Developer Guide

Amazon CodeCatalyst

Amazon CodeCatalyst: Developer Guide

Table of Contents

Developing workflow actions for Amazon CodeCatalyst

Amazon CodeCatalyst provides software development teams one place to plan work, collaborate on code, and build, test, and deploy applications with continuous integration and continuous delivery (CI/CD) tools. For more information, see What is Amazon CodeCatalyst?

In CodeCatalyst, an action is the main building block of a workflow. The actions you author define a logical unit of work to perform during a workflow run. This guide provides steps on how to create custom actions that you can use in workflows and publish to the CodeCatalyst actions catalog for others to use. By creating actions and workflows, you can automate procedures that describe how to build, test, and deploy your code as part of a continuous integration and continuous delivery (CI/CD) system. For more information, see Working with actions.

Getting started with action development

With the Action Development Kit (ADK), you can develop custom actions. This ADK provides tooling and support to help you develop actions using libraries and frameworks. To learn more about ADK, see Custom actions concepts.

Testing and publishing custom actions

After creating custom actions with the ADK, you can use the CodeCatalyst console to test the custom actions before publishing the actions to the CodeCatalyst actions catalog, where other users can add them to workflows. For more information, see Set up, test, and administer custom actions in CodeCatalyst.

> **Important**
>
> Currently, only verified partners can create custom actions, test unpublished action versions in workflows, and publish actions to the CodeCatalyst actions catalog.

Custom actions concepts

Here are some concepts to know about as you work with the Action Development Kit (ADK) to develop custom actions.

ADK components

The ADK has two components:

- **ADK command line interface (CLI)** – Tool to interact with a set of commands you can use to create, validate, and test actions.
- **ADK software development kit (SDK)** – A set of library interfaces you can use to interact with action metadata and CodeCatalyst resources, including actions, workflows, secrets, logs, input variables, output variables, artifacts, and reports.

Action components

An action contains two components:

- **Action definition** – Provides the specification for integration with Amazon CodeCatalyst CI/CD workflows. It defines the basic configuration for the action such as inputs, outputs, language, permissions, and run entry point. This `action.yml` file provides necessary information to a CodeCatalyst workflow of what the action interface and the execution profile looks like.
- **Action code** – The actual source code that is run when an action starts on CodeCatalyst. For the action to succeed, the action code must conform with the runtime profile as defined in the action definition. The code then runs on the compute provided in the action definition. For example, a runtime profile can include Node.js and the Amazon Elastic Compute Cloud (Amazon EC2) compute type.

Getting started with custom actions in CodeCatalyst

CodeCatalyst Action Development Guide provide the ability to create custom actions for your projects. You can get get started by creating your action workspace, bootstrapping and developing your action, and then testing and validating the action.

Topics

Prerequisites

To create an action, you must have completed the tasks in Setting up CodeCatalyst.

> **Important**
>
> Currently, only verified partners can create custom actions, test unpublished action versions in workflows, and publish actions to the CodeCatalyst actions catalog.

Step 1: Set up your project and Dev Environment

Create a Dev Environment to work on code stored in source repositories of your CodeCatalyst project. For more information, see Dev Environments.

You can also build actions on your local machine and push the code to your CodeCatalyst remote repository. For more information, see Setting up a project on a local machine.

To set up your project

1. Create an empty project in CodeCatalyst.

   > **Note**
   >
   > Before you create a project, you must have the **Space administrator** role, and you must create or join the space where you want to create the project. For more information, see Creating a space in CodeCatalyst.

   a. Open the CodeCatalyst console at https://codecatalyst.aws/.

   b. Navigate to the space where you want to create a project.

   c. On the space dashboard, choose **Create project**.

   d. Choose **Start from scratch**.

   e. Under **Give a name to your project**, enter the name that you want to assign to your project. The name must be unique within your space.

   f. Choose **Create project**.

2. Create an empty repository in your new project.

   > **Note**
   >
   > Third-party repositories, such as GitHub repositories, aren't supported for developing, testing, and publishing actions. The actions must be developed in a CodeCatalyst repository.

   a. Navigate to your project.

   b. In the navigation pane, choose **Code**, and then choose **Source repositories**.

   c. Choose **Add repository**, and then choose **Create repository**.

   d. In **Repository name**, provide a name for the repository. Repository names must be unique within a project. For more information about the requirements for repository names, see Quotas for source repositories in CodeCatalyst.

The action name defaults to the repository name, but it can be changed in CodeCatalyst.

e. (Optional) In **Description**, add a description for the repository that will help other users in the project understand what the repository is used for.

f. (Optional) Add a `.gitignore` file for the type of code you plan to push.

g. Choose **Create**.

> **Note**
>
> CodeCatalyst adds a `README.md` file to your repository when you create it. CodeCatalyst also creates an initial commit for the repository in a default branch named **main**. You can edit or delete the `README.md` file, but you can't change or delete the default branch.

3. Create a new feature branch.

a. In the navigation pane, choose **Code**, choose **Source repositories**, and then choose the empty repository you created.

b. Choose **Actions**, and then choose **Create branch**.

c. In the **Branch name** text input field, enter a *`feature-action-name`*.

d. In the **Create branch from** dropdown menu, ensure **main**, the source branch you're creating the new branch from, is selected, and then choose **Create**.

4. Create a Dev Environment to work on code with a supported integrated development environment (IDE).

a. In the navigation pane, do one of the following:

i. Choose **Overview**, and then navigate to the **My Dev Environments** section.

ii. Choose **Code**, and then choose **Dev Environments**.

iii. Choose **Code**, choose **Source repositories** and choose the repository for which you want to create a Dev Environment.

b. Choose **Create Dev Environment**.

c. Choose a supported IDE from the drop-down menu. See Supported integrated development environments for Dev Environments for more information.

d. Choose **Work in existing branch**, and from the **Existing branch** dropdown menu, choose the feature branch you created.

e. (Optional) In the **Alias - *optional*** text input field, enter an alias to identify the Dev Environment.

f. Choose **Create**. While your Dev Environment is being created, the Dev Environment status column will display **Starting**, and the status column will display **Running** once the Dev Environment has been created.

After setting up your project, you can create secrets for your custom actions if you're working with sensitive data that will be used in your action's workflow. For more information, see Creating secrets in CodeCatalyst for sensitive data.

Step 2: Install tools and packages

The first step in authoring actions is to install the following required tools and packages. To develop actions, you will need npm and TypeScript.

> **Note**
>
> ADK supports the following versions of tools and packages:
>
> - npm – 8+ (for example, 8.15.0)
> - node – 16+ (for example, v16.17.1)
> - tsc (TypeScript) – 4+ (for example, Version 4.9.5)
> - AWS CLI – aws-cli/2.7.27 Python/3.9.11 Darwin/22.3.0 exe/x86_64 prompt/off (minimum)
>
> For node 17+, you may run into an error: `ERR_OSSL_EVP_UNSUPPORTED`. If so, run the following:
>
> ```
> npm audit fix --force
> ```

Open a working terminal in your Dev Environment to install the necessary tools and packages.

To navigate to your Dev Environment and open terminal

1. In the navigation pane, choose **Code**, and then choose **Dev Environments**.

2. From the **IDE** column, choose **Resume in (IDE)** for the Dev Environment.

   - For JetBrains IDEs, choose **Open Link** to confirm when prompted to **Allow this site to open the JetBrains-gateway link with JetBrains Gateway?**.
   - For the VS Code IDE, choose **Open Link** to confirm when prompted to **Allow this site to open the VS Code link with Visual Studio Code?**.

   > **Note**
   >
   > Resuming a Dev Environment may take a few minutes.

   For more information, see Resuming a Dev Environment.
3. Open a new terminal window in the Dev Environment.

To install npm

Download the latest version of npm. We recommend using a Node version manager like nvm to install Node.js and npm.

To install the AWS CLI

Follow the instructions for Installing or updating the latest version of the AWS CLI.

You'll use the TypeScript programming language along with npm to build actions. It's the only language supported by the ADK.

To install TypeScript

Download tsc through npm. You can also run the following npm command:

```
npm i typescript
```

The ADK Command Line Interface (CLI) is necessary to manage and interact with the ADK files.

To install the ADK CLI

1. Run the following npm command to install the ADK CLI package:

```
npm install -g @aws/codecatalyst-adk
```

2. Validate that the ADK is running with the following command:

```
adk help
```

Step 3: Initialize your action project

Initializing the action project provides the CodeCatalyst ADK with essential information about your action such as the development language, action name, and CodeCatalyst metadata. The initialization creates an action definition file used by CodeCatalyst workflows to integrate the action within the workflow.

To initialize your action workspace

Run the following command in the `feature-action-name` branch to create an action definition YAML file (`.codecatalyst/actions/action.yml`).

```
adk init --lang typescript --space [CODECATALYST-SPACE-NAME] --proj [CODECATALYST-
PROJECT-NAME] --repo [CODECATALYST-REPO-NAME] --action [ACTION-NAME]
```

For example:

```
adk init --lang typescript --space MySpace --proj HelloWorldProject --repo
 HelloWorldAction --action HelloWorldAction
```

Ensure your space, project, and repository names are entered correctly.

Step 4: Bootstrap the action code

After the action project is initialized, you must bootstrap the action itself. Bootstrapping provides all the language-specific tools and libraries preconfigured to build, test, and release the action project.

