AWS Step Functions

Developer Guide

AWS Step Functions: Developer Guide

Table of Contents

What Is AWS Step Functions?

AWS Step Functions is a web service that enables you to coordinate the components of distributed applications and microservices using visual workflows. You build applications from individual components that each perform a discrete function, or *task*, allowing you to scale and change applications quickly.

Step Functions provides a reliable way to coordinate components and step through the functions of your application. Step Functions offers a graphical console to visualize the components of your application as a series of steps. It automatically triggers and tracks each step, and retries when there are errors, so your application executes in order and as expected, every time. Step Functions logs the state of each step, so when things do go wrong, you can diagnose and debug problems quickly.

Step Functions manages the operations and underlying infrastructure for you to ensure your application is available at any scale.

You can run your tasks in the AWS Cloud, on your servers, or on any system that has access to AWS. Access and use Step Functions by using the Step Functions console, the AWS SDKs, or an HTTP API.

This guide shows you how to develop, test, and troubleshoot your own state machine using these methods.

Overview of Step Functions

Here are some of the key features of AWS Step Functions:

- Step Functions is based on the concepts of tasks (p. 91) and state machines (p. 86).
- You define state machines using the JSON-based Amazon States Language (p. 87).
- The Step Functions console displays a graphical view of your state machine's structure, which provides you with a way to visually check your state machine's logic and monitor executions.

Service Integrations

AWS Step Functions integrates with other AWS services. You can call API actions and coordinate executions directly from the Amazon States Language. For more information, see the following topics.

- Service Integration Patterns (p. 139)
- Supported AWS Service Integrations for Step Functions (p. 146)
- Callback Pattern Example (p. 182)

Supported Regions

For a list of the regions where AWS Step Functions is available, see AWS Regions and Endpoints in the Amazon Web Services General Reference.

About Amazon Web Services

Amazon Web Services (AWS) is a collection of digital infrastructure services that developers can leverage when developing their applications. The services include computing, storage, database, and application synchronization (messaging and queuing). AWS uses a pay-as-you-go service model: you are charged only for the services that you—or your applications—use. For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see Use the AWS Free Tier. To obtain an AWS account, visit the AWS home page and choose **Create a Free Account**.

Setting Up Step Functions

Topics

Prerequisites For Setting Up Step Functions

Before you can get started using AWS Step Functions, you must create these AWS resources:

Create an AWS Account

To access any AWS service, you first need to create an AWS account, an Amazon.com account that can use AWS products. You can use your AWS account to view your activity and usage reports and to manage authentication and access.

To avoid using your AWS account root user for Step Functions actions, it is a best practice to create an IAM user for each person who needs administrative access to Step Functions.

To set up a new account

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

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

Create an IAM User

To create an administrator user for yourself and add the user to an administrators group (console)

1. Use your AWS account email address and password to sign in as the *AWS account root user* to the IAM console at https://console.aws.amazon.com/iam/.

   > **Note**
   > We strongly recommend that you adhere to the best practice of using the **`Administrator`** IAM user below and securely lock away the root user credentials. Sign in as the root user only to perform a few account and service management tasks.

2. In the navigation pane, choose **Users** and then choose **Add user**.
3. For **User name**, enter **`Administrator`**.
4. Select the check box next to **AWS Management Console access**. Then select **Custom password**, and then enter your new password in the text box.
5. (Optional) By default, AWS requires the new user to create a new password when first signing in. You can clear the check box next to **User must create a new password at next sign-in** to allow the new user to reset their password after they sign in.
6. Choose **Next: Permissions**.
7. Under **Set permissions**, choose **Add user to group**.
8. Choose **Create group**.

9. In the **Create group** dialog box, for **Group name** enter `Administrators`.
10. Choose **Filter policies**, and then select **AWS managed -job function** to filter the table contents.
11. In the policy list, select the check box for **AdministratorAccess**. Then choose **Create group**.

    Note
    You must activate IAM user and role access to Billing before you can use the `AdministratorAccess` permissions to access the AWS Billing and Cost Management console. To do this, follow the instructions in step 1 of the tutorial about delegating access to the billing console.
12. Back in the list of groups, select the check box for your new group. Choose **Refresh** if necessary to see the group in the list.
13. Choose **Next: Tags**.
14. (Optional) Add metadata to the user by attaching tags as key-value pairs. For more information about using tags in IAM, see Tagging IAM Entities in the *IAM User Guide*.
15. Choose **Next: Review** to see the list of group memberships to be added to the new user. When you are ready to proceed, choose **Create user**.

You can use this same process to create more groups and users and to give your users access to your AWS account resources. To learn about using policies that restrict user permissions to specific AWS resources, see Access Management and Example Policies.

Step 3: Get Your Access Key ID and Secret Access Key

To use Step Functions actions (for example, using Java or through the AWS Command Line Interface), you need an access key ID and a secret access key.

> **Note**
> The access key ID and secret access key are specific to AWS Identity and Access Management. Don't confuse them with credentials for other AWS services, such as Amazon EC2 key pairs.

Access keys consist of an access key ID and secret access key, which are used to sign programmatic requests that you make to AWS. If you don't have access keys, you can create them from the AWS Management Console. As a best practice, do not use the AWS account root user access keys for any task where it's not required. Instead, create a new administrator IAM user with access keys for yourself.

The only time that you can view or download the secret access key is when you create the keys. You cannot recover them later. However, you can create new access keys at any time. You must also have permissions to perform the required IAM actions. For more information, see Permissions Required to Access IAM Resources in the *IAM User Guide*.

To create access keys for an IAM user

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Users**.
3. Choose the name of the user whose access keys you want to create, and then choose the **Security credentials** tab.
4. In the **Access keys** section, choose **Create access key**.
5. To view the new access key pair, choose **Show**. You will not have access to the secret access key again after this dialog box closes. Your credentials will look something like this:

    - Access key ID: AKIAIOSFODNN7EXAMPLE
    - Secret access key: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
6. To download the key pair, choose **Download .csv file**. Store the keys in a secure location. You will not have access to the secret access key again after this dialog box closes.

Keep the keys confidential in order to protect your AWS account and never email them. Do not share them outside your organization, even if an inquiry appears to come from AWS or Amazon.com. No one who legitimately represents Amazon will ever ask you for your secret key.

7. After you download the `.csv` file, choose **Close**. When you create an access key, the key pair is active by default, and you can use the pair right away.

Related topics

- What Is IAM? in the *IAM User Guide*
- AWS Security Credentials in *AWS General Reference*

Setting Up Step Functions Local (Downloadable Version)

The downloadable version of AWS Step Functions is provided as an executable .jar file, and as a Docker image. The Java application runs on Windows, Linux, macOS X, and other platforms that support Java. In addition to Java, you need to install the AWS Command Line Interface. For information on installing and configuring the AWS CLI, see the AWS Command Line Interface User Guide.

> **Warning**
> The downloadable version of AWS Step Functions is only intended to be used for testing and shouldn't be used to process sensitive information.

Follow these steps to set up and run Step Functions on your computer:

1. Download Step Functions using the following link:

Download Links	Checksum
.tar.gz	8672b951e4dad1e354598898b56f3bf0
.zip	c3c0956a1f9b1bfcab539665602523ba

2. Extract the zip file.
3. Test the download and view version information.

```
$ java -jar StepFunctionsLocal.jar -v
Step Function Local
Version: 1.0.0
Build: 2019-01-21
```

4. (Optional) View a listing of available commands:

```
$ java -jar StepFunctionsLocal.jar -h
```

5. To start Step Functions on your computer, open a command prompt window, navigate to the directory where you extracted `StepFunctionsLocal.jar` and type the following command:

```
java -jar StepFunctionsLocal.jar
```

6. To access Step Functions running locally, use the `--endpoint-url` parameter. For example, using the AWS Command Line Interface, you would specify Step Functions commands as:

```
aws stepfunctions --endpoint-url http://localhost:8083 command
```

> **Note**
> By default Step Functions Local uses a fake account and credentials, and the region is set to US East (N. Virginia). To use Step Functions Local with AWS Lambda, or other supported services, you must configure your credentials and region.

To configure and run Step Functions Local to work with AWS Lambda, AWS SAM CLI Local, or other supported services, see the following topics.

Topics

- Step Functions (Downloadable Version) on Your Computer (p. 6)
- Step Functions (Downloadable Version) and Docker (p. 7)
- Step Functions Local Configuration Options (p. 7)
- Step Functions and AWS SAM CLI Local (p. 9)

Step Functions (Downloadable Version) on Your Computer

Run a HelloWorld State Machine Locally

Once you have run Step Functions locally with the AWS CLI, you can start a state machine execution.

1. Create a state machine from the AWS CLI by escaping the state machine definition.

```
aws stepfunctions --endpoint-url http://localhost:8083 create-state-machine --
definition "{\
  \"Comment\": \"A Hello World example of the Amazon States Language using a Pass state
\",\
  \"StartAt\": \"HelloWorld\",\
  \"States\": {\
    \"HelloWorld\": {\
      \"Type\": \"Pass\",\
      \"End\": true\
    }\
  }}" --name "HelloWorld" --role-arn "arn:aws:iam::012345678901:role/DummyRole"
```

> **Note**
> The `role-arn` is not used for Step Functions Local, but you must have it included with the proper syntax. You can use the ARN from the above example.

If you successfully create the state machine, Step Functions will respond with the creation date and the state machine ARN:

```
{
    "creationDate": 1548454198.202,
    "stateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld"
}
```

2. Start an execution using the ARN of the state machine you created.

```
aws stepfunctions --endpoint-url http://localhost:8083 start-execution --state-machine-
arn arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld
```

Step Functions Local with AWS SAM CLI Local

You can use the local version of Step Functions along with a local version of AWS Lambda. To configure this, you must install and configure the AWS Serverless Application Model (AWS SAM).

For information on configuring and running AWS SAM, see:

- Set Up AWS SAM
- Start AWS SAM CLI Local

Once Lambda is running on your local system, you can start Step Functions Local. From the directory where you extracted your Step Functions local jar files, start Step Functions Local, configuring the local Lambda endpoint:

```
java -jar StepFunctionsLocal.jar --lambda-endpoint http://127.0.0.1:3001 command
```

For more information is running Step Functions Local with AWS Lambda, see Step Functions and AWS SAM CLI Local (p. 9)

Step Functions (Downloadable Version) and Docker

The Step Functions Local Docker image enables you to get started with Step Functions Local quickly by using a docker image with all the needed dependencies. The Docker image enables you to include Step Functions local in your containerized builds, and as part of your continuous integration testing.

To get the Docker image for Step Functions Local, visit https://hub.docker.com/r/amazon/aws-stepfunctions-local, or type the Docker pull command:

```
docker pull amazon/aws-stepfunctions-local
```

To start the downloadable version of Step Functions on Docker, run:

```
docker run -p 8083:8083 amazon/aws-stepfunctions-local
```

In order to interact with AWS Lambda or other supported services you need to configure your credentials and other configuration options first. See:

- Step Functions Local Configuration Options (p. 7)
- Credentials and Configuration for Docker (p. 8)

Step Functions Local Configuration Options

To use Step Functions Local by starting the jar file, you can set configuration options by either setting them with the AWS CLI, or by including them in the system environment. For Docker, you must specify these options in a file that you reference when starting Step Functions Local.

Configuration Options

Option	Command Line	Environment
Account	-account,--aws-account	AWS_ACCOUNT_ID
Region	-region,--aws-region	AWS_DEFAULT_REGION
Wait Time Scale	-waitTimeScale,--wait-time-scale	WAIT_TIME_SCALE
Lambda Endpoint	-lambdaEndpoint,--lambda-endpoint	LAMBDA_ENDPOINT
Batch Endpoint	-batchEndpoint,--batch-endpoint	BATCH_ENDPOINT
DynamoDB Endpoint	-dynamoDBEndpoint,--dynamodb-endpoint	DYNAMODB_ENDPOINT
ECS Endpoint	-ecsEndpoint,--ecs-endpoint	ECS_ENDPOINT
Glue Endpoint	-glueEndpoint,--glue-endpoint	GLUE_ENDPOINT
SageMaker Endpoint	-sageMakerEndpoint,--sagemaker-endpoint	SAGE_MAKER_ENDPOINT
SQS Endpoint	-sqsEndpoint,--sqs-endpoint	SQS_ENDPOINT
SNS Endpoint	-snsEndpoint,--sns-endpoint	SNS_ENDPOINT

Credentials and Configuration for Docker

To configure Step Functions Local for Docker, create a file: `aws-stepfunctions-local-credentials.txt`.

This file contains your credentials and other configuration options, such as:

```
AWS_DEFAULT_REGION=AWS_REGION_OF_YOUR_AWS_RESOURCES
AWS_ACCESS_KEY_ID=YOUR_AWS_ACCESS_KEY
AWS_SECRET_ACCESS_KEY=YOUR_AWS_SECRET_KEY
WAIT_TIME_SCALE=VALUE
LAMBDA_ENDPOINT=VALUE
BATCH_ENDPOINT=VALUE
DYNAMODB_ENDPOINT=VALUE
ECS_ENDPOINT=VALUE
GLUE_ENDPOINT=VALUE
SAGE_MAKER_ENDPOINT=VALUE
SQS_ENDPOINT=VALUE
SNS_ENDPOINT=VALUE
```

Once you have configured your credentials and configuration options in `aws-stepfunctions-local-credentials.txt`, start Step Functions with the following command:

```
docker run -p 8083:8083 --env-file aws-stepfunctions-local-credentials.txt amazon/aws-
stepfunctions-local
```

Step Functions and AWS SAM CLI Local

With both Step Functions and Lambda running on your local machine, you can test your state machine and Lambda functions without deploying your code to AWS.

For more information, see:

- Setting Up Step Functions Local (Downloadable Version) (p. 5)
- Set Up AWS SAM

Topics

Step 1: Set Up The AWS Serverless Application Model

AWS SAM CLI Local requires the AWS Command Line Interface, the AWS Serverless Application Model, and Docker to be installed.

1. Install the AWS SAM CLI.

   For more information see: Installing the AWS SAM CLI.

   > **Note**
   > Before installing the AWS SAM CLI, you will need to install the AWS CLI and Docker. See the Prerequisites for installing the AWS SAM CLI.

2. Go through the AWS SAM Quick Start documentation. Be sure to follow the steps to:

   1. Initialize the Application
   2. Test the Application Locally

   This will create a `sam-app` directory, and will build an environment that includes a Python-based hello world Lambda function.

Step 2: Test AWS SAM CLI Local

Now that you have installed AWS SAM and created the Hello World Lambda function, test that it works. In the `sam-app` directory, type:

```
sam local start-api
```

This launches a local instance of your Lambda function.

```
2019-01-31 16:40:27 Found credentials in shared credentials file: ~/.aws/credentials
2019-01-31 16:40:27 Mounting HelloWorldFunction at http://127.0.0.1:3000/hello [GET]
2019-01-31 16:40:27 You can now browse to the above endpoints to invoke your functions.
 You do not need to restart/reload SAM CLI while working on your functions changes will be
 reflected instantly/automatically. You only need to restart SAM CLI if you update your AWS
 SAM template
2019-01-31 16:40:27  * Running on http://127.0.0.1:3000/ (Press CTRL+C to quit)
```

Open a browser and enter:

```
http://127.0.0.1:3000/hello
```

This will show output from your function:

```
{"message": "hello world", "location": "72.21.198.66"}
```

Enter **CTRL+C** to end the Lambda API.

Step 3: Start AWS SAM CLI Local

Now that you've tested that the function works, start AWS SAM CLI Local. In the `sam-app` directory, type:

```
sam local start-lambda
```

This will start AWS SAM CLI Local and will provide the endpoint to use:

```
2019-01-29 15:33:32 Found credentials in shared credentials file: ~/.aws/credentials
2019-01-29 15:33:32 Starting the Local Lambda Service. You can now invoke your Lambda
 Functions defined in your template through the endpoint.
2019-01-29 15:33:32  * Running on http://127.0.0.1:3001/ (Press CTRL+C to quit)
```

Step 4: Start Step Functions Local

Jar File

If using the `.jar` file version of Step Functions Local, start step functions specifying the Lambda endpoint. In the directory where you extracted the `.jar` files, type:

```
java -jar StepFunctionsLocal.jar --lambda-endpoint http://localhost:3001
```

When Step Functions Local starts, it will check the environment, and then the credentials configured in your `~/.aws/credentials` file. By default, it will start using a fake user ID, and will be listed as `region us-east-1`:

```
2019-01-29 15:38:06.324: Failed to load credentials from environment because Unable to load
 AWS credentials from environment variables (AWS_ACCESS_KEY_ID (or AWS_ACCESS_KEY) and
 AWS_SECRET_KEY (or AWS_SECRET_ACCESS_KEY))
2019-01-29 15:38:06.326: Loaded credentials from profile: default
2019-01-29 15:38:06.326: Starting server on port 8083 with account 123456789012, region us-
east-1
```

Docker

If using the Docker version of Step Functions Local, launch Step Functions with the following command.

```
docker run -p 8083:8083 amazon/aws-stepfunctions-local
```

For information on installing the Docker version of Step Functions, see Step Functions (Downloadable Version) and Docker (p. 7).

> **Note**
> You can specify the endpoint through the command line or by setting environment variables if you launch Step Functions from the `.jar` file. For the Docker version, you must specify the endpoints and credentials in a text file. See Step Functions Local Configuration Options (p. 7).

Step 5: Create a State Machine That References Your AWS SAM CLI Local Function

Once Step Functions Local is running, create a state machine that references the `HelloWorldFunction` that you initialized in Step 1: Set Up The AWS Serverless Application Model (p. 9).

```
aws stepfunctions --endpoint http://localhost:8083 create-state-machine --definition "{\
  \"Comment\": \"A Hello World example of the Amazon States Language using an AWS Lambda
 Local function\",\
  \"StartAt\": \"HelloWorld\",\
  \"States\": {\
    \"HelloWorld\": {\
      \"Type\": \"Task\",\
      \"Resource\": \"arn:aws:lambda:us-east-1:123456789012:function:HelloWorldFunction\",\
      \"End\": true\
    }\
  }\
}\
}}" --name "HelloWorld" --role-arn "arn:aws:iam::012345678901:role/DummyRole"
```

This will create a state machine and provide an ARN that you can use to start an execution:

```
{
    "creationDate": 1548805711.403,
    "stateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld"
}
```

Step 6: Start an Execution of Your Local State Machine

Once you have created a state machine, start an execution referencing the endpoint and state machine ARN.

```
aws stepfunctions --endpoint http://localhost:8083 start-execution --state-machine
 arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld --name test
```

This will start an execution of your `HelloWorld` state machine and give it the name `test`.

```
{
    "startDate": 1548810641.52,
    "executionArn": "arn:aws:states:us-east-1:123456789012:execution:HelloWorld:test"
}
```

Now that Step Functions is running locally, you can interact with it using the AWS CLI. For instance, to get information about this execution:

```
aws stepfunctions --endpoint http://localhost:8083 describe-execution --execution-arn
 arn:aws:states:us-east-1:123456789012:execution:HelloWorld:test
```

Calling `describe-execution` for an execution provides more complete details. For example:

```
{
    "status": "SUCCEEDED",
    "startDate": 1549056334.073,
    "name": "test",
    "executionArn": "arn:aws:states:us-east-1:123456789012:execution:HelloWorld:test",
    "stateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
    "stopDate": 1549056351.276,
    "output": "{\"statusCode\": 200, \"body\": \"{\\\"message\\\": \\\"hello world\\\", \\
\"location\\\": \\\"72.21.198.64\\\"}\"}",
    "input": "{}"
}
```

Getting Started

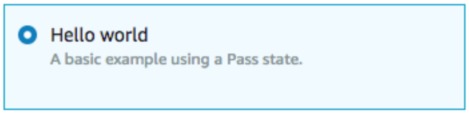
This tutorial introduces you to the basics of working with AWS Step Functions. To get started, you create a simple, independently running state machine using a `Pass` state. The `Pass` state represents a *no-op* (an instruction with no operation).

Topics

Step Functions offers various predefined state machines as *templates*. Create your first state machine using the **Hello World** template.

Create a state machine

1. Sign in to the AWS Management Console and open the Step Functions console at https://console.aws.amazon.com/states/home?#/statemachines/create.
2. On the **Define state machine** page select **Templates** and choose **Hello world**. Review the **State machine definition** and the visual workflow.

   ● Hello world
   A basic example using a Pass state.

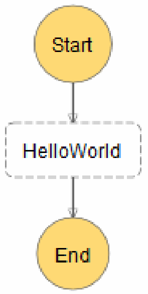
   Step Functions fills in the name of the state machine automatically. It also populates the **Code** pane with the Amazon States Language description of the state machine.

   This JSON text defines a `Pass` state named `HelloWorld`. For more information, see State Machine Structure (p. 88).

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

3. Use the graph in the **Visual Workflow** pane to check that your Amazon States Language code describes your state machine correctly.

   If you don't see the graph, choose ↻ in the **Visual Workflow** pane.



4. Choose **Next**.
5. Create or enter an IAM role:

   - To create an IAM role for Step Functions, select **Create an IAM role for me**, and enter a **Name** for your role.
   - If you have previously created an IAM role (p. 231) with the correct permissions for your state machine, select **Choose an existing IAM role**. Select a role from the list, or provide an ARN for that role.

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

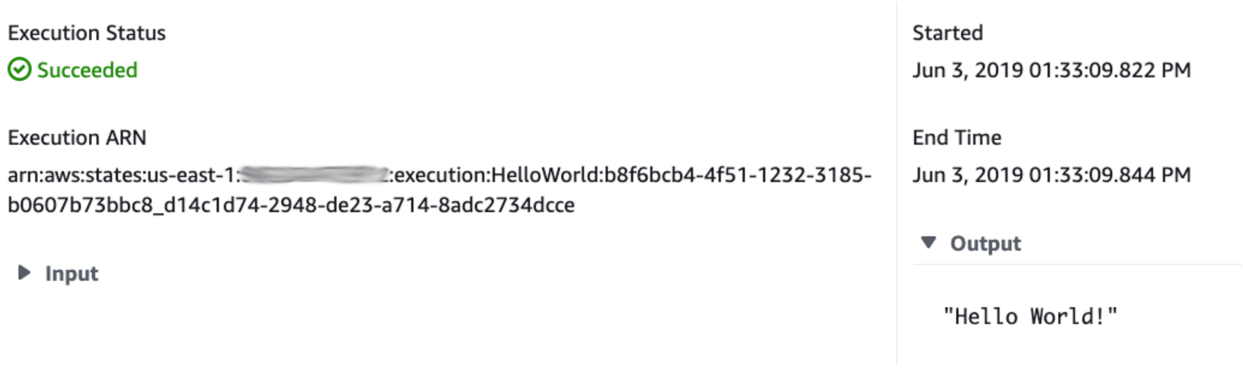
Start a new execution

After you create your state machine, you can start an execution.

1. On the ***Helloworld*** page, choose **New execution** or, if you have started an execution before, choose **Start execution**.
2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   Note
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. On the **New execution** page, choose **Start execution**.

   A new execution of your state machine starts, and a new page showing your running execution is displayed.
4. (Optional) In the **Execution Details** section, choose the **Info** tab to view the **Execution Status** and the **Started** and **Closed** timestamps.
5. To view the results of your execution, choose the **Output** tab.



Execution Status

Succeeded

Execution ARN

arn:aws:states:us-east-1:[redacted]:execution:HelloWorld:b8f6bcb4-4f51-1232-3185-b0607b73bbc8_d14c1d74-2948-de23-a714-8adc2734dcce

Input

Started

Jun 3, 2019 01:33:09.822 PM

End Time

Jun 3, 2019 01:33:09.844 PM

Output

```
"Hello World!"
```

Update a state machine

You can update your state machine for future executions.

> **Note**
> State machine updates in Step Functions are *eventually consistent*. All executions within a few seconds will use the updated definition and `roleArn`. Executions started immediately after updating a state machine can use the previous state machine definition and `roleArn`.

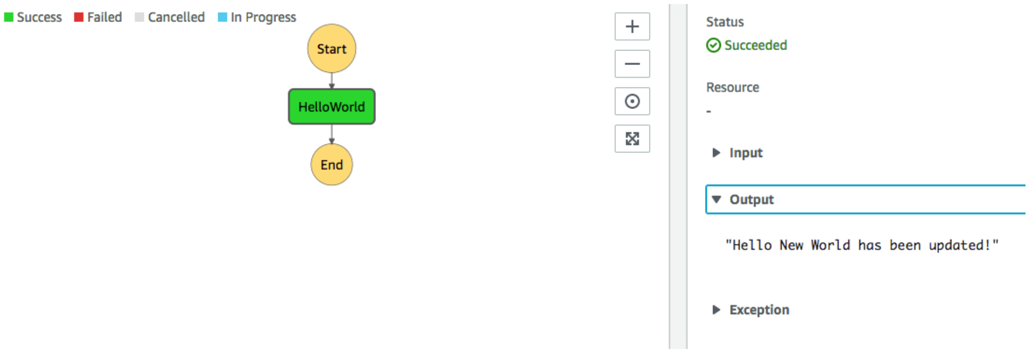
1. On the ***`Helloworld`*** page, choose **Edit**.
2. In the **Code** pane on the **Edit** page, edit the Amazon States Language description of the state machine. Update the `Result` to read `Hello World has been updated!`

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

3. (Optional) Select a new AWS Identity and Access Management (IAM) role from the **IAM role for executions** list.

   > **Note**
   > You can also choose **Create new role** to create an IAM role. For more information, see Creating IAM Roles for AWS Step Functions (p. 231).

4. Choose **Save**, and then choose **Start execution**.
5. On the **New execution** page, choose **Start Execution**.
6. To view the results of your execution, select the **HelloWorld** state in the **Visual workflow**, and expand the **Output** section under **Step details**.



```
"Hello New World has been updated!"
```

Note
The output text matches your newly updated state machine.

Next Steps

Now that you've created a simple state machine using a `Pass` state, try the following:

- Create a Lambda State Machine (p. 19)
- Create a Lambda State Machine Using AWS CloudFormation (p. 23)
- Create an Activity State Machine (p. 31)
- Handle Error Conditions Using a State Machine (p. 35)
- Start a State Machine Using Amazon CloudWatch Events (p. 40)
- Create a Step Functions API Using Amazon API Gateway (p. 48)

Tutorials

The tutorials in this section will help you understand different aspects of working with AWS Step Functions.

To complete these tutorials, you need an AWS account. If you don't have an AWS account, navigate to http://aws.amazon.com/ and choose **Sign In to the Console**.

Topics

Development Options

You can implement your AWS Step Functions state machines in several ways.

Topics

Step Functions Console

You can define a state machine using the Step Functions console. You can write complex state machines in the cloud without using a local development environment by using AWS Lambda to supply code for your tasks, and the Step Functions console to define your state machine using the Amazon States Language.

The Creating a Lambda State Machine (p. 19) tutorial uses this technique to create a simple state machine, execute it, and view its results.

AWS SDKs

Step Functions is supported by the AWS SDKs for Java, .NET, Ruby, PHP, Python (Boto 3), JavaScript, Go, and C++. These SDKs provide a convenient way to use the Step Functions HTTPS API actions in various programming languages.

You can develop state machines, activities, or state machine starters using the API actions exposed by these SDK libraries. You can also access visibility operations using these libraries to develop your own Step Functions monitoring and reporting tools.

To use Step Functions with other AWS services, see the reference documentation for the current AWS SDKs and Tools for Amazon Web Services.

> **Note**
> Step Functions supports only an HTTPS endpoint.

HTTPS Service API

Step Functions provides service operations that are accessible through HTTPS requests. You can use these operations to communicate directly with Step Functions and to develop your own libraries in any language that can communicate with Step Functions through HTTPS.

You can develop state machines, workers, or state machine starters using the service API actions. You can also access visibility operations through the API actions to develop your own monitoring and reporting tools.

For detailed information about API actions, see the AWS Step Functions API Reference.

Development Environments

You must set up a development environment that's appropriate to the programming language that you plan to use.

For example, if you intend to develop for Step Functions with Java, you should install a Java development environment (such as the AWS SDK for Java) on each of your development workstations. If you use Eclipse IDE for Java Developers, you should also install the AWS Toolkit for Eclipse. This Eclipse plugin adds features that are useful for developing on AWS.

If your programming language requires a runtime environment, you must set up the environment on each computer where these processes run.

Endpoints

To reduce latency and to store data in a location that meets your requirements, Step Functions provides endpoints in different AWS Regions.

Each endpoint in Step Functions is completely independent. A state machine or activity exists only within the Region where it was created. Any state machines and activities that you create in one Region don't share any data or attributes with those created in another Region. For example, you can register a state machine named `STATES-Flows-1` in two different Regions, but the two state machines won't share data or attributes with each other because they are completely independent from each other.

For a list of Step Functions endpoints, see Regions and Endpoints: AWS Step Functions in the *Amazon Web Services General Reference*.

AWS CLI

You can access many Step Functions features from the AWS CLI. The AWS CLI provides an alternative to using the Step Functions console or, in some cases, to programming using the AWS Step Functions API actions. For example, you can use the AWS CLI to create a state machine and then list your state machines.

The Step Functions commands in the AWS CLI enable you to start and manage executions, poll for activities, record task heartbeats, and so on. For a complete list of Step Functions commands and the descriptions of the available arguments and examples showing their use, see the *AWS CLI Command Reference*.

The AWS CLI commands follow the Amazon States Language closely, so you can use the AWS CLI to learn about the Step Functions API actions. You can also use your existing API knowledge to prototype code or perform Step Functions actions from the command line.

Step Functions Local

For testing and development purposes, you can install and run Step Functions on your local machine. With a Step Functions Local, you can start an execution on any machine.

The local version of Step Functions can invoke Lambda functions, both in AWS and running locally. You can also coordinate other supported AWS services (p. 138). For more information, see Setting Up Step Functions Local (Downloadable Version) (p. 5).

Creating a Lambda State Machine

In this tutorial, you create an AWS Step Functions state machine that uses an AWS Lambda function to implement a `Task` state. A `Task` state performs a single unit of work.

Lambda is well suited for implementing `Task` states, because Lambda functions are *stateless* (they have a predictable input-output relationship), easy to write, and don't require deploying code to a server instance. You can write code in the AWS Management Console or your favorite editor, and AWS handles the details of providing a computing environment for your function and running it.

Topics

Step 1: Create an IAM Role for Lambda

Both AWS Lambda and AWS Step Functions can execute code and access AWS resources (for example, data stored in Amazon S3 buckets). To maintain security, you must grant Lambda and Step Functions access to these resources.

Lambda requires you to assign an AWS Identity and Access Management (IAM) role when you create a Lambda function, in the same way Step Functions requires you to assign an IAM role when you create a state machine.

You use the IAM console to create a service-linked role.

To create a role (console)

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane of the IAM console, choose **Roles**. Then choose **Create role**.
3. Choose the **AWS Service** role type, and then choose **Lambda**.
4. Choose the **Lambda** use case. Use cases are defined by the service to include the trust policy required by the service. Then choose **Next: Permissions**.
5. Choose one or more permissions policies to attach to the role (for example, `AWSLambdaBasicExecutionRole`). See AWS Lambda Permissions Model.

   Select the box next to the policy that assigns the permissions that you want the role to have, and then choose **Next: Review**.
6. Enter a **Role name**.
7. (Optional) For **Role description**, edit the description for the new service-linked role.
8. Review the role, and then choose **Create role**.

Step 2: Create a Lambda Function

Your Lambda function receives input (a name) and returns a greeting that includes the input value.

> **Important**
> Ensure that your Lambda function is under the same AWS account and AWS Region as your state machine.

1. Open the Lambda console and choose **Create a function**.
2. In the **Create function** section, choose **Author from scratch**.
3. In the **Basic information** section, configure your Lambda function:

   a. For **Function name**, enter `HelloFunction`.

   b. For **Runtime**, choose **Node.js 6.10**.

   c. For **Role**, select **Choose an existing role**.

   d. For **Existing role**, select the Lambda role that you created earlier (p. 19).

   > **Note**
   > If the IAM role that you created doesn't appear in the list, the role might still need a few minutes to propagate to Lambda.

   e. Choose **Create function**.

   When your Lambda function is created, note its Amazon Resource Name (ARN) in the upper-right corner of the page. For example:

   ```
   arn:aws:lambda:us-east-1:123456789012:function:HelloFunction
   ```

4. Copy the following code for the Lambda function into the **Function code** section of the ***`HelloFunction`*** page.

   ```
   exports.handler = (event, context, callback) => {
       callback(null, "Hello, " + event.who + "!");
   };
   ```

   This code assembles a greeting using the `who` field of the input data, which is provided by the `event` object passed into your function. You add input data for this function later, when you start a new execution (p. 22). The `callback` method returns the assembled greeting from your function.

5. Choose **Save**.

Step 3: Test the Lambda Function

Test your Lambda function to see it in operation.

1. For **Select a test event**, choose **Configure test event**. For **Event name**, enter `HelloFunction`.
2. Replace the example data with the following.

```
{
    "who": "AWS Step Functions"
}
```

   The `"who"` entry corresponds to the `event.who` field in your Lambda function, completing the greeting. You will use the same input data when running the function as a Step Functions task.
3. Choose **Create**.
4. On the ***HelloFunction*** page, **Test** your Lambda function using the new data.

   The results of the test are displayed at the top of the page. Expand **Details** to see the output.

Step 4: Create a State Machine

Use the Step Functions console to create a state machine with a `Task` state. Add a reference to your Lambda function in the `Task` state. The Lambda function is invoked when an execution of the state machine reaches the `Task` state.

1. Open the Step Functions console and choose **Create a state machine**.
2. On the **Define state machine** page, choose **Author with code snippets**. Enter a **Name for your state machine**, for example, *LambdaStateMachine*.

   Note
   State machine, execution, and activity names must be 1–80 characters in length, must be unique for your account and AWS Region, and must not contain any of the following:

   - Whitespace
   - Wildcard characters (? *)
   - Bracket characters (< > { } [ ])
   - Special characters (: ; , \ | ^ ~ $ # % & ` ")
   - Control characters (\\u0000 - \\u001f or \\u007f - \\u009f).

   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. In the **State machine definition** pane, add the following state machine definition using the ARN of the Lambda function that you created earlier (p. 20), for example:

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloFunction",
      "End": true
    }
  }
}
```

This is a description of your state machine using the Amazon States Language. It defines a single `Task` state named `HelloWorld`. For more information, see State Machine Structure (p. 88).

> **Note**
> You can also set up a `Retry` for `Task` states. As a best practice, ensure production code can handle Lambda service exceptions (`Lambda.ServiceException` and `Lambda.SdkclientException`). For more information, see:
>
> - Handle Lambda Service Exceptions (p. 204).
> - Retrying after an Error (p. 133).

Choose **Next**.

4. Create or enter an IAM role:

   - To create an IAM role for Step Functions, select **Create an IAM role for me**, and enter a **Name** for your role.
   - If you have previously created an IAM role (p. 231) with the correct permissions for your state machine, select **Choose an existing IAM role**. Select a role from the list, or provide an ARN for that role.

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

5. Select **Next**.

Step 5: Start a New Execution

After you create your state machine, you start an execution.

1. On the ***`LambdaStateMachine`*** page, choose **Start execution**.

   The **New execution** page is displayed.
2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   > **Note**
   > Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. In the execution input area, replace the example data with the following.

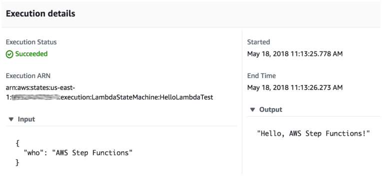
```
{
    "who" : "AWS Step Functions"
}
```

`"who"` is the key name that your Lambda function uses to get the name of the person to greet.

4. Choose **Start Execution**.

   A new execution of your state machine starts, and a new page showing your running execution is displayed.

5. To view the results of your execution, expand the **Output** section under **Execution details**.



Execution details

Execution Status
Succeeded

Execution ARN
arn:aws:states:us-east-1:[redacted]:execution:LambdaStateMachine:HelloLambdaTest

▼ Input

```
{
  "who": "AWS Step Functions"
}
```

Started
May 18, 2018 11:13:25.778 AM

End Time
May 18, 2018 11:13:26.273 AM

▼ Output

```
"Hello, AWS Step Functions!"
```

Creating a Lambda State Machine Using AWS CloudFormation

This tutorial shows you how to create a basic AWS Lambda function using AWS CloudFormation. You use the AWS CloudFormation console and a YAML *template* to create the *stack* (IAM roles, the Lambda function, and the state machine). Then you use the AWS Step Functions console to start the state machine execution.

For more information, see Working with CloudFormation Templates and the `AWS::StepFunctions::StateMachine` resource in the *AWS CloudFormation User Guide*.

Topics

- Step 1: Set Up Your AWS CloudFormation Template (p. 23)
- Step 2: Use the AWS CloudFormation Template to Create a Lambda State Machine (p. 27)
- Step 3: Start a State Machine Execution (p. 30)

Step 1: Set Up Your AWS CloudFormation Template

Before you use the example templates (p. 27), you should understand their parts.

Topics

- To create an IAM role for Lambda (p. 23)
- To create a Lambda function (p. 24)
- To create an IAM role for the state machine execution (p. 25)
- To create a Lambda state machine (p. 26)

To create an IAM role for Lambda

Define the trust policy associated with the IAM role for the Lambda function.

YAML

```
LambdaExecutionRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
          Action: "sts:AssumeRole"
```

JSON

```
          "LambdaExecutionRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            },
                            "Action": "sts:AssumeRole"
                        }
                    ]
                }
            }
```

To create a Lambda function

Define the following properties of the Lambda function that prints the message `Hello World`.

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

YAML

```
MyLambdaFunction:
  Type: "AWS::Lambda::Function"
  Properties:
    Handler: "index.handler"
    Role: !GetAtt [ LambdaExecutionRole, Arn ]
    Code:
      ZipFile: |
        exports.handler = (event, context, callback) => {
            callback(null, "Hello World!");
        };
    Runtime: "nodejs8.10"
    Timeout: "25"
```

JSON

```
        "MyLambdaFunction": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Handler": "index.handler",
                "Role": {
                    "Fn::GetAtt": [
                        "LambdaExecutionRole",
                        "Arn"
                    ]
                },
                "Code": {
                    "ZipFile": "exports.handler = (event, context, callback) => {\n
 callback(null, \"Hello World!\");\n};\n"
                },
                "Runtime": "nodejs8.10",
                "Timeout": "25"
            }
        },
```

To create an IAM role for the state machine execution

Define the trust policy associated with the IAM role for the state machine execution.

YAML

```
StatesExecutionRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: "Allow"
          Principal:
            Service:
              - !Sub states.${AWS::Region}.amazonaws.com
          Action: "sts:AssumeRole"
    Path: "/"
    Policies:
      - PolicyName: StatesExecutionPolicy
        PolicyDocument:
          Version: "2012-10-17"
          Statement:
            - Effect: Allow
              Action:
                - "lambda:InvokeFunction"
              Resource: "*"
```

JSON

```
        "StatesExecutionRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "Service": [
                                    {
                                        "Fn::Sub": "states.
${AWS::Region}.amazonaws.com"
                                    }
                                ]
                            },
                            "Action": "sts:AssumeRole"
                        }
                    ]
                },
                "Path": "/",
                "Policies": [
                    {
                        "PolicyName": "StatesExecutionPolicy",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "lambda:InvokeFunction"
                                    ],
                                    "Resource": "*"
                                }
                            ]
                        }
                    }
                ]
            }
        },
```

To create a Lambda state machine

Define the Lambda state machine.

YAML

```
MyStateMachine:
  Type: "AWS::StepFunctions::StateMachine"
  Properties:
    DefinitionString:
      !Sub
        - |-
          {
            "Comment": "A Hello World example using an AWS Lambda function",
            "StartAt": "HelloWorld",
            "States": {
              "HelloWorld": {
                "Type": "Task",
                "Resource": "${lambdaArn}",
                "End": true
              }
            }
          }
        - {lambdaArn: !GetAtt [ MyLambdaFunction, Arn ]}
    RoleArn: !GetAtt [ StatesExecutionRole, Arn ]
```

JSON

```
        "MyStateMachine": {
            "Type": "AWS::StepFunctions::StateMachine",
            "Properties": {
                "DefinitionString": {
                    "Fn::Sub": [
                        "{\n  \"Comment\": \"A Hello World example using an AWS Lambda
 function\",\n  \"StartAt\": \"HelloWorld\",\n  \"States\": {\n    \"HelloWorld\": {\n
     \"Type\": \"Task\",\n      \"Resource\": \"${lambdaArn}\",\n      \"End\": true\n
   }\n  }\n}",
                        {
                            "lambdaArn": {
                                "Fn::GetAtt": [
                                    "MyLambdaFunction",
                                    "Arn"
                                ]
                            }
                        }
                    ]
                },
                "RoleArn": {
                    "Fn::GetAtt": [
                        "StatesExecutionRole",
                        "Arn"
                    ]
                }
            }
        }
```

Step 2: Use the AWS CloudFormation Template to Create a Lambda State Machine

After you understand the parts of the AWS CloudFormation template, you put them together and use the template to create an AWS CloudFormation stack.

To create the Lambda state machine

1. Copy the following example data to a file named `MyStateMachine.yaml` for the YAML example, or `MyStateMachine.json` for JSON.

   YAML

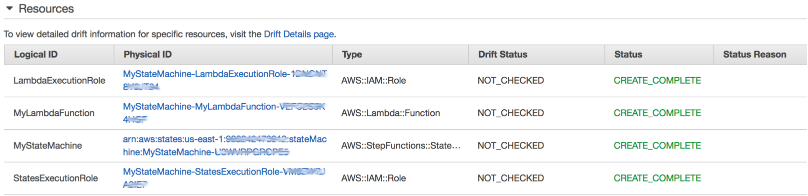
```
AWSTemplateFormatVersion: "2010-09-09"
Description: "An example template with an IAM role for a Lambda state machine."
Resources:
  LambdaExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: "sts:AssumeRole"

  MyLambdaFunction:
    Type: "AWS::Lambda::Function"
    Properties:
      Handler: "index.handler"
      Role: !GetAtt [ LambdaExecutionRole, Arn ]
      Code:
        ZipFile: |
          exports.handler = (event, context, callback) => {
              callback(null, "Hello World!");
          };
      Runtime: "nodejs8.10"
      Timeout: "25"

  StatesExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: "Allow"
            Principal:
              Service:
                - !Sub states.${AWS::Region}.amazonaws.com
            Action: "sts:AssumeRole"
      Path: "/"
      Policies:
        - PolicyName: StatesExecutionPolicy
          PolicyDocument:
            Version: "2012-10-17"
            Statement:
              - Effect: Allow
                Action:
                  - "lambda:InvokeFunction"
                Resource: "*"

  MyStateMachine:
    Type: "AWS::StepFunctions::StateMachine"
    Properties:
      DefinitionString:
        !Sub
          - |-
            {
              "Comment": "A Hello World example using an AWS Lambda function",
              "StartAt": "HelloWorld",
              "States": {
                "HelloWorld": {
                  "Type": "Task",
                  "Resource": "${lambdaArn}",
                  "End": true
                }
              }
            }
          - {lambdaArn: !GetAtt [ MyLambdaFunction, Arn ]}
      RoleArn: !GetAtt [ StatesExecutionRole, Arn ]
```

JSON

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "An example template with an IAM role for a Lambda state
 machine.",
    "Resources": {
        "LambdaExecutionRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            },
                            "Action": "sts:AssumeRole"
                        }
                    ]
                }
            }
        },
        "MyLambdaFunction": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Handler": "index.handler",
                "Role": {
                    "Fn::GetAtt": [
                        "LambdaExecutionRole",
                        "Arn"
                    ]
                },
                "Code": {
                    "ZipFile": "exports.handler = (event, context, callback) => {\n
    callback(null, \"Hello World!\");\n};\n"
                },
                "Runtime": "nodejs8.10",
                "Timeout": "25"
            }
        },
        "StatesExecutionRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "Service": [
                                    {
                                        "Fn::Sub": "states.
${AWS::Region}.amazonaws.com"
                                    }
                                ]
                            },
                            "Action": "sts:AssumeRole"
                        }
                    ]
                },
                "Path": "/",
                "Policies": [
                    {
                        "PolicyName": "StatesExecutionPolicy",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "lambda:InvokeFunction"
                                    ],
                                    "Resource": "*"
                                }
                            ]
                        }
                    }
                ]
            }
        },
        "MyStateMachine": {
            "Type": "AWS::StepFunctions::StateMachine",
            "Properties": {
                "DefinitionString": {
                    "Fn::Sub": [
                        "{\n  \"Comment\": \"A Hello World example using
 an AWS Lambda function\",\n  \"StartAt\": \"HelloWorld\",\n  \"States\":
 {\n    \"HelloWorld\": {\n      \"Type\": \"Task\",\n      \"Resource\":
 \"${lambdaArn}\",\n      \"End\": true\n    }\n  }\n}",
                        {
                            "lambdaArn": {
                                "Fn::GetAtt": [
                                    "MyLambdaFunction",
                                    "Arn"
                                ]
                            }
                        }
                    ]
                },
                "RoleArn": {
                    "Fn::GetAtt": [
                        "StatesExecutionRole",
                        "Arn"
                    ]
                }
            }
        }
    }
}
```

2. Open the AWS CloudFormation console and choose **Create Stack**.
3. On the **Select Template** page, select **Upload a template to Amazon S3**. Choose your `MyStateMachine` file, and then choose **Next**.
4. On the **Specify Details** page, for **Stack name**, enter `MyStateMachine`, and then choose **Next**.
5. On the **Options** page, choose **Next**.
6. On the **Review** page, choose **I acknowledge that AWS CloudFormation might create IAM resources.** and then choose **Create**.

   AWS CloudFormation begins to create the `MyStateMachine` stack and displays the **CREATE_IN_PROGRESS** status. When the process is complete, AWS CloudFormation displays the **CREATE_COMPLETE** status.
7. (Optional) To display the resources in your stack, select the stack and choose the **Resources** tab.



▾ Resources

To view detailed drift information for specific resources, visit the Drift Details page.

Logical ID	Physical ID	Type	Drift Status	Status	Status Reason
LambdaExecutionRole	MyStateMachine-LambdaExecutionRole-1[redacted]	AWS::IAM::Role	NOT_CHECKED	CREATE_COMPLETE	
MyLambdaFunction	MyStateMachine-MyLambdaFunction-V[redacted]	AWS::Lambda::Function	NOT_CHECKED	CREATE_COMPLETE	
MyStateMachine	arn:aws:states:us-east-1:[redacted]:stateMachine:MyStateMachine-U[redacted]	AWS::StepFunctions::State...	NOT_CHECKED	CREATE_COMPLETE	
StatesExecutionRole	MyStateMachine-StatesExecutionRole-V[redacted]	AWS::IAM::Role	NOT_CHECKED	CREATE_COMPLETE	

Step 3: Start a State Machine Execution

After you create your Lambda state machine, you start an execution.

To start the state machine execution

1. Open the Step Functions console and choose the name of the state machine that you created using AWS CloudFormation.
2. On the ***`MyStateMachine-ABCDEFGHIJ1K`*** page, choose **New execution**.

   The **New execution** page is displayed.
3. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   Note
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
4. Choose **Start Execution**.

A new execution of your state machine starts, and a new page showing your running execution is displayed.

5. (Optional) In the **Execution Details**, review the **Execution Status** and the **Started** and **Closed** timestamps.
6. To view the results of your execution, choose **Output**.

Creating an Activity State Machine

This tutorial shows you how to create an activity-based state machine using Java and AWS Step Functions. Activities allow you to control worker code that runs somewhere else in your state machine. For an overview, see Activities (p. 94) in How Step Functions Works (p. 86).

To complete this tutorial, you need the following:

- The SDK for Java. The example activity in this tutorial is a Java application that uses the AWS SDK for Java to communicate with AWS.
- AWS credentials in the environment or in the standard AWS configuration file. For more information, see Set Up Your AWS Credentials in the *AWS SDK for Java Developer Guide*.

Topics

Step 1: Create a New Activity

You must make Step Functions aware of the *activity* whose *worker* (a program) you want to create. Step Functions responds with an ARN that establishes an identity for the activity. Use this identity to coordinate the information passed between your state machine and worker.

> **Important**
> Ensure that your activity task is under the same AWS account as your state machine.

1. In the Step Functions console, choose **Activities** in the left navigation panel.
2. Choose **Create activity**.
3. Enter an **Activity Name**, for example, *`get-greeting`*, and then choose **Create Activity**.
4. When your activity task is created, note its Amazon Resource Name (ARN), for example:

```
arn:aws:states:us-east-1:123456789012:activity:get-greeting
```

Step 2: Create a State Machine

Create a state machine that determines when your activity is invoked and when your worker should perform its primary work, collect its results, and return them.

1. In the Step Functions console, choose **State machines** in the left navigation panel.
2. On the **State machines** page, choose **Create state machine**, choose **Author with code snippets**, and then enter a name under **Details** (for example, *ActivityStateMachine*).

   Note
   State machine, execution, and activity names must be 1–80 characters in length, must be unique for your account and AWS Region, and must not contain any of the following:

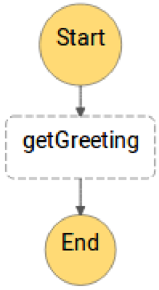
   - Whitespace
   - Wildcard characters (`? *`)
   - Bracket characters (`< > { } [ ]`)
   - Special characters (`: ; , \ | ^ ~ $ # % & ` "`)
   - Control characters (`\\u0000` - `\\u001f` or `\\u007f` - `\\u009f`).

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

   Under **State machine definition**, enter the following code, and include the ARN of the activity task that you created earlier (p. 31) in the `Resource` field. For example:

```
{
  "Comment": "An example using a Task state.",
  "StartAt": "getGreeting",
  "Version": "1.0",
  "TimeoutSeconds": 300,
  "States":
  {
    "getGreeting": {
      "Type": "Task",
      "Resource": "arn:aws:states:us-east-1:123456789012:activity:get-greeting",
      "End": true
    }
  }
}
```

   This is a description of your state machine using the Amazon States Language. It defines a single `Task` state named `getGreeting`. For more information, see State Machine Structure (p. 88).
3. Use the graph in the **Visual Workflow** pane to check that your Amazon States Language code describes your state machine correctly.



   If you don't see the graph, choose ↻ in the **Visual Workflow** pane.
4. Choose **Next**.
5. Create or enter an IAM role:

- To create an IAM role for Step Functions, select **Create an IAM role for me**, and enter a **Name** for your role.
- If you have previously created an IAM role (p. 231) with the correct permissions for your state machine, select **Choose an existing IAM role**. Select a role from the list, or provide an ARN for that role.

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

6. Choose **Create state machine**.

Step 3: Implement a Worker

Create a *worker*. A worker is a program that is responsible for:

- Polling Step Functions for activities using the `GetActivityTask` API action.
- Performing the work of the activity using your code, (for example, the `getGreeting()` method in the following code).
- Returning the results using the `SendTaskSuccess`, `SendTaskFailure`, and `SendTaskHeartbeat` API actions.

> **Note**
> For a more complete example of an activity worker, see Example Activity Worker in Ruby (p. 96). This example provides an implementation based on best practices, which you can use as a reference for your activity worker. The code implements a consumer-producer pattern with a configurable number of threads for pollers and activity workers.

To implement the worker

1. Create a file named `GreeterActivities.java`.
2. Add the following code to it.

```
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.EnvironmentVariableCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.stepfunctions.AWSStepFunctions;
import com.amazonaws.services.stepfunctions.AWSStepFunctionsClientBuilder;
import com.amazonaws.services.stepfunctions.model.GetActivityTaskRequest;
import com.amazonaws.services.stepfunctions.model.GetActivityTaskResult;
import com.amazonaws.services.stepfunctions.model.SendTaskFailureRequest;
import com.amazonaws.services.stepfunctions.model.SendTaskSuccessRequest;
import com.amazonaws.util.json.Jackson;
import com.fasterxml.jackson.databind.JsonNode;
import java.util.concurrent.TimeUnit;


public class GreeterActivities {

    public String getGreeting(String who) throws Exception {
        return "{\"Hello\": \"" + who + "\"}";
    }

    public static void main(final String[] args) throws Exception {
        GreeterActivities greeterActivities = new GreeterActivities();
        ClientConfiguration clientConfiguration = new ClientConfiguration();
        clientConfiguration.setSocketTimeout((int)TimeUnit.SECONDS.toMillis(70));

        AWSStepFunctions client = AWSStepFunctionsClientBuilder.standard()
                .withRegion(Regions.US_EAST_1)
                .withCredentials(new EnvironmentVariableCredentialsProvider())
                .withClientConfiguration(clientConfiguration)
                .build();

        while (true) {
            GetActivityTaskResult getActivityTaskResult =
                    client.getActivityTask(
                            new
 GetActivityTaskRequest().withActivityArn(ACTIVITY_ARN));

            if (getActivityTaskResult.getTaskToken() != null) {
                try {
                    JsonNode json =
 Jackson.jsonNodeOf(getActivityTaskResult.getInput());
                    String greetingResult =
                            greeterActivities.getGreeting(json.get("who").textValue());
                    client.sendTaskSuccess(
                            new SendTaskSuccessRequest().withOutput(

 greetingResult).withTaskToken(getActivityTaskResult.getTaskToken()));
                } catch (Exception e) {
                    client.sendTaskFailure(new SendTaskFailureRequest().withTaskToken(
                            getActivityTaskResult.getTaskToken()));
                }
            } else {
                Thread.sleep(1000);
            }
        }
    }
}
```

Note

The `EnvironmentVariableCredentialsProvider` class in this example assumes that the `AWS_ACCESS_KEY_ID` (or `AWS_ACCESS_KEY`) and `AWS_SECRET_KEY` (or `AWS_SECRET_ACCESS_KEY`) environment variables are set. For more information about providing the required credentials to the factory, see AWSCredentialsProvider in the *AWS SDK for Java API Reference* and Set Up AWS Credentials and Region for Development in the *AWS SDK for Java Developer Guide*.

By default the AWS SDK will wait up to 50 seconds to receive data from the server for any operation. The `GetActivityTask` operation is a long-poll operation that will wait up to 60 seconds for the next available task. To prevent receiving a `SocketTimeoutException` errors, set SocketTimeout to 70 seconds.

3. In the parameter list of the `GetActivityTaskRequest().withActivityArn()` constructor, replace the `ACTIVITY_ARN` value with the ARN of the activity task that you created earlier (p. 31).

Step 4: Start an Execution

When you start the execution of the state machine, your worker polls Step Functions for activities, performs its work (using the input that you provide), and returns its results.

1. On the ***`ActivityStateMachine`*** page, choose **Start execution**.

   The **New execution** page is displayed.

2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   > **Note**
   > Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

3. In the execution input area, replace the example data with the following.

   ```
   {
       "who" : "AWS Step Functions"
   }
   ```

4. Choose **Start Execution**.

   A new execution of your state machine starts, and a new page showing your running execution is displayed.

5. In the **Execution Details** section, choose **Info** to view the **Execution Status** and the **Started** and **Closed** timestamps.
6. In the **Execution Details** section, expand the **Output** section to view the output of your workflow.

Step 5: Run and Stop the Worker

To have the worker poll your state machine for activities, you must run the worker.

1. On the command line, navigate to the directory in which you created `GreeterActivities.java`.
2. To use the AWS SDK, add the full path of the `lib` and `third-party` directories to the dependencies of your build file and to your Java `CLASSPATH`. For more information, see Downloading and Extracting the SDK in the *AWS SDK for Java Developer Guide*.
3. Compile the file.

   ```
   $ javac GreeterActivities.java
   ```

4. Run the file.

   ```
   $ java GreeterActivities
   ```

5. In the Step Functions console, navigate to the **Execution Details** page.
6. When the execution completes, choose **Output** to see the results of your execution.
7. Stop the worker.

Handling Error Conditions Using a State Machine

In this tutorial, you create an AWS Step Functions state machine with a `Catch` field, which uses an AWS Lambda function to respond with conditional logic based on error message type. This is a technique called *function error handling*.

For more information, see Function Error Handling in the *AWS Lambda Developer Guide*.

> **Note**
> You can also create state machines that `Retry` on timeouts or those that use `Catch` to transition to a specific state when an error or timeout occurs. For examples of these error handling techniques, see Examples Using Retry and Using Catch (p. 135).

Topics

Step 1: Create an IAM Role for Lambda

Both AWS Lambda and AWS Step Functions can execute code and access AWS resources (for example, data stored in Amazon S3 buckets). To maintain security, you must grant Lambda and Step Functions access to these resources.

Lambda requires you to assign an AWS Identity and Access Management (IAM) role when you create a Lambda function, in the same way Step Functions requires you to assign an IAM role when you create a state machine.

1. Sign in to the IAM console and choose **Roles**, **Create role**.
2. On the **Select type of trusted entity** page, under **AWS service**, select **Lambda** from the list, and then choose **Next: Permissions**.

   Note

   The role is automatically provided with a trust relationship that allows Lambda to use the role.
3. On the **Attach permissions policy** page, choose **Next: Review**.
4. On the **Review** page, enter `MyLambdaRole` for **Role Name**, and then choose **Create role**.

The IAM role appears in the list of roles.

Step 2: Create a Lambda Function That Fails

Use a Lambda function to simulate an error condition.

> **Important**
>
> Ensure that your Lambda function is under the same AWS account and AWS Region as your state machine.

1. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.

   Choose **Create a function**.
2. In the **Blueprints** section, enter `step-functions` into the filter, and then choose the **step-functions-error** blueprint.
3. In the **Basic information** section, configure your Lambda function:

   a. For **Name**, enter `FailFunction`.

   b. For **Role**, select **Choose an existing role**.

   c. For **Existing role**, choose the Lambda role that you created earlier (p. 36).

      Note

      If the IAM role that you created doesn't appear in the list, the role might still need a few minutes to propagate to Lambda.
4. The following code is displayed in the **Lambda function code** pane.

```
'use strict';
```

```
exports.handler = (event, context, callback) => {
    function CustomError(message) {
        this.name = 'CustomError';
        this.message = message;
    }
    CustomError.prototype = new Error();

    const error = new CustomError('This is a custom error!');
    callback(error);
};
```

The `context` object returns the error message `This is a custom error!`.

5. Choose **Create function**.

   When your Lambda function is created, note its Amazon Resource Name (ARN) in the upper-right corner of the page. For example:

```
arn:aws:lambda:us-east-1:123456789012:function:FailFunction
```

Step 3: Test the Lambda Function

Test your Lambda function to see it in operation.

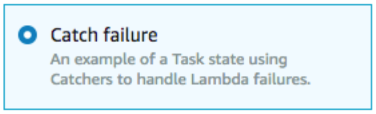
1. On the ***`FailFunction`*** page, choose **Test**.
2. In the **Configure test event** dialog box, enter `FailFunction` for **Event name**, and then choose **Create**.
3. On the ***`FailFunction`*** page, **Test** your Lambda function.

   The results of the test (the simulated error) are displayed at the bottom of the page.

Step 4: Create a State Machine with a Catch Field

Use the Step Functions console to create a state machine that uses a `Task` state with a `Catch` field. Add a reference to your Lambda function in the `Task` state. The Lambda function is invoked and fails during execution. Step Functions retries the function twice using exponential backoff between retries.

1. Open the Step Functions console and choose **Create state machine**.
2. On the **Create a state machine** page, choose **Templates**, and then choose **Catch failure**.



● Catch failure
An example of a Task state using Catchers to handle Lambda failures.

3. **Name your state machine**, for example, *`Catchfailure`*.

   Note
   State machine, execution, and activity names must be 1–80 characters in length, must be unique for your account and AWS Region, and must not contain any of the following:

   - Whitespace
   - Wildcard characters (`? *`)
   - Bracket characters (`< > { } [ ]`)
   - Special characters (``: ; , \ | ^ ~ $ # % & ` "``)

- Control characters (`\\u0000` - `\\u001f` or `\\u007f` - `\\u009f`).

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

4. In the **Code** pane, add the ARN of the Lambda function that you created earlier (p. 36) to the `Resource` field, for example:

```
{
   "Comment": "A Catch example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "CreateAccount",
   "States": {
      "CreateAccount": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FailFunction",
         "Catch": [ {
            "ErrorEquals": ["CustomError"],
            "Next": "CustomErrorFallback"
         }, {
            "ErrorEquals": ["States.TaskFailed"],
            "Next": "ReservedTypeFallback"
         }, {
            "ErrorEquals": ["States.ALL"],
            "Next": "CatchAllFallback"
         } ],
         "End": true
      },
      "CustomErrorFallback": {
         "Type": "Pass",
         "Result": "This is a fallback from a custom Lambda function exception",
         "End": true
      },
      "ReservedTypeFallback": {
         "Type": "Pass",
         "Result": "This is a fallback from a reserved error code",
         "End": true
      },
      "CatchAllFallback": {
         "Type": "Pass",
         "Result": "This is a fallback from any error code",
         "End": true
      }
   }
}
```

This is a description of your state machine using the Amazon States Language. It defines a single `Task` state named `CreateAccount`. For more information, see State Machine Structure (p. 88).

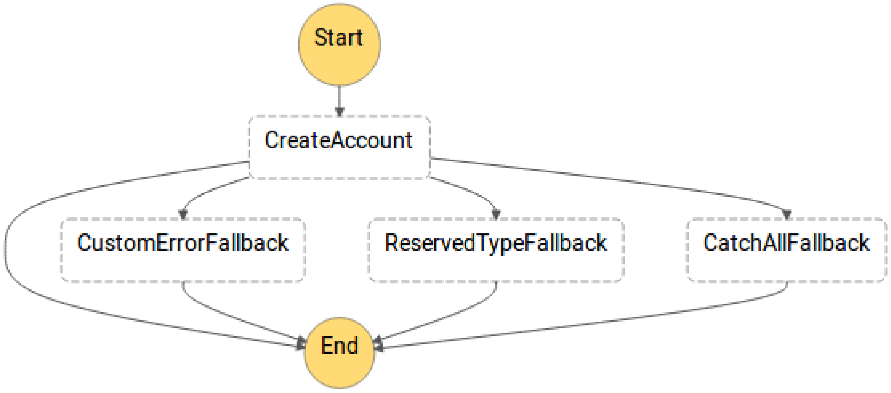
For more information about the syntax of the `Retry` field, see Examples Using Retry and Using Catch (p. 135).

Note

Unhandled errors in Lambda are reported as `Lambda.Unknown` in the error output. These include out-of-memory errors, function timeouts, and hitting the concurrent Lambda invoke limit. You can match on `Lambda.Unknown`, `States.ALL`, or `States.TaskFailed` to handle these errors. When Lambda hits the invocation limit, the error is `Lambda.TooManyRequestsException`. For more information about Lambda

`Handled` and `Unhandled` errors, see `FunctionError` in the AWS Lambda Developer Guide.

5. Use the graph in the **Visual Workflow** pane to check that your Amazon States Language code describes your state machine correctly.



If you don't see the graph, choose ↻ in the **Visual Workflow** pane.

6. Choose **Next**.
7. Create or enter an IAM role:

   - To create an IAM role for Step Functions, select **Create an IAM role for me**, and enter a **Name** for your role.
   - If you have previously created an IAM role (p. 231) with the correct permissions for your state machine, select **Choose an existing IAM role**. Select a role from the list, or provide an ARN for that role.

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

Step 5: Start a New Execution

After you create your state machine, you can start an execution.

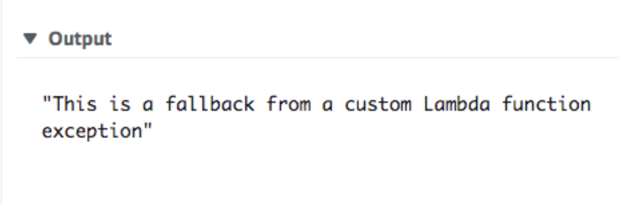
1. On the *`CatchStateMachine`* page, choose **New execution**.

   The **New execution** page is displayed.
2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   Note
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. Choose **Start Execution**.

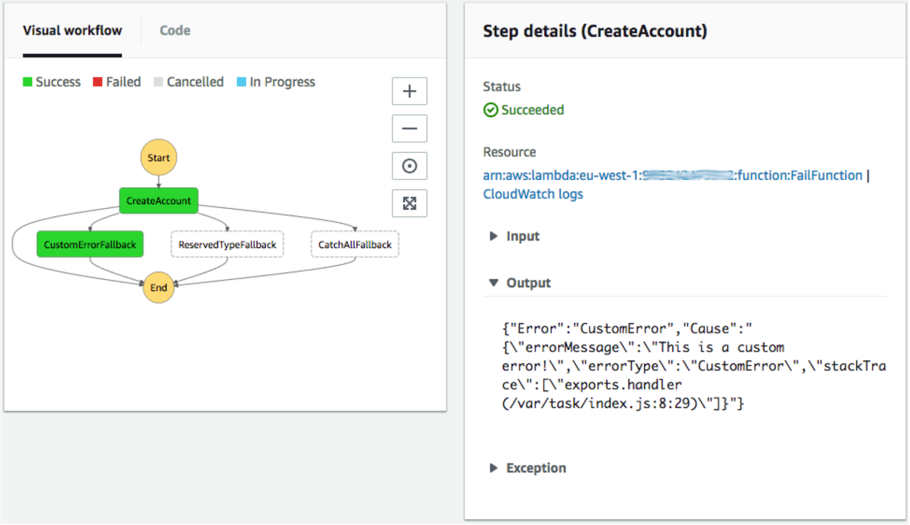
   A new execution of your state machine starts, and a new page showing your running execution is displayed.

4. In the **Execution Details** section, expand the **Output** section to view the output of your workflow.



```
"This is a fallback from a custom Lambda function
exception"
```

5. To view your custom error message, select `CreateAccount` in the **Visual workflow** and expand the **Output** section.



Status
Succeeded

Resource
arn:aws:lambda:eu-west-1:[redacted]:function:FailFunction | CloudWatch logs

```
{"Error":"CustomError","Cause":"
{\"errorMessage\":\"This is a custom
error!\",\"errorType\":\"CustomError\",\"stackTra
ce\":[\"exports.handler
(/var/task/index.js:8:29)\"]}"}
```

Note
You can preserve the state input with the error by using `ResultPath`. See Use ResultPath to Include Both Error and Input in a `Catch` (p. 125)

Periodically Start a State Machine Execution Using CloudWatch Events

You can execute a Step Functions state machine in response to an event pattern or on a schedule using Amazon CloudWatch Events. This tutorial shows you how to set a state machine as a target for a CloudWatch Events rule that starts the execution of a state machine every five minutes.

For more information about setting a Step Functions state machine as a target using the `PutTarget` Amazon CloudWatch Events API action, see Add a Step Functions state machine as a target.

Topics

Step 1: Create a State Machine

Before you can set a CloudWatch Events target, you must create a state machine.

- To create a basic state machine, use the Getting Started (p. 13) tutorial.

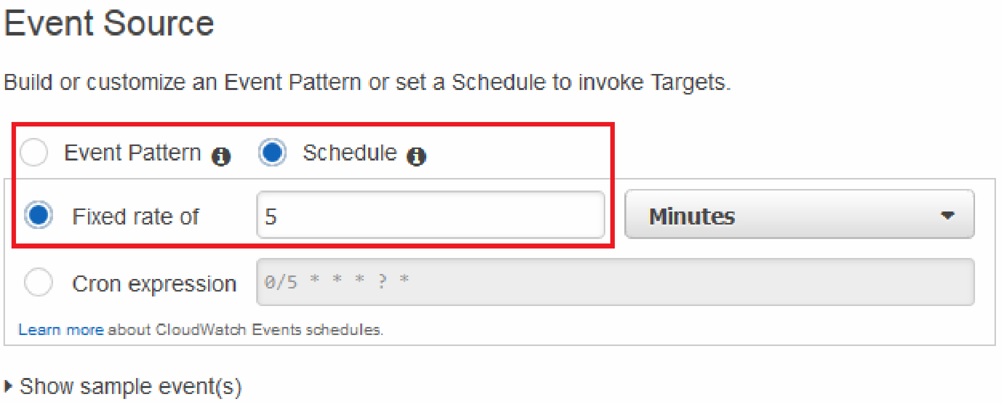
- If you already have a state machine, proceed to the next step.

Step 2: Create a CloudWatch Events Rule

After you create your state machine, you can create your CloudWatch Events rule.

1. Navigate to the CloudWatch Events console, choose **Events**, and then choose **Create Rule**.

   The **Step 1: Create rule** page is displayed.
2. In the **Event source** section, choose **Schedule**, and then enter 5 for **Fixed rate of**.



Event Source

Build or customize an Event Pattern or set a Schedule to invoke Targets.

○ Event Pattern ● Schedule

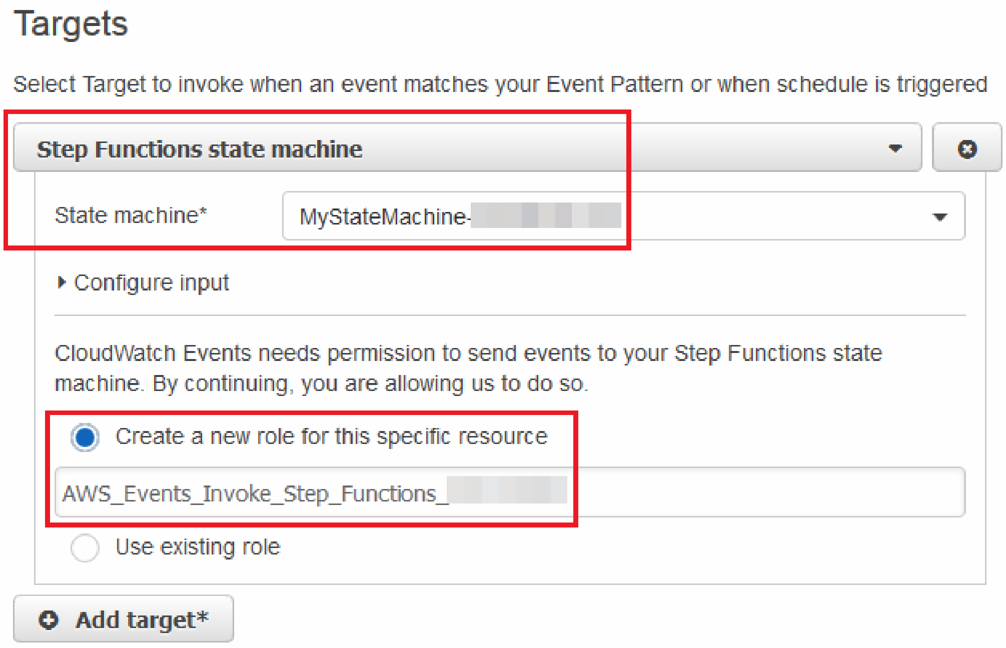
● Fixed rate of 5 Minutes

○ Cron expression 0/5 * * * ? *

Learn more about CloudWatch Events schedules.

Show sample event(s)

3. In the **Targets** section, choose **Add target**, and then from the list choose **Step Functions state machine**.



Targets

Select Target to invoke when an event matches your Event Pattern or when schedule is triggered

Step Functions state machine

State machine* MyStateMachine-[redacted]

Configure input

CloudWatch Events needs permission to send events to your Step Functions state machine. By continuing, you are allowing us to do so.

● Create a new role for this specific resource

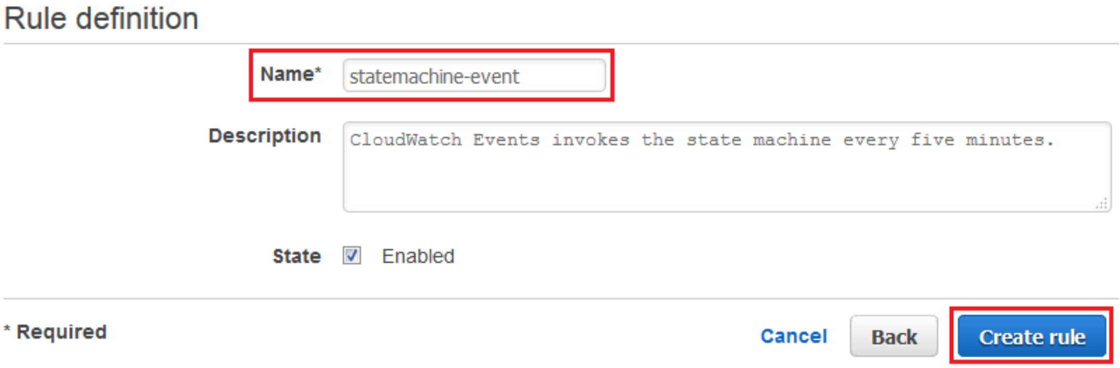
AWS_Events_Invoke_Step_Functions_[redacted]

○ Use existing role

4. CloudWatch Events can create the IAM role needed for your event to run:
   - To create an IAM role automatically, select **Create a new role for this specific resource**.
   - To use an IAM role that you created previously, choose **Use existing role**.
5. Choose **Configure details**.

The **Step 2: Configure rule details** page is displayed.

6. Enter a **Name** for your rule (for example, `statemachine-event`), choose **Enabled** for **State**, and then choose **Create rule**.



Rule definition

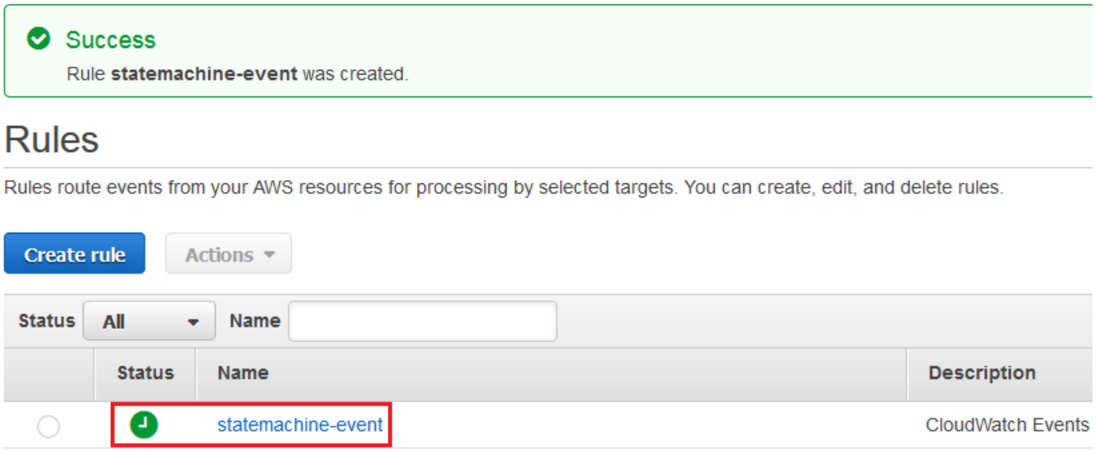
Name* statemachine-event

Description CloudWatch Events invokes the state machine every five minutes.

State ☑ Enabled

* Required

The rule is created and the **Rules** page is displayed, listing all your CloudWatch Events rules.



Success

Rule **statemachine-event** was created.

Rules

Rules route events from your AWS resources for processing by selected targets. You can create, edit, and delete rules.

	Status	Name	Description
○	●	statemachine-event	CloudWatch Events

A new execution of your state machine starts every five minutes.

Starting a State Machine Execution in Response to Amazon S3 Events

You can use Amazon CloudWatch Events to execute a Step Functions state machine in response to an event or on a schedule.

This tutorial shows you how to configure a state machine as a target for a CloudWatch Events rule. This will start an execution when files are added to an Amazon S3 bucket.

For a practical application, you could launch a state machine that performs operations on files that you add to the bucket, such as creating thumbnails or running Amazon Rekognition analysis on image and video files.

For this tutorial you start an execution of a simple `Helloworld`state machine by adding a file to an Amazon S3 bucket. Then you review example input of that execution to show what information is included in the input from CloudTrail.

Topics

Prerequisite: Create a State Machine

Before you can configure a CloudWatch Events target, you must create a state machine.

- To create a basic state machine, use the Getting Started (p. 13) tutorial.
- If you already have a `Helloworld`state machine, proceed to the next step.

Step 1: Create a Bucket in Amazon S3

Now that you have a `Helloworld` state machine, you need an Amazon S3 bucket. In Step 3 of this tutorial, you set up a rule so that when a file is added to this bucket, CloudWatch Events triggers an execution of the state machine.

1. Navigate to the Amazon S3 console, and then choose **Create bucket**.
2. Enter a **Bucket name**, such as *`username`*`-sfn-tutorial`.

   Note
   Bucket names must be unique across all existing bucket names in all AWS Regions in Amazon S3. Use your own *`username`* to make this name unique. You need to create all resources in the same AWS Region.
3. Choose **Create**.

Step 2: Create a Trail in AWS CloudTrail

After you create an Amazon S3 bucket, create a trail in CloudTrail.

For API events in Amazon S3 to match your CloudWatch Events rule, you must configure a trail in CloudTrail to receive those events.

1. Navigate to the AWS CloudTrail console, choose **View trails**, and then choose **Create trail**.
2. For **Trail name**, enter `S3Event`.
3. On the **S3** tab, choose **Add S3 bucket**.
4. For **Bucket name**, enter the name of the Amazon S3 bucket you created earlier: *`username`*`-sfn-tutorial` (Step 1: Create a Bucket in Amazon S3 (p. 43)).
5. Under **Storage location**, choose **Yes** next to **Create a new S3 bucket**.
6. For **S3 bucket**, enter a name for a new bucket to store information about the actions of the Amazon S3 bucket you created earlier.

   Note
   This bucket name must be unique across all of Amazon S3. Include your *`username`* in the bucket name so that the name will be unique: *`username`*`-sfn-tutorial-storage`.
7. Choose **Create**.

Step 3: Create a CloudWatch Events Rule

After you have a state machine, and have created the Amazon S3 bucket and a trail in AWS CloudTrail, create your Amazon CloudWatch Events rule.

> **Note**
> You must configure CloudWatch Events in the same AWS Region as the Amazon S3 bucket.

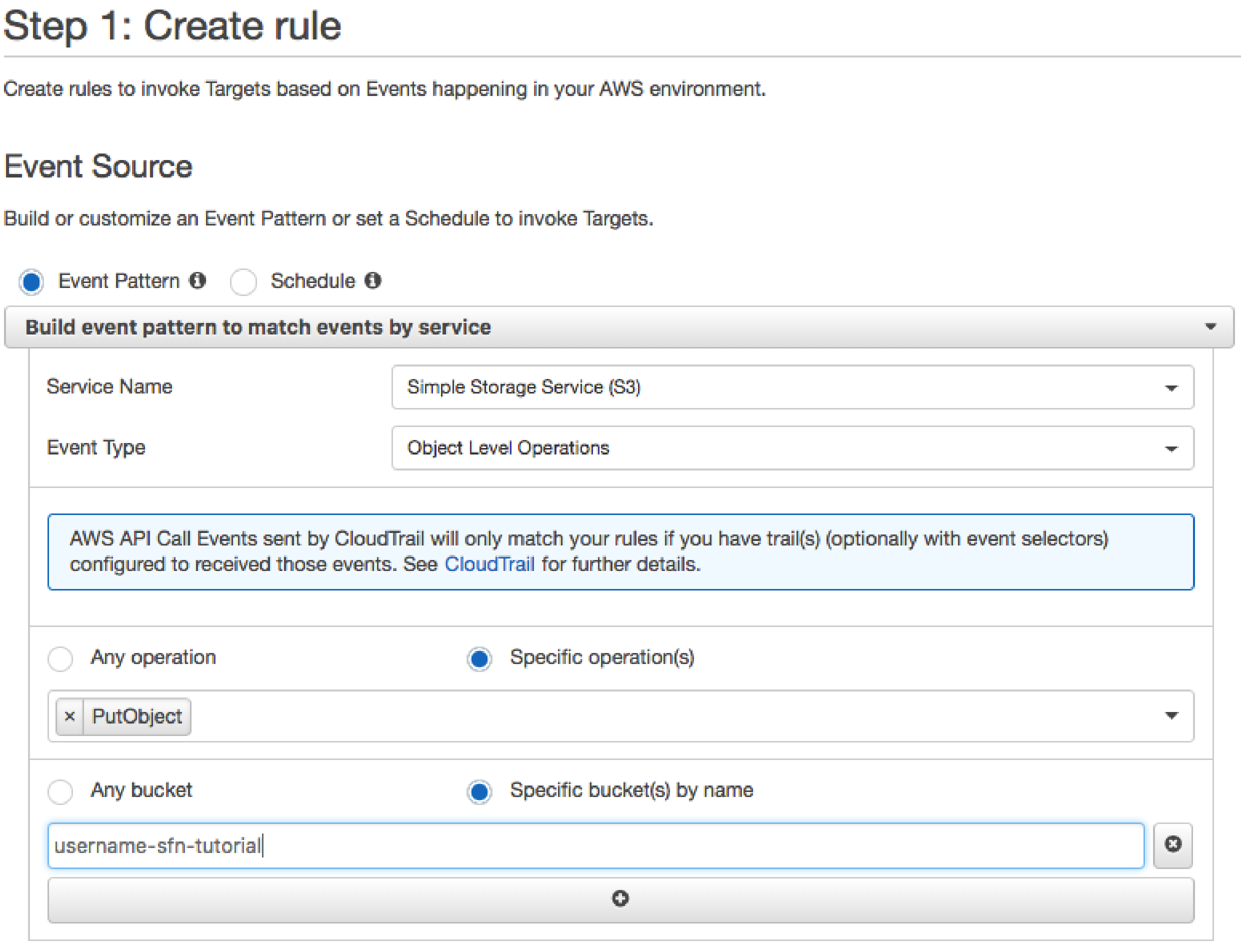
To create the rule

1. Navigate to the CloudWatch console, choose **Events**, and then choose **Create Rule**.

   The **Step 1: Create rule** page is displayed.
2. In **Event source**, choose **Event Pattern**.
3. For **Service Name**, choose **Simple Storage Service (S3)**.
4. For **Event Type**, choose **Object Level Operations**.
5. Choose **Specific operation(s)**, and then choose **PutObject**.

   > **Note**
   > If the object size is bigger than the Multipart threshold used in the `PutObject` operation, the AWS CloudTrail API logged will be `CompleteMultipartUpload` instead of `PutObject`. See, Multipart Upload Overview in the AWS CloudTrail User Guide.
6. Choose **Specific bucket(s) by name** and enter the bucket name you created in Step 1 (`username-sfn-tutorial`).

The **Event Source** page should look like the following.



Step 1: Create rule

Create rules to invoke Targets based on Events happening in your AWS environment.

Event Source

Build or customize an Event Pattern or set a Schedule to invoke Targets.

● Event Pattern ○ Schedule

Build event pattern to match events by service

Service Name: Simple Storage Service (S3)

Event Type: Object Level Operations

AWS API Call Events sent by CloudTrail will only match your rules if you have trail(s) (optionally with event selectors) configured to received those events. See CloudTrail for further details.

○ Any operation ● Specific operation(s)

PutObject

○ Any bucket ● Specific bucket(s) by name

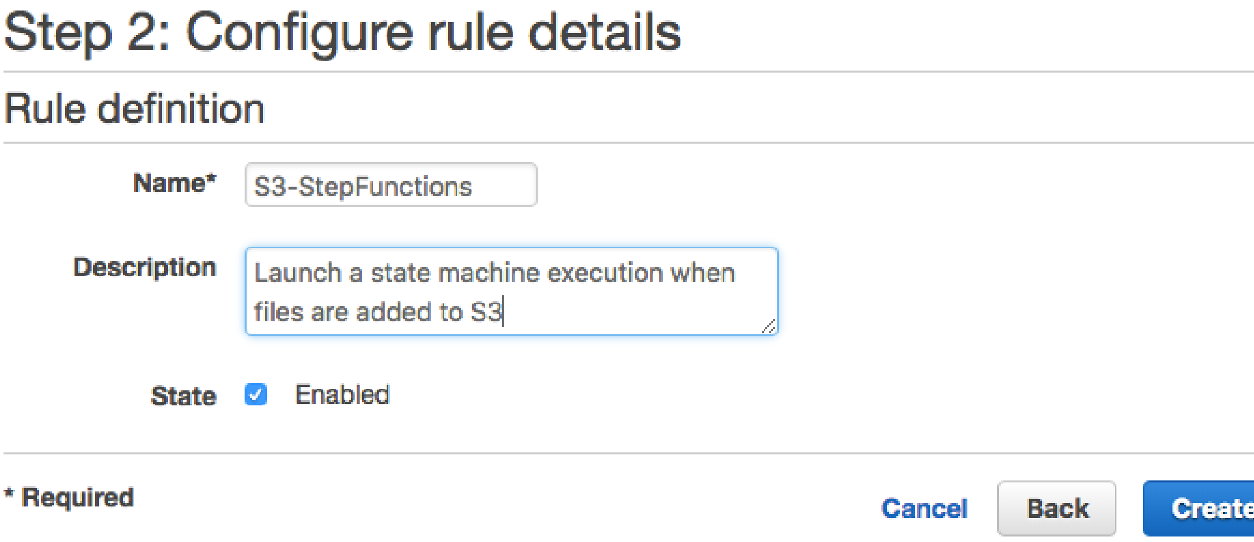
username-sfn-tutorial

To create the target

1. In the **Targets** section, choose **Add target**.
2. From the list, choose **Step Functions state machine**, and in the **State machine** list, choose the state machine from Step 1 (`Helloworld`).
3. CloudWatch Events can create the IAM role that your event needs to run:
   - To create an IAM role automatically, choose **Create a new role for this specific resource**.
   - To use an IAM role that you created before, choose **Use existing role**.
4. Choose **Configure details**.

   The **Step 2: Configure rule details** page is displayed.
5. Enter a **Name** for your rule (for example, `S3StepFunctions`), choose **Enabled** for **State**, and then choose **Create rule**.

   The **Configure rule details** section should look like the following.



Step 2: Configure rule details

Rule definition

Name* S3-StepFunctions

Description Launch a state machine execution when files are added to S3

State ☑ Enabled

* Required

The rule is created and the **Rules** page is displayed, listing all your CloudWatch Events rules.

Step 4: Test the CloudWatch Rule

Now that everything is in place, test adding a file to the Amazon S3 bucket, and then look at the input of the resulting state machine execution.

1. Add a file to your Amazon S3 bucket.

   Navigate to the Amazon S3 console, select the bucket you created (*username*-sfn-tutorial), and then choose **Upload**.
2. Add a file (test.png in the following example), and then choose **Upload**.

   This launches an execution of your state machine, passing information from AWS CloudTrail as the input.
3. Check the execution for your state machine.

   Navigate to the Step Functions console and select the state machine used in your CloudWatch Events rule (Helloworld).
4. Select the most recent execution of that state machine and expand the **Input** section.

   This input includes information such as the bucket name and the object name. In a real-world use case, a state machine can use this input to perform actions on that object.

Example of Execution Input

The following example shows typical input to the state machine execution.

```
{
  "version": "0",
  "id": "8d6f9246-b781-44f8-a026-f1c1ab2c61f0",
  "detail-type": "AWS API Call via CloudTrail",
  "source": "aws.s3",
  "account": "123456789012",
  "time": "2018-09-12T00:25:10Z",
  "region": "us-east-2",
  "resources": [],
  "detail": {
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "AKIAIOSFODNN7EXAMPLE",
      "arn": "arn:aws:iam::123456789012:user/username",
      "accountId": "123456789012",
      "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
      "userName": "username",
      "sessionContext": {
        "attributes": {
          "creationDate": "2018-09-11T20:10:38Z",
          "mfaAuthenticated": "true"
        }
      },
      "invokedBy": "signin.amazonaws.com"
    },
    "eventTime": "2018-09-12T00:25:10Z",
    "eventSource": "s3.amazonaws.com",
    "eventName": "PutObject",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "203.0.113.34",
    "userAgent": "signin.amazonaws.com",
    "requestParameters": {
      "X-Amz-Date": "20180912T002509Z",
      "bucketName": "username-sfn-tutorial",
      "X-Amz-Algorithm": "AWS4-HMAC-SHA256",
      "x-amz-acl": "private",
      "X-Amz-SignedHeaders": "content-type;host;x-amz-acl;x-amz-storage-class",
      "X-Amz-Expires": "300",
      "key": "test.png",
      "x-amz-storage-class": "STANDARD"
    },
    "responseElements": null,
    "additionalEventData": {
      "x-amz-id-2": "IOWQ4fDEXAMPLEQM+ey7N9WgVhSnQ6JEXAMPLEZb7hSQDASK+Jd1vEXAMPLEa3Km"
    },
    "requestID": "79104EXAMPLEB723",
    "eventID": "cdc4b7ed-e171-4cef-975a-ad829d4123e8",
    "readOnly": false,
    "resources": [
      {
        "type": "AWS::S3::Object",
        "ARN": "arn:aws:s3:::username-sfn-tutorial-2/test.png"
      },
      {
        "accountId": "123456789012",
        "type": "AWS::S3::Bucket",
        "ARN": "arn:aws:s3:::username-sfn-tutorial"
      }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
  }
}
```

Creating a Step Functions API Using API Gateway

You can use Amazon API Gateway to associate your AWS Step Functions APIs with methods in an API Gateway API. When an HTTPS request is sent to an API method, API Gateway invokes your Step Functions API actions.

This tutorial shows you how to create an API that uses one resource and the `POST` method to communicate with the `StartExecution` API action. You'll use the IAM console to create a role for API Gateway. Then, you'll use the API Gateway console to create an API Gateway API, create a resource and method, and map the method to the `StartExecution` API action. Finally, you'll deploy and test your API. For more information about this API action, see StartExecution in the *AWS Step Functions API Reference*.

> **Note**
> Although Amazon API Gateway can start a Step Functions execution by calling `StartExecution`, you must call `DescribeExecution` to get the result.

Topics

Step 1: Create an IAM Role for API Gateway

Before you create your API Gateway API, you need to give API Gateway permission to call Step Functions API actions.

1. Log in to the IAM console and choose **Roles**, **Create role**.
2. On the **Select type of trusted entity** page, under **AWS service**, select **API Gateway** from the list and then choose **Next: Permissions**.
3. On the **Attached permissions policy** page, choose **Next: Review**.
4. On the **Review** page, enter `APIGatewayToStepFunctions` for **Role name** and then choose **Create role**.

   The IAM role appears in the list of roles.
5. Choose the name of your role and note the **Role ARN**, for example:

```
arn:aws:iam::123456789012:role/APIGatewayToStepFunctions
```

Attach a policy to the IAM role

1. On the **Roles** page, search for your role (`APIGatewayToStepFunctions`) and then choose the role.
2. On the **Permissions** tab, choose **Attach Policy**.
3. On the **Attach Policy** page, search for `AWSStepFunctionsFullAccess`, choose the policy, and then choose **Attach Policy**.

Step 2: Create your API Gateway API

After you create your IAM role, you can create your custom API Gateway API.

Create the API

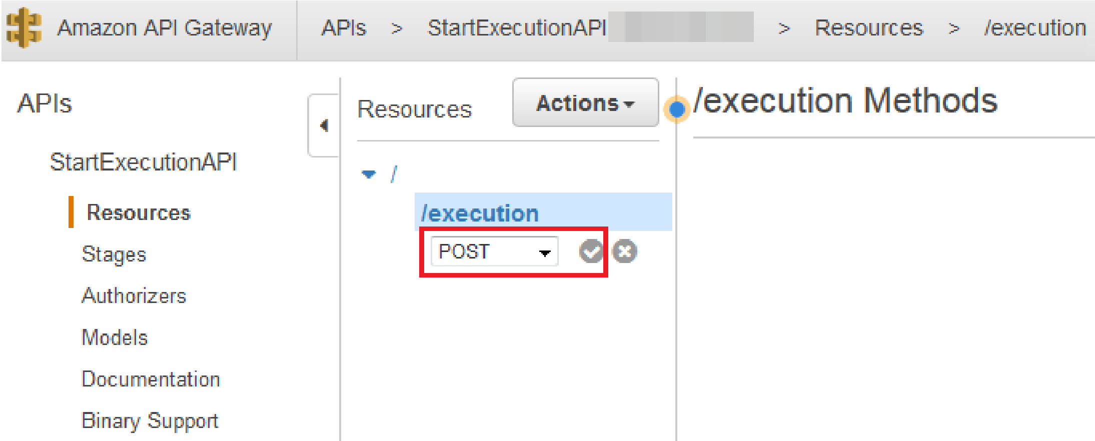
1. Navigate to the Amazon API Gateway console and choose **Get Started**.
2. On the **Create new API** page, choose **New API**.
3. In the **Settings** section, enter `StartExecutionAPI` for the **API name**, and then choose **Create API**.

Create a resource

1. On the **Resources** page of ***`StartExecutionAPI`***, choose **Actions**, **Create Resource**.
2. On the **New Child Resource** page, enter `execution` for **Resource Name**, and then choose **Create Resource**.

Create a POST Method

1. On the **/execution Methods** page, choose **Actions**, **Create Method**.
2. From the list, choose `POST`, and then select the check mark.



Configure the method

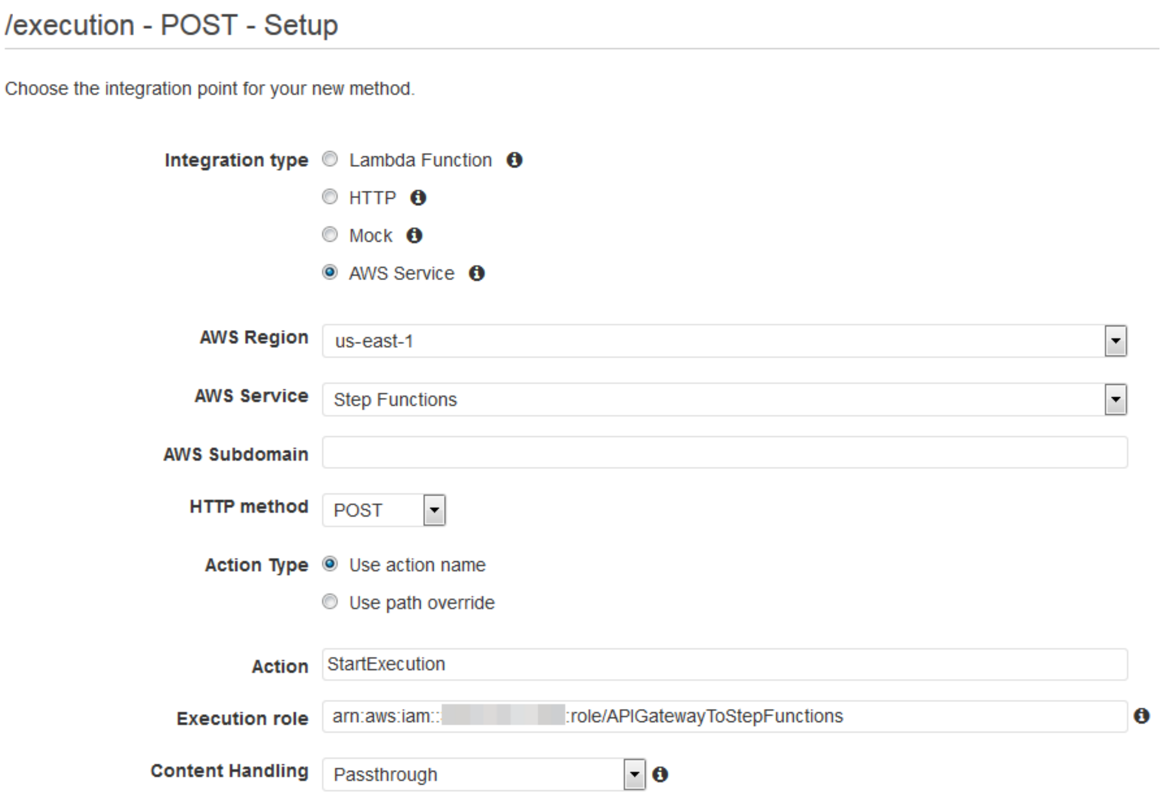
On the **/execution - POST - Setup** page, configure the integration point for your method.

1. For **Integration Type**, choose **AWS Service**.
2. For **AWS Region**, choose a Region from the list.

   Note
   For Regions that currently support Step Functions, see Supported Regions (p. 1).
3. For **AWS Service**, choose **Step Functions** from the list.
4. For **HTTP Method**, choose **POST** from the list.

   Note
   All Step Functions API actions use the HTTP `POST` method.
5. For **Action Type**, choose **Use action name**.
6. For **Action**, enter `StartExecution`.
7. For **Execution Role**, enter the role ARN of the IAM role that you created earlier (p. 48), for example:

```
arn:aws:iam::123456789012:role/APIGatewayToStepFunctions
```



/execution - POST - Setup

Choose the integration point for your new method.

Integration type ○ Lambda Function ○ HTTP ○ Mock ● AWS Service

AWS Region: us-east-1

AWS Service: Step Functions

AWS Subdomain:

HTTP method: POST

Action Type ● Use action name ○ Use path override

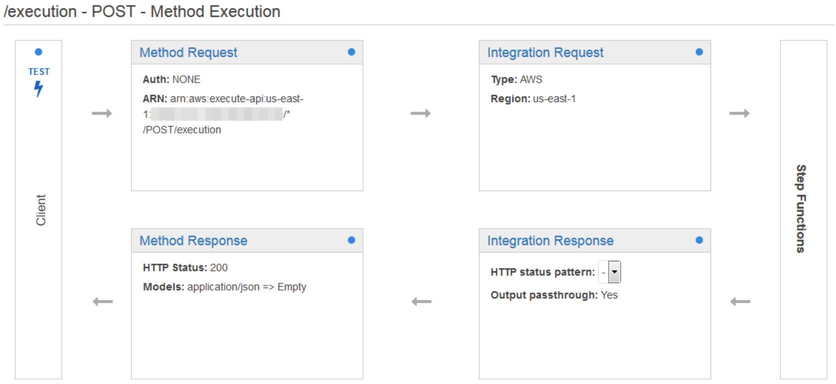
Action: StartExecution

Execution role: arn:aws:iam::[redacted]:role/APIGatewayToStepFunctions

Content Handling: Passthrough

8. Choose **Save**.

   The visual mapping between API Gateway and Step Functions is displayed on the **/execution - POST - Method Execution** page.



/execution - POST - Method Execution

Step 3: Test and Deploy the API Gateway API

Once you have created the API, test and deploy it.

Test the communication between API Gateway and Step Functions

1. On the **/execution - POST - Method Execution** page, choose **Test**.
2. On the **/execution - POST - Method Test** page, copy the following request parameters into the **Request Body** section using the ARN of an existing state machine (or create a new state machine (p. 13)), and then choose **Test**.

```
{
   "input": "{}",
   "name": "MyExecution",
   "stateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld"
}
```

    Note
    For more information, see the `StartExecution` Request Syntax in the *AWS Step Functions API Reference*.
    If you don't want to include the ARN of your state machine in the body of your API Gateway call, you can configure a body-mapping template, for example:

```
{
    "input": "$util.escapeJavaScript($input.json('$'))",
    "stateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:HelloWorld"
}
```

    This approach enables you to have different state machines based on your development stages (for example, `dev`, `test`, and `prod`). To release an update, you need to change only the stage variable, for example:

```
{
    "input": "$util.escapeJavaScript($input.json('$'))",
    "stateMachineArn":
 "$util.escapeJavaScript($stageVariables.get(arn:aws:states:us-
east-1:123456789012:stateMachine:HelloWorld))"
}
```

3. The execution starts and the execution ARN and its epoch date are displayed under **Response Body**.

```
{
   "executionArn": "arn:aws:states:us-
east-1:123456789012:execution:HelloWorld:MyExecution",
   "startDate": 1486768956.878
}
```

    Note
    You can view the execution by choosing your state machine on the AWS Step Functions console.

Deploy your API

1. On the **Resources** page of ***`StartExecutionAPI`***, choose **Actions**, **Deploy API**.
2. In the **Deploy API** dialog box, select **[New Stage]** from the **Deployment stage** list, enter `alpha` for **Stage name**, and then choose **Deploy**.

To test your deployment

1. On the **Stages** page of ***StartExecutionAPI***, expand **alpha**, **/**, **/execution**, **POST**.
2. On the **alpha - POST - /execution** page, note the **Invoke URL**, for example:

```
https://a1b2c3d4e5.execute-api.us-east-1.amazonaws.com/alpha/execution
```

3. From the command line, run the `curl` command using the ARN of your state machine, and then invoke the URL of your deployment, for example:

```
curl -X POST -d '{"input": "{}","name": "MyExecution","stateMachineArn":
 "arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld"}' https://
a1b2c3d4e5.execute-api.us-east-1.amazonaws.com/alpha/execution
```

   The execution ARN and its epoch date are returned, for example:

```
{"executionArn":"arn:aws:states:us-
east-1:123456789012:execution:HelloWorld:MyExecution","startDate":1.486772644911E9}
```

Iterating a Loop Using Lambda

In this tutorial, you implement a design pattern that uses a state machine and an AWS Lambda function to iterate a loop a specific number of times.

Use this design pattern any time you need to keep track of the number of loops in a state machine. This implementation can help you break up large tasks or long-running executions into smaller chunks, or to end an execution after a specific number of events. You can use a similar implementation to periodically end and restart a long-running execution to avoid exceeding service limits for AWS Step Functions, AWS Lambda, or other AWS services.

Before you begin, go through the Creating a Lambda State Machine (p. 19) tutorial to ensure you have created the necessary IAM role, and are familiar with using Lambda and Step Functions together.

Topics

Step 1: Create a Lambda Function to Iterate a Count

By using a Lambda function you can track the number of iterations of a loop in your state machine. The following Lambda function receives input values for `count`, `index`, and `step`. It returns these values with an updated `index` and a Boolean value named `continue`. The Lambda function sets `continue` to `true` if the `index` is less than `count`.

Your state machine then implements a `Choice` state that executes some application logic if `continue` is `true`, or exits if it is `false`.

To create the Lambda function

1. Sign in to the Lambda console, and then choose **Create function**.

2. In the **Create function** section, choose **Author from scratch**.
3. In the **Basic information** section, configure your Lambda function, as follows:

   a. For **Function name**, enter `Iterator`.
   b. For **Runtime**, choose **Node.js 6.10**.
   c. For **Role**, select **Use an existing role**.
   d. For **Existing role**, choose the Lambda role that you created in the Creating a Lambda State Machine (p. 19) tutorial.

      Note
      If the IAM role that you created doesn't appear in the list, the role might still need a few minutes to propagate to Lambda.
   e. Choose **Create function**.

      When your Lambda function is created, make a note of its Amazon Resource Name (ARN) in the upper-right corner of the page. For example:

      ```
      arn:aws:lambda:us-east-1:123456789012:function:Iterator
      ```
4. Copy the following code for the Lambda function into the **Configuration** section of the ***`Iterator`*** page in the Lambda console.

   ```
   exports.iterator = function iterator (event, context, callback) {
     let index = event.iterator.index
     let step = event.iterator.step
     let count = event.iterator.count

     index += step

     callback(null, {
       index,
       step,
       count,
       continue: index < count
     })
   }
   ```

   This code accepts input values for `count`, `index`, and `step`. It increments the `index` by the value of `step` and returns these values, and the Boolean `continue`. The value of `continue` is `true` if `index` is less than `count`.
5. Choose **Save**.

Step 2: Test the Lambda Function

Run your Lambda function with numeric values to see it in operation. You can provide input values for your Lambda function that mimic an iteration, to see what output you get with specific input values.

To test your Lambda function

1. In the **Configure test event** dialog box, choose **Create new test event**, and then enter `TestIterator` for **Event name**.
2. Replace the example data with the following.

   ```
   {
     "Comment": "Test my Iterator function",
     "iterator": {
   ```

```
    "count": 10,
    "index": 5,
    "step": 1
  }
}
```

These values mimic what would come from your state machine during an iteration. The Lambda function will increment the index and return `continue` as `true`. When the index isn't less than the `count`, it returns `continue` as `false`. For this test, the index has already incremented to `5`. The results should increment the `index` to `6` and set `continue` to `true`.

3. Choose **Create**.
4. On the ***`Iterator`*** page in your Lambda console, be sure **TestIterator** is listed, and then choose **Test**.

   The results of the test are displayed at the top of the page. Choose **Details** and review the result.

```
{
  "index": 6,
  "step": 1,
  "count": 10,
  "continue": true
}
```

> **Note**
> If you set `index` to `9` for this test, the `index` increments to `10`, and `continue` is `false`.

Step 3: Create a State Machine

To create the state machine

1. Sign in to the Step Functions console, and then choose **Create a state machine**.

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

2. On the **Create a state machine** page, choose **Author with code snippets**. For **Give a name to your state machine**, enter `IterateCount`.

   > **Note**
   > State machine, execution, and activity names must be 1–80 characters in length, must be unique for your account and AWS Region, and must not contain any of the following:
   >
   > - Whitespace
   > - Wildcard characters (`? *`)
   > - Bracket characters (`< > { } [ ]`)
   > - Special characters (`: ; , \ | ^ ~ $ # % & ` "`)
   > - Control characters (`\\u0000` - `\\u001f` or `\\u007f` - `\\u009f`).
   >
   > Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

   Select **Next**.

3. Create or enter an IAM role:

- To create an IAM role for Step Functions, select **Create an IAM role for me**, and enter a **Name** for your role.
- If you have previously created an IAM role (p. 231) with the correct permissions for your state machine, select **Choose an existing IAM role**. Select a role from the list, or provide an ARN for that role.

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

4. The following code describes a state machine with the following states.

   - `ConfigureCount` – Sets the default values for `count`, `index`, and `step`.

   ```
   "ConfigureCount": {
       "Type": "Pass",
       "Result": {
           "count": 10,
           "index": 0,
           "step": 1
       },
       "ResultPath": "$.iterator",
       "Next": "Iterator"
   },
   ```

   - `Iterator` – References the Lambda function you created earlier, passing in the values configured in `ConfigureCount`.

   ```
   "Iterator": {
       "Type": "Task",
       "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Iterate",
       "ResultPath": "$.iterator",
       "Next": "IsCountReached"
   },
   ```

   - `IsCountReached` – A choice state that either runs your sample work again or goes to `Done`, based on a Boolean value returned from your `Iterator` Lambda function.

   ```
   "IsCountReached": {
       "Type": "Choice",
       "Choices": [
           {
               "Variable": "$.iterator.continue",
               "BooleanEquals": true,
               "Next": "ExampleWork"
           }
       ],
       "Default": "Done"
   },
   ```

   - `ExampleWork` – A stub for the work you want to accomplish in your execution. In this example, it's a `pass` state. In an actual implementation, this would be a `task` state. See Task (p. 91).
   - `Done` – The end state of your execution.

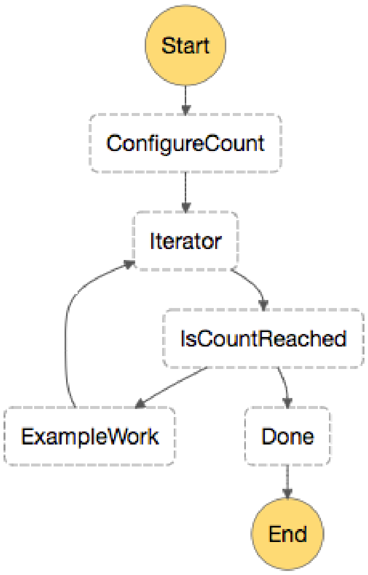
   In the **Code** pane, add the following state machine definition using the Amazon Resource Name (ARN) of the Lambda function that you created earlier (p. 52).

```
{
    "Comment": "Iterator State Machine Example",
    "StartAt": "ConfigureCount",
    "States": {

        "ConfigureCount": {
            "Type": "Pass",
            "Result": {
                "count": 10,
                "index": 0,
                "step": 1
            },
            "ResultPath": "$.iterator",
            "Next": "Iterator"
        },
        "Iterator": {
            "Type": "Task",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Iterate",
            "ResultPath": "$.iterator",
            "Next": "IsCountReached"
        },
        "IsCountReached": {
            "Type": "Choice",
            "Choices": [
                {
                    "Variable": "$.iterator.continue",
                    "BooleanEquals": true,
                    "Next": "ExampleWork"
                }
            ],
            "Default": "Done"
        },
        "ExampleWork": {
            "Comment": "Your application logic, to run a specific number of times",
            "Type": "Pass",
            "Result": {
              "success": true
            },
            "ResultPath": "$.result",
            "Next": "Iterator"
        },
        "Done": {
            "Type": "Pass",
            "End": true

        }
    }
}
```

Be sure to update the ARN in the `Iterator` state above, so that it references the Lambda function you created earlier. For more information about the Amazon States Language, see State Machine Structure (p. 88).

5. Use the graph in the **Visual Workflow** pane to check that your Amazon States Language code describes your state machine correctly.

   This graph shows the logic expressed in the previous state machine code.



If you don't see the graph, choose ↻ in the **Visual Workflow** pane.

6. Choose **Next**.
7. Create or enter an IAM role:

   - To create an IAM role for Step Functions, select **Create an IAM role for me**, and enter a **Name** for your role.
   - If you have previously created an IAM role (p. 231) with the correct permissions for your state machine, select **Choose an existing IAM role**. Select a role from the list, or provide an ARN for that role.

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

Step 4: Start a New Execution

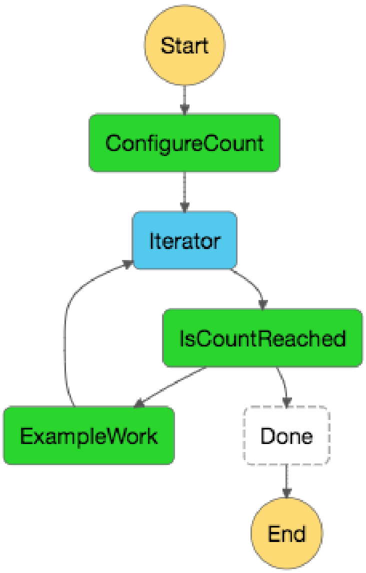
After you create your state machine, you can start an execution.

To start a new execution

1. On the **IterateCount** page, choose **New execution**.
2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

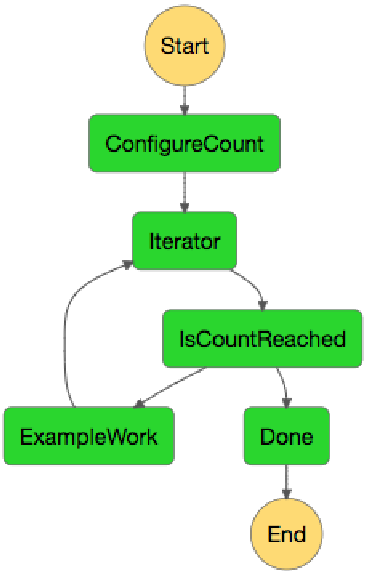
   Note
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. Choose **Start Execution**.

A new execution of your state machine starts, showing your running execution.



The execution increments in steps, tracking the count using your Lambda function. On each iteration, it performs the example work referenced in the `ExampleWork` state in your state machine.

4. (Optional) In the **Execution Details** section, choose the **Info** tab to view the **Execution Status** and the **Started** and **Closed** timestamps.
5. When the count reaches the number specified in the `ConfigureCount` state in your state machine, the execution quits iterating and ends.



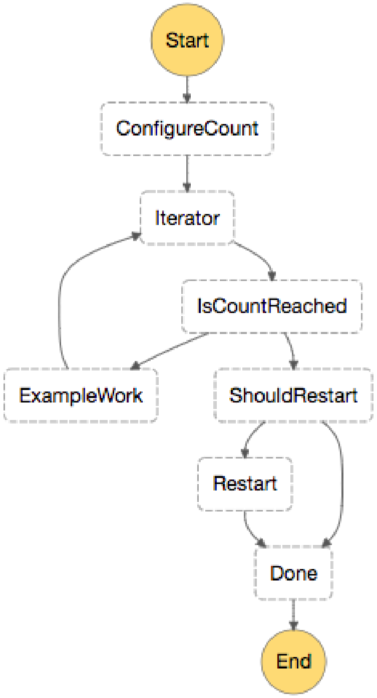
Continue as a New Execution

This tutorial shows you how to create a state machine with a Lambda function that can start a new execution, continuing your ongoing work in that new execution.

AWS Step Functions is designed to run workflows that have a finite duration and number of steps. Executions are limited to a duration of one year, and a maximum of 25,000 events (see Limits (p. 206)).

However, you can create a state machine that uses an AWS Lambda function to start a new execution, before allowing the current execution to terminate. This enables you to have a state machine that can break large jobs into smaller workflows, or to have a state machine that runs indefinitely.

This tutorial builds on the concept of using an external Lambda function to modify your workflow, which was demonstrated in the Iterating a Loop Using Lambda (p. 52) tutorial. You use the same Lambda function (`Iterator`) to iterate a loop for a specific number of times. In addition, you create another Lambda function to start a new execution of your workflow, and to decrement a count each time it starts a new execution. By setting the number of executions in the input, this state machine ends and restarts an execution a specified number of times.



The state machine you'll create implements the following states.

State	Purpose
`ConfigureCount`	A `Pass (p. 90)` state that configures the `count`, `index`, and `step` values that the `Iterator` Lambda function uses to step through iterations of work.
`Iterator`	A `Task (p. 91)` state that references the `Iterator` Lambda function.

State	Purpose
`IsCountReached`	A `Choice (p. 102)` state that uses a Boolean value from the `Iterator` function to decide if the state machine should continue the example work, or move to the `ShouldRestart` choice state.
`ExampleWork`	In this example, `ExampleWork` is a `Pass` state that represents the `Task` state that would perform work in an actual implementation.
`ShouldRestart`	A `Choice (p. 102)` state that uses the `executionCount` value to decide if it should end one execution and start another, or simply end.
`Restart`	A `Task (p. 91)` state that uses a Lambda function to start a new execution of your state machine. Like the `Iterator` function, this function also decrements a count. It passes that value to the input of the new execution.

Prerequisites

Before you begin, go through the Creating a Lambda State Machine (p. 19) tutorial to ensure you have created an initial IAM role, and that you are familiar with using Lambda and Step Functions together.

Topics

Step 1: Create an Iterate Lambda Function to Iterate a Count

> **Note**
> If you have completed the Iterating a Loop Using Lambda (p. 52) tutorial, you can skip this step and use that Lambda function.

This section and the Iterating a Loop Using Lambda (p. 52) tutorial show how you can use a Lambda function to track a count so that you can track the number of iterations of a loop in your state machine.

The following Lambda function receives input values for `count`, `index`, and `step`. It returns these values with an updated `index` and a Boolean named `continue`. The Lambda function sets `continue` to `true` if the `index` is less than `count`.

Your state machine then implements a `Choice` state that executes some application logic if `continue` is `true`, or moves on to `ShouldRestart` if `continue` is `false`.

To create the Iterate Lambda function

1. Open the Lambda console, and then choose **Create function**.
2. In the **Create function** section, choose **Author from scratch**.
3. In the **Author with code snippets** section, configure your Lambda function, as follows:

a. For **Name**, enter `Iterator`.

b. For **Runtime**, choose **Node.js 10.15**.

c. For **Role**, select **Choose an existing role**.

d. For **Existing role**, choose the Lambda role that you created in the Creating a Lambda State Machine (p. 19) tutorial.

> **Note**
> If the IAM role that you created doesn't appear in the list, the role might still need a few minutes to propagate to Lambda.

e. Choose **Create function**.

When your Lambda function is created, make a note of its Amazon Resource Name (ARN) in the upper-right corner of the page, for example:

```
arn:aws:lambda:us-east-1:123456789012:function:Iterator
```

4. Copy the following code for the Lambda function into the **Configuration** section of the ***`Iterator`*** page in the Lambda console.

```
exports.iterator = function iterator (event, context, callback) {
  let index = event.iterator.index
  let step = event.iterator.step
  let count = event.iterator.count

  index += step

  callback(null, {
    index,
    step,
    count,
    continue: index < count
  })
}
```

This code accepts input values for `count`, `index`, and `step`. It increments the `index` by the value of `step` and returns these values, and the Boolean value of `continue`. The value of `continue` is `true` if `index` is less than `count`.

5. Choose **Save**.

Test the Iterate Lambda Function

To see your `Iterate` function working, run it with numeric values. You can provide input values for your Lambda function that mimic an iteration to see what output you get with specific input values.

To test your Lambda function

1. In the **Configure test event** dialog box, choose **Create new test event**, and then type `TestIterator` for **Event name**.
2. Replace the example data with the following.

```
{
  "Comment": "Test my Iterator function",
  "iterator": {
    "count": 10,
    "index": 5,
    "step": 1
  }
}
```

These values mimic what would come from your state machine during an iteration. The Lambda function increments the index and returns `continue` as `true`. When the index is not less than the `count`, it returns `continue` as `false`. For this test, the index has already incremented to `5`. The results should increment the `index` to `6` and set `continue` to `true`.

3. Choose **Create**.
4. On the ***`Iterator`*** page in your Lambda console, be sure **TestIterator** is listed, and then choose **Test**.

   The results of the test are displayed at the top of the page. Choose **Details** and review the result.

```
{
  "index": 6,
  "step": 1,
  "count": 10,
  "continue": true
}
```

> **Note**
> If you set `index` to `9` for this test, the `index` increments to `10`, and `continue` is `false`.

Step 2: Create a Restart Lambda Function to Start a New Step Functions Execution

1. Open the Lambda console, and then choose **Create function**.
2. In the **Author with code snippets** section, configure your Lambda function, as follows:

   a. For **Name**, enter `Restart`.

   b. For **Runtime**, choose **Node.js 10.15**.

   c. For **Role**, select **Choose an existing role**.

   d. Under **Existing role**, choose the role that includes the IAM policy you created previously.

   e. Choose **Create function**.

   When your Lambda function is created, make a note of its Amazon Resource Name (ARN) in the upper-right corner of the page, for example:

```
arn:aws:lambda:us-east-1:123456789012:function:Restart
```

3. Copy the following code for the Lambda function into the **Configuration** section of the ***`Restart`*** page in the Lambda console.

   The following code decrements a count of the number of executions, and starts a new execution of your state machine, including the decremented value.

```
var aws = require('aws-sdk');
var sfn = new aws.StepFunctions();

exports.restart = function(event, context, callback) {

  let StateMachineArn = event.restart.StateMachineArn;
  event.restart.executionCount -= 1;
  event = JSON.stringify(event);
  let params = {
      input: event,
      stateMachineArn: StateMachineArn
  };

  sfn.startExecution(params, function(err, data) {
      if (err) callback(err);
      else callback(null,event);
  });

}
```

4. Choose **Save**.

Step 3: Create a State Machine

Now that you've created your two Lambda functions, create a state machine. In this state machine, the `ShouldRestart` and `Restart` states are how you break your work across multiple executions.

Example ShouldRestart Choice state

This excerpt of your state machine shows the `ShouldRestart` `Choice (p. 102)` state. This state determines whether you should restart the execution.

```
"ShouldRestart": {
"Type": "Choice",
"Choices": [
  {
    "Variable": "$.restart.executionCount",
    "NumericGreaterThan": 1,
    "Next": "Restart"
  }
],
```

The `$.restart.executionCount` value is included in the input of the initial execution. It's decremented by one each time the `Restart` function is called, and then placed into the input for each subsequent execution.

Example Restart Task state

This excerpt of your state machine shows the `Restart` `Task (p. 91)` state. This state uses the Lambda function you created earlier to restart the execution, and to decrement the count to track the remaining number of executions to start.

```
"Restart": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Restart",
  "Next": "Done"
},
```

1. On the Step Functions console, choose **Create a state machine**.
2. Select **Author with code snippets**, and enter `ContinueAsNew` as your state machine name.
3. Paste the following into the `Code` pane.

   ### Example ContinueAsNew state machine

   ```
   {
   ```

```
    "Comment": "Continue-as-new State Machine Example",
    "StartAt": "ConfigureCount",
    "States": {
        "ConfigureCount": {
            "Type": "Pass",
            "Result": {
                "count": 100,
                "index": -1,
                "step": 1
            },
            "ResultPath": "$.iterator",
            "Next": "Iterator"
        },
        "Iterator": {
            "Type": "Task",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Iterator",
            "ResultPath": "$.iterator",
            "Next": "IsCountReached"
        },
        "IsCountReached": {
            "Type": "Choice",
            "Choices": [
                {
                    "Variable": "$.iterator.continue",
                    "BooleanEquals": true,
                    "Next": "ExampleWork"
                }
            ],
            "Default": "ShouldRestart"
        },
        "ExampleWork": {
            "Comment": "Your application logic, to run a specific number of times",
            "Type": "Pass",
            "Result": {
              "success": true
            },
            "ResultPath": "$.result",
            "Next": "Iterator"
        },
        "ShouldRestart": {
          "Type": "Choice",
          "Choices": [
            {
              "Variable": "$.restart.executionCount",
              "NumericGreaterThan": 0,
              "Next": "Restart"
            }
          ],
          "Default": "Done"
        },
        "Restart": {
          "Type": "Task",
          "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Restart",
          "Next": "Done"
        },
        "Done": {
            "Type": "Pass",
            "End": true
        }
    }
}
```

4. Update the `Resource` string in the `Restart` and `Iterator` states to reference the respective Lambda functions you created earlier.

5. Choose **Next**.

6. Create or enter an IAM role:

   - To create an IAM role for Step Functions, select **Create an IAM role for me**, and enter a **Name** for your role.
   - If you have previously created an IAM role (p. 231) with the correct permissions for your state machine, select **Choose an existing IAM role**. Select a role from the list, or provide an ARN for that role.

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

7. Choose **Create state machine**.

   Note
   Save the Amazon Resource Name (ARN) of this state machine.

Step 4: Update the IAM Policy

To ensure your Lambda function has permissions to start a new Step Functions execution, attach an inline policy to the IAM role you use for your `Restart` Lambda function. For more information, see Embedding Inline Policies in the *IAM User Guide*.

```
{
 "Version": "2012-10-17",
 "Statement": [
     {
         "Sid": "VisualEditor0",
         "Effect": "Allow",
         "Action": [
             "states:StartExecution"
         ],
         "Resource": "*"
     }
 ]
}
```

Note
You can update the `"Resource": "*"` line in the previous example to reference the ARN of your `ContinueAsNew` state machine. This restricts the policy so that it can only start an execution of that specific state machine.

Step 5: Run an Execution

To start an execution, provide input that includes the ARN of the state machine and an `executionCount` for how many times it should start a new execution.

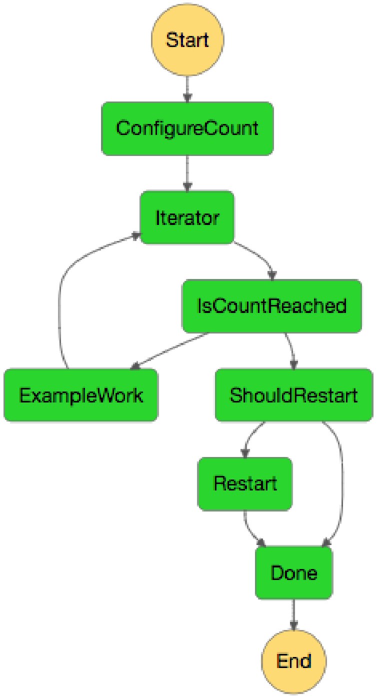
1. On the **ContinueAsNew** page, choose **New execution**.
2. In the **Input** section, on the **New execution** page, enter `Test1` for the execution name. Then enter the following in the **Input**.

   ```
   {
     "restart": {
       "StateMachineArn": "arn:aws:states:us-
   east-1:123456789012:stateMachine:ContinueAsNew",
   ```

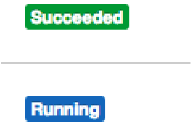
```
    "executionCount": 4
  }
}
```

3. Update the `StateMachineArn` field with the ARN for your `ContinueAsNew` state machine.
4. Choose **Start Execution**.

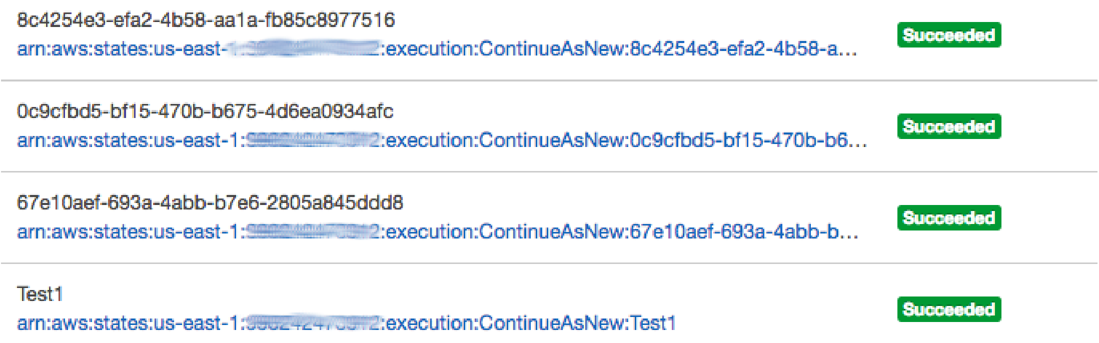
The **Visual Workflow** graph displays the first of the four executions. Before it completes, it will pass through the `Restart` state and start a new execution.



With this execution complete, you can look at the next execution that's running. Select the **ContinueAsNew** link at the top to see the list of executions. You should see both the recently closed execution, and an ongoing execution that the `Restart` Lambda function kicked off.



When all the executions are complete, you should see four successful executions in the list. The first execution that was started displays the name you chose, and subsequent executions have a generated name.



8c4254e3-efa2-4b58-aa1a-fb85c8977516
arn:aws:states:us-east-[redacted]:execution:ContinueAsNew:8c4254e3-efa2-4b58-a… Succeeded

0c9cfbd5-bf15-470b-b675-4d6ea0934afc
arn:aws:states:us-east-1:[redacted]:execution:ContinueAsNew:0c9cfbd5-bf15-470b-b6… Succeeded

67e10aef-693a-4abb-b7e6-2805a845ddd8
arn:aws:states:us-east-1:[redacted]:execution:ContinueAsNew:67e10aef-693a-4abb-b… Succeeded

Test1
arn:aws:states:us-east-1:[redacted]:execution:ContinueAsNew:Test1 Succeeded

Using Code Snippets to Create a State to Send an Amazon SNS message

Step Functions integrates with certain AWS services, such as Amazon Simple Notification Service (SNS). In this tutorial, you generate a code snippet that sends a text message using Amazon SNS. You pass parameters directly to Amazon SNS from your state machine definition.

For more information about how Step Functions integrates with other AWS services directly from the Amazon States Language, see:

- Service Integrations (p. 138)
- Code Snippets (p. 145)
- Pass Parameters to a Service API (p. 144)

Topics

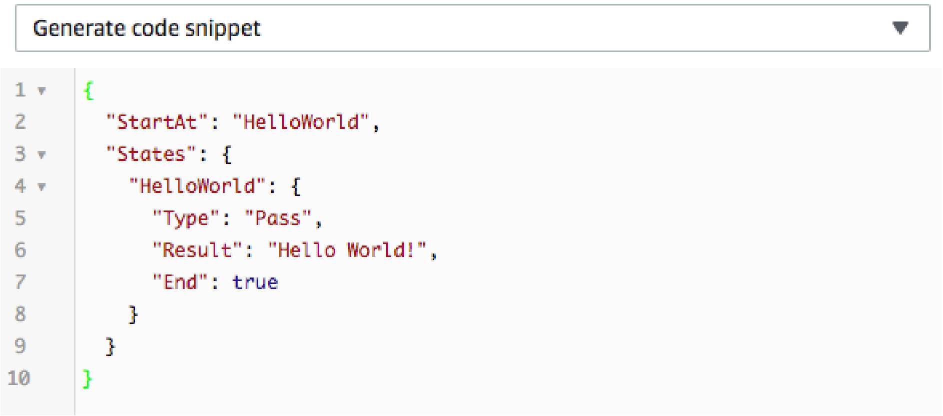
- Step 1: Generate a Code Snippet (p. 67)
- Step 2: Update Your State Machine Definition (p. 69)
- Step 3: Start an Execution (p. 71)

Step 1: Generate a Code Snippet

To generate a code snippet, you must start by editing a state machine definition.

1. Sign in to the AWS Management Console and open the Step Functions console at https://console.aws.amazon.com/states/home?#/statemachines/create.
2. Choose **Author with code snippets**, and then enter a name for your state machine.

   The default `HelloWorld` state machine is displayed in the **State machine definition**.



```
{
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Pass",
      "Result": "Hello World!",
      "End": true
    }
  }
}
```

3. For **Generate Code Snippet**, choose **Amazon SNS: Publish a message**.

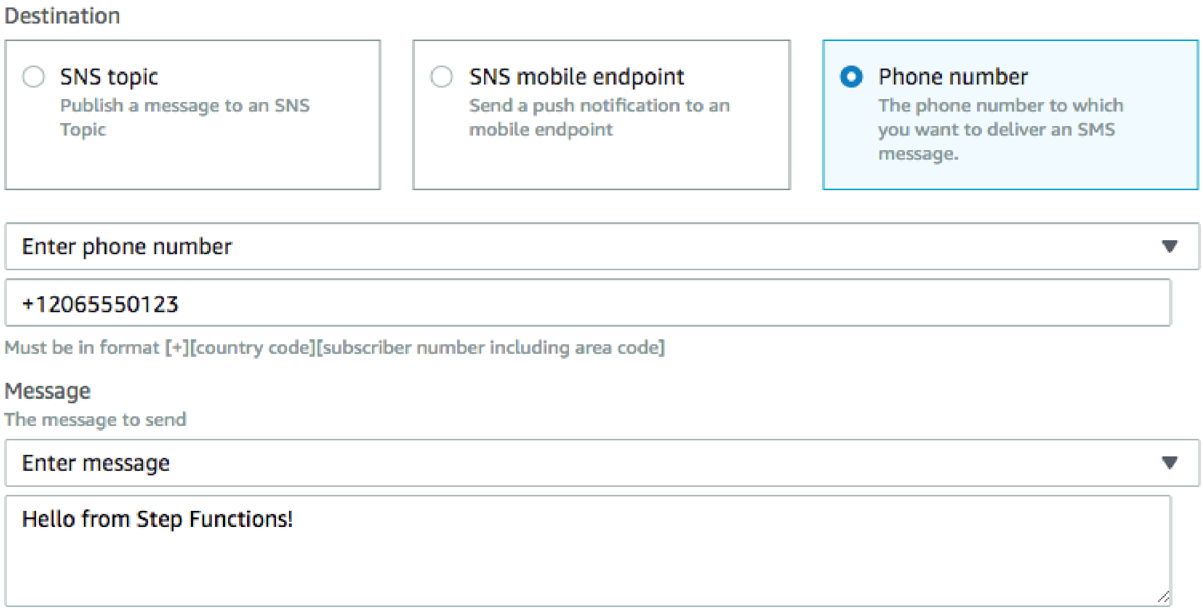
   The **Generate SNS Publish task state** window is displayed.
4. On the **Generate SNS Publish task state** page, under **Destination**, choose **Enter phone number** and then enter your cell phone number.

   Use the format `[+][country code][subscriber number including area code]`. For example: `+12065550123`.
5. Under **Message**, choose **Enter message**, and then enter some text to send as an SMS message.

   > **Note**
   > You can also choose **Specify message at runtime with state input**. This option enables you to use a reference path to select a message from the input of your state machine execution. For more information, see:
   >
   > - Input and Output Processing in Step Functions (p. 115)
   > - Reference Paths (p. 117)
   > - Pass State Input as Parameters Using Paths (p. 144)

As you configure options on the **Generate SNS Publish task state** page, the **Preview** section updates with the Amazon States Language code for a task state with the necessary options.

For example, choose these options.



Destination

○ SNS topic
Publish a message to an SNS Topic

○ SNS mobile endpoint
Send a push notification to an mobile endpoint

● Phone number
The phone number to which you want to deliver an SMS message.

Enter phone number ▼

+12065550123

Must be in format [+][country code][subscriber number including area code]

Message
The message to send

Enter message ▼

Hello from Step Functions!

With these options selected, this is the generated code snippet that's displayed in the **Preview** area.

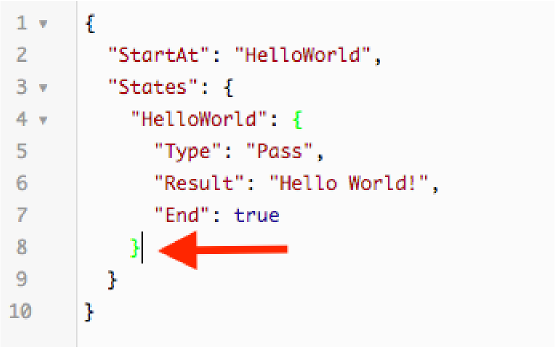
```
"Amazon SNS: Publish a message": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sns:publish",
  "Parameters": {
    "Message": "Hello from Step Functions!",
    "PhoneNumber": "+12065550123"
  },
  "Next": "NEXT_STATE"
}
```

> **Note**
> Under the **Task state options** section, you can also configure `Retry`, `Catch`, and `TimeoutSeconds` options. See Error Handling (p. 132).

Step 2: Update Your State Machine Definition

Now that you have configured your Amazon SNS options, paste the generated code snippet into your state machine definition and update the existing Amazon States Language code.

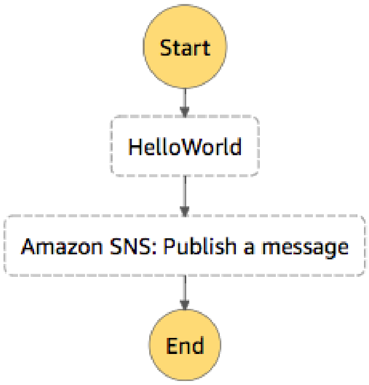
1. After you have reviewed the code in the **Preview** section, choose **Copy to clipboard**.
2. Place your cursor after the closing bracket of the `HelloWorld` state in your state machine definition.



```
{
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Pass",
      "Result": "Hello World!",
      "End": true
    }
  }
}
```

Enter a comma, press **Enter** to start a new line, and then paste your code snippet into your state machine definition.

3. Change the last line of the `Amazon SNS: Publish a message` state from *`"Next": "NEXT_STATE"`* to *`"End": true`*.
4. Change the last line of the `HelloWorld` state from *`"End": true`* to *`"Next": "Amazon SNS: Publish a message"`*.
5. Choose ↻ in the **Visual Workflow** pane. Check the visual workflow to ensure your new state is included.



6. (Optional) Indent the JSON to make your code easier to read. Your state machine definition should look like this.

```
{
   "StartAt":"HelloWorld",
   "States":{
      "HelloWorld":{
         "Type":"Pass",
         "Result":"Hello World!",
         "Next":"Amazon SNS: Publish a message"
      },
      "Amazon SNS: Publish a message":{
         "Type":"Task",
         "Resource":"arn:aws:states:::sns:publish",
         "Parameters":{
            "Message":"Hello from Step Functions!",
            "PhoneNumber":"+12065550123"
         },
         "End":true
      }
   }
}
```

7. Choose **Next**.
8. Create or enter an IAM role:

   - To create an IAM role for Step Functions, select **Create an IAM role for me**, and enter a **Name** for your role.
   - If you have previously created an IAM role (p. 231) with the correct permissions for your state machine, select **Choose an existing IAM role**. Select a role from the list, or provide an ARN for that role.

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

Step 3: Start an Execution

After it's created, the page from your new state machine is displayed.

1. Review the details of your state machine, including the Amazon Resource Name (ARN), the related IAM ARN, and the state machine definition.
2. On the **Executions** tab, choose **Start execution**.
3. (Optional) Enter a name for your execution.

   Note
   If we had chosen **Specify message at runtime with state input** when creating our Amazon SNS code snippet, we would include a message in the **Input - optional**. For now you can use the default state input.

   Choose **Start execution**.

If you configured a valid cell phone number in your code snippet, you should have received a text message from Amazon SNS that was triggered directly by your state machine execution.

Deploy an Example Human Approval Project

This tutorial shows you how to deploy a human approval project that allows an AWS Step Functions execution to pause during a task, and wait for a user to respond to an email. The workflow progresses to the next state once the user has approved the task to proceed.

Deploying the AWS CloudFormation stack included in this tutorial will create all necessary resources, including:

- Amazon API Gateway resources
- An AWS Lambda functions
- An AWS Step Functions state machine
- An Amazon Simple Notification Service email topic
- Related AWS Identity and Access Management roles and permissions

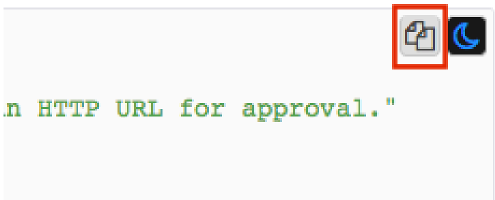
> **Note**
> You will need to provide a valid email address that you have access to when you create the AWS CloudFormation stack.

For more information, see Working with CloudFormation Templates and the `AWS::StepFunctions::StateMachine` resource in the *AWS CloudFormation User Guide*.

Topics

Step 1: Create an AWS CloudFormation Template

1. Copy the example code from the AWS CloudFormation Template Source Code (p. 74) section.

   ```
   n HTTP URL for approval."
   ```

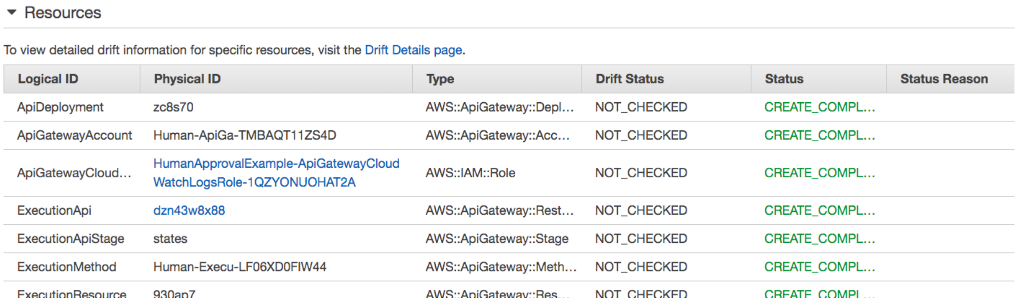
2. Paste the source of the AWS CloudFormation template into a file on your local machine.

   For this example the file is called `human-approval.yaml`.

Step 2: Create a Stack

1. Log into the AWS CloudFormation console.
2. Choose **Create Stack**.
3. Under **Choose a template**, select **Upload a template to Amazon S3** and then **Choose File**.
4. Browse to the `human-approval.yaml` file you created earlier that includes the template source code (p. 74).
5. Choose **Open** and then **Next**.
6. Under **Specify Details** enter a **Stack name**.
7. Under **Parameters** enter a valid **Email** address and choose **Next**.
8. On the **Options** page, scroll down and choose **Next**.
9. On the **Review** page, choose **I acknowledge that AWS CloudFormation might create IAM resources** and then choose **Create**.

   AWS CloudFormation begins to create your stack and displays the **CREATE_IN_PROGRESS** status. When the process is complete, AWS CloudFormation displays the **CREATE_COMPLETE** status.
10. (Optional) To display the resources in your stack, select the stack and choose the **Resources** tab.



▾ Resources

To view detailed drift information for specific resources, visit the Drift Details page.

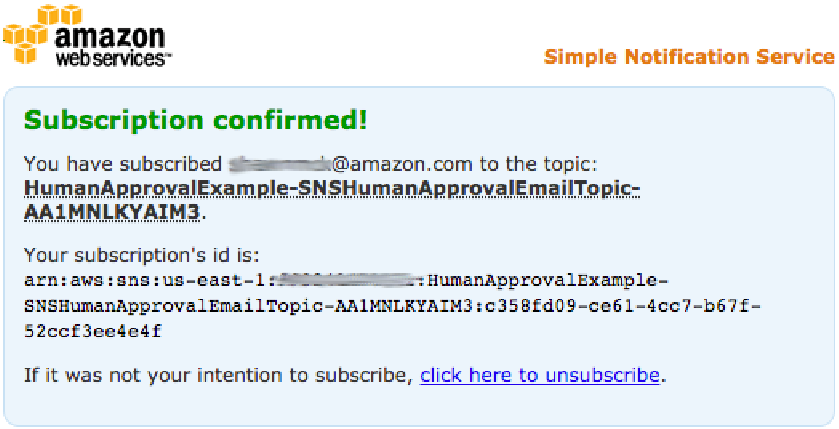
Logical ID	Physical ID	Type	Drift Status	Status	Status Reason
ApiDeployment	zc8s70	AWS::ApiGateway::Depl…	NOT_CHECKED	CREATE_COMPL…	
ApiGatewayAccount	Human-ApiGa-TMBAQT11ZS4D	AWS::ApiGateway::Acc…	NOT_CHECKED	CREATE_COMPL…	
ApiGatewayCloud…	HumanApprovalExample-ApiGatewayCloudWatchLogsRole-1QZYONUOHAT2A	AWS::IAM::Role	NOT_CHECKED	CREATE_COMPL…	
ExecutionApi	dzn43w8x88	AWS::ApiGateway::Rest…	NOT_CHECKED	CREATE_COMPL…	
ExecutionApiStage	states	AWS::ApiGateway::Stage	NOT_CHECKED	CREATE_COMPL…	
ExecutionMethod	Human-Execu-LF06XD0FIW44	AWS::ApiGateway::Meth…	NOT_CHECKED	CREATE_COMPL…	
ExecutionResource	930ap7	AWS::ApiGateway::Res…	NOT_CHECKED	CREATE_COMPL…	

Step 3: Approve the Amazon Simple Notification Service Subscription

Once the Amazon SNS topic is created, you will receive an email requesting that you confirm subscription.

1. Open the email account you provided when you created the AWS CloudFormation stack.
2. Open the message **AWS Notification - Subscription Confirmation** from **no-reply@sns.amazonaws.com**

   The email will list the Amazon Resource Name for the Amazon SNS topic, and a confirmation link.
3. Choose the **confirm subscription** link.



Subscription confirmed!

You have subscribed [redacted]@amazon.com to the topic: HumanApprovalExample-SNSHumanApprovalEmailTopic-AA1MNLKYAIM3.

Your subscription's id is:
arn:aws:sns:us-east-1:[redacted]:HumanApprovalExample-SNSHumanApprovalEmailTopic-AA1MNLKYAIM3:c358fd09-ce61-4cc7-b67f-52ccf3ee4e4f

If it was not your intention to subscribe, click here to unsubscribe.

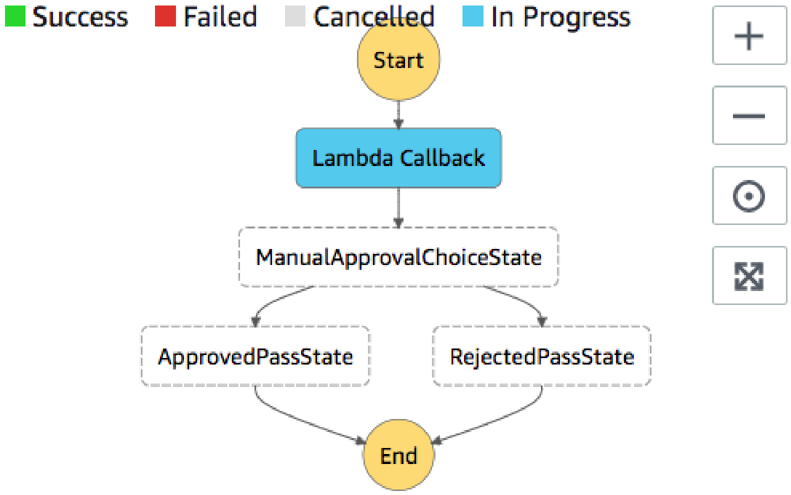
Step 4: Run an Execution

1. Log into the Step Functions console.
2. On the **State machines** page, choose **HumanApprovalLambdaStateMachine**.
3. Choose **Start execution**.
4. Enter a name for your execution, such as `ApprovalTest`.
5. (Optional) enters some input for the execution.

```
{
    "Comment": "Testing the human approval tutorial."
}
```

6. Choose **Start execution**.

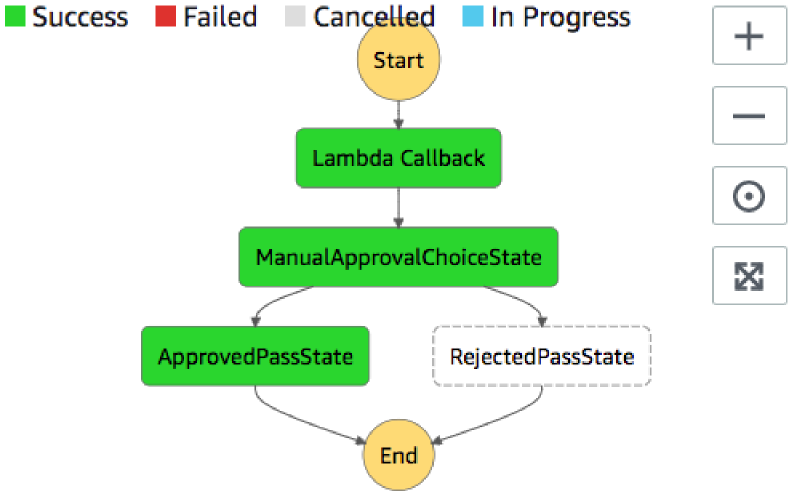
   The **ApprovalTest** execution starts, and pauses at the **Lambda Callback** task.



7. In the email account you used for the Amazon SNS topic earlier, open the message with the subject **Required approval from AWS Step Functions**.

   The message includes separate URLs for **Approve** and **Reject**.
8. Choose the **Approve** URL.

   The workflow continues based on your choice.



AWS CloudFormation Template Source Code

Use this AWS CloudFormation template to deploy an example of a human approval process workflow.

```
AWSTemplateFormatVersion: "2010-09-09"
```

```
Description: "AWS Step Functions Human based task example. It sends an email with an HTTP
 URL for approval."
Parameters:
  Email:
    Type: String
    AllowedPattern: "^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$"
    ConstraintDescription: Must be a valid email address.
Resources:
  # Begin API Gateway Resources
  ExecutionApi:
    Type: "AWS::ApiGateway::RestApi"
    Properties:
      Name: "Human approval endpoint"
      Description: "HTTP Endpoint backed by API Gateway and Lambda"
      FailOnWarnings: true

  ExecutionResource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref ExecutionApi
      ParentId: !GetAtt "ExecutionApi.RootResourceId"
      PathPart: execution

  ExecutionMethod:
    Type: "AWS::ApiGateway::Method"
    Properties:
      AuthorizationType: NONE
      HttpMethod: GET
      Integration:
        Type: AWS
        IntegrationHttpMethod: POST
        Uri: !Sub "arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/
${LambdaApprovalFunction.Arn}/invocations"
        IntegrationResponses:
          - StatusCode: 302
            ResponseParameters:
              method.response.header.Location: "integration.response.body.headers.Location"
        RequestTemplates:
          application/json: |
            {
              "body" : $input.json('$'),
              "headers": {
                #foreach($header in $input.params().header.keySet())
                "$header": "$util.escapeJavaScript($input.params().header.get($header))"
 #if($foreach.hasNext),#end

                #end
              },
              "method": "$context.httpMethod",
              "params": {
                #foreach($param in $input.params().path.keySet())
                "$param": "$util.escapeJavaScript($input.params().path.get($param))"
 #if($foreach.hasNext),#end

                #end
              },
              "query": {
                #foreach($queryParam in $input.params().querystring.keySet())
                "$queryParam":
 "$util.escapeJavaScript($input.params().querystring.get($queryParam))"
 #if($foreach.hasNext),#end

                #end
              }
            }
      ResourceId: !Ref ExecutionResource
      RestApiId: !Ref ExecutionApi
      MethodResponses:
        - StatusCode: 302
          ResponseParameters:
            method.response.header.Location: true

  ApiGatewayAccount:
    Type: 'AWS::ApiGateway::Account'
    Properties:
      CloudWatchRoleArn: !GetAtt "ApiGatewayCloudWatchLogsRole.Arn"

  ApiGatewayCloudWatchLogsRole:
    Type: 'AWS::IAM::Role'
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - apigateway.amazonaws.com
            Action:
              - 'sts:AssumeRole'
      Policies:
        - PolicyName: ApiGatewayLogsPolicy
          PolicyDocument:
            Version: 2012-10-17
            Statement:
              - Effect: Allow
                Action:
                  - "logs:*"
                Resource: !Sub "arn:${AWS::Partition}:logs:*:*:*"

  ExecutionApiStage:
    DependsOn:
      - ApiGatewayAccount
    Type: 'AWS::ApiGateway::Stage'
    Properties:
      DeploymentId: !Ref ApiDeployment
      MethodSettings:
        - DataTraceEnabled: true
          HttpMethod: '*'
          LoggingLevel: INFO
          ResourcePath: /*
      RestApiId: !Ref ExecutionApi
      StageName: states

  ApiDeployment:
    Type: "AWS::ApiGateway::Deployment"
    DependsOn:
      - ExecutionMethod
    Properties:
      RestApiId: !Ref ExecutionApi
      StageName: DummyStage
  # End API Gateway Resources

  # Begin
  # Lambda that will be invoked by API Gateway
  LambdaApprovalFunction:
    Type: 'AWS::Lambda::Function'
    Properties:
      Code:
        ZipFile:
          Fn::Sub: |
            const AWS = require('aws-sdk');
```

```
           var redirectToStepFunctions = function(lambdaArn, statemachineName,
executionName, callback) {
             const lambdaArnTokens = lambdaArn.split(":");
             const partition = lambdaArnTokens[1];
             const region = lambdaArnTokens[3];
             const accountId = lambdaArnTokens[4];

             console.log("partition=" + partition);
             console.log("region=" + region);
             console.log("accountId=" + accountId);

             const executionArn = "arn:" + partition + ":states:" + region + ":" +
accountId + ":execution:" + statemachineName + ":" + executionName;
             console.log("executionArn=" + executionArn);

             const url = "https://console.aws.amazon.com/states/home?region=" + region +
"#/executions/details/" + executionArn;
             callback(null, {
                 statusCode: 302,
                 headers: {
                   Location: url
                 }
             });
           };

           exports.handler = (event, context, callback) => {
             console.log('Event= ' + JSON.stringify(event));
             const action = event.query.action;
             const taskToken = event.query.taskToken;
             const statemachineName = event.query.sm;
             const executionName = event.query.ex;

             const stepfunctions = new AWS.StepFunctions();

             var message = "";

             if (action === "approve") {
               message = { "Status": "Approved! Task approved by ${Email}" };
             } else if (action === "reject") {
               message = { "Status": "Rejected! Task rejected by ${Email}" };
             } else {
               console.error("Unrecognized action. Expected: approve, reject.");
               callback({"Status": "Failed to process the request. Unrecognized
Action."});
             }

             stepfunctions.sendTaskSuccess({
               output: JSON.stringify(message),
               taskToken: event.query.taskToken
             })
             .promise()
             .then(function(data) {
               redirectToStepFunctions(context.invokedFunctionArn, statemachineName,
executionName, callback);
             }).catch(function(err) {
               console.error(err, err.stack);
               callback(err);
             });
           }
     Description: Lambda function that callback to AWS Step Functions
     FunctionName: LambdaApprovalFunction
     Handler: index.handler
     Role: !GetAtt "LambdaApiGatewayIAMRole.Arn"
     Runtime: nodejs8.10

 LambdaApiGatewayInvoke:
    Type: "AWS::Lambda::Permission"
    Properties:
      Action: "lambda:InvokeFunction"
      FunctionName: !GetAtt "LambdaApprovalFunction.Arn"
      Principal: "apigateway.amazonaws.com"
      SourceArn: !Sub "arn:aws:execute-api:${AWS::Region}:${AWS::AccountId}:
${ExecutionApi}/*"

  LambdaApiGatewayIAMRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Action:
              - "sts:AssumeRole"
            Effect: "Allow"
            Principal:
              Service:
                - "lambda.amazonaws.com"
      Policies:
        - PolicyName: CloudWatchLogsPolicy
          PolicyDocument:
            Statement:
              - Effect: Allow
                Action:
                  - "logs:*"
                Resource: !Sub "arn:${AWS::Partition}:logs:*:*:*"
        - PolicyName: StepFunctionsPolicy
          PolicyDocument:
            Statement:
              - Effect: Allow
                Action:
                  - "states:SendTaskFailure"
                  - "states:SendTaskSuccess"
                Resource: "*"
  # End Lambda that will be invoked by API Gateway

  # Begin state machine that publishes to Lambda and sends an email with the link for
 approval
  HumanApprovalLambdaStateMachine:
    Type: AWS::StepFunctions::StateMachine
    Properties:
      RoleArn: !GetAtt LambdaStateMachineExecutionRole.Arn
      DefinitionString:
        Fn::Sub: |
          {
              "StartAt": "Lambda Callback",
              "TimeoutSeconds": 3600,
              "States": {
                  "Lambda Callback": {
                      "Type": "Task",
                      "Resource": "arn:
${AWS::Partition}:states:::lambda:invoke.waitForTaskToken",
                      "Parameters": {
                        "FunctionName": "${LambdaHumanApprovalSendEmailFunction.Arn}",
                        "Payload": {
                          "ExecutionContext.$": "$$",
                          "APIGatewayEndpoint": "https://${ExecutionApi}.execute-api.
${AWS::Region}.amazonaws.com/states"
                        }
                      },
                      "Next": "ManualApprovalChoiceState"
                  },
                  "ManualApprovalChoiceState": {
                    "Type": "Choice",
                    "Choices": [
                      {
                        "Variable": "$.Status",
                        "StringEquals": "Approved! Task approved by ${Email}",
                        "Next": "ApprovedPassState"
                      },
                      {
                        "Variable": "$.Status",
                        "StringEquals": "Rejected! Task rejected by ${Email}",
                        "Next": "RejectedPassState"
                      }
                    ]
                  },
                  "ApprovedPassState": {
                    "Type": "Pass",
                    "End": true
                  },
                  "RejectedPassState": {
                    "Type": "Pass",
                    "End": true
                  }
              }
          }

  SNSHumanApprovalEmailTopic:
    Type: AWS::SNS::Topic
    Properties:
      Subscription:
        -
           Endpoint: !Sub ${Email}
           Protocol: email

  LambdaHumanApprovalSendEmailFunction:
    Type: "AWS::Lambda::Function"
    Properties:
      Handler: "index.lambda_handler"
      Role: !GetAtt LambdaSendEmailExecutionRole.Arn
      Runtime: "nodejs8.10"
      Timeout: "25"
      Code:
        ZipFile:
          Fn::Sub: |
            console.log('Loading function');
            const AWS = require('aws-sdk');
            exports.lambda_handler = (event, context, callback) => {
                console.log('event= ' + JSON.stringify(event));
                console.log('context= ' + JSON.stringify(context));

                const executionContext = event.ExecutionContext;
                console.log('executionContext= ' + executionContext);

                const executionName = executionContext.Execution.Name;
                console.log('executionName= ' + executionName);

                const statemachineName = executionContext.StateMachine.Name;
                console.log('statemachineName= ' + statemachineName);

                const taskToken = executionContext.Task.Token;
                console.log('taskToken= ' + taskToken);

                const apigwEndpint = event.APIGatewayEndpoint;
                console.log('apigwEndpint = ' + apigwEndpint)

                const approveEndpoint = apigwEndpint + "/execution?action=approve&ex=" +
 executionName + "&sm=" + statemachineName + "&taskToken=" + encodeURIComponent(taskToken);
                console.log('approveEndpoint= ' + approveEndpoint);
                const rejectEndpoint = apigwEndpint + "/execution?action=reject&ex=" +
 executionName + "&sm=" + statemachineName + "&taskToken=" + encodeURIComponent(taskToken);
                console.log('rejectEndpoint= ' + rejectEndpoint);

                const emailSnsTopic = "${SNSHumanApprovalEmailTopic}";
                console.log('emailSnsTopic= ' + emailSnsTopic);

                var emailMessage = 'Welcome! \n\n';
                emailMessage += 'This is an email requiring an approval for a step
 functions execution. \n\n'
                emailMessage += 'Please check the following information and click "Approve"
 link if you want to approve. \n\n'
                emailMessage += 'Execution Name -> ' + executionName + '\n\n'
                emailMessage += 'Approve ' + approveEndpoint + '\n\n'
                emailMessage += 'Reject ' + rejectEndpoint + '\n\n'
                emailMessage += 'Thanks for using Step functions!'

                const sns = new AWS.SNS();
                var params = {
                  Message: emailMessage,
                  Subject: "Required approval from AWS Step Functions",
                  TopicArn: emailSnsTopic
                };

                sns.publish(params)
                  .promise()
                  .then(function(data) {
                    console.log("MessageID is " + data.MessageId);
                    callback(null);
                  }).catch(
                    function(err) {
                    console.error(err, err.stack);
                    callback(err);
                  });
            }

  LambdaStateMachineExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Principal:
              Service: states.amazonaws.com
            Action: "sts:AssumeRole"
      Policies:
        - PolicyName: InvokeCallbackLambda
          PolicyDocument:
            Statement:
              - Effect: Allow
                Action:
                  - "lambda:InvokeFunction"
                Resource:
                  - !Sub "${LambdaHumanApprovalSendEmailFunction.Arn}"

  LambdaSendEmailExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
```

```
            Action: "sts:AssumeRole"
      Policies:
        - PolicyName: CloudWatchLogsPolicy
          PolicyDocument:
            Statement:
              - Effect: Allow
                Action:
                  - "logs:CreateLogGroup"
                  - "logs:CreateLogStream"
                  - "logs:PutLogEvents"
                Resource: !Sub "arn:${AWS::Partition}:logs:*:*:*"
        - PolicyName: SNSSendEmailPolicy
          PolicyDocument:
            Statement:
              - Effect: Allow
                Action:
                  - "SNS:Publish"
                Resource:
                  - !Sub "${SNSHumanApprovalEmailTopic}"

# End state machine that publishes to Lambda and sends an email with the link for approval
Outputs:
  ApiGatewayInvokeURL:
    Value: !Sub "https://${ExecutionApi}.execute-api.${AWS::Region}.amazonaws.com/states"
  StateMachineHumanApprovalArn:
    Value: !Ref HumanApprovalLambdaStateMachine
```

Use a Map State to Call Lambda Multiple Times

In this tutorial, you will learn how to use a `Map` state to call a AWS Lambda function multiple times, based on the state machine input.

The Creating a Lambda State Machine (p. 19) tutorial walks you though creating a state machine that calls a Lambda function. If you have completed that tutorial, skip to Step 4 (p. 83) and use the IAM role and Lambda function that you previously created.

Topics

Step 1: Create an IAM Role for Lambda

Both AWS Lambda and AWS Step Functions can execute code and access AWS resources (for example, data stored in Amazon S3 buckets). To maintain security, you must grant Lambda and Step Functions access to these resources.

Lambda requires you to assign an AWS Identity and Access Management (IAM) role when you create a Lambda function, in the same way Step Functions requires you to assign an IAM role when you create a state machine.

You use the IAM console to create a service-linked role.

To create a role (console)

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane of the IAM console, choose **Roles**. Then choose **Create role**.
3. Choose the **AWS Service** role type, and then choose **Lambda**.
4. Choose the **Lambda** use case. Use cases are defined by the service to include the trust policy required by the service. Then choose **Next: Permissions**.
5. Choose one or more permissions policies to attach to the role (for example, `AWSLambdaBasicExecutionRole`). See AWS Lambda Permissions Model.

   Select the box next to the policy that assigns the permissions that you want the role to have, and then choose **Next: Review**.
6. Enter a **Role name**.
7. (Optional) For **Role description**, edit the description for the new service-linked role.
8. Review the role, and then choose **Create role**.

Step 2: Create a Lambda Function

Your Lambda function receives input (a name) and returns a greeting that includes the input value.

> **Important**
> Ensure that your Lambda function is under the same AWS account and AWS Region as your state machine.

1. Open the Lambda console and choose **Create a function**.
2. In the **Blueprints** section, choose **Author from scratch**.
3. In the **Basic information** section, configure your Lambda function:

   a. For **Function name**, enter `HelloFunction`.

   b. For **Runtime**, choose **Node.js 10.x**.

   c. For **Role**, select **Choose an existing role**.

   d. For **Existing role**, select the Lambda role that you created earlier (p. 81).

      > **Note**
      > If the IAM role that you created doesn't appear in the list, the role might still need a few minutes to propagate to Lambda.

   e. Choose **Create function**.

      When your Lambda function is created, note its Amazon Resource Name (ARN) in the upper-right corner of the page. For example:

      ```
      arn:aws:lambda:us-east-1:123456789012:function:HelloFunction
      ```
4. Copy the following code for the Lambda function into the **Function code** section of the ***HelloFunction*** page.

   ```
   exports.handler = (event, context, callback) => {
       callback(null, "Hello, " + event.who + "!");
   };
   ```

   This code assembles a greeting using the `who` field of the input data, which is provided by the `event` object passed into your function. You add input data for this function later, when you start a new execution (p. 84). The `callback` method returns the assembled greeting from your function.

5. Choose **Save**.

Step 3: Test the Lambda Function

Test your Lambda function to see it in operation.

1. For **Select a test event**, choose **Configure test event**. For **Event name**, enter `HelloFunction`.
2. Replace the example data with the following.

```
{
    "who": "AWS Step Functions"
}
```

   The `"who"` entry corresponds to the `event.who` field in your Lambda function, completing the greeting. You will use the same input data when running the function as a Step Functions task.
3. Choose **Create**.
4. On the ***`HelloFunction`*** page, **Test** your Lambda function using the new data.

   The results of the test are displayed at the top of the page. Expand **Details** to see the output.

Step 4: Create a State Machine

Use the Step Functions console to create a state machine with a `Map` state. Add a `Task`state with a reference to your Lambda. The Lambda function is invoked for each iteration of the `Map` state, based on the state machine input.

1. Open the Step Functions console and choose **Create a state machine**.
2. On the **Define state machine** page, choose **Author with code snippets**. Enter a **Name for your state machine**, for example, *`MapLambda`*.

   Note
   State machine, execution, and activity names must be 1–80 characters in length, must be unique for your account and AWS Region, and must not contain any of the following:

   - Whitespace
   - Wildcard characters (`? *`)
   - Bracket characters (`< > { } [ ]`)
   - Special characters (`: ; , \ | ^ ~ $ # % & ` "`)
   - Control characters (`\\u0000` - `\\u001f` or `\\u007f` - `\\u009f`).

   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. In the **State machine definition** pane, add the following state machine definition using the ARN of the Lambda function that you created earlier (p. 82), for example:

```
{
  "StartAt": "ExampleMapState",
  "States": {
    "ExampleMapState": {
      "Type": "Map",
      "Iterator": {
         "StartAt": "CallLambda",
         "States": {
           "CallLambda": {
             "Type": "Task",
             "Resource": "arn:aws:lambda:us-
east-1:123456789012:function:HelloFunction",
             "End": true
           }
         }
      },
      "End": true
    }
  }
}
```

This is a description of your state machine using the Amazon States Language. It defines a `Map` state named `ExampleMapState` that includes a `Task` state (`CallLambda`) that calls your Lambda function. For more information, see State Machine Structure (p. 88).

> **Note**
> You can also set up a `Retry` for `Task` states. As a best practice, ensure production code can handle Lambda service exceptions (`Lambda.ServiceException` and `Lambda.SdkclientException`). For more information, see:
>
> - Handle Lambda Service Exceptions (p. 204).
> - Retrying after an Error (p. 133).

Choose **Next**.

4. Create or enter an IAM role:

   - To create an IAM role for Step Functions, select **Create an IAM role for me**, and enter a **Name** for your role.
   - If you have previously created an IAM role (p. 231) with the correct permissions for your state machine, select **Choose an existing IAM role**. Select a role from the list, or provide an ARN for that role.

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

5. Select **Next**.

Step 5: Start a New Execution

After you create your state machine, you start an execution.

1. On the ***`LambdaStateMachine`*** page, choose **Start execution**.

   The **New execution** page is displayed.

2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   > **Note**
   > Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch.

To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

3. In the execution input area, replace the example data with the following.

   ```
   [
     {
       "who": "bob"
     },
     {
       "who": "meg"
     },
     {
       "who": "joe"
     }
   ]
   ```

   Your `Map` state will iterate and run the `CallLambda` task state for each of these items in the input. `"who"` is the key name that your Lambda function uses to get the name of the person to greet.

4. Choose **Start Execution**.

   A new execution of your state machine starts, and a new page showing your running execution is displayed.

5. To view the results of your execution, expand the **Output** section under **Execution details**.

   ```
   [
     "Hello, bob!",
     "Hello, meg!",
     "Hello, joe!"
   ]
   ```

   The output for each of the Lambda function executions are combined into the output for the state machine.

For more information on using `Map` states, see the following.

- ??? (p. 109)
- Map State Example (p. 110)
- Map State Input and Output Processing (p. 112)
- ItemsPath (p. 119)

How Step Functions Works

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

Topics

States

Individual states can make decisions based on their input, perform actions, and pass output to other states. In Step Functions you express your workflows in the Amazon States Language, and the Step Functions console provides a graphical representation of that state machine to help visualize your application logic.

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

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

- Do some work in your state machine (a Task (p. 91) state).
- Make a choice between branches of execution (a Choice (p. 102) state)
- Stop an execution with a failure or success (a Fail (p. 106) or Succeed (p. 106) state)
- Simply pass its input to its output or inject some fixed data (a Pass (p. 90) state)
- Provide a delay for a certain amount of time or until a specified time/date (a Wait (p. 105) state)
- Begin parallel branches of execution (a Parallel (p. 107) state)
- Dynamically iterate steps using a Map (p. 109) state

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

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

States share many common features:

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

> **Note**
> A `Choice` state may have more than one `Next`, but only one within each Choice Rule. A `Choice` state cannot use `End`.

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

After you create a state machine and have executed it, you can access information about each state, its input and output, when it was active and for how long, by viewing the **Execution Details** page on the Step Functions console.

Topics

Amazon States Language

The Amazon States Language is a JSON-based, structured language used to define your state machine, a collection of states (p. 86), that can do work (`Task` states), determine which states to transition to next (`Choice` states), stop an execution with an error (`Fail` states), and so on.

For more information, see the Amazon States Language Specification and Statelint, a tool that validates Amazon States Language code.

To create a state machine on the Step Functions console using Amazon States Language, see Getting Started (p. 13).

Example Amazon States Language Specification

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

    "FirstMatchState": {
      "Type" : "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:OnFirstMatch",
      "Next": "NextState"
    },

    "SecondMatchState": {
      "Type" : "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:OnSecondMatch",
      "Next": "NextState"
    },

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

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

Topics

- State Machine Structure (p. 88)
- Common State Fields (p. 89)

State Machine Structure

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

`Comment` (Optional)

A human-readable description of the state machine.

`StartAt` (Required)

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

`TimeoutSeconds` (Optional)

The maximum number of seconds an execution of the state machine can run. If it runs longer than the specified time, the execution fails with a `States.Timeout` Error Name (p. 132).

`Version` (Optional)

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

`States` (Required)

An object containing a comma-delimited set of states.

The `States` field contains States (p. 86).

```
{
    "State1" : {
    },

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

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

The following is an example.

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

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

The following rules apply to states within a state machine:

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

Common State Fields

`Type` (Required)

The state's type.

`Next`

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

`End`

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

`Comment` (Optional)

Holds a human-readable description of the state.

`InputPath` (Optional)

A path (p. 115) that selects a portion of the state's input to be passed to the state's task for processing. If omitted, it has the value `$` which designates the entire input. For more information, see Input and Output Processing (p. 115)).

`OutputPath` (Optional)

A path (p. 115) that selects a portion of the state's input to be passed to the state's output. If omitted, it has the value `$` which designates the entire input. For more information, see Input and Output Processing (p. 115).

Pass

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

In addition to the common state fields (p. 89), `Pass` states allow the following fields.

`Result` (Optional)

Treated as the output of a virtual task to be passed to the next state, and filtered as specified by the `ResultPath` field (if present).

`ResultPath` (Optional)

Specifies where (in the input) to place the "output" of the virtual task specified in `Result`. The input is further filtered as specified by the `OutputPath` field (if present) before being used as the state's output. For more information, see Input and Output Processing (p. 115).

`Parameters` (Optional)

Create a collection of key-value pairs that will be passed as input. Values can be static, or selected from the input with a path. For more information, see InputPath and Parameters (p. 117).

Pass State Example

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

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

Suppose the input to this state is this:

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

Then the output would be this.

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

Task

A `Task` state (`"Type": "Task"`) represents a single unit of work performed by a state machine.

All work in your state machine is done by *tasks*. A task performs work by using an activity or an AWS Lambda function, or by passing parameters to the API actions of other services.

AWS Step Functions can invoke Lambda functions directly from a task state. A Lambda function is a cloud-native task that runs on AWS Lambda. You can write Lambda functions in a variety of programming languages, using the AWS Management Console or by uploading code to Lambda. For more information see ??? (p. 19).

> **Note**
> Step Functions can coordinate some AWS services directly from a task state. For more information see Service Integrations (p. 138).

An activity consists of program code that waits for an operator to perform an action or to provide input. You can host activities on Amazon EC2, on Amazon ECS, or even on mobile devices. Activities poll Step Functions using the `GetActivityTask` and `SendTaskSuccess`, `SendTaskFailure`, and `SendTaskHeartbeat` API actions.

The Amazon States Language represents tasks by setting a state's type to `Task` and by providing the task with the Amazon Resource Name (ARN) of the activity or Lambda function.

In addition to the common state fields (p. 89), `Task` states have the following fields.

`Resource` (Required)

A URI, especially an Amazon Resource Name (ARN) that uniquely identifies the specific task to execute.

`Parameters` (Optional)

Used to pass information to the API actions of connected resources. The parameters can use a mix of static JSON and JsonPath. For more information, see Pass Parameters to a Service API (p. 144).

`ResultPath` (Optional)

Specifies where (in the input) to place the results of executing the task that's specified in `Resource`. The input is then filtered as specified by the `OutputPath` field (if present) before being used as the state's output. For more information, see path (p. 115).

`Retry` (Optional)

An array of objects, called Retriers, that define a retry policy if the state encounters runtime errors. For more information, see Examples Using Retry and Using Catch (p. 135).

`Catch` (Optional)

An array of objects, called Catchers, that define a fallback state. This state is executed if the state encounters runtime errors and its retry policy is exhausted or isn't defined. For more information, see Fallback States (p. 134).

`TimeoutSeconds` (Optional)

If the task runs longer than the specified seconds, this state fails with a `States.Timeout` error name. Must be a positive, non-zero integer. If not provided, the default value is `99999999`. The count begins after the task has been started, for example, when `ActivityStarted` or `LambdaFunctionStarted` are logged in the **Execution event history**.

`HeartbeatSeconds` (Optional)

If more time than the specified seconds elapses between heartbeats from the task, this state fails with a `States.Timeout` error name. Must be a positive, non-zero integer less than the number of seconds specified in the `TimeoutSeconds` field. If not provided, the default value is `99999999`. For Activities, the count begins when `GetActivityTask` receives a token and `ActivityStarted` is logged in the **Execution event history**.

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

Task State Example

Here is an example.

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

In this example, `ActivityState` will schedule the `HelloWorld` activity for execution in the `us-east-1` AWS Region on the caller's AWS account. When `HelloWorld` completes, the next state (here called `NextState`) will be run.

If this task fails to complete within 300 seconds, or doesn't send heartbeat notifications in intervals of 60 seconds, the task is marked as `failed`. It's a good practice to set a timeout value and a heartbeat interval for long-running activities.

Specifying Resource ARNs in Tasks

The `Resource` field's Amazon Resource Name (ARN) is specified using the following pattern.

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

In this pattern:

- `partition` is the AWS Step Functions partition to use, most commonly `aws`.
- `service` indicates the AWS service used to execute the task, and is:

  - `states` for an activity (p. 93).
  - `lambda` for a Lambda function (p. 93).
- `region` is the AWS Region in which the Step Functions activity or state machine type or Lambda function has been created.
- `account` is your AWS account ID.
- `task_type` is the type of task to run. It is one of the following values:
  - `activity` – An activity (p. 93).
  - `function` – A Lambda function (p. 93).
  - *`servicename`* – The name of a supported connected service (see Supported AWS Service Integrations for Step Functions (p. 146)).
- `name` is the registered resource name (activity name, Lambda function name, or service API action).

> **Note**
> Step Functions doesn't support referencing ARNs across partitions (for example, "aws-cn" can't invoke tasks in the "aws" partition, and vice versa).

Task Types

The following task types are supported:

- activity (p. 93)
- Lambda functions (p. 93)
- A supported AWS service (p. 138)

The following sections provide more detail about each task type.

Activity

Activities represent workers (processes or threads), implemented and hosted by you, that perform a specific task.

Activity `resource` ARNs use the following syntax.

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

For more information about these fields, see Specifying Resource ARNs in Tasks (p. 92).

> **Note**
> You must create activities with Step Functions (using a CreateActivity, API action, or the Step Functions console) before their first use.

For more information about creating an activity and implementing workers, see Activities (p. 94).

Lambda Functions

Lambda tasks execute a function using AWS Lambda. To specify a Lambda function, use the Amazon Resource Name (ARN) of the Lambda function in the `Resource` field.

Lambda function `Resource` ARNs use the following syntax.

```
arn:partition:lambda:region:account:function:function_name
```

For more information about these fields, see Specifying Resource ARNs in Tasks (p. 92).

For example:

```
"LambdaState": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloWorld",
  "Next": "NextState"
}
```

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

A Supported AWS Service

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

Connected service `Resource` ARNs use the following syntax.

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

> **Note**
> To create a synchronous connection to a connected resource, append `.sync` to the *`APIname`* entry in the ARN. For more information, see Service Integrations (p. 138).

For example:

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

Activities

Activities are an AWS Step Functions feature that enables you to have a task in your state machine where the work is performed by a *worker* that can be hosted on Amazon Elastic Compute Cloud (Amazon EC2), Amazon Elastic Container Service (Amazon ECS), mobile devices—basically anywhere.

Topics

Overview

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

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

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

APIs Related to Activity Tasks

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

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

- `CreateActivity`
- `GetActivityTask`
- `ListActivities`
- `SendTaskFailure`
- `SendTaskHeartbeat`
- `SendTaskSuccess`

> **Note**
> Polling for activity tasks with `GetActivityTask` can cause latency in some implementations. See Avoid Latency When Polling for Activity Tasks (p. 205).

Waiting for an Activity Task to Complete

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

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

1. Create an activity by using the console, or by using `CreateActivity`. Make a note of the activity ARN.
2. Reference that ARN in an activity task state in your state machine definition and set `TimeoutSeconds`.
3. Implement an activity worker that polls for work by using `GetActivityTask`, referencing that activity ARN.
4. Use `SendTaskHeartbeat` periodically within the time you set in `HeartbeatSeconds (p. 91)` in your state machine task definition to keep the task from timing out.
5. Start an execution of your state machine.

6. Start your activity worker process.

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

Next Steps

For a detailed look at creating a state machine that uses an activity worker, see Creating an Activity State Machine (p. 31).

Example Activity Worker in Ruby

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

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

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

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

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

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

# The start method takes as argument the block that is the actual logic of your custom
activity.
activity.start do |input|
  { result: :SUCCESS, echo: input['value'] }
```

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

Item	Description
`require_relative`	Relative path to the following example activity worker code.
`region`	AWS Region of your activity.

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

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

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

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

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

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

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

```
end

class Activity
  def initialize(options = {})
    @states = Aws::States::Client.new(
      credentials: Validate.required(options[:credentials]),
      region: Validate.required(options[:region]),
      http_read_timeout: Validate.positive(options[:http_read_timeout] || 60)
    )
    @activity_arn = Validate.required(options[:activity_arn])
    @heartbeat_delay = Validate.positive(options[:heartbeat_delay] || 60)
    @queue_max = Validate.positive(options[:queue_max] || 5)
    @pollers_count = Validate.positive(options[:pollers_count] || 1)
    @workers_count = Validate.positive(options[:workers_count] || 1)
    @max_retry = Validate.positive(options[:workers_count] || 3)
    @logger = Logger.new(STDOUT)
  end

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

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

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

  private

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

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

  def start_heartbeat_worker(activities)
```

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

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

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

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

  def shutdown_worker(worker)
    worker.kill
  end

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

  def wait_activities_drained
    wait_condition { @sink.empty? }
  end

  def wait_activities_completed
    wait_condition { @activities.empty? }
  end

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

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

  class Worker
    def initialize
      @logger = Logger.new(STDOUT)
      @running = false
    end
    def run
      raise 'Method run hasn\'t been implemented'
    end

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

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

    def stop
      @running = false
    end

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

    def wait
      @thread.join
    end
  end

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

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

  class ActivityWorker < Worker
```

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

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

    def send_task_success(activity_task, result)
      StepFunctions.with_retries(max_retry: @max_retry) do
        begin
          @states.send_task_success(
            task_token: activity_task.task_token,
            output: JSON.dump(result)
          )
        rescue => e
          @logger.error('Failed to send task success')
          @logger.error(e)
        end
      end
    end

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

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

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

    def send_heartbeat(token)
      StepFunctions.with_retries(max_retry: @max_retry) do
```

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

Choice

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

In addition to the common state fields (p. 89), `Choice` states introduce the following additional fields.

`Choices` (Required)

An array of Choice Rules (p. 103) that determines which state the state machine transitions to next.

`Default` (Optional, Recommended)

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

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

Choice State Example

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

Note
You must specify the `$.type` field. If the state input doesn't contain the `$.type` field, the execution fails and an error is displayed in the execution history.

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

"Public": {
  "Type" : "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Foo",
  "Next": "NextState"
},

"ValueIsZero": {
  "Type" : "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Zero",
  "Next": "NextState"
},

"ValueInTwenties": {
  "Type" : "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Bar",
  "Next": "NextState"
},

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

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

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

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

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

Choice Rules

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

- A **comparison** – Two fields that specify an input variable to compare, the type of comparison, and the value to compare the variable to.
- A **`Next` field** – The value of this field must match a state name in the state machine.

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

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

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

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

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

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

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

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

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

The following comparison operators are supported:

- `And`
- `BooleanEquals`
- `Not`
- `NumericEquals`
- `NumericGreaterThan`
- `NumericGreaterThanEquals`
- `NumericLessThan`
- `NumericLessThanEquals`
- `Or`
- `StringEquals`
- `StringGreaterThan`
- `StringGreaterThanEquals`
- `StringLessThan`
- `StringLessThanEquals`
- `TimestampEquals`
- `TimestampGreaterThan`
- `TimestampGreaterThanEquals`
- `TimestampLessThan`
- `TimestampLessThanEquals`

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

> **Note**
> For interoperability, don't assume that numeric comparisons work with values outside the magnitude or precision that the IEEE 754-2008 `binary64` data type represents. In particular, integers outside of the range `[-2`^{53}`+1, 2`^{53}`-1]` might fail to compare in the expected way.
> Timestamps (for example, `2016-08-18T17:33:00Z`) must conform to RFC3339 profile ISO 8601, with further restrictions:
>
> - An uppercase `T` must separate the date and time portions.
> - An uppercase `Z` must denote that a numeric time zone offset isn't present.
>
> To understand the behavior of string comparisons, see the Java `compareTo` documentation.
> The values of the `And` and `Or` operators must be non-empty arrays of Choice Rules that must not themselves contain `Next` fields. Likewise, the value of a `Not` operator must be a single Choice Rule that must not contain `Next` fields.
> You can create complex, nested Choice Rules using `And`, `Not`, and `Or`. However, the `Next` field can appear only in a top-level Choice Rule.

Wait

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

In addition to the common state fields (p. 89), `Wait` states have one of the following fields.

`Seconds`

A time, in seconds, to wait before beginning the state specified in the `Next` field.

`Timestamp`

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

Timestamps must conform to the RFC3339 profile of ISO 8601, with the further restrictions that an uppercase `T` must separate the date and time portions, and an uppercase `Z` must denote that a numeric time zone offset is not present, for example, `2016-08-18T17:33:00Z`.

`SecondsPath`

A time, in seconds, to wait before beginning the state specified in the `Next` field, specified using a path (p. 115) from the state's input data.

`TimestampPath`

An absolute time to wait until beginning the state specified in the `Next` field, specified using a path (p. 115) from the state's input data.

> **Note**
> You must specify exactly one of `Seconds`, `Timestamp`, `SecondsPath`, or `TimestampPath`.

Wait State Examples

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

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

In the next example, the `Wait` state waits until an absolute time: March 14th, 2016, at 1:59 PM UTC.

```
"wait_until" : {
  "Type": "Wait",
  "Timestamp": "2016-03-14T01:59:00Z",
  "Next": "NextState"
}
```

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

```
{
  "expirydate": "2016-03-14T01:59:00Z"
}
```

You can select the value of "expirydate" from the input using a reference path (p. 115) to select it from the input data.

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

Succeed

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

Because `Succeed` states are terminal states, they have no `Next` field, and don't need an `End` field, for example:

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

Fail

A `Fail` state (`"Type": "Fail"`) stops the execution of the state machine and marks it as a failure.

The `Fail` state only allows the use of `Type` and `Comment` fields from the set of common state fields (p. 89). In addition, the `Fail` state allows the following fields.

`Cause` (Optional)

Provides a custom failure string that can be used for operational or diagnostic purposes.

`Error` (Optional)

Provides an error name that can be used for error handling (`Retry`/`Catch`), operational, or diagnostic purposes.

Because `Fail` states always exit the state machine, they have no `Next` field and don't require an `End` field.

For example:

```
"FailState": {
  "Type": "Fail",
  "Cause": "Invalid response.",
  "Error": "ErrorA"
}
```

Parallel

The `Parallel` state (`"Type": "Parallel"`) can be used to create parallel branches of execution in your state machine.

In addition to the common state fields (p. 89), `Parallel` states include these additional fields.

`Branches` (Required)

> An array of objects that specify state machines to execute in parallel. Each such state machine object must have fields named `States` and `StartAt`, whose meanings are exactly like those in the top level of a state machine.

`ResultPath` (Optional)

> Specifies where (in the input) to place the output of the branches. The input is then filtered as specified by the `OutputPath` field (if present) before being used as the state's output. For more information, see Input and Output Processing (p. 115).

`Retry` (Optional)

> An array of objects, called Retriers, that define a retry policy in case the state encounters runtime errors. For more information, see Examples Using Retry and Using Catch (p. 135).

`Catch` (Optional)

> An array of objects, called Catchers, that define a fallback state that is executed if the state encounters runtime errors and its retry policy is exhausted or isn't defined. For more information, see Fallback States (p. 134).

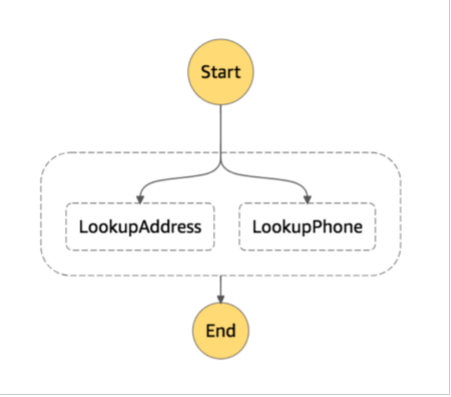
A `Parallel` state causes AWS Step Functions to execute each branch, starting with the state named in that branch's `StartAt` field, as concurrently as possible, and wait until all branches terminate (reach a terminal state) before processing the `Parallel` state's `Next` field.

Here is an example.

Parallel State Example

```
{
  "Comment": "Parallel Example.",
  "StartAt": "LookupCustomerInfo",
  "States": {
    "LookupCustomerInfo": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
         "StartAt": "LookupAddress",
         "States": {
           "LookupAddress": {
             "Type": "Task",
             "Resource":
               "arn:aws:lambda:us-east-1:123456789012:function:AddressFinder",
             "End": true
           }
         }
       },
       {
         "StartAt": "LookupPhone",
         "States": {
           "LookupPhone": {
             "Type": "Task",
             "Resource":
               "arn:aws:lambda:us-east-1:123456789012:function:PhoneFinder",
             "End": true
           }
         }
       }
      ]
    }
  }
}
```

In this example, the `LookupAddress` and `LookupPhone` branches are executed in parallel. Here is how the visual workflow looks in the Step Functions console.



Each branch must be self-contained. A state in one branch of a `Parallel` state must not have a `Next` field that targets a field outside of that branch, nor can any other state outside the branch transition into that branch.

Parallel State Input and Output Processing

A `Parallel` state provides each branch with a copy of its own input data (subject to modification by the `InputPath` field). It generates output that is an array with one element for each branch, containing the output from that branch. There is no requirement that all elements be of the same type. The output array can be inserted into the input data (and the whole sent as the `Parallel` state's output) by using a `ResultPath` field in the usual way (see Input and Output Processing (p. 115)).

```
{
  "Comment": "Parallel Example.",
  "StartAt": "FunWithMath",
  "States": {
    "FunWithMath": {
    "Type": "Parallel",
    "End": true,
    "Branches": [
      {
        "StartAt": "Add",
        "States": {
          "Add": {
            "Type": "Task",
            "Resource": "arn:aws:swf:us-east-1:123456789012:task:Add",
            "End": true
          }
        }
      },
      {
        "StartAt": "Subtract",
        "States": {
          "Subtract": {
            "Type": "Task",
            "Resource": "arn:aws:swf:us-east-1:123456789012:task:Subtract",
            "End": true
          }
        }
      }
    ]
   }
  }
}
```

If the `FunWithMath` state was given the array `[3, 2]` as input, then both the `Add` and `Subtract` states receive that array as input. The output of `Add` would be `5`, that of `Subtract` would be `1`, and the output of the `Parallel` state would be an array.

```
[ 5, 1 ]
```

Error Handling

If any branch fails, because of an unhandled error or by transitioning to a `Fail` state, the entire `Parallel` state is considered to have failed and all its branches are stopped. If the error is not handled by the `Parallel` state itself, Step Functions stops the execution with an error.

> **Note**
> When a parallel state fails, invoked Lambda functions continue to run and activity workers processing a task token are not stopped.
>
> - To stop long-running activities, use heartbeats to detect if its branch has been stopped by Step Functions, and stop workers that are processing tasks. Calling `SendTaskHeartbeat`, `SendTaskSuccess`, or `SendTaskFailure` will throw an error if the state has failed. See Heartbeat Errors.
> - Running Lambda functions cannot be stopped. If you have implemented a fallback, use a `Wait` state so that cleanup work happens after the Lambda function has finished.

Map

The `Map` state (`"Type": "Map"`) can be used to run a set of steps for each element of an input array. While the `Parallel (p. 107)` state executes multiple branches of steps using the same input, a `Map` state will execute the same steps for multiple entries of an array in the state input.

For an introduction to using a `Map` state, see the Map State Tutorial (p. 81).

In addition to the common state fields (p. 89), `Map` states include these additional fields.

`Iterator` (Required)

The “`Iterator`” field’s value is an object that defines a state machine which will process each element of the array.

`ItemsPath` (Optional)

The “`ItemsPath`” field’s value is a reference path identifying where in the effective input the array field is found. For more information, see ItemsPath (p. 119).

States within an `"Iterator"` field can only transition to each other, and no state outside the “`ItemsPath`” field can transition to a state within it.

If any iteration fails, entire Map state fails, and all iterations are terminated.

`MaxConcurrency` (Optional)

The “`MaxConcurrency`”field’s value is an integer that provides an upper bound on how many invocations of the Iterator may run in parallel. For instance, a `"MaxConcurrency"` value of 10 will limit your `Map` state to 10 concurrent iterations running at one time.

> **Note**
> The “`MaxConcurrency`” value is an upper bound limit, and not a guarantee that it will run that many concurrent iterations.

The default value is `"0"`, which places no limit on parallelism and iterations are invoked as concurrently as possible.

A `"MaxConcurrency"` value of `"1"` invokes the `"Iterator"` once for each array element in the order of their appearance in the input, and will not start a new iteration until the previous has completed.

`ResultPath` (Optional)

Specifies where (in the input) to place the output of the branches. The input is then filtered as specified by the `OutputPath` field (if present) before being used as the state's output. For more information, see Input and Output Processing (p. 115).

`Retry` (Optional)

An array of objects, called Retriers, that define a retry policy in case the state encounters runtime errors. For more information, see Examples Using Retry and Using Catch (p. 135).

`Catch` (Optional)

An array of objects, called Catchers, that define a fallback state that is executed if the state encounters runtime errors and its retry policy is exhausted or isn't defined. For more information, see Fallback States (p. 134).

Map State Example

Consider the following input data for a `Map` state.

```
{
  "ship-date": "2016-03-14T01:59:00Z",
  "detail": {
    "delivery-partner": "UQS",
    "shipped": [
      { "prod": "R31", "dest-code": 9511, "quantity": 1344 },
      { "prod": "S39", "dest-code": 9511, "quantity": 40 },
      { "prod": "R31", "dest-code": 9833, "quantity": 12 },
      { "prod": "R40", "dest-code": 9860, "quantity": 887 },
      { "prod": "R40", "dest-code": 9511, "quantity": 1220 }
    ]
  }
}
```

Given the previous input, the `Map` state in the following example will invoke a AWS Lambda function (`ship-val`) once for each item of the array in the `"shipped"` field.

```
"Validate-All": {
  "Type": "Map",
  "InputPath": "$.detail",
  "ItemsPath": "$.shipped",
  "MaxConcurrency": 0,
  "Iterator": {
    "StartAt": "Validate",
    "States": {
      "Validate": {
        "Type": "Task",
 "Resource": "arn:aws:lambda:us-east-1:123456789012:function:ship-val",
        "End": true
      }
    }
  },
  "ResultPath": "$.detail.shipped",
  "End": true
}
```

Each iteration of the `Map` state will send an item in the array (selected with the `"ItemsPath"` (p. 119) field) as input to the Lambda function. For instance, the input to one invocation of Lambda would be the following.

```
{
  "prod": "R31",
  "dest-code": 9511,
  "quantity": 1344
}
```

When complete, the output of the `Map` state is a JSON array where each item is the output of an iteration (in this case, the output of the `ship-val` Lambda function).

Map State Example With Parameters

Suppose that the `ship-val` Lambda function in the previous example also needs information about the shipment's courier as well as the items in the array for each iteration. You can include information from the input, along with information specific to the current iteration of the map state. Note the `"Parameters"` field in the following example.

```
"Validate-All": {
  "Type": "Map",
  "InputPath": "$.detail",
  "ItemsPath": "$.shipped",
  "MaxConcurrency": 0,
  "ResultPath": "$.detail.shipped",
  "Parameters": {
    "parcel.$": "$$.Map.Item.Value",
    "courier.$": "$.delivery-partner"
  },
  "Iterator": {
    "StartAt": "Validate",
    "States": {
      "Validate": {
        "Type": "Task",
 "Resource": "arn:aws:lambda:us-east-1:123456789012:function:ship-val",
        "End": true
      }
    }
  },
  "End": true
}
```

The "Parameters" block replaces the input to the iterations with a JSON node that contains both the current item data from the context object (p. 129), and the courier information from the "delivery-partner" field from the Map state input. The following is an example of input to a single iteration, that is passed to an invocation of the ship-val Lambda function.

```
{
  "parcel": {
    "prod": "R31",
    "dest-code": 9511,
    "quantity": 1344
   },
   "courier": "UQS"
}
```

In the previous Map state example, the "ResultPath" (p. 120) field produces output the same as the input, but with the “detail.shipped” field overwritten by an array in which each element is the output of the “ship-val” Lambda function for each iteration.

For more information see the following.

- Use a Map State to Call Lambda Multiple Times (p. 81)
- Input and Output Processing in Step Functions (p. 115)
- ItemsPath (p. 119)
- Context Object Data For Map States (p. 129)

Map State Input and Output Processing

For a map state, InputPath (p. 117) works as it does for other state types, selecting a subset of the input.

The input of a Map state must include a JSON array, and it will run the Iterator section once for each item in the array. You specify where in the input to find this array using the ItemsPath (p. 119) field. If not specified, the value of ItemsPath is "$", and the Iterator section expects that the array is the only input. A Map state may also include an ItemsPath (p. 119) field, whose value must be a Reference Path (p. 117). The ItemsPath field selects where in the input to find the array to use for iterations. The Reference Path is applied to the effective input (after InputPath is applied) and must identify a field whose value is a JSON array.

The input to each iteration, by default, is a single element of the array field identified by the ItemsPath value, This may be overridden using the Parameters (p. 118) field.

When complete, the output of the Map state is a JSON array, where each item is the output of an iteration.

For more information, see the following.

- Map State Tutorial (p. 81)
- Map State Example With Parameters (p. 111)
- Input and Output Processing in Step Functions (p. 115)
- Context Object Data For Map States (p. 129)
- Dynamically Process Data with a Map State (p. 187)

Transitions

When an execution of a state machine is launched, the system begins with the state referenced in the top-level `StartAt` field. This field (a string) must exactly match, including case, the name of one of the states.

After executing a state, AWS Step Functions uses the value of the `Next` field to determine the next state to advance to.

`Next` fields also specify state names as strings, and must match the name of a state specified in the state machine description exactly (case sensitive).

For example, the following state includes a transition to `NextState`.

```
"SomeState" : {
  ...,
  "Next" : "NextState"
}
```

Most states permit only a single transition rule via the `Next` field. However, certain flow-control states (for example, a `Choice` state) allow you to specify multiple transition rules, each with its own `Next` field. The Amazon States Language (p. 87) provides details about each of the state types you can specify, including information about how to specify transitions.

States can have multiple incoming transitions from other states.

The process repeats until it reaches a terminal state (a state with `"Type": Succeed`, `"Type": Fail`, or `"End": true`), or a runtime error occurs.

The following rules apply to states within a state machine:

- States can occur in any order within the enclosing block, but the order in which they're listed doesn't affect the order in which they're run. That order is determined by the contents of the states.
- Within a state machine, there can be only one state designated as the `start` state, which is designated by the value of the `StartAt` field in the top-level structure.
- Depending on your state machine logic—for example, if your state machine has multiple branches of execution—you may have more than one `end` state.
- If your state machine consists of only one state, it can be both the `start` state and the `end` state.

State Machine Data

State machine data takes the following forms:

- The initial input into a state machine

- Data passed between states
- The output from a state machine

This section describes how state machine data is formatted and used in AWS Step Functions.

Topics

Data Format

State machine data is represented by JSON text, so you can provide values using any data type supported by JSON:

> **Note**
>
> - Numbers in JSON text format conform to JavaScript semantics. These numbers typically correspond to double-precision IEEE-854 values.
> - The following is valid JSON text: standalone, quote-delimited strings; objects; arrays; numbers; Boolean values; and `null`.
> - The output of a state becomes the input into the next state. However, you can restrict states to working on a subset of the input data by using Input and Output Processing (p. 115).

State Machine Input/Output

You can give AWS Step Functions initial input data by passing it to a `StartExecution` action when you start an execution, or by passing initial data using the Step Functions console. Initial data is passed to the state machine's `StartAt` state. If no input is provided, the default is an empty object (`{}`).

The output of the execution is returned by the last state (`terminal`). This output appears as JSON text in the execution's result. You can retrieve execution results from the execution history using external callers (for example, in the `DescribeExecution` action). You can view execution results on the Step Functions console.

State Input/Output

Each state's input consists of JSON text from the preceding state or, for the `StartAt` state, the input into the execution. Certain flow-control states echo their input to their output.

In the following example, the state machine adds two numbers together.

1. Define the AWS Lambda function.

   ```
   function Add(input) {
     var numbers = JSON.parse(input).numbers;
     var total = numbers.reduce(
       function(previousValue, currentValue, index, array) {
         return previousValue + currentValue; });
     return JSON.stringify({ result: total });
   }
   ```

2. Define the state machine.

```
{
  "Comment": "An example that adds two numbers together.",
  "StartAt": "Add",
  "Version": "1.0",
  "TimeoutSeconds": 10,
  "States":
    {
        "Add": {
          "Type": "Task",
          "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Add",
          "End": true
        }
    }
}
```

3. Start an execution with the following JSON text.

```
{ "numbers": [3, 4] }
```

The `Add` state receives the JSON text and passes it to the Lambda function.

The Lambda function returns the result of the calculation to the state.

The state returns the following value in its output.

```
{ "result": 7 }
```

Because `Add` is also the final state in the state machine, this value is returned as the state machine's output.

If the final state returns no output, then the state machine returns an empty object (`{ }`).

For more information, see Input and Output Processing in Step Functions (p. 115).

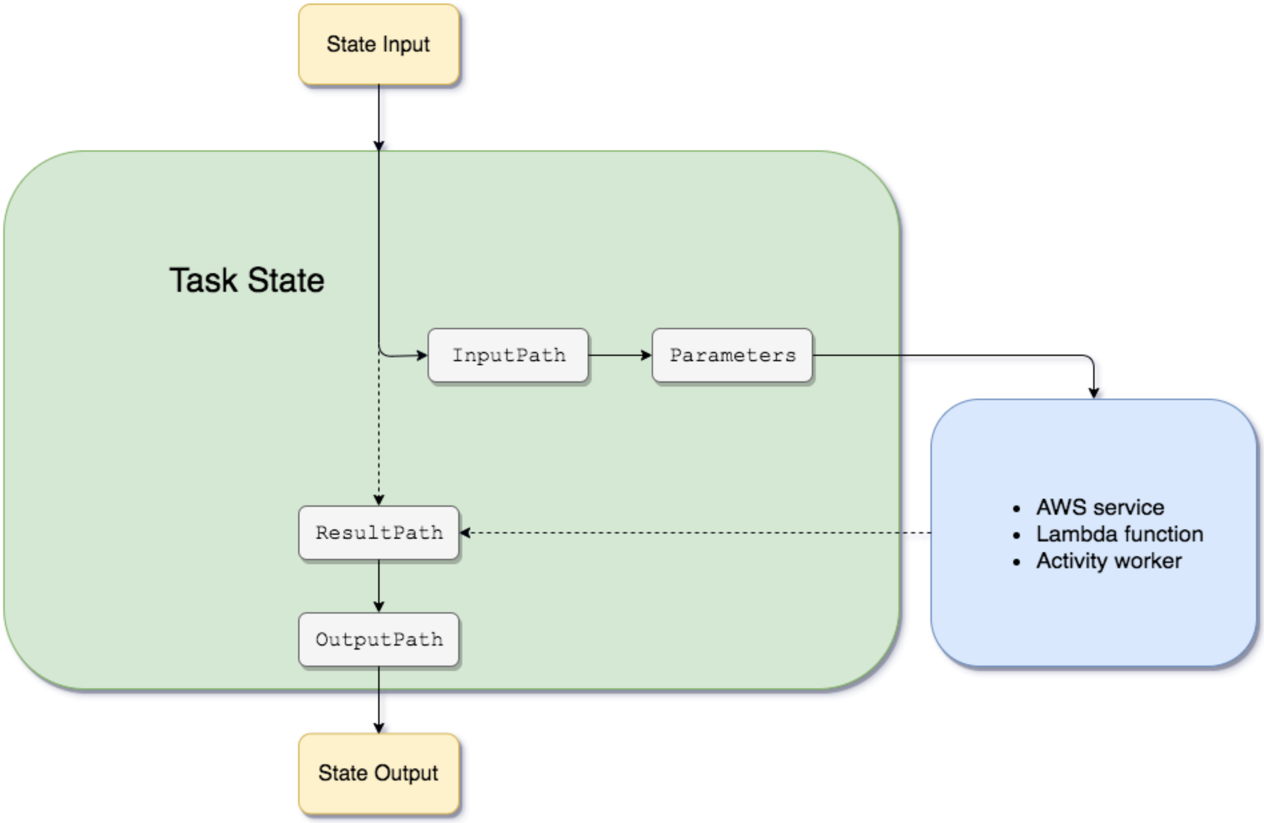
Input and Output Processing in Step Functions

A Step Functions execution receives a JSON file as input and passes that input to the first state in the workflow. Individual states receive JSON as input and usually pass JSON as output to the next state. Understanding how this information flows from state to state, and learning how to filter and manipulate this data, is key to effectively designing and implementing workflows in AWS Step Functions.

In the Amazon States Language, these fields filter and control the flow of JSON from state to state:

- `InputPath`
- `OutputPath`
- `ResultPath`
- `Parameters`

The following diagram shows how JSON information moves through a task state. `InputPath` selects which parts of the JSON input to pass to the task of the `Task` state (for example, an AWS Lambda function). `ResultPath` then selects what combination of the state input and the task result to pass to the output. `OutputPath` can filter the JSON output to further limit the information that's passed to the output.



`InputPath`, `Parameters`, `ResultPath`, and `OutputPath` each manipulate JSON as it moves through each state in your workflow.

Each can use paths (p. 116) to select portions of the JSON from the input or the result. A path is a string, beginning with `$`, that identifies nodes within JSON text. Step Functions paths use JsonPath syntax.

Topics

Paths

In the Amazon States Language, a *path* is a string beginning with `$` that you can use to identify components within JSON text. Paths follow JsonPath syntax. You can specify a path to access subsets of the input when specifying values for `InputPath`, `ResultPath`, and `OutputPath`. For more information see Input and Output Processing in Step Functions (p. 115).

> **Note**
> You can also specify a JSON node of the input or the context object by using paths within the `"Parameters"` field of a state definition. See Pass Parameters to a Service API (p. 144).

Reference Paths

A *reference path* is a path whose syntax is limited in such a way that it can identify only a single node in a JSON structure:

- You can access object fields using only dot (`.`) and square bracket (`[ ]`) notation.
- The operators `@ .. , : ? *` aren't supported.
- Functions such as `length()` aren't supported.

For example, if state input data contains the following values:

```
{
    "foo": 123,
    "bar": ["a", "b", "c"],
    "car": {
        "cdr": true
    }
}
```

The following reference paths would return the following.

```
$.foo => 123
$.bar => ["a", "b", "c"]
$.car.cdr => true
```

Certain states use paths and reference paths to control the flow of a state machine or configure a state's settings or options.

InputPath and Parameters

Both the `"InputPath"` and `"Parameters"` fields provide a way to manipulate JSON as it moves through your workflow. `InputPath` can limit the input that is passed by filtering the JSON notation by using a path (see Paths (p. 116)). The `"Parameters"` field enables you to pass a collection of key-value pairs, where the values are either static values that you define in your state machine definition, or that are selected from the input using a path.

AWS Step Functions applies the `"InputPath"` field first, and then the `"Parameters"` field. You can first filter your raw input to a selection you want using `InputPath`, and then apply `Parameters` to manipulate that input further, or add new values.

InputPath

Use `InputPath` to select a portion of the state input.

For example, suppose the input to your state includes the following.

```
{
  "comment": "Example for InputPath.",
  "dataset1": {
    "val1": 1,
    "val2": 2,
    "val3": 3
  },
  "dataset2": {
    "val1": "a",
    "val2": "b",
    "val3": "c"
  }
}
```

You could apply the `InputPath`.

```
"InputPath": "$.dataset2",
```

With the previous `InputPath`, the following is the JSON that is passed as the input.

```
{
  "val1": "a",
  "val2": "b",
  "val3": "c"
}
```

Note

A path can yield a selection of values. Consider the following example.

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

If you apply the path `$.a[0:2]`, the following is the result.

```
[ 1, 2 ]
```

Parameters

Use `"Parameters"` field to create a collection of key-value pairs that are passed as input. The values of each can either be static values that you include in your state machine definition, or selected from either the input or the context object with a path. For key-value pairs where the value is selected using a path, the key name must end in `.$`.

For example, suppose you provide the following input.

```
{
  "comment": "Example for Parameters.",
  "product": {
    "details": {
       "color": "blue",
       "size": "small",
       "material": "cotton"
    },
    "availability": "in stock",
    "sku": "2317",
    "cost": "$23"
  }
}
```

To select some of the information, you could specify these parameters in your state machine definition.

```
"Parameters": {
        "comment": "Selecting what I care about.",
        "MyDetails": {
          "size.$": "$.product.details.size",
          "exists.$": "$.product.availability",
          "StaticValue": "foo"
        }
      },
```

Given the previous input and the `"Parameters"` field, this is the JSON that is passed.

```
{
  "comment": "Selecting what I care about.",
  "MyDetails": {
      "size": "small",
      "exists": "in stock",
      "StaticValue": "foo"
  }
},
```

In addition to the input, you can access a special JSON object, known as the context object. The context object includes information about your state machine execution. See The Context Object (p. 127).

> **Note**
> The `"Parameters"` field can also pass information to connected resources. For example, if your task state is orchestrating an AWS Batch job, you can pass the relevant API parameters directly to the API actions of that service. For more information, see:
>
> - Pass Parameters to a Service API (p. 144)
> - Service Integrations (p. 138)

ItemsPath

The `ItemsPath` field is used in a `Map` state (p. 109) to select an array in the input. A `Map` state is used to iterate steps for each item in an array contained in the input. By default, a `Map` state sets `ItemsPath` to `"$"` selecting the entire input. If the input to the `Map` state is a JSON array it will run an iteration for each item in the array, passing that item to the iteration as input. For an example of this, see the Map State Tutorial (p. 81).

The `ItemsPath` field allows you to specify a location in the input to find the JSON array to use for iterations. The value of `ItemsPath` must be a Reference Path (p. 117), and it must identify a value that is a JSON array. For instance, consider input to a `Map` state that includes two arrays, like the following example.

```
{
  "ThingsPiratesSay": [
    {
      "say": "Avast!"
    },
    {
      "say": "Yar!"
    },
    {
      "say": "Walk the Plank!"
    }
  ],
  "ThingsGiantsSay": [
    {
      "say": "Fee!"
    },
    {
      "say": "Fi!"
    },
    {
      "say": "Fo!"
    },
    {
      "say": "Fum!"
    }
  ]
}
```

In this case, you could specify which array to use for `Map` state iterations by selecting a specific array with `ItemsPath`. The following state machine definition specifies the `"ThingsPiratesSay"` array in the input using `ItemsPath`, and will run an iteration of the `SayWord` pass state for each item in the `"ThingsPiratesSay"` array.

```
{
  "StartAt": "PiratesSay",
  "States": {
    "PiratesSay": {
      "Type": "Map",
      "ItemsPath": "$.ThingsPiratesSay",
      "Iterator": {
         "StartAt": "SayWord",
         "States": {
           "SayWord": {
             "Type": "Pass",
             "End": true
           }
         }
      },
      "End": true
    }
  }
}
```

When processing input, `ItemsPath` is applied after `InputPath` (p. 117). It operates on the effective input to the state, after `InputPath` has filtered the input.

For more information on `Map` states, see the following.

- Map State (p. 109)
- Map (p. 109)
- Map State Example (p. 110)
- Use a Map State to Call Lambda Multiple Times (p. 81)
- Map State Input and Output Processing (p. 112)
- Dynamically Process Data with a Map State (p. 187)

ResultPath

The output of a state can be a copy of its input, the result it produces (for example, output from a `Task` state's Lambda function), or a combination of its input and result. Use `ResultPath` to control which combination of these is passed to the state output.

The following state types can generate a result and can include `ResultPath:`

- Pass (p. 90)

- Task (p. 91)
- Parallel (p. 107)

Use `ResultPath` to combine a task result with task input, or to select one of these. The path you provide to `ResultPath` controls what information passes to the output.

> **Note**
>
> `ResultPath` is limited to using reference paths (p. 117), which limit scope so that it can identify only a single node in JSON. See Reference Paths (p. 117) in the Amazon States Language (p. 87).

These examples are based on the state machine and Lambda function described in the Creating a Lambda State Machine (p. 19) tutorial. Work through that tutorial and test different outputs by trying various paths in a `ResultPath` field.

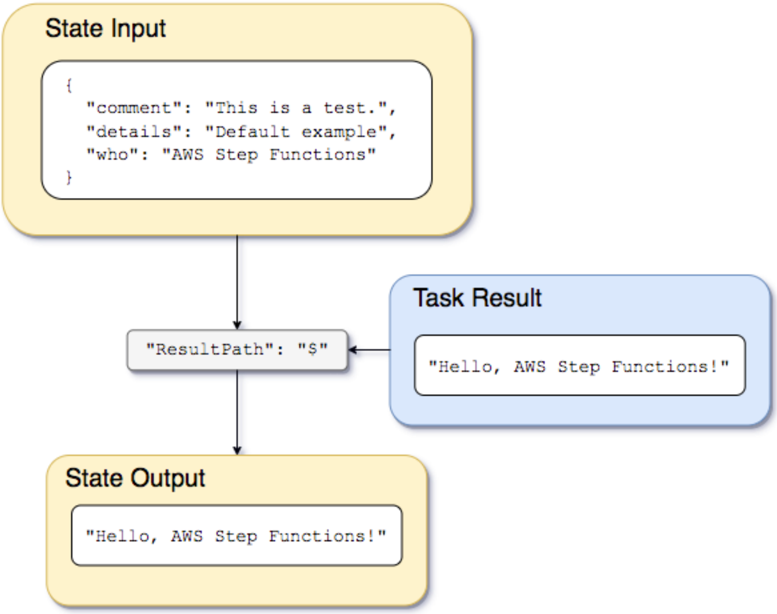
Use ResultPath to:

- Use ResultPath to Replace the Input with the Result (p. 121)
- Use ResultPath to Include the Result with the Input (p. 122)
- Use ResultPath to Update a Node in the Input with the Result (p. 123)
- Use ResultPath to Include Both Error and Input in a Catch (p. 125)

Use ResultPath to Replace the Input with the Result

If you don't specify a `ResultPath`, the default behavior is as if you had specified `"ResultPath": "$"`. Because this tells the state to replace the entire input with the result, the state input is completely replaced by the result coming from the task result.

The following diagram shows how `ResultPath` can completely replace the input with the result of the task.


State Input
{
"comment": "This is a test.",
"details": "Default example",
"who": "AWS Step Functions"
}
Task Result
"Hello, AWS Step Functions!"
"ResultPath": "$"
State Output
"Hello, AWS Step Functions!"

Using the state machine and Lambda function described in Creating a Lambda State Machine (p. 19), if we pass the following input:

```
{
  "comment": "This is a test of the input and output of a Task state.",
  "details": "Default example",
  "who": "AWS Step Functions"
}
```

The Lambda function provides the following result.

```
"Hello, AWS Step Functions!"
```

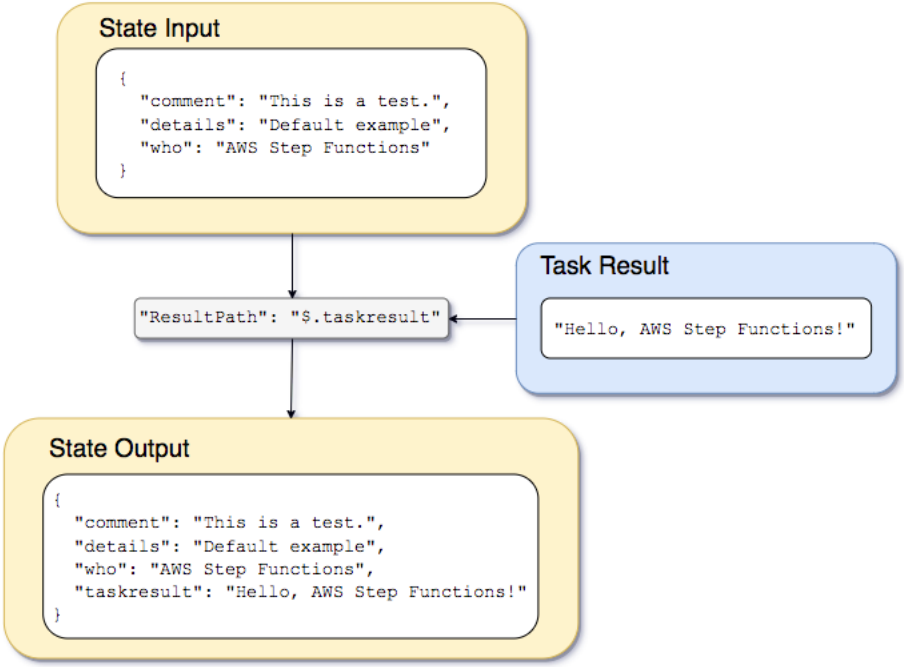
If `ResultPath` isn't specified in the state, or if `"ResultPath": "$"` is set, the input of the state is replaced by the result of the Lambda function, and the output of the state is the following.

```
"Hello, AWS Step Functions!"
```

> **Note**
> `ResultPath` is used to include content from the result with the input, before passing it to the output. But, if `ResultPath` isn't specified, the default is to replace the entire input.

Use ResultPath to Include the Result with the Input

The following diagram shows how `ResultPath` can include the result with the input.


State Input
{
"comment": "This is a test.",
"details": "Default example",
"who": "AWS Step Functions"
}
Task Result
"Hello, AWS Step Functions!"
"ResultPath": "$.taskresult"
State Output
{
"comment": "This is a test.",
"details": "Default example",
"who": "AWS Step Functions",
"taskresult": "Hello, AWS Step Functions!"
}

Using the state machine and Lambda function described in the Creating a Lambda State Machine (p. 19) tutorial, we could pass the following input.

```
{
  "comment": "This is a test of the input and output of a Task state.",
  "details": "Default example",
  "who": "AWS Step Functions"
}
```

The result of the Lambda function is the following.

```
"Hello, AWS Step Functions!"
```

If we want to preserve the input, insert the result of the Lambda function, and then pass the combined JSON to the next state, we could set `ResultPath` to the following.

```
"ResultPath": "$.taskresult"
```

This includes the result of the Lambda function with the original input.

```
{
  "comment": "This is a test of input and output of a Task state.",
  "details": "Default behavior example",
  "who": "AWS Step Functions",
  "taskresult": "Hello, AWS Step Functions!"
}
```

The output of the Lambda function is appended to the original input as a value for `taskresult`. The input, including the newly inserted value, is passed to the next state.

You can also insert the result into a child node of the input. Set the `ResultPath` to the following.

```
"ResultPath": "$.strings.lambdaresult"
```

Start an execution using the following input.

```
{
  "comment": "An input comment.",
  "strings": {
    "string1": "foo",
    "string2": "bar",
    "string3": "baz"
  },
  "who": "AWS Step Functions"
}
```

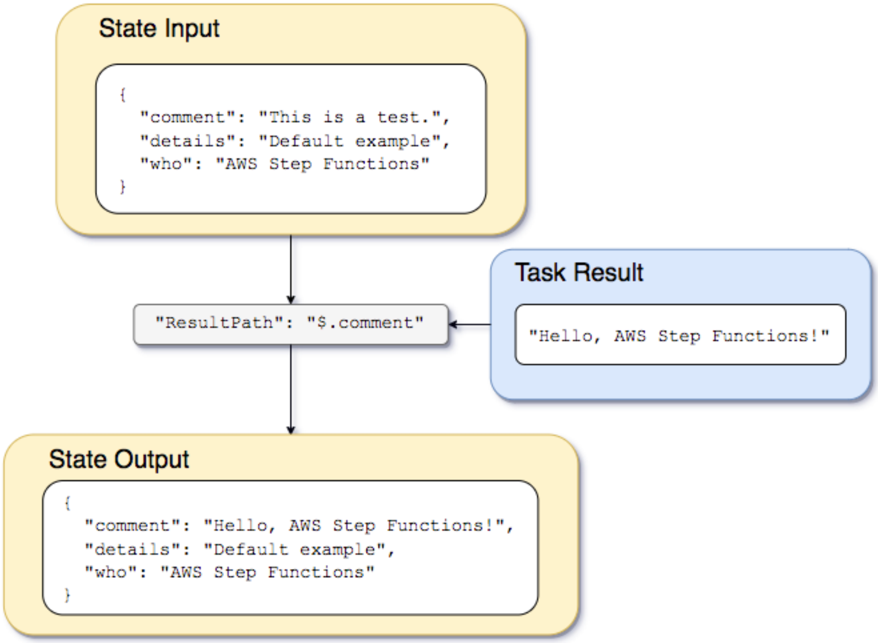
The result of the Lambda function is inserted as a child of the `strings` node in the input:

```
{
  "comment": "An input comment.",
  "strings": {
    "string1": "foo",
    "string2": "bar",
    "string3": "baz",
    "lambdaresult": "Hello, AWS Step Functions!"
  },
  "who": "AWS Step Functions"
}
```

The state output now includes the original input JSON with the result as a child node.

Use ResultPath to Update a Node in the Input with the Result

The following diagram shows how `ResultPath` can update the value of existing JSON nodes in the input with values from the task result.



```
{
  "comment": "This is a test.",
  "details": "Default example",
  "who": "AWS Step Functions"
}
```

```
"ResultPath": "$.comment"
```

```
"Hello, AWS Step Functions!"
```

```
{
  "comment": "Hello, AWS Step Functions!",
  "details": "Default example",
  "who": "AWS Step Functions"
}
```

Using the example of the state machine and Lambda function described in the Creating a Lambda State Machine (p. 19) tutorial, we could pass the following input.

```
{
  "comment": "This is a test of the input and output of a Task state.",
  "details": "Default example",
  "who": "AWS Step Functions"
}
```

The result of the Lambda function is the following.

```
Hello, AWS Step Functions!
```

Instead of preserving the input and inserting the result as a new node in the JSON, we can overwrite an existing node.

For example, just as omitting or setting `"ResultPath": "$"` overwrites the entire node, you can specify an individual node to overwrite with the result.

```
"ResultPath": "$.comment"
```

Because the `comment` node already exists in the state input, setting `ResultPath` to `"$.comment"` replaces that node in the input with the result of the Lambda function. Without further filtering by `OutputPath`, the following is passed to the output.

```
{
  "comment": "Hello, AWS Step Functions!",
  "details": "Default behavior example",
  "who": "AWS Step Functions",
}
```

The value for the comment node, "This is a test of the input and output of a Task state.", is replaced by the result of the Lambda function: "Hello, AWS Step Functions!" in the state output.

Use ResultPath to Include Both Error and Input in a Catch

The Handling Error Conditions Using a State Machine (p. 35) tutorial shows how to use a state machine to catch an error. In some cases, you might want to preserve the original input with the error. Use ResultPath in a Catch to include the error with the original input, instead of replacing it:

```
"Catch": [{
  "ErrorEquals": ["States.ALL"],
  "Next": "NextTask",
  "ResultPath": "$.error"
}]
```

If the previous Catch statement catches an error, it includes the result in an error node within the state input. For example, with the following input:

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

The state output when catching an error is the following.

```
{
  "foo": "bar",
  "error": {
    "Error": "Error here"
  }
}
```

For more information about error handling, see:

- Error Handling (p. 132)
- Handling Error Conditions Using a State Machine (p. 35)

OutputPath

OutputPath enables you to select a portion of the state output to pass to the next state. This enables you to filter out unwanted information, and pass only the portion of JSON that you care about.

If you don't specify an OutputPath the default value is $. This passes the entire JSON node (determined by the state input, the task result, and ResultPath) to the next state.

For more information, see:

- Paths in the Amazon States Language (p. 116)
- InputPath, ResultPath and OutputPath Example (p. 125)
- Pass Static JSON as Parameters (p. 144)
- Input and Output Processing in Step Functions (p. 115)

InputPath, ResultPath and OutputPath Example

Any state other than a Fail state can include InputPath, ResultPath or OutputPath. These allow you to use a path to filter the JSON as it moves through your workflow.

For example, start with the AWS Lambda function and state machine described in the Creating a Lambda State Machine (p. 19) tutorial. Modify the state machine so that it includes the following `InputPath`, `ResultPath`, and `OutputPath`.

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloFunction",
      "InputPath": "$.lambda",
      "ResultPath": "$.data.lambdaresult",
      "OutputPath": "$.data",
      "End": true
    }
  }
}
```

Start an execution using the following input.

```
{
  "comment": "An input comment.",
  "data": {
    "val1": 23,
    "val2": 17
  },
  "extra": "foo",
  "lambda": {
    "who": "AWS Step Functions"
  }
}
```

Assume that the `comment` and `extra` nodes can be discarded, but that we want to include the output of the Lambda function, and preserve the information in the `data` node.

In the updated state machine, the `Task` state is altered to process the input to the task.

```
"InputPath": "$.lambda",
```

This line in the state machine definition limits the task input to only the `lambda` node from the state input. The Lambda function receives only the JSON object `{"who": "AWS Step Functions"}` as input.

```
"ResultPath": "$.data.lambdaresult",
```

This `ResultPath` tells the state machine to insert the result of the Lambda function into a node named `lambdaresult`, as a child of the `data` node in the original state machine input. Without further processing with `OutputPath`, the input of the state now includes the result of the Lambda function with the original input.

```
{
  "comment": "An input comment.",
  "data": {
    "val1": 23,
    "val2": 17,
    "lambdaresult": "Hello, AWS Step Functions!"
  },
  "extra": "foo",
  "lambda": {
    "who": "AWS Step Functions"
  }
}
```

But, our goal was to preserve only the `data` node, and include the result of the Lambda function. `OutputPath` filters this combined JSON before passing it to the state output.

```
"OutputPath": "$.data",
```

This selects only the `data` node from the original input (including the `lambdaresult` child inserted by `ResultPath`) to be passed to the output. The state output is filtered to the following.

```
{
  "val1": 23,
  "val2": 17,
  "lambdaresult": "Hello, AWS Step Functions!"
}
```

In this `Task` state:

1. `InputPath` sends only the `lambda` node from the input to the Lambda function.
2. `ResultPath` inserts the result as a child of the `data` node in the original input.
3. `OutputPath` filters the state input (which now includes the result of the Lambda function) so that it passes only the `data` node to the state output.

For more information, see Input and Output Processing in Step Functions (p. 115).

The Context Object

The context object is an internal JSON structure that is available during an execution. It includes information about your state machine and execution that you can access from within the `"Parameters"` field of a state definition. This allows your workflows access to information about their specific execution.

Context Object Format

The context object includes information about the state machine, state, execution, and task. This JSON object includes nodes for each type of data, and is in the following format.

```
{
    "Execution": {
        "Id": "String",
        "Input": {},
        "StartTime": "Format: ISO 8601"
    },
    "State": {
        "EnteredTime": "Format: ISO 8601",
        "Name": "String",
        "RetryCount": Number
    },
    "StateMachine": {
        "Id": "String"
    },
    "Task": {
        "Token": "String"
    }
}
```

During an execution, the context object is populated with relevant data for the `"Parameters"` field from where it is accessed. The value for a `"Task"` field is null if the `"Parameters"` field is outside of a task state.

Content from a running execution includes specifics in the following format.

```
{
    "Execution": {
        "Id": "arn:aws:states:us-
east-1:123456789012:execution:stateMachineName:executionName",
        "Input": {
           "key": "value"
        },
        "Name": "executionName",
        "RoleArn": "arn:aws:iam::123456789012:role...",
        "StartTime": "2019-03-26T20:14:13.192Z"
    },
    "State": {
        "EnteredTime": "2019-03-26T20:14:13.192Z",
        "Name": "Test",
        "RetryCount": 3
    },
    "StateMachine": {
        "Id": "arn:aws:states:us-east-1:123456789012:stateMachine:stateMachineName",
        "Name": "name"
    },
    "Task": {
        "Token": "h7XRiCdLtd/83p1E0dMccoxlzFhglsdkzpK9mBVKZsp7d9yrT1W"
    }
}
```

Note

For context object data related to `Map` states, see Context Object Data For Map States (p. 129).

Accessing the Context Object

To access the context object, first specify the parameter name by appending `.$` to the end, as you do when selecting state input with a path. Then, to access context object data instead of the input, prepend the path with `$$.`. This tells Step Functions to use the path to select a node in the context object.

This example task state uses a path to retrieve and pass the execution ARN to an Amazon SQS message.

```
{
  "Order Flight Ticket Queue": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sqs:sendMessage",
    "Parameters": {
        "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/flight-purchase",
        "MessageBody": {
          "From": "YVR",
          "To": "SEA",
      "Execution.$": "$$.Execution.Id"
      }
    },
    "Next": "NEXT_STATE"
  }
}
```

Note

For more information about using the task token when calling an integrated service, see Wait for a Callback with the Task Token (p. 140).

Context Object Data For Map States

There are two additional items available in the context object when processing a `Map` state (p. 109): `Index` and `Value`. The `Index` contains the index number for the array item that is being processed in the current iteration. Within a `Map` state, the context object includes the following.

```
"Map": {
   "Item": {
      "Index": "Number",
      "Value": "String"
   }
}
```

These are available only in a `Map` state, and can be specified in the `"Parameters"` (p. 118) field, before the `"Iterator"` section.

Note

You must define parameters from the context object in the `"Parameters"` block of the main `Map` state, not within the states included in the `"Iterator"` section.

Given a state machine with a simple `Map` state, we can inject information from the context object as follows.

```
{
  "StartAt": "ExampleMapState",
  "States": {
    "ExampleMapState": {
      "Type": "Map",
      "Parameters": {
               "ContextIndex.$": "$$.Map.Item.Index",
               "ContextValue.$": "$$.Map.Item.Value"
             },
      "Iterator": {
         "StartAt": "TestPass",
         "States": {
           "TestPass": {
             "Type": "Pass",
             "End": true
           }
         }
      },
      "End": true
    }
  }
}
```

If you execute the previous state machine with the following input, `Index` and `Value` are inserted in the output.

```
[
  {
    "who": "bob"
  },
  {
    "who": "meg"
  },
  {
    "who": "joe"
  }
]
```

The output for the execution is the following.

```
[
  {
    "ContextValue": {
      "who": "bob"
    },
    "ContextIndex": 0
  },
  {
    "ContextValue": {
      "who": "meg"
    },
    "ContextIndex": 1
  },
  {
    "ContextValue": {
      "who": "joe"
    },
    "ContextIndex": 2
  }
]
```

Executions

A state machine *execution* occurs when an AWS Step Functions state machine runs and performs its tasks. Each Step Functions state machine can have multiple simultaneous executions, which you can initiate from the Step Functions console, or by using the AWS SDKs, the Step Functions API actions, or the AWS CLI. An execution receives JSON input and produces JSON output. You can start a Step Functions execution in the following ways.

- Call the `StartExecution` API action.
- Start a new execution (p. 14) in the Step Functions console.
- Use Amazon CloudWatch Events (p. 42) to start an execution.
- Start an execution with Amazon API Gateway (p. 48).
- Start a nested workflow execution (p. 130) from a `Task` state.

For more information about the different ways of working with Step Functions, see Development Options (p. 17).

Start Workflow Executions From a Task State

AWS Step Functions can start workflow executions directly from a `Task` state of a state machine. This allows you to break your workflows into smaller state machines, and to start executions of these other state machines. By starting these new workflow executions you can:

- Separate higher level workflow from lower level, task-specific workflows.
- Avoid repetitive elements by calling a separate state machine multiple times.
- Create a library of modular reusable workflows for faster development.
- Reduce complexity and make it easier to edit and troubleshoot state machines.

Step Functions can start these workflow executions by calling its own API as an integrated service (p. 138). Simply call the `StartExecution` API action from your `Task` state and pass the necessary parameters. You can call the Step Functions API using any of the service integration patterns (p. 139). To start a new execution of a state machine, use a `Task` state similar to the following.

```
{
   "Type":"Task",
   "Resource":"arn:aws:states:::states:startExecution",
   "Parameters":{
      "StateMachineArn":"arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
      "Input":{
         "Comment":"Hello world!"
      },
      "Retry":[
         {
            "ErrorEquals":[
               "StepFunctions.ExecutionLimitExceeded"
            ]
         }
      ],
      "End":true
   }
}
```

This `Task` state will start a new execution of the `HelloWorld` state machine, and will pass the JSON comment as input.

> **Note**
> The `StartExecution` API action limits can limit the number of executions that you can start. Use the `Retry` on `StepFunctions.ExecutionLimitExceeded` to ensure your execution is started. See the following.
>
> - Limits Related to API Action Throttling (p. 208)
> - Error Handling (p. 132)

Associate Workflow Executions

To associate a started workflow execution with the execution that started it, pass the execution ID from the context object to the execution input. You can access the ID from the context object from your `Task` state in a running execution. Pass the execution ID by appending `.$` to the parameter name, and referencing the ID in the context object with `$$.Execution.Id`.

```
"AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
```

You can use a special parameter named `AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID` when you start an execution. If included, this association provides links in the **Step details** section of the Step Functions console. When provided, you can easily trace the executions of your workflows from starting executions to their started workflow executions. Using the previous example, associate the execution ID with the started execution of the `HelloWorld` state machine as follows.

```
{
   "Type":"Task",
   "Resource":"arn:aws:states:::states:startExecution",
   "Parameters":{
      "StateMachineArn":"arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
      "Input": {
        "Comment": "Hello world!",
        "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
       }
   },
   "End":true
}
```

For more information, see:

- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)
- Accessing the Context Object (p. 128)
- AWS Step Functions (p. 162)

Error Handling

Any state can encounter runtime errors. Errors can happen for various reasons:

- State machine definition issues (for example, no matching rule in a `Choice` state)
- Task failures (for example, an exception in a Lambda function)
- Transient issues (for example, network partition events)

By default, when a state reports an error, AWS Step Functions causes the execution to fail entirely.

Error Names

Step Functions identifies errors in the Amazon States Language using case-sensitive strings, known as *error names*. The Amazon States Language defines a set of built-in strings that name well-known errors, all beginning with the `States.` prefix.

`States.ALL`

A wildcard that matches any known error name.

`States.Timeout`

A `Task` state either ran longer than the `TimeoutSeconds` value, or failed to send a heartbeat for a period longer than the `HeartbeatSeconds` value.

`States.TaskFailed`

A `Task` state failed during the execution.

`States.Permissions`

A `Task` state failed because it had insufficient privileges to execute the specified code.

States can report errors with other names. However, these must not begin with the `States.` prefix.

As a best practice, ensure production code can handle AWS Lambda service exceptions (`Lambda.ServiceException` and `Lambda.SdkclientException`). For more information, see Handle Lambda Service Exceptions (p. 204).

> **Note**
> Unhandled errors in Lambda are reported as `Lambda.Unknown` in the error output. These include out-of-memory errors, function timeouts, and hitting the concurrent Lambda invoke limit. You can match on `Lambda.Unknown`, `States.ALL`, or `States.TaskFailed` to handle these errors. When Lambda hits the invocation limit, the error is `Lambda.TooManyRequestsException`. For more information about Lambda `Handled` and `Unhandled` errors, see `FunctionError` in the AWS Lambda Developer Guide.

Retrying after an Error

`Task` and `Parallel` states can have a field named `Retry`, whose value must be an array of objects known as *retriers*. An individual retrier represents a certain number of retries, usually at increasing time intervals.

> **Note**
> Retries are treated as state transitions. For information about how state transitions affect billing, see Step Functions Pricing.

A retrier contains the following fields.

`ErrorEquals` (Required)

A non-empty array of strings that match error names. When a state reports an error, Step Functions scans through the retriers. When the error name appears in this array, it implements the retry policy described in this retrier.

`IntervalSeconds` (Optional)

An integer that represents the number of seconds before the first retry attempt (`1` by default).

`MaxAttempts` (Optional)

A positive integer that represents the maximum number of retry attempts (`3` by default). If the error recurs more times than specified, retries cease and normal error handling resumes. A value of `0` specifies that the error or errors are never retried.

`BackoffRate` (Optional)

The multiplier by which the retry interval increases during each attempt (`2.0` by default).

This example of a `Retry` makes 2 retry attempts after waiting for 3 and 4.5 seconds.

```
"Retry": [ {
   "ErrorEquals": [ "States.Timeout" ],
   "IntervalSeconds": 3,
   "MaxAttempts": 2,
   "BackoffRate": 1.5
} ]
```

The reserved name `States.ALL` that appears in a retrier's `ErrorEquals` field is a wildcard that matches any error name. It must appear alone in the `ErrorEquals` array and must appear in the last retrier in the `Retry` array.

This example of a `Retry` field retries any error except `States.Timeout`.

```
"Retry": [ {
   "ErrorEquals": [ "States.Timeout" ],
   "MaxAttempts": 0
}, {
   "ErrorEquals": [ "States.ALL" ]
} ]
```

Complex Retry Scenarios

A retrier's parameters apply across all visits to the retrier in the context of a single-state execution.

Consider the following `Task` state:

```
"X": {
   "Type": "Task",
   "Resource": "arn:aws:states:us-east-1:123456789012:task:X",
   "Next": "Y",
   "Retry": [ {
      "ErrorEquals": [ "ErrorA", "ErrorB" ],
      "IntervalSeconds": 1,
      "BackoffRate": 2.0,
      "MaxAttempts": 2
   }, {
      "ErrorEquals": [ "ErrorC" ],
      "IntervalSeconds": 5
   } ],
   "Catch": [ {
      "ErrorEquals": [ "States.ALL" ],
      "Next": "Z"
   } ]
}
```

This task fails five times in succession, outputting these error names: `ErrorA`, `ErrorB`, `ErrorC`, `ErrorB`, and `ErrorB`. The following occurs as a result:

- The first two errors match the first retrier and cause waits of 1 and 2 seconds.
- The third error matches the second retrier and causes a wait of 5 seconds.
- The fourth error matches the first retrier and causes a wait of 4 seconds.
- The fifth error also matches the first retrier. However, it has already reached its limit of two retries (`MaxAttempts`) for that particular error (`ErrorB`), so it fails and execution is redirected to the `Z` state via the `Catch` field.

Fallback States

`Task` and `Parallel` states can have a field named `Catch`. This field's value must be an array of objects, known as *catchers*.

A catcher contains the following fields.

`ErrorEquals` (Required)

A non-empty array of strings that match error names, specified exactly as they are with the retrier field of the same name.

`Next` (Required)

A string that must exactly match one of the state machine's state names.

`ResultPath` (Optional)

A path (p. 115) that determines what input is sent to the state specified in the `Next` field.

When a state reports an error and either there is no `Retry` field, or if retries fail to resolve the error, Step Functions scans through the catchers in the order listed in the array. When the error name appears in the value of a catcher's `ErrorEquals` field, the state machine transitions to the state named in the `Next` field.

The reserved name `States.ALL` that appears in a catcher's `ErrorEquals` field is a wildcard that matches any error name. It must appear alone in the `ErrorEquals` array and must appear in the last catcher in the `Catch` array.

The following example of a `Catch` field transitions to the state named `RecoveryState` when a Lambda function outputs an unhandled Java exception. Otherwise, the field transitions to the `EndState` state.

```
"Catch": [ {
   "ErrorEquals": [ "java.lang.Exception" ],
   "ResultPath": "$.error-info",
   "Next": "RecoveryState"
}, {
   "ErrorEquals": [ "States.ALL" ],
   "Next": "EndState"
} ]
```

> **Note**
> Each catcher can specify multiple errors to handle.

Error Output

When Step Functions transitions to the state specified in a catch name, the object usually contains the field `Cause`. This field's value is a human-readable description of the error. This object is known as the *error output*.

In this example, the first catcher contains a `ResultPath` field. This works similarly to a `ResultPath` field in a state's top level, resulting in two possibilities:

- It takes the results of executing the state and overwrites a portion of the state's input (or all of the state's input).
- It takes the results and adds them to the input. In the case of an error handled by a catcher, the result of executing the state is the error output.

Thus, in this example, for the first catcher the error output is added to the input as a field named `error-info` (if there isn't already a field with this name in the input). Then, the entire input is sent to `RecoveryState`. For the second catcher, the error output overwrites the input and only the error output is sent to `EndState`.

> **Note**
> If you don't specify the `ResultPath` field, it defaults to `$`, which selects and overwrites the entire input.

When a state has both `Retry` and `Catch` fields, Step Functions uses any appropriate retriers first, and only afterward applies the matching catcher transition if the retry policy fails to resolve the error.

Examples Using Retry and Using Catch

The state machines defined in the following examples assume the existence of two Lambda functions: one that always fails and one that waits long enough to allow a timeout defined in the state machine to occur.

This is a definition of a Lambda function that always fails, returning the message `error`. In the state machine examples that follow, this Lambda function is named `FailFunction`.

```
exports.handler = (event, context, callback) => {
    callback("error");
};
```

This is a definition of a Lambda function that sleeps for 10 seconds. In the state machine examples that follow, this Lambda function is named `sleep10`.

> **Note**
> When you create this Lambda function in the Lambda console, remember to change the **Timeout** value in the **Advanced settings** section from 3 seconds (default) to 11 seconds.

```
exports.handler = (event, context, callback) => {
    setTimeout(function(){
    }, 11000);
};
```

Handling a Failure Using Retry

This state machine uses a `Retry` field to retry a function that fails and outputs the error name `HandledError`. The function is retried twice with an exponential backoff between retries.

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FailFunction",
         "Retry": [ {
            "ErrorEquals": ["HandledError"],
            "IntervalSeconds": 1,
            "MaxAttempts": 2,
            "BackoffRate": 2.0
         } ],
      "End": true
      }
   }
}
```

This variant uses the predefined error code `States.TaskFailed`, which matches any error that a Lambda function outputs.

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FailFunction",
         "Retry": [ {
            "ErrorEquals": ["States.TaskFailed"],
            "IntervalSeconds": 1,
            "MaxAttempts": 2,
            "BackoffRate": 2.0
         } ],
         "End": true
      }
   }
}
```

Note
As a best practice, tasks that reference a Lambda function should handle Lambda service exceptions. For more information, see Handle Lambda Service Exceptions (p. 204).

Handling a Failure Using Catch

This example uses a `Catch` field. When a Lambda function outputs an error, the error is caught and the state machine transitions to the `fallback` state.

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FailFunction",
         "Catch": [ {
            "ErrorEquals": ["HandledError"],
            "Next": "fallback"
         } ],
         "End": true
      },
      "fallback": {
         "Type": "Pass",
         "Result": "Hello, AWS Step Functions!",
         "End": true
      }
   }
}
```

This variant uses the predefined error code `States.TaskFailed`, which matches any error that a Lambda function outputs.

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FailFunction",
         "Catch": [ {
            "ErrorEquals": ["States.TaskFailed"],
            "Next": "fallback"
         } ],
      "End": true
      },
      "fallback": {
         "Type": "Pass",
         "Result": "Hello, AWS Step Functions!",
         "End": true
      }
   }
}
```

Handling a Timeout Using Retry

This state machine uses a `Retry` field to retry a function that times out. The function is retried twice with an exponential backoff between retries.

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:us-east-1:123456789012:function:sleep10",
         "TimeoutSeconds": 2,
         "Retry": [ {
            "ErrorEquals": ["States.Timeout"],
            "IntervalSeconds": 1,
            "MaxAttempts": 2,
            "BackoffRate": 2.0
         } ],
         "End": true
      }
   }
}
```

Handling a Timeout Using Catch

This example uses a `Catch` field. When a timeout occurs, the state machine transitions to the `fallback` state.

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:us-east-1:123456789012:function:sleep10",
         "TimeoutSeconds": 2,
         "Catch": [ {
            "ErrorEquals": ["States.Timeout"],
            "Next": "fallback"
         } ],
         "End": true
      },
      "fallback": {
         "Type": "Pass",
         "Result": "Hello, AWS Step Functions!",
         "End": true
      }
   }
}
```

Note
You can preserve the state input and the error by using `ResultPath`. See Use ResultPath to Include Both Error and Input in a `Catch` (p. 125).

AWS Service Integrations

AWS Step Functions integrates with some AWS services so that you can call API actions, and coordinate executions directly from the Amazon States Language in Step Functions. You can directly call and pass parameters to the API of those services. You coordinate these services directly from a task state in the Amazon States Language. For example, using Step Functions, you can call other services to:

- Invoke an AWS Lambda function.
- Run an AWS Batch job and then perform different actions based on the results.
- Insert or get an item from Amazon DynamoDB.
- Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
- Publish to a topic in Amazon Simple Notification Service (Amazon SNS).
- Send a message in Amazon Simple Queue Service (Amazon SQS).
- Manage a job for AWS Glue or Amazon SageMaker.
- Launch an AWS Step Functions workflow execution.

Supported Service Integrations

Service	Request Response (p. 140)	Run a Job (`.sync`) (p. 140)	Wait for Callback (`.waitForTaskToken`) (p. 140)
AWS Lambda (p. 146)	✓		✓
AWS Batch (p. 148)	✓	✓	
Amazon DynamoDB (p. 149)	✓		
Amazon ECS/ Fargate (p. 151)	✓	✓	✓
Amazon Simple Notification Service (p. 153)	✓		✓
Amazon Simple Queue Service (p. 154)	✓		✓
AWS Glue (p. 156)	✓	✓	
Amazon SageMaker (p. 156)	✓	✓	
AWS Step Functions (p. 162)	✓	✓	✓

Topics

Service Integration Patterns

AWS Step Functions integrates with services directly in the Amazon States Language. You can control these AWS services using three different service integration patterns.

- Call a service and let Step Functions progress to the next state immediately after it gets an HTTP response.
- Call a service and have Step Functions wait for a job to complete.
- Call a service with a task token and have Step Functions wait until that token is returned along with a payload.

Each of these service integration patterns is controlled by how you create a URI in the `"Resource"` field of your task definition (p. 91).

Ways to Call an Integrated Service

> **Note**
> For information about configuring AWS Identity and Access Management (IAM) for integrated services, see IAM Policies for Integrated Services (p. 234).

Request Response

When you specify a service in the `"Resource"` string of your task state, and you *only* provide the resource, Step Functions will wait for an HTTP response and then progress to the next state. Step Functions will not wait for a job to complete.

The following example shows how you can publish an Amazon SNS topic.

```
"Send message to SNS":{
   "Type":"Task",
   "Resource":"arn:aws:states:::sns:publish",
   "Parameters":{
      "TopicArn":"arn:aws:sns:us-east-1:123456789012:myTopic",
      "Message":"Hello from Step Functions!"
   },
   "Next":"NEXT_STATE"
}
```

This example references the Publish API of Amazon SNS. The workflow will progress to the next state after calling the `Publish` API.

Run a Job

For integrated services such as AWS Batch and Amazon ECS, Step Functions can wait for a request to complete before progressing to the next state. To have Step Functions wait, specify the `"Resource"` field in your task state definition with the `.sync` suffix appended after the resource URI.

For example, when submitting an AWS Batch job, see the `"Resource"` field in the state machine definition that follows.

```
"Manage Batch task": {
  "Type": "Task",
  "Resource": "arn:aws:states:::batch:submitJob.sync",
  "Parameters": {
    "JobDefinition": "arn:aws:batch:us-east-2:123456789012:job-definition/
testJobDefinition",
    "JobName": "testJob",
    "JobQueue": "arn:aws:batch:us-east-2:123456789012:job-queue/testQueue"
  },
  "Next": "NEXT_STATE"
}
```

The `.sync` portion appended to the resource ARN tells Step Functions to wait for the job to complete. After calling AWS Batch `submitJob`, the workflow pauses. When the job is complete, Step Functions progresses to the next state. For more information, see the AWS Batch sample project: Manage a Batch Job (AWS Batch, Amazon SNS) (p. 168).

> **Note**
> To see a list of what integrated services support waiting for a job to complete (`.sync`), see Supported AWS Service Integrations for Step Functions (p. 146).

Wait for a Callback with the Task Token

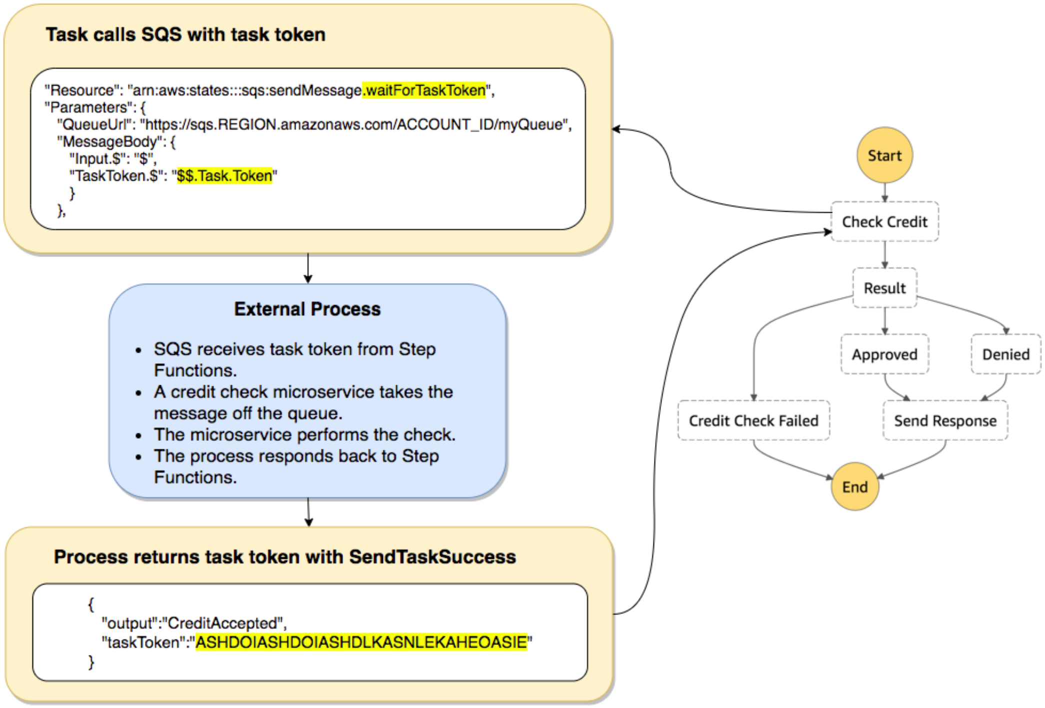
Callback tasks provide a way to pause a workflow until a task token is returned. A task might need to wait for a human approval, integrate with a third party, or call legacy systems. For tasks like these, you can pause AWS Step Functions indefinitely, and wait for an external process or workflow to complete. For these situations Step Functions allows you to pass a task token to some integrated services. The task will pause until it receives that task token back with a `SendTaskSuccess` or `SendTaskFailure` call.

> **Note**
> To see a list of what integrated services support waiting for a task token (`.waitForTaskToken`), see Supported AWS Service Integrations for Step Functions (p. 146).

Topics

Task Token Example

In this example a Step Functions workflow needs to integrate with an external microservice to perform a credit check as a part of an approval workflow. Step Functions publishes an Amazon SQS message that includes a task token as a part of the message. An external system integrates with Amazon SQS, and pulls the message off the queue. When that's finished, it returns the result and the original task token. Step Functions then continues with its workflow.



Task calls SQS with task token

```
"Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
"Parameters": {
  "QueueUrl": "https://sqs.REGION.amazonaws.com/ACCOUNT_ID/myQueue",
  "MessageBody": {
    "Input.$": "$",
    "TaskToken.$": "$$.Task.Token"
    }
  },
```

External Process

- SQS receives task token from Step Functions.
- A credit check microservice takes the message off the queue.
- The microservice performs the check.
- The process responds back to Step Functions.

Process returns task token with SendTaskSuccess

```
{
  "output":"CreditAccepted",
  "taskToken":"ASHDOIASHDOIASHDLKASNLEKAHEOASIE"
}
```

The `"Resource"` field of the task definition that references Amazon SQS includes `.waitForTaskToken` appended to the end.

```
"Send message to SQS": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
  "Parameters": {
    "QueueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/myQueue",
    "MessageBody": {
        "Message": "Hello from Step Functions!",
        "TaskToken.$": "$$.Task.Token"
     }
  },
  "Next": "NEXT_STATE"
}
```

This tells Step Functions to pause and wait for the task token. When you specify a resource using `.waitForTaskToken`, the task token can be accessed in the `"Parameters"` field of your state definition with a special path designation (`$$.Task.Token`). The initial `$$.` designates that the path accesses the context object (p. 142), and gets the task token for the current task in a running execution.

When it's complete, the external service calls `SendTaskSuccess` or `SendTaskFailure` with the `taskToken` included. Only then does the workflow continue to the next state.

> **Note**
>
> To avoid waiting indefinitely if a process fails to send the task token along with `SendTaskSuccess` or `SendTaskFailure`, see Configure a Heartbeat Timeout for a Waiting Task (p. 143).

Get a Token from the Context Object

The context object is an internal JSON object that contains information about your execution. Like state input, it can be accessed with a path from the `"Parameters"` field during an execution. When accessed from within a task definition, it includes information about the specific execution, including the task token.

```
{
    "Execution": {
        "Id": "arn:aws:states:us-
east-1:123456789012:execution:stateMachineName:executionName",
        "Input": {
           "key": "value"
        },
        "Name": "executionName",
        "RoleArn": "arn:aws:iam::123456789012:role...",
        "StartTime": "2019-03-26T20:14:13.192Z"
    },
    "State": {
        "EnteredTime": "2019-03-26T20:14:13.192Z",
        "Name": "Test",
        "RetryCount": 3
    },
    "StateMachine": {
        "Id": "arn:aws:states:us-east-1:123456789012:stateMachine:stateMachineName",
        "Name": "name"
    },
    "Task": {
        "Token": "h7XRiCdLtd/83p1E0dMccoxlzFhglsdkzpK9mBVKZsp7d9yrT1W"
    }
}
```

You can access the task token by using a special path from inside the `"Parameters"` field of your task definition. To access the input or the context object, you first specify that the parameter will be a path by appending a `.$` to the parameter name. The following specifies nodes from both the input and the context object in a `"Parameters"` specification.

```
"Parameters": {
   "Input.$": "$",
   "TaskToken.$": "$$.Task.Token"
 },
```

In both cases appending `.$` to the parameter name tells Step Functions to expect a path. In the first case, `"$"` is a path that includes the entire input. In the second case, `$$.` specifies that the path will access the context object, and `$$.Task.Token` sets the parameter to the value of the task token in the context object of a running execution.

In the Amazon SQS example, `.waitForTaskToken` in the `"Resource"` field tells Step Functions to wait for the task token to be returned. The `"TaskToken.$": "$$.Task.Token"` parameter passes that token as a part of the Amazon SQS message.

```
"Send message to SQS": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
  "Parameters": {
    "QueueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/myQueue",
    "MessageBody": {
        "Message": "Hello from Step Functions!",
        "TaskToken.$": "$$.Task.Token"
     }
  },
  "Next": "NEXT_STATE"
}
```

For more information about the context object, see The Context Object (p. 127) in the Input and Output Processing (p. 115) section in this guide.

Configure a Heartbeat Timeout for a Waiting Task

A task that is waiting for a task token will wait until the execution reaches the one year service limit (see, Limits Related to State Throttling (p. 210)). To avoid stuck executions you can configure a heartbeat timeout interval in your state machine definition. Use the `HeartbeatSeconds` (p. 91) field to specify the timeout interval.

```
{
  "StartAt": "Push based to SQS",
  "States": {
    "Push to SQS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
      "HeartbeatSeconds": 600,
      "Parameters": {
        "MessageBody": { "myTaskToken.$": "$$.Task.Token" },
        "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/push-based-queue"
      },
      "ResultPath": "$.SQS",
      "End": true
    }
  }
}
```

In this state machine definition, a task pushes a message to Amazon SQS and waits for an external process to call back with the provided task token. The `"HeartbeatSeconds": 600` field sets the heartbeat timeout interval to 10 minutes. The task will wait for the task token to be returned with one of these API actions:

- `SendTaskSuccess`
- `SendTaskFailure`
- `SendTaskHeartbeat`

If the waiting task doesn't receive a valid task token within that 10 minute period, the task fails with a `States.Timeout` error name.

For more information, see the callback task sample project: Callback Pattern Example (p. 182).

Pass Parameters to a Service API

Use the `"Parameters"` field in a `Task` state to control what parameters are passed to a service API.

Pass Static JSON as Parameters

You can include a JSON object directly in your state machine definition to pass as a parameter to a resource.

For example, to set the `RetryStrategy` parameter for the `SubmitJob` API for AWS Batch, you could include the following in your parameters.

```
"RetryStrategy": {
  "attempts": 5
}
```

You can also pass multiple parameters with static JSON. As a more complete example, the following are the `"Resource"` and `"Parameters"` fields of the specification of a task that publishes an Amazon SNS topic.

```
"Resource": "arn:aws:states:::sns:publish",
     "Parameters": {
     "TopicArn": "arn:aws:sns:us-east-1:123456789012:myTopic",
     "Message": "test message",
     "MessageAttributes": {
       "my attribute no 1": {
         "DataType": "String",
         "StringValue": "value of my attribute no 1"
       },
       "my attribute no 2": {
         "DataType": "String",
         "StringValue": "value of my attribute no 2"
       }
     }
   },
```

Pass State Input as Parameters Using Paths

You can pass portions of the state input into parameters by using paths (p. 116). A path is a string, beginning with `$`, that's used to identify components within JSON text. Step Functions paths use JsonPath syntax.

To specify that a parameter use a path to reference a JSON node in the input, end the parameter name with `.$`. For example, if you have text in your state input in a node named `message`, you could pass that to a parameter by referencing the input JSON with a path.

Using the following state input:

```
{
  "comment": "A message in the state input",
  "input": {
    "message": "foo",
    "otherInfo": "bar"
  },
  "data": "example"
}
```

You could pass the message `foo` as a parameter using: the following.

```
"Parameters": {"Message.$": "$.input.message"},
```

For more information about using parameters in Step Functions, see:

- Input and Output Processing (p. 115)
- InputPath and Parameters (p. 117)

Pass Context Object Nodes as Parameters

In addition to static content, and nodes from the state input, you can pass nodes from the context object as parameters. The context object is dynamic JSON data that exists during a state machine execution. It includes information about your state machine and the current execution. You can access the context object using a path in the `"Parameters"` field of a state definition.

For more information about the context object and how to access that data from a `"Parameters"` field, see:

- The Context Object (p. 127)
- Accessing the Context Object (p. 128)
- Get a Token from the Context Object (p. 142)

Code Snippets

In AWS Step Functions, code snippets are a way to easily configure the options for a new state in your state machine definition. When you edit or create a state machine, the top of the code pane includes a **Generate code snippet** menu. Selecting an option from the **Generate code snippet** menu opens a window to configure parameters specific to that state, and generates Amazon States Language code based on the options you choose.

For example, if you choose the **AWS Batch: Manage a job** code snippet, you can configure the following:

- **Batch job name** – You can either specify the job name, or specify it at runtime using a path.
- **Batch job definition** – You can select the ARN of an existing AWS Batch job in your account, enter the job definition, or choose to specify it at runtime using a path.
- **Batch job queue** – You can select the ARN of an existing AWS Batch job queue in your account, enter the job queue definition, or choose to specify it at runtime using a path.
- **Run synchronously** – Selecting this option configures Step Functions to wait until the AWS Batch job completes before continuing to the next state.

> **Note**
> For more information about specifying service parameters, see Pass Parameters to a Service API (p. 144).

After you configure your AWS Batch options, you can specify error handling options for your state, such as `Retry`, `Catch`, and `TimeoutSeconds`.

For more information, see Error Names (p. 132) in Amazon States Language.

To learn more about Step Functions service integrations, see:

- AWS Service Integrations (p. 138)
- Supported AWS Service Integrations for Step Functions (p. 146)

- Using Code Snippets (p. 67)

Supported AWS Service Integrations for Step Functions

The following topics include the supported APIs, parameters, and request/response syntax in the Amazon States Language for coordinating other AWS services, and provide example code. You can call integrated services directly from the Amazon States Language in the `Resource` field of a task state. There are three different service integration patterns you can use.

- Default response (p. 140)
- Wait for a job to complete (`.sync`). (p. 140)
- Wait for a task token (`.waitForTaskToken`). (p. 140)

For more information on service integrations, see Service Integration Patterns (p. 139).

Supported Service Integrations

Service	Request Response (p. 140)	Run a Job (`.sync`) (p. 140)	Wait for Callback (`.waitForTaskToken`) (p. 140)
Lambda (p. 146)	✓		✓
AWS Batch (p. 148)	✓	✓	
DynamoDB (p. 149)	✓		
Amazon ECS/ Fargate (p. 151)	✓	✓	✓
Amazon SNS (p. 153)	✓		✓
Amazon SQS (p. 154)	✓		✓
AWS Glue (p. 156)	✓	✓	
Amazon SageMaker (p. 156)	✓	✓	
Step Functions (p. 162)	✓	✓	✓

Invoke Lambda with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information, see:

- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)

For more information about managing state input, output and results, see Input and Output Processing in Step Functions (p. 115).

Supported APIs:

- `Invoke`
  - Request Syntax
  - Supported Parameters
    - `ClientContext`
    - `FunctionName`
    - `InvocationType`
    - `Qualifier`
    - `Payload`
  - Response syntax

The following includes a `Task` state that invokes a Lambda function.

```
{
   "StartAt":"CallLambda",
   "States":{
      "CallLambda":{
         "Type":"Task",
         "Resource":"arn:aws:states:::lambda:invoke",
         "Parameters":{
            "FunctionName":"MyFunction"
         },
         "End":true
      }
   }
}
```

The following includes a `Task` state that implements the callback (p. 140) service integration pattern.

```
{
   "StartAt":"GetManualReview",
   "States":{
      "GetManualReview":{
         "Type":"Task",
         "Resource":"arn:aws:states:::lambda:invoke.waitForTaskToken",
         "Parameters":{
            "FunctionName":"get-model-review-decision",
            "Payload":{
               "model.$":"$.new_model",
               "token.$":"$$.Task.Token"
            },
            "Qualifier":"prod-v1"
         },
         "End":true
      }
   }
}
```

> **Note**
>
> The Lambda invoke API includes logs in the response by default. Multiple Lambda invocations in a workflow can trigger `States.DataLimitExceeded` errors. To avoid this, include `"LogType" = "None"` as a parameter when you invoke your Lambda functions.

To simply invoke a Lambda function, you can also call the resource ARN directly in the `"Resource"` string.

```
{
   "StartAt":"CallFunction",
   "States":{
      "CallFunction": {
         "Type":"Task",
         "Resource":"arn:aws:lambda:us-east-1:123456789012:function:HelloFunction",
         "End": true
      }
   }
}
```

When you invoke Lambda using an ARN directly from the `"Resource"` field, you cannot specify `.waitForTaskToken`, and the state input is passed to Lambda as the payload. The output of the Lambda function is the result. To have a Lambda task wait for a task token, see the previous callback pattern example.

You can invoke a specific Lambda function version or alias by specifying those options in the Amazon Resource Name in the `Resource` field. See the following in the Lambda documentation.

- AWS Lambda versioning
- AWS Lambda aliases

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Manage AWS Batch with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information, see:

- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)

Supported APIs:

> **Note**
> Parameters in Step Functions are expressed in `PascalCase`, even when the native service API is `camelCase`.

- `SubmitJob`
  - Request syntax
  - Supported parameters:
    - `ArrayProperties`
    - `ContainerOverrides`
    - `DependsOn`
    - `JobDefinition`
    - `JobName`
    - `JobQueue`
    - `Parameters`
    - `RetryStrategy`
    - `Timeout`
  - Response syntax

The following includes a `Task` state that submits an AWS Batch job and waits for it to complete.

```
{
 "StartAt": "BATCH_JOB",
 "States": {
   "BATCH_JOB": {
     "Type": "Task",
     "Resource": "arn:aws:states:::batch:submitJob.sync",
     "Parameters": {
       "JobDefinition": "preprocessing",
       "JobName": "PreprocessingBatchJob",
       "JobQueue": "SecondaryQueue",
       "Parameters.$": "$.batchjob.parameters",
       "ContainerOverrides": {
          "Vcpus": 4
        }
     },
     "End": true
    }
  }
}
```

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Call DynamoDB APIs with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information, see:

- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)

> **Note**
> There is a limit on the maximum input or result data size for a task in Step Functions. This limits you to 32,768 characters of data when you send to, or receive data from, another service. See Limits Related to State Machine Executions (p. 207).

Supported DynamoDB APIs and syntax:

- `GetItem`
  - Request syntax
  - Supported parameters:
    - `Key`
    - `TableName`
    - `AttributesToGet`
    - `ConsistentRead`
    - `ExpressionAttributeNames`
    - `ProjectionExpression`
    - `ReturnConsumedCapacity`
  - Response syntax
- `PutItem`
  - Request syntax
  - Supported parameters:
    - `Item`
    - `TableName`
    - `ConditionalOperator`
    - `ConditionExpression`

    - `Expected`
    - `ExpressionAttributeNames`
    - `ExpressionAttributeValues`
    - `ReturnConsumedCapacity`
    - `ReturnItemCollectionMetrics`
    - `ReturnValues`
  - Response syntax
- `DeleteItem`
  - Request syntax
  - Supported parameters:
    - `Key`
    - `TableName`
    - `ConditionalOperator`
    - `ConditionExpression`
    - `Expected`
    - `ExpressionAttributeNames`
    - `ExpressionAttributeValues`
    - `ReturnConsumedCapacity`
    - `ReturnItemCollectionMetrics`
    - `ReturnValues`
  - Response syntax
- `UpdateItem`
  - Request syntax
  - Supported parameters:
    - `Key`
    - `TableName`
    - `AttributeUpdates`
    - `ConditionalOperator`
    - `ConditionExpression`
    - `Expected`
    - `ExpressionAttributeNames`
    - `ExpressionAttributeValues`
    - `ReturnConsumedCapacity`
    - `ReturnItemCollectionMetrics`
    - `ReturnValues`
    - `UpdateExpression`
  - Response syntax

The following is a `Task` state that retrieves a message from Amazon DynamoDB.

```
"Read Next Message from DynamoDB": {
  "Type": "Task",
  "Resource": "arn:aws:states:::dynamodb:getItem",
  "Parameters": {
    "TableName": "TransferDataRecords-DDBTable-3I41R5L5EAGT",
    "Key": {
      "MessageId": {"S.$": "$.List[0]"}
    }
  },
  "ResultPath": "$.DynamoDB",
  "Next": "Send Message to SQS"
},
```

> **Note**
> You cannot pass a map or list to DynamoDB inside a map.

To see this state in a working example, see the Transfer Data Records (Lambda, DynamoDB, Amazon SQS) (p. 174) sample project.

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Manage Amazon ECS or Fargate Tasks with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information, see:

- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)

Supported Amazon ECS APIs and syntax:

> **Note**
> Parameters in Step Functions are expressed in `PascalCase`, even when the native service API is `camelCase`.

- `RunTask` starts a new task using the specified task definition.
  - Request syntax
  - Supported parameters:
    - `Cluster`
    - `Group`
    - `LaunchType`
    - `NetworkConfiguration`
    - `Overrides`
    - `PlacementConstraints`
    - `PlacementStrategy`
    - `PlatformVersion`
    - `TaskDefinition`
  - Response syntax

> **Note**
> For the `Overrides` parameter, Step Functions does not support `executionRoleArn` or `taskRoleArn` as `ContainerOverrides`.

Passing Data to an Amazon ECS Task

Step Functions can control certain AWS services directly from the Amazon States Language. For more information, see:

- Service Integrations (p. 138)

- Pass Parameters to a Service API (p. 144)

You can use `overrides` to override the default command for a container, and pass input to your Amazon ECS tasks. See `ContainerOverride`. In the example, we have used JsonPath to pass values to the Task from the input to the Task state.

The following includes a `Task` state that runs an Amazon ECS task and waits for it to complete.

```
{
 "StartAt": "Run an ECS Task and wait for it to complete",
 "States": {
   "Run an ECS Task and wait for it to complete": {
     "Type": "Task",
     "Resource": "arn:aws:states:::ecs:runTask.sync",
     "Parameters": {
                "Cluster": "cluster-arn",
                "TaskDefinition": "job-id",
                "Overrides": {
                    "ContainerOverrides": [
                        {
                            "Name": "container-name",
                            "Command.$": "$.commands"
                        }
                    ]
                }
            },
     "End": true
    }
  }
}
```

The `"Command.$": "$.commands"` line in `ContainerOverrides` passes the commands from the state input to the container.

For the previous example, if the input to the execution is the following.

```
{
  "commands": [
    "test command 1",
    "test command 2",
    "test command 3"
  ]
}
```

The following includes a `Task` state that runs an Amazon ECS task, and then waits for the task token to be returned. See Wait for a Callback with the Task Token (p. 140).

```
{
   "StartAt":"Manage ECS task",
   "States":{
      "Manage ECS task":{
         "Type":"Task",
         "Resource":"arn:aws:states:::ecs:runTask.waitForTaskToken",
         "Parameters":{
            "LaunchType":"FARGATE",
            "Cluster":"cluster-arn",
            "TaskDefinition":"job-id",
            "Overrides":{
               "ContainerOverrides":[
                  {
                     "Name":"container-name",
                     "Environment":[
                        {
                           "Name":"TASK_TOKEN_ENV_VARIABLE",
                           "Value.$":"$$.Task.Token"
                        }
                     ]
                  }
               ]
            }
         },
         "End":true
      }
   }
}
```

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Call Amazon SNS with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information, see:

- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)

Supported APIs:

> **Note**
> There is a limit on the maximum input or result data size for a task in Step Functions. This limits you to 32,768 characters of data when you send to, or receive data from, another service. See Limits Related to State Machine Executions (p. 207).

- `Publish`
  - Request syntax
  - Supported Parameters
    - `Message`
    - `MessageAttributes`
    - `MessageStructure`
    - `PhoneNumber`
    - `Subject`
    - `TargetArn`
    - `TopicArn`
  - Response syntax

The following includes a `Task` state that publishes an Amazon SNS topic.

```
{
  "StartAt": "Publish to SNS",
  "States": {
    "Publish to SNS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "TopicArn": "arn:aws:sns:us-east-1:123456789012:myTopic",
        "Message.$": "$.input.message",
       "MessageAttributes": {
         "my attribute no 1": {
           "DataType": "String",
           "StringValue": "value of my attribute no 1"
         },
         "my attribute no 2": {
           "DataType": "String",
           "StringValue": "value of my attribute no 2"
         }
       }
     },
     "End": true
    }
  }
}
```

The following includes a `Task` state that publishes an Amazon SNS topic, and then waits for the task token to be returned. See Wait for a Callback with the Task Token (p. 140).

```
{
   "StartAt":"Send message to SNS",
   "States":{
      "Send message to SNS":{
         "Type":"Task",
         "Resource":"arn:aws:states:::sns:publish.waitForTaskToken",
         "Parameters":{
            "TopicArn":"arn:aws:sns:us-east-1:123456789012:myTopic",
            "Message":{
               "Input.$":"$",
               "TaskToken.$":"$$.Task.Token"
            }
         },
         "End":true
      }
   }
}
```

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Call Amazon SQS with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information, see:

- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)

Supported APIs:

> **Note**
> There is a limit on the maximum input or result data size for a task in Step Functions. This limits you to 32,768 characters of data when you send to, or receive data from, another service. See Limits Related to State Machine Executions (p. 207).

- `SendMessage`

  Supported parameters:

  - `DelaySeconds`
  - `MessageAttribute`

  - MessageBody
  - MessageDeduplicationId
  - MessageGroupId
  - QueueUrl
- Response syntax

The following includes a `Task` state that sends an Amazon SQS message.

```
{
 "StartAt": "Send to SQS",
 "States": {
   "Send to SQS": {
     "Type": "Task",
     "Resource": "arn:aws:states:::sqs:sendMessage",
     "Parameters": {
       "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/myQueue",
       "MessageBody.$": "$.input.message",
       "MessageAttributes": {
         "my attribute no 1": {
           "DataType": "String",
           "StringValue": "attribute1"
         },
         "my attribute no 2": {
           "DataType": "String",
           "StringValue": "attribute2"
         }
       }
     },
     "End": true
    }
  }
}
```

The following includes a `Task` state that publishes to an Amazon SQS queue, and then waits for the task token to be returned. See Wait for a Callback with the Task Token (p. 140).

```
{
   "StartAt":"Send message to SQS",
   "States":{
      "Send message to SQS":{
         "Type":"Task",
         "Resource":"arn:aws:states:::sqs:sendMessage.waitForTaskToken",
         "Parameters":{
            "QueueUrl":"https://sqs.us-east-1.amazonaws.com/123456789012/myQueue",
            "MessageBody":{
               "Input.$":"$",
               "TaskToken.$":"$$.Task.Token"
            }
         },
         "End":true
      }
   }
}
```

To learn more about receiving messages in Amazon SQS, see Receive and Delete Your Message in the *Amazon Simple Queue Service Developer Guide*.

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Manage AWS Glue Jobs with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information, see:

- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)

Supported APIs:

- `StartJobRun`
- Supported Parameters:
  - `JobName`
  - `JobRunId`
  - `Arguments`
  - `AllocatedCapacity`
  - `Timeout`
  - `SecurityConfiguration`
  - `NotificationProperty`

The following includes a `Task` state that starts an AWS Glue job.

```
"Glue StartJobRun": {
      "Type": "Task",
      "Resource": "arn:aws:states:::glue:startJobRun.sync",
      "Parameters": {
        "JobName": "GlueJob-JTrRO5l98qMG"
      },
      "Next": "ValidateOutput"
    },
```

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Manage Amazon SageMaker with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information, see:

- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)

Supported Amazon SageMaker APIs and syntax:

- `CreateEndpoint`
  - Request syntax
  - Supported parameters:
    - `EndpointConfigName`
    - `EndpointName`
    - `Tags`
  - Response syntax

- `CreateEndpointConfig`
  - Request syntax
  - Supported parameters:
    - `EndpointConfigName`
    - `KmsKeyId`
    - `ProductionVariants`
    - `Tags`
  - Response syntax
- `CreateHyperParameterTuningJob`
  - Request syntax
  - Supported parameters:
    - `HyperParameterTuningJobConfig`
    - `HyperParameterTuningJobName`
    - `Tags`
    - `TrainingJobDefinition`
    - `WarmStartConfig`
  - Response syntax
- `CreateLabelingJob`
  - Request syntax
  - Supported parameters:
    - `HumanTaskConfig`
    - `InputConfig`
    - `LabelAttributeName`
    - `LabelCategoryConfigS3Uri`
    - `LabelingJobAlgorithmsConfig`
    - `LabelingJobName`
    - `OutputConfig`
    - `RoleArn`
    - `StoppingConditions`
    - `Tags`
  - Response syntax
- `CreateModel`
  - Request syntax
  - Supported parameters:
    - `Containers`
    - `EnableNetworkIsolation`
    - `ExecutionRoleArn`
    - `ModelName`
    - `PrimaryContainer`
    - `Tags`
    - `VpcConfig`
- `CreateTrainingJob`
  - Request syntax
  - Supported parameters:
    - `AlgorithmSpecification`
    - `HyperParameters`

    - `InputDataConfig`
    - `OutputDataConfig`
    - `ResourceConfig`
    - `RoleArn`
    - `StoppingCondition`
    - `Tags`
    - `TrainingJobName`
    - `VpcConfig`
  - Response syntax
- `CreateTransformJob`

  Note
  AWS Step Functions will not automatically create a policy for `CreateTransformJob`. You must attach an inline policy to the created role. For more information, see this example IAM policy: `CreateTrainingJob` (p. 241).

  - Request syntax
  - Supported parameters:
    - `BatchStrategy`
    - `Environment`
    - `MaxConcurrentTransforms`
    - `MaxPayloadInMB`
    - `ModelName`
    - `Tags`
    - `TransformInput`
    - `TransformJobName`
    - `TransformOutput`
    - `TransformResources`
  - Response syntax
- `UpdateEndpoint`
  - Request syntax
  - Supported parameters:
    - `EndpointConfigName`
    - `EndpointName`
  - Response syntax

Amazon SageMaker Transform Job Example

The following includes a `Task` state that creates an Amazon SageMaker transform job, specifying the Amazon S3 location for `DataSource` and `TransformOutput`.

```
{
"SageMaker CreateTransformJob": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sagemaker:createTransformJob.sync",
  "Parameters": {
    "ModelName": "SageMakerCreateTransformJobModel-9iFBKsYti9vr",
    "TransformInput": {
      "CompressionType": "None",
      "ContentType": "text/csv",
      "DataSource": {
        "S3DataSource": {
          "S3DataType": "S3Prefix",
          "S3Uri": "s3://my-s3bucket-example-1/TransformJobDataInput.txt"
        }
      }
    },
    "TransformOutput": {
      "S3OutputPath": "s3://my-s3bucket-example-1/TransformJobOutputPath"
    },
    "TransformResources": {
      "InstanceCount": 1,
      "InstanceType": "ml.m4.xlarge"
    },
    "TransformJobName": "sfn-binary-classification-prediction"
  },
  "Next": "ValidateOutput"
},
```

Amazon SageMaker Training Job Example

The following includes a `Task` state that creates an Amazon SageMaker training job.

```
{
   "SageMaker CreateTrainingJob":{
      "Type":"Task",
      "Resource":"arn:aws:states:::sagemaker:createTrainingJob.sync",
      "Parameters":{
         "TrainingJobName":"search-model",
         "ResourceConfig":{
            "InstanceCount":4,
            "InstanceType":"ml.c4.8xlarge",
            "VolumeSizeInGB":20
         },
         "HyperParameters":{
            "mode":"batch_skipgram",
            "epochs":"5",
            "min_count":"5",
            "sampling_threshold":"0.0001",
            "learning_rate":"0.025",
            "window_size":"5",
            "vector_dim":"300",
            "negative_samples":"5",
            "batch_size":"11"
         },
         "AlgorithmSpecification":{
            "TrainingImage":"...",
            "TrainingInputMode":"File"
         },
         "OutputDataConfig":{
            "S3OutputPath":"s3://bucket-name/doc-search/model"
         },
         "StoppingCondition":{
            "MaxRuntimeInSeconds":100000
         },
         "RoleArn":"arn:aws:iam::123456789012:role/docsearch-stepfunction-iam-role",
         "InputDataConfig":[
            {
               "ChannelName":"train",
               "DataSource":{
                  "S3DataSource":{
                     "S3DataType":"S3Prefix",
                     "S3Uri":"s3://bucket-name/doc-search/interim-data/training-data/",
                     "S3DataDistributionType":"FullyReplicated"
                  }
               }
            }
         ]
      },
      "Retry":[
         {
            "ErrorEquals":[
               "SageMaker.AmazonSageMakerException"
            ],
            "IntervalSeconds":1,
            "MaxAttempts":100,
            "BackoffRate":1.1
         },
         {
            "ErrorEquals":[
               "SageMaker.ResourceLimitExceededException"
            ],
            "IntervalSeconds":60,
            "MaxAttempts":5000,
            "BackoffRate":1
         },
         {
            "ErrorEquals":[
               "States.Timeout"
            ],
            "IntervalSeconds":1,
            "MaxAttempts":5,
            "BackoffRate":1
         }
      ],
      "Catch":[
         {
            "ErrorEquals":[
               "States.ALL"
            ],
            "ResultPath":"$.cause",
            "Next":"Sagemaker Training Job Error"
         }
      ],
      "Next":"Delete Interim Data Job"
   }
}
```

Amazon SageMaker Labeling Job Example

The following includes a `Task` state that creates an Amazon SageMaker labeling job.

```
{
  "StartAt": "SageMaker CreaateLabelingJob",
  "TimeoutSeconds": 3600,
  "States": {
    "SageMaker CreaateLabelingJob": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sagemaker:createLabelingJob.sync",
      "Parameters": {
        "HumanTaskConfig": {
          "AnnotationConsolidationConfig": {
            "AnnotationConsolidationLambdaArn": "arn:aws:lambda:us-
west-2:123456789012:function:ACS-TextMultiClass"
          },
          "NumberOfHumanWorkersPerDataObject": 1,
          "PreHumanTaskLambdaArn": "arn:aws:lambda:us-west-2:123456789012:function:PRE-
TextMultiClass",
     "TaskDescription": "Classify the following text",
     "TaskKeywords": [
      "tc",
      "Labeling"
     ],
     "TaskTimeLimitInSeconds": 300,
     "TaskTitle": "Classify short bits of text",
     "UiConfig": {
      "UiTemplateS3Uri": "s3://s3bucket-example/TextClassification.template"
     },
     "WorkteamArn": "arn:aws:sagemaker:us-west-2:123456789012:workteam/private-crowd/
ExampleTesting"
    },
    "InputConfig": {
     "DataAttributes": {
      "ContentClassifiers": [
       "FreeOfPersonallyIdentifiableInformation",
       "FreeOfAdultContent"
      ]
     },
     "DataSource": {
      "S3DataSource": {
       "ManifestS3Uri": "s3://s3bucket-example/manifest.json"
      }
     }
    },
    "LabelAttributeName": "Categories",
    "LabelCategoryConfigS3Uri": "s3://s3bucket-example/labelcategories.json",
    "LabelingJobName": "example-job-name",
    "OutputConfig": {
     "S3OutputPath": "s3://s3bucket-example/output"
    },
    "RoleArn": "arn:aws:iam::123456789012:role/service-role/AmazonSageMaker-
ExecutionRole",
    "StoppingConditions": {
     "MaxHumanLabeledObjectCount": 10000,
     "MaxPercentageOfInputDatasetLabeled": 100
    }
   },
   "Next": "ValidateOutput"
  },
  "ValidateOutput": {
   "Type": "Choice",
   "Choices": [
    {
     "Not": {
      "Variable": "$.LabelingJobArn",
      "StringEquals": ""
     },
     "Next": "Succeed"
    }
   ],
   "Default": "Fail"
  },
  "Succeed": {
   "Type": "Succeed"
  },
  "Fail": {
   "Type": "Fail",
   "Error": "InvalidOutput",
   "Cause": "Output is not what was expected. This could be due to a service outage or a
 misconfigured service integration."
  }
 }
}
```

Manage AWS Step Functions Executions as an Integrated Service

Step Functions integrates with its own API as a service integration. This allows Step Functions to start a new execution of a state machine directly from the task state of a running execution. Use nested workflow executions (p. 130) to reduce the complexity of your main workflows and to reuse common processes when building new workflows.

For more information, see:

- Start Executions From a Task (p. 130)
- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)

Supported Step Functions APIs and syntax:

- `StartExecution`
  - Request Syntax
  - Supported Parameters
    - `input`
    - `name`
    - `stateMachineArn`
  - Response syntax

The following includes a `Task` state that starts an execution of another state machine and waits for it to complete.

```
{
   "Type":"Task",
   "Resource":"arn:aws:states:::states:startExecution.sync",
   "Parameters":{
      "Input":{
        "Comment": "Hello world!"
       },
      "StateMachineArn":"arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
      "Name":"ExecutionName"
   },
   "End":true
}
```

The following includes a `Task` state that starts an execution of another state machine.

```
{
   "Type":"Task",
   "Resource":"arn:aws:states:::states:startExecution",
   "Parameters":{
      "Input":{
        "Comment": "Hello world!"
       },
      "StateMachineArn":"arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
      "Name":"ExecutionName"
   },
   "End":true
}
```

The following includes a `Task` state that implements the callback (p. 140) service integration pattern.

```
{
   "Type":"Task",
   "Resource":"arn:aws:states:::states:startExecution.waitForTaskToken",
   "Parameters":{
      "Input":{
        "Comment": "Hello world!",
        "token.$": "$$.Task.Token"
       },
      "StateMachineArn":"arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
      "Name":"ExecutionName"
   },
   "End":true
}
```

To associate a nested workflow execution with the parent execution that started it, pass a specially named parameter that includes the execution Id pulled from the context object (p. 127). When starting a nested execution, use a parameter named `AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID`. Pass the execution ID by appending `.$` to the parameter name, and referencing the ID in the context object with `$$.Execution.Id`. For more information, see Accessing the Context Object (p. 128).

```
{
   "Type":"Task",
   "Resource":"arn:aws:states:::states:startExecution.sync",
   "Parameters":{
      "Input":{
        "Comment": "Hello world!",
        "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
       },
      "StateMachineArn":"arn:aws:states:us-east-1:123456789012:stateMachine:HelloWorld",
      "Name":"ExecutionName"
   },
   "End":true
}
```

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Read Consistency

State machine updates in AWS Step Functions are eventually consistent. All `StartExecution` calls within a few seconds will use the updated definition and `roleArn` (the Amazon Resource Name for the IAM role). Executions started immediately after calling `UpdateStateMachine` might use the previous state machine definition and `roleArn`.

For more information, see:

- `UpdateStateMachine` in the *AWS Step Functions API Reference*
- Update a state machine (p. 15) in Getting Started (p. 13)

Templates

In the Step Functions console, you can choose one of the following state machine templates to automatically fill the **Code** pane. Each of the templates is fully functional and you can use any blueprint as the template for your own state machine.

> **Note**
> Choosing any of the templates overwrites the contents of the **Code** pane.

- **Hello world** – A state machine with a `Pass` state.
- **Wait state** – A state machine that demonstrates different ways of injecting a `Wait` state into a running state machine:
  - By waiting for a number of seconds.
  - By waiting for an absolute time (timestamp).
  - By specifying the `Wait` state's definition.
  - By using the state's input data.
- **Retry failure** – A state machine that retries a task after the task fails. This blueprint demonstrates how to handle multiple retries and various failure types.
- **Parallel** – A state machine that demonstrates how to execute two branches at the same time.
- **Catch failure** – A state machine that performs a different task after its primary task fails. This blueprint demonstrates how to call different tasks depending on the failure type.
- **Choice state** – A state machine that makes a choice: It either runs a `Task` state from a set of `Task` states or runs a `Fail` state after the initial state is complete.

Tagging

AWS Step Functions supports tagging of state machines and activities. This can help you track and manage the costs associated with your resources, and provide better security in your IAM policies. Tagging AWS Step Functions resources allows them to be managed by AWS Resource Groups. For more information on Resource Groups, see the AWS Resource Groups User Guide.

To review the restrictions related to resource tagging, see Restrictions Related to Tagging (p. 210).

Topics

Tagging for Cost Allocation

To organize and identify your Step Functions resources for cost allocation, you can add metadata *tags* that identify the purpose of a state machine or activity. This is especially useful when you have many resources. You can use cost allocation tags to organize your AWS bill to reflect your own cost structure. To do this, sign up to get your AWS account bill to include the tag keys and values. For more information, see Setting Up a Monthly Cost Allocation Report in the *AWS Billing and Cost Management User Guide*.

For instance, you could add tags that represent the cost center and purpose of your Step Functions resources:

Resource	Key	Value
`StateMachine1`	`Cost Center`	`34567`
	`Application`	`Image processing`

Resource	Key	Value
`StateMachine2`	`Cost Center`	`34567`
	`Application`	`Rekognition processing`
`Activity1`	`Cost Center`	`12345`
	`Application`	`Legacy database`

This tagging scheme allows you to group two state machines performing related tasks in the same cost center, while tagging an unrelated activity with a different cost allocation tag.

Tagging for Security

IAM supports controlling access to resources based on tags. To control access based on tags, provide information about your resource tags in the condition element of an IAM policy.

For instance, you could restrict access to all Step Functions resources that include a tag with the key `environment` and the value `production`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "states:TagResource",
                "states:DeleteActivity",
                "states:DeleteStateMachine",
                "states:StopExecution"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {"aws:ResourceTag/environment": "production"}
            }
        }
    ]
}
```

For more information, see Controlling Access Using Tags in the IAM User Guide.

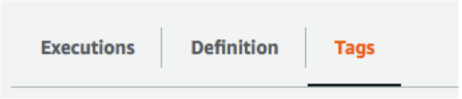
Viewing and Managing Tags in the Step Functions Console

AWS Step Functions allows you to view and manage the tags for your state machines in the Step Functions console. From the **Details** page of a state machine, select **Tags**. Here, you can view the existing tags associated with your state machine.

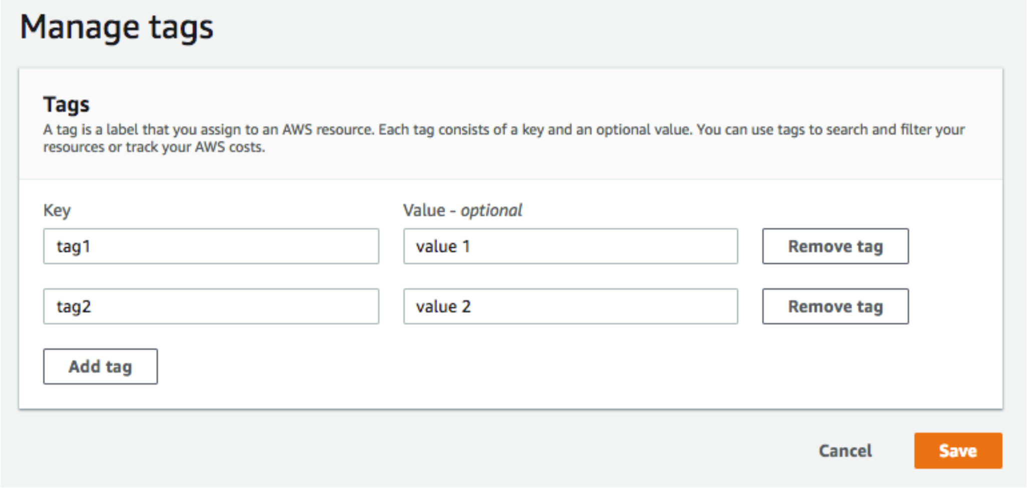
> **Note**
> To manage tags for activites, see Manage Tags With Step Functions API Actions. (p. 166).

To add or delete tags that are associated with your state machine, select the **Manage Tags** button.

1. Browse to the details page of a state machine.
2. Select **Tags**, next to **Executions** and **Definition**.



3. Choose **Manage tags**.
4. - To modify existing tags, edit the **Key** and **Value**.
   - To remove existing tags, select **Remove tag**.
   - To add a new tag, select **Add tag** and enter a **Key** and **Value**.



Manage tags

Tags

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

Key	Value - *optional*
tag1	value 1
tag2	value 2

5. Choose **Save**.

Manage Tags With Step Functions API Actions.

To manage tags using the Step Functions API, use the following API actions:

- `ListTagsForResource`
- `TagResource`
- `UntagResource`

AWS Step Functions Data Science SDK for Python

The AWS Step Functions Data Science SDK is an open source library that allows data scientists to easily create workflows that process and publish machine learning models using Amazon SageMaker and Step Functions. You can create multi-step machine learning workflows in Python that orchestrate AWS infrastructure at scale, without having to provision and integrate the AWS services separately.

The AWS Step Functions Data Science SDK provides a Python API that can create and invoke Step Functions workflows. You can manage and execute these workflows directly in Python, and in Jupyter notebooks.

In addition to creating production-ready workflows directly in Python, the AWS Step Functions Data Science SDK allows you to copy that workflow, experiment with new options, and then put the refined workflow in production.

For more information on the AWS Step Functions Data Science SDK, see the following.

- Project on Github
- SDK Documentation
- The following Example Notebooks, which are available in Jupyter notebook instances in the Amazon SageMaker console and the related GitHub project.
  - `hello_world_workflow.ipynb`
  - `machine_learning_workflow_abalone.ipynb`
  - `training_pipeline_pytorch_mnist.ipynb`

Sample Projects

In the AWS Step Functions console, you can choose one of the following state machine sample projects to automatically create the state machine **Code**, **Visual Workflow**, and all related AWS resources for the project.

Each of the sample projects provisions a fully functional state machine, and creates the related resources for it to run. When you create a sample project, Step Functions uses AWS CloudFormation to create the related resources referenced by the state machine.

Topics

Manage a Batch Job (AWS Batch, Amazon SNS)

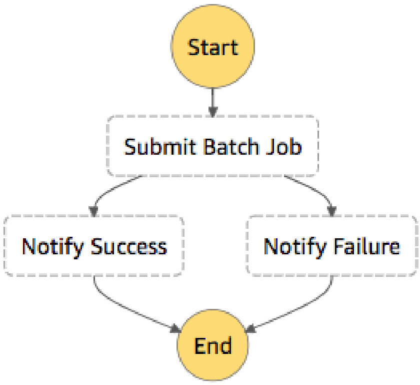
This sample project demonstrates how to submit an AWS Batch job, and then send an Amazon SNS notification based on whether that job succeeds or fails. Deploying this sample project creates a Step Functions state machine, an AWS Batch job, and an Amazon SNS topic.

In this project, Step Functions uses a state machine to call the AWS Batch job synchronously. It then waits for the job to succeed or fail, and it sends an Amazon SNS topic with a message about whether the job succeeded or failed.

To create the Manage a Batch Job state machine and provision all resources

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Manage a Batch Job**.

   The state machine **Code** and **Visual Workflow** are displayed.



3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

   - An AWS Batch job
   - An Amazon SNS topic

4. Choose **Deploy Resources**.

   > **Note**
   > It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

Start a New Execution

1. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   > **Note**
   > Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

3. Optionally, you can go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.
4. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.

Example State Machine Code

The state machine in this sample project integrates with AWS Batch and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls AWS Batch and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the `Resource` field, and by passing `Parameters` to the service API.

For more information about how AWS Step Functions can control other AWS services, see AWS Service Integrations (p. 138).

```
{
  "Comment": "An example of the Amazon States Language for notification on an AWS Batch job
 completion",
  "StartAt": "Submit Batch Job",
  "TimeoutSeconds": 3600,
  "States": {
    "Submit Batch Job": {
      "Type": "Task",
      "Resource": "arn:aws:states:::batch:submitJob.sync",
      "Parameters": {
        "JobName": "BatchJobNotification",
        "JobQueue": "arn:aws:batch:us-east-1:123456789012:job-queue/
BatchJobQueue-7049d367474b4dd",
        "JobDefinition": "arn:aws:batch:us-east-1:123456789012:job-definition/
BatchJobDefinition-74d55ec34c4643c:1"
      },
      "Next": "Notify Success",
      "Catch": [
          {
            "ErrorEquals": [ "States.ALL" ],
            "Next": "Notify Failure"
          }
      ]
    },
    "Notify Success": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "Message": "Batch job submitted through Step Functions succeeded",
        "TopicArn": "arn:aws:sns:us-east-1:123456789012:batchjobnotificatiointemplate-
SNSTopic-1J757CVBQ2KHM"
      },
      "End": true
    },
    "Notify Failure": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "Message": "Batch job submitted through Step Functions failed",
        "TopicArn": "arn:aws:sns:us-east-1:123456789012:batchjobnotificatiointemplate-
SNSTopic-1J757CVBQ2KHM"
      },
      "End": true
    }
  }
}
```

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "sns:Publish"
            ],
            "Resource": [
                "arn:aws:sns:ap-northeast-1:123456789012:ManageBatchJob-SNSTopic-
JHLYYG7AZPZI"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "batch:SubmitJob",
                "batch:DescribeJobs",
                "batch:TerminateJob"
            ],
            "Resource": "*",
            "Effect": "Allow"
        },
        {
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:ap-northeast-1:123456789012:rule/
StepFunctionsGetEventsForBatchJobsRule"
            ],
            "Effect": "Allow"
        }
    ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Manage a Container Task (Amazon ECS, Amazon SNS)

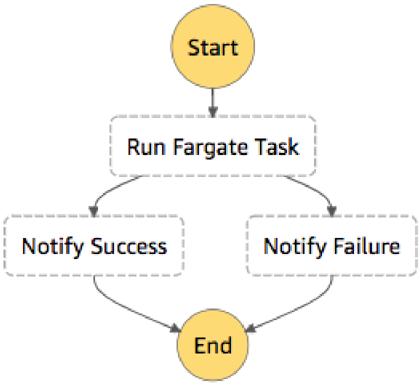
This sample project demonstrates how to run an AWS Fargate task, and then send an Amazon SNS notification based on whether that job succeeds or fails. Deploying this sample project will create a Step Functions state machine, a Fargate cluster, and an Amazon SNS topic.

In this project, Step Functions uses a state machine to call the Fargate task synchronously. It then waits for the task to succeed or fail, and it sends an Amazon SNS topic with a message about whether the job succeeded or failed.

To create the Manage a container task state machine and provision all resources

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Manage a container task**.

   The state machine **Code** and **Visual Workflow** are displayed.



3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project the resources include:

   - A Fargate cluster
   - An Amazon SNS topic
4. Choose **Deploy Resources**.

   Note
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

Start a New Execution

1. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   Note
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. Optionally, you can go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.
4. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.

Example State Machine Code

The state machine in this sample project integrates with AWS Fargate and Amazon SNS by passing parameters directly to those resources. Browse through this example state machine to see how Step Functions uses a state machine to call the Fargate task synchronously, waits for the task to succeed or fail, and sends an Amazon SNS topic with a message about whether the job succeeded or failed.

For more information about how AWS Step Functions can control other AWS services, see AWS Service Integrations (p. 138).

```
{
  "Comment": "An example of the Amazon States Language for notification on an AWS Fargate
 task completion",
  "StartAt": "Run Fargate Task",
  "TimeoutSeconds": 3600,
  "States": {
    "Run Fargate Task": {
      "Type": "Task",
      "Resource": "arn:aws:states:::ecs:runTask.sync",
      "Parameters": {
        "LaunchType": "FARGATE",
        "Cluster": "arn:aws:ecs:ap-northeast-1:123456789012:cluster/
FargateTaskNotification-ECSCluster-VHLR20IF9IMP",
        "TaskDefinition": "arn:aws:ecs:ap-northeast-1:123456789012:task-definition/
FargateTaskNotification-ECSTaskDefinition-13YOJT8Z2LY5Q:1",
        "NetworkConfiguration": {
          "AwsvpcConfiguration": {
            "Subnets": [
              "subnet-07e1ad3abcfce6758",
              "subnet-04782e7f34ae3efdb"
            ],
            "AssignPublicIp": "ENABLED"
          }
        }
      },
      "Next": "Notify Success",
      "Catch": [
          {
            "ErrorEquals": [ "States.ALL" ],
            "Next": "Notify Failure"
          }
      ]
    },
    "Notify Success": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "Message": "AWS Fargate Task started by Step Functions succeeded",
        "TopicArn": "arn:aws:sns:ap-northeast-1:123456789012:FargateTaskNotification-
SNSTopic-1XYW5YD5V0M7C"
      },
      "End": true
    },
    "Notify Failure": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "Message": "AWS Fargate Task started by Step Functions failed",
        "TopicArn": "arn:aws:sns:ap-northeast-1:123456789012:FargateTaskNotification-
SNSTopic-1XYW5YD5V0M7C"
      },
      "End": true
    }
  }
}
```

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. It's a best practice to include only those permissions that are necessary in your IAM policies.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "sns:Publish"
            ],
            "Resource": [
                "arn:aws:sns:ap-northeast-1:123456789012:FargateTaskNotification-
SNSTopic-1XYW5YD5V0M7C"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "ecs:RunTask"
            ],
            "Resource": [
                "arn:aws:ecs:ap-northeast-1:123456789012:task-definition/
FargateTaskNotification-ECSTaskDefinition-13YOJT8Z2LY5Q:1"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "ecs:StopTask",
                "ecs:DescribeTasks"
            ],
            "Resource": "*",
            "Effect": "Allow"
        },
        {
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:ap-northeast-1:123456789012:rule/
StepFunctionsGetEventsForECSTaskRule"
            ],
            "Effect": "Allow"
        }
    ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Transfer Data Records (Lambda, DynamoDB, Amazon SQS)

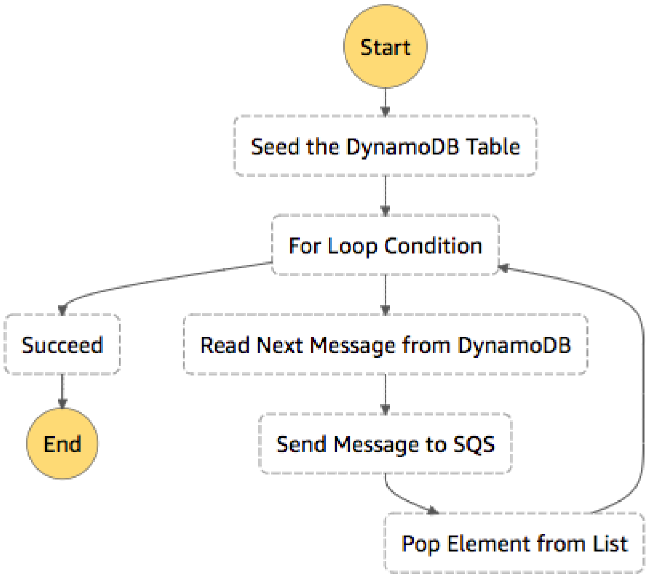
This sample project demonstrates how to read values from an Amazon DynamoDB table and send them to Amazon SQS using AWS Step Functions. Deploying this sample project will create a Step Functions state machine, a DynamoDB table, an AWS Lambda function, and an Amazon SQS topic.

In this project, Step Functions uses the Lambda function to populate the DynamoDB table, uses a `for` loop to read each of the entries, and then sends each entry to Amazon SQS.

To create the Transfer Data Records state machine and provision all resources

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Transfer Data Records**.

   The state machine **Code** and **Visual Workflow** are displayed.

   Note
   The **Code** section in this state machine references the AWS resources that will be created for this sample project.
3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project the resources include:

   - A Lambda function for seeding the DynamoDB table
   - An Amazon SQS queue
   - A DynamoDB table
4. Choose **Deploy Resources**.

   Note
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

Start a New Execution

1. On the **New execution** page, enter an execution name (optional) and choose **Start Execution.**
2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   Note
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch.

To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

3. Optionally, you can go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.
4. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**

Example State Machine Code

The state machine in this sample project integrates with DynamoDB and Amazon SQS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls DynamoDB and Amazon SQS by connecting to the Amazon Resource Name (ARN) in the `Resource` field, and by passing `Parameters` to the service API.

For more information about how AWS Step Functions can control other AWS services, see AWS Service Integrations (p. 138).

```
{
  "Comment" : "An example of the Amazon States Language for reading messages from a
 DynamoDB table and sending them to SQS",
  "StartAt": "Seed the DynamoDB Table",
  "TimeoutSeconds": 3600,
  "States": {
    "Seed the DynamoDB Table": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:sqsconnector-
SeedingFunction-T3U43VYDU5OQ",
      "ResultPath": "$.List",
      "Next": "For Loop Condition"
    },
    "For Loop Condition": {
      "Type": "Choice",
      "Choices": [
        {
          "Not": {
            "Variable": "$.List[0]",
            "StringEquals": "DONE"
          },
          "Next": "Read Next Message from DynamoDB"
        }
      ],
      "Default": "Succeed"
    },
    "Read Next Message from DynamoDB": {
      "Type": "Task",
      "Resource": "arn:aws:states:::dynamodb:getItem",
      "Parameters": {
        "TableName": "sqsconnector-DDBTable-1CAFOJWP8QD6I",
        "Key": {
          "MessageId": {"S.$": "$.List[0]"}
        }
      },
      "ResultPath": "$.DynamoDB",
      "Next": "Send Message to SQS"
    },
    "Send Message to SQS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage",
      "Parameters": {
        "MessageBody.$": "$.DynamoDB.Item.Message.S",
        "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/sqsconnector-
SQSQueue-QVGQBW134PWK"
      },
      "ResultPath": "$.SQS",
      "Next": "Pop Element from List"
    },
    "Pop Element from List": {
      "Type": "Pass",
      "Parameters": {
        "List.$": "$.List[1:]"
      },
      "Next": "For Loop Condition"
    },
    "Succeed": {
      "Type": "Succeed"
    }
  }
}
```

For more information about passing parameters and managing results, see:

- Pass Parameters to a Service API (p. 144)
- ResultPath (p. 120)

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. It's a best practice to include only those permissions that are necessary in your IAM policies.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:GetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:ap-northeast-1:123456789012:table/TransferDataRecords-
DDBTable-3I41R5L5EAGT"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "sqs:SendMessage"
            ],
            "Resource": [
                "arn:aws:sqs:ap-northeast-1:123456789012:TransferDataRecords-SQSQueue-
BKWXTS09LIW1"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "lambda:invokeFunction"
            ],
            "Resource": [
                "arn:aws:lambda:ap-northeast-1:123456789012:function:TransferDataRecords-
SeedingFunction-VN4KY2TPAZSR"
            ],
            "Effect": "Allow"
        }
    ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Poll for Job Status (Lambda, AWS Batch)

This sample project creates an AWS Batch job poller. It implements an AWS Step Functions state machine that uses AWS Lambda to create a `Wait` state loop that checks on an AWS Batch job.

This sample project creates and configures all resources so that your Step Functions workflow will submit an AWS Batch job, and will wait for that job to complete before ending successfully.

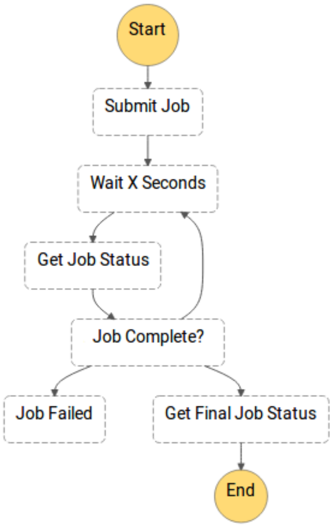
> **Note**
> You can also implement this pattern without using a Lambda function. For information about controlling AWS Batch directly, see AWS Service Integrations (p. 138).

This sample project creates the state machine, two Lambda functions, and an AWS Batch queue, and configures the related IAM permissions.

To create the Job Poller state machine and provision all resources

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Job Status Poller**.

   The state machine **Code** and **Visual Workflow** are displayed.



   > **Note**
   > The **Code** section in this state machine references the AWS resources that will be created for this sample project.
3. Choose **Create Resources**.

   The **Create Project Resources** page is displayed, listing the resources that will be created. For this sample project the resources include:

   - A `SubmitJob` Lambda function

- A `CheckJob` Lambda function
- A `SampleJobQueue` Batch Job Queue

> **Note**
> It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Create Project Resources** page displays **Creating resources**, you can open the **Stack ID** link to see which resources are being provisioned.

When complete, the **New execution** page is displayed, with example input similar to this.

```
{
  "jobName": "my-job",
  "jobDefinition": "arn:aws:batch:us-east-2:123456789012:job-definition/
SampleJobDefinition-343f54b445d5312:1",
  "jobQueue": "arn:aws:batch:us-east-2:123456789012:job-queue/
SampleJobQueue-4d9d696031e1449",
  "wait_time": 60
}
```

Starting an Execution

After you create your state machine, you can start an execution.

To start a new execution

1. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   > **Note**
   > Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. Optionally, you can go to the newly created state machine on the Step Functions **Dashboard**, choose **New execution**, and then enter the input code using the name or Amazon Resource Name (ARN) of each newly created resource.

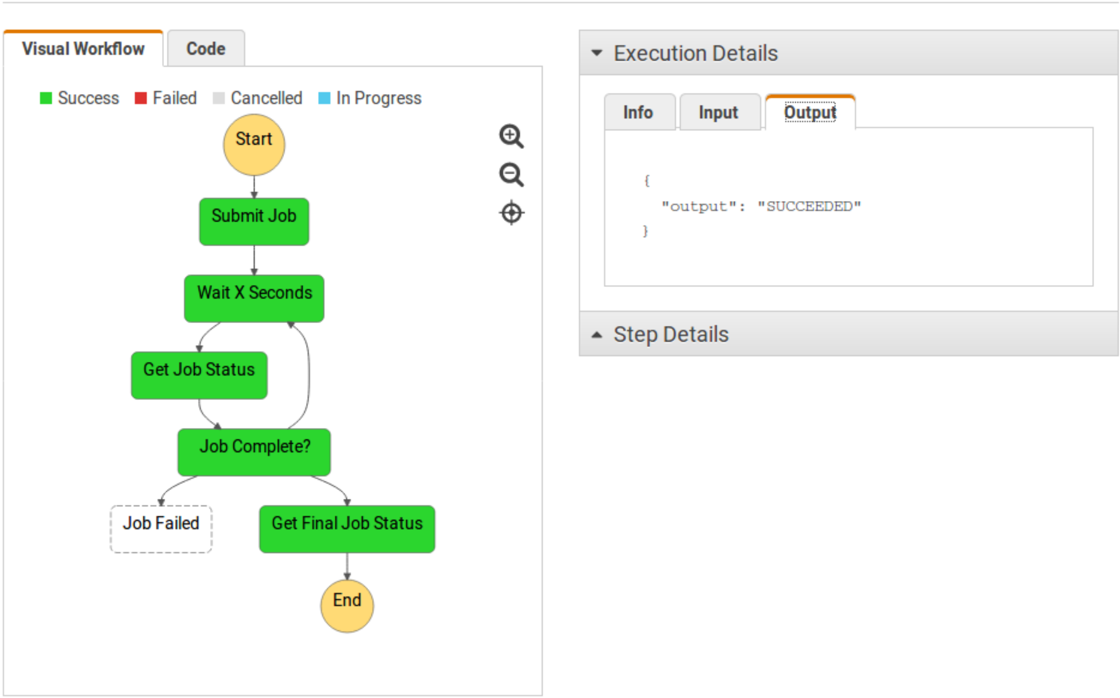
   For example, the input for the previous execution using only the resource names would be as follows.

   ```
   {
     "jobName": "my-job",
     "jobDefinition": "SampleJobDefinition-343f54b445d5312",
     "jobQueue": "SampleJobQueue-4d9d696031e1449",
     "wait_time": 60
   }
   ```

   > **Note**
   > `wait_time` instructs the `Wait` state to loop every 60 seconds.
4. Choose **Start Execution**.

   A new execution of your state machine starts, and a new page showing your running execution is displayed.

5. (Optional) In the **Execution Details** section, choose **Info** to view the **Execution Status** and the **Started** and **Closed** timestamps.
6. To view the changing status of your AWS Batch job and the looping results of your execution, choose **Output**.



```
{
  "output": "SUCCEEDED"
}
```

Task Timer

This sample project creates a task timer. It implements an AWS Step Functions state machine that implements a `Wait` state, and uses a Lambda function that sends an Amazon Simple Notification Service (Amazon SNS) notification. A `Wait` state is a state type that waits for a trigger to perform a single unit of work.

This sample project creates the state machine, a Lambda function, and an Amazon SNS topic, and configures the related AWS Identity and Access Management (IAM) permissions. For more information about the resources that are created with the **Task Timer** sample project, see:

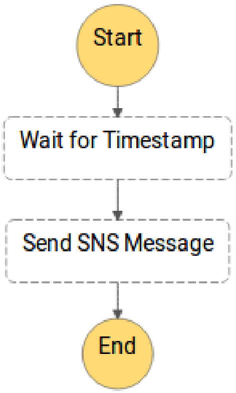
> **Note**
> This sample project implements an AWS Lambda function to send an Amazon Simple Notification Service (Amazon SNS) notification. You can also send an Amazon SNS notification directly from the Amazon States Language. See AWS Service Integrations (p. 138).

- AWS CloudFormation User Guide
- Amazon Simple Notification Service Developer Guide
- AWS Lambda Developer Guide
- IAM Getting Started Guide

To create the Task Timer state machine and provision all resources

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Task Timer**.

The state machine **Code** and **Visual Workflow** are displayed.



> **Note**
> The **Code** section in this state machine references the AWS resources that will be created for this sample project.

3. Choose **Create Sample Project**.

   The **Create Project Resources** page is displayed, listing the resources that will be created. For this sample project the resources include:

   - A SendToSNS Lambda function
   - A TaskTimerTopic Amazon SNS topic

   > **Note**
   > It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Create Project Resources** page displays **Creating resources**, you can open the **Stack ID:** link to see which resources are being provisioned.

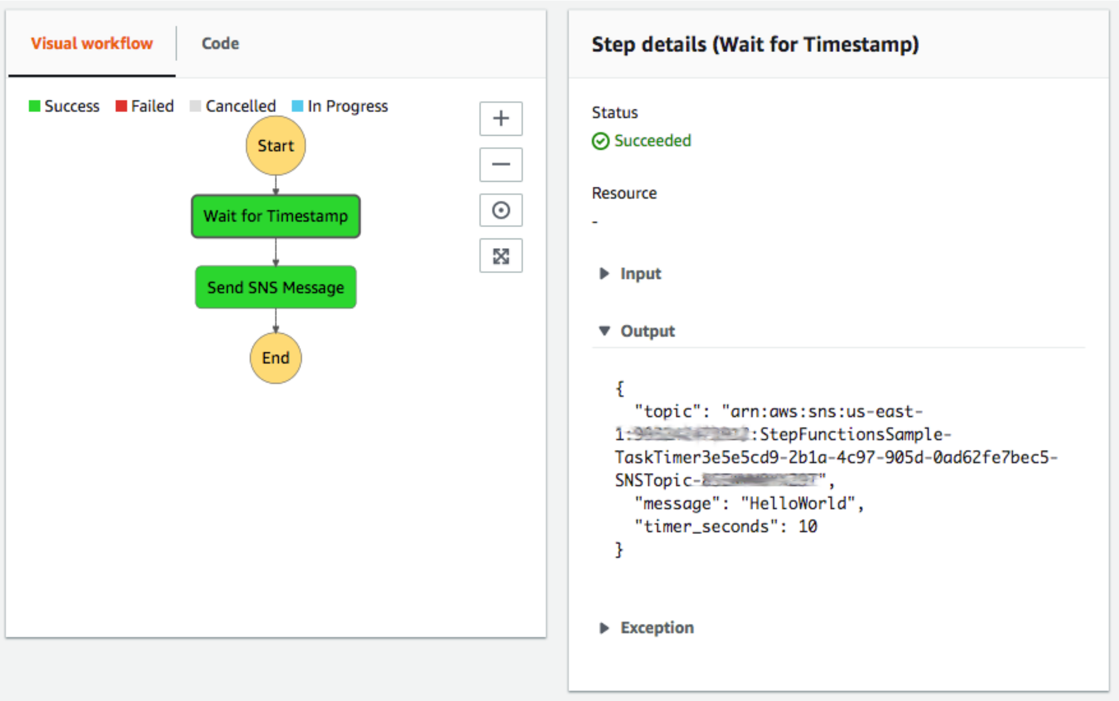
   When complete, the **New execution** page is displayed, with example input similar to the following.

```
{
  "topic": "arn:aws:sns:us-east-2:123456789012:StepFunctionsSample-TaskTimer-517b8680-e0ad-07cf-feee-65aa5fc63ac0-SNSTopic-96RHT77RAKTS",
  "message": "HelloWorld",
  "timer_seconds": 10
}
```

4. Choose **Start Execution**.

   A new execution of your state machine starts, and a new page showing your running execution is displayed.

5. (Optional) In the **Execution Details** section, choose **Info** to view the **Execution Status** and the **Started** and **Closed** timestamps.
6. To view the status, input, or output of each step in your execution, select the step in the **Visual workflow** and review the **Step details.**



Step details (Wait for Timestamp)

Status

Succeeded

Resource

-

Output

```
{
  "topic": "arn:aws:sns:us-east-
1:[redacted]:StepFunctionsSample-
TaskTimer3e5e5cd9-2b1a-4c97-905d-0ad62fe7bec5-
SNSTopic-[redacted]",
  "message": "HelloWorld",
  "timer_seconds": 10
}
```

Callback Pattern Example

This sample project demonstrates how to have AWS Step Functions pause during a task, and wait for an external process to return a task token that was generated when the task started.

When this sample project is deployed and an execution is started, the following steps occur.

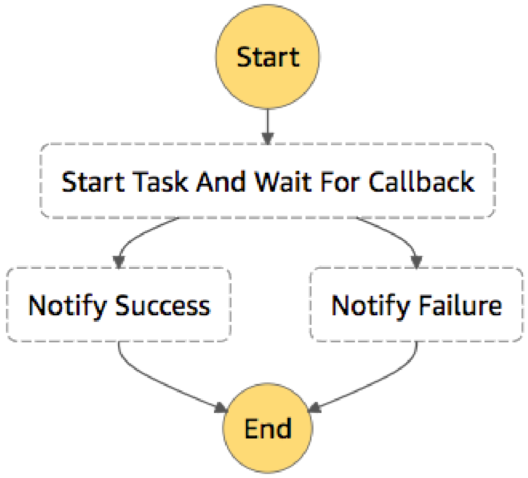
1. Step Functions passes a message that includes a task token to an Amazon SQS queue.
2. Step Functions then pauses, waiting for that token to be returned.
3. The Amazon Simple Queue Service queue triggers an AWS Lambda function that calls `SendTaskSuccess` with that same task token.
4. When the task token is received, the workflow continues.
5. The `"Notify Success"` task publishes an Amazon SNS message that the callback was received.

To learn how to implement the callback pattern in Step Functions see Wait for a Callback with the Task Token (p. 140).

To create the state machine and provision all resources

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Callback Pattern Example**.

   The state machine **Code** and **Visual Workflow** are displayed.



3. Choose **Create Sample Project**.

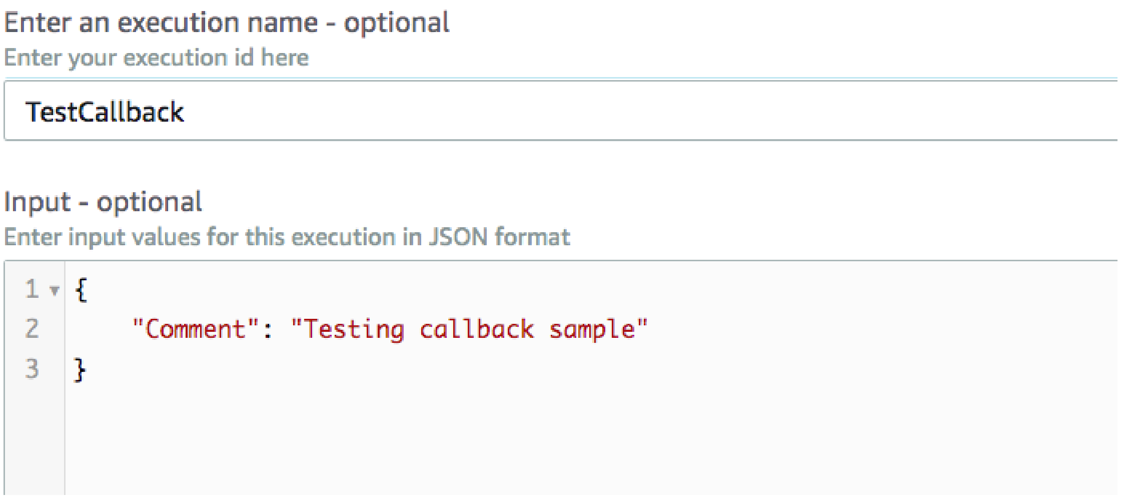
   The **Create Project Resources** page is displayed, listing the resources that will be created. For this sample project the resources include:

   - An Amazon SQS message queue.
   - An AWS Lambda function, that calls the Step Functions API action `SendTaskSuccess`.
   - An Amazon SNS topic to notify success or failure when the workflow can continues.

   > **Note**
   > It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Create Project Resources** page displays **Creating resources**, you can open the **Stack ID:** link to see which resources are being provisioned.

   When complete, the **New execution** page is displayed.

4. (Optional) Enter an execution name, and sample input.



Enter an execution name - optional

Enter your execution id here

TestCallback

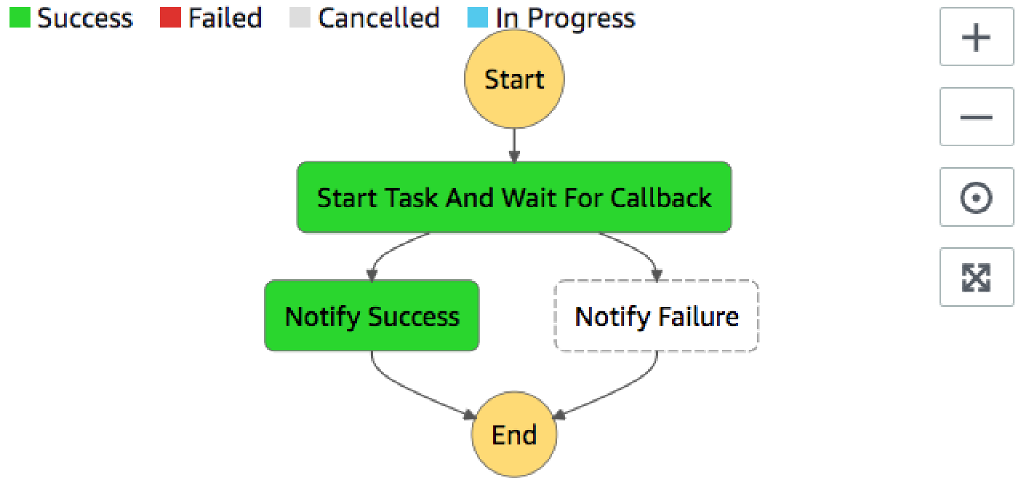
Input - optional

Enter input values for this execution in JSON format

```
{
    "Comment": "Testing callback sample"
}
```

5. Choose **Start Execution**.

A new execution of your state machine starts, and a new page showing your running execution is displayed.



6. To review how Step Functions progressed through the workflow and received a callback from Amazon SQS, review the entries in the **Execution event history**.

Lambda Callback Example

To see how the components of this sample project work together, see the resources that were deployed in your AWS account. For instance, here is the Lambda function that calls Step Functions with the task token.

```
console.log('Loading function');
const aws = require('aws-sdk');

exports.lambda_handler = (event, context, callback) => {
    const stepfunctions = new aws.StepFunctions();

    for (const record of event.Records) {
        const messageBody = JSON.parse(record.body);
        const taskToken = messageBody.TaskToken;

        const params = {
            output: "\"Callback task completed successfully.\"",
            taskToken: taskToken
        };

        console.log(`Calling Step Functions to complete callback task with params
 ${JSON.stringify(params)}`);

        stepfunctions.sendTaskSuccess(params, (err, data) => {
            if (err) {
                console.error(err.message);
                callback(err.message);
                return;
            }
            console.log(data);
            callback(null);
        });
    }
};
```

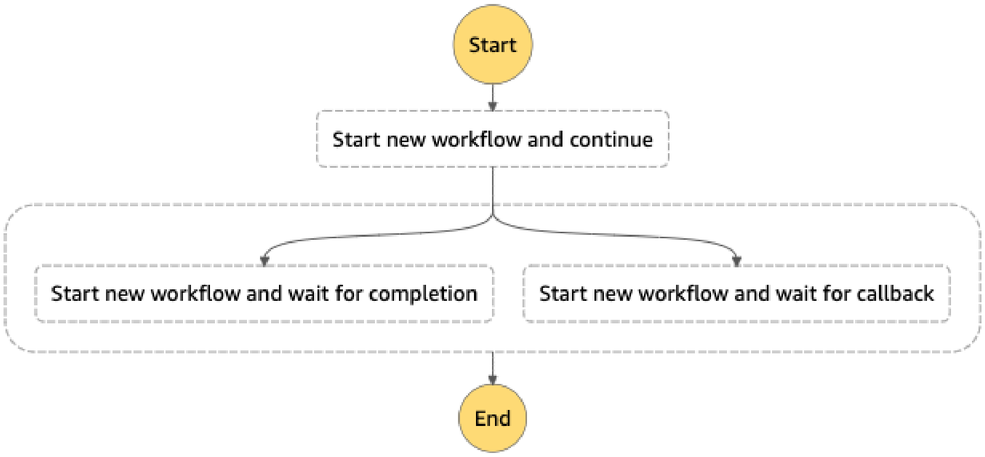
Start a Workflow Within a Workflow (AWS Step Functions, AWS Lambda)

This sample project demonstrates how to use a state machine to launch other state machine executions. See Start Workflow Executions From a Task State (p. 130).

To create the Start a Workflow Within a Workflow state machine and provision all resources

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Start a Workflow Within a Workflow**.

   The state machine **Code** and **Visual Workflow** are displayed.


Start
Start new workflow and continue
Start new workflow and wait for completion
Start new workflow and wait for callback
End

3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

   - An additional Step Functions state machine
   - A Lambda function for callback
4. Choose **Deploy Resources**.

   > **Note**
   > It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

Start a New Execution

1. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
2. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

> **Note**
> Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

3. Optionally, you can go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.
4. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.

Example State Machine Code

The state machine in this sample project integrates another state machine and AWS Lambda by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions calls the `StartExecution` API action for the other state machine. It launches two instances of the other state machine in parallel: one using the Run a Job (p. 140) pattern and one using the Wait for a Callback with the Task Token (p. 140) pattern.

For more information about how AWS Step Functions can control other AWS services, see AWS Service Integrations (p. 138).

```
{
  "Comment": "An example of combining workflows using a Step Functions StartExecution task
 state with various integration patterns.",
  "StartAt": "Start new workflow and continue",
  "States": {
    "Start new workflow and continue": {
      "Comment": "Start an execution of another Step Functions state machine and continue",
      "Type": "Task",
      "Resource": "arn:aws:states:::states:startExecution",
      "Parameters": {
        "StateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:NestingPatternAnotherStateMachine-HZ9gtgspmdun",
        "Input": {
          "NeedCallback": false,
          "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
        }
      },
      "Next": "Start in parallel"
    },
    "Start in parallel": {
      "Comment": "Start two executions of the same state machine in parallel",
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "Start new workflow and wait for completion",
          "States": {
            "Start new workflow and wait for completion": {
              "Comment": "Start an execution of the same
 'NestingPatternAnotherStateMachine' and wait for its completion",
              "Type": "Task",
              "Resource": "arn:aws:states:::states:startExecution.sync",
              "Parameters": {
                "StateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:NestingPatternAnotherStateMachine-HZ9gtgspmdun",
                "Input": {
                  "NeedCallback": false,
                  "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
                }
              },
              "OutputPath": "$.Output",
              "End": true
            }
          }
        },
        {
          "StartAt": "Start new workflow and wait for callback",
          "States": {
            "Start new workflow and wait for callback": {
              "Comment": "Start an execution and wait for it to call back with a task
 token",
              "Type": "Task",
              "Resource": "arn:aws:states:::states:startExecution.waitForTaskToken",
              "Parameters": {
                "StateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:NestingPatternAnotherStateMachine-HZ9gtgspmdun",
                "Input": {
                  "NeedCallback": true,
                  "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id",
                  "TaskToken.$": "$$.Task.Token"
                }
              },
              "End": true
            }
          }
        }
      ]
    }
  }
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Dynamically Process Data with a Map State

This sample project demonstrates dynamic parallelism using a `Map` state. This sample project creates the following.

- Two AWS Lambda functions
- An Amazon Simple Queue Service queue
- An Amazon Simple Notification Service topic
- An Amazon DynamoDB table
- An AWS Step Functions state machine

In this project, Step Functions uses an AWS Lambda function to pull messages off an Amazon SQS queue, and pass a JSON array of those message to a `Map` state. For each message in the queue, the state machine writes the message to DynamoDB, invokes the other Lambda function to remove the message from Amazon SQS, and then publishes the message to the Amazon SNS topic.

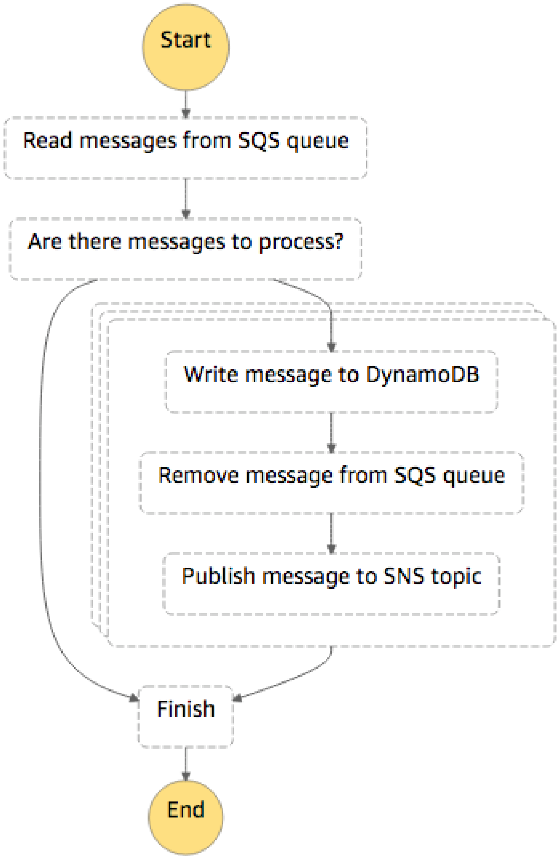
For more information on `Map` states and Step Functions service integrations, see the following.

- Map (p. 109)
- AWS Service Integrations (p. 138)

To create the state machine and provision all resources

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Iterate steps with a Map state**.

   The state machine **Code** and **Visual Workflow** are displayed.


Start
Read messages from SQS queue
Are there messages to process?
Write message to DynamoDB
Remove message from SQS queue
Publish message to SNS topic
Finish
End

3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

   - An Amazon Simple Queue Service queue
   - An Amazon Simple Notification Service topic
   - An Amazon DynamoDB table
   - Two AWS Lambda functions
   - An AWS Step Functions state machine
4. Choose **Deploy Resources**.

   Note
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

Once the resources of the sample project have been deployed, you'll need to add items to the Amazon SQS queue and subscribe to the Amazon SNS topic before you start an execution of the state machine.

Subscribe to the Amazon SNS Topic

1. Open the Amazon SNS console.
2. Select **Topics** and choose the topic that was created by the `Map` state sample project.

   The name will be similar to **MapSampleProj-SNSTopic-1CQO4HQ3IR1KN**.
3. Under **Subscriptions** choose **Create subscription**.

   The **Create subscription** page is displayed, listing the **Topic ARN** for the topic.
4. Under **Protocol** select **Email**.
5. Under **Endpoint** enter an email address to subscribe to the topic.
6. Select **Create subscription**.

   Note
   You must confirm the subscription in your email before it is active.
7. Open the **Subscription Confirmation** email in the related account and follow open the **Confirm subscription** URL.

   The **Subscription confirmed!** page is displayed.

Add Messages to the Amazon SQS Queue

1. Open the Amazon SQS console.
2. Select the queue that was created by the `Map` state sample project.

   The name will be similar to **MapSampleProj-SQSQueue-1UDIC9VZDORN7**.
3. On the **Queue Actions** drop-down, select **Send a Message**.
4. On the **Send a Message** window, type a message and select the **Send Message** button.
5. Select **Send Another Message**.

   Continue entering messages until you have several in the Amazon SQS queue.
6. Select **Close**.

Start a New Execution

Note
Queues in Amazon Simple Notification Service are eventually consistent. For best results, wait a few minutes between populating your queue and running an execution of your state machine.

1. Open the Step Functions console.
2. On the **State machines** page, choose the **MapStateStateMachine** state machine that was created by the sample project and select **Start execution**.
3. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
4. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   Note
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch.

   To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

5. Optionally, you can go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.
6. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.

Example State Machine Code

The state machine in this sample project integrates with Amazon SQS, Amazon SNS, and Lambda by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls Lambda, DynamoDB, Amazon SNS by connecting to the Amazon Resource Name (ARN) in the `Resource` field, and by passing `Parameters` to the service API.

For more information about how AWS Step Functions can control other AWS services, see AWS Service Integrations (p. 138).

```
{
  "Comment": "An example of the Amazon States Language for reading messages from an SQS
 queue and iteratively processing each message.",
  "StartAt": "Read messages from SQS Queue",
  "States": {
    "Read Messages from SQS Queue": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "OutputPath": "$.Payload",
      "Parameters": {
        "FunctionName": "MapSampleProj-ReadFromSQSQueueLambda-1MY3M63RMJVA9"
      },
      "Next": "Are there messages to process?"
    },
    "Are there messages to process?": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$",
          "StringEquals": "No messages",
          "Next": "Finish"
        }
      ],
      "Default": "Process messages"
    },
    "Process messages": {
      "Type": "Map",
      "Next": "Finish",
      "ItemsPath": "$",
      "Parameters": {
        "MessageNumber.$": "$$.Map.Item.Index",
        "MessageDetails.$": "$$.Map.Item.Value"
      },
      "Iterator": {
        "StartAt": "Write message to DynamoDB",
        "States": {
          "Write message to DynamoDB": {
            "Type": "Task",
            "Resource": "arn:aws:states:::dynamodb:putItem",
            "ResultPath": null,
            "Parameters": {
              "TableName": "MapSampleProj-DDBTable-YJDJ1MKIN6C5",
              "ReturnConsumedCapacity": "TOTAL",
              "Item": {
                "MessageId": {
                  "S.$": "$.MessageDetails.MessageId"
                },
                "Body": {
                  "S.$": "$.MessageDetails.Body"
                }
              }
            },
            "Next": "Remove message from SQS queue"
          },
          "Remove message from SQS queue": {
            "Type": "Task",
            "Resource": "arn:aws:states:::lambda:invoke",
            "InputPath": "$.MessageDetails",
            "ResultPath": null,
            "Parameters": {
              "FunctionName": "MapSampleProj-DeleteFromSQSQueueLambda-198J2839ZO5K2",
              "Payload": {
                "ReceiptHandle.$": "$.ReceiptHandle"
              }
            },
            "Next": "Publish message to SNS topic"
          },
          "Publish message to SNS topic": {
            "Type": "Task",
            "Resource": "arn:aws:states:::sns:publish",
            "InputPath": "$.MessageDetails",
            "Parameters": {
              "Subject": "Message from Step Functions!",
              "Message.$": "$.Body",
              "TopicArn": "arn:aws:sns:us-east-1:012345678910:MapSampleProj-
SNSTopic-1CQO4HQ3IR1KN"
            },
            "End": true
          }
        }
      }
    },
    "Finish": {
      "Type": "Succeed"
    }
  }
}
```

IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "arn:aws:lambda:us-east-1:012345678901:function:MapSampleProj-
ReadFromSQSQueueLambda-1MY3M63RMJVA9",
                "arn:aws:lambda:us-east-1:012345678901:function:MapSampleProj-
DeleteFromSQSQueueLambda-198J2839ZO5K2"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "dynamodb:PutItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:012345678901:table/MapSampleProj-DDBTable-
YJDJ1MKIN6C5"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "sns:Publish"
            ],
            "Resource": [
                "arn:aws:sns:us-east-1:012345678901:MapSampleProj-SNSTopic-1CQO4HQ3IR1KN"
            ],
            "Effect": "Allow"
        }
    ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Train a Machine Learning Model

This sample project demonstrates using Amazon SageMaker to train a machine learning model and how to batch transform a test dataset. This sample project creates the following.

- An AWS Lambda function
- An Amazon S3 bucket
- An AWS Step Functions state machine
- Related IAM roles

In this project, Step Functions uses an AWS Lambda function to seed an Amazon S3 bucket with a test dataset. It then trains a machine learning model and performs a batch transform, using the Amazon SageMaker service integration (p. 156).

For more information on Amazon SageMaker and Step Functions service integrations, see the following.

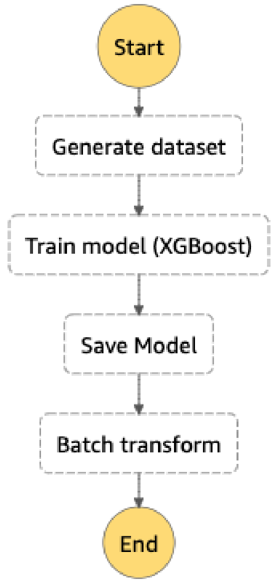
- AWS Service Integrations (p. 138)
- Manage Amazon SageMaker with Step Functions (p. 156)

Note
This sample project may incur charges.
For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see Amazon SageMaker Pricing.

To create the state machine and provision all resources

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Train a machine learning model**.

   The state machine **Code** and **Visual Workflow** are displayed.

3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

   - A AWS Lambda function
   - An Amazon S3 bucket
   - An AWS Step Functions state machine
   - Related IAM roles
4. Choose **Deploy Resources**.

   > **Note**
   > It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

Start a New Execution

1. Open the Step Functions console.
2. On the **State machines** page, choose the **TrainAndBatchTransformStateMachine** state machine that was created by the sample project and select **Start execution**.
3. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
4. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

> **Note**
> Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

5. Optionally, you can go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.
6. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.

Example State Machine Code

The state machine in this sample project integrates with Amazon SageMaker and AWS Lambda by passing parameters directly to those resources, and uses an Amazon S3 bucket for the training data source and output.

Browse through this example state machine to see how Step Functions controls Lambda and Amazon SageMaker.

For more information about how AWS Step Functions can control other AWS services, see AWS Service Integrations (p. 138).

```
{
  "StartAt": "Generate dataset",
  "States": {
    "Generate dataset": {
      "Resource": "arn:aws:lambda:us-west-2:123456789012:function:TrainAndBatchTransform-
SeedingFunction-17RNSOTG97HPV",
      "Type": "Task",
      "Next": "Train model (XGBoost)"
    },
    "Train model (XGBoost)": {
      "Resource": "arn:aws:states:::sagemaker:createTrainingJob.sync",
      "Parameters": {
        "AlgorithmSpecification": {
          "TrainingImage": "433757028032.dkr.ecr.us-west-2.amazonaws.com/xgboost:latest",
          "TrainingInputMode": "File"
        },
        "OutputDataConfig": {
          "S3OutputPath": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/models"
        },
        "StoppingCondition": {
          "MaxRuntimeInSeconds": 86400
        },
        "ResourceConfig": {
          "InstanceCount": 1,
          "InstanceType": "ml.m4.xlarge",
          "VolumeSizeInGB": 30
        },
        "RoleArn": "arn:aws:iam::123456789012:role/TrainAndBatchTransform-
SageMakerAPIExecutionRole-Y9IX3DLF6EUO",
        "InputDataConfig": [
          {
            "DataSource": {
              "S3DataSource": {
                "S3DataDistributionType": "ShardedByS3Key",
                "S3DataType": "S3Prefix",
                "S3Uri": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/csv/train.csv"
              }
            },
            "ChannelName": "train",
            "ContentType": "text/csv"
          }
        ],
        "HyperParameters": {
          "objective": "reg:logistic",
          "eval_metric": "rmse",
          "num_round": "5"
        },
        "TrainingJobName.$": "$$.Execution.Name"
      },
      "Type": "Task",
      "Next": "Save Model"
    },
    "Save Model": {
      "Parameters": {
        "PrimaryContainer": {
          "Image": "433757028032.dkr.ecr.us-west-2.amazonaws.com/xgboost:latest",
          "Environment": {},
          "ModelDataUrl.$": "$.ModelArtifacts.S3ModelArtifacts"
        },
        "ExecutionRoleArn": "arn:aws:iam::123456789012:role/TrainAndBatchTransform-
SageMakerAPIExecutionRole-Y9IX3DLF6EUO",
        "ModelName.$": "$.TrainingJobName"
      },
      "Resource": "arn:aws:states:::sagemaker:createModel",
      "Type": "Task",
      "Next": "Batch transform"
    },
    "Batch transform": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sagemaker:createTransformJob.sync",
      "Parameters": {
        "ModelName.$": "$$.Execution.Name",
        "TransformInput": {
          "CompressionType": "None",
          "ContentType": "text/csv",
          "DataSource": {
            "S3DataSource": {
              "S3DataType": "S3Prefix",
              "S3Uri": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/csv/test.csv"
            }
          }
        },
        "TransformOutput": {
          "S3OutputPath": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/output"
        },
        "TransformResources": {
          "InstanceCount": 1,
          "InstanceType": "ml.m4.xlarge"
        },
        "TransformJobName.$": "$$.Execution.Name"
      },
      "End": true
    }
  }
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "cloudwatch:PutMetricData",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:DescribeLogStreams",
                "s3:GetObject",
                "s3:PutObject",
                "s3:ListBucket",
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage"
            ],
            "Resource": "*",
            "Effect": "Allow"
        }
    ]
}
```

The following policy allows the AWS Lambda function to seed the Amazon S3 bucket with sample data.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::trainandbatchtransform-s3bucket-1jn1le6gadwfz/*",
            "Effect": "Allow"
        }
    ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Tune a Machine Learning Model

This sample project demonstrates using Amazon SageMaker to tune the hyperparameters of a machine learning model, and to batch transform a test dataset. This sample project creates the following.

- Three AWS Lambda functions
- An Amazon S3 bucket
- An AWS Step Functions state machine

- Related IAM roles

In this project, Step Functions uses an AWS Lambda function to seed an Amazon S3 bucket with a test dataset. It then creates a hyperparameter tuning job using the Amazon SageMaker service integration (p. 156). It then uses a Lambda function to extract the data path, saves the tuning model, extracts the model name, and then runs a batch transform job to perform inference in Amazon SageMaker.

For more information on Amazon SageMaker and Step Functions service integrations, see the following.

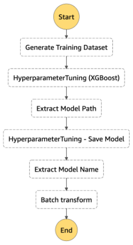
- AWS Service Integrations (p. 138)
- Manage Amazon SageMaker with Step Functions (p. 156)

> **Note**
> This sample project may incur charges.
> For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see Amazon SageMaker Pricing.

To create the state machine and provision all resources

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Tune a machine learning model**.

   The state machine **Code** and **Visual Workflow** are displayed.

   
   Start
   Generate Training Dataset
   HyperparameterTuning (XGBoost)
   Extract Model Path
   HyperparameterTuning - Save Model
   Extract Model Name
   Batch transform
   End

3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

   - Three AWS Lambda functions
   - An Amazon S3 bucket
   - An AWS Step Functions state machine
   - Related IAM roles
4. Choose **Deploy Resources**.

   > **Note**
   > It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

Start a New Execution

1. Open the Step Functions console.

2. On the **State machines** page, choose the **HyperparamTuningAndBatchTransformStateMachine** state machine that was created by the sample project and select **Start execution**.
3. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
4. (Optional) To help identify your execution, you can specify an ID for it in the **Enter an execution name** box. If you don't enter an ID, Step Functions generates a unique ID automatically.

   Note
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
5. Optionally, you can go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.
6. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.

Example State Machine Code

The state machine in this sample project integrates with Amazon SageMaker and AWS Lambda by passing parameters directly to those resources, and uses an Amazon S3 bucket for the training data source and output.

Browse through this example state machine to see how Step Functions controls Lambda and Amazon SageMaker.

For more information about how AWS Step Functions can control other AWS services, see AWS Service Integrations (p. 138).

```
{
    "StartAt": "Generate Training Dataset",
    "States": {
        "Generate Training Dataset": {
            "Resource": "arn:aws:lambda:us-
west-2:012345678912:function:StepFunctionsSample-SageMa-
LambdaForDataGeneration-1TF67BUE5A12U",
            "Type": "Task",
            "Next": "HyperparameterTuning (XGBoost)"
        },
        "HyperparameterTuning (XGBoost)": {
            "Resource": "arn:aws:states:::sagemaker:createHyperParameterTuningJob.sync",
            "Parameters": {
                "HyperParameterTuningJobName.$": "$.body.jobName",
                "HyperParameterTuningJobConfig": {
                    "Strategy": "Bayesian",
                    "HyperParameterTuningJobObjective": {
                        "Type": "Minimize",
                        "MetricName": "validation:rmse"
                    },
                    "ResourceLimits": {
                        "MaxNumberOfTrainingJobs": 2,
                        "MaxParallelTrainingJobs": 2
                    },
                    "ParameterRanges": {
                        "ContinuousParameterRanges": [{
                                "Name": "alpha",
                                "MinValue": "0",
                                "MaxValue": "1000",
                                "ScalingType": "Auto"
                            },
                            {
                                "Name": "gamma",
                                "MinValue": "0",
                                "MaxValue": "5",
                                "ScalingType": "Auto"
                            }
                        ],
                        "IntegerParameterRanges": [{
                                "Name": "max_delta_step",
                                "MinValue": "0",
                                "MaxValue": "10",
                                "ScalingType": "Auto"
                            },
                            {
                                "Name": "max_depth",
                                "MinValue": "0",
                                "MaxValue": "10",
                                "ScalingType": "Auto"
                            }
                        ]
                    }
                },
                "TrainingJobDefinition": {
                    "AlgorithmSpecification": {
                        "TrainingImage": "433757028032.dkr.ecr.us-west-2.amazonaws.com/
xgboost:latest",
                        "TrainingInputMode": "File"
                    },
                    "OutputDataConfig": {
                        "S3OutputPath": "s3://stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/models"
                    },
                    "StoppingCondition": {
                        "MaxRuntimeInSeconds": 86400
                    },
                    "ResourceConfig": {
                        "InstanceCount": 1,
                        "InstanceType": "ml.m4.xlarge",
                        "VolumeSizeInGB": 30
                    },
                    "RoleArn": "arn:aws:iam::012345678912:role/StepFunctionsSample-SageM-
SageMakerAPIExecutionRol-1MNH1VS5CGGOG",
                    "InputDataConfig": [{
                        "DataSource": {
                            "S3DataSource": {
                                "S3DataDistributionType": "FullyReplicated",
                                "S3DataType": "S3Prefix",
                                "S3Uri": "s3://stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/csv/train.csv"
                            }
                        },
                        "ChannelName": "train",
                        "ContentType": "text/csv"
                    },
                    {
                        "DataSource": {
                            "S3DataSource": {
                                "S3DataDistributionType": "FullyReplicated",
                                "S3DataType": "S3Prefix",
                                "S3Uri": "s3://stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/csv/validation.csv"
                            }
                        },
                        "ChannelName": "validation",
                        "ContentType": "text/csv"
                    }],
                    "StaticHyperParameters": {
                        "precision_dtype": "float32",
                        "num_round": "2"
                    }
                }
            },
            "Type": "Task",
            "Next": "Extract Model Path"
        },
        "Extract Model Path": {
            "Resource": "arn:aws:lambda:us-
west-2:012345678912:function:StepFunctionsSample-SageM-LambdaToExtractModelPath-
V0R37CVARUS9",
            "Type": "Task",
            "Next": "HyperparameterTuning - Save Model"
        },
        "HyperparameterTuning - Save Model": {
            "Parameters": {
                "PrimaryContainer": {
                    "Image": "433757028032.dkr.ecr.us-west-2.amazonaws.com/xgboost:latest",
                    "Environment": {},
                    "ModelDataUrl.$": "$.body.modelDataUrl"
                },
                "ExecutionRoleArn": "arn:aws:iam::012345678912:role/StepFunctionsSample-
SageM-SageMakerAPIExecutionRol-1MNH1VS5CGGOG",
                "ModelName.$": "$.body.bestTrainingJobName"
            },
            "Resource": "arn:aws:states:::sagemaker:createModel",
            "Type": "Task",
            "Next": "Extract Model Name"
        },
        "Extract Model Name": {
            "Resource": "arn:aws:lambda:us-
west-2:012345678912:function:StepFunctionsSample-SageM-
LambdaToExtractModelName-8FUOB30SM5EM",
            "Type": "Task",
            "Next": "Batch transform"
        },
        "Batch transform": {
            "Type": "Task",
            "Resource": "arn:aws:states:::sagemaker:createTransformJob.sync",
            "Parameters": {
                "ModelName.$": "$.body.jobName",
                "TransformInput": {
                    "CompressionType": "None",
                    "ContentType": "text/csv",
                    "DataSource": {
                        "S3DataSource": {
                            "S3DataType": "S3Prefix",
                            "S3Uri": "s3://stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/csv/test.csv"
                        }
                    }
                },
                "TransformOutput": {
                    "S3OutputPath": "s3://stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/output"
                },
                "TransformResources": {
                    "InstanceCount": 1,
                    "InstanceType": "ml.m4.xlarge"
                },
                "TransformJobName.$": "$.body.jobName"
            },
            "End": true
        }
```

```
    }
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

IAM Examples

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

The following AWS Identity and Access Management policy is attached to the state machine, and allows the state machine execution to access necessary Amazon SageMaker, Lambda, and Amazon S3 resources.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "sagemaker:CreateHyperParameterTuningJob",
                "sagemaker:DescribeHyperParameterTuningJob",
                "sagemaker:StopHyperParameterTuningJob",
                "sagemaker:ListTags",
                "sagemaker:CreateModel",
                "sagemaker:CreateTransformJob",
                "iam:PassRole"
            ],
            "Resource": "*",
            "Effect": "Allow"
        },
        {
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "arn:aws:lambda:us-west-2:012345678912:function:StepFunctionsSample-SageMa-
LambdaForDataGeneration-1TF67BUE5A12U",
                "arn:aws:lambda:us-west-2:012345678912:function:StepFunctionsSample-SageM-
LambdaToExtractModelPath-V0R37CVARUS9",
                "arn:aws:lambda:us-west-2:012345678912:function:StepFunctionsSample-SageM-
LambdaToExtractModelName-8FUOB30SM5EM"
            ],
            "Effect": "Allow"
        },
        {
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:*:*:rule/
StepFunctionsGetEventsForSageMakerTrainingJobsRule",
                "arn:aws:events:*:*:rule/
StepFunctionsGetEventsForSageMakerTransformJobsRule",
                "arn:aws:events:*:*:rule/StepFunctionsGetEventsForSageMakerTuningJobsRule"
            ],
            "Effect": "Allow"
        }
    ]
}
```

The following AWS Identity and Access Management policy is referenced in the `TrainingJobDefinition` and `HyperparameterTuning` fields of the `HyperparameterTuning` state.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "cloudwatch:PutMetricData",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:DescribeLogStreams",
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage",
                "sagemaker:DescribeHyperParameterTuningJob",
                "sagemaker:StopHyperParameterTuningJob",
                "sagemaker:ListTags"
            ],
            "Resource": "*",
            "Effect": "Allow"
        },
        {
            "Action": [
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/*",
            "Effect": "Allow"
        },
        {
            "Action": [
                "s3:ListBucket"
            ],
            "Resource": "arn:aws:s3:::stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f",
            "Effect": "Allow"
        }
    ]
}
```

The following policy allows the AWS Lambda function to seed the Amazon S3 bucket with sample data.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::stepfunctionssample-sagemak-
bucketformodelanddata-80fblmdlcs9f/*",
            "Effect": "Allow"
        }
    ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for Integrated Services (p. 234).

Best Practices for Step Functions

The following best practices for implementing AWS Step Functions workflows can help you optimize the performance of your implementations.

Topics

Use Timeouts to Avoid Stuck Executions

By default, the Amazon States Language doesn't set timeouts in state machine definitions. Without an explicit timeout, Step Functions often relies solely on a response from an activity worker to know that a task is complete. If something goes wrong and `TimeoutSeconds` isn't specified, an execution is stuck waiting for a response that will never come.

To avoid this, specify a reasonable timeout limit when you create a task in your state machine. For example:

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

For more information, see Task (p. 91) in the Amazon States Language documentation.

Use ARNs Instead of Passing Large Payloads

Executions that pass large payloads of data between states can be terminated. If the data you are passing between states might grow to over 32 KB, use Amazon Simple Storage Service (Amazon S3) to store the data, and pass the Amazon Resource Name (ARN) instead of the raw data. Alternatively, adjust your implementation so that you pass smaller payloads in your executions.

For instance, a state machine can pass input to a Lambda function, like in the following example.

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

Rather than pass a large amount of data in the input, you could save that data in an Amazon S3 bucket, and pass the Amazon Resource Name of that bucket in the Payload parameter. Your Lambda function can then use that ARN to access the data directly. The following is example input for an state machine execution, where the data is stored in data.json in an Amazon S3 bucket.

```
{
  "Data": "arn:aws:s3:::MyBucket/data.json"
}
```

For more information, see:

- Limits (p. 206)
- Amazon Simple Storage Service Developer Guide
- Amazon Resource Names (ARNs)

Avoid Reaching the History Limit

AWS Step Functions has a hard limit of 25,000 entries in the execution history. To avoid reaching this limit for long-running executions, implement a pattern that uses an AWS Lambda function that can start a new execution of your state machine to split ongoing work across multiple workflow executions.

For more information, see the Continue as a New Execution (p. 59) tutorial.

Handle Lambda Service Exceptions

AWS Lambda can occasionally experience transient service errors. In this case, invoking Lambda will result in a 500 error such as ServiceException, AWSLambdaException, or SdkClientException. As a best practice, proactively handle these exceptions in your state machine to Retry invoking your Lambda function, or to Catch the error.

Lambda errors are reported as Lambda.*ErrorName*. To retry a Lambda service exception error, you could use the following Retry code.

```
"Retry": [ {
   "ErrorEquals": [ "Lambda.ServiceException", "Lambda.AWSLambdaException",
 "Lambda.SdkClientException"],
   "IntervalSeconds": 2,
   "MaxAttempts": 6,
   "BackoffRate": 2
} ]
```

Note
Unhandled errors in Lambda are reported as Lambda.Unknown in the error output. These include out-of-memory errors, function timeouts, and hitting the concurrent Lambda invoke limit. You can match on Lambda.Unknown, States.ALL, or States.TaskFailed to handle these errors. When Lambda hits the invocation limit, the error

is `Lambda.TooManyRequestsException`. For more information about Lambda `Handled` and `Unhandled` errors, see `FunctionError` in the AWS Lambda Developer Guide.

For more information, see:

- Retrying after an Error (p. 133)
- Handling Error Conditions Using a State Machine (p. 35)
- Lambda Invoke Errors

Avoid Latency When Polling for Activity Tasks

The `GetActivityTask` API is designed to provide a `taskToken` *exactly-once*. If a `taskToken` is dropped while communicating with an activity worker, a number of `GetActivityTask` requests can be blocked for 60 seconds waiting for a response until `GetActivityTask` times out. If you only have a small number of polls waiting for a response, it is possible that all requests will queue up behind the blocked request and stop. However, if you have a large number of outstanding polls for each activity ARN, and some percentage of your requests are stuck waiting, there will be many more that can still get a `taskToken` and begin to process work.

For production systems, we recommend at least 100 open polls per activity ARN's at each point in time. If one poll gets blocked, and a portion of those polls queue up behind it, there are still many more requests that will receive a `taskToken` to process work while the `GetActivityTask` request is blocked.

To avoid these kinds of latency problems when polling for tasks:

- Implement your pollers as separate threads from the work in your activity worker implementation.
- Have at least 100 open polls per activity ARN at each point in time.

  Note
  Scaling to 100 open polls per ARN can be expensive. For example, 100 Lambda functions polling per ARN is 100 times more expensive than having a single Lambda function with 100 polling threads. To both reduce latency *and* minimize cost, use a language that has asynchronous I/O, and implement multiple polling threads per worker. For an example activity worker where the poller threads are separate from the work threads, see Example Activity Worker in Ruby (p. 96).

For more information on activities and activity workers see Activities (p. 94).

Limits

AWS Step Functions places limits on the sizes of certain state machine parameters, such as the number of API actions that you can make during a certain time period or the number of state machines that you can define. Although these limits are designed to prevent a misconfigured state machine from consuming all of the resources of the system, many aren't hard limits.

> **Note**
> If a particular stage of your state machine execution or activity execution takes too long, you can configure a state machine timeout to cause a timeout event.

Topics

General Limits

Limit	Description
Names in Step Functions	State machine, execution, and activity names must be 1–80 characters in length, must be unique for your account and AWS Region, and must not contain any of the following: • Whitespace • Wildcard characters (`? *`) • Bracket characters (`< > { } [ ]`) • Special characters (`: ; , \ | ^ ~ $ # % & ` "`) • Control characters (`\\u0000` - `\\u001f` or `\\u007f` - `\\u009f`). Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

Limits Related to Accounts

Limit	Description
Maximum number of registered activities	10,000
Maximum number of registered state machines	10,000
Maximum number of API actions	Beyond infrequent spikes, applications can be throttled if they make a large number of API actions in a very short period of time.
Maximum request size	1 MB per request. This is the total data size per Step Functions API request, including the request header and all other associated request data.
Maximum open executions per account	1,000,000 executions per AWS account. Exceeding this limit will cause an `ExecutionLimitExceeded` error.

Limits Related to State Machine Executions

Limit	Description
Maximum execution time	1 year. If an execution runs for more than the 1-year limit, it will fail with a `States.Timeout` error and emit a `ExecutionsTimedout` CloudWatch metric.
Maximum execution history size	25,000 events. If the execution history reaches this limit the execution will fail. To avoid this, see Avoid Reaching the History Limit (p. 204).
Maximum execution idle time	1 year (constrained by execution time limit)
Maximum execution history retention time	90 days. After this time, you can no longer retrieve or view the execution history. There is no further limit to the number of closed executions that Step Functions retains.

Limits Related to Task Executions

Limit	Description
Maximum task execution time	1 year (constrained by execution time limit)
Maximum time Step Functions keeps a task in the queue	1 year (constrained by execution time limit)
Maximum activity pollers per Amazon Resource Name (ARN)	1,000 pollers calling `GetActivityTask` per ARN. Exceeding this limit results in this error: *"The maximum number of workers concurrently polling for activity tasks has been reached."*

Limit	Description
Maximum input or result data size for a task, state, or execution	32,768 characters. This limit affects tasks (activity or Lambda function), state or execution result data, and input data when scheduling a task, entering a state, or starting an execution.

Limits Related to API Action Throttling

Some Step Functions API actions are throttled using a token bucket scheme to maintain service bandwidth.

Note
Throttling limits are per account, per AWS Region. AWS Step Functions may increase both the bucket size and refill rate at any time. Do not rely on these throttling rates to limit your costs.

Limits In US East (N. Virginia), US West (Oregon), and EU (Ireland)

API Name	Bucket Size	Refill Rate per Second
`CreateActivity`	100	1
`CreateStateMachine`	100	1
`DeleteActivity`	100	1
`DeleteStateMachine`	100	1
`DescribeActivity`	200	1
`DescribeExecution`	300	15
`DescribeStateMachine`	200	20
`DescribeStateMachineForExecution`	200	1
`GetActivityTask`	3,000	500
`GetExecutionHistory`	400	20
`ListActivities`	100	10
`ListExecutions`	200	5
`ListStateMachines`	100	5
`ListTagsForResource`	100	1
`SendTaskFailure`	3,000	500
`SendTaskHeartbeat`	3,000	500
`SendTaskSuccess`	3,000	500
`StartExecution`	1,300	300

API Name	Bucket Size	Refill Rate per Second
`StopExecution`	1,000	200
`TagResource`	200	1
`UntagResource`	200	1
`UpdateStateMachine`	100	1

Limits In All Other Regions

API Name	Bucket Size	Refill Rate per Second
`CreateActivity`	100	1
`CreateStateMachine`	100	1
`DeleteActivity`	100	1
`DeleteStateMachine`	100	1
`DescribeActivity`	200	1
`DescribeExecution`	250	10
`DescribeStateMachine`	200	20
`DescribeStateMachineForExecution`	200	1
`GetActivityTask`	1,500	300
`GetExecutionHistory`	400	20
`ListActivities`	100	5
`ListExecutions`	100	2
`ListStateMachines`	100	5
`ListTagsForResource`	100	1
`SendTaskFailure`	1,500	300
`SendTaskHeartbeat`	1,500	300
`SendTaskSuccess`	1,500	300
`StartExecution`	800	150
`StopExecution`	500	25
`TagResource`	200	1
`UntagResource`	200	1
`UpdateStateMachine`	200	1

Limits Related to State Throttling

Step Functions state transitions are throttled using a token bucket scheme to maintain service bandwidth.

> **Note**
> Throttling on the `StateTransition` service metric is reported as `ExecutionThrottled` in Amazon CloudWatch. For more information, see the `ExecutionThrottled` CloudWatch metric (p. 212).

Service Metric	Bucket Size	Refill Rate per Second
`StateTransition` — *In US East (N. Virginia), US West (Oregon), and EU (Ireland)*	5,000	1,500
`StateTransition` — *All other regions*	800	500

Restrictions Related to Tagging

Be aware of these restrictions when tagging Step Functions resources.

> **Note**
> Tagging restrictions cannot be increased like other limits.

Restriction	Description
Maximum number of tags per resource	50
Maximum key length	128 Unicode characters in UTF-8
Maximum value length	256 Unicode characters in UTF-8
Prefix restriction	Do not use the `aws:` prefix in your tag names or values because it is reserved for AWS use. You can't edit or delete tag names or values with this prefix. Tags with this prefix do not count against your tags per resource limit.
Character restrictions	Tags may only contain Unicode letters, digits, whitespace, or these symbols: `_ . : / = + - @`

Requesting a Limit Increase

Use the **Support Center** page in the AWS Management Console to request a limit increase for resources provided by AWS Step Functions on a per-Region basis. For more information, see To Request a Limit Increase in the *AWS General Reference*.

Monitoring and Logging

This section provides information about monitoring and logging AWS Step Functions.

Topics

Monitoring Step Functions Using CloudWatch

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS Step Functions and your AWS solutions. You should collect as much monitoring data from the AWS services that you use so that you can more easily debug any multi-point failures. Before you start monitoring Step Functions, you should create a monitoring plan that answers the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

The next step is to establish a baseline for normal Step Functions performance in your environment. To do this, measure performance at various times and under different load conditions. As you monitor Step Functions, consider storing historical monitoring data. Such data can give you a baseline to compare against current performance data, to identify normal performance patterns and performance anomalies, and to devise ways to address issues.

For example, with Step Functions, you can monitor how many activities or Lambda tasks fail due to a heartbeat timeout. When performance falls outside your established baseline, you might have to change your heartbeat interval.

To establish a baseline you should, at a minimum, monitor the following metrics:

- `ActivitiesStarted`
- `ActivitiesTimedOut`
- `ExecutionsStarted`
- `ExecutionsTimedOut`
- `LambdaFunctionsStarted`
- `LambdaFunctionsTimedOut`

The following sections describe metrics that Step Functions provides to Amazon CloudWatch. You can use these metrics to track your state machines and activities and to set alarms on threshold values. You can view metrics using the AWS Management Console.

Topics

Metrics That Report a Time Interval

Some of the Step Functions CloudWatch metrics are *time intervals*, always measured in milliseconds. These metrics generally correspond to stages of your execution for which you can set state machine, activity, and Lambda function timeouts, with descriptive names.

For example, the `ActivityRunTime` metric measures the time it takes for an activity to complete after it begins to execute. You can set a timeout value for the same time period.

In the CloudWatch console, you can get the best results if you choose **average** as the display statistic for time interval metrics.

Metrics That Report a Count

Some of the Step Functions CloudWatch metrics report results as a *count*. For example, `ExecutionsFailed` records the number of failed state machine executions.

In the CloudWatch console, you can get the best results if you choose **sum** as the display statistic for count metrics.

Execution Metrics

The `AWS/States` namespace includes the following metrics for Step Functions executions.

Metric	Description
`ExecutionTime`	The interval, in milliseconds, between the time the execution starts and the time it closes.
`ExecutionThrottled`	The number of `StateEntered` events and retries that have been throttled. This is related to `StateTransition` throttling. For more information, see Limits Related to State Throttling in the *AWS Step Functions Developer Guide*.
`ExecutionsAborted`	The number of aborted or terminated executions.
`ExecutionsFailed`	The number of failed executions.
`ExecutionsStarted`	The number of started executions.
`ExecutionsSucceeded`	The number of successfully completed executions.
`ExecutionsTimedOut`	The number of executions that time out for any reason.

Dimension for Step Functions Execution Metrics

Dimension	Description
`StateMachineArn`	The Amazon Resource Name (ARN) of the state machine for the execution in question.

Activity Metrics

The `AWS/States` namespace includes the following metrics for Step Functions activities.

Metric	Description
`ActivityRunTime`	The interval, in milliseconds, between the time the activity starts and the time it closes.
`ActivityScheduleTime`	The interval, in milliseconds, for which the activity stays in the schedule state.
`ActivityTime`	The interval, in milliseconds, between the time the activity is scheduled and the time it closes.
`ActivitiesFailed`	The number of failed activities.
`ActivitiesHeartbeatTimedOut`	The number of activities that time out due to a heartbeat timeout.
`ActivitiesScheduled`	The number of scheduled activities.
`ActivitiesStarted`	The number of started activities.
`ActivitiesSucceeded`	The number of successfully completed activities.
`ActivitiesTimedOut`	The number of activities that time out on close.

Dimension for Step Functions Activity Metrics

Dimension	Description
`ActivityArn`	The ARN of the activity.

Lambda Function Metrics

The `AWS/States` namespace includes the following metrics for Step Functions Lambda functions.

Metric	Description
`LambdaFunctionRunTime`	The interval, in milliseconds, between the time the Lambda function starts and the time it closes.
`LambdaFunctionScheduleTime`	The interval, in milliseconds, for which the Lambda function stays in the schedule state.

Metric	Description
`LambdaFunctionTime`	The interval, in milliseconds, between the time the Lambda function is scheduled and the time it closes.
`LambdaFunctionsFailed`	The number of failed Lambda functions.
`LambdaFunctionsScheduled`	The number of scheduled Lambda functions.
`LambdaFunctionsStarted`	The number of started Lambda functions.
`LambdaFunctionsSucceeded`	The number of successfully completed Lambda functions.
`LambdaFunctionsTimedOut`	The number of Lambda functions that time out on close.

Dimension for Step Functions Lambda Function Metrics

Dimension	Description
`LambdaFunctionArn`	The ARN of the Lambda function.

Service Integration Metrics

The `AWS/States` namespace includes the following metrics for Step Functions service integrations. For more information see AWS Service Integrations (p. 138).

Metric	Description
`ServiceIntegrationRunTime`	The interval, in milliseconds, between the time the Service Task starts and the time it closes.
`ServiceIntegrationScheduleTime`	The interval, in milliseconds, for which the Service Task stays in the schedule state.
`ServiceIntegrationTime`	The interval, in milliseconds, between the time the Service Task is scheduled and the time it closes.
`ServiceIntegrationsFailed`	The number of failed Service Tasks.
`ServiceIntegrationsScheduled`	The number of scheduled Service Tasks.
`ServiceIntegrationsStarted`	The number of started Service Tasks.
`ServiceIntegrationsSucceeded`	The number of successfully completed Service Tasks.
`ServiceIntegrationsTimedOut`	The number of Service Tasks that time out on close.

Dimension for Step Functions Service Integration Metrics

Dimension	Description
`ServiceIntegrationResourceArn`	The resource ARN of the integrated service.

Service Metrics

The `AWS/States` namespace includes the following metrics for the Step Functions service.

Metric	Description
`ThrottledEvents`	The count of requests that have been throttled.
`ProvisionedBucketSize`	The count of available requests per second.
`ProvisionedRefillRate`	The count of requests per second that are allowed into the bucket.
`ConsumedCapacity`	The count of requests per second.

Dimension for Step Functions Service Metrics

Dimension	Description
`StateTransition`	Filters data to show State Transitions metrics.

API Metrics

The `AWS/States` namespace includes the following metrics for the Step Functions API.

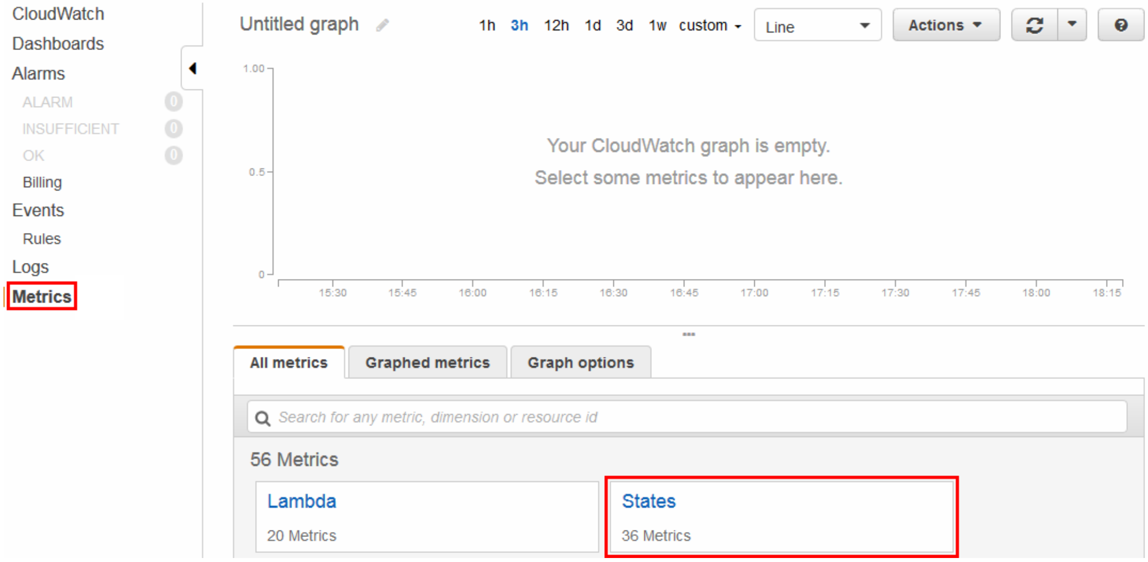
Metric	Description
`ThrottledEvents`	The count of requests that have been throttled.
`ProvisionedBucketSize`	The count of available requests per second.
`ProvisionedRefillRate`	The count of requests per second that are allowed into the bucket.
`ConsumedCapacity`	The count of requests per second.

Dimension for Step Functions API Metrics

Dimension	Description
`APIName`	Filters data to an API of the specified API name.

Viewing Metrics for Step Functions

1. Sign in to the AWS Management Console and open the CloudWatch console.
2. Choose **Metrics**, and on the **All Metrics** tab, choose **States**.

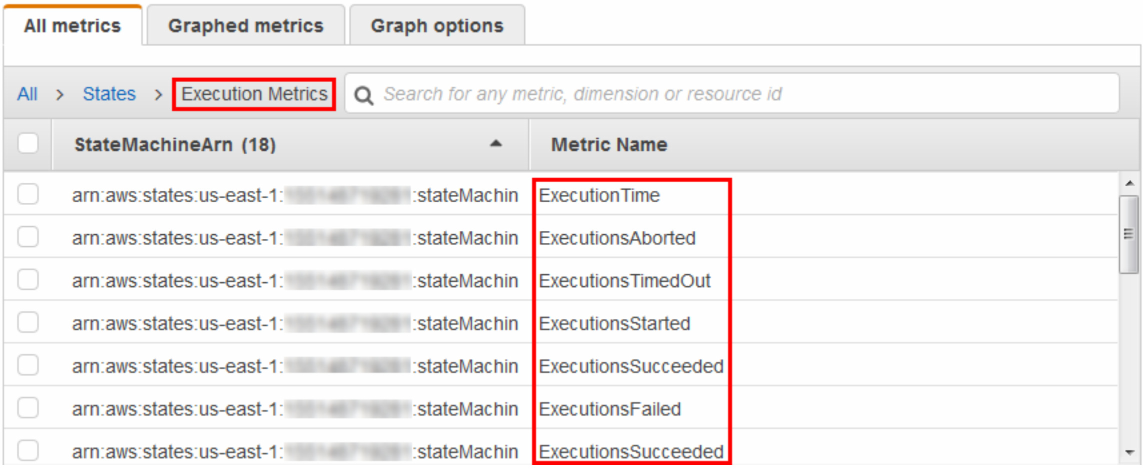


Your CloudWatch graph is empty.
Select some metrics to appear here.

If you ran any executions recently, you will see up to three types of metrics:

- **Execution Metrics**
- **Activity Function Metrics**
- **Lambda Function Metrics**

3. Choose a metric type to see a list of metrics.


All metrics
Graphed metrics
Graph options
All > States > Execution Metrics
Search for any metric, dimension or resource id

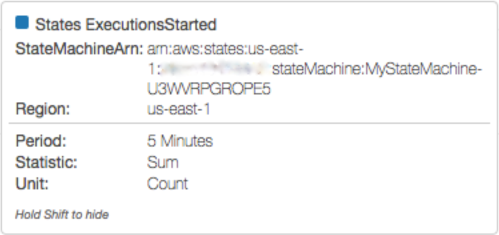
☐	StateMachineArn (18)	Metric Name
☐	arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionTime
☐	arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsAborted
☐	arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsTimedOut
☐	arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsStarted
☐	arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsSucceeded
☐	arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsFailed
☐	arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsSucceeded

- To sort your metrics by **Metric Name** or **StateMachineArn**, use the column headings.
- To view graphs for a metric, choose the box next to the metric on the list. You can change the graph parameters using the time range controls above the graph view.

  You can choose custom time ranges using relative or absolute values (specific days and times). You can also use the dropdown list to display values as lines, stacked areas, or numbers (values).
- To view the details about a graph, hover over the metric color code that appears below the graph.

  🟦 ExecutionsAborted 🟧 ExecutionsStarted 🟩 ExecutionsSucceeded 🟥 ExecutionsTimedOut

  The metric's details are displayed.



States ExecutionsStarted
StateMachineArn: arn:aws:states:us-east-1:[redacted]:stateMachine:MyStateMachine-U3WVRPGROPE5
Region: us-east-1
Period: 5 Minutes
Statistic: Sum
Unit: Count
Hold Shift to hide

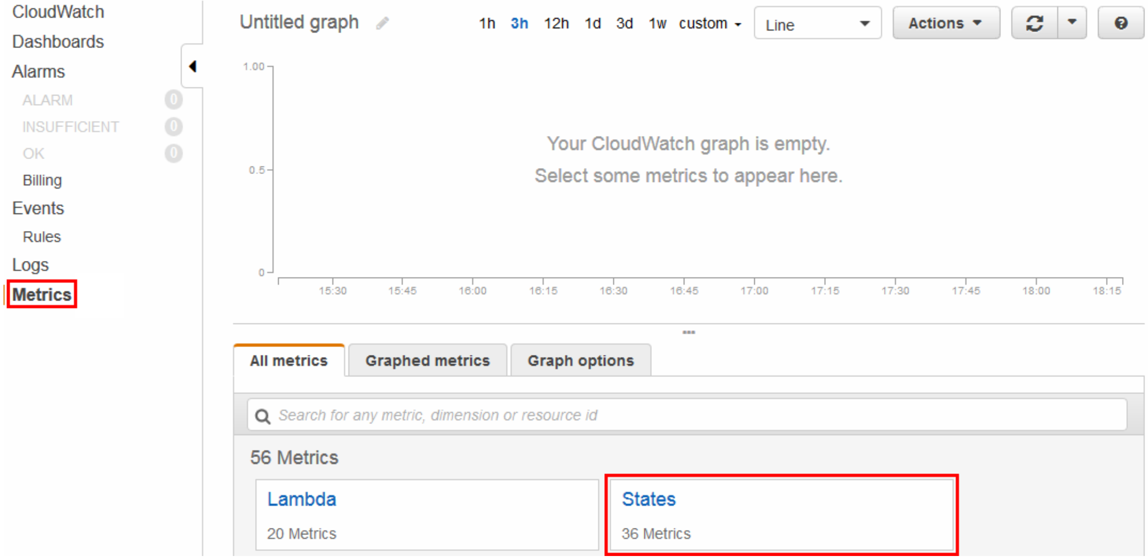
For more information about working with CloudWatch metrics, see Using Amazon CloudWatch Metrics in the *Amazon CloudWatch User Guide*.

Setting Alarms for Step Functions

You can use CloudWatch alarms to perform actions. For example, if you want to know when an alarm threshold is reached, you can set an alarm to send a notification to an Amazon SNS topic or to send an email when the `StateMachinesFailed` metric rises above a certain threshold.

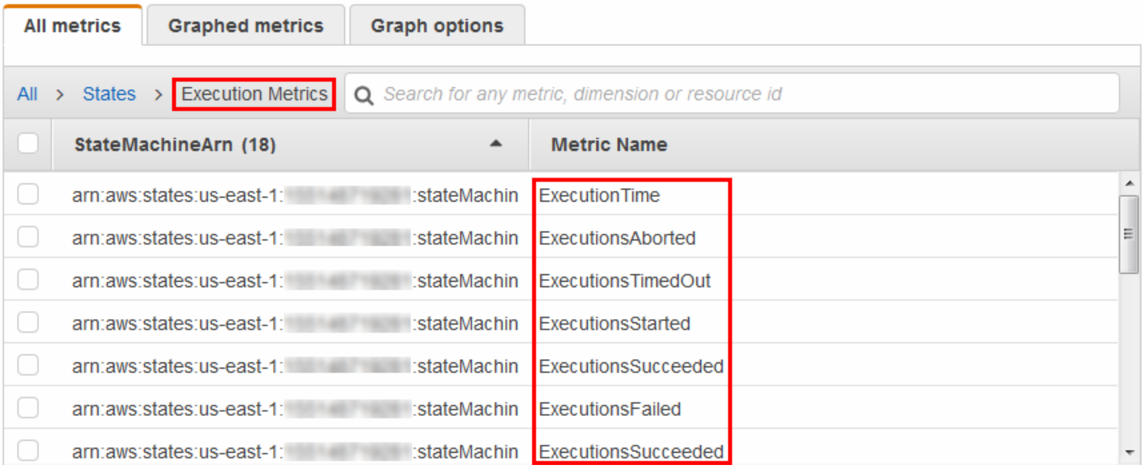
To set an alarm on a metric

1. Sign in to the AWS Management Console and open the CloudWatch console.
2. Choose **Metrics**, and on the **All Metrics** tab, choose **States**.



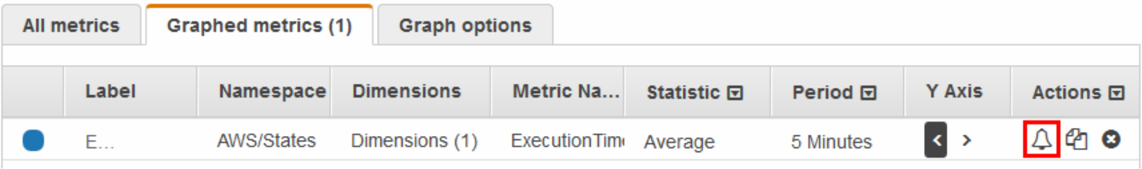
   If you ran any executions recently, you will see up to three types of metrics:

   - **Execution Metrics**
   - **Activity Function Metrics**
   - **Lambda Function Metrics**
3. Choose a metric type to see a list of metrics.


All metrics
Graphed metrics
Graph options
All > States > Execution Metrics
Search for any metric, dimension or resource id

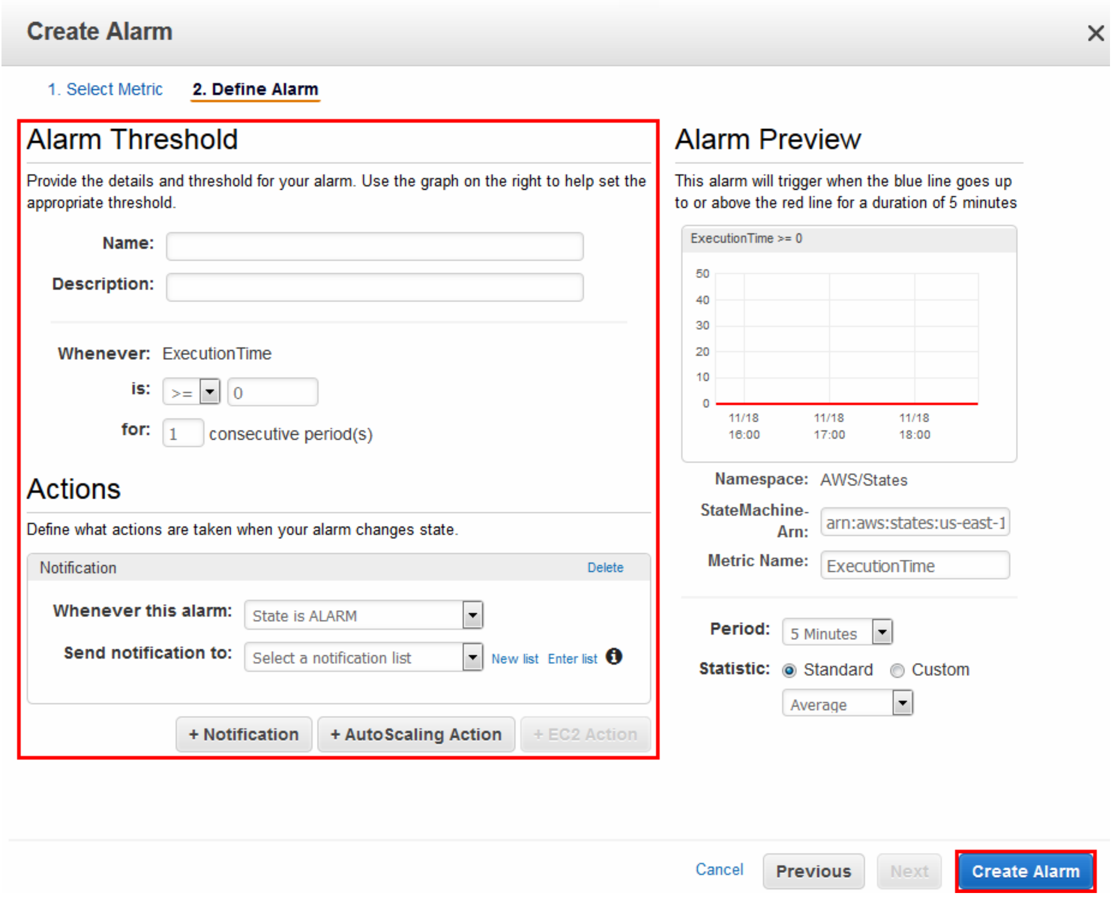
□ StateMachineArn (18)	Metric Name
□ arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionTime
□ arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsAborted
□ arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsTimedOut
□ arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsStarted
□ arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsSucceeded
□ arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsFailed
□ arn:aws:states:us-east-1:[redacted]:stateMachin	ExecutionsSucceeded

4. Choose a metric, and then choose **Graphed metrics**.
5. Choose 🔔 next to a metric on the list.



	Label	Namespace	Dimensions	Metric Na...	Statistic	Period	Y Axis	Actions
●	E...	AWS/States	Dimensions (1)	ExecutionTim	Average	5 Minutes	< >	🔔

The **Create Alarm** page is displayed.



Create Alarm

1. Select Metric 2. Define Alarm

Alarm Threshold

Provide the details and threshold for your alarm. Use the graph on the right to help set the appropriate threshold.

Name:

Description:

Whenever: ExecutionTime

is: >= 0

for: 1 consecutive period(s)

Actions

Define what actions are taken when your alarm changes state.

Notification

Whenever this alarm: State is ALARM

Send notification to: Select a notification list

Alarm Preview

This alarm will trigger when the blue line goes up to or above the red line for a duration of 5 minutes

Namespace: AWS/States

StateMachine-Arn: arn:aws:states:us-east-1

Metric Name: ExecutionTime

Period: 5 Minutes

Statistic: ● Standard ○ Custom

Average

6. Enter the values for the **Alarm threshold** and **Actions**, and then choose **Create Alarm**.

For more information about setting and using CloudWatch alarms, see Creating Amazon CloudWatch Alarms in the *Amazon CloudWatch User Guide*.

CloudWatch Events For Step Functions Execution Status Changes

Amazon CloudWatch Events is an AWS service that enables you to respond to state changes in an AWS resource. You can use AWS Step Functions with CloudWatch Events in two ways.

You can configure Step Functions to emit CloudWatch Events when an execution status changes. This enables you to monitor your workflows without having to constantly poll using the `DescribeExecution` API. Based on changes in state machine executions you can use a CloudWatch Events target to start new state machine executions, call AWS Lambda functions, publish messages to Amazon Simple Notification Service (Amazon SNS) topics, and more.

You can also configure a Step Functions state machine as a target in CloudWatch Events. This enables you to trigger an execution of a Step Functions workflow in response to an event from another AWS service.

For more information, see the Amazon CloudWatch Events User Guide.

Topics

Step Functions Event Examples

Step Functions supports sending events to CloudWatch Events when the following changes occur.

Examples

In each case, the `detail` section in the event data provides the same information as the `DescribeExecution` API. The `status` field indicates the status of the execution at the time the event was sent, one of `RUNNING`, `SUCCEEDED`, `FAILED`, `TIMED_OUT`, or `ABORTED` depending on the event emitted.

Execution Started

```
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "012345678912",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-1",
    "resources": [
      "arn:aws:states:us-east-1:012345678912:execution:state-machine-name:execution-name"
    ],
    "detail": {
        "executionArn": "arn:aws:states:us-east-1:012345678912:execution:state-machine-
name:execution-name",
        "stateMachineArn": "arn:aws:states:us-east-1:012345678912:stateMachine:state-
machine",
        "name": "execution-name",
        "status": "RUNNING",
        "startDate": 1551225271984,
        "stopDate":  null,
        "input": "{}",
        "output": null
    }
}
```

Execution Succeeded

```
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "012345678912",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-1",
    "resources": [
      "arn:aws:states:us-east-1:012345678912:execution:state-machine-name:execution-name"
    ],
    "detail": {
        "executionArn": "arn:aws:states:us-east-1:012345678912:execution:state-machine-
name:execution-name",
        "stateMachineArn": "arn:aws:states:us-east-1:012345678912:stateMachine:state-
machine",
        "name": "execution-name",
        "status": "SUCCEEDED",
        "startDate": 1547148840101,
        "stopDate": 1547148840122,
        "input": "{}",
        "output": "\"Hello World!\""
    }
}
```

Execution Failed

```
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "012345678912",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-1",
    "resources": [
      "arn:aws:states:us-east-1:012345678912:execution:state-machine-name:execution-name"
    ],
    "detail": {
        "executionArn": "arn:aws:states:us-east-1:012345678912:execution:state-machine-
name:execution-name",
        "stateMachineArn": "arn:aws:states:us-east-1:012345678912:stateMachine:state-
machine",
        "name": "execution-name",
        "status": "FAILED",
        "startDate": 1551225146847,
        "stopDate": 1551225151881,
        "input": "{}",
        "output": null
    }
}
```

Execution Timed Out

```
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "012345678912",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-1",
    "resources": [
      "arn:aws:states:us-east-1:012345678912:execution:state-machine-name:execution-name"
    ],
    "detail": {
        "executionArn": "arn:aws:states:us-east-1:012345678912:execution:state-machine-
name:execution-name",
        "stateMachineArn": "arn:aws:states:us-east-1:012345678912:stateMachine:state-
machine",
        "name": "execution-name",
        "status": "TIMED_OUT",
        "startDate": 1551224926156,
        "stopDate": 1551224927157,
        "input": "{}",
        "output": null
    }
}
```

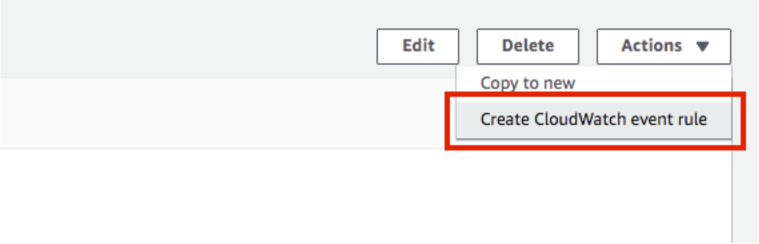
Execution Aborted

```
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "012345678912",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-1",
    "resources": [
      "arn:aws:states:us-east-1:012345678912:execution:state-machine-name:execution-name"
    ],
    "detail": {
        "executionArn": "arn:aws:states:us-east-1:012345678912:execution:state-machine-
name:execution-name",
        "stateMachineArn": "arn:aws:states:us-east-1:012345678912:stateMachine:state-
machine",
        "name": "execution-name",
        "status": "ABORTED",
        "startDate": 1551225014968,
        "stopDate": 1551225017576,
        "input": "{}",
        "output": null
    }
}
```

Routing a Step Functions Event to a CloudWatch Events Target in the CloudWatch Events Console

Create a CloudWatch Events rule for Step Functions events

1. On the **Details** page of a state machine, choose **Actions**, **Create CloudWatch event rule**.



   Alternatively, open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/. In the navigation pane, choose **Events**, **Rules**.

2. Choose **Create rule**.

3. Under **Event Source**, ensure **Event Pattern** is selected.
4. For **Service Name**, choose **Step Functions**.
5. For **Event Type**, choose **Step Functions Execution Status Change**.
6. Optionally, configure a specific status, state machine ARN, or execution ARN.
7. Under **Targets**, choose **Add target** and select a target from the drop-down. For example, you could launch a Lambda function, or start an execution of a Step Functions state machine.
8. Add detailed information for your specific target.
9. Choose **Configure details**. On the **Configure rule details** page, enter a **Name** and **Description**.
10. Choose **Create rule**.

Logging Step Functions Using AWS CloudTrail

Step Functions is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Step Functions. CloudTrail captures all API calls for Step Functions as events, including calls from the Step Functions console and from code calls to the Step Functions APIs.

If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon Simple Storage Service (Amazon S3) bucket, including events for Step Functions. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**.

Using the information collected by CloudTrail, you can determine the request that was made to Step Functions, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Step Functions Information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Step Functions, that activity is recorded in a CloudTrail event with other AWS service events in **Event history**.

You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for Step Functions, create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act on the event data collected in CloudTrail logs.

For more information, see:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or IAM user credentials

- Whether the request was made with temporary security credentials for a role or federated user
- Whether the request was made by another AWS service

For more information, see the CloudTrail userIdentity Element.

Example: Step Functions Log File Entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files are not an ordered stack trace of the public API calls, so they don't appear in any specific order.

CreateActivity

The following example shows a CloudTrail log entry that demonstrates the `CreateActivity` action.

```
{
    "eventVersion": "1.04",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDAJYDLDBVBI4EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/test-user",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "test-user"
    },
    "eventTime": "2016-10-28T01:17:56Z",
    "eventSource": "states.amazonaws.com",
    "eventName": "CreateActivity",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "10.61.88.189",
    "userAgent": "Coral/Netty",
    "requestParameters": {
        "name":
 "OtherActivityPrefix.2016-10-27-18-16-56.894c791e-2ced-4cf4-8523-376469410c25"
    },
    "responseElements": {
        "activityArn": "arn:aws:states:us-
east-1:123456789012:activity:OtherActivityPrefix.2016-10-27-18-16-56.894c791e-2ced-4cf4-8523-376469410c
        "creationDate": "Oct 28, 2016 1:17:56 AM"
    },
    "requestID": "37c67602-9cac-11e6-aed5-5b57d226e9ef",
    "eventID": "dc3becef-d06d-49bf-bc93-9b76b5f00774",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

CreateStateMachine

The following example shows a CloudTrail log entry that demonstrates the `CreateStateMachine` action.

```
{
    "eventVersion": "1.04",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDAJYDLDBVBI4EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/test-user",
        "accountId": "123456789012",
        "accessKeyId": "AKIAJL5C75K4ZEXAMPLE",
        "userName": "test-user"
    },
    "eventTime": "2016-10-28T01:18:07Z",
    "eventSource": "states.amazonaws.com",
    "eventName": "CreateStateMachine",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "10.61.88.189",
    "userAgent": "Coral/Netty",
    "requestParameters": {
        "name": "testUser.2016-10-27-18-17-06.bd144e18-0437-476e-9bb",
        "roleArn": "arn:aws:iam::123456789012:role/graphene/tests/graphene-execution-role",
        "definition": "{   \"StartAt\": \"SinglePass\",    \"States\": {        \"SinglePass
\": {           \"Type\": \"Pass\",           \"End\": true       }   }}"
    },
    "responseElements": {
        "stateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:testUser.2016-10-27-18-17-06.bd144e18-0437-476e-9bb",
        "creationDate": "Oct 28, 2016 1:18:07 AM"
    },
    "requestID": "3da6370c-9cac-11e6-aed5-5b57d226e9ef",
    "eventID": "84a0441d-fa06-4691-a60a-aab9e46d689c",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

DeleteActivity

The following example shows a CloudTrail log entry that demonstrates the `DeleteActivity` action.

```
{
      "eventVersion": "1.04",
      "userIdentity": {
          "type": "IAMUser",
          "principalId": "AIDAJYDLDBVBI4EXAMPLE",
          "arn": "arn:aws:iam::123456789012:user/test-user",
          "accountId": "123456789012",
          "accessKeyId": "AKIAJL5C75K4ZEXAMPLE",
          "userName": "test-user"
      },
      "eventTime": "2016-10-28T01:18:27Z",
      "eventSource": "states.amazonaws.com",
      "eventName": "DeleteActivity",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "10.61.88.189",
      "userAgent": "Coral/Netty",
      "requestParameters": {
          "activityArn": "arn:aws:states:us-
east-1:123456789012:activity:testUser.2016-10-27-18-11-35.f017c391-9363-481a-be2e-"
      },
      "responseElements": null,
      "requestID": "490374ea-9cac-11e6-aed5-5b57d226e9ef",
      "eventID": "e5eb9a3d-13bc-4fa1-9531-232d1914d263",
      "eventType": "AwsApiCall",
      "recipientAccountId": "123456789012"
  }
```

DeleteStateMachine

The following example shows a CloudTrail log entry that demonstrates the `DeleteStateMachine` action.

```
{
    "eventVersion": "1.04",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDAJABK5MNKNAEXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/graphene/tests/test-user",
        "accountId": "123456789012",
        "accessKeyId": "AKIAJA2ELRVCPEXAMPLE",
        "userName": "test-user"
    },
    "eventTime": "2016-10-28T01:17:37Z",
    "eventSource": "states.amazonaws.com",
    "eventName": "DeleteStateMachine",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "10.61.88.189",
    "userAgent": "Coral/Netty",
    "errorCode": "AccessDenied",
    "errorMessage": "User: arn:aws:iam::123456789012:user/graphene/tests/test-user is
 not authorized to perform: states:DeleteStateMachine on resource: arn:aws:states:us-
east-1:123456789012:stateMachine:testUser.2016-10-27-18-16-38.ec6e261f-1323-4555-9fa",
    "requestParameters": null,
    "responseElements": null,
    "requestID": "2cf23f3c-9cac-11e6-aed5-5b57d226e9ef",
    "eventID": "4a622d5c-e9cf-4051-90f2-4cdb69792cd8",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

StartExecution

The following example shows a CloudTrail log entry that demonstrates the `StartExecution` action.

```
{
    "eventVersion": "1.04",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDAJYDLDBVBI4EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/test-user",
        "accountId": "123456789012",
        "accessKeyId": "AKIAJL5C75K4ZEXAMPLE",
        "userName": "test-user"
    },
    "eventTime": "2016-10-28T01:17:25Z",
    "eventSource": "states.amazonaws.com",
    "eventName": "StartExecution",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "10.61.88.189",
    "userAgent": "Coral/Netty",
    "requestParameters": {
        "input": "{}",
        "stateMachineArn": "arn:aws:states:us-
east-1:123456789012:stateMachine:testUser.2016-10-27-18-16-26.482bea32-560f-4a36-bd",
        "name": "testUser.2016-10-27-18-16-26.6e229586-3698-4ce5-8d"
    },
    "responseElements": {
        "startDate": "Oct 28, 2016 1:17:25 AM",
        "executionArn": "arn:aws:states:us-
east-1:123456789012:execution:testUser.2016-10-27-18-16-26.482bea32-560f-4a36-
bd:testUser.2016-10-27-18-16-26.6e229586-3698-4ce5-8d"
    },
    "requestID": "264c6f08-9cac-11e6-aed5-5b57d226e9ef",
    "eventID": "30a20c8e-a3a1-4b07-9139-cd9cd73b5eb8",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

StopExecution

The following example shows a CloudTrail log entry that demonstrates the `StopExecution` action.

```
{
    "eventVersion": "1.04",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDAJYDLDBVBI4EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/test-user",
        "accountId": "123456789012",
        "accessKeyId": "AKIAJL5C75K4ZEXAMPLE",
        "userName": "test-user"
    },
    "eventTime": "2016-10-28T01:18:20Z",
    "eventSource": "states.amazonaws.com",
    "eventName": "StopExecution",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "10.61.88.189",
    "userAgent": "Coral/Netty",
    "requestParameters": {
        "executionArn": "arn:aws:states:us-
east-1:123456789012:execution:testUser.2016-10-27-18-17-00.337b3344-83:testUser.2016-10-27-18-17-00.3a0
    },
    "responseElements": {
        "stopDate": "Oct 28, 2016 1:18:20 AM"
    },
    "requestID": "4567625b-9cac-11e6-aed5-5b57d226e9ef",
    "eventID": "e658c743-c537-459a-aea7-dafb83c18c53",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

Security in AWS Step Functions

This section provides information about AWS Step Functions security and authentication.

Topics

Step Functions uses IAM to control access to other AWS services and resources. For an overview of how IAM works, see Overview of Access Management in the *IAM User Guide*. For an overview of security credentials, see AWS Security Credentials in the Amazon Web Services General Reference.

Authentication

You can access AWS as any of the following types of identities:

- **AWS account root user** – When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.
- **IAM user** – An IAM user is an identity within your AWS account that has specific custom permissions (for example, permissions to create a state machine in Step Functions). You can use an IAM user name and password to sign in to secure AWS webpages like the AWS Management Console, AWS Discussion Forums, or the AWS Support Center.

  In addition to a user name and password, you can also generate access keys for each user. You can use these keys when you access AWS services programmatically, either through one of the several SDKs or by using the AWS Command Line Interface (CLI). The SDK and CLI tools use the access keys to cryptographically sign your request. If you don't use AWS tools, you must sign the request yourself. Step Functions supports *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 Signing Process in the *AWS General Reference*.
- **IAM role** – An IAM role is an IAM identity that you can create in your account that has specific permissions. An IAM role is similar to an IAM user in that it is an AWS identity with permissions policies that determine what the identity can and cannot do in AWS. However, instead of being uniquely associated with one person, a role is intended to be assumable by anyone who needs it. Also, a role does not have standard long-term credentials such as a password or access keys associated with it. Instead, when you assume a role, it provides you with temporary security credentials for your role session. IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated Users and Roles in the *IAM User Guide*.

- **AWS service access** – A service role is an IAM role that a service assumes to perform actions in your account on your behalf. When you set up some AWS service environments, you must define a role for the service to assume. This service role must include all the permissions that are required for the service to access the AWS resources that it needs. Service roles vary from service to service, but many allow you to choose your permissions as long as you meet the documented requirements for that service. Service roles provide access only within your account and cannot be used to grant access to services in other accounts. You can create, modify, and delete a service role from within IAM. For example, you can create a role that allows Amazon Redshift to access an Amazon S3 bucket on your behalf and then load data from that bucket into an Amazon Redshift cluster. For more information, see Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide*.

- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM Role to Grant Permissions to Applications Running on Amazon EC2 Instances in the *IAM User Guide*.

Data Protection in AWS Step Functions

AWS Step Functions conforms to the AWS shared responsibility model, which includes regulations and guidelines for data protection. AWS is responsible for protecting the global infrastructure that runs all the AWS services. AWS maintains control over data hosted on this infrastructure, including the security configuration controls for handling customer content and personal data. AWS customers and APN partners, acting either as data controllers or data processors, are responsible for any personal data that they put in the AWS Cloud.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM), so that each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources.

- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.

We strongly recommend that you never put sensitive identifying information, such as your customers' account numbers, into free-form fields such as a **Name** field. This includes when you work with Step

Functions or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into Step Functions or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, don't include credentials information in the URL to validate your request to that server.

For more information about data protection, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*. AWS Step Functions creates an AWS managed CMK for you and uses it on your behalf. For more information about CMKs, see Customer Master Keys (CMKs) in the AWS Key Management Service Developer Guide.

Encryption in AWS Step Functions

Encryption at Rest

Step Functions always encrypts your data at rest. Data in AWS Step Functions is encrypted at rest using transparent server-side encryption. This helps reduce the operational burden and complexity involved in protecting sensitive data. With encryption at rest, you can build security-sensitive applications that meet encryption compliance and regulatory requirements

Encryption in Transit

Step Functions encrypts data in transit between the service and other integrated AWS services (see AWS Service Integrations (p. 138)). All data that passes between Step Functions and integrated services is encrypted using Transport layer Security (TLS).

Tag-based Policies

Step Functions supports policies based on tags. For instance, you could restrict access to all Step Functions resources that include a tag with the key `environment` and the value `production`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "states:TagResource",
                "states:UntagResource",
                "states:DeleteActivity",
                "states:DeleteStateMachine",
                "states:StopExecution"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {"aws:ResourceTag/environment": "production"}
            }
        }
    ]
}
```

This policy will `Deny` the ability to delete state machines or activities, stop executions, and add or delete new tags for all resources that have been tagged as `environment/production`.

For more information on tagging, see:

- Tagging (p. 164)

- Controlling Access Using IAM Tags

Creating IAM Roles for AWS Step Functions

AWS Step Functions can execute code and access AWS resources (such as invoking an AWS Lambda function). To maintain security, you must grant Step Functions access to those resources by using an IAM role.

The Tutorials (p. 17) in this guide enable you to take advantage of automatically generated IAM roles that are valid for the AWS Region in which you create the state machine. To create your own IAM role for a state machine, follow the steps in this section.

In this example, you create an IAM role with permission to invoke a Lambda function.

To create a role for Step Functions

1. Sign in to the IAM console, and then choose **Roles**, **Create role**.
2. On the **Select type of trusted entity** page, under **AWS service**, select **Step Functions** from the list and then choose **Next: Permissions**.
3. On the **Attached permissions policy** page, choose **Next: Review**.
4. On the **Review** page, enter `StepFunctionsLambdaRole` for **Role Name**, and then choose **Create role**.

   The IAM role appears in the list of roles.

For more information about IAM permissions and policies, see Access Management in the *IAM User Guide*.

Attach an Inline Policy

Step Functions can control other services directly in a task state. Attach inline policies to allow Step Functions to access the API actions of the services you need to control.

1. Open the IAM console, choose **Roles**, search for your Step Functions role, and select that role.
2. Select **Add inline policy**.
3. Use the **Visual editor** or the **JSON** tab to create policies for your role.

For more information about how AWS Step Functions can control other AWS services, see: AWS Service Integrations (p. 138).

> **Note**
> For examples of IAM policies created by the Step Functions console, see IAM Policies for Integrated Services (p. 234)

Creating Granular IAM Permissions for Non-Admin Users

The default managed policies in IAM, such as `ReadOnly`, don't fully cover all types of Step Functions permissions. This section describes these different types of permissions and provides some example configurations.

AWS Step Functions has four categories of permissions. Depending on what access you want to provide to a user, you can control access by using permissions in these categories.

Service-Level Permissions (p. 232)

Apply to components of the API that don't act on a specific resource.

State Machine-Level Permissions (p. 232)

Apply to all API components that act on a specific state machine.

Execution-Level Permissions (p. 233)

Apply to all API components that act on a specific execution.

Activity-Level Permissions (p. 233)

Apply to all API components that act on a specific activity or on a particular instance of an activity.

Service-Level Permissions

This permission level applies to all API actions that don't act on a specific resource. These include `CreateStateMachine`, `CreateActivity`, `ListStateMachines`, and `ListActivities`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:ListStateMachines",
        "states:ListActivities",
        "states:CreateStateMachine",
        "states:CreateActivity"
      ],
      "Resource": [
        "arn:aws:states:*:*:*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam:::role/my-execution-role"
      ]
    }
  ]
}
```

State Machine-Level Permissions

This permission level applies to all API actions that act on a specific state machine. These API operations require the Amazon Resource Name (ARN) of the state machine as part of the request, such as `DeleteStateMachine`, `DescribeStateMachine`, `StartExecution`, and `ListExecutions`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeStateMachine",
        "states:StartExecution",
        "states:DeleteStateMachine",
        "states:ListExecutions",
        "states:UpdateStateMachine"
      ],
      "Resource": [
        "arn:aws:states:*:*:stateMachine:StateMachinePrefix*"
      ]
    }
  ]
}
```

Execution-Level Permissions

This permission level applies to all the API actions that act on a specific execution. These API operations require the ARN of the execution as part of the request, such as `DescribeExecution`, `GetExecutionHistory`, and `StopExecution`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeExecution",
        "states:DescribeStateMachineForExecution",
        "states:GetExecutionHistory",
        "states:StopExecution"
      ],
      "Resource": [
        "arn:aws:states:*:*:execution:*:ExecutionPrefix*"
      ]
    }
  ]
}
```

Activity-Level Permissions

This permission level applies to all the API actions that act on a specific activity or on a particular instance of it. These API operations require the ARN of the activity or the token of the instance as part of the request, such as `DeleteActivity`, `DescribeActivity`, `GetActivityTask`, `SendTaskSuccess`, `SendTaskFailure`, and `SendTaskHeartbeat`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeActivity",
        "states:DeleteActivity",
        "states:GetActivityTask",
        "states:SendTaskSuccess",
        "states:SendTaskFailure",
        "states:SendTaskHeartbeat"
      ],
      "Resource": [
    "arn:aws:states:*:*:activity:ActivityPrefix*"
   ]
  }
 ]
}
```

IAM Policies for Integrated Services

When you create a state machine in the AWS Step Functions console, Step Functions produces an AWS Identity and Access Management (IAM) policy based on the resources used in your state machine definition.

These examples show how Step Functions generates an IAM policy based on your state machine definition. Items in the example code such as *[[resourceName]]* are replaced with the static resources listed in your state machine definition. If you have multiple static resources, there will be an entry for each in the IAM role.

Dynamic vs. Static Resources

Static resources are defined directly in the task state of your state machine. When you include the information about the API actions you call directly in your task states, Step Functions creates an IAM role for only those resources.

Dynamic resources are those that are passed in to your state input, and accessed using a Path (see, Paths (p. 116)). If you are passing dynamic resources to your task, Step Functions will create a more privileged policy that specifies: `"Resource": "*"`.

Request Response vs Run a Job/Wait for Callback IAM Policies

For connections using the Run a Job (p. 139) or Wait for Callback (p. 139) patterns (those ending in `.sync` or `.waitForTaskToken`), additional permissions are needed to monitor and receive a response from the API actions of connected services. The related policies need more permissions than non-synchronous connected services. See Service Integration Patterns (p. 139) for information about synchronous connections.

> **Note**
> Review these templates to understand how Step Functions creates your IAM policies, and as an example of how to manually create IAM policies for Step Functions when working with other AWS services. For more information about Step Functions service integration, see AWS Service Integrations (p. 138).

Topics

- Activities or no Tasks (p. 249)

IAM Policy for AWS Lambda

AWS Step Functions generates an IAM policy based on your state machine definition. For a state machine with two Lambda task states that call `function1` and `function2`, a policy with `lambda:Invoke` permissions for the two functions must be used.

For example:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "arn:aws:lambda:[[region]]:[[accountId]]:function:[[function1]]",
                "arn:aws:lambda:[[region]]:[[accountId]]:function:[[function2]]"
            ]
        }
    ]
}
```

AWS Batch

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for Integrated Services (p. 234)
- Service Integration Patterns (p. 139)

AWS Batch doesn't support resource-level access control. You must use `"Resource": "*"`.

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "batch:SubmitJob",
                "batch:DescribeJobs",
                "batch:TerminateJob"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
               "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForBatchJobsRule"
            ]
        }
    ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "batch:SubmitJob"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon DynamoDB

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for Integrated Services (p. 234)
- Service Integration Patterns (p. 139)

Static resources:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:[[region]]:[[accountId]]:table/[[tableName]]"
            ]
        }
    ]
}
```

Dynamic resources:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon Elastic Container Service/AWS Fargate

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for Integrated Services (p. 234)
- Service Integration Patterns (p. 139)

Because the value for `TaskId` is not known until the task is submitted, Step Functions creates a more privileged `"Resource": "*"` policy.

> **Note**
> You can only stop Amazon ECS tasks that were started by Step Functions, despite the `"*"` IAM policy.

Run a Job (.sync)

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask"
            ],
            "Resource": [
                "arn:aws:ecs:[[region]]:
[[accountId]]:task-definition/[[taskDefinition]]"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ecs:StopTask",
                "ecs:DescribeTasks"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
               "arn:aws:events:[[region]]:
[[accountId]]:rule/StepFunctionsGetEventsForECSTaskRule"
            ]
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask",
                "ecs:StopTask",
                "ecs:DescribeTasks"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
               "arn:aws:events:[[region]]:
[[accountId]]:rule/StepFunctionsGetEventsForECSTaskRule"
            ]
        }
    ]
}
```

Request Response and Callback (.waitForTaskToken)

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask"
            ],
            "Resource": [
                "arn:aws:ecs:[[region]]:
[[accountId]]:task-definition/[[taskDefinition]]"
            ]
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon Simple Notification Service

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for Integrated Services (p. 234)
- Service Integration Patterns (p. 139)

Static resources:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": [
                "arn:aws:sns:[[region]]:[[accountId]]:[[topicName]]"
            ]
        }
    ]
}
```

Resources based on a Path, or publishing to `TargetArn` or `PhoneNumber`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon Simple Queue Service

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for Integrated Services (p. 234)
- Service Integration Patterns (p. 139)

Static resources:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sqs:SendMessage"
            ],
            "Resource": [
                "arn:aws:sqs:[[region]]:[[accountId]]:[[queueName]]"
            ]
        }
    ]
}
```

Dynamic resources:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sqs:SendMessage"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS Glue

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for Integrated Services (p. 234)
- Service Integration Patterns (p. 139)

AWS Glue does not have resource-based control.

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "glue:StartJobRun",
                "glue:GetJobRun",
                "glue:GetJobRuns",
                "glue:BatchStopJobRun"
            ],
            "Resource": "*"
        }
    ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "glue:StartJobRun"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon SageMaker

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for Integrated Services (p. 234)
- Service Integration Patterns (p. 139)

> **Note**
> For these examples, *[[roleArn]]* refers to the Amazon Resource Name (ARN) of the IAM role that Amazon SageMaker uses to access model artifacts and docker images for deployment on ML compute instances, or for batch transform jobs. For more information, see Amazon SageMaker Roles.

Topics

CreateTrainingJob

Static resources:

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",
        "sagemaker:DescribeTrainingJob",
        "sagemaker:StopTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:training-job/[[trainingJobName]]*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "[[roleArn]]"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForSageMakerTrainingJobsRule"
      ]
    }
  ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:training-job/[[trainingJobName]]*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "[[roleArn]]"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}
```

Dynamic resources:

.sync or .waitForTaskToken

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",
        "sagemaker:DescribeTrainingJob",
        "sagemaker:StopTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:training-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "[[roleArn]]"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForSageMakerTrainingJobsRule"
      ]
    }
  ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:training-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "[[roleArn]]"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}
```

`CreateTransformJob`

Note

AWS Step Functions will not automatically create a policy for `CreateTransformJob` when you create a state machine that integrates with Amazon SageMaker. You must attach an inline policy to the created role based on one of the following IAM examples.

Static resources:

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTransformJob",
        "sagemaker:DescribeTransformJob",
        "sagemaker:StopTransformJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:transform-
job/[[transformJobName]]*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "[[roleArn]]"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForSageMakerTransformJobsRule"
      ]
    }
  ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTransformJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:transform-
job/[[transformJobName]]*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "[[roleArn]]"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}
```

Dynamic resources:

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTransformJob",
        "sagemaker:DescribeTransformJob",
        "sagemaker:StopTransformJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:transform-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "[[roleArn]]"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForSageMakerTransformJobsRule"
      ]
    }
  ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTransformJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:[[region]]:[[accountId]]:transform-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "[[roleArn]]"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}
```

AWS Step Functions

For a state machine that calls `StartExecution` for a single nested workflow execution, use an IAM policy that limits permissions to that state machine.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "states:StartExecution"
            ],
            "Resource": [
                "arn:aws:states:[[region]]:[[accountId]]:stateMachine:[[stateMachineName]]"
            ]
        }
    ]
}
```

For more information, see:

- Service Integrations (p. 138)
- Pass Parameters to a Service API (p. 144)
- AWS Step Functions (p. 162)

Synchronous

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "states:StartExecution"
            ],
            "Resource": [
                "arn:aws:states:[[region]]:[[accountId]]:stateMachine:
[[stateMachineName]]"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "states:DescribeExecution",
                "states:StopExecution"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
               "arn:aws:events:[[region]]:[[accountId]]:rule/
StepFunctionsGetEventsForStepFunctionsExecutionRule"
            ]
        }
    ]
}
```

Asynchronous

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "states:StartExecution"
            ],
            "Resource": [

 "arn:aws:states:[[region]]:[[accountId]]:stateMachine:[[stateMachineName]]"
            ]
        }
    ]
}
```

For more information on nested workflow executions, see: Start Workflow Executions From a Task State (p. 130).

Activities or no Tasks

For a state machine that has only Activity tasks, or no tasks at all, use an IAM policy that denies access to all actions and resources.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": "*",
            "Resource": "*"
        }
    ]
}
```

For more information on using activity tasks, see: Activities (p. 94).

Related Information

The following table lists related resources that you might find useful as you work with this service.

Resource	Description
AWS Step Functions API Reference	Descriptions of API actions, parameters, and data types and a list of errors that the service returns.
AWS Step Functions Command Line Reference	Descriptions of the AWS CLI commands that you can use to work with AWS Step Functions.
Product information for Step Functions	The primary webpage for information about Step Functions.
Discussion Forums	A community-based forum for developers to discuss technical questions related to Step Functions and other AWS services.
AWS Premium Support Information	The primary webpage for information about AWS Premium Support, a one-on-one, fast-response support channel to help you build and run applications on AWS infrastructure services.

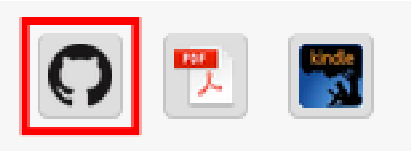
Document History

This section lists major changes to the *AWS Step Functions Developer Guide*.

Change	Description	Date Changed
Update	AWS Step Functions has released the AWS Step Functions Data Science SDK. For more information, see the following. • Project on Github • SDK Documentation • The following Example Notebooks, which are available in the Amazon SageMaker console and the related GitHub project. • `hello_world_workflow.ipynb` • `machine_learning_workflow_abalone.ipynb` • `training_pipeline_pytorch_mnist.ipynb`	November 7, 2019
Update	Step Functions now supports more API actions for Amazon SageMaker, and includes two new sample projects to demonstrate the functionality. For more information, see the following. • Manage Amazon SageMaker with Step Functions (p. 156) • AWS Service Integrations (p. 138) • Train a Machine Learning Model (p. 192) • Tune a Machine Learning Model (p. 196)	October 3, 2019
New feature	Step Functions includes a new `Map` state type. You can use a `Map` state to run a series of steps for each item in a JSON array in the input. For more information, see the following. • Map State (p. 109) • Map State Example (p. 110) • Map State Tutorial (p. 81) • Map State Input and Output Processing (p. 112) • ItemsPath (p. 119) • Context Object Data For Map States (p. 129) • Map State Sample Project (p. 187)	September 18, 2019
New feature	Step Functions supports starting new workflow executions by calling `StartExecution` as an integrated service API. See: • Start Workflow Executions From a Task State (p. 130) • Manage AWS Step Functions Executions as an Integrated Service (p. 162) • AWS Service Integrations (p. 138) • IAM Policies for Starting Step Functions Workflow Executions (p. 248)	August 12, 2019
New feature	Step Functions includes the ability to pass a task token to integrated services, and pause the execution until that task token is returned with `SendTaskSuccess` or `SendTaskFailure`. See:	May 23, 2019

Change	Description	Date Changed
	- Service Integration Patterns (p. 139) - Wait for a Callback with the Task Token (p. 140) - Callback Pattern Example (p. 182) - Supported AWS Service Integrations for Step Functions (p. 146) - Deploy an Example Human Approval Project (p. 71) - Service Integration Metrics (p. 214) Step Functions now provides a way to access dynamic information about your current execution directly in the `"Parameters"` field of a state definition. See: - The Context Object (p. 127) - Pass Context Object Nodes as Parameters (p. 145)	
New feature	Step Functions supports CloudWatch Events for execution status changes, see: - CloudWatch Events For Step Functions Execution Status Changes (p. 219) - Amazon CloudWatch Events User Guide	May 8, 2019
New feature	Step Functions supports IAM permissions using tags. For more information, see: - Tagging (p. 164) - Tag-based Policies (p. 230)	March 5, 2019
New feature	Step Functions Local is now available. You can run Step Functions on your local machine for testing and development. Step Functions Local is available for download as either a Java application, or as a Docker image. See Setting Up Step Functions Local (Downloadable Version) (p. 5).	February 4, 2019
New feature	AWS Step Functions is now available in the Beijing and Ningxia regions. See Supported Regions (p. 1).	January 15, 2018
New feature	Step Functions supports resource tagging to help track your cost allocation. You can tag state machines on the **Details** page, or through API actions. See Tagging (p. 164).	January 7, 2019
New feature	AWS Step Functions is now available in the EU (Paris), and South America (São Paulo) regions. See Supported Regions (p. 1).	December 13, 2018
New feature	AWS Step Functions is now available the EU (Stockholm) region. See Supported Regions (p. 1) for a list of supported regions.	December 12, 2018
New feature	You can now easily configure and generate a state definition for integrated services when editing your state definition. For more information, see: - Code Snippets (p. 145) - Using Code Snippets (p. 67)	December 10, 2018

Change	Description	Date Changed
New feature	Step Functions now integrates with some AWS services. You can now directly call and pass parameters to the API of these integrated services from a task state in the Amazon States Language. For more information, see: • AWS Service Integrations (p. 138) • Pass Parameters to a Service API (p. 144) • Supported AWS Service Integrations for Step Functions (p. 146)	November 29, 2018
Update	Improved the description of `TimeoutSeconds` and `HeartbeatSeconds` in the documentation for task states. See Task (p. 91).	October 24, 2018
Update	Improved the description for the *Maximum execution history size* limit and provided a link to the related best practices topic. • Limits Related to State Machine Executions (p. 207) • Avoid Reaching the History Limit (p. 204)	October 17, 2018
Update	Added a new tutorial to the AWS Step Functions documentation: See Starting a State Machine Execution in Response to Amazon S3 Events (p. 42).	September 25, 2018
Update	Removed the entry *Maximum executions displayed in Step Functions console* from the limits documentation. See Limits (p. 206).	September 13, 2018
Update	Added a best practices topic to the AWS Step Functions documentation on improving latency when polling for activity tasks. See Avoid Latency When Polling for Activity Tasks (p. 205).	August 30, 2018
Update	Improved the AWS Step Functions topic on activities and activity workers. See Activities (p. 94).	August 29, 2018
Update	Improved the AWS Step Functions topic on CloudTrail integration. See Logging Step Functions Using AWS CloudTrail (p. 223).	August 7, 2018
Update	Added JSON examples to AWS CloudFormation tutorial. See Creating a Lambda State Machine Using AWS CloudFormation (p. 23).	June 23, 2018
Update	Added a new topic on handling Lambda service errors. See Handle Lambda Service Exceptions (p. 204).	June 20, 2018
New feature	AWS Step Functions is now available the Asia Pacific (Mumbai) region. See Supported Regions (p. 1) for a list of supported regions.	June 28, 2018
New feature	AWS Step Functions is now available the AWS GovCloud (US-West) region. See Supported Regions (p. 1) for a list of supported regions. For information about using Step Functions in the AWS GovCloud (US-West) Region, see AWS GovCloud (US) Endpoints.	June 28, 2018
Update	Improved documentation on error handling for `Parallel` states. See Error Handling (p. 109).	June 20, 2018
Update	Improved documentation about Input and Output processing in Step Functions. Learn how to use `InputPath`, `ResultPath`, and `OutputPath` to control the flow of JSON through your workflows, states, and tasks. See: • Input and Output Processing in Step Functions (p. 115) • ResultPath (p. 120)	June 7, 2018

Change	Description	Date Changed
Update	Improved code examples for parallel states. See Parallel (p. 107).	June 4, 2018
New feature	You can now monitor API and Service metrics in CloudWatch. See Monitoring Step Functions Using CloudWatch (p. 211).	May 25, 2018
Update	`StartExecution`, `StopExecution`, and `StateTransition` now have increased throttling limits in the following regions: • US East (N. Virginia) • US West (Oregon) • EU (Ireland) For more information see Limits (p. 206).	May 16, 2018
New feature	AWS Step Functions is now available the US West (N. California) and Asia Pacific (Seoul) regions. See Supported Regions (p. 1) for a list of supported regions.	May 5, 2018
Update	Updated procedures and images to match changes to the interface.	April 25, 2018
Update	Added a new tutorial that shows how to start a new execution to continue your work. See Continue as a New Execution (p. 59). This tutorial describes a design pattern that can help avoid some service limitations. See Avoid Reaching the History Limit (p. 204).	April 19, 2018
Update	Improved introduction to states documentation by adding conceptual information about state machines. See States (p. 86).	March 9, 2018
Update	In addition to HTML, PDF, and Kindