

Motivation

Multiple microservices can collaborate to handle requests, and they communicate through events. These events can result in a change in state (data). Storing event objects in the order in which they occur provides valuable information on the current state of the data entity and additional information about how it arrived at that state.

Applicability

Use the event sourcing pattern when:

- An immutable history of the events that occur in an application is required for tracking.
- Polyglot data projections are required from a single source of truth (SSOT).
- Point-in time reconstruction of the application state is needed.
- Long-term storage of application state isn't required, but you might want to reconstruct it as needed.
- Workloads have different read and write volumes. For example, you have write-intensive workloads that don't require real-time processing.
- Change data capture (CDC) is required to analyze the application performance and other metrics.
- Audit data is required for all events that happen in a system for reporting and compliance purposes.
- You want to derive what-if scenarios by changing (inserting, updating, or deleting) events during the replay process to determine the possible end state.

Issues and considerations

- **Optimistic concurrency control:** This pattern stores every event that causes a state change in the system. Multiple users or services can try to update the same piece of data at the same time, causing event collisions. These collisions happen when conflicting events are created and applied at the same time, which results in a final data state that doesn't match reality. To solve this issue, you can implement strategies to detect and resolve event collisions. For example, you can implement an optimistic concurrency control scheme by including versioning or by adding timestamps to events to track the order of updates.
- **Complexity:** Implementing event sourcing necessitates a shift in mindset from traditional CRUD operations to event-driven thinking. The replay process, which is used to restore the system to its original state, can be complex in order to ensure data idempotency. Event storage, backups, and snapshots can also add additional complexity.
- **Eventual consistency**: Data projections derived from the events are eventually consistent because of the latency in updating data by using the command query responsibility segregation (CQRS) pattern or materialized views. When consumers process data from an event store and publishers send new data, the data projection or the application object might not represent the current state.
- **Querying**: Retrieving current or aggregate data from event logs can be more intricate and slower compared to traditional databases, particularly for complex queries and reporting tasks. To mitigate this issue, event sourcing is often implemented with the CQRS pattern.
- **Size and cost of the event store:** The event store can experience exponential growth in size as events are continuously persisted, especially in systems that have high event throughput or extended retention periods. Consequently, you must periodically archive event data to cost-effective storage to prevent the event store from becoming too large.
- **Scalability of the event store:** The event store must efficiently handle high volumes of both write and read operations. Scaling an event store can be challenging, so it's important to have a data store that provides shards and partitions.
- **Efficiency and optimization**: Choose or design an event store that handles both write and read operations efficiently. The event store should be optimized for the expected event volume and query patterns for the application. Implementing indexing and query mechanisms can speed up the retrieval of events when reconstructing the application state. You can also consider using specialized event store databases or libraries that offer query optimization features.
- **Snapshots:** You must back up event logs at regular intervals with time-based activation. Replaying the events on the last known successful backup of the data should lead to point-

in-time recovery of the application state. The recovery point objective (RPO) is the maximum acceptable amount of time since the last data recovery point. RPO determines what is considered an acceptable loss of data between the last recovery point and the interruption of service. The frequency of the daily snapshots of the data and event store should be based on the RPO for the application.

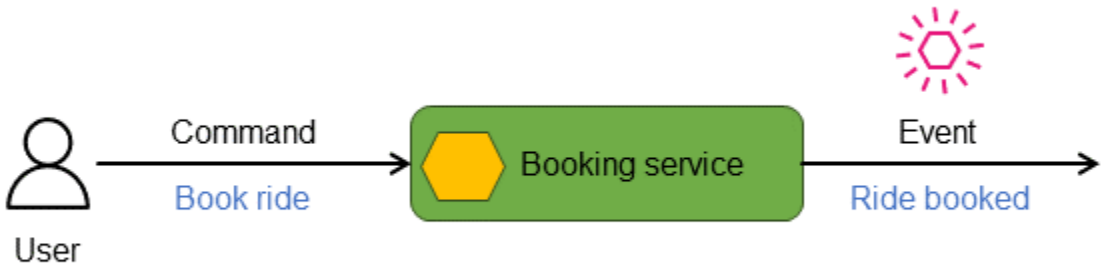
- **Time sensitivity:** The events are stored in the order in which they occur. Therefore, network reliability is an important factor to consider when you implement this pattern. Latency issues can lead to an incorrect system state. Use first in, first out (FIFO) queues with at-most-once delivery to carry the events to the event store.
- **Event replay performance**: Replaying a substantial number of events to reconstruct the current application state can be time-consuming. Optimization efforts are required to enhance performance, particularly when replaying events from archived data.
- **External system updates:** Applications that use the event sourcing pattern might update data stores in external systems, and might capture these updates as event objects. During event replays, this might become an issue if the external system doesn't expect an update. In such cases, you can use feature flags to control external system updates.
- **External system queries:** When external system calls are sensitive to the date and time of the call, the received data can be stored in internal data stores for use during replays.
- **Event versioning:** As the application evolves, the structure of the events (schema) can change. Implementing a versioning strategy for events to ensure backward and forward compatibility is necessary. This can involve including a version field in the event payload and handling different event versions appropriately during replay.

Implementation

High-level architecture

Commands and events

In distributed, event-driven microservices applications, commands represent the instructions or requests sent to a service, typically with the intent of initiating a change in its state. The service processes these commands and evaluates the command's validity and applicability to its current state. If the command runs successfully, the service responds by emitting an event that signifies the action taken and the relevant state information. For example, in the following diagram, the booking service responds to the Book ride command by emitting the Ride booked event.



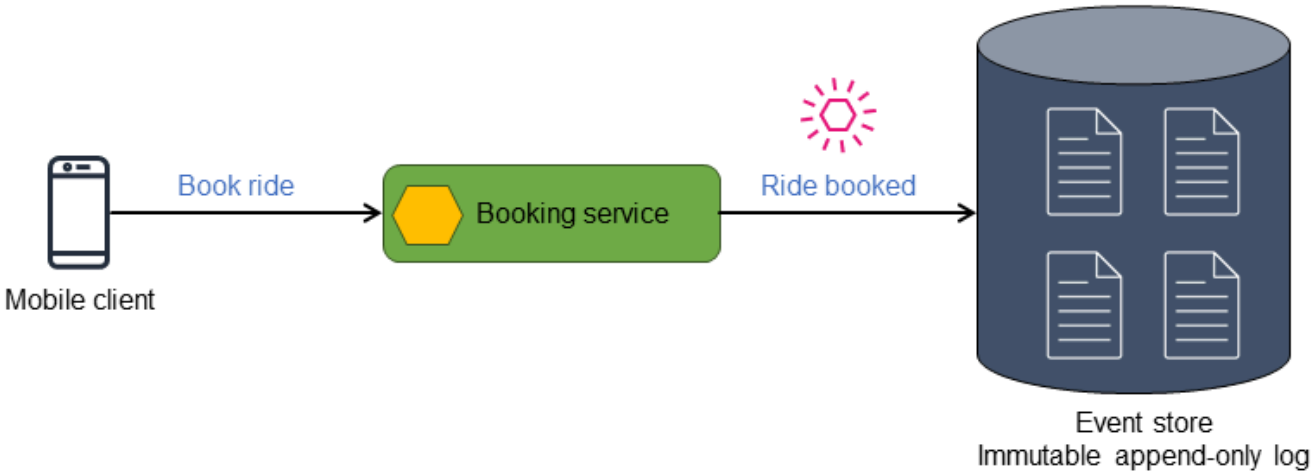
Event stores

Events are logged into an immutable, append-only, chronologically ordered repository or data store known as the *event store*. Each state change is treated as an individual event object. An entity object or a data store with a known initial state, its current state, and any point-in-time view can be reconstructed by replaying the events in the order of their occurrence.

The event store acts as a historical record of all actions and state changes, and serves as a valuable single source of truth. You can use the event store to derive the final, up-to-date state of the system by passing the events through a replay processor, which applies these events to produce an accurate representation of the latest system state. You can also use the event store to generate the point-in-time perspective of the state by replaying the events through a replay processor. In the event sourcing pattern, the current state might not be entirely represented by the most recent event object. You can derive the current state in one of three ways:

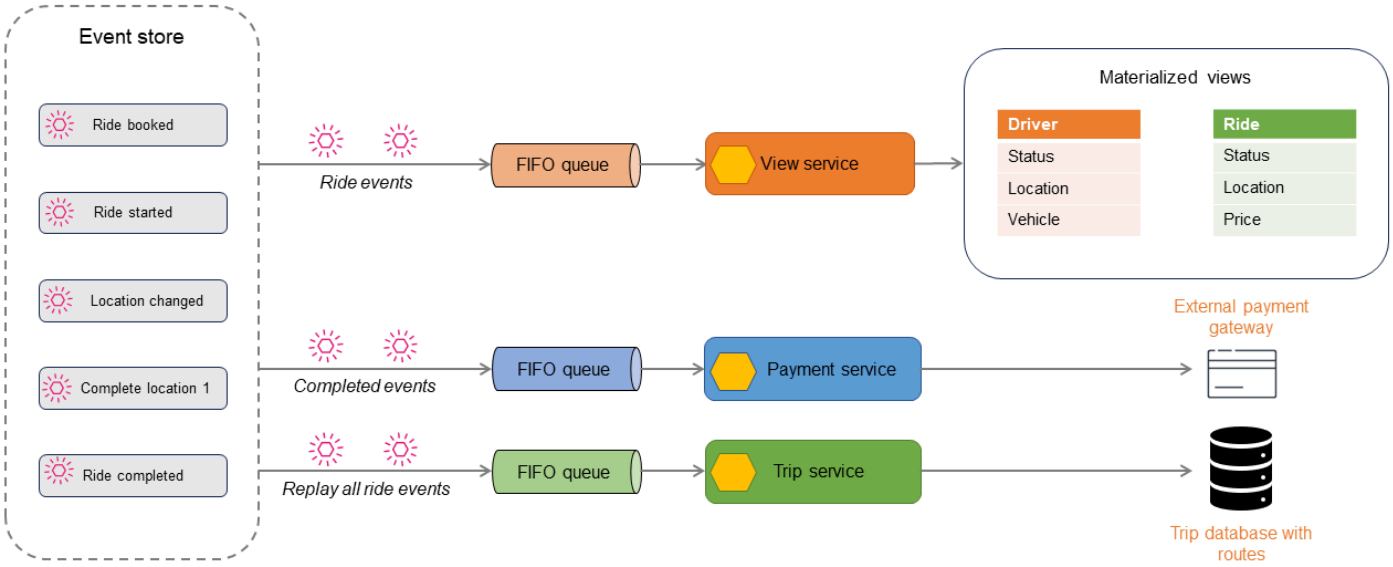
- By aggregating related events. The related event objects are combined to generate the current state for querying. This approach is often used in conjunction with the CQRS pattern, in that the events are combined and written into the read-only data store.
- By using materialized views. You can employ event sourcing with the materialized view pattern to compute or summarize the event data and obtain the current state of related data.
- By replaying events. Event objects can be replayed to carry out actions for generating the current state.

The following diagram shows the `Ride booked` event being stored in an event store.



The event store publishes the events it stores, and the events can be filtered and routed to the appropriate processor for subsequent actions. For example, events can be routed to a view processor that summarizes the state and shows a materialized view. The events are transformed to the data format of the target data store. This architecture can be extended to derive different types of data stores, which leads to polyglot persistence of the data.

The following diagram describes the events in a ride booking application. All the events that occur within the application are stored in the event store. Stored events are then filtered and routed to different consumers.



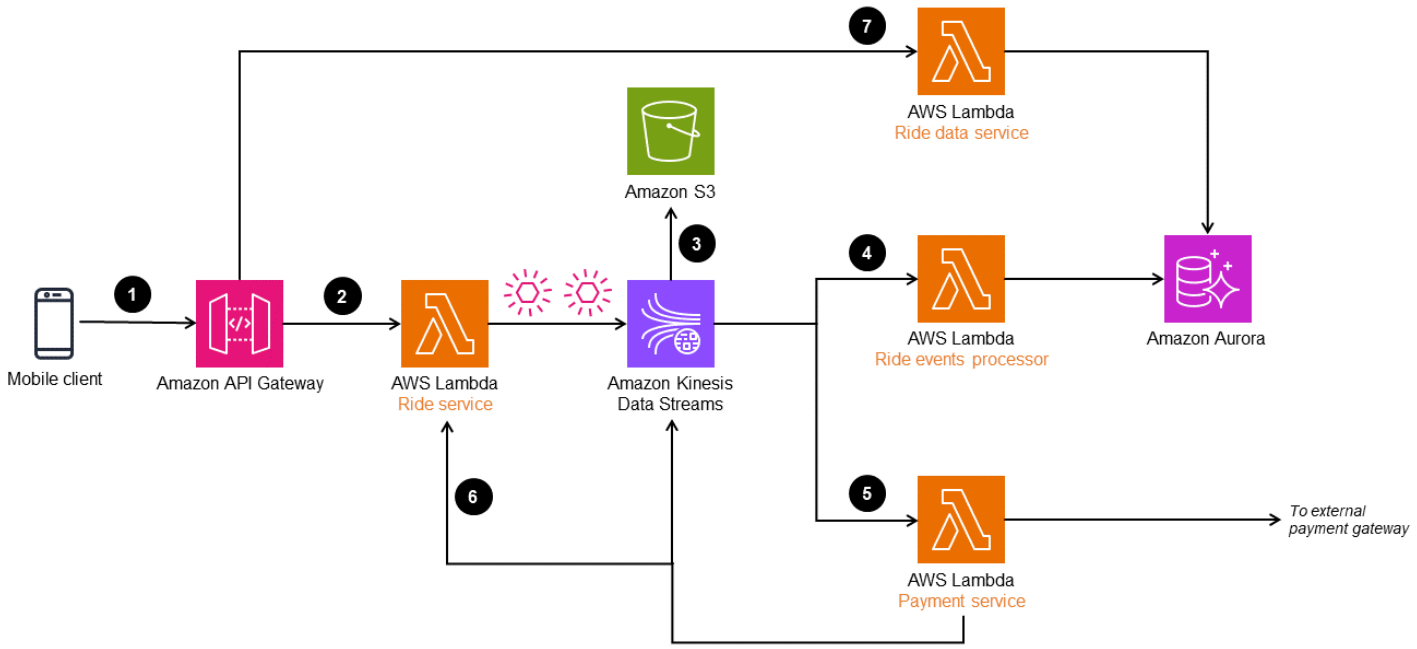
The ride events can be used to generate read-only data stores by using the CQRS or materialized view pattern. You can obtain the current state of the ride, the driver, or the booking by querying the read stores. Some events, such as `Location changed` or `Ride completed`, are published to another consumer for payment processing. When the ride is complete, all ride events are replayed to build a history of the ride for auditing or reporting purposes.

The event sourcing pattern is frequently used in applications that require a point-in-time recovery, and also when the data has to be projected in different formats by using a single source of truth. Both of these operations require a replay process to run the events and derive the required end state. The replay processor might also require a known starting point—ideally not from application launch, because that would not be an efficient process. We recommend that you take regular snapshots of the system state and apply a smaller number of events to derive an up-to-date state.

Implementation using AWS services

In the following architecture, Amazon Kinesis Data Streams is used as the event store. This service captures and manages application changes as events, and offers a high-throughput and real-time data streaming solution. To implement the event sourcing pattern on AWS, you can also use services such as Amazon EventBridge and Amazon Managed Streaming for Apache Kafka (Amazon MSK) based on your application's needs.

To enhance durability and enable auditing, you can archive the events that are captured by Kinesis Data Streams in Amazon Simple Storage Service (Amazon S3). This dual-storage approach helps retain historical event data securely for future analysis and compliance purposes.



The workflow consists of the following steps:

1. A ride booking request is made through a mobile client to an Amazon API Gateway endpoint.
2. The ride microservice (`Ride service` Lambda function) receives the request, transforms the objects, and publishes to Kinesis Data Streams.
3. The event data in Kinesis Data Streams is stored in Amazon S3 for compliance and audit history purposes.
4. The events are transformed and processed by the `Ride event processor` Lambda function and stored in an Amazon Aurora database to provide a materialized view for the ride data.
5. The completed ride events are filtered and sent for payment processing to an external payment gateway. When payment is completed, another event is sent to Kinesis Data Streams to update the Ride database.
6. When the ride is complete, the ride events are replayed to the `Ride service` Lambda function to build routes and the history of the ride.
7. Ride information can be read through the `Ride data service`, which reads from the Aurora database.

API Gateway can also send the event object directly to Kinesis Data Streams without the `Ride service` Lambda function. However, in a complex system such as a ride hailing service, the event object might need to be processed and enriched before it gets ingested into the data stream. For

this reason, the architecture has a `Ride service` that processes the event before sending it to Kinesis Data Streams.

Blog references

- New for AWS Lambda – SQS FIFO as an event source

Hexagonal architecture pattern

Intent

The hexagonal architecture pattern, which is also known as the ports and adapters pattern, was proposed by Dr. Alistair Cockburn in 2005. It aims to create loosely coupled architectures where application components can be tested independently, with no dependencies on data stores or user interfaces (UIs). This pattern helps prevent technology lock-in of data stores and UIs. This makes it easier to change the technology stack over time, with limited or no impact to business logic. In this loosely coupled architecture, the application communicates with external components over interfaces called *ports*, and uses *adapters* to translate the technical exchanges with these components.

Motivation

The hexagonal architecture pattern is used to isolate business logic (domain logic) from related infrastructure code, such as code to access a database or external APIs. This pattern is useful for creating loosely coupled business logic and infrastructure code for AWS Lambda functions that require integration with external services. In traditional architectures, a common practice is to embed business logic in the database layer as stored procedures and in the user interface. This practice, along with using UI-specific constructs within business logic, leads to closely coupled architectures that cause bottlenecks in database migrations and user experience (UX) modernization efforts. The hexagonal architecture pattern enables you to design your systems and applications by purpose rather than by technology. This strategy results in easily exchangeable application components such as databases, UX, and service components.

Applicability

Use the hexagonal architecture pattern when:

- You want to decouple your application architecture to create components that can be fully tested.
- Multiple types of clients can use the same domain logic.
- Your UI and database components require periodical technology refreshes that don't affect application logic.

- Your application requires multiple input providers and output consumers, and customizing the application logic leads to code complexity and lack of extensibility.

Issues and considerations

- **Domain-driven design**: Hexagonal architecture works especially well with domain-driven design (DDD). Each application component represents a sub-domain in DDD, and hexagonal architectures can be used to achieve loose coupling among application components.
- **Testability**: By design, a hexagonal architecture uses abstractions for inputs and outputs. Therefore, writing unit tests and testing in isolation become easier because of the inherent loose coupling.
- **Complexity**: The complexity of separating business logic from infrastructure code, when handled carefully, can bring great benefits such as agility, test coverage, and technology adaptability. Otherwise, issues can become complex to solve.
- **Maintenance overhead**: The additional adapter code that makes the architecture pluggable is justified only if the application component requires several input sources and output destinations to write to, or when the inputs and output data store has to change over time. Otherwise, the adapter becomes another additional layer to maintain, which introduces maintenance overhead.
- **Latency issues**: Using ports and adapters adds another layer, which might result in latency.

Implementation

Hexagonal architectures support the isolation of application and business logic from infrastructure code and from code that integrates the application with UIs, external APIs, databases, and message brokers. You can easily connect business logic components to other components (such as databases) in the application architecture through ports and adapters.

Ports are technology-agnostic entry points into an application component. These custom interfaces determine the interface that allows external actors to communicate with the application component, regardless of who or what implements the interface. This is similar to how a USB port allows many different types of devices to communicate with a computer, as long as they use a USB adapter.

Adapters interact with the application through a port by using a specific technology. Adapters plug into these ports, receive data from or provide data to the ports, and transform the data for further

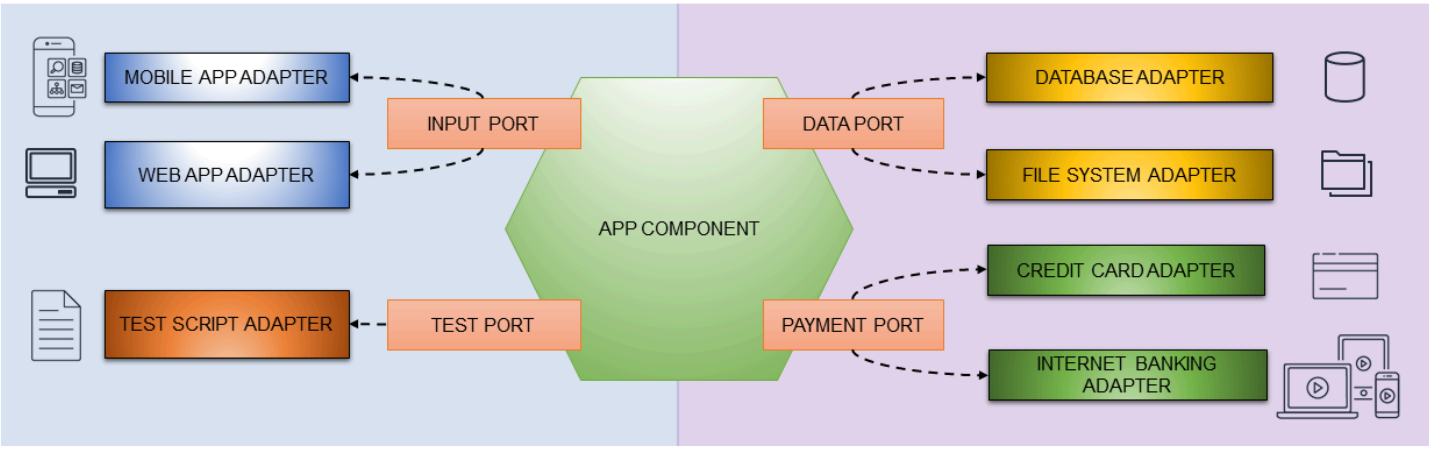
processing. For example, a REST adapter enables actors to communicate with the application component through a REST API. A port can have multiple adapters without any risk to the port or to the application component. To extend the previous example, adding a GraphQL adapter to the same port provides an additional means for actors to interact with the application through a GraphQL API without affecting the REST API, the port, or the application.

Ports connect to the application, and adapters serve as a connection to the outside world. You can use ports to create loosely coupled application components, and exchange dependent components by changing the adapter. This enables the application component to interact with external input and outputs without needing to have any contextual awareness. Components are exchangeable at any level, which facilitates automated testing. You can test components independently without any dependencies on the infrastructure code instead of provisioning an entire environment to conduct testing. The application logic doesn't depend on external factors, so testing is simplified and it becomes easier to mock dependencies.

For example, in a loosely coupled architecture, an application component should be able to read and write data without knowing the details of the data store. The responsibility of the application component is to supply data to an interface (port). An adapter defines the logic of writing to a data store, which can be a database, a file system, or an object storage system such as Amazon S3, depending on the application's needs.

High-level architecture

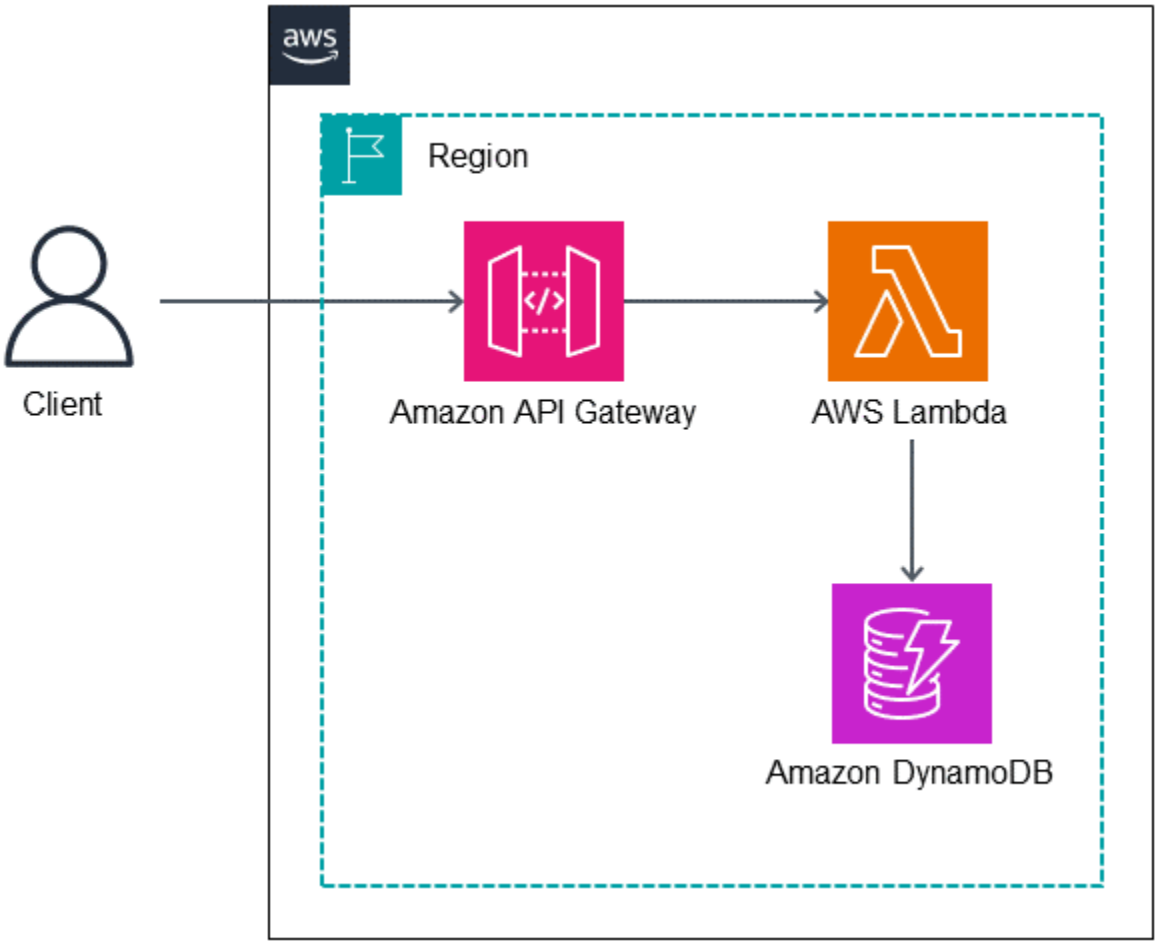
The application or application component contains the core business logic. It receives commands or queries from the ports, and sends requests out through the ports to external actors, which are implemented through adapters, as illustrated in the following diagram.



Implementation using AWS services

AWS Lambda functions often contain both business logic and database integration code, which are tightly coupled to meet an objective. You can use the hexagonal architecture pattern to separate business logic from infrastructure code. This separation enables unit testing of the business logic without any dependencies on the database code, and improves the agility of the development process.

In the following architecture, a Lambda function implements the hexagonal architecture pattern. The Lambda function is initiated by the Amazon API Gateway REST API. The function implements business logic and writes data to DynamoDB tables.



Sample code

The sample code in this section shows how to implement the domain model by using Lambda, separate it from infrastructure code (such as the code to access DynamoDB), and implement unit testing for the function.

Domain model

The domain model class has no knowledge of external components or dependencies—it only implements the business logic. In the following example, the class `Recipient` is a domain model class that checks for overlaps in the reservation date.

```
class Recipient:
def init(self, recipient_id:str, email:str, first_name:str, last_name:str, age:int):
self.__recipient_id = recipient_id
self.__email = email
self.__first_name = first_name
self.__last_name = last_name
self.__age = age
self.__slots = []

@property
def recipient_id(self):
return self.__recipient_id
.....

def are_slots_same_date(self, slot:Slot) -> bool:
for selfslot in self.__slots:
if selfslot.reservation_date == slot.reservation_date:
return True
return False

def is_slot_counts_equal_or_over_two(self) -> bool:
.....
```

Input port

The `RecipientInputPort` class connects to the recipient class and runs the domain logic.

```
class RecipientInputPort(IRecipientInputPort):
def init(self, recipient_output_port: IRecipientOutputPort, slot_output_port:
 ISlotOutputPort):
self.__recipient_output_port = recipient_output_port
self.__slot_output_port = slot_output_port

def make_reservation(self, recipient_id:str, slot_id:str) -> Status:
status = None
```

```
recipient = self.__recipient_output_port.get_recipient_by_id(recipient_id)
slot = self.__slot_output_port.get_slot_by_id(slot_id)
.....

# ----------------------------------------------------
# execute domain logic
# ----------------------------------------------------
ret = recipient.add_reserve_slot(slot)
.....

if ret == True:
status = Status(200, "The recipient's reservation is added.")
else:
status = Status(200, "The recipient's reservation is NOT added!")
return status
```

DynamoDB adapter class

The `DDBRecipientAdapter` class implements access to the DynamoDB tables.

```
class DDBRecipientAdapter(IRecipientAdapter):
def init(self):
ddb = boto3.resource('dynamodb')
self.__table = ddb.Table(table_name)

def load(self, recipient_id:str) -> Recipient:
try:
response = self.__table.get_item(
Key={'pk': pk_prefix + recipient_id})
  ...

def save(self, recipient:Recipient) -> bool:
try:
item = {
"pk": pk_prefix + recipient.recipient_id,
"email": recipient.email,
"first_name": recipient.first_name,
"last_name": recipient.last_name,
"age": recipient.age,
"slots": []
}
...
```

The Lambda function `get_recipient_input_port` is a factory for instances of the `RecipientInputPort` class. It constructs instances of output port classes with related adapter instances.

```
def get_recipient_input_port():
return RecipientInputPort(
RecipientOutputPort(DDBRecipientAdapter()),
SlotOutputPort(DDBSlotAdapter()))

def lambda_handler(event, context):

body = json.loads(event['body'])
recipient_id = body['recipient_id']
slot_id = body['slot_id']

# get an input port instance
recipient_input_port = get_recipient_input_port()
status = recipient_input_port.make_reservation(recipient_id, slot_id)

return {
"statusCode": status.status_code,
"body": json.dumps({
"message": status.message
}),
}
```

Unit testing

You can test the business logic for domain model classes by injecting mock classes. The following example provides the unit test for the domain model `Recipent` class.

```
def test_add_slot_one(fixture_recipient, fixture_slot):
slot = fixture_slot
target = fixture_recipient
target.add_reserve_slot(slot)
assert slot != None
assert target != None
assert 1 == len(target.slots)
assert slot.slot_id == target.slots[0].slot_id
assert slot.reservation_date == target.slots[0].reservation_date
assert slot.location == target.slots[0].location
assert False == target.slots[0].is_vacant
```

```
def test_add_slot_two(fixture_recipient, fixture_slot, fixture_slot_2):
.....

def test_cannot_append_slot_more_than_two(fixture_recipient, fixture_slot,
 fixture_slot_2, fixture_slot_3):
.....

def test_cannot_append_same_date_slot(fixture_recipient, fixture_slot):
.....
```

GitHub repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/aws-lambda-domain-model-sample.

Related content

- Hexagonal architecture, article by Alistair Cockburn
- Developing evolutionary architectures with AWS Lambda (AWS blog post in Japanese)

Videos

The following video (in Japanese) discusses the use of hexagonal architecture in the implementation of a domain model by using a Lambda function.

Publish-subscribe pattern

Intent

The publish-subscribe pattern, which is also known as the pub-sub pattern, is a messaging pattern that decouples a message sender (*publisher*) from interested receivers (*subscribers*). This pattern implements asynchronous communications by publishing messages or events through an intermediary known as a *message broker* or *router* (message infrastructure). The publish-subscribe pattern increases scalability and responsiveness for senders by offloading the responsibility of the message delivery to the message infrastructure, so the sender can focus on core message processing.

Motivation

In distributed architectures, system components often need to provide information to other components as events take place within the system. The publish-subscribe pattern separates concerns so that applications can focus on their core capabilities while the message infrastructure handles communication responsibilities such as message routing and reliable delivery. The publish-subscribe pattern enables asynchronous messaging to decouple the publisher and subscribers. Publishers can also send messages without the knowledge of subscribers.

Applicability

Use the publish-subscribe pattern when:

- Parallel processing is required if a single message has different workflows.
- Broadcasting messages to multiple subscribers and real-time responses from receivers aren't required.
- The system or application can tolerate eventual consistency for data or state.
- The application or component has to communicate with other applications or services that might use different languages, protocols, or platforms.

Issues and considerations

- **Subscriber availability:** The publisher isn't aware whether the subscribers are listening, and they might not be. Published messages are transient in nature and can result in being dropped if the subscribers aren't available.
- **Message delivery guarantee:** Typically, the publish-subscribe pattern can't guarantee the delivery of messages to all subscriber types, although certain services such as Amazon Simple Notification Service (Amazon SNS) can provide exactly-once delivery to some subscriber subsets.
- **Time to live (TTL):** Messages have a lifetime and expire if they aren't processed within the time period. Consider adding the published messages to a queue so they can persist, and guarantee processing beyond the TTL period.
- **Message relevancy:** Producers can set a time span for relevancy as part of the message data, and the message can be discarded after this date. Consider designing consumers to examine this information before you decide how to process the message.
- **Eventual consistency:** There is a delay between the time the message is published and the time it's consumed by the subscriber. This might result in the subscriber data stores becoming eventually consistent when strong consistency is required. Eventual consistency might also be an issue when producers and consumers require near real time interaction.
- **Unidirectional communication:** The publish-subscribe pattern is considered unidirectional. Applications that require bidirectional messaging with a return subscription channel should consider using a request-reply pattern if a synchronous response is required.
- **Message order:** Message ordering isn't guaranteed. If consumers require ordered messages, we recommend that you use Amazon SNS FIFO topics to guarantee ordering.
- **Message duplication:** Based on the messaging infrastructure, duplicate messages can be delivered to consumers. The consumers must be designed to be idempotent to handle duplicate message processing. Alternatively, use Amazon SNS FIFO topics to guarantee an exactly-once delivery.
- **Message filtering:** Consumers are often interested only in a subset of the messages published by a producer. Provide mechanisms to allow subscribers to filter or narrow down the messages they receive by providing topics or content filters.
- **Message replay:** Message replay capabilities might depend on the messaging infrastructure. You can also provide custom implementations depending on the use case.

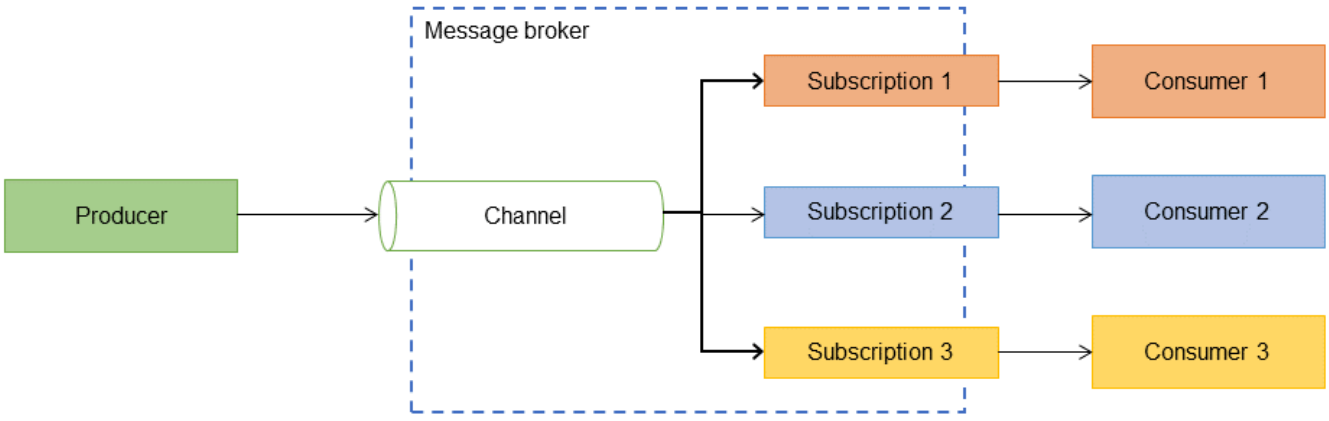
- **Dead-letter queues:** In a postal system, a dead-letter office is a facility for processing undeliverable mail. In pub/sub messaging, a dead-letter queue (DLQ) is a queue for messages that can't be delivered to a subscribed endpoint.

Implementation

High-level architecture

In a publish-subscribe pattern, the asynchronous messaging subsystem known as a message broker or router keeps track of subscriptions. When a producer publishes an event, the messaging infrastructure sends a message to each consumer. After a message is sent to subscribers, it is removed from the message infrastructure so it can't be replayed, and new subscribers do not see the event. Message brokers or routers decouple the event producer from message consumers by:

- Providing an input channel for the producer to publish events that are packaged into messages, using a defined message format.
- Creating an individual output channel per subscription. A *subscription* is the consumer's connection, where they listen for event messages that are associated with a specific input channel.
- Copying messages from the input channel to the output channel for all consumers when the event is published.



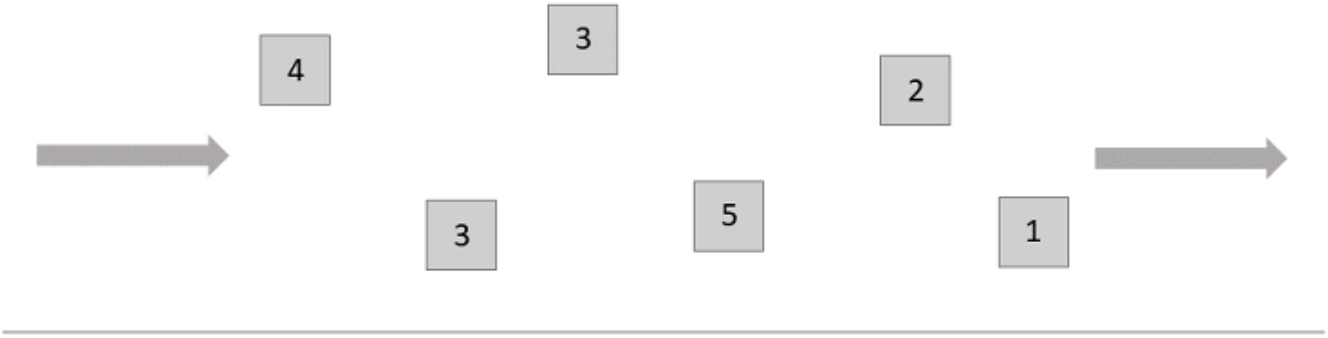
Implementation using AWS services

Amazon SNS

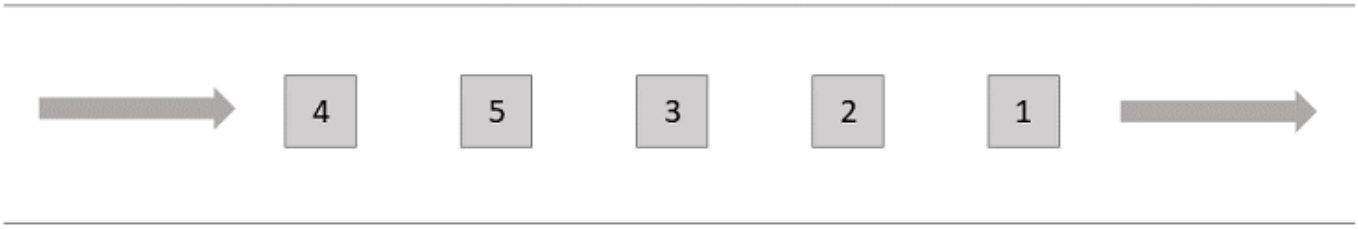
Amazon SNS is a fully managed publisher-subscriber service that provides application-to-application (A2A) messaging to decouple distributed applications. It also provides application-to-person (A2P) messaging for sending SMS, email, and other push notifications.

Amazon SNS provides two types of topics: standard and first in, first out (FIFO).

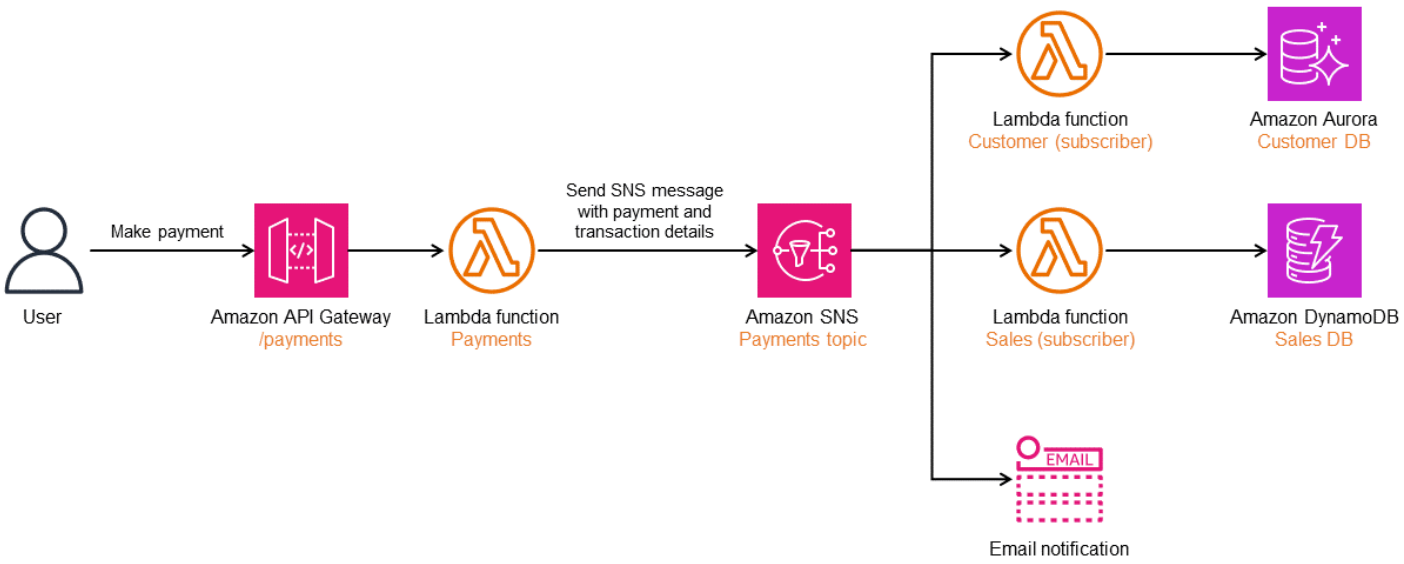
- Standard topics support an unlimited number of messages per second, and provide best-effort ordering and deduplication.



- FIFO topics provide strict ordering and deduplication, and support up to 300 messages per second or 10 MB per second per FIFO topic (whichever comes first).

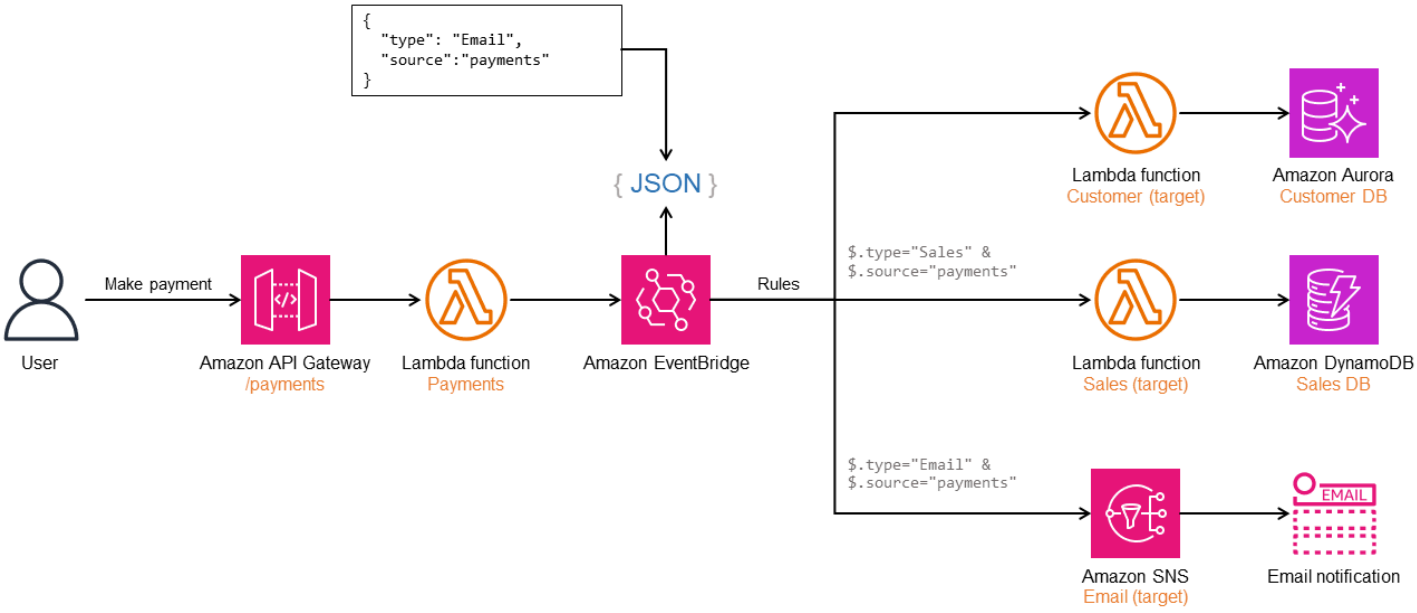


The following illustration shows how you can use Amazon SNS to implement the publish-subscribe pattern. After a user makes a payment, an SNS message is sent by the `Payments` Lambda function to the `Payments` SNS topic. This SNS topic has three subscribers. Each subscriber receives a copy of the message and processes it.



Amazon EventBridge

You can use Amazon EventBridge when you need more complex routing of messages from multiple producers across different protocols to subscribed consumers, or direct and fan-out subscriptions. EventBridge also supports content-based routing, filtering, sequencing, and splitting or aggregation. In the following illustration, EventBridge is used to build a version of the publish-subscribe pattern in which subscribers are defined by using event rules. After a user makes a payment, the `Payments` Lambda function sends a message to EventBridge by using the default event bus based on a custom schema that has three rules pointing to different targets. Each microservice processes the messages and performs the required actions.



Workshop

- Building event-driven architectures on AWS
- Send Fanout Event Notifications with Amazon Simple Queue Service (Amazon SQS) and Amazon Simple Notification Service (Amazon SNS)

Blog references

- Choosing between messaging services for serverless applications
- Designing durable serverless applications with DLQs for Amazon SNS, Amazon SQS, AWS Lambda
- Simplify your pub/sub messaging with Amazon SNS message filtering

Related content

- Features of pub/sub messaging

Retry with backoff pattern

Intent

The retry with backoff pattern improves application stability by transparently retrying operations that fail due to transient errors.

Motivation

In distributed architectures, transient errors might be caused by service throttling, temporary loss of network connectivity, or temporary service unavailability. Automatically retrying operations that fail because of these transient errors improves the user experience and application resilience. However, frequent retries can overload network bandwidth and cause contention. Exponential backoff is a technique where operations are retried by increasing wait times for a specified number of retry attempts.

Applicability

Use the retry with backoff pattern when:

- Your services frequently throttle the request to prevent overload, resulting in a *429 Too many requests* exception to the calling process.
- The network is an unseen participant in distributed architectures, and temporary network issues result in failures.
- The service being called is temporarily unavailable, causing failures. Frequent retries might cause service degradation unless you introduce a backoff timeout by using this pattern.

Issues and considerations

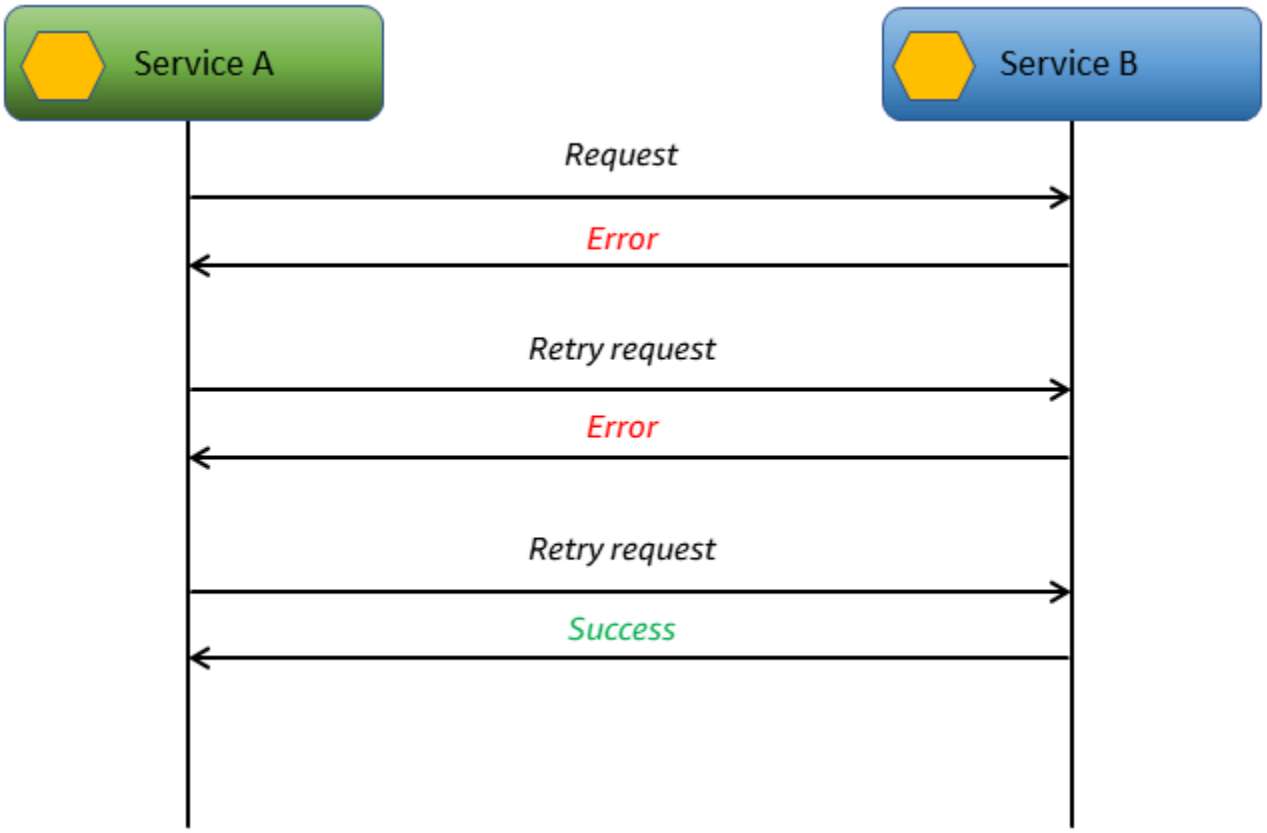
- **Idempotency**: If multiple calls to the method have the same effect as a single call on the system state, the operation is considered idempotent. Operations should be idempotent when you use the retry with backoff pattern. Otherwise, partial updates might corrupt the system state.
- **Network bandwidth**: Service degradation can occur if too many retries occupy network bandwidth, leading to slow response times.

- **Fail fast scenarios**: For non-transient errors, if you can determine the cause of the failure, it is more efficient to fail fast by using the circuit breaker pattern.
- **Backoff rate**: Introducing exponential backoff can have an impact on the service timeout, resulting in longer wait times for the end user.

Implementation

High-level architecture

The following diagram illustrates how Service A can retry the calls to Service B until a successful response is returned. If Service B doesn't return a successful response after a few tries, Service A can stop retrying and return a failure to its caller.



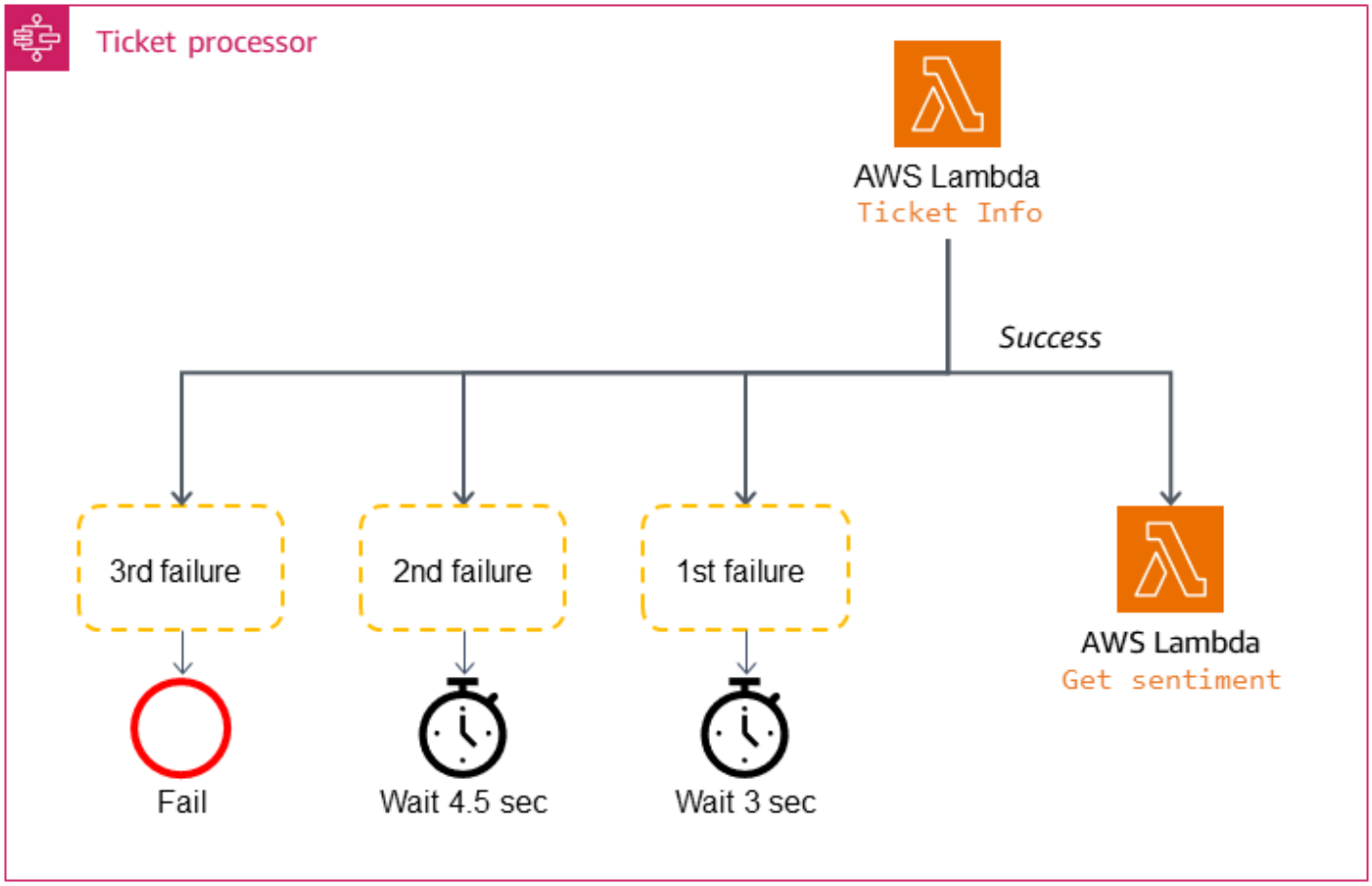
Implementation using AWS services

The following diagram shows a ticket processing workflow on a customer support platform. Tickets from unhappy customers are expedited by automatically escalating the ticket priority. The `Ticket info` Lambda function extracts the ticket details and calls the `Get sentiment` Lambda function.

The `Get sentiment` Lambda function checks the customer sentiments by passing the description to Amazon Comprehend (not shown).

If the call to the `Get sentiment` Lambda function fails, the workflow retries the operation three times. AWS Step Functions allows exponential backoff by letting you configure the backoff value.

In this example, a maximum of three retries are configured with an increase multiplier of 1.5 seconds. If the first retry occurs after 3 seconds, the second retry occurs after 3 x 1.5 seconds = 4.5 seconds, and the third retry occurs after 4.5 x 1.5 seconds = 6.75 seconds. If the third retry is unsuccessful, the workflow fails. The backoff logic doesn't require any custom code—it's provided as a configuration by AWS Step Functions.



Sample code

The following code shows the implementation of the retry with backoff pattern.

```
public async Task DoRetriesWithBackOff()
  {
```

```
    int retries = 0;
    bool retry;
    do
    {
      //Sample object for sending parameters
      var parameterObj = new InputParameter { SimulateTimeout = "false" };
      var content = new StringContent(JsonConvert.SerializeObject(parameterObj),
                              System.Text.Encoding.UTF8, "application/json");
      var waitInMilliseconds = Convert.ToInt32((Math.Pow(2, retries) - 1) * 100);
      System.Threading.Thread.Sleep(waitInMilliseconds);
      var response =  await _client.PostAsync(_baseURL, content);
      switch (response.StatusCode)
      {
        //Success
        case HttpStatusCode.OK:
          retry = false;
          Console.WriteLine(response.Content.ReadAsStringAsync().Result);
          break;
        //Throttling, timeouts
        case HttpStatusCode.TooManyRequests:
        case HttpStatusCode.GatewayTimeout:
          retry = true;
          break;
        //Some other error occured, so stop calling the API
        default:
          retry = false;
          break;
      }
      retries++;
    } while (retry && retries < MAX_RETRIES);
  }
```

GitHub repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/retry-with-backoff.

Related content

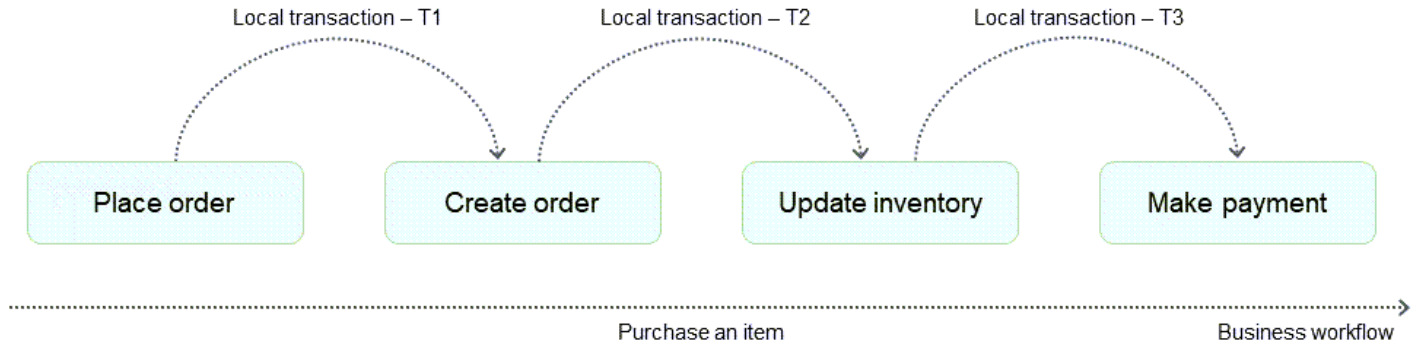
- Timeouts, retries, and backoff with jitter (Amazon Builders' Library)

Saga patterns

A *saga* consists of a sequence of local transactions. Each local transaction in a saga updates the database and triggers the next local transaction. If a transaction fails, the saga runs compensating transactions to revert the database changes made by the previous transactions.

This sequence of local transactions helps achieve a business workflow by using continuation and compensation principles. The *continuation principle* decides the forward recovery of the workflow, whereas the *compensation principle* decides the backward recovery. If the update fails at any step in the transaction, the saga publishes an event for either continuation (to retry the transaction) or compensation (to go back to the previous data state). This ensures that data integrity is maintained and is consistent across the data stores.

For example, when a user purchases a book from an online retailer, the process consists of a sequence of transactions—such as order creation, inventory update, payment, and shipping—that represents a business workflow. In order to complete this workflow, the distributed architecture issues a sequence of local transactions to create an order in the order database, update the inventory database, and update the payment database. When the process is successful, these transactions are invoked sequentially to complete the business workflow, as the following diagram shows. However, if any of these local transactions fails, the system should be able to decide on an appropriate next step—that is, either a forward recovery or a backward recovery.



The following two scenarios help determine whether the next step is forward recovery or backward recovery:

- Platform-level failure, where something goes wrong with the underlying infrastructure and causes the transaction to fail. In this case, the saga pattern can perform a forward recovery by retrying the local transaction and continuing the business process.

- Application-level failure, where the payment service fails because of an invalid payment. In this case, the saga pattern can perform a backward recovery by issuing a compensatory transaction to update the inventory and the order databases, and reinstate their previous state.

The saga pattern handles the business workflow and ensures that a desirable end state is reached through forward recovery. In case of failures, it reverts the local transactions by using backward recovery to avoid data consistency issues.

The saga pattern has two variants: choreography and orchestration.

Saga choreography

The saga choreography pattern depends on the events published by the microservices. The saga participants (microservices) subscribe to the events and act based on the event triggers. For example, the order service in the following diagram emits an `OrderPlaced` event. The inventory service subscribes to that event and updates the inventory when the `OrderPlaced` event is emitted. Similarly, the participant services act based on the context of the emitted event.

The saga choreography pattern is suitable when there are only a few participants in the saga, and you need a simple implementation with no single point of failure. When more participants are added, it becomes harder to track the dependencies between the participants by using this pattern.

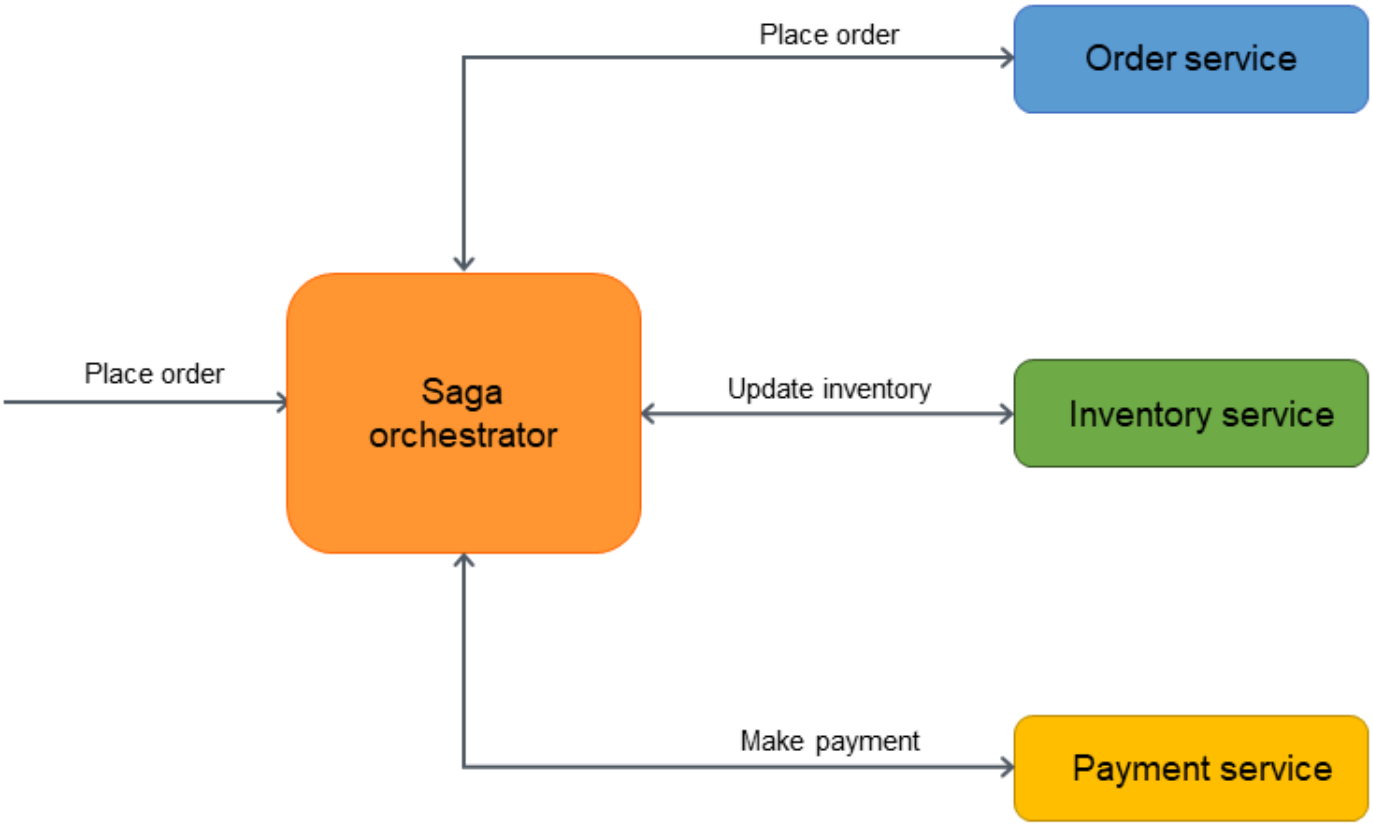
For a detailed review, see the Saga choreography section of this guide.

Saga orchestration

The saga orchestration pattern has a central coordinator called an *orchestrator*. The saga orchestrator manages and coordinates the entire transaction lifecycle. It is aware of the series of steps to be performed to complete the transaction. To run a step, it sends a message to the participant microservice to perform the operation. The participant microservice completes the operation and sends a message back to the orchestrator. Based on the message it receives, the orchestrator decides which microservice to run next in the transaction.

The saga orchestration pattern is suitable when there are many participants, and loose coupling is required between saga participants. The orchestrator encapsulates the complexity in the logic by

making the participants loosely coupled. However, the orchestrator can become a single point of failure because it controls the entire workflow.



For a detailed review, see the Saga orchestration section of this guide.

Saga choreography pattern

Intent

The saga choreography pattern helps preserve data integrity in distributed transactions that span multiple services by using event subscriptions. In a distributed transaction, multiple services can be called before a transaction is completed. When the services store data in different data stores, it can be challenging to maintain data consistency across these data stores.

Motivation

A *transaction* is a single unit of work that might involve multiple steps, where all steps are completely executed or no step is executed, resulting in a data store that retains its consistent

state. The terms *atomicity, consistency, isolation, and durability (ACID)* define the properties of a transaction. Relational databases provide ACID transactions to maintain data consistency.

To maintain consistency in a transaction, relational databases use the two-phase commit (2PC) method. This consists of a *prepare phase* and a *commit phase*.

- In the prepare phase, the coordinating process requests the transaction's participating processes (participants) to promise to either commit or roll back the transaction.
- In the commit phase, the coordinating process requests the participants to commit the transaction. If the participants cannot agree to commit in the prepare phase, the transaction is rolled back.

In distributed systems that follow a database-per-service design pattern, the two-phase commit is not an option. This is because each transaction is distributed across various databases, and there is no single controller that can coordinate a process that's similar to the two-phase commit in relational data stores. In this case, one solution is to use the saga choreography pattern.

Applicability

Use the saga choreography pattern when:

- Your system requires data integrity and consistency in distributed transactions that span multiple data stores.
- The data store (for example, a NoSQL database) doesn't provide 2PC to provide ACID transactions, you need to update multiple tables within a single transaction, and implementing 2PC within the application boundaries would be a complex task.
- A central controlling process that manages the participant transactions might become a single point of failure.
- The saga participants are independent services and need to be loosely coupled.
- There is communication between bounded contexts in a business domain.

Issues and considerations

- **Complexity:** As the number of microservices increases, saga choreography can become difficult to manage because of the number of interactions between the microservices. Additionally, compensatory transactions and retries add complexities to the application code, which can result in maintenance overhead. Choreography is suitable when there are only a few participants in

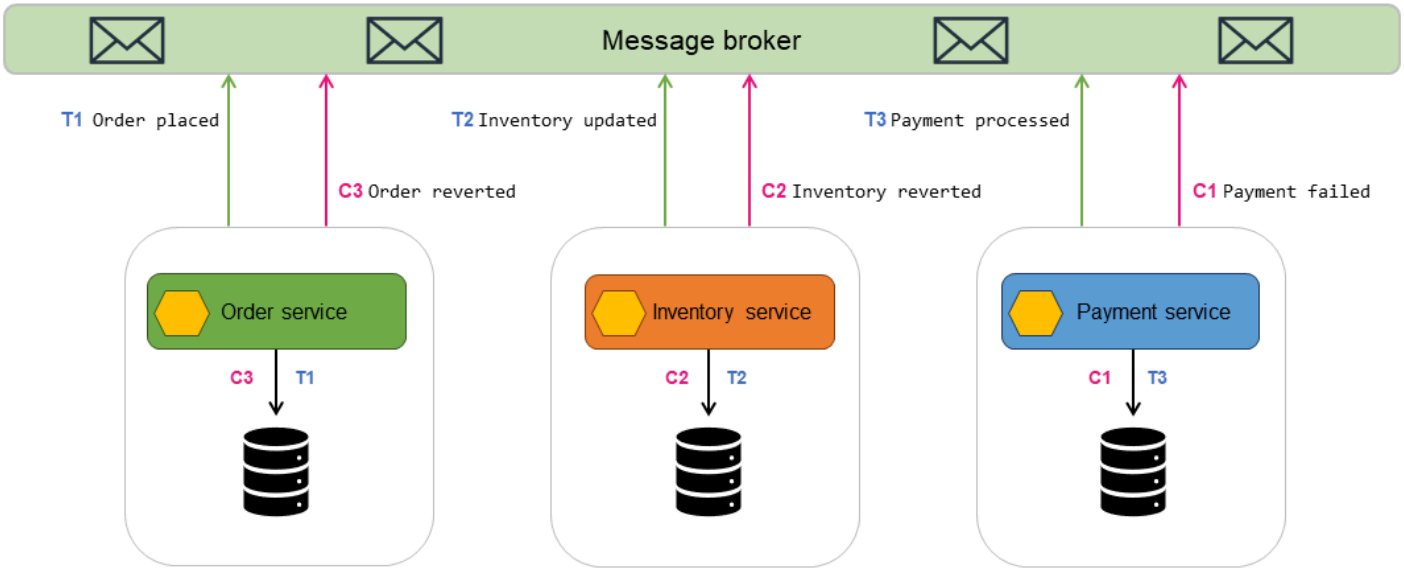
the saga, and you need a simple implementation with no single point of failure. When more participants are added, it becomes harder to track the dependencies between the participants by using this pattern.

- **Resilient implementation:** In saga choreography, it's more difficult to implement timeouts, retries, and other resiliency patterns globally, compared with saga orchestration. Choreography must be implemented on individual components instead of at an orchestrator level.
- **Cyclic dependencies:** The participants consume messages that are published by one another. This might result in cyclic dependencies, leading to code complexities and maintenance overheads, and possible deadlocks.
- **Dual writes issue:** The microservice has to atomically update the database and publish an event. The failure of either operation might lead to an inconsistent state. One way to solve this is to use the transactional outbox pattern.
- **Preserving events:** The saga participants act based on the events published. It's important to save the events in the order they occur for audit, debugging, and replay purposes. You can use the event sourcing pattern to persist the events in an event store in case a replay of the system state is required to restore data consistency. Event stores can also be used for auditing and troubleshooting purposes because they reflect every change in the system.
- **Eventual consistency**: The sequential processing of local transactions results in eventual consistency, which can be a challenge in systems that require strong consistency. You can address this issue by setting your business teams' expectations for the consistency model or reassess the use case and switch to a database that provides strong consistency.
- **Idempotency**: Saga participants have to be idempotent to allow repeated execution in case of transient failures that are caused by unexpected crashes and orchestrator failures.
- **Transaction isolation**: The saga pattern lacks transaction isolation, which is one of the four properties in ACID transactions. The degree of isolation of a transaction determines how much other concurrent transactions can affect the data that the transaction operates on. Concurrent orchestration of transactions can lead to stale data. We recommend using semantic locking to handle such scenarios.
- **Observability**: Observability refers to detailed logging and tracing to troubleshoot issues in the implementation and orchestration process. This becomes important when the number of saga participants increases, resulting in complexities in debugging. End-to-end monitoring and reporting are more difficult to achieve in saga choreography, compared with saga orchestration.
- **Latency issues**: Compensatory transactions can add latency to the overall response time when the saga consists of several steps. If the transactions make synchronous calls, this can increase the latency further.

Implementation

High-level architecture

In the following architecture diagram, the saga choreography has three participants: the order service, the inventory service, and the payment service. Three steps are required to complete the transaction: T1, T2, and T3. Three compensatory transactions restore the data to the initial state: C1, C2, and C3.



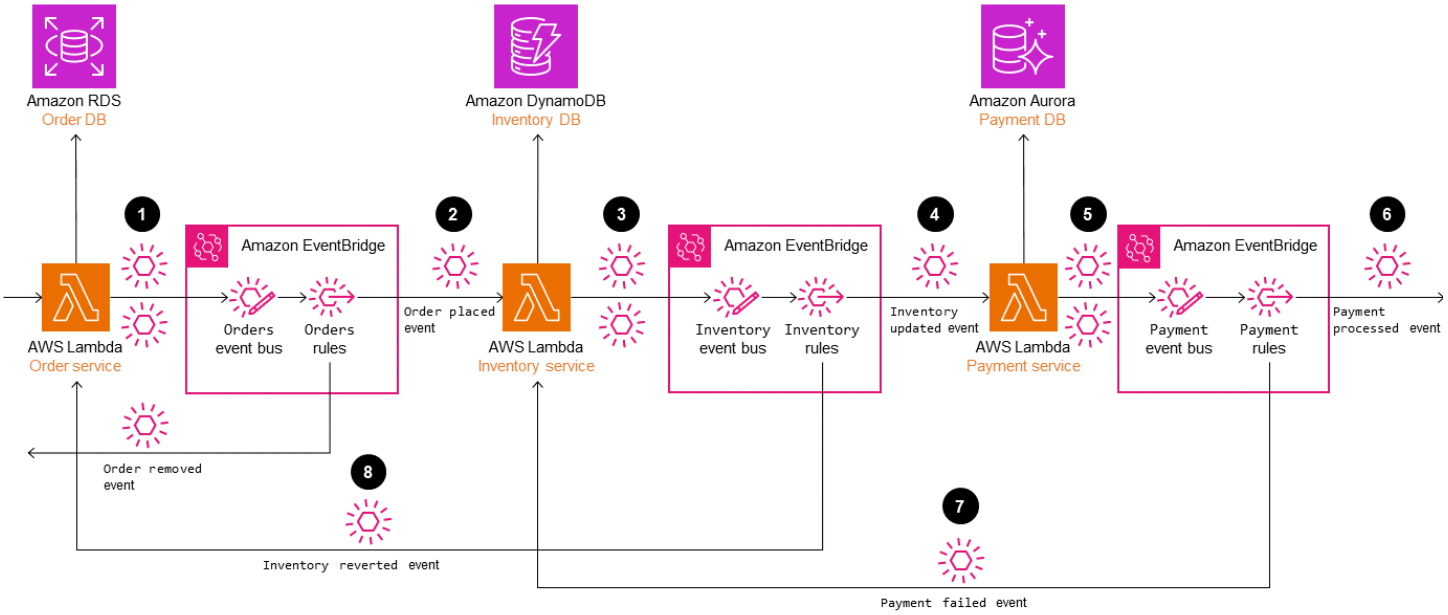
- The order service runs a local transaction, T1, which atomically updates the database and publishes an `Order placed` message to the message broker.
- The inventory service subscribes to the order service messages and receives the message that an order has been created.
- The inventory service runs a local transaction, T2, which atomically updates the database and publishes an `Inventory updated` message to the message broker.
- The payment service subscribes to the messages from the inventory service and receives the message that the inventory has been updated.
- The payment service runs a local transaction, T3, which atomically updates the database with payment details and publishes a `Payment processed` message to the message broker.
- If the payment fails, the payment service runs a compensatory transaction, C1, which atomically reverts the payment in the database and publishes a `Payment failed` message to the message broker.
- The compensatory transactions C2 and C3 are run to restore data consistency.

Implementation using AWS services

You can implement the saga choreography pattern by using Amazon EventBridge. EventBridge uses events to connect application components. It processes events through event buses or pipes. An event bus is a router that receives events and delivers them to zero or more destinations, or *targets*. Rules associated with the event bus evaluate events as they arrive and send them to targets for processing.

In the following architecture:

- The microservices—order service, inventory service, and payment service—are implemented as Lambda functions.
- There are three custom EventBridge buses: `Orders` event bus, `Inventory` event bus, and `Payment` event bus.
- `Orders` rules, `Inventory` rules, and `Payment` rules match the events that are sent to the corresponding event bus and invoke the Lambda functions.



In a successful scenario, when an order is placed:

1. The order service processes the request and sends the event to the `Orders` event bus.
2. The `Orders` rules match the events and starts the inventory service.

3. The inventory service updates the inventory and sends the event to the `Inventory` event bus.
4. The `Inventory` rules match the events and start the payment service.
5. The payment service processes the payment and sends the event to the `Payment` event bus.
6. The `Payment` rules match the events and send the `Payment processed` event notification to the listener.

   Alternatively, when there is an issue in order processing, the EventBridge rules start the compensatory transactions for reverting the data updates to maintain data consistency and integrity.
7. If the payment fails, the `Payment` rules process the event and start the inventory service. The inventory service runs compensatory transactions to revert the inventory.
8. When the inventory has been reverted, the inventory service sends the `Inventory reverted` event to the `Inventory` event bus. This event is processed by `Inventory` rules. It starts the order service, which runs the compensatory transaction to remove the order.

Related content

- Saga orchestration pattern
- Transactional outbox pattern
- Retry with backoff pattern

Saga orchestration pattern

Intent

The saga orchestration pattern uses a central coordinator (*orchestrator*) to help preserve data integrity in distributed transactions that span multiple services. In a distributed transaction, multiple services can be called before a transaction is completed. When the services store data in different data stores, it can be challenging to maintain data consistency across these data stores.

Motivation

A *transaction* is a single unit of work that might involve multiple steps, where all steps are completely executed or no step is executed, resulting in a data store that retains its consistent

state. The terms *atomicity, consistency, isolation, and durability (ACID)* define the properties of a transaction. Relational databases provide ACID transactions to maintain data consistency.

To maintain consistency in a transaction, relational databases use the two-phase commit (2PC) method. This consists of a *prepare phase* and a *commit phase*.

- In the prepare phase, the coordinating process requests the transaction's participating processes (participants) to promise to either commit or roll back the transaction.
- In the commit phase, the coordinating process requests the participants to commit the transaction. If the participants cannot agree to commit in the prepare phase, the transaction is rolled back.

In distributed systems that follow a database-per-service design pattern, the two-phase commit is not an option. This is because each transaction is distributed across various databases, and there is no single controller that can coordinate a process that's similar to the two-phase commit in relational data stores. In this case, one solution is to use the saga orchestration pattern.

Applicability

Use the saga orchestration pattern when:

- Your system requires data integrity and consistency in distributed transactions that span multiple data stores.
- The data store doesn't provide 2PC to provide ACID transactions, and implementing 2PC within the application boundaries is a complex task.
- You have NoSQL databases, which do not provide ACID transactions, and you need to update multiple tables within a single transaction.

Issues and considerations

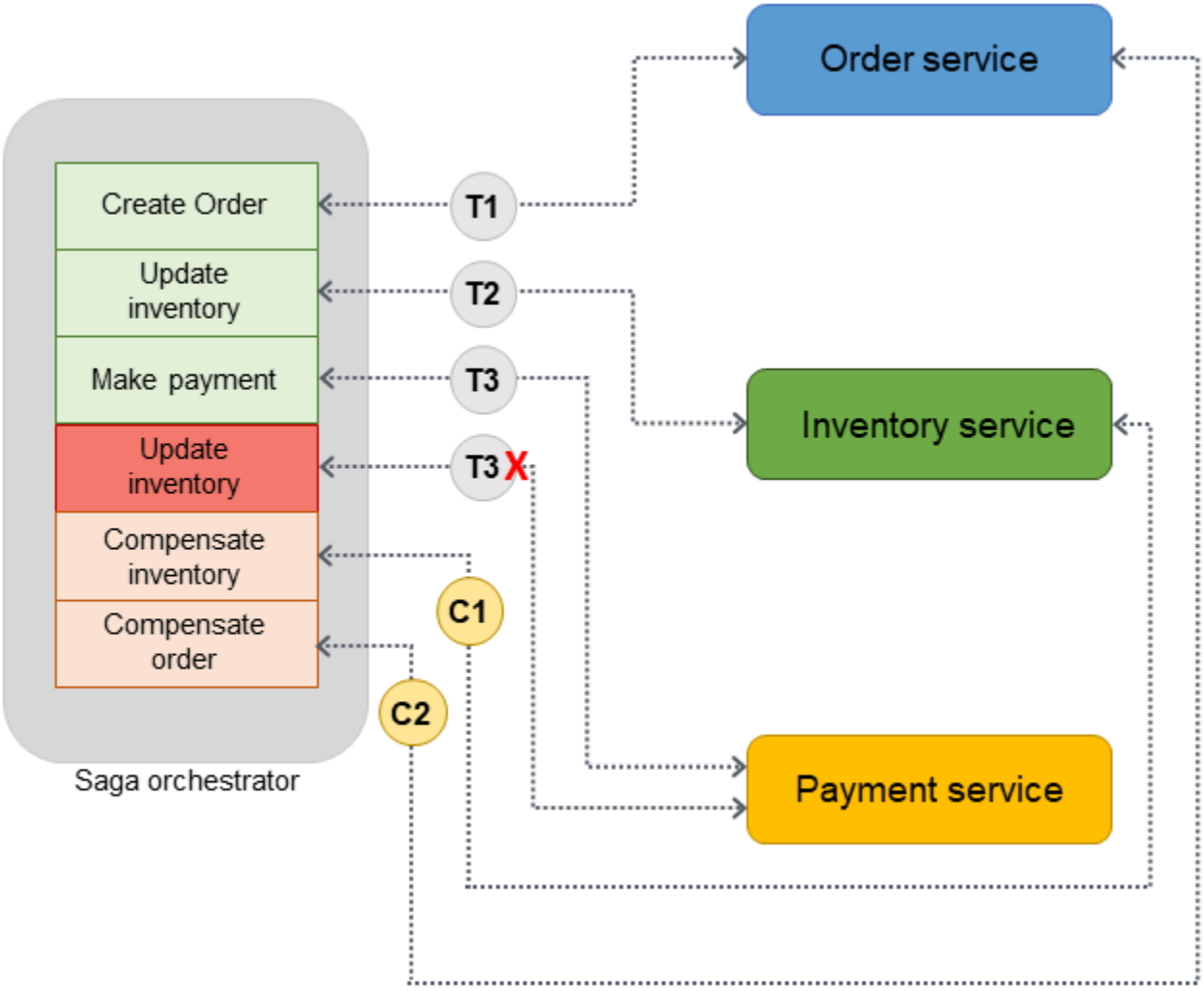
- **Complexity**: Compensatory transactions and retries add complexities to the application code, which can result in maintenance overhead.
- **Eventual consistency**: The sequential processing of local transactions results in eventual consistency, which can be a challenge in systems that require strong consistency. You can address this issue by setting your business teams' expectations for the consistency model or by switching to a data store that provides strong consistency.

- **Idempotency**: Saga participants need to be idempotent to allow repeated execution in case of transient failures caused by unexpected crashes and orchestrator failures.
- **Transaction isolation**: Saga lacks transaction isolation. Concurrent orchestration of transactions can lead to stale data. We recommend using semantic locking to handle such scenarios.
- **Observability**: Observability refers to detailed logging and tracing to troubleshoot issues in the execution and orchestration process. This becomes important when the number of saga participants increases, resulting in complexities in debugging.
- **Latency issues**: Compensatory transactions can add latency to the overall response time when the saga consists of several steps. Avoid synchronous calls in such cases.
- **Single point of failure**: The orchestrator can become a single point of failure because it coordinates the entire transaction. In some cases, the saga choreography pattern is preferred because of this issue.

Implementation

High-level architecture

In the following architecture diagram, the saga orchestrator has three participants: the order service, the inventory service, and the payment service. Three steps are required to complete the transaction: T1, T2, and T3. The saga orchestrator is aware of the steps and runs them in the required order. When step T3 fails (payment failure), the orchestrator runs the compensatory transactions C1 and C2 to restore the data to the initial state.



You can use AWS Step Functions to implement saga orchestration when the transaction is distributed across multiple databases.

Implementation using AWS services

The sample solution uses the standard workflow in Step Functions to implement the saga orchestration pattern.

When a customer calls the API, the Lambda function is invoked, and preprocessing occurs in the Lambda function. The function starts the Step Functions workflow to start processing the distributed transaction. If preprocessing isn't required, you can initiate the Step Functions workflow directly from API Gateway without using the Lambda function.

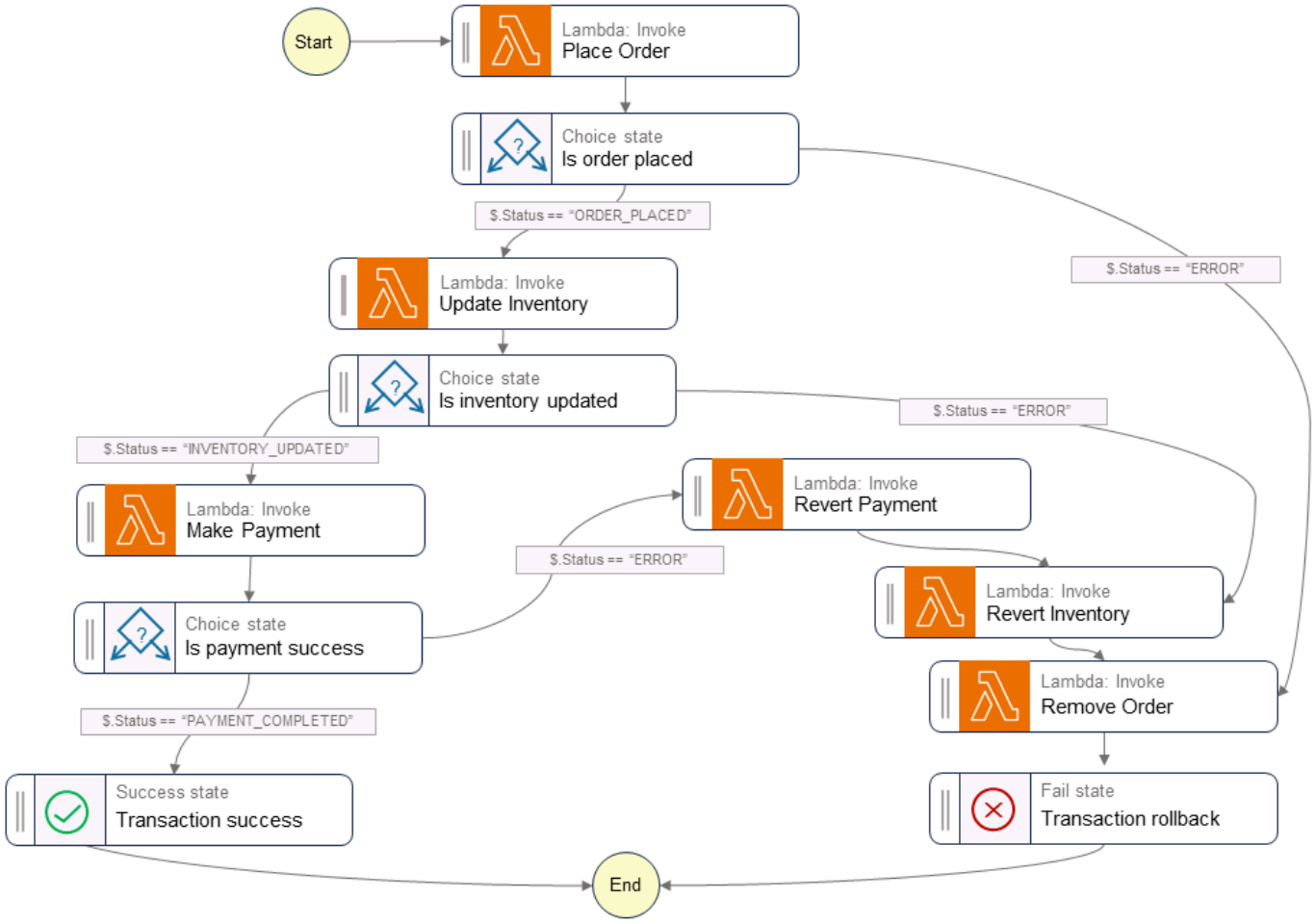
The use of Step Functions mitigates the single point of failure issue, which is inherent in the implementation of the saga orchestration pattern. Step Functions has built-in fault tolerance and maintains service capacity across multiple Availability Zones in each AWS Region to protect applications against individual machine or data center failures. This helps ensure high availability for both the service itself and for the application workflow it operates.

The Step Functions workflow

The Step Functions state machine allows you to configure the decision-based control flow requirements for the pattern implementation. The Step Functions workflow calls the individual services for order placement, inventory update, and payment processing to complete the transaction and sends an event notification for further processing. The Step Functions workflow acts as the orchestrator to coordinate the transactions. If the workflow contains any errors, the orchestrator runs the compensatory transactions to ensure that data integrity is maintained across services.

The following diagram shows the steps that run inside the Step Functions workflow. The `Place Order`, `Update Inventory`, and `Make Payment` steps indicate the success path. The order is placed, the inventory is updated, and the payment is processed before a `Success` state is returned to the caller.

The `Revert Payment`, `Revert Inventory`, and `Remove Order` Lambda functions indicate the compensatory transactions that the orchestrator runs when any step in the workflow fails. If the workflow fails at the `Update Inventory` step, the orchestrator calls the `Revert Inventory` and `Remove Order` steps before returning a `Fail` state to the caller. These compensatory transactions ensure that data integrity is maintained. The inventory returns to its original level and the order is reverted.



Sample code

The following sample code shows how you can create a saga orchestrator by using Step Functions. To view the complete code, see the GitHub repository for this example.

Task definitions

```
var successState = new Succeed(this,"SuccessState");
var failState = new Fail(this, "Fail");

var placeOrderTask = new LambdaInvoke(this, "Place Order", new LambdaInvokeProps
{
    LambdaFunction = placeOrderLambda,
    Comment = "Place Order",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
});
```

```
var updateInventoryTask = new LambdaInvoke(this,"Update Inventory", new
 LambdaInvokeProps
{
    LambdaFunction = updateInventoryLambda,
    Comment = "Update inventory",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
});

var makePaymentTask = new LambdaInvoke(this,"Make Payment", new LambdaInvokeProps
{
    LambdaFunction = makePaymentLambda,
    Comment = "Make Payment",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
});

var removeOrderTask = new LambdaInvoke(this, "Remove Order", new LambdaInvokeProps
{
    LambdaFunction = removeOrderLambda,
    Comment = "Remove Order",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
}).Next(failState);

var revertInventoryTask = new LambdaInvoke(this,"Revert Inventory", new
 LambdaInvokeProps
{
    LambdaFunction = revertInventoryLambda,
    Comment = "Revert inventory",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
}).Next(removeOrderTask);

var revertPaymentTask = new LambdaInvoke(this,"Revert Payment", new LambdaInvokeProps
{
    LambdaFunction = revertPaymentLambda,
    Comment = "Revert Payment",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
}).Next(revertInventoryTask);

var waitState = new Wait(this, "Wait state", new WaitProps
```

```
{
    Time = WaitTime.Duration(Duration.Seconds(30))
}).Next(revertInventoryTask);
```

Step function and state machine definitions

```
var stepDefinition = placeOrderTask
                .Next(new Choice(this, "Is order placed")
                    .When(Condition.StringEquals("$.Status", "ORDER_PLACED"),
 updateInventoryTask
                        .Next(new Choice(this, "Is inventory updated")
                            .When(Condition.StringEquals("$.Status",
 "INVENTORY_UPDATED"),
                                makePaymentTask.Next(new Choice(this, "Is payment
 success")
                                    .When(Condition.StringEquals("$.Status",
 "PAYMENT_COMPLETED"), successState)
                                    .When(Condition.StringEquals("$.Status", "ERROR"),
 revertPaymentTask)))
                            .When(Condition.StringEquals("$.Status", "ERROR"),
 waitState)))
                    .When(Condition.StringEquals("$.Status", "ERROR"), failState));

var stateMachine = new StateMachine(this, "DistributedTransactionOrchestrator", new
 StateMachineProps {
    StateMachineName = "DistributedTransactionOrchestrator",
    StateMachineType = StateMachineType.STANDARD,
    Role = iamStepFunctionRole,
    TracingEnabled = true,
    Definition = stepDefinition
});
```

GitHub repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/saga-orchestration-netcore-blog.

Blog references

- Building a serverless distributed application using Saga Orchestration pattern

Related content

- Saga choreography pattern
- Transactional outbox pattern

Videos

The following video discusses how to implement the saga orchestration pattern by using AWS Step Functions.

Scatter-gather pattern

Intent

The scatter-gather pattern is a message routing pattern that involves broadcasting similar or related requests to multiple recipients, and aggregating their responses back into a single message by using a component called an *aggregator*. This pattern helps achieve parallelization, reduces processing latency, and handles asynchronous communication. It's straightforward to implement the scatter-gather pattern by using a synchronous approach, but a more powerful approach involves implementing it as message routing in asynchronous communication, either with or without a messaging service.

Motivation

In application processing, a request that might take a long time to process sequentially can be split into multiple requests that are processed in parallel. You can also send requests to multiple external systems through API calls to get a response. The scatter-gather pattern is useful when you need input from multiple sources. Scatter-gather aggregates the results to help you make an informed decision or to select the best response for the request.

The scatter-gather pattern consists of two phases, as its name implies:

- The *scatter phase* processes the request message and sends it to multiple recipients in parallel. During this phase, the application scatters requests across the network and continues to run without waiting for immediate responses.
- During the *gather phase*, the application collects the responses from recipients, and filters or combines them into a unified response. When all the responses have been collected, they can either be aggregated into a single response or the best one can be chosen for further processing.

Applicability

Use the scatter-gather pattern when:

- You plan to aggregate and consolidate data from various APIs to create an accurate response. The pattern consolidates information from disparate sources into a cohesive whole. For example,

a booking system can make a request to multiple recipients to get quotes from multiple external partners.

- The same request has to be sent to multiple recipients simultaneously to complete a transaction. For example, you can use this pattern to query inventory data in parallel to check a product's availability.
- You want to implement a reliable and scalable system where load balancing can be achieved by distributing requests across multiple recipients. If one recipient fails or experiences a high load, other recipients can still process requests.
- You want to optimize performance when implementing complex queries that involve multiple data sources. You can scatter the query to relevant databases, gather the partial results, and combine them into a comprehensive answer.
- You are implementing a type of map-reduce processing where the data request is routed to multiple data processing endpoints for sharding and replication. Partial results are filtered and combined to compose the right response.
- You want to distribute write operations across a partition key space in write-heavy workloads in key-value databases. The aggregator reads the results by querying the data in each shard, and then consolidates them into a single response.

Issues and considerations

- **Fault tolerance**: This pattern relies on multiple recipients that work in parallel, so it is essential to handle failures gracefully. To mitigate the impact of recipient failures on the overall system, you can implement strategies such as redundancy, replication, and fault detection.
- **Scale-out limits**: As the total number of processing nodes increases, the associated network overhead also increases. Every request that involves communication over the network can increase latency and negatively affect the benefits of parallelization.
- **Response time bottlenecks**: For operations that require all the recipients to be processed before the final processing is done, the performance of the overall system is constrained by the slowest recipient's response time.
- **Partial responses**: When requests are scattered to multiple recipients, some recipients can time out. In these cases, the implementation should communicate to the client that the response is incomplete. You can also display the response aggregation details by using a UI frontend.

- **Data consistency**: When you process data across multiple recipients, you must carefully consider data synchronization and conflict resolution techniques, to ensure that the final aggregated results are accurate and consistent.

Implementation

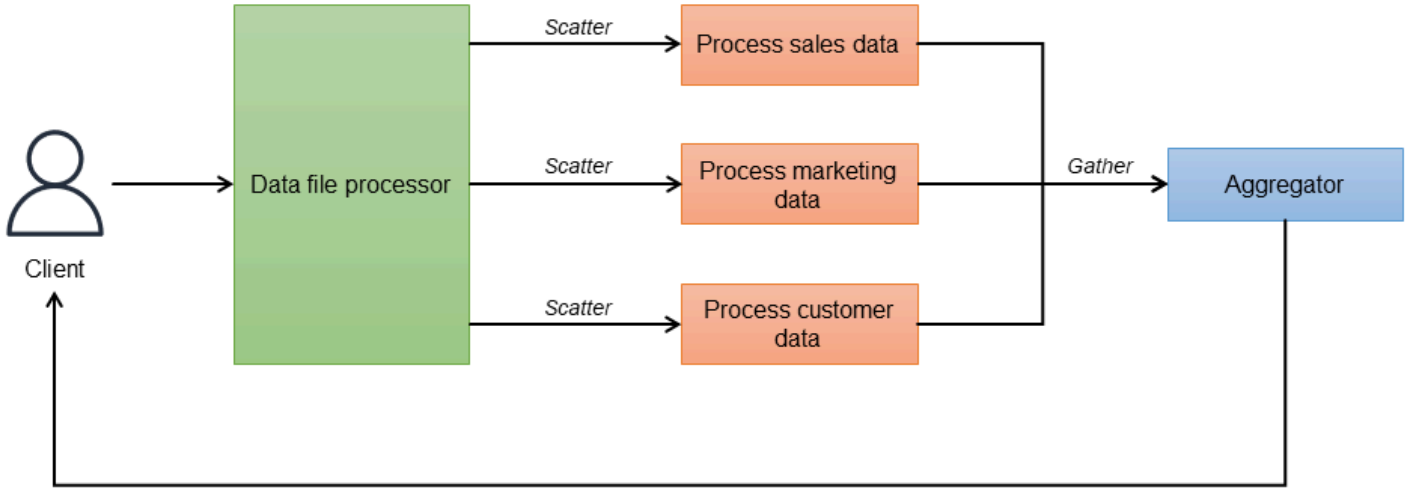
High-level architecture

The scatter-gather pattern uses a root controller to distribute requests to recipients that will process the requests. During the scatter phase, this pattern can use two mechanisms to send messages to recipients:

- Scatter by distribution: The application has a known list of recipients that must be called to get the results. The recipients can be different processes that have unique functions or a single process that has been scaled out to distribute the processing load. If any of the processing nodes time out or show delays in responding, the controller can redistribute processing to another node.
- Scatter by auction: The application broadcasts the message to interested recipients by using a publish-subscribe pattern. In this case, recipients can subscribe to the message or withdraw from the subscription at any point.

Scatter by distribution

In the scatter by distribution method, the root controller divides the incoming request into independent tasks and assigns them to available recipients (the *scatter* phase). Each recipient (process, container, or Lambda function) works independently and in parallel on its computation, and produces a portion of the response. When the recipients complete their tasks, they send their responses to an aggregator (the *gather* phase). The aggregator combines the partial responses and returns the final result to the client. The following diagram illustrates this workflow.

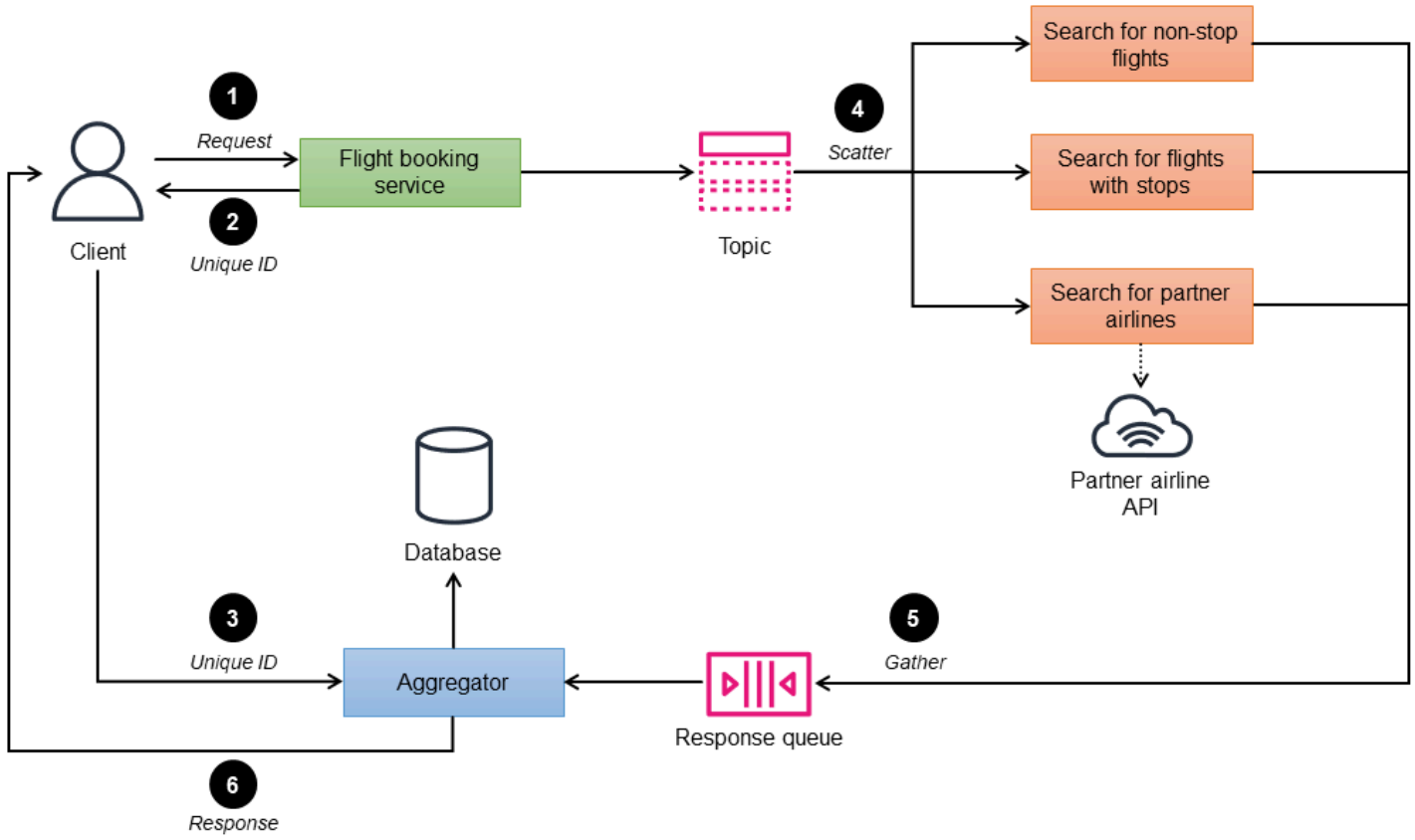


The controller (data file processor) orchestrates the entire set of invocations, and is aware of all the booking endpoints to call. It can configure a timeout parameter to ignore responses that take too long. When the requests have been sent, the aggregator waits for the responses back from each endpoint. To implement resilience, each microservice can be deployed with multiple instances for load balancing. The aggregator gets the results, combines them into a single response message, and removes duplicate data before further processing. The responses that time out are ignored. The controller can also act as an aggregator instead of using a separate aggregator service.

Scatter by auction

If the controller isn't aware of the recipients or the recipients are loosely coupled, you can use the scatter by auction method. In this method, the recipients subscribe to a topic and the controller publishes the request to the topic. Recipients publish the results to a response queue. Because the root controller isn't aware of the recipients, the gathering process uses an aggregator (another messaging pattern) to collect the responses and distill them into a single response message. The aggregator uses a unique ID to identify a group of requests.

For example, in the following diagram, the scatter by auction method is used to implement a flight booking service for an airline's website. The website allows users to search and display flights from the airline's own carrier and its partners' carriers, and must display the status of the search in real time. The flight booking service consists of three search microservices: non-stop flights, flights with stops, and partner airlines. The partner airline search calls the partner's API endpoints to get the responses.



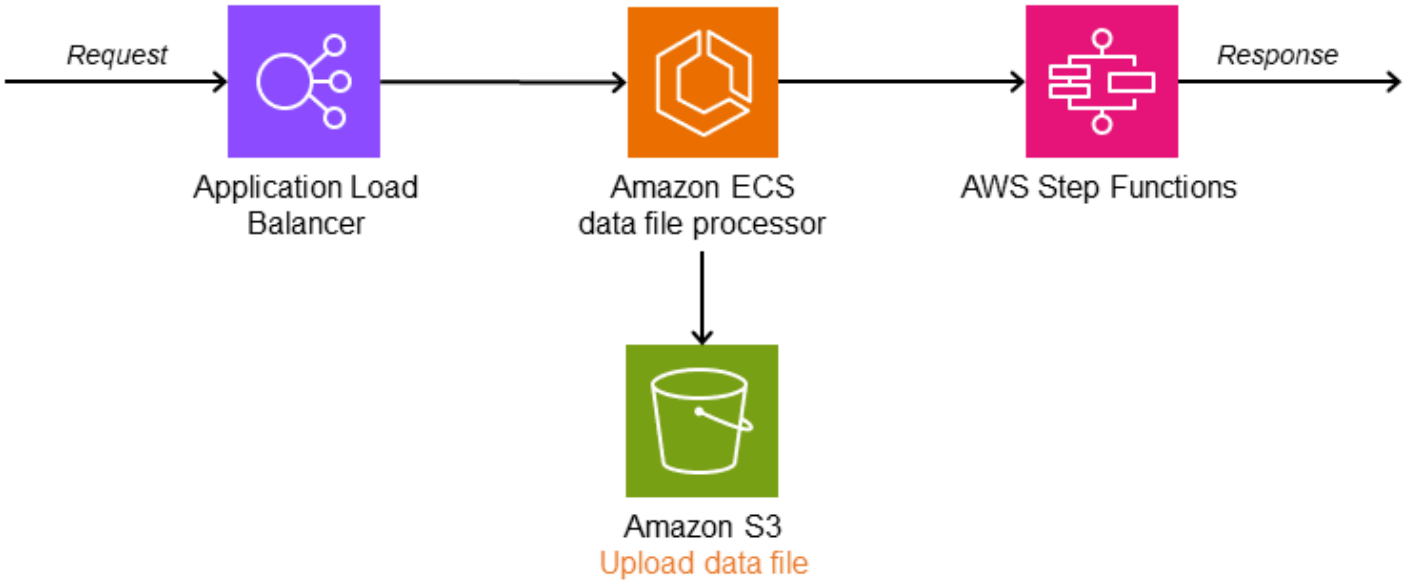
1. The flight booking service (controller) takes the search criteria as input from the client, and processes and publishes the request to the topic.
2. The controller uses a unique ID to identify each group of requests.
3. The client sends the unique ID to the aggregator for step 6.
4. The booking search microservices that have subscribed to the booking topic receive the request.
5. The microservices process the request and return seat availability for the given search criteria to a response queue.
6. The aggregator collates all the response messages that are stored in a temporary database, groups the flights by unique ID, creates a single unified response, and sends it back to the client.

Implementation using AWS services

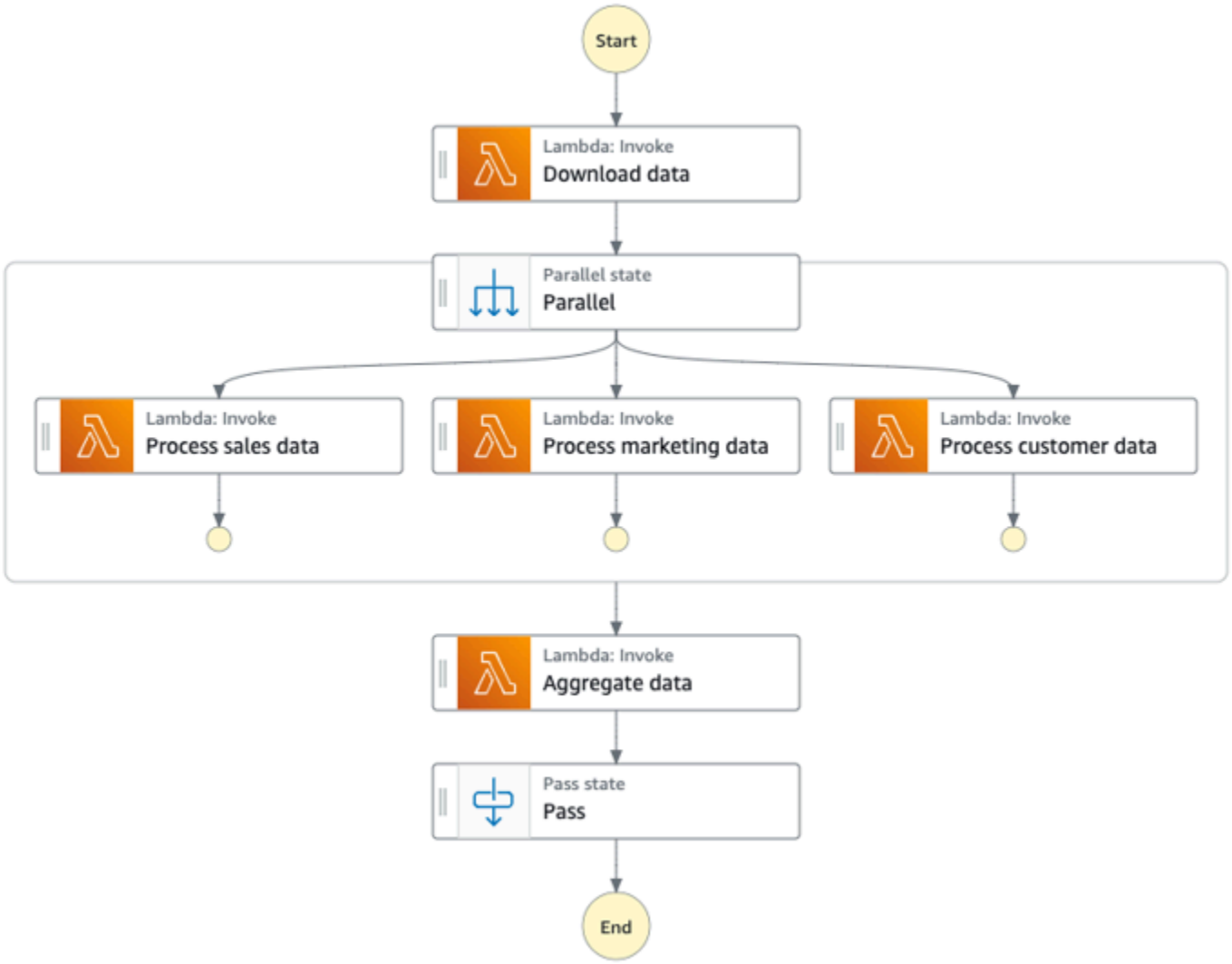
Scatter by distribution

In the following architecture, the root controller is a data file processor (Amazon ECS) that splits the incoming request data into individual Amazon Simple Storage Service (Amazon S3) buckets

and starts an AWS Step Functions workflow. The workflow downloads the data and initiates parallel file processing. The `Parallel` state waits for all the tasks to return a response. An AWS Lambda function aggregates the data and saves it back to Amazon S3.

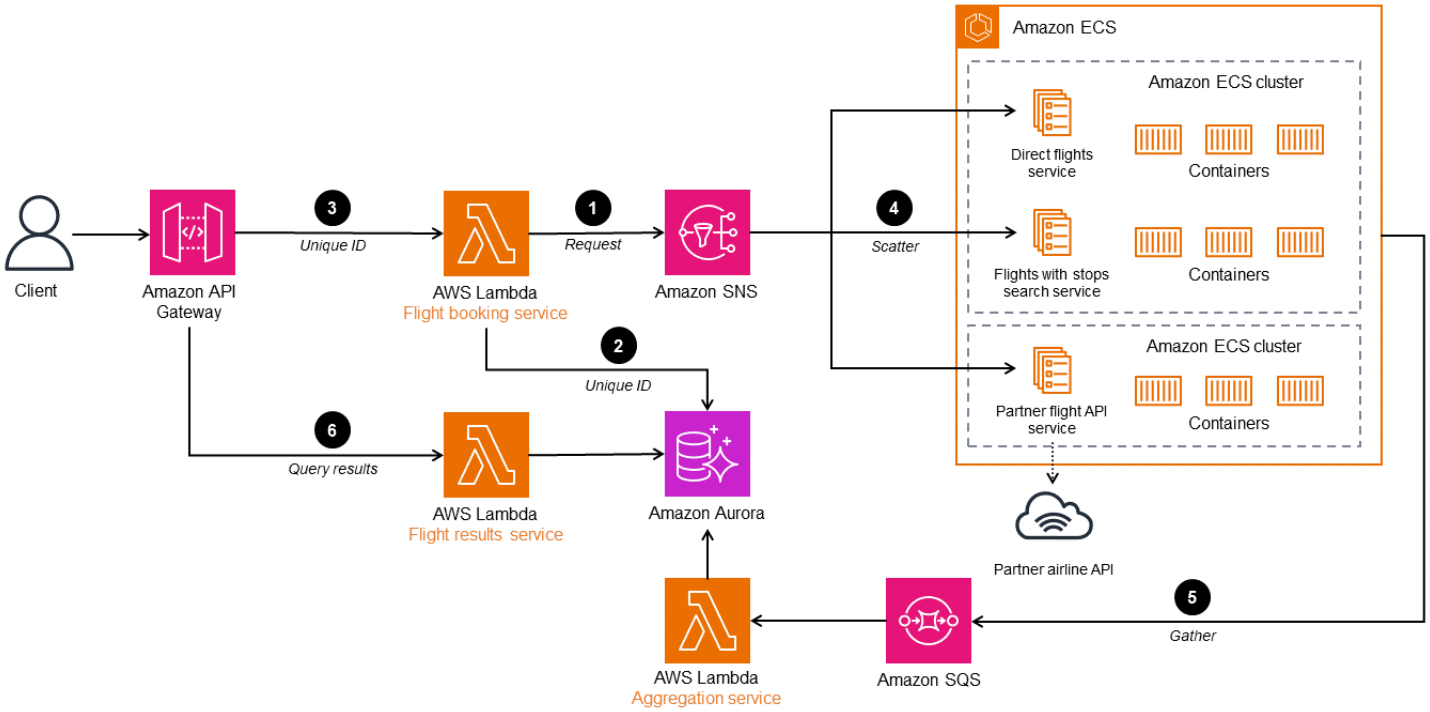


The following diagram illustrates the Step Functions workflow with the `Parallel` state.



Scatter by auction

The following diagram shows an AWS architecture for the scatter by auction method. The root controller **flight booking service** scatters the flight search request to multiple microservices. A publish-subscribe channel is implemented with Amazon Simple Notification Service (Amazon SNS), which is a managed messaging service for communications. Amazon SNS supports messages between decoupled microservice applications or direct communications to users. You can deploy the recipient microservices on Amazon Elastic Kubernetes Service (Amazon EKS) or Amazon Elastic Container Service (Amazon ECS) for better management and scalability. **The flight results service** returns the results to the client. It can be implemented in AWS Lambda or other container orchestration services such as Amazon ECS or Amazon EKS.



1. The flight booking service (controller) takes the search criteria as input from the client, and processes and publishes the request to the SNS topic.
2. The controller publishes the unique ID to an Amazon Aurora database to identify the request.
3. The client sends the unique ID to the client for step 6.
4. The booking search microservices that have subscribed to the booking topic receive the request.
5. The microservices process the request and return seat availability for the given search criteria to a response queue in Amazon Simple Queue Service (Amazon SQS). The aggregator collates all the response messages and stores them in a temporary database.
6. The flight results service groups the flights by unique ID, creates a single unified response, and sends it back to the client.

If you want to add another airline search to this architecture, you add a microservice that subscribes to the SNS topic and publishes to the SQS queue.

To summarize, the scatter-gather pattern enables distributed systems to achieve efficient parallelization, reduce latency, and seamlessly handle asynchronous communication.

GitHub repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/asynchronous-messaging-workshop/tree/master/code/lab-3.

Workshop

- Scatter-gather lab in the *Decoupled Microservices* workshop

Blog references

- Application integration patterns for microservices

Related content

- Publish-subscribe pattern

Strangler fig pattern

Intent

The strangler fig pattern helps migrate a monolithic application to a microservices architecture incrementally, with reduced transformation risk and business disruption.

Motivation

Monolithic applications are developed to provide most of their functionality within a single process or container. The code is tightly coupled. As a result, application changes require thorough retesting to avoid regression issues. The changes cannot be tested in isolation, which impacts the cycle time. As the application is enriched with more features, high complexity can lead to more time spent on maintenance, increased time to market, and, consequently, slow product innovation.

When the application scales in size, it increases the cognitive load on the team and can cause unclear team ownership boundaries. Scaling individual features based on the load isn't possible—the entire application has to be scaled to support peak load. As the systems age, the technology can become obsolete, which drives up support costs. Monolithic, legacy applications follow best practices that were available at the time of development and weren't designed to be distributed.

When a monolithic application is migrated into a microservices architecture, it can be split into smaller components. These components can scale independently, can be released independently, and can be owned by individual teams. This results in a higher velocity of change, because changes are localized and can be tested and released quickly. Changes have a smaller scope of impact because components are loosely coupled and can be deployed individually.

Replacing a monolith completely with a microservices application by rewriting or refactoring the code is a huge undertaking and a big risk. A big bang migration, where the monolith is migrated in a single operation, introduces transformation risk and business disruption. While the application is being refactored, it is extremely hard or even impossible to add new features.

One way to resolve this issue is to use the strangler fig pattern, which was introduced by Martin Fowler. This pattern involves moving to microservices by gradually extracting features and creating a new application around the existing system. The features in the monolith are replaced by microservices gradually, and application users are able to use the newly migrated features

progressively. When all features are moved out to the new system, the monolithic application can be decommissioned safely.

Applicability

Use the strangler fig pattern when:

- You want to migrate your monolithic application gradually to a microservices architecture.
- A big bang migration approach is risky because of the size and complexity of the monolith.
- The business wants to add new features and cannot wait for the transformation to be complete.
- End users must be minimally impacted during the transformation.

Issues and considerations

- **Code base access:** To implement the strangler fig pattern, you must have access to the monolith application's code base. As features are migrated out of the monolith, you will need to make minor code changes and implement an anti-corruption layer within the monolith to route calls to new microservices. You cannot intercept calls without code base access. Code base access is also critical for redirecting incoming requests—some code refactoring might be required so that the proxy layer can intercept the calls for migrated features and route them to microservices.
- **Unclear domain:** The premature decomposition of systems can be costly, especially when the domain isn't clear, and it's possible to get the service boundaries wrong. Domain-driven design (DDD) is a mechanism for understanding the domain, and event storming is a technique for determining domain boundaries.
- **Identifying microservices:** You can use DDD as a key tool for identifying microservices. To identify microservices, look for the natural divisions between service classes. Many services will own their own data access object and will decouple easily. Services that have related business logic and classes that have no or few dependencies are good candidates for microservices. You can refactor code before breaking down the monolith to prevent tight coupling. You should also consider compliance requirements, the release cadence, the geographical location of the teams, scaling needs, use case-driven technology needs, and the cognitive load of teams.
- **Anti-corruption layer:** During the migration process, when the features within the monolith have to call the features that were migrated as microservices, you should implement an anti-corruption layer (ACL) that routes each call to the appropriate microservice. In order to decouple and prevent changes to existing callers within the monolith, the ACL works as an adapter

or a facade that converts the calls into the newer interface. This is discussed in detail in the Implementation section of the ACL pattern earlier in this guide.

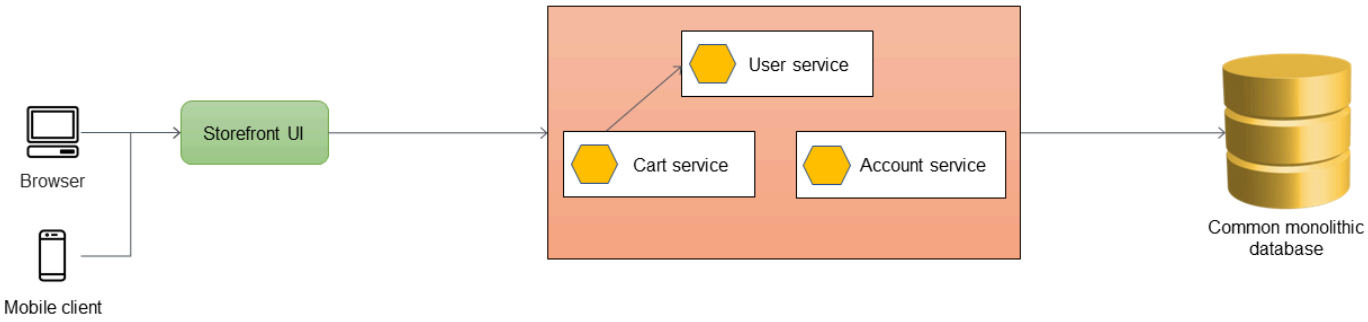
- **Proxy layer failure:** During migration, a proxy layer intercepts the requests that go to the monolithic application and routes them to either the legacy system or the new system. However, this proxy layer can become a single point of failure or a performance bottleneck.
- **Application complexity:** Large monoliths benefit the most from the strangler fig pattern. For small applications, where the complexity of complete refactoring is low, it might be more efficient to rewrite the application in microservices architecture instead of migrating it.
- **Service interactions:** Microservices can communicate synchronously or asynchronously. When synchronous communication is required, consider whether the timeouts can cause connection or thread pool consumption, resulting in application performance issues. In such cases, use the circuit breaker pattern to return immediate failure for operations that are likely to fail for extended periods of time. Asynchronous communication can be achieved by using events and messaging queues.
- **Data aggregation:** In a microservices architecture, data is distributed across databases. When data aggregation is required, you can use AWS AppSync in the front end, or the command query responsibility segregation (CQRS) pattern in the backend.
- **Data consistency:** The microservices own their data store, and the monolithic application can also potentially use this data. To enable sharing, you can synchronize the new microservices' data store with the monolithic application's database by using a queue and agent. However, this can cause data redundancy and eventual consistency between two data stores, so we recommend that you treat it as a tactical solution until you can establish a long-term solution such as a data lake.

Implementation

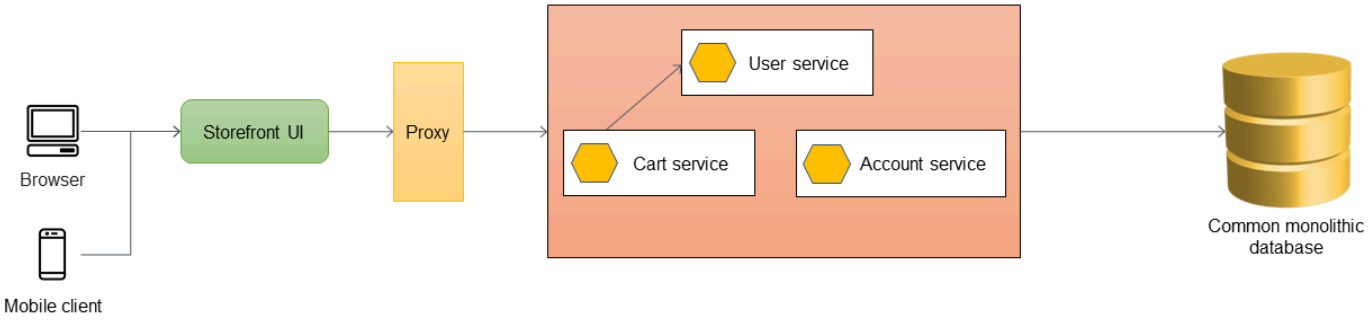
In the strangler fig pattern, you replace specific functionality with a new service or application, one component at a time. A proxy layer intercepts requests that go to the monolithic application and routes them to either the legacy system or the new system. Because the proxy layer routes users to the correct application, you can add features to the new system while ensuring that the monolith continues to function. The new system eventually replaces all the features of the old system, and you can decommission it.

High-level architecture

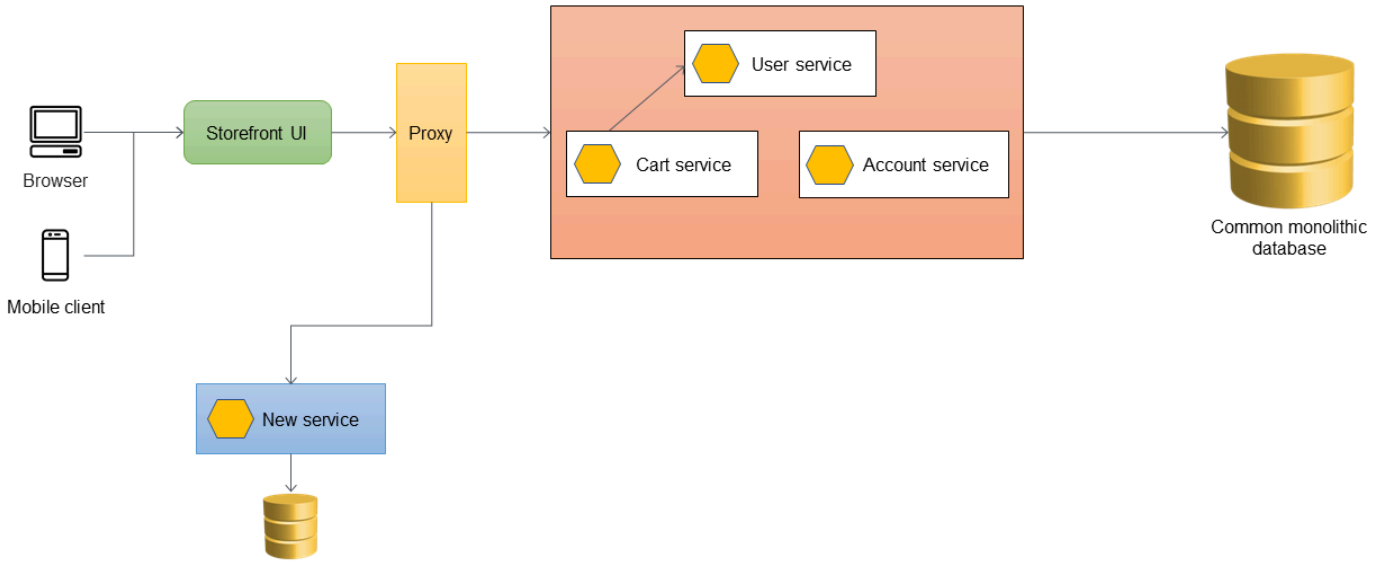
In the following diagram, a monolithic application has three services: user service, cart service, and account service. The cart service depends on the user service, and the application uses a monolithic relational database.



The first step is to add a proxy layer between the storefront UI and the monolithic application. At the start, the proxy routes all traffic to the monolithic application.

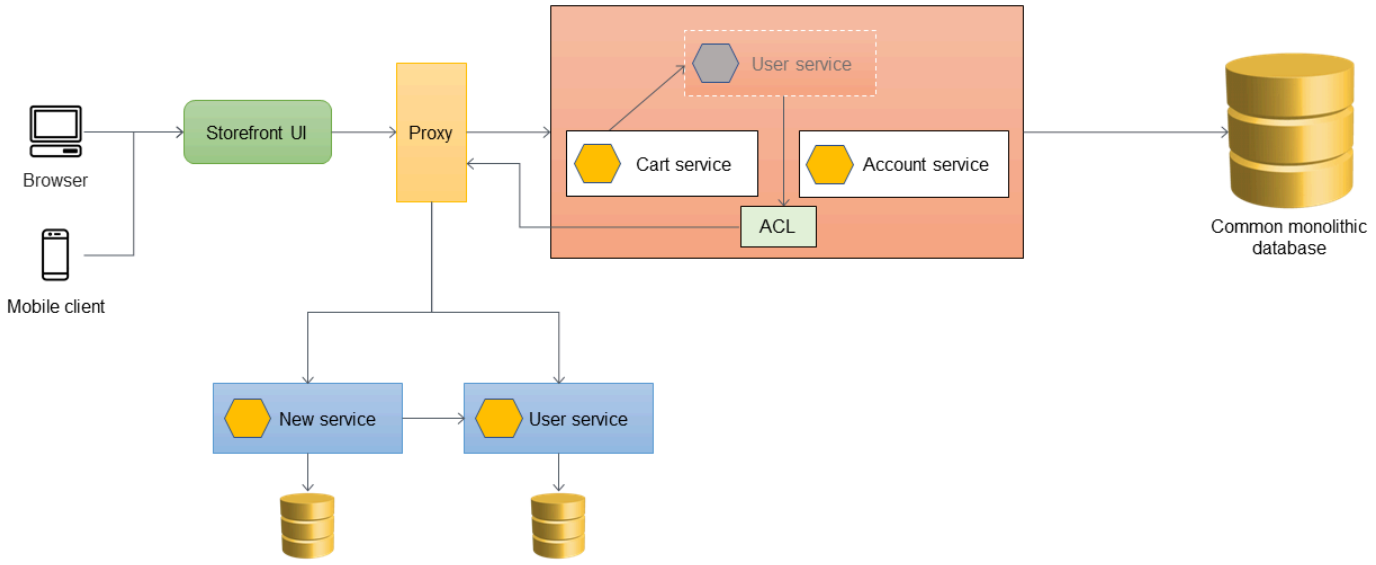


When you want to add new features to your application, you implement them as new microservices instead of adding features to the existing monolith. However, you continue to fix bugs in the monolith to ensure application stability. In the following diagram, the proxy layer routes the calls to the monolith or to the new microservice based on the API URL.



Adding an anti-corruption layer

In the following architecture, the user service has been migrated to a microservice. The cart service calls the user service, but the implementation is no longer available within the monolith. Also, the interface of the newly migrated service might not match its previous interface inside the monolithic application. To address these changes, you implement an ACL. During the migration process, when the features within the monolith need to call the features that were migrated as microservices, the ACL converts the calls to the new interface and routes them to the appropriate microservice.

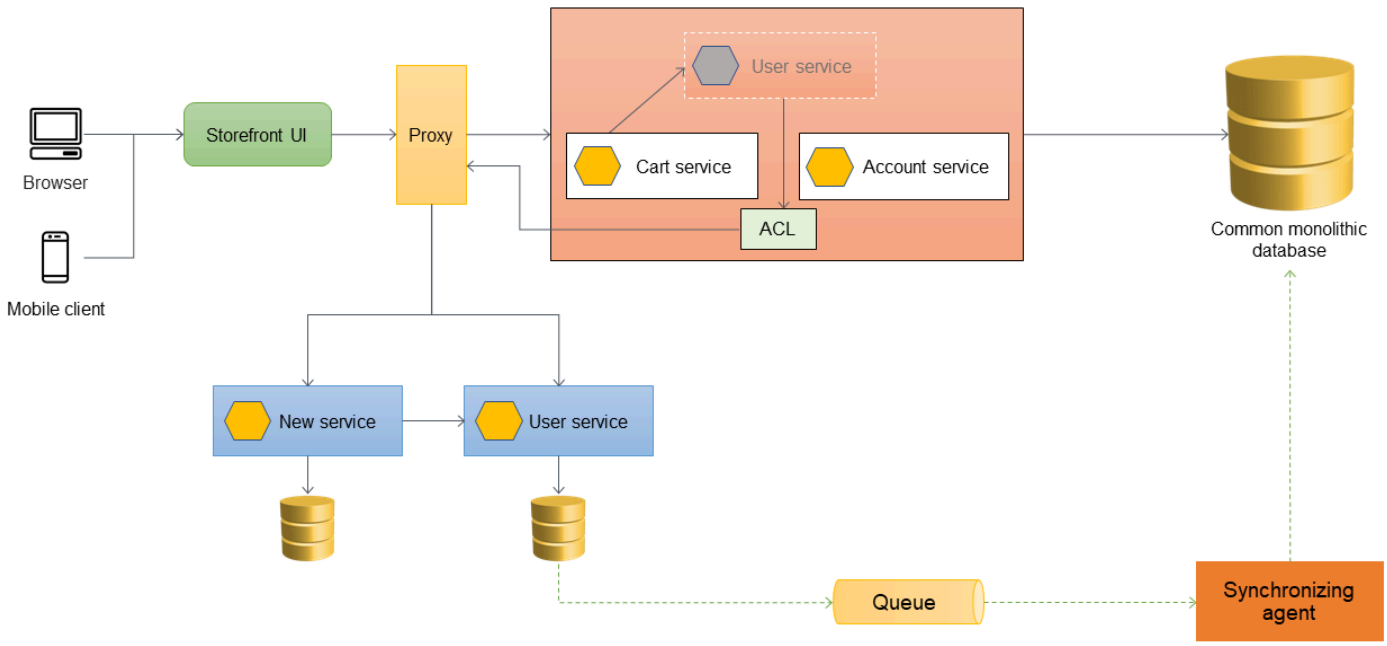


You can implement the ACL inside the monolithic application as a class that's specific to the service that was migrated; for example, `UserServiceFacade` or `UserServiceAdapter`. The ACL must be decommissioned after all dependent services have been migrated into the microservices architecture.

When you use the ACL, the cart service still calls the user service within the monolith, and the user service redirects the call to the microservice through the ACL. The cart service should still call the user service without being aware of the microservice migration. This loose coupling is required to reduce regression and business disruption.

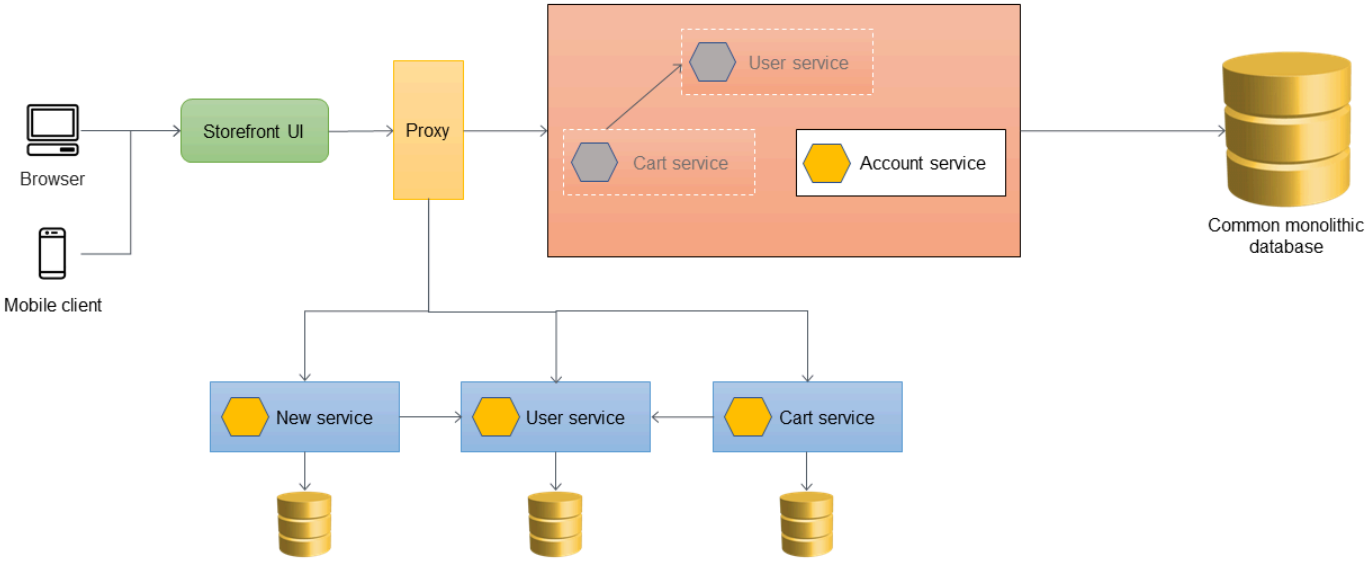
Handling data synchronization

As a best practice, the microservice should own its data. The user service stores its data in its own data store. It might need to synchronize data with the monolithic database to handle dependencies such as reporting and to support downstream applications that are not yet ready to access the microservices directly. The monolithic application might also require the data for other functions and components that haven't been migrated to microservices yet. So data synchronization is necessary between the new microservice and the monolith. To synchronize the data, you can introduce a synchronizing agent between the user microservice and the monolithic database, as shown in the following diagram. The user microservice sends an event to the queue whenever its database is updated. The synchronizing agent listens to the queue and continuously updates the monolithic database. The data in the monolithic database is eventually consistent for the data that is being synchronized.

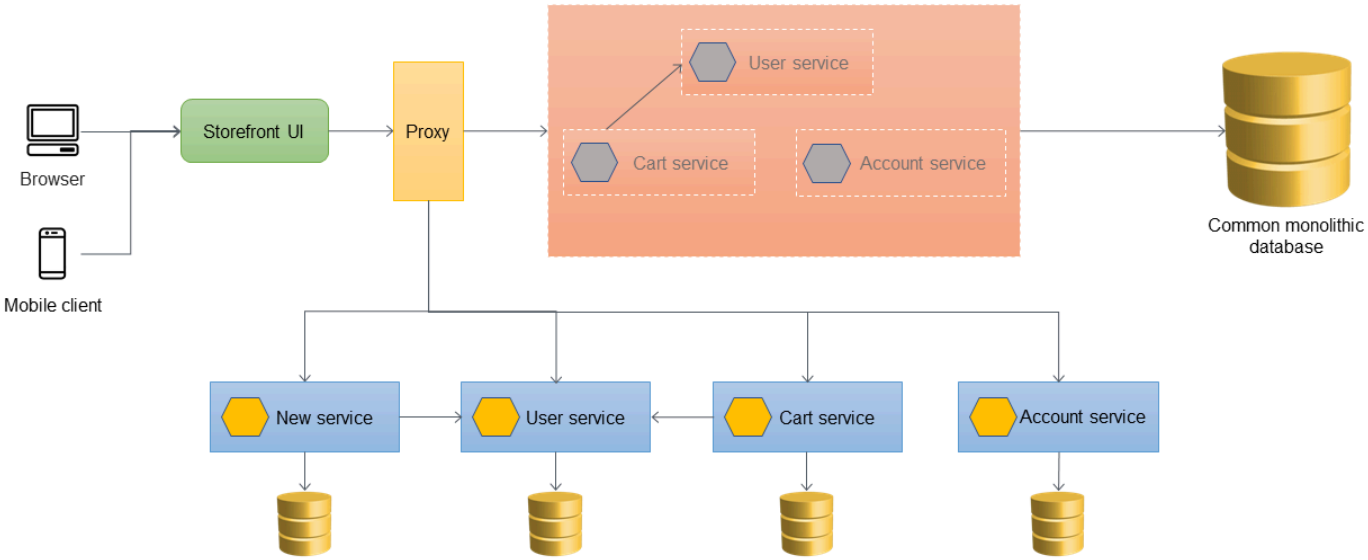


Migrating additional services

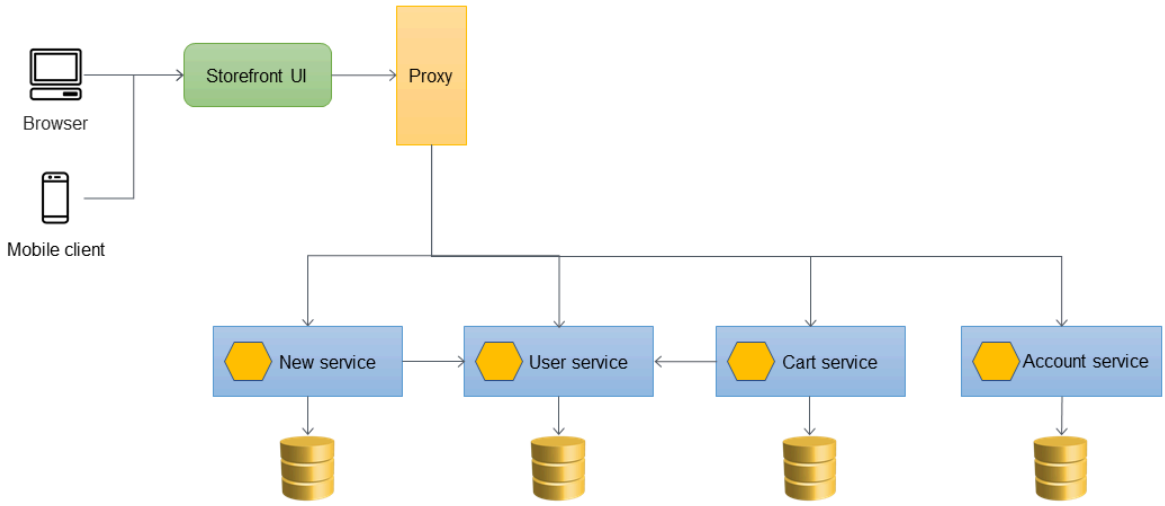
When the cart service is migrated out of the monolithic application, its code is revised to call the new service directly, so the ACL no longer routes those calls. The following diagram illustrates this architecture.



The following diagram shows the final strangled state where all services have been migrated out of the monolith and only the skeleton of the monolith remains. Historical data can be migrated to data stores owned by individual services. The ACL can be removed, and the monolith is ready to be decommissioned at this stage.



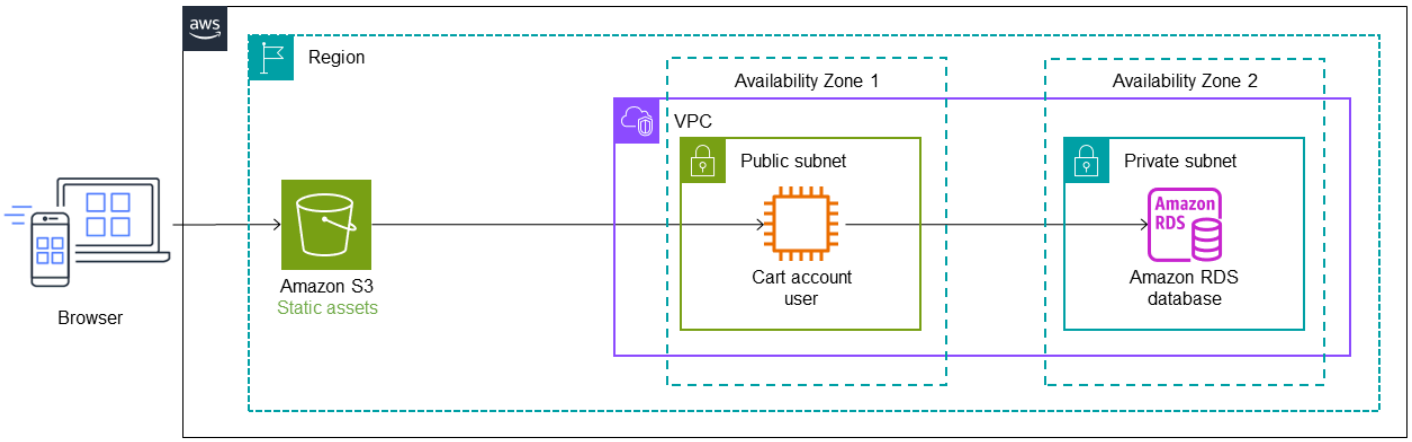
The following diagram shows the final architecture after the monolithic application has been decommissioned. You can host the individual microservices through a resource-based URL (such as `http://www.storefront.com/user`) or through their own domain (for example, `http://user.storefront.com`) based on your application's requirements. For more information about the major methods for exposing HTTP APIs to upstream consumers by using hostnames and paths, see the API routing patterns section.



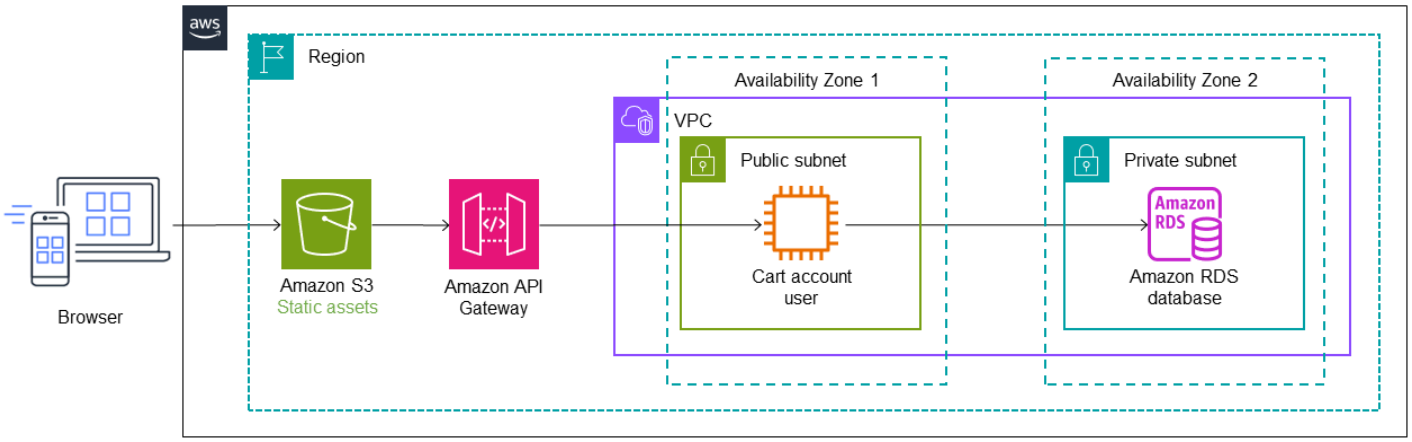
Implementation using AWS services

Using API Gateway as the application proxy

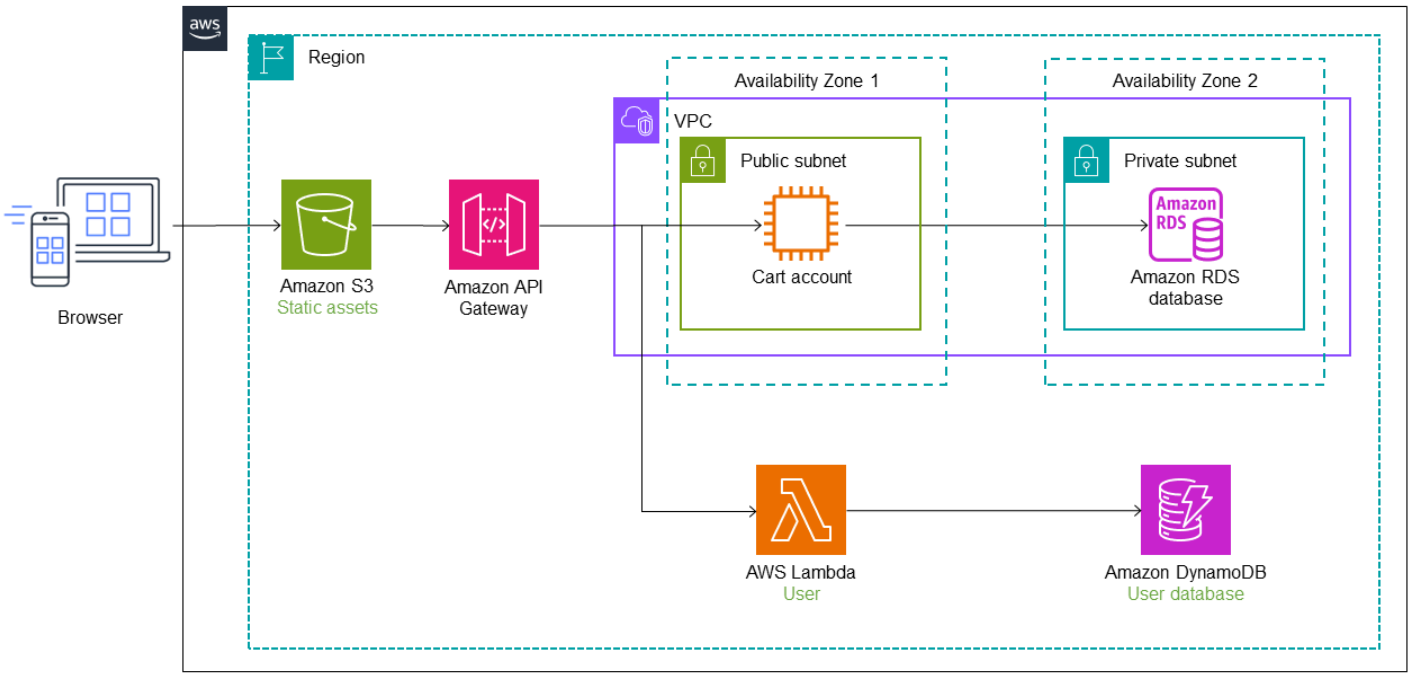
The following diagram shows the initial state of the monolithic application. Let's assume that it was migrated to AWS by using a lift-and-shift strategy, so it's running on an Amazon Elastic Compute Cloud (Amazon EC2) instance and uses an Amazon Relational Database Service (Amazon RDS) database. For simplicity, the architecture uses a single virtual private cloud (VPC) with one private and one public subnet, and let's assume that the microservices will initially be deployed within the same AWS account. (The best practice in production environments is to use a multi-account architecture to ensure deployment independence.) The EC2 instance resides in a single Availability Zone in the public subnet, and the RDS instance resides in a single Availability Zone in the private subnet. Amazon Simple Storage Service (Amazon S3) stores static assets such as the JavaScript, CSS, and React files for the website.



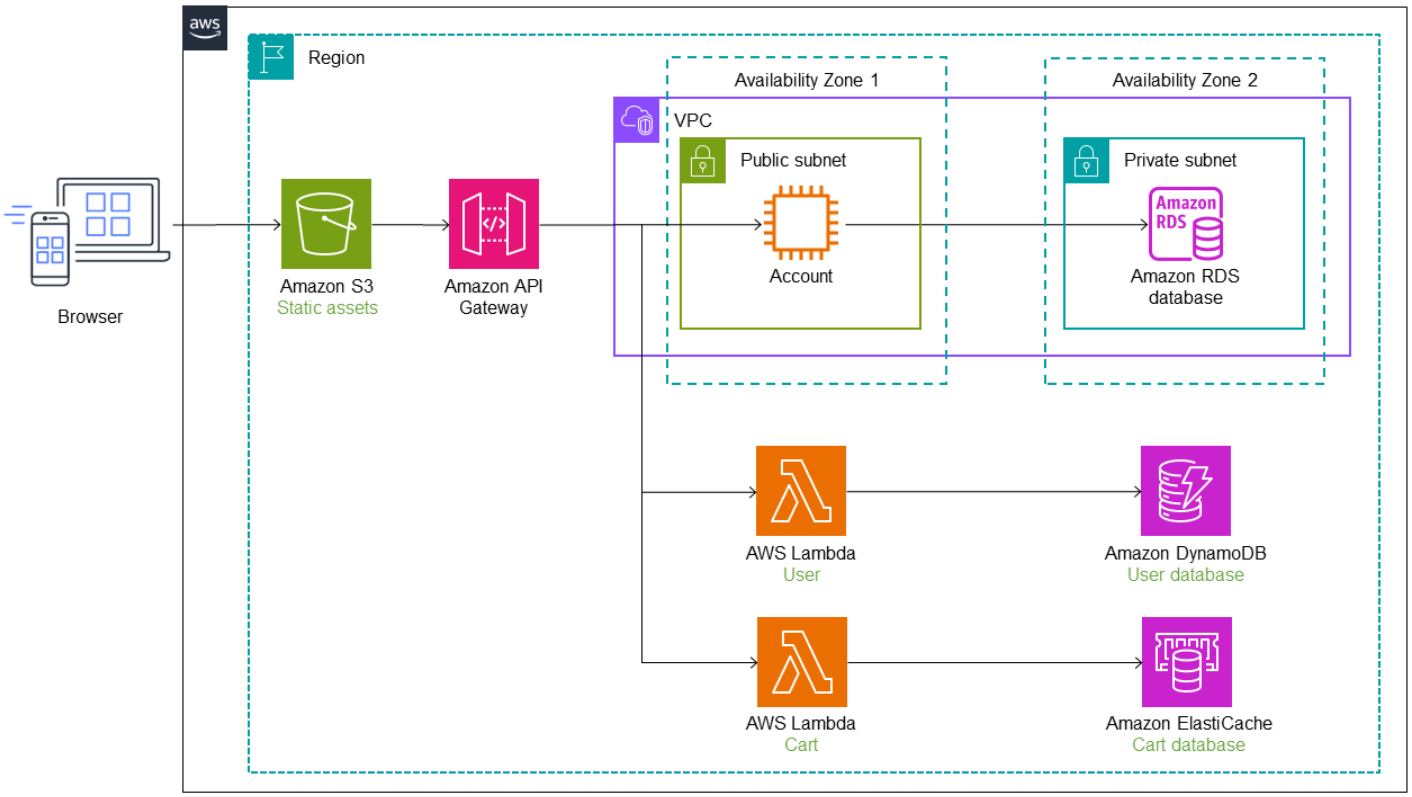
In the following architecture, AWS Migration Hub Refactor Spaces deploys Amazon API Gateway in front of the monolithic application. Refactor Spaces creates a refactoring infrastructure inside your account, and API Gateway acts as the proxy layer for routing calls to the monolith. Initially, all calls are routed to the monolithic application through the proxy layer. As discussed earlier, proxy layers can become a single point of failure. However, using API Gateway as the proxy mitigates the risk because it is a serverless, Multi-AZ service.



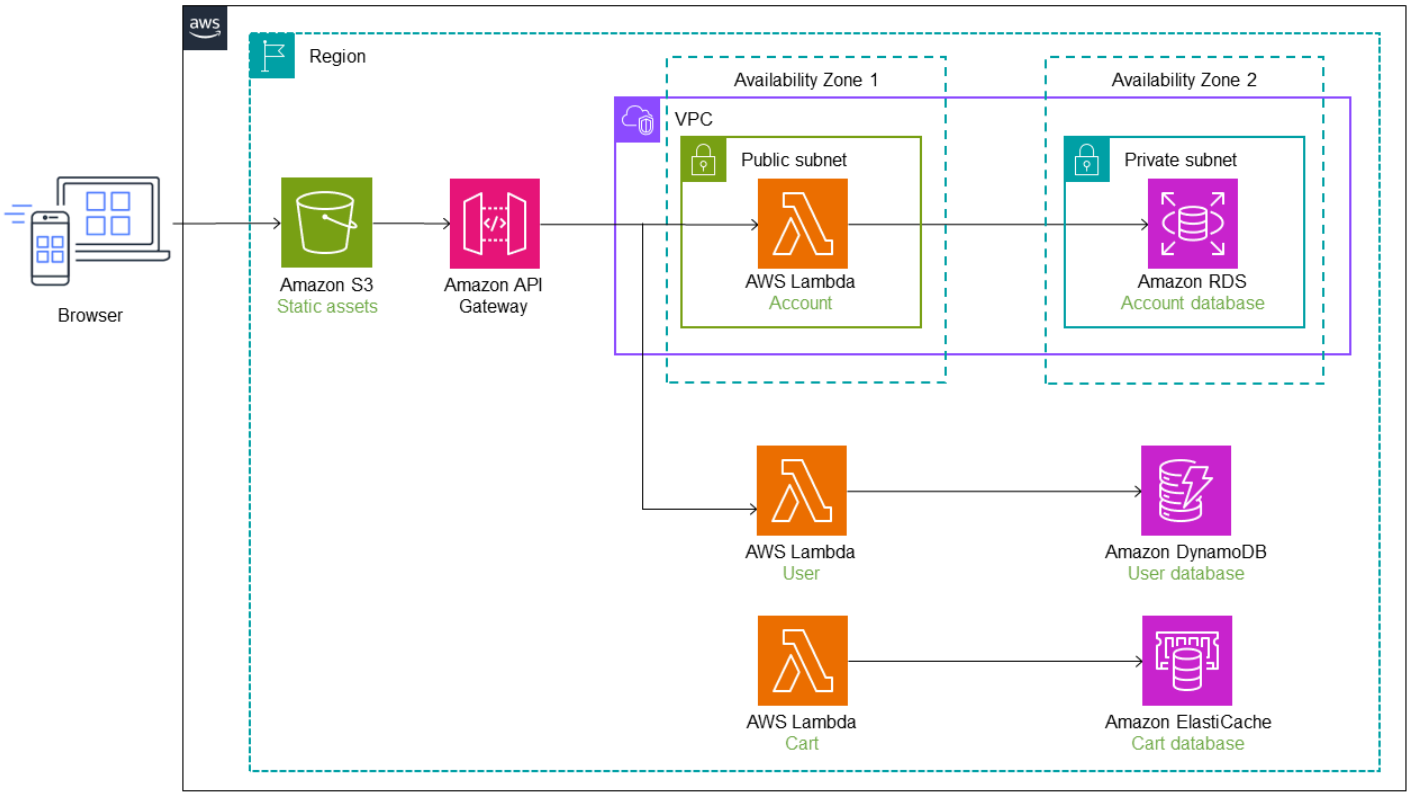
The user service is migrated into a Lambda function, and an Amazon DynamoDB database stores its data. A Lambda service endpoint and default route are added to Refactor Spaces, and API Gateway is automatically configured to route the calls to the Lambda function. For implementation details, see Module 2 in the Iterative App Modernization Workshop.



In the following diagram, the cart service has also been migrated out of the monolith and into a Lambda function. An additional route and service endpoint are added to Refactor Spaces, and traffic automatically cuts over to the `Cart` Lambda function. The data store for the Lambda function is managed by Amazon ElastiCache. The monolithic application still remains in the EC2 instance along with the Amazon RDS database.



In the next diagram, the last service (account) is migrated out of the monolith into a Lambda function. It continues to use the original Amazon RDS database. The new architecture now has three microservices with separate databases. Each service uses a different type of database. This concept of using purpose-built databases to meet the specific needs of microservices is called *polyglot persistence*. The Lambda functions can also be implemented in different programming languages, as determined by the use case. During refactoring, Refactor Spaces automates the cutover and routing of traffic to Lambda. This saves your builders the time needed to architect, deploy, and configure the routing infrastructure.

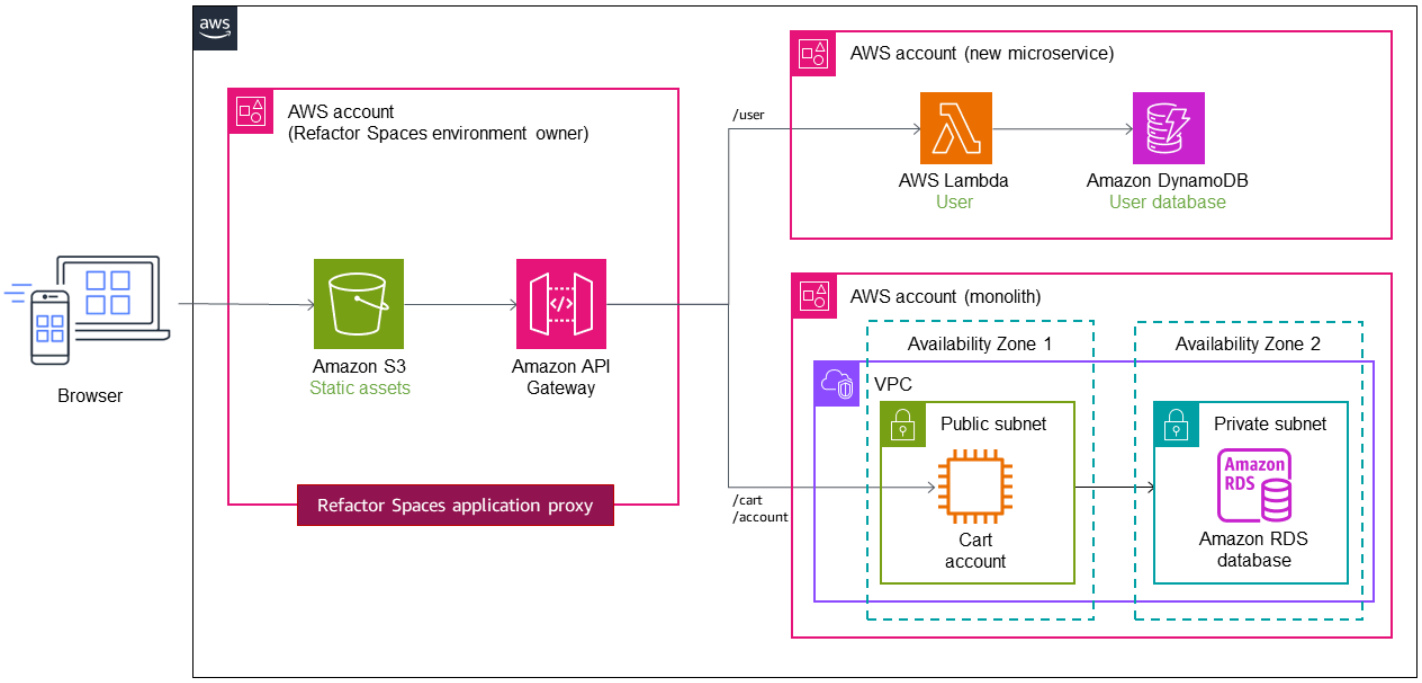


Using multiple accounts

In the previous implementation, we used a single VPC with a private and a public subnet for the monolithic application, and we deployed the microservices within the same AWS account for the sake of simplicity. However, this is rarely the case in real-world scenarios, where microservices are often deployed in multiple AWS accounts for deployment independence. In a multi-account structure, you need to configure routing traffic from the monolith to the new services in different accounts.

Refactor Spaces helps you create and configure the AWS infrastructure for routing API calls away from the monolithic application. Refactor Spaces orchestrates API Gateway, Network Load Balancer, and resource-based AWS Identity and Access Management (IAM) policies inside your AWS accounts as part of its application resource. You can transparently add new services in a single AWS account or across multiple accounts to an external HTTP endpoint. All of these resources are orchestrated inside your AWS account and can be customized and configured after deployment.

Let's assume that the user and cart services are deployed to two different accounts, as shown in the following diagram. When you use Refactor Spaces, you only need to configure the service endpoint and the route. Refactor Spaces automates the API Gateway–Lambda integration and the creation of Lambda resource policies, so you can focus on safely refactoring services off the monolith.



For a video tutorial on using Refactor Spaces, see Refactor Apps Incrementally with AWS Migration Hub Refactor Spaces.

Workshop

- Iterative app modernization workshop

Blog references

- AWS Migration Hub Refactor Spaces
- Deep Dive on an AWS Migration Hub Refactor Spaces
- Deployment Pipelines Reference Architecture and Reference Implementations

Related content

- API routing patterns
- Refactor Spaces documentation

Transactional outbox pattern

Intent

The transactional outbox pattern resolves the dual write operations issue that occurs in distributed systems when a single operation involves both a database write operation and a message or event notification. A dual write operation occurs when an application writes to two different systems; for example, when a microservice needs to persist data in the database and send a message to notify other systems. A failure in one of these operations might result in inconsistent data.

Motivation

When a microservice sends an event notification after a database update, these two operations should run atomically to ensure data consistency and reliability.

- If the database update is successful but the event notification fails, the downstream service will not be aware of the change, and the system can enter an inconsistent state.
- If the database update fails but the event notification is sent, data could get corrupted, which might affect the reliability of the system.

Applicability

Use the transactional outbox pattern when:

- You're building an event-driven application where a database update initiates an event notification .
- You want to ensure atomicity in operations that involve two services.
- You want to implement the event sourcing pattern.

Issues and considerations

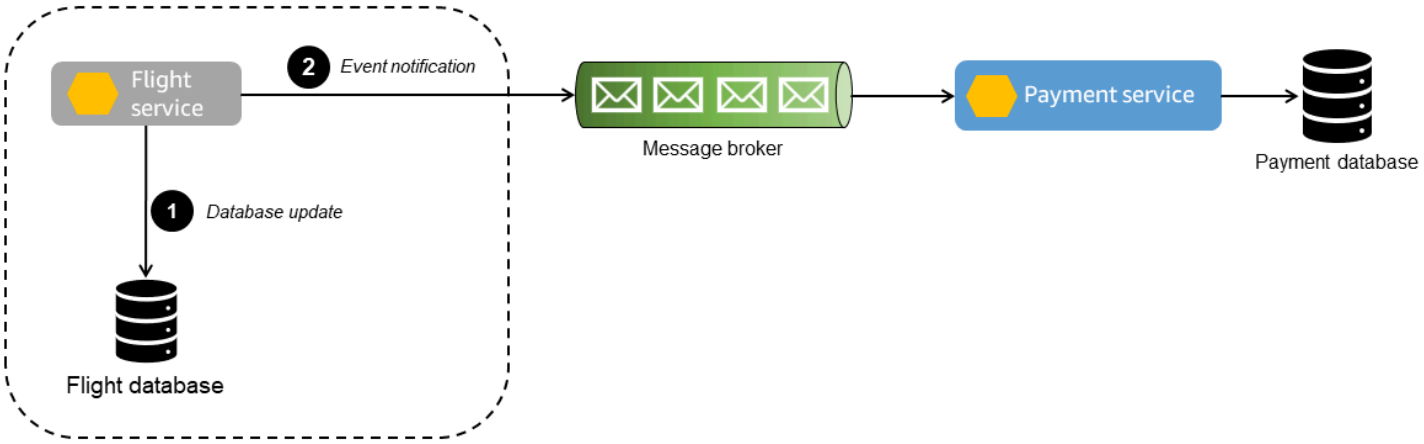
- **Duplicate messages**: The events processing service might send out duplicate messages or events, so we recommend that you make the consuming service idempotent by tracking the processed messages.

- **Order of notification**: Send messages or events in the same order in which the service updates the database. This is critical for the event sourcing pattern where you can use an event store for point-in-time recovery of the data store. If the order is incorrect, it might compromise the quality of the data. Eventual consistency and database rollback can compound the issue if the order of notifications isn't preserved.
- **Transaction rollback**: Do not send out an event notification if the transaction is rolled back.
- **Service-level transaction handling**: If the transaction spans services that require data store updates, use the saga orchestration pattern to preserve data integrity across the data stores.

Implementation

High-level architecture

The following sequence diagram shows the order of events that happen during dual write operations.



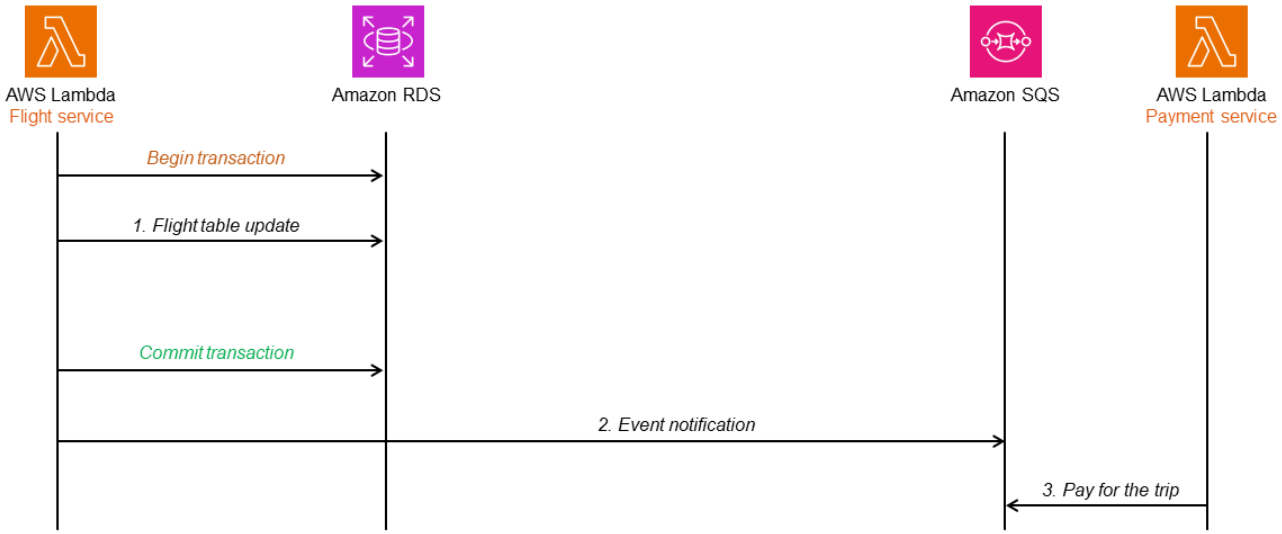
1. The flight service writes to the database and sends out an event notification to the payment service.
2. The message broker carries the messages and events to the payment service. Any failure in the message broker prevents the payment service from receiving the updates.

If the flight database update fails but the notification is sent out, the payment service will process the payment based on the event notification. This will cause downstream data inconsistencies.

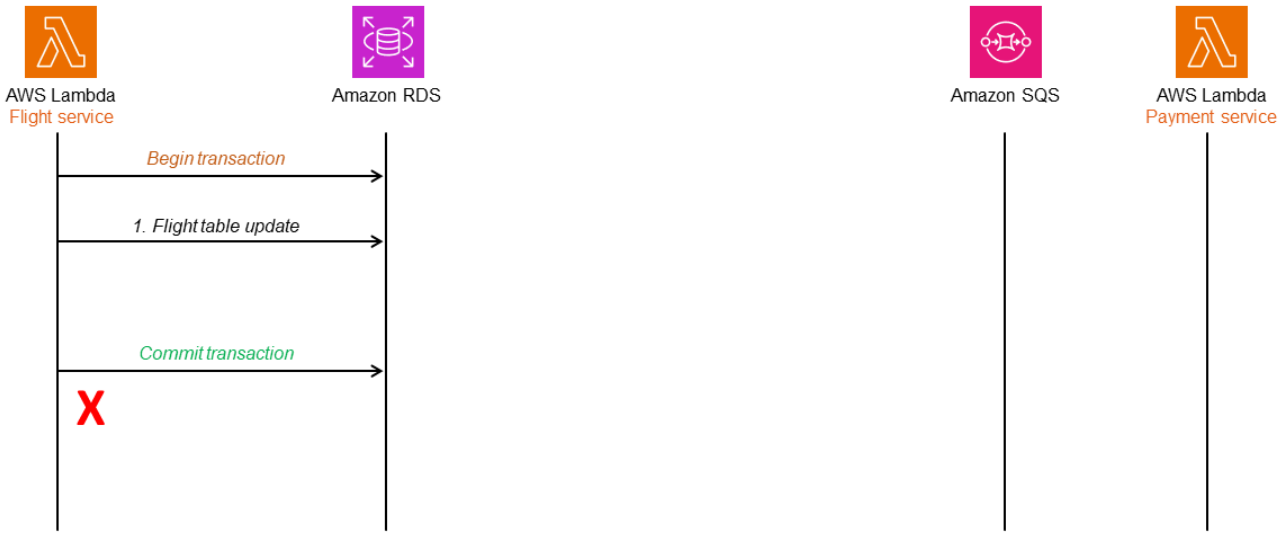
Implementation using AWS services

To demonstrate the pattern in the sequence diagram, we will use the following AWS services, as shown in the following diagram.

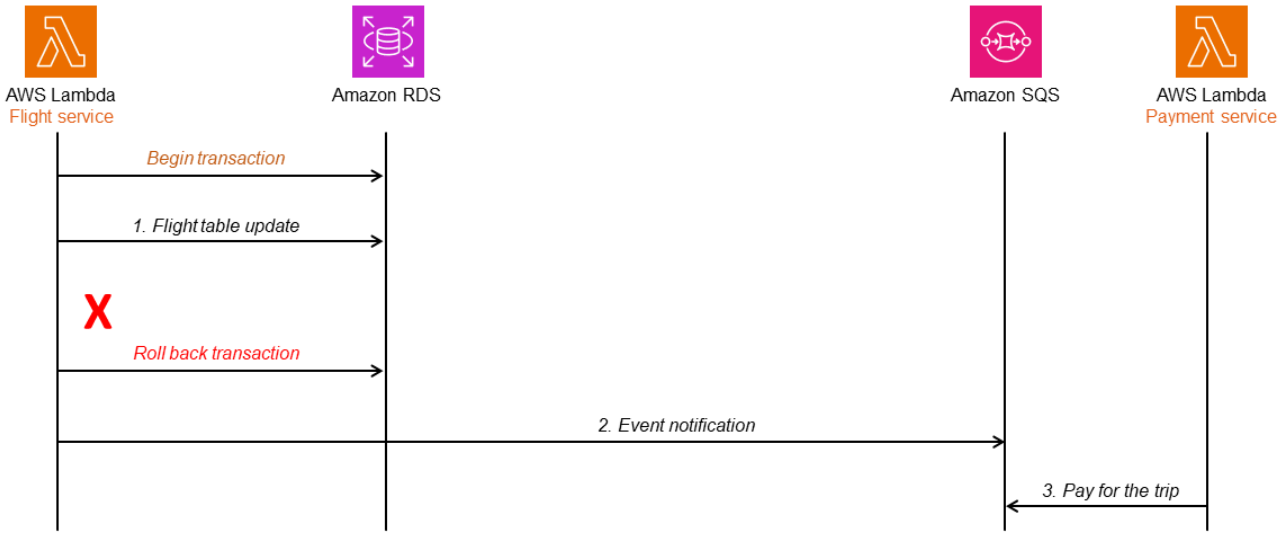
- Microservices are implemented by using AWS Lambda.
- The primary database is managed by Amazon Relational Database Service (Amazon RDS) .
- Amazon Simple Queue Service (Amazon SQS) acts as the message broker that receives event notifications.



If the flight service fails after committing the transaction, this might result in the event notification not being sent.



However, the transaction could fail and roll back, but the event notification might still be sent, causing the payment service to process the payment.



To address this problem, you can use an outbox table or change data capture (CDC). The following sections discuss these two options and how you can implement them by using AWS services.

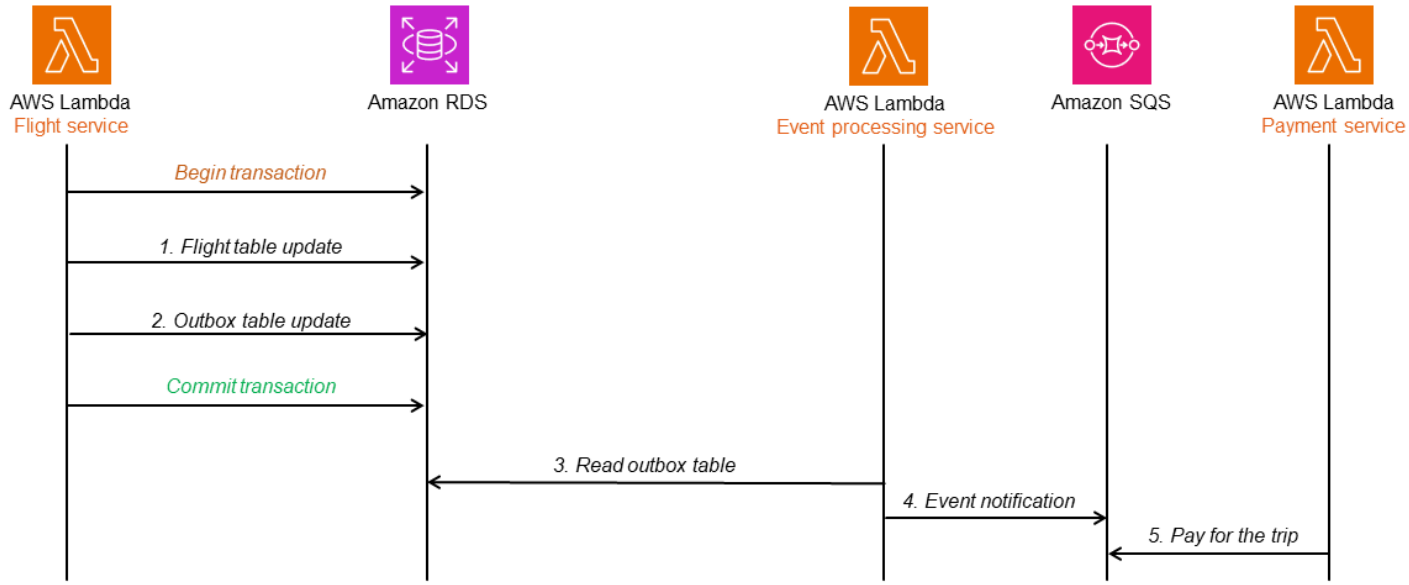
Using an outbox table with a relational database

An outbox table stores all the events from the flight service with a timestamp and a sequence number.

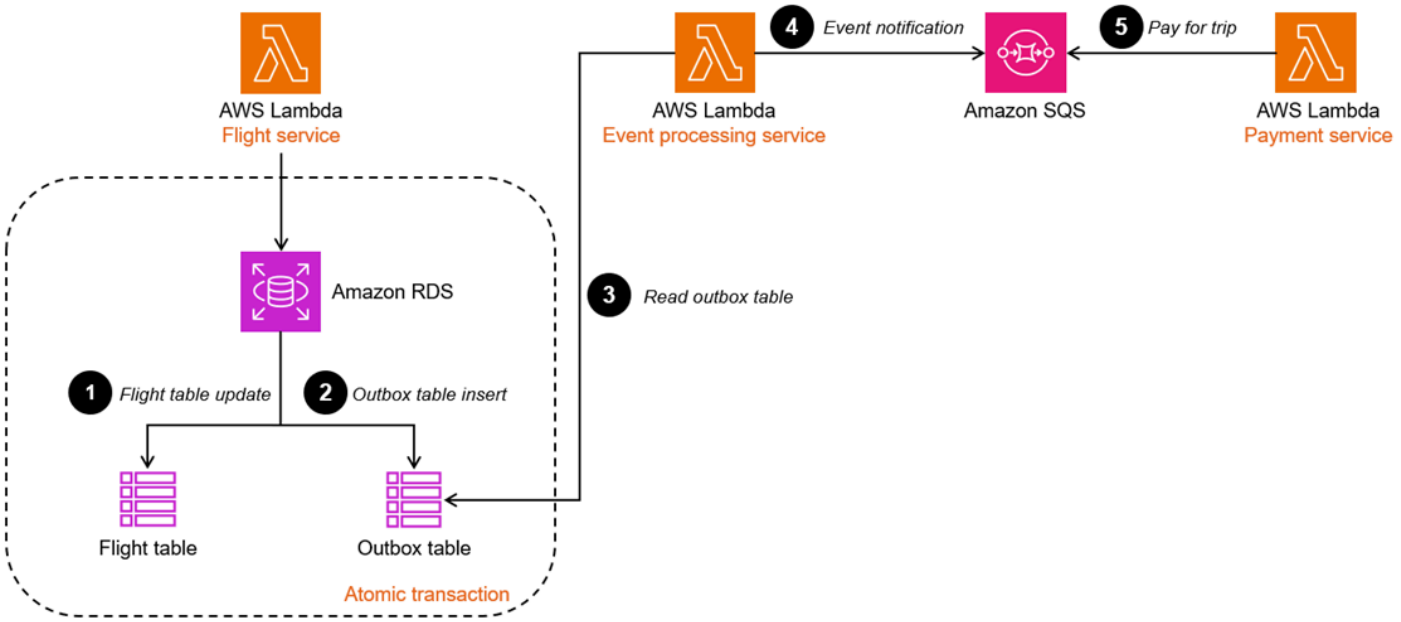
When the flight table is updated, the outbox table is also updated in the same transaction. Another service (for example, the event processing service) reads from the outbox table and sends the event

to Amazon SQS. Amazon SQS sends a message about the event to the payment service for further processing. Amazon SQS standard queues guarantee that the message is delivered at least once and doesn't get lost. However, when you use Amazon SQS standard queues, the same message or event might be delivered more than once, so you should ensure that the event notification service is idempotent (that is, processing the same message multiple times shouldn't have an adverse effect). If you require the message to be delivered exactly once, with message ordering, you can use Amazon SQS first in, first out (FIFO) queues.

If the flight table update fails or the outbox table update fails, the entire transaction is rolled back, so there are no downstream data inconsistencies.



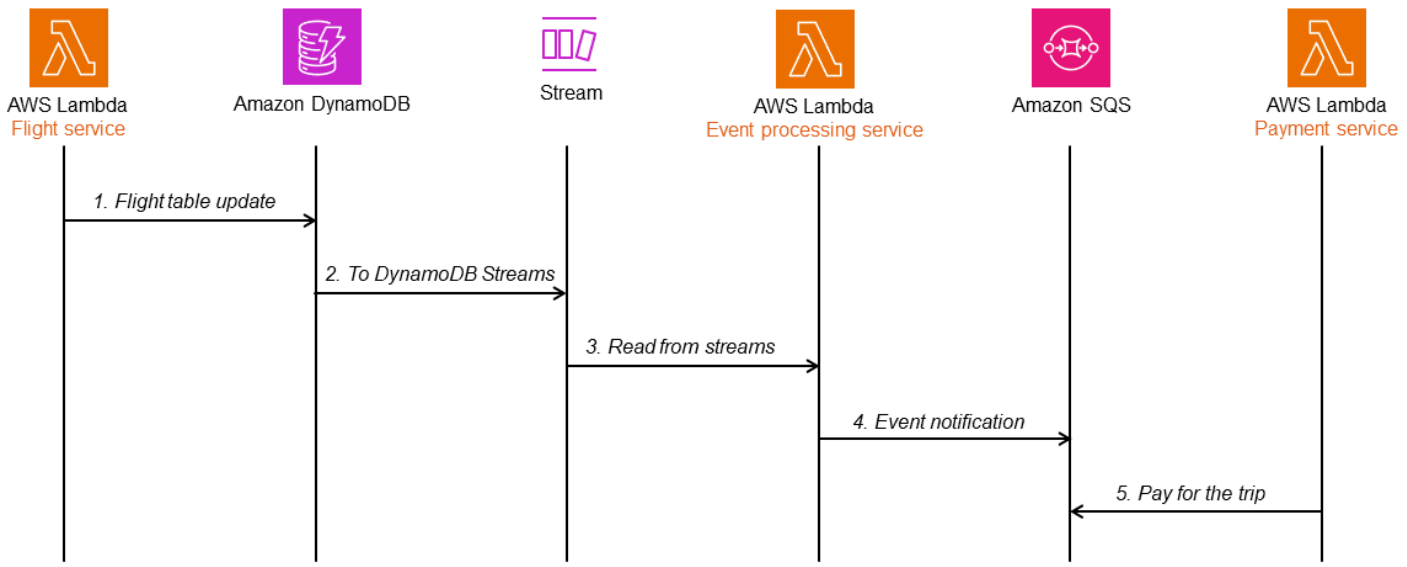
In the following diagram, the transactional outbox architecture is implemented by using an Amazon RDS database. When the events processing service reads the outbox table, it recognizes only those rows that are part of a committed (successful) transaction, and then places the message for the event in the SQS queue, which is read by the payment service for further processing. This design resolves the dual write operations issue and preserves the order of messages and events by using timestamps and sequence numbers.



Using change data capture (CDC)

Some databases support the publishing of item-level modifications to capture changed data. You can identify the changed items and send an event notification accordingly. This saves the overhead of creating another table to track the updates. The event initiated by the flight service is stored in another attribute of the same item.

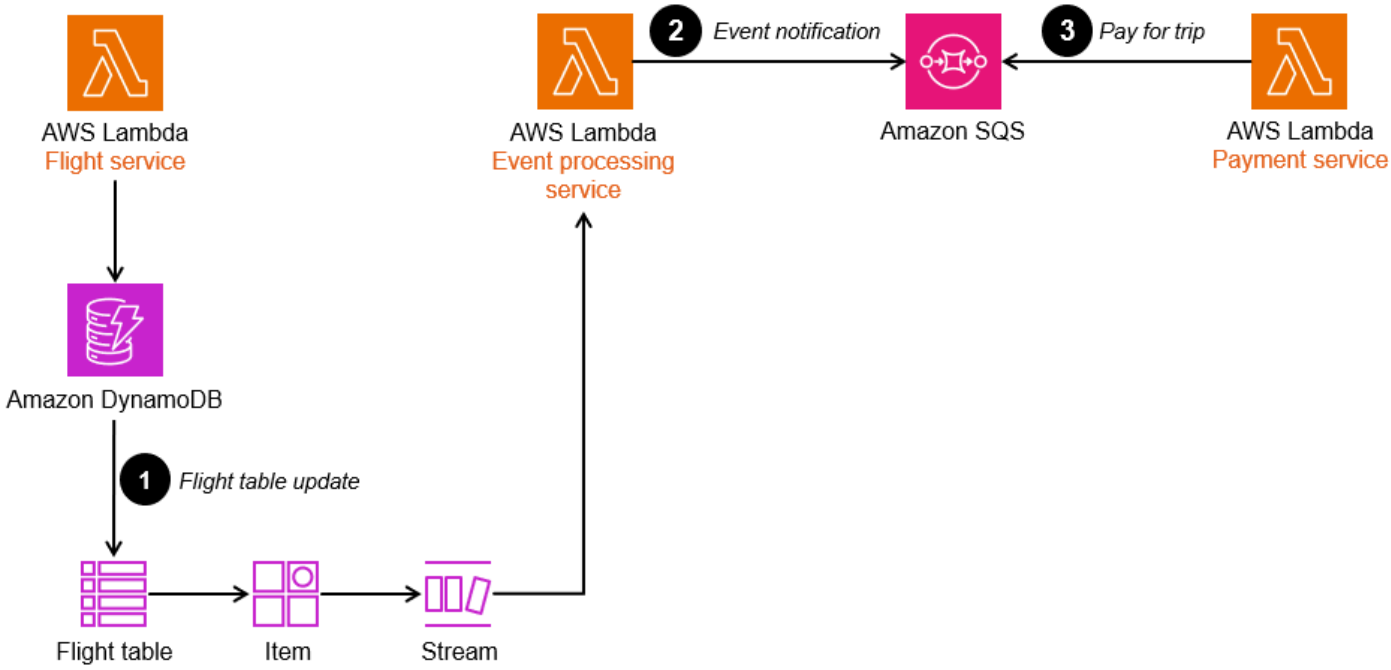
Amazon DynamoDB is a key-value NoSQL database that supports CDC updates. In the following sequence diagram, DynamoDB publishes item-level modifications to Amazon DynamoDB Streams. The event processing service reads from the streams and publishes the event notification to the payment service for further processing.



DynamoDB Streams captures the flow of information relating to item-level changes in a DynamoDB table by using a time-ordered sequence.

You can implement a transactional outbox pattern by enabling streams on the DynamoDB table. The Lambda function for the event processing service is associated with these streams.

- When the flight table is updated, the changed data is captured by DynamoDB Streams, and the events processing service polls the stream for new records.
- When new stream records become available, the Lambda function synchronously places the message for the event in the SQS queue for further processing. You can add an attribute to the DynamoDB item to capture timestamp and sequence number as needed to improve the robustness of the implementation.



Sample code

Using an outbox table

The sample code in this section shows how you can implement the transactional outbox pattern by using an outbox table. To view the complete code, see the GitHub repository for this example.

The following code snippet saves the `Flight` entity and the `Flight` event in the database in their respective tables within a single transaction.

```
@PostMapping("/flights")
    @Transactional
    public Flight createFlight(@Valid @RequestBody Flight flight) {
        Flight savedFlight = flightRepository.save(flight);
        JsonNode flightPayload = objectMapper.convertValue(flight, JsonNode.class);
        FlightOutbox outboxEvent = new FlightOutbox(flight.getId().toString(),
 FlightOutbox.EventType.FLIGHT_BOOKED,
                flightPayload);
        outboxRepository.save(outboxEvent);
        return savedFlight;
    }
```

A separate service is in charge of regularly scanning the outbox table for new events, sending them to Amazon SQS, and deleting them from the table if Amazon SQS responds successfully. The polling rate is configurable in the `application.properties` file.

```
@Scheduled(fixedDelayString = "${sqs.polling_ms}")
    public void forwardEventsToSQS() {
        List<FlightOutbox> entities =
 outboxRepository.findAllByOrderByIdAsc(Pageable.ofSize(batchSize)).toList();
        if (!entities.isEmpty()) {
            GetQueueUrlRequest getQueueRequest = GetQueueUrlRequest.builder()
                    .queueName(sqsQueueName)
                    .build();
            String queueUrl = this.sqsClient.getQueueUrl(getQueueRequest).queueUrl();
            List<SendMessageBatchRequestEntry> messageEntries = new ArrayList<>();
            entities.forEach(entity ->
 messageEntries.add(SendMessageBatchRequestEntry.builder()
                    .id(entity.getId().toString())
                    .messageGroupId(entity.getAggregateId())
                    .messageDeduplicationId(entity.getId().toString())
                    .messageBody(entity.getPayload().toString())
                    .build())
            );
            SendMessageBatchRequest sendMessageBatchRequest =
 SendMessageBatchRequest.builder()
                    .queueUrl(queueUrl)
                    .entries(messageEntries)
                    .build();
            sqsClient.sendMessageBatch(sendMessageBatchRequest);
            outboxRepository.deleteAllInBatch(entities);
        }
    }
```

Using change data capture (CDC)

The sample code in this section shows how you can implement the transactional outbox pattern by using the change data capture (CDC) capabilities of DynamoDB. To view the complete code, see the GitHub repository for this example.

The following AWS Cloud Development Kit (AWS CDK) code snippet creates a DynamoDB flight table and an Amazon Kinesis data stream (`cdcStream`), and configures the flight table to send all its updates to the stream.

```
Const cdcStream = new kinesis.Stream(this, 'flightsCDCStream', {
    streamName: 'flightsCDCStream'
})

const flightTable = new dynamodb.Table(this, 'flight', {
    tableName: 'flight',
    kinesisStream: cdcStream,
    partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
    }

});
```

The following code snippet and configuration define a spring cloud stream function that picks up the updates in the Kinesis stream and forwards these events to an SQS queue for further processing.

```
applications.properties
spring.cloud.stream.bindings.sendToSQS-in-0.destination=${kinesisstreamname}
spring.cloud.stream.bindings.sendToSQS-in-0.content-type=application/ddb

QueueService.java
@Bean
public Consumer<Flight> sendToSQS() {
    return this::forwardEventsToSQS;
}

public void forwardEventsToSQS(Flight flight) {
    GetQueueUrlRequest getQueueRequest = GetQueueUrlRequest.builder()
            .queueName(sqsQueueName)
            .build();
    String queueUrl = this.sqsClient.getQueueUrl(getQueueRequest).queueUrl();
    try {
        SendMessageRequest send_msg_request = SendMessageRequest.builder()
                .queueUrl(queueUrl)
                .messageBody(objectMapper.writeValueAsString(flight))
                .messageGroupId("1")
                .messageDeduplicationId(flight.getId().toString())
                .build();
        sqsClient.sendMessage(send_msg_request);
    } catch (IOException | AmazonServiceException e) {
```

```
        logger.error("Error sending message to SQS", e);
    }
}
```

GitHub repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/transactional-outbox-pattern.

Resources

References

- AWS Architecture Center
- AWS Developer Center
- The Amazon Builders Library

Tools

- AWS Well-Architected Tool
- AWS App2Container
- AWS Microservice Extractor for .NET

Methodologies

- The Twelve-Factor App (ePub by Adam Wiggins)
- Nygard, Michael T. Release It!: Design and Deploy Production-Ready Software. 2nd ed. Raleigh, NC: Pragmatic Bookshelf, 2018.
- Polyglot Persistence (blog post by Martin Fowler)
- StranglerFigApplication (blog post by Martin Fowler)

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
New patterns	Added two new patterns: hexagonal architecture and scatter-gather.	May 7, 2024
New code examples	Added sample code for the change data capture (CDC) use case to the transactional outbox pattern pattern.	February 23, 2024
New code examples	• Updated the transactional outbox pattern with sample code. • Removed the section on orchestration and choreography patterns, which were superseded by saga choreography and saga orchestration.	November 16, 2023
New patterns	Added three new patterns: saga choreography, publish-subscribe, and event sourcing.	November 14, 2023
Update	Updated the strangler fig pattern implementation section.	October 2, 2023
Initial publication	This first release includes eight design patterns: anti-	July 28, 2023

	corruption layer (ACL), API routing, circuit breaker, orchestration and choreography, retry with backoff, saga orchestration, strangler fig, and transactional outbox.

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Numbers

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. You migrate servers from an on-premises platform to a cloud service for the same platform. Example: Migrate a Microsoft Hyper-V application to AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

ABAC

See attribute-based access control.

abstracted services

See managed services.

ACID

See atomicity, consistency, isolation, durability.

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than active-passive migration.

active-passive migration

A database migration method in which in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include `SUM` and `MAX`.

AI

See artificial intelligence.

AIOps

See artificial intelligence operations.

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see ABAC for AWS in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

bad bot

A bot that is intended to disrupt or cause harm to individuals or organizations.

BCP

See business continuity planning.

behavior graph

A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

A system that stores the most significant byte first. See also endianness.

binary classification

A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

blue/green deployment

A deployment strategy where you create two separate but identical environments. You run the current application version in one environment (blue) and the new application version in the other environment (green). This strategy helps you quickly roll back with minimal impact.

bot

A software application that runs automated tasks over the internet and simulates human activity or interaction. Some bots are useful or beneficial, such as web crawlers that index information on the internet. Some other bots, known as *bad bots*, are intended to disrupt or cause harm to individuals or organizations.

botnet

Networks of bots that are infected by malware and are under the control of a single party, known as a *bot herder* or *bot operator*. Botnets are the best-known mechanism to scale bots and their impact.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

See AWS Cloud Adoption Framework.

canary deployment

The slow and incremental release of a version to end users. When you are confident, you deploy the new version and replace the current version in its entirety.

CCoE

See Cloud Center of Excellence.

CDC

See change data capture.

change data capture (CDC)

The process of tracking changes to a data source, such as a database table, and recording metadata about the change. You can use CDC for various purposes, such as auditing or replicating changes in a target system to maintain synchronization.

chaos engineering

Intentionally introducing failures or disruptive events to test a system's resilience. You can use AWS Fault Injection Service (AWS FIS) to perform experiments that stress your AWS workloads and evaluate their response.

CI/CD

See continuous integration and continuous delivery.

classification

A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

client-side encryption

Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or AWS CodeCommit. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision (CV)

A field of AI that uses machine learning to analyze and extract information from visual formats such as digital images and videos. For example, AWS Panorama offers devices that add CV to on-premises camera networks, and Amazon SageMaker provides image processing algorithms for CV.

configuration drift

For a workload, a configuration change from the expected state. It might cause the workload to become noncompliant, and it's typically gradual and unintentional.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

CV

See computer vision.

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see Data classification.

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data mesh

An architectural framework that provides distributed, decentralized data ownership with centralized management and governance.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see Building a data perimeter on AWS.

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a star schema, a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a disaster. For more information, see Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML

See database manipulation language.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

DR

See disaster recovery.

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to detect drift in system resources, or you can use AWS Control Tower to detect changes in your landing zone that might affect compliance with governance requirements.

DVSM

See development value stream mapping.

E

EDA

See exploratory data analysis.

edge computing

The technology that increases the computing power for smart devices at the edges of an IoT network. When compared with cloud computing, edge computing can reduce communication latency and improve response time.

encryption

A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys can vary in length, and each key is designed to be unpredictable and unique.

endianness

The order in which bytes are stored in computer memory. Big-endian systems store the most significant byte first. Little-endian systems store the least significant byte first.

endpoint

See service endpoint.

endpoint service

A service that you can host in a virtual private cloud (VPC) to share with other users. You can create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts or to AWS Identity and Access Management (IAM) principals. These accounts or principals can connect to your endpoint service privately by creating interface VPC endpoints. For more information, see Create an endpoint service in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

enterprise resource planning (ERP)

A system that automates and manages key business processes (such as accounting, MES, and project management) for an enterprise.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see Envelope encryption in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.
- lower environments – All development environments for an application, such as those used for initial builds and tests.
- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.
- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.

ERP

See enterprise resource planning.

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a star schema. It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see AWS Fault Isolation Boundaries.

feature branch

See branch.

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with :AWS.

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the "2021-05-27 00:15:37" date into "2021", "May", "Thu", and "15", you can help the learning algorithm learn nuanced patterns associated with different data components.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

G

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries. *Detective guardrails* detect policy violations and compliance issues, and generate alerts

for remediation. They are implemented by using AWS Config, AWS Security Hub, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

H

HA

See high availability.

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends

setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

Industry 4.0

A term that was introduced by Klaus Schwab in 2016 to refer to the modernization of manufacturing processes through advances in connectivity, real-time data, automation, analytics, and AI/ML.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see What is IoT?

interpretability

A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

IoT

See Internet of Things.

IT information library (ITIL)

A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.

ITIL

See IT information library.

ITSM

See IT service management.

L

label-based access control (LBAC)

An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

malware

Software that is designed to compromise computer security or privacy. Malware might disrupt computer systems, leak sensitive information, or gain unauthorized access. Examples of malware include viruses, worms, ransomware, Trojan horses, spyware, and keyloggers.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

manufacturing execution system (MES)

A software system for tracking, monitoring, documenting, and controlling production processes that convert raw materials to finished products on the shop floor.

MAP

See Migration Acceleration Program.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

MES

See manufacturing execution system.

Message Queuing Telemetry Transport (MQTT)

A lightweight, machine-to-machine (M2M) communication protocol, based on the publish/subscribe pattern, for resource-constrained IoT devices.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include

microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

Migration Acceleration Program (MAP)

An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

migration factory

Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners, migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and

milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

MPA

See Migration Portfolio Assessment.

MQTT

See Message Queuing Telemetry Transport.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

See origin access control.

OAI

See origin access identity.

OCM

See organizational change management.

offline migration

A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

See operations integration.

OLA

See operational-level agreement.

online migration

A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

OPC-UA

See Open Process Communications - Unified Architecture.

Open Process Communications - Unified Architecture (OPC-UA)

A machine-to-machine (M2M) communication protocol for industrial automation. OPC-UA provides an interoperability standard with data encryption, authentication, and authorization schemes.

operational-level agreement (OLA)

An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operational technology (OT)

Hardware and software systems that work with the physical environment to control industrial operations, equipment, and infrastructure. In manufacturing, the integration of OT and information technology (IT) systems is a key focus for Industry 4.0 transformations.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic `PUT` and `DELETE` requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See operational readiness review.

OT

See operational technology.

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The AWS Security Reference Architecture recommends

setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see Permissions boundaries in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See personally identifiable information.

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

PLC

See programmable logic controller.

PLM

See product lifecycle management.

policy

An object that can define permissions (see identity-based policy), specify access conditions (see resource-based policy), or define the maximum permissions for all accounts in an organization in AWS Organizations (see service control policy).

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see Enabling data persistence in microservices.

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see Preventative controls in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in Roles terms and concepts in the IAM documentation.

Privacy by Design

An approach in system engineering that takes privacy into account throughout the whole engineering process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see Working with private hosted zones in the Route 53 documentation.

proactive control

A security control designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the Controls reference guide in the AWS Control Tower documentation and see Proactive controls in *Implementing security controls on AWS*.

product lifecycle management (PLM)

The management of data and processes for a product throughout its entire lifecycle, from design, development, and launch, through growth and maturity, to decline and removal.

production environment

See environment.

programmable logic controller (PLC)

In manufacturing, a highly reliable, adaptable computer that monitors machines and automates manufacturing processes.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

publish/subscribe (pub/sub)

A pattern that enables asynchronous communications among microservices to improve scalability and responsiveness. For example, in a microservices-based MES, a microservice can publish event messages to a channel that other microservices can subscribe to. The system can add new microservices without changing the publishing service.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Specify which AWS Regions your account can use.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resiliency

An application's ability to resist or recover from disruptions. High availability and disaster recovery are common considerations when planning for resiliency in the AWS Cloud. For more information, see AWS Cloud Resilience.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the

matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

rotation

The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

See recovery point objective.

RTO

See recovery time objective.

runbook

A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API

operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCADA

See supervisory control and data acquisition.

SCP

See service control policy.

secret

In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata. The secret value can be binary, a single string, or multiple strings. For more information, see What's in a Secrets Manager secret? in the Secrets Manager documentation.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: preventative, detective, responsive, and proactive.

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as detective or responsive security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see Service control policies in the AWS Organizations documentation.

service endpoint

The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see AWS service endpoints in *AWS General Reference*.

service-level agreement (SLA)

An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

A target metric that represents the health of a service, as measured by a service-level indicator.

shared responsibility model

A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see Shared responsibility model.

SIEM

See security information and event management system.

single point of failure (SPOF)

A failure in a single, critical component of an application that can disrupt the system.

SLA

See service-level agreement.

SLI

See service-level indicator.

SLO

See service-level objective.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your organization's capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

supervisory control and data acquisition (SCADA)

In manufacturing, a system that uses hardware and software to monitor physical assets and production operations.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

T

tags

Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

See environment.

training

To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

transit gateway

A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see Using AWS Organizations with other AWS services in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data. For more information, see the Quantifying uncertainty in deep learning systems guide.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See environment.

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.