To create an action

1. Run the following ADK CLI command in the directory for your remote repository:

```
adk bootstrap
```

(Optional) By default, the `adk bootstrap` command searches for the action definition file in `.codecatalyst/actions/action.yml`. You can use the following argument to specify a different path to the action definition file:

```
adk bootstrap -f .codecatalyst/actions/action.yml
```

Because TypeScript is used to develop the action, the bootstrap command creates TypeScript code to set up the workspace with all the node- and npm-specific toolchains and libraries. You should see the following contents:

- `.codecatalyst/actions/action.yml` – Action definition file that contains interface and implementation metadata for the action to be ingested into CodeCatalyst. This action definition file is the interface that is used by CodeCatalyst workflows to integrate the action within the workflow itself. The file defines inputs, outputs, and resource integrations within CodeCatalyst.
- `.codecatalyst/workflows/`*`actionName`*`-CI-Validation.yml` – Workflow definition file that describes a continuous integration (CI) workflow generated by the ADK bootstrap.
- `README.md` – Readme file that contains information about what the action does and how to use the action with CodeCatalyst workflows. It is used for the action documentation.
- `package.json` – File that records metadata about your project that is necessary before publishing to npm.
- `lib/index.ts` – Main file referred to in the package.json file. It is the main entry into the action.
- `test/index.test.ts` – Test file for the index.ts file.
- `tsconfig.json` – TypeScript configuration file that provides configuration options that are passed on to the tsc command.
- `jest.config.js` – Jest configuration file that is used during test runs.
- `.prettierrc.json` – Opinionated code formatter that remove original styling and makes sure for outputted code conforms to a consistent style.
- `.gitignore` – Specifies intentionally untracked files that Git should ignore.
- `.eslintrc.js` – Configuration file for ESLINT tool used to make the code consistent and avoid bugs.

- `LICENSE` – Plain text file that supplies required license information.

After your custom action's files are generated, you can configure them to your requirements, including integrating third-party functionality. To learn about specific files you can configure, see Configuring custom actions for third-party integrations in CodeCatalyst.

The ADK bootstrapping runs a pre-validation check that verifies if any of the generated files already exist. If so, the ADK will print an error message and fail. For example:

```
% adk bootstrap
   Starting action bootstrap based on definition file .codecatalyst/actions/
action.yml
   File 'tsconfig.json' already exists
   File '.prettierrc.json' already exists
   File '.gitignore' already exists
   File '.eslintrc.js' already exists
   File 'jest.config.js' already exists
   File 'LICENSE' already exists
   File 'package.json' already exists
   File 'README.md' already exists
   File 'lib/index.ts' already exists
   File 'test/index.test.ts' already exists
    => Either bootstrap in an empty directory or use 'adk bootstrap -o' to override
 existing files
   Bootstrap pre-validation failed
   Command exit code 1
```

(Optional) You can give the ADK permission to override existing files by running the following command:

```
adk bootstrap -o
```

> **Important**
>
> Running the `adk bootstrap -o` command will overwrite any code changes you make and regenerates the initial code. Any changes that aren't committed will be overwritten if the command is run.

The action definition generated by the ADK should look something like the following:

```
SchemaVersion: '1.0'
 Name: 'MyAction'
 Version: '0.0.0'
 Description: 'This Action greets someone and records the time'
 Configuration:
   WhoToGreet:
     Description: 'Who are we greeting here'
     Required: true
     DisplayName: 'Who to greet'
     Type: string
   HowToGreet:
     Description: 'How to greet the person'
     Required: false
     DisplayName: 'How to greet'
     Type: string
     Default: 'Hello there,'
 Inputs:
   Sources:
     Required: true
 Environment:
   Required: false
 Runs:
   Using: 'node16'
   Main: 'dist/index.js'
```

The CI workflow generated by the ADK should look something like this:

```
Name: MyAction-CI-Validation
 SchemaVersion: "1.0"
 Triggers:
   - Type: PullRequest
     Events: [ open, revision ]
     Branches:
       - feature-.*
 Actions:
   ValidateMyAction:
     Identifier: .
     Inputs:
       Sources:
```

```
        - WorkflowSource
    Configuration:
      WhoToGreet : 'TEST'
      HowToGreet : 'TEST'
```

2. After the bootstrapping is complete, run the following commands to commit the changes to your *feature-action-name* branch:

```
git add .
```

```
git commit -m "commit message"
```

You can also use the source control options available in the IDE you're using for your Dev Environment.

Step 5: Build the package locally

As an action author, you must build and package the action using `npm` commands. The ADK only supports actions implemented in JavaScript (js) and TypeScript (tsc). Building an action will produce `.js` files, including source code bundled with dependencies under the `dist/` folder. The bundle must be updated and pushed to the action's repository when changes are made to the source or dependencies.

To build your package locally

1. Run the following npm command to install all the dependencies. These are the necessary packages your project depends on to run:

```
npm install
```

After running the npm command, you should see the total number of added packages.

2. Run the following command to catch action errors in your action definition YAML file:

```
adk validate
```

(Optional) By default, the `adk validate` command searches for the action definition file in `.codecatalyst/actions/action.yml`. You can use the following argument to specify a different path to the action definition file:

```
adk validate -f .codecatalyst/actions/action.yml
```

3. Run the following npm command to run npm scripts:

```
npm run all
```

A successful build generates an `index.js` that contains the action's source code bundled with dependencies under the `dist/` folder. This file is ready to be run by the action runner without any other dependencies needed. To rebuild the action after making changes to the source code, run `npm run all` and commit the updated content of the `dist/` folder.

> **Important**
>
> If the size of the bundle (`dist/index.js`) is more than 10 MB, you will not be able to publish the action to the CodeCatalyst actions catalog. The bundle grows to 10 MB or more when an action has many large dependencies . For more information Quotas for source repositories in CodeCatalyst.

4. After the action is built, run the following commands to commit the changes to your remote repository:

> **Important**
>
> Make sure the code you're pushing doesn't contain any sensitive information that you don't want to be shared publicly.

```
git add .
```

```
git commit -m "commit message"
```

```
git push
```

You can also use the source control options available in the IDE you're using for your Dev Environment.

Step 5: Set action results

If you don't set status feedback for your action, the action will succeed by default. You can set an action failure status and return an error message to troubleshoot the error. Run the workflow to test your action and view the results, including results, in CodeCatalyst. For more information, Testing actions in workflows.

We recommend running business logic in a try-catch block to set errors or action feedback. The ADK provides two APIs to configure error messages and surface them:

- `core.setFailed('Action Failed, reason: ${error});` – Logs the error message. The workflow stops running and any remaining steps are skipped.
- `RunSummaries.addRunSummary(Action Failed, reason: ${error}, codecatalystRunSummaries.RunSummaryLevel.ERROR);` – Sets the workflow summary run message. This provides context about the run such as the number of tests that passed or failed, time to complete, and other relevant information added to the output variable.

  The following example shows how you can use a try-catch block for error handling:

```
export function main(): void {
        try {
            // action business logic
        } catch (error) {
            // the recommended error handling approach
            console.log(`Action Failed, reason: ${error}`);
            RunSummaries.addRunSummary(`${error}`, RunSummaryLevel.ERROR);
            core.setFailed(`Action Failed, reason: ${error}`);
        }
    }
```

Use `setFailed` to indicate that a step has failed, and use `RunSummaries` to provide additional context when the action fails in the workflow. For more information, see ADK Core's setFailed details and RunSummaries construct details.

Step 6: Test the action

Adding unit tests

The ADK CLI bootstraps actions with an empty unit test that you can use as a starting point to write sophisticated unit tests. For more information, see Adding unit tests.

Testing actions in workflows

Test your custom action before publishing to the CodeCatalyst actions catalog. To make sure your action works as expected, you can run it within the workflow and view the run's details. For more information, see Testing actions in workflows.

> **Important**
>
> Currently, only verified partners can test unpublished action versions in workflows.

The ADK generates a continuous integration (CI) workflow that is ready to be used in CodeCatalyst. By default, a bootstrapped action produces a `dist/` folder with an artifact that contains the dependencies the workflow requires to run successfully in CodeCatalyst. You must build the actions locally and push the content of the `dist/` folder to the action's source repository before testing the actions in a workflow.

After making changes to your source code following Step 4: Bootstrap the action code, build your action locally and push your code again to your CodeCatalyst repository before testing the action in a workflow.

To build and push action source code and the bundle

1. Run the following npm commands to build your action:

```
npm install
```

```
npm run all
```

2. Run the following commands to commit the changes to your remote repository:

> **Important**
>
> Make sure the code you're pushing doesn't contain any sensitive information that you don't want to be shared publicly.

```
git add .
```

```
git commit -m "commit message"
```

```
git push
```

You can also use the source control options available in the IDE you're using for your Dev Environment.

The action can now be tested with the ADK-generated workflow. By default, the workflow's name is *ActionName*-CI-Validation.

To test an action within a ADK-generated CI workflow

1. Navigate to the CodeCatalyst project page.
2. Choose the **CI/CD** dropdown menu, and then choose **Workflows**.
3. From the repository and branch dropdown menus, select the repository and feature branch in which you created the action and its workflow.
4. Choose the workflow you want to test.
5. Choose **Run** to perform the actions defined in the workflow configuration file and get the associated logs, artifacts, and variables.
6. View the workflow run status and details. For more information, see Viewing workflow run status and details.

Step 7: Publish the action

You can publish the actions in your local catalog to the CodeCatalyst actions catalog so that other CodeCatalyst users can use them.

> **Important**
>
> Currently, only verified partners can publish actions to the CodeCatalyst actions catalog.

> **Important**
>
> When your action is published to the CodeCatalyst actions catalog, it is available to all CodeCatalyst users, so make sure that you want the action to be publicly available. Other users don't have to be in your space or project to view your published action.

> **Important**
>
> If the size of the bundle (`dist/index.js`) is more than 10 MB, you will not be able to publish the action to the CodeCatalyst actions catalog. The bundle grows to 10 MB or more when an action has many large dependencies. For more information Quotas for source repositories in CodeCatalyst.

The action can only published from the default branch of the source repository. If you developed the action on a feature branch, merge your feature branch with the action to the default branch.

To merge your feature branch to the default branch

Create a pull request for other members to review and merge the changes from the feature branch to the default branch. For more information, see Working with pull requests in Amazon CodeCatalyst.

After the action information is merged to the default branch, you can publish the action to the CodeCatalyst actions catalog. Before publishing, you can also edit the metadata details of the action version.

To edit details and publish the action

1. Navigate to the CodeCatalyst project page
2. In the navigation pane, choose **CI/CD**, choose **Actions**, and then choose the action you want to publish.
3. Choose **Edit details** to edit the details for your action:

a. (Optional) In the **Action display name** field, change the action display name. This is the name that appears in the **Actions** list before the action is published, as well as in the CodeCatalyst actions catalog after the action is published.

b. (Optional) In the **Action name** field, change the action name. This name is combined with the space name and action version to form the action identifier (for example, test-space/test-45tzuy@v1.0.0). In your workflow, the action identifier is used to specify the action.

> **Note**
>
> The **Action name** can't be changed after the action is published.

c. (Optional) In the **Description** field, change the description. This description appears for the action in the **Actions** list and the Amazon CodeCatalyst catalog (after the action is published).

d. From the **Categories** dropdown list, choose the type of actions that are part of your workflow. These categories appear when you or other CodeCatalyst users choose the action's name from CodeCatalyst catalog while working with workflows.

e. (Optional) In the **Support contact** field, enter an email other CodeCatalyst users can reach out to regarding the action you published.

f. Choose **Save**

4. (Optional) Edit the license file. This file is created when the action is bootstrapped and is stored at the root of action's source repository.

   a. Choose **View license file** to open the file.

   b. Choose **Edit** and make your changes.

   c. Choose **Commit**, add a message in the **Commit message** field, and then choose **Commit**.

5. Choose **Publish version** to view the publish version details.

6. Choose the **Commit** dropdown list, and then choose the commit from the default branch you want to publish.

> **Note**
>
> The commit must meet publishing requirements, including a valid action definition, a readme file, a license file, and entry files.

The **Code quality** section displays the code quality of your results. Not meeting the quality results doesn't block you from publishing the action version. The **Test details** section provides testing and code coverage results. You can add and run unit tests to meet your requirements for the action. For more information, see Testing a custom action.

7. Choose **Publish** to publish the action to the CodeCatalyst actions catalog. In the **Versions** table, the status of the version displays **Published** once the action version has been successfully published.

Next steps

After developing your custom action, you can update and publish new action versions to the CodeCatalyst actions catalog:

- Publishing a new custom action version

Set up, test, and administer custom actions in CodeCatalyst

Using the CodeCatalyst console, you can view and publish custom actions in CodeCatalyst. Before managing your actions, you can test them by running workflows and viewing the log details of the run. As a verified partner, you can publish to the CodeCatalyst actions catalog to make the actions publicly available, or they can remain local in your project. After publishing, you can make changes and publish the latest versions of your actions. After an action is published, any CodeCatalyst user can use the action in their workflow.

Topics

Setting up a project on a local machine

While it's recommended that you create a Dev Environment and build your action within a CodeCatalyst-supported IDE, you can also set up your project and build your action on your local machine. For more information about setting up a Dev Environment, see Step 1: Set up your project and Dev Environment.

To set up your project

1. Create an empty project in CodeCatalyst.

> **Note**
>
> Before you create a project, you must have the **Space administrator** role, and you must create or join the space where you want to create the project. For more information, see Creating a space in CodeCatalyst.

a. Open the CodeCatalyst console at https://codecatalyst.aws/.

b. Navigate to the space where you want to create a project.

c. On the space dashboard, choose **Create project**.

d. Choose **Start from scratch**.

e. Under **Give a name to your project**, enter the name that you want to assign to your project. The name must be unique within your space.

f. Choose **Create project**.

2. Create an empty repository in your new project.

> **Note**
>
> Third-party repositories, such as GitHub repositories, aren't supported for developing, testing, and publishing actions. The actions must be developed in a CodeCatalyst repository.

a. Navigate to your project.

b. In the navigation pane, choose **Code**, and then choose **Source repositories**.

c. Choose **Add repository**, and then choose **Create repository**.

d. In **Repository name**, provide a name for the repository. Repository names must be unique within a project. For more information about the requirements for repository names, see Quotas for source repositories in CodeCatalyst.

The action name defaults to the repository name, but it can be changed in CodeCatalyst.

e. (Optional) In **Description**, add a description for the repository that will help other users in the project understand what the repository is used for.

f. (Optional) Add a `.gitignore` file for the type of code you plan to push.

g. Choose **Create**.

> **Note**
>
> CodeCatalyst adds a `README.md` file to your repository when you create it. CodeCatalyst also creates an initial commit for the repository in a default branch

named **main**. You can edit or delete the `README.md` file, but you can't change or delete the default branch.

3. Create a new feature branch and clone the remote repository.

   a. In the navigation pane, choose **Code**, choose **Source repositories**, and then choose the empty repository you created.

   b. Choose **Actions**, and then choose **Create branch**.

   c. In the **Branch name** text input field, enter a *`feature-action-name`*.

   d. In the **Create branch from** dropdown menu, ensure **main**, the source branch you're creating the new branch from, is selected.

   e. Choose **Clone repository** and **Copy** the **HTTPS clone URL** for the remote repository.

   f. Choose **Create token** for a personal access token (PAT) needed to clone the repository.

   g. Choose **Copy** and save the copied PAT for a later step.

   h. From your working terminal, clone the remote repository in a local folder with the following git command:

   ```
   git clone https://[CODECATALYST-USER]@[GIT-ENDPOINT]/v1/[CODECATALYST-SPACE-NAME]/[CODECATALYST-PROJECT-NAME]/[CODECATALYST-REPO-NAME]
          # The url should be available when you visit the repository created.
   ```

   When prompted for a password, paste the copied PAT as the password and enter it in your working terminal.

   i. Change your directory to the repository you cloned:

   ```
   cd [CODECATALYST-PROJECT-NAME]
   ```

   j. Switch to the new branch:

   ```
   git checkout feature-action-name
   ```

After setting up your project, you can continue on to the next step before you can begin developing your action using the CodeCatalyst ADK. For more information, see Step 2: Install tools and packages.

Testing a custom action

Use the following instructions to add unit tests and also test your custom actions in CodeCatalyst workflows.

> **Important**
>
> Currently, only verified partners can test unpublished action versions in workflows.

Contents

Adding unit tests

The ADK CLI bootstraps actions with an empty unit test using Jest's testing framework. You can use the empty unit test as a starting point to write sophisticated unit tests. The tests are executed when an action is built, and the action build fails if the tests fail or if the test coverage doesn't meet the expected percentage. You can configure the Jest configuration file (`jest.config.js`) generated by the ADK CLI to incorporate test coverage and reporting, as well as other forms of testing.

The following JavaScript example uses the Jest testing framework to define a test for an outgoing webhook action:

```
// @ts-ignore
import * as core from '@aws/codecatalyst/adk-core';
import { expect, test, describe } from '@jest/globals';
import { getHeadersInput } from '../lib/utils/input-util';
import { WEBHOOK_HEADERS_MALFORMED_MESSAGE } from '../lib/constants';

const SAMPLE_INPUT_URL = 'https://hooks.sample.com';
const SAMPLE_INPUT_BODY = '{"Sample": "BODY"}';

describe('Outgoing Webhook Action', () => {
    test('Raises Validation error if webhook headers aren not JSON format', async () =>
 {
        core.getInput = jest.fn().mockImplementation(inputName => {
```

```
            switch (inputName) {
                case 'WebhookRequestURL': {
                    return SAMPLE_INPUT_URL;
                }
                case 'WebhookRequestHeaders': {
                    return 'invalidHeaders';
                }
                case 'WebhookRequestBody': {
                    return SAMPLE_INPUT_BODY;
                }
                default: {
                    throw new Error('Unknown input provided');
                }
            }
        });
        expect(() => {
            getHeadersInput();
        }).toThrowError(WEBHOOK_HEADERS_MALFORMED_MESSAGE);
    });
});
```

Testing actions in workflows

To test your action before publishing as a verified partner, you can run it within the workflow and view the run's details. Your workflow generally runs automatically due to a trigger that can include one or more events, such as a code push or pull request. If a trigger isn't defined in the workflow, the workflow can only be started manually. For more information, see Creating, editing, and deleting a workflow.

The ADK generates a continuous integration (CI) workflow that is ready to be used in CodeCatalyst. By default, a bootstrapped action produces a `dist/` folder with an artifact that contains the dependencies the workflow requires to run successfully in CodeCatalyst. You must build the actions locally and push the content of the `dist/` folder to the action's source repository before testing the actions in a workflow.

An action is determined by an action identifier, which consists of the action name and action version. In the workflow definition file, this information indicates which action and version to run in the workflow. When an action is not published, `.` is used as an action identifier in the CI workflow, which is generated by the ADK, while testing. This can help to reference an action that is in the same repository as the workflow file (`.codecatalyst/workflows/`*`actionName`*`-CI-Validation.yml`).

```
Name: MyAction-CI-Validation
SchemaVersion: "1.0"
Triggers:
  - Type: PullRequest
    Events: [ open, revision ]
    Branches:
      - feature-.*
Actions:
  ValidateMyAction:
    Identifier: .
    Inputs:
      Sources:
        - WorkflowSource
    Configuration:
      WhoToGreet : 'TEST'
      HowToGreet : 'TEST'
```

After making any changes to your source code, build your action locally and push your code again to your CodeCatalyst repository before testing the action in a workflow.

To build and push action source code and the bundle

1. Run the following npm commands to build your action:

```
npm install
```

```
npm run all
```

2. Run the following commands to commit the changes to your remote repository:

> **Important**
>
> Make sure the code you're pushing doesn't contain any sensitive information that you don't want to be shared publicly.

```
git add .
```

```
git commit -m "commit message"
```

```
git push
```

If you're making changes using a Dev Environment with a supported IDE, you can also use the source control options available in the IDE.

The action can now be tested with the ADK-generated workflow. By default, the workflow's name is *ActionName*-CI-Validation.

To test an action within a ADK-generated workflow

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to the CodeCatalyst project page.
3. In the navigation pane, choose **CI/CD**, and then choose **Workflows**.
4. From the repository and branch dropdown menus, select the repository and feature branch in which you created the action and its workflow.
5. Choose the workflow you want to test.
6. Choose **Run** to perform the actions defined in the workflow configuration file and get the associated logs, artifacts, and variables.
7. View the workflow run status and details. For more information, see Viewing workflow run status and details.

To test an action in a new workflow

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to the CodeCatalyst project page.
3. In the navigation pane, choose **CI/CD**, and then choose **Workflows**.
4. From the repository and branch dropdown menus, select the repository and feature branch in which you created the action.
5. Choose **Create workflow**, confirm the repository and feature branch in which you created the action, and then choose **Create**.
6. Choose **+ Actions**, choose the **Actions** dropdown menu, and then choose **Local** to view your custom action.

7. (Optional) Choose the name of the custom action to view the action's details, including the description, documentation information, YAML preview, and license file.
8. Choose **+** to add your custom action to the workflow and configure the workflow to meet your requirements using the YAML editor or the visual editor. For more information, see Build, test, and deploy with workflows in CodeCatalyst.
9. (Optional) Choose **Validate** to validate the workflow's YAML code before committing.
10. Choose **Commit**, and on the **Commit workflow** dialog box, do the following:
    a. For **Workflow file name**, leave the default name or enter your own.
    b. For **Commit message**, leave the default message or enter your own.
    c. For **Repository** and **Branch**, choose the source repository and branch for the workflow definition file. These fields should be set to the repository and branch that you specified earlier in the **Create workflow** dialog box. You can change the repository and branch now, if you'd like.

       Note

       After committing your workflow definition file, it cannot be associated with another repository or branch, so make sure to choose them carefully.

    d. Choose **Commit** to commit the workflow definition file.
11. View the workflow run status and details. For more information, see Viewing workflow run status and details.

Publishing a custom action

In CodeCatalyst, you can publish multiple versions of an action, retrieve action metadata, and manage your actions. You can publish the actions in your local catalog to the CodeCatalyst actions catalog so that other CodeCatalyst users can use them.

Important

Currently, only verified partners can publish actions to the CodeCatalyst actions catalog.

Important

When your action is published to the CodeCatalyst actions catalog, it is available to all CodeCatalyst users, so make sure that you want the action to be publicly available. Other users don't have to be in your space or project to view your published action. The action's source code, including the workflow YAML file and git history, become visible after the action is published.

Important

If the size of the bundle (`dist/index.js`) is more than 10 megabyte (MB), you will not be able to publish the action to the CodeCatalyst actions catalog. The bundle grows to 10 MB or more when an action has many large dependencies.

The action can only published from the default branch of the source repository. If you developed the action on a feature branch, merge your feature branch with the action to the default branch.

To merge your feature branch to the default branch

Create a pull request for other members to review and merge the changes from the feature branch to the default branch. For more information, see Working with pull requests in Amazon CodeCatalyst.

After merging the feature branch with the action information to the default branch, the action details can be configured before publishing the action to the CodeCatalyst actions catalog. The following details can be edited:

- **Action display name** – The name that appears in the **Actions** list and the Amazon CodeCatalyst catalog (after the action is published). This is initially set in the action definition file `action.yml`.
- **Action name** – The name is derived from the space name and action version to form the action identifier (for example, test-space/nad-test-45tzuy@v1.0.0). In your workflow, the action identifier is used to specify the action. After publishing the action, this name can't be changed.
- **Description** – The description that appears for the action in the **Actions** list and the Amazon CodeCatalyst catalog (after the action is published).

- **Categories** – The catgories that best describe the action. These categories appear when you or other CodeCatalyst users choose the action's name from CodeCatalyst catalog while working with workflows. This category is initially empty, and at least one category is required before you can publish an action.
- **Support contact** – The email other CodeCatalyst users can reach out to regarding the action you published. This detail is initially empty and not required to publish your action.
- **License** – A plain text file that supplies required license information. This file is created when the action is bootstrapped and is stored at the root of action's source repository.

To edit action details

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to the CodeCatalyst project page
3. In the navigation pane, choose **CI/CD**, choose **Actions**, and then choose the action you want to publish.
4. Choose **Edit details** to edit the details for your action:
   a. (Optional) In the **Action display name** field, change the action display name.
   b. (Optional) In the **Action name** field, change the action name.

      > **Note**
      >
      > The **Action name** can't be changed after the action is published.

   c. (Optional) In the **Description** field, change the description.
   d. From the **Categories** dropdown list, choose the type of actions that are part of your workflow. This field is initially empty and requires at least one category before you can publish the action.
   e. (Optional) In the **Support contact** field, enter an email.
   f. Choose **Save**
5. (Optional) Edit the license file.
   a. Choose **View license file** to open the file.
   b. Choose **Edit** and make your changes.
   c. Choose **Commit**, add a message in the **Commit message** field, and then choose **Commit**.

After configuring the action details to meet all the requirements to publish, you can choose the action version you want to publish to the Amazon CodeCatalyst actions catalog.

To publish your custom action

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to the CodeCatalyst project page.
3. In the navigation pane, choose **CI/CD**, choose **Actions**, and then choose the action you want to publish.
4. Choose **Publish version** to view the publish version details.
5. Choose the **Commit** dropdown list, and then choose the commit from the default branch you want to publish.

   > **Note**
   >
   > The commit must meet publishing requirements, including a valid action definition, a readme file, a license file, and entry files.

   The **Code quality** section displays the code quality of your results. Not meeting the quality results doesn't block you from publishing the action version. The **Test details** section provides testing and code coverage results. You can add and run unit tests to meet your requirements for the action. For more information, see Testing a custom action.
6. Choose **Publish** to publish the action to the Amazon CodeCatalyst actions catalog. In the **Versions** table, the status of the version displays **Published** once the action version has been successfully published.

Optionally, you can view and test the action version you published to the Amazon CodeCatalyst actions catalog to ensure it works as expected.

(Optional) To view and use your published action

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to the CodeCatalyst project page.
3. In the navigation pane, choose **CI/CD**, and then choose **Workflows**.

4. From the repository and branch dropdown menus, select the repository and feature branch in which you want to test the published action.
5. Choose **Create workflow**, confirm the repository and feature branch in which you want to test the published action, and then choose **Create**.
6. Choose **+ Actions**, and then search for your custom action that you published. You can search the name of your action by entering it in the *Search for actions* field.
7. (Optional) Choose the name of the published action to view the action's details, including the description, documentation information, YAML preview, and license file.
8. Add the action to the workflow, choose the visual editor, and then choose the action to view the **Inputs**, **Configuration**, and **Outputs** fields.

   > **Note**
   >
   > The action now contains the identifier (for example, test-space/test-45tzuy@v1.0.0), which is not available for the action when initially created before publishing.

9. Choose **Commit**, and on the **Commit workflow** dialog box, do the following:

   a. For **Workflow file name**, leave the default name or enter your own.

   b. For **Commit message**, leave the default message or enter your own.

   c. For **Repository** and **Branch**, choose the source repository and branch for the workflow definition file. These fields should be set to the repository and branch that you specified earlier in the **Create workflow** dialog box. You can change the repository and branch now, if you'd like.

      > **Note**
      >
      > After committing your workflow definition file, it cannot be associated with another repository or branch, so make sure to choose them carefully.

   d. Choose **Commit** to commit the workflow definition file.

10. View the workflow run status and details. For more information, see Viewing workflow run status and details.

Publishing a new custom action version

After publishing your initial action version, you can update your action definition and publish a new version to the Amazon CodeCatalyst actions catalog. Unlike publishing an action initially, you can't change the **Action name** when editing details for later versions.

Consider following Semantic Versioning (SemVar) standards when working with updated versions of your actions. SemVar provides a standardized way to assign and increment version numbers for software packages. When dependencies become complex as your system grows and more packages are integrated into a software, a clear and precise way to convey meaning about changes can help other CodeCatalyst users understand the intentions while you make flexible and reliable specifications. For more information, see Semantic Versioning 2.0.0.

The Conventional Commits specification provides a lightweight convention for structing commit messages, which gives you the ability create an explicit commit history and write automateed tools on top of it. This convention fits with SemVar by describing features, fixes, and beaking changes made in commit messages, including guidelines on how commit messages should be structured. For more information, see Conventional Commits.

To publish your new action version

1. Navigate to your project that was cloned in a local folder, and make your changes in your existing *`feature-action-name`* branch that was created in Step 1: Set up your project and Dev Environment. You can also create a new feature branch from your default branch and make changes in the new branch.
2. Build the package locally and push the source code and bundle:

   a. Run the following npm commands to build your action:

   ```
   npm install
   ```

   ```
   npm run all
   ```

   b. Run the following commands to commit the changes to your remote repository:

> **Important**
>
> Make sure the code you're pushing doesn't contain any sensitive information that you don't want to be shared publicly.

```
git add .
```

```
git commit -m "commit message"
```

```
git push
```

You can also use the source control options available in the IDE you're using for your Dev Environment.

3. Create a pull request and merge your feature branch to the default branch, and then publish your new action version to the CodeCatalyst actions catalog. For more information, see Publishing a custom action.

Optionally, you can view and test the action version you published to the Amazon CodeCatalyst actions catalog to ensure it works as expected.

(Optional) To view and use your published action

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to the CodeCatalyst project page.
3. In the navigation pane, choose **CI/CD**, and then choose **Workflows**.
4. From the repository and branch dropdown menus, select the repository and feature branch in which you want to test the published action.
5. Choose **Create workflow**, confirm the repository and feature branch in which you want to test the published action, and then choose **Create**.
6. Choose **+ Actions**, and then search for your custom action that you published. You can search the name of your action by entering it in the *Search for actions* field.
7. (Optional) Choose the name of the published action to view the action's details, including the description, documentation information, YAML preview, and license file.

8. Add the action to the workflow, choose the visual editor, and then choose the action to view and configure the **Inputs**, **Configuration**, and **Outputs** fields.

   > **Note**
   >
   > The published action contains the identifier (for example, test-space/test-45tzuy@v1.0.0), which is not available for the action when initially created and not published.

9. (Optional) Choose **Validate** to validate the workflow's YAML code before committing.
10. Choose **Commit**, and on the **Commit workflow** dialog box, do the following:

    a. For **Workflow file name**, leave the default name or enter your own.

    b. For **Commit message**, leave the default message or enter your own.

    c. For **Repository** and **Branch**, choose the source repository and branch for the workflow definition file. These fields should be set to the repository and branch that you specified earlier in the **Create workflow** dialog box. You can change the repository and branch now, if you'd like.

       > **Note**
       >
       > After committing your workflow definition file, it cannot be associated with another repository or branch, so make sure to choose them carefully.

    d. Choose **Commit** to commit the workflow definition file.

11. View the workflow run status and details. For more information, see Viewing workflow run status and details.

Deleting a custom action version

Use the following instructions to delete a published version of an action. Deleting a version removes it from the action catalog so that it is no longer available for use in workflows. Any workflows that currently use the deleted version will stop working.

Important

To avoid disruption to those who are currently using your action in their workflows, only delete an action version if you've reached the version limit, or if the version contains security vulnerabilities or other critical issues that are impossible to solve with a new version.

To delete an action version

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to the CodeCatalyst project page.
3. In the navigation pane, choose **CI/CD**, and then choose **Actions**.

   Your custom actions appear.
4. Choose the name of the action whose version you want to delete.
5. Choose the radio button next to the version.
6. Choose **Delete**.

Note

If there is only one version available, it cannot be deleted.

Custom action examples for CodeCatalyst

You can use the following examples to develop actions:

- Example: AWS CodeBuild action using ADK: This example action initiates a build in AWS CodeBuild, which compiles source code, run tests, and packages artifacts. The action invokes the AWS CLI, preinstalled on the CodeCatalyst runtime environment image. The CLI command output is streamed to the console.
- Example: Outgoing webhook action using ADK: This example action can initiate an outgoing webhook (OW) and make a POST request to a provided URL. With the action, you can bridge Amazon CodeCatalyst workflows with predefined web services like status reporting and sharing artifacts.

Example: AWS CodeBuild action using ADK

The following example action initiates a build in AWS CodeBuild. CodeBuild is an AWS service that compiles source code, runs tests, and packages the code into artifacts. For more information, see the AWS CodeBuild Documentation.

This action invokes the AWS Command Line Interface (AWS CLI), which is preinstalled on the action's runtime environment image within CodeCatalyst. The output of the CLI command is streamed to the console using `stdout`. For more information about what tools are installed on the runtime image, see Curated images.

Topics

Prerequisites

Complete all of the steps in Getting started with custom actions in CodeCatalyst before moving on with developing the action.

Languages and toolchains

In this example, we'll develop an action using npm and TypeScript.

Update the action definition

Update the action definition (`action.yml`) that was generated in Step 3: Initialize your action project with the following `AWSCodeBuildProject` and `AWSRegion` input parameters:

```
SchemaVersion: '1.0'
    Name: 'AWSCodeBuildAction Action'
    Version: '0.0.0'
    Description: 'This Action starts a build in CodeBuild'
    Configuration:
      AWSCodeBuildProject:
        Description: 'Project name for AWS CodeBuild project'
        Required: true
        DisplayName: 'AWSCodeBuildProject'
        Type: string
      AWSRegion:
        Description: 'AWS Region'
        Required: false
        DisplayName: 'AWSRegion'
        Type: string
    Environment:
      Required: true
    Runs:
      Using: 'node16'
      Main: 'dist/index.js'
```

Update the action code

Update the entrypoint code in the `lib/index.ts` file that was generated in Step 4: Bootstrap the action code:

```
// @ts-ignore
    import * as core from '@aws/codecatalyst-adk-core';
    // @ts-ignore
    import * as codecatalystProject from '@aws/codecatalyst-project';
    // @ts-ignore
    import * as codecatalystSpace from '@aws/codecatalyst-space';
```

```
   try {
     // Get inputs from the action
    const input_AWSCodeBuildProject = core.getInput('AWSCodeBuildProject'); // Project
name for AWS CodeBuild project
    console.log(input_AWSCodeBuildProject);
    const input_AWSRegion = core.getInput('AWSRegion'); // AWS Region
    console.log(input_AWSRegion);

     // Interact with codecatalyst entities
     console.log(`Current CodeCatalyst space ${codecatalystSpace.getSpace().name}`);
     console.log(`Current codecatalyst project
${codecatalystProject.getProject().name}`);
     console.log(`AWS Region ${input_AWSRegion}`);

     // Action Code start

     console.log(core.command(`aws codebuild start-build --project-name
${input_AWSCodeBuildProject}`));
     // Set outputs of the action

   } catch(error) {
     core.setFailed(`Action Failed, reason: ${error}`);
   }
```

After bootstrapping and updating the action code, continue with Step 4: Bootstrap the action code to complete the local build.

Validate the action within the CodeCatalyst workflow

After Testing a custom action, validate the action.

To validate the action

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to your project.
3. In the navigation pane, choose **CI/CD**, and then choose **Workflows**.
4. Choose the workflow with the action that you want to validate, then view **Logs** to confirm a successful run.

Example: Outgoing webhook action using ADK

The outgoing webhook action can initiate an outgoing webhook (OW) and make a POST request to a provided URL. With the action, you can bridge Amazon CodeCatalyst workflows with predefined web services like status reporting and sharing artifacts.

Topics

Prerequisites

Complete all of the steps in Getting started with custom actions in CodeCatalyst before moving on with developing the action.

Languages and toolchains

In this example, we'll develop an action using npm and TypeScript.

Update the action definition

Update the action definition (`action.yml`) that was generated in Step 3: Initialize your action project with the following `WebhookRequestURL` and `WebhookRequestHeaders` input parameters, in addition to `WebhookRequestBody` (optional):

```
SchemaVersion: '1.0'
    Name: 'OutgoingWebhookAction'
    Version: '0.1.0'
    Description: 'Outgoing Webhook Action allows user to send messages within workflow
 to an arbitrary web server using HTTP request'
    Configuration:
      WebhookRequestURL:
        Description: 'Outgoing webhook URL from an arbitrary web server'
        Required: true
        DisplayName: 'Request URL'
        Type: string
```

```
      WebhookRequestHeaders:
        Description: 'The JSON that you want to provide to add HTTP request headers. '
        Required: false
        DisplayName: 'Request Headers'
        Type: string
        Default: false
      WebhookRequestBody:
        Description: 'The JSON that you want to provide to add HTTP request body. '
        Required: false
        DisplayName: 'Request Body'
        Type: string
        Default: false
    Environment:
      Required: false
    Runs:
      Using: 'node16'
      Main: 'dist/index.js'
```

This action invokes the AWS Command Line Interface (AWS CLI), which is preinstalled on the action's runtime environment image within CodeCatalyst. The output of the CLI command is streamed to the console using stdout.

Update the action code

The outgoing webhook action contains several source files under the `lib/` folder. This example code provides configuration of the entry point and the action itself. Update the entry point code in the `lib/index.ts` file that was generated in Step 4: Bootstrap the action code.

While building your action, you can also catch errors by setting summary run messages. For more information, see Handling errors when running workflows.

```
// @ts-ignore
    import * as core from '@aws/codecatalyst-adk-core';
    // @ts-ignore
    import { RunSummaryLevel, RunSummaries } from '@aws/codecatalyst-run-summaries';
    import { runOutgoingWebhookAction } from './action';
    import { OutgoingWebhookInput } from './constants/types';
    import { getBodyInput, getHeadersInput } from './utils/input-util';

    export function main(): void {
        try {
            // Get inputs from the action
```

```
            const webhookUrl: string = core.getInput('WebhookRequestURL'); // Outgoing
 webhook URL from an arbitrary web server
            const headers: Map<string, string> | undefined = getHeadersInput(); // The
 JSON that you want to provide to add HTTP request headers.
            const body: string | undefined = getBodyInput(); // The JSON that you want
 to provide to add HTTP request body.

            const actionInput: OutgoingWebhookInput = {
                webhookUrl,
                headers,
                body
            };

            // Run the webhook action
            runOutgoingWebhookAction(actionInput);
        } catch (error) {
            console.log(`Action Failed, reason: ${error}`);
            RunSummaries.addRunSummary(`${error}`, RunSummaryLevel.ERROR);
            core.setFailed(`Action Failed, reason: ${error}`);
        }
    }

    if (require.main === module) {
        main();
    }
```

The action first gets the inputs using the `core.getInput()` ADK API to initialize required and optional variables. The action then calls the `runOutgoingWebhookAction()` function to send the HTTP POST request with the earlier provided input. The source code of the `runOutgoingWebhookAction()` function is implemented in the `action.ts` source file. The following code example validates user input, constructs an executable shell command using `code.command()`, and logs the result:

```
// @ts-ignore
    import * as core from '@aws/codecatalyst-adk-core';
    import { OUTGOING_WEBHOOK_ERROR } from './constants';
    import { OutgoingWebhookInput } from './constants/types';
    import { validateActionInputs } from './validation/validation';

    export function runOutgoingWebhookAction(input: OutgoingWebhookInput): void {
        validateActionInputs(input);
        const shell_command = webhookRequestCommand(input);
```

```
        const { code, stderr } = core.command(shell_command);
        console.log(`shell command: ${shell_command}`);
        if (code !== 0) {
            console.log(stderr);
            throw new Error(OUTGOING_WEBHOOK_ERROR);
        }

        console.log('Outgoing Webhook command was successful');
    }

    export function webhookRequestCommand(input: OutgoingWebhookInput): string {
        const headersCommand = constructHeadersCommand(input.headers);
        const bodyCommand = input.body === undefined ? undefined : `-d '${input.body}'
`;
        return constructRequestCommand(input.webhookUrl, headersCommand, bodyCommand);
    }

    export function constructRequestCommand(url: string, headerCommand: string |
undefined, bodyCommand: string | undefined): string {
        let command = 'curl -X POST ';
        if (headerCommand) {
            command += headerCommand;
        }
        if (bodyCommand) {
            command += bodyCommand;
        }
        command += url;
        return command;
    }

    export function constructHeadersCommand(headers: Map<string, string> | undefined |
string): string | undefined {
        let headerCommand = '';
        if (headers == undefined) return undefined;
        for (const [key, value] of headers) {
            headerCommand += `-H "${key}: ${value}" `;
        }
        return headerCommand;
    }
```

This action invokes the AWS Command Line Interface (AWS CLI), which is preinstalled on the action's runtime environment image within CodeCatalyst. The output of the CLI command is streamed to the console using stdout.

After bootstrapping and updating the action code, continue with Step 4: Bootstrap the action code to complete the local build.

Validate the action within the CodeCatalyst workflow

After Testing a custom action, validate the action.

To validate the action

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to your project.
3. In the navigation pane, choose **CI/CD**, and then choose **Workflows**.
4. Choose the workflow with the action that you want to validate, then view **Logs** to confirm a successful run.

Configuring custom actions for third-party integrations in CodeCatalyst

This section details how to develop a custom action with source code that can be used by AWS partners to integrate third-party capabilities and publish to the CodeCatalyst actions catalog.

Important

Currently, only verified AWS partners with approved and allowlisted CodeCatalyst spaces can create custom actions, test unpublished action versions in workflows, and publish actions to the CodeCatalyst actions catalog. To be allowlisted, reach out to your AWS Account Manager or Partner Development Manager.

Important

As a third-party provider, you're responsible for testing the custom action, making updates, and publishing the action.

Before you can configure files for third-party capabilities, you must complete the prerequisites and the first four steps from the Getting started with custom actions in CodeCatalyst walkthrough:

- Prerequisites
- Step 1: Set up your project and Dev Environment
- Step 2: Install tools and packages
- Step 3: Initialize your action project
- Step 4: Bootstrap the action code

Topics

Step 1: Configure custom action files

You can configure the generated files to incorporate secrets, include installation, setup instructions, and descriptions. The following steps provide a walkthrough, including examples, on how you can configure your custom action's files:

To configure an action's source files

1. Configure the `.codecatalyst/actions/action.yml` file to update the `Description` of your custom action. Additionally, you can add code to accept secrets as user inputs, or use the `Default` value from the CodeCatalyst secrets to access secrets. You can also add your own secrets. For more information, Creating secrets in CodeCatalyst for sensitive data.

   Example:

```
SchemaVersion: '1.0'
Name: 'custom-action'
Version: '0.0.0'
Description: 'This Action creates custom Actions source code'
Configuration:
  AwsAccessKeyId:
    Description: 'AWS Access Key ID'
    Required: true
    DisplayName: 'AWS_ACCESS_KEY_ID'
    Type: string
    Default: ${Secrets.AWS_ACCESS_KEY_ID}
  AwsSecretAccessKey:
    Description: 'AWS Secret Access Key'
    Required: true
    DisplayName: 'AWS_SECRET_ACCESS_KEY'
    Type: string
    Default: ${Secrets.AWS_SECRET_ACCESS_KEY}
Inputs:
  Sources:
    Required: true
Environment:
  Required: false
Runs:
  Using: 'node16'
  Main: 'dist/index.js'
```

2. Configure the `.codecatalyst/workflows/custom-action-CI-Validation.yaml` file to pass data like secrets.

   Example:

```
Name: custom-action-CI-Validation
SchemaVersion: "1.0"
Triggers:
  - Type: PullRequest
    Events: [ open, revision ]
    Branches:
      - feature-.*
Actions:
  Validatecustom-action:
    Identifier: .
    Inputs:
      Sources:
        - WorkflowSource
    Configuration:
      AwsAccessKeyId : ${Secrets.AWS_ACCESS_KEY_ID}
      AwsSecretAccessKey : ${Secrets.AWS_SECRET_ACCESS_KEY}
```

3. Configure the `lib/index.ts` file to describe the CodeCatalyst action, and identify the installation and setup instructions for the solutions or offerings. For example, this tutorial installs AWS CLI, sets up the secrets created, and then installs the Amazon CloudWatch agent. Such a setup can be done by running CLI commands like `curl`, `unzip`, `sudo`, and `yum`. Since such commands can be run by CLI, the `@aws/codecatalyst-adk-core` can be used to write the commands. You can also add your own code.

   Example:

```
// @ts-ignore
import * as core from '@aws/codecatalyst-adk-core';
// @ts-ignore
import * as project from '@aws/codecatalyst-project';
// @ts-ignore
import * as runSummaries from '@aws/codecatalyst-run-summaries';
// @ts-ignore
import * as space from '@aws/codecatalyst-space';

try {
    // Get inputs from the action
```

```
    const input_AwsAccessKeyId = core.getInput('AwsAccessKeyId');
    console.log(input_AwsAccessKeyId);
    const input_AwsSecretAccessKey = core.getInput('AwsSecretAccessKey');
    console.log(input_AwsSecretAccessKey);

    // Interact with CodeCatalyst entities
    console.log(`Current CodeCatalyst space ${space.getSpace().name}`);
    console.log(`Current CodeCatalyst project ${project.getProject().name}`);

    // Action Code start

    //aws cli install
    core.command('curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -
o "awscliv2.zip"');
    core.command('unzip awscliv2.zip');
    core.command('sudo ./aws/install --update');
    core.command(`aws configure set aws_access_key_id '${input_AwsAccessKeyId}'`);
    core.command(`aws configure set aws_secret_access_key
 '${input_AwsSecretAccessKey}'`);

    //Amazon CloudWatch install
    core.command('wget https://amazoncloudwatch-agent.s3.amazonaws.com/
amazon_linux/amd64/latest/amazon-cloudwatch-agent.rpm');

    // Set outputs of the action
} catch (error) {
    core.setFailed(`Action Failed, reason: ${error}`);
}
```

4. Configure the `package.json` file to add a description for the custom action.

   Example:

   ```
   "description": "This Action creates Actions source code",
   ```

5. Configure the `README.md` file to add necessary content and procedures for the action. This documentation should commmunicate the information to successfully work with the action.
6. (Optional) Navigate to the custom action's directory, and use the print working directory (`pwd`) command to list the directory's contents.

   Example:

   ```
   [mde-user@ip-10-4-87-5 custom-action-repo]$ pwd
   ```

```
/projects/custom-action-repo[mde-user@ip-10-4-87-5 custom-action-repo]$ ls -ltr
total 456
-rw-rw-r--   1 mde-user mde-user    497 Sep 25 02:37 tsconfig.json
-rw-rw-r--   1 mde-user mde-user    808 Sep 25 02:37 jest.config.js
-rw-r--r--   1 mde-user root       5923 Sep 25 02:37 README.md
-rw-rw-r--   1 mde-user mde-user   1051 Sep 25 02:37 LICENSE
-rw-rw-r--   1 mde-user mde-user 413245 Sep 25 02:40 package-lock.json
drwxrwxr-x   2 mde-user mde-user   4096 Sep 25 02:41 lib
drwxrwxr-x   2 mde-user mde-user   4096 Sep 25 02:41 test
drwxrwxr-x 352 mde-user mde-user  12288 Sep 25 02:41 node_modules
drwxrwxr-x   3 mde-user mde-user   4096 Sep 25 02:41 coverage
drwxrwxr-x   2 mde-user mde-user   4096 Sep 25 02:41 dist
-rw-rw-r--   1 mde-user mde-user   1157 Sep 27 02:28 package.json
[mde-user@ip-10-4-87-5 custom-action-repo]$
```

7. Continue with Step 5: Build the package locally. The following steps provide examples of possible outputs.

   a. Run the following npm command to install all the dependencies. These are the necessary packages your project depends on to run:

   ```
   npm install
   ```

   After running the npm command, you should see the total number of added packages.

   Example:

   ```
   [mde-user@ip-10-4-87-5 custom-action-repo]$ npm install

   up to date, audited 492 packages in 3s

   113 packages are looking for funding
     run `npm fund` for details

   found 0 vulnerabilities
   ```

   b. Run the following command to catch action errors in your action definition YAML file:

   ```
   adk validate
   ```

   Example:

```
[mde-user@ip-10-4-87-5 custom-action-repo]$ adk validate
Starting action validation
Command exit code 0
```

c. Run the following npm command to run npm scripts:

```
npm run all
```

A successful build generates an `index.js` that contains the action's source code bundled with dependencies under the `dist/` folder. This file is ready to be run by the action runner without any other dependencies needed. To rebuild the action after making changes to the source code, run `npm run all` and commit the updated content of the `dist/` folder.

> **Important**
>
> If the size of the bundle (`dist/index.js`) is more than 10 MB, you will not be able to publish the action to the CodeCatalyst actions catalog. The bundle grows to 10 MB or more when an action has many large dependencies . For more information Quotas for source repositories in CodeCatalyst.

Example:

```
[mde-user@ip-10-4-87-5 custom-action-repo]$ npm run all

> custom-action@1.0.0 all
> npm run build && npm run format && npm run lint && npm run package


> custom-action@1.0.0 build
> tsc


> custom-action@1.0.0 format
> prettier --write '**/*.ts'

lib/index.ts 541ms
test/index.test.ts 14ms
```

```
> custom-action@1.0.0 lint
> eslint **/*.ts

=============

WARNING: You are currently running a version of TypeScript which is not
 officially supported by @typescript-eslint/typescript-estree.

You may find that it works just fine, or you may not.

SUPPORTED TYPESCRIPT VERSIONS: >=3.3.1 <5.2.0

YOUR TYPESCRIPT VERSION: 5.2.2

Please only submit bug reports when using the officially supported version.

=============

> custom-action@1.0.0 package
> tsc && jest && ncc build -o dist

 PASS  test/index.test.js
  CodeCatalyst action custom-action
    # should test the action (1 ms)


------------------------------- Coverage summary
 -------------------------------
Statements   : Unknown% ( 0/0 )
Branches     : Unknown% ( 0/0 )
Functions    : Unknown% ( 0/0 )
Lines        : Unknown% ( 0/0 )
================================================================================
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.943 s
Ran all test suites.
ncc: Version 0.36.1
ncc: Compiling file index.js into CJS
  1kB  dist/exec-child.js
211kB  dist/index.js
```

```
212kB  [2403ms] - ncc 0.36.1
```

d. After the action is built, run the following commands to commit the changes to your remote repository:

> **Important**
>
> Make sure the code you're pushing doesn't contain any sensitive information that you don't want to be shared publicly.

(Optional) You can view the changes that took place, as well as new files generated. Use the `git status` command to view the updates.

Example:

```
[mde-user@ip-10-4-87-5 custom-action-repo]$ git status
On branch feature-custom-action
Your branch is up to date with 'origin/feature-custom-action'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   .codecatalyst/actions/action.yml
        modified:   .codecatalyst/workflows/custom-action-CI-Validation.yaml
        modified:   dist/index.js
        modified:   lib/index.ts
        modified:   package.json

no changes added to commit (use "git add" and/or "git commit -a")
```

```
git add .
```

```
git commit -m "commit message"
```

Example:

```
[mde-user@ip-10-4-87-5 custom-action-repo]$ git commit -m "added changes and
 generated files"
[feature-custom-action c5adf30] added changes and generated files
```

```
 5 files changed, 40 insertions(+), 20 deletions(-)
```

```
git push
```

Example:

```
[mde-user@ip-10-4-87-5 custom-action-repo]$ git push
Enumerating objects: 23, done.
Counting objects: 100% (23/23), done.
Delta compression using up to 2 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (12/12), 2.04 KiB | 94.00 KiB/s, done.
Total 12 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Validating objects: 100%
   0564729..c5adf30  feature-custom-action -> feature-custom-action
```

You can also use the source control options available in the IDE you're using for your Dev Environment.

Step 2: Test a custom action in a workflow

The action can now be tested with the ADK-generated workflow. By default, the workflow's name is *`ActionName`*`-CI-Validation`.

To test an action within a ADK-generated workflow

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to the CodeCatalyst project page.
3. In the navigation pane, choose **CI/CD**, and then choose **Workflows**.
4. From the repository and branch dropdown menus, select the repository and feature branch in which you created the action and its workflow.
5. Choose the *`ActionName`*`-CI-Validation` workflow, and then choose **Definition** to view the YAML code for the workflow.
6. Choose **Run** to perform the actions defined in the workflow configuration file and get the associated logs, artifacts, and variables.
7. View the workflow run status and details.

To view the status and details of the run

a. Choose **View Run-*ID*** from the alert message that appears.

b. In the workflow diagram, choose the **Validateaction-vs-code** action to view the logs.

c. Choose the **Logs** tab and expand the sections to reveal the log messages, including possible errors.

Example:

```
691
You can now run: /usr/local/bin/aws --version
692--2023-09-30 20:15:56--  https://amazoncloudwatch-agent.s3.amazonaws.com/
amazon_linux/amd64/latest/amazon-cloudwatch-agent.rpm
693Resolving amazoncloudwatch-agent.s3.amazonaws.com (amazoncloudwatch-
agent.s3.amazonaws.com)... 54.231.165.137, 52.217.165.169, 3.5.25.69, ...
694Connecting to amazoncloudwatch-agent.s3.amazonaws.com (amazoncloudwatch-
agent.s3.amazonaws.com)|54.231.165.137|:443... connected.
695HTTP request sent, awaiting response... 200 OK
696Length: 66425644 (63M) [application/octet-stream]
697Saving to: 'amazon-cloudwatch-agent.rpm'

1998
2023-09-30 20:15:58 (35.8 MB/s) - 'amazon-cloudwatch-agent.rpm' saved
 [66425644/66425644]
1999
```

For more information, see Viewing workflow run status and details and Viewing the deployment logs.

To test an action in a new workflow

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. Navigate to the CodeCatalyst project page.
3. In the navigation pane, choose **CI/CD**, and then choose **Workflows**.
4. From the repository and branch dropdown menus, select the repository and feature branch in which you created the action.

5. Choose **Create workflow**, confirm the repository and feature branch in which you created the action, and then choose **Create**.
6. Choose **+ Actions**, choose the **Actions** dropdown menu, and then choose **Local** to view your custom action.
7. (Optional) Choose the name of the custom action to view the action's details, including the description, documentation information, YAML preview, and license file.
8. Choose **+** to add your custom action to the workflow and configure the workflow to meet your requirements using the YAML editor or the visual editor. For more information, see Build, test, and deploy with workflows in CodeCatalyst.
9. (Optional) Choose **Validate** to validate the workflow's YAML code before committing.
10. Choose **Commit**, and on the **Commit workflow** dialog box, do the following:

    a. For **Workflow file name**, leave the default name or enter your own.

    b. For **Commit message**, leave the default message or enter your own.

    c. For **Repository** and **Branch**, choose the source repository and branch for the workflow definition file. These fields should be set to the repository and branch that you specified earlier in the **Create workflow** dialog box. You can change the repository and branch now, if you'd like.

    > **Note**
    >
    > After committing your workflow definition file, it cannot be associated with another repository or branch, so make sure to choose them carefully.

    d. Choose **Commit** to commit the workflow definition file.
11. View the workflow run status and details. For more information, see Viewing workflow run status and details.

Step 3: Merge changes into default branches and publish actions

After a successful workflow run, merge the feature branch to the default branch in order to publish the action to the CodeCatalyst actions catalog. For more information, see Publishing a custom action.

Creating secrets in CodeCatalyst for sensitive data

As a developer, you might have API keys, secrets, or tokens. You can create secrets to use sensitive data in your workflows. The action is the main building block of a CodeCatalyst workflow and used by the workflow to integrate the action within the workflow itself. values shouldn't be used directly in any workflow definitions because they will be visible as files in your repository. With CodeCatalyst, you can protect these values by adding a secret to your project, and then referencing the secret in your workflow definition file. For more information, see Creating a secret. To learn more about workflows and actions, see Working with workflows and Working with actions.

Example: Creating AWS access key and ID

In this example, two secrets are created: AWS access key ID and an AWS secret access key that will be passed to the action.

To create secrets

1. Open the CodeCatalyst console at https://codecatalyst.aws/.
2. In the navigation pane, choose **CI/CD**, and then choose **Secrets**.
3. Choose **Create secret**.
4. Enter the following information:

   For **Name**, enter *AWS_ACCESS_KEY_ID*. This is the name for your secret.

   For **Value**, enter *AWS Access Key ID*. Enter the value for the secret. This is the sensitive information that you want to hide from view. By default, the value is not displayed. To display the value, choose **Show value**.

   (Optional) For **Description**, enter a description for your secret.
5. Choose **Create**. The secret can later be accessed using the reference ID (`$(Secrets.AWS_SECRET_ACCESS_KEY)`).
6. Choose **Create secret** to create a second secret.
7. Enter the following information:

   For **Name**, enter *AWS_ACCESS_KEY*. This is the name for your secret.

For **Value**, enter *AWS Secrets Access Key*. Enter the value for the secret. This is the sensitive information that you want to hide from view. By default, the value is not displayed. To display the value, choose **Show value**.

(Optional) For **Description**, enter a description for your secret.

8. Choose **Create**. The secret can later be accessed using the reference ID (`$(Secrets.AWS_SECRET_ACCESS_KEY)`).

Accessing data with CodeCatalyst ADK APIs

Using ADK APIs, you can access data that is set on the actions. Here is some of the data you can access.

Topics

Environment variables

CodeCatalyst sets environment variables available at the action runtime. An action can be configured with input variables and pre-defined variables that can be accessed by the action. The variables can be accessed for auditing purposes, metrics, access tokens, and other information. The following ADK API can be used to get both input variables and pre-defined variables: `code.getEnvironmentVariable('variableName');`. For more information, see ADK Core's getEnvironmentVariable details.

```
Identifier: my_org/my_action
   Configuration:
      MyEnvironment: 'MY_PROJECT'
```

```
const projectName = core.getEnvironmentVariable('MyEnvironment')
```

Action inputs

Action inputs are values passed into an action at runtime. These inputs are defined in the action definition file (`action.yml`) and can be used to specify parameters. You can access and configure the action inputs in order to customize the behavior of the action based on a specific use case. The following ADK API can be used to get the action inputs: `core.getInput(`${inputName}`)`. For more information, see ADK Core's getInput details.

```
Identifier: my_org/my_action
```

```
   Configuration:
      MyInput: 'MY_ACTION_INPUT'
```

```
const actionInput = core.getInput('MyInput')
```

Secrets

Sensitive data like authentication credentials and other values can be stored and protected in secrets with CodeCatalyst. You can then reference the secrets in your workflow definition file. For more information, see Working with secrets.

In the following workflow, the value of the `core.getInput(`${StackName}`)` secret is assigned to the `StackName` action input at runtime. For more information, see ADK Core's getInput details.

```
Actions:
  ACTIONNAME:
    Identifier: aws/cdk-deploy@v2
    Environment:
      Name: codecatalyst-cdk-deploy-environment
      Connections:
        - Name: codecatalyst-account-connection
          Role: codecatalyst-cdk-deploy-role
    Inputs:
      Sources:
        - WorkflowSource
    Configuration:
      StackName: ${Secrets.MY_SECRET_STACK_NAME}
      Region: ${Secrets.MY_REGION}
```

```
const stackName = core.getInput('StackName')
```

```
const region = core.getInput('Region')
```

Application URLs

Your workflow that deploys an application can display a URL in the workflow diagram. The clickable URL in the CodeCatalyst console can help to quickly verify your application. For more information, see Surfacing the URL of the deployed application.

You can also configure action source code with an output variable to get a URL link for your application. In the following code, the URL is first defined so the variable holds the URL you want to set as the output value. The output variable is named `AppUrl` in `code.setOutput('AppUrl, url);` with the value of the `url` variable. The output variable can then be accessed in a workflow. For more information, see Working with variables.

```
const url = "https://mycompany.myapp.com";
```

```
core.setOutput('AppUrl', url);
```

YAML - build and test custom actions

The following is the action definition YAML reference for your custom actions. You can define inputs, outputs, and resource integrations in the action definition YAML file. Once the action is defined, it can be referenced in your CI workflow file.

Choose a YAML property in the following code to see a description of it.

> **Note**
>
> Most of the YAML properties that follow have corresponding UI elements in the visual editor. To look up a UI element, use **Ctrl+F**. The element will be listed with its associated YAML property.

```
SchemaVersion: 1.0
Name: MyAction   # Name of the action - string
Id: my-action   # String
Description: This is my action.   # String
Version: 1.0.0   # SEMVER

Configuration:
    Param1:
        Description: 'First parameter'
        Required: true | false
        DisplayName: 'Param1'
        Type: number | boolean | string
    Param2:
        Description: 'Second parameter'
        Required: true | false
        Default: 'Second value'
        DisplayName: 'Param with space'
        Type: number | boolean | string

SupportedComputeType:
    - 'EC2'
    - 'LAMBDA'

# Whether the action requires an environment
# Automatically pulls in the connection/role fields
Environment:
```

```
    Required: true | false
    Connection:
        Required: true | false

# Whether the action requires any input sources/artifacts
# If required is true, then action expects at least one Inputs -> Sources
# or Inputs -> Artifacts

Inputs:
    Sources:
        Required: true | false
    Artifacts:
        Required: true | false
Outputs:   # Top 10 variables selected (if more than 10 produced)
    Variables:
        variable-name-1:
            Description: 'Output variable description.'
Runs:
    # Node
    Using: 'node16' | 'node18'
    Main: 'index.js'
    Pre: 'setup.js'
    Post: 'cleanup.js'
```

Configuration

(**Configuration**)

(Required) A section where you can define the configuration properties of the action.

Corresponding UI: **Configuration** tab

Description

(Configuration/Param/**Description**)

(Required)

Provide a description of the action.

Corresponding UI: *none*

Required

(Configuration/Param/**Required**)

(Required)

Specify whether the parameter is required. Set to `true` or `false`.

Corresponding UI: *none*

Default

(Configuration/Param/**Default**)

(Optional)

Specify the default value of the parameter.

Corresponding UI: *none*

DisplayName

(Configuration/Param/**DisplayName**)

(Optional)

Set the display name of the parameter.

Corresponding UI: *none*

Type

(Configuration/Param/**Type**)

(Optional)

The type of parameter. You can use one of the following values (default is string):

- Number (whole number)
- Boolean (TRUE or FALSE)
- String

Corresponding UI: *none*

SupportedComputeTypes

(SupportedComputeType)

(Optional)

Specify the compute types to use for the action. You can specify the following types:

- EC2
- Lambda

If you don't define **SupportedComputeTypes**, the corresponding UI will show all available options (EC2 and Lambda). Otherwise, the UI will display the specified compute types.

Corresponding UI: Configuration tab/**Compute type**

Environment

(Environment)

(Optional)

Specify the CodeCatalyst environment to use with the action.

For more information about environments, see Working with environments and Environment.

Corresponding UI: Configuration tab/**Environment**

Connection

(Connection)

(Optional)

If you define the `Connection` field in your action definition YAML file, action users will be able to leverage the default connection and role capabilities that come with an environment by specifying the environment name in the workflow YAML. If you don’t include this connection field in your

action definition YAML file, then the action users will need to specify the connection and the role they want to use with their environment.

> **Tip**
>
> By defining the `Connection` field in your action definition YAML file, action users would only need to set the `Environment` in the action, and the connection and role are automatically applied to the action if the `Environment` is configured with a default connection and role.

For more information about account connections, see Allowing access to AWS resources with connected AWS accounts.

For information about how to associate an account connection with your environment, see Creating an environment.

Corresponding UI: Configuration tab/Environment/**What's in my `my-environment`**

Inputs

(Inputs)

(Optional)

The `Inputs` section defines the data that an action needs during a workflow run.

> **Note**
>
> A maximum of four inputs (one source and three artifacts) are allowed per build action or test action. Variables don't count towards this total.

If you need to refer to files residing in different inputs (say a source and an artifact), the source input is the primary input, and the artifact is the secondary input. References to files in secondary inputs take a special prefix to distiguish them from the primary. For details, see Example: Referencing files within an artifact.

Corresponding UI: **Inputs** tab

Sources

(Inputs/**Sources**)

(Required)

Specify the labels that represent the source repositories that will be needed by the action. Currently, the only supported label is `WorkflowSource`, which represents the source repository where your workflow definition file is stored.

If you omit a source, then you must specify at least one input artifact under `action-name/Inputs/Artifacts`.

For more information, see Working with sources.

Corresponding UI: *none*

Artifacts - input

(Inputs/**Artifacts**)

(Optional)

Specify artifacts from previous actions that you want to provide as input to this action. These artifacts must already be defined as output artifacts in previous actions.

If you do not specify any input artifacts, then you must specify at least one source repository under `action-name/Inputs/Sources`.

Corresponding UI: Inputs tab/**Artifacts - optional**

Outputs

(**Outputs**)

(Optional)

Defines the data that is output by the action during a workflow run. If more than 10 output variables are produced, the top 10 variables are selected.

Corresponding UI: **Outputs** tab

Variables - output

(Outputs/**Variables**)

(Optional)

Specify the variables that you want the action to export so that they are available for use by the subsequent actions.

For more information about variables, including examples, see Working with variables.

Corresponding UI: Outputs tab/Variables/**Add variable**

variable-name-1

(Outputs/Variables/**variable-name-1**)

(Optional)

Specify the name of a variable that you want the action to export.

Corresponding UI: *none*

Description

(Outputs/Variables/Time/**Description**)

(Optional)

Provide a description of the output variable.

Corresponding UI: *none*

Runs

(**Runs**)

(Required)

Defines the runtime environment and main entry point for the action.

Corresponding UI: *none*

Using

Runs/(**Using**)

(Required)

Specify the type of runtime environment. Currently, Node 16 and Node 18 are the options.

Corresponding UI: *none*

Main

Runs/(**Main**)

(Optional)

Specify the file for the entry point of a Node.js application. This file contains your action code. Required if Node 16 or Node 18 runtime is specified for Using.

Corresponding UI: *none*

Pre

Runs/(**Pre**)

(Optional)

Allows you to run a script at the beginning of the action run. Can be defined if Node 16 or Node 18 runtime is specified for Using.

Corresponding UI: *none*

Post

Runs/(**Post**)

(Optional)

Allows your to run a script at the end of the action run. Can be defined if Node 16 or Node 18 runtime is specified for Using.

Corresponding UI: *none*

ADK API reference and CLI commands

The following information is about the ADK API reference and CLI commands to build an action.

ADK API reference

The ADK API reference provides descriptions of the available operations and data types. You can work with the ADK API by including the supported objects in your YAML file. For more information, see Build, test, and deploy with workflows in CodeCatalyst.

ADK CLI commands

The following list contains the ADK CLI commands and information about how to use each command:

- `init` – Initializes the ADK project locally and produces required configuration files with specific language support.
- `bootstrap` – Bootstraps CodeCatalyst action code by reading the action definition file. The ADK SDK is used to develop actions.
- `validate` – Validates the action definition and `README` file.
- `version` – Returns the current version of ADK.
- `help` – Shows the current set of commands.

Troubleshooting

The following information can help you troubleshoot common issues in the Amazon CodeCatalyst ADK.

Topics

Handling errors when running workflows

Problem: I see "Internal Error" when running my test workflow but I'm not sure what the issue is.

Possible fixes: Your action definition YAML file may have an error. Run the following command to catch errors in the `action.yml` file: `adk validate`.

Running action workflows for third-party repositories

Problem: My workflow run fails with a custom action's source code that is in a third-party repository (GitHub).

Possible fixes: Your custom action's source code can only be in a Amazon CodeCatalyst source repository. You can create a new repository in CodeCatalyst, move your source code to that repository, and run your workflow again with the action. For more information, see Step 1: Set up your project and Dev Environment.

Contributing to open-source library

The CodeCatalyst Action Development Guide (ADK) is an open-source library that you can contribute to. As a contributor, consider the contribution guidelines, feedback, and defects. For more information, see the ADK GitHub repository.

Developers can create a new feature branch, triggering a CI workflow to validate the changes. They can then implement the feature and create a pull request to the main branch, which initiates another CI workflow. Reviewers, including at least one from the AEF team, must approve the pull request. The developer can then merge the pull request using the squash option, simplifying any future rollbacks. Once merged, the release workflows in the ADK repository automatically triggers, updating the ADK package version.

Document history for the Amazon CodeCatalyst Action Developer Guide

The following table describes the documentation releases for the CodeCatalyst Action Development Guide Developer Guide.

Change	Description	Date
Updated content	Updated topic titles and reorganized content to improve readability and discovery. If you'd like to provide feedback on these changes, use this Provide feedback link.	August 29, 2024
Update content: Action reference `Using` field	Updated Action reference `Using` field with Node 18 option.	August 9, 2024
New content: Creating secrets	Added Creating secrets topic.	January 25, 2024
New content: Configuring custom actions for third-party integrations	Added Configuring custom actions for third-party integrations topic.	January 25, 2024
Updated content: Step 1: Set up your project and Dev Environment	Updated Getting started section to use Dev Environments to create actions instead of creating actions on local machine. Building actions locally moved to Working with custom actions section.	August 11, 2023
New content: Deleting an action version	Added a Deleting an action version topic.	June 28, 2023

New content: Action reference	Added Action reference topic.	April 1, 2023
New content	Initial publication of the Amazon CodeCatalyst Action Development Kit guide.	March 31, 2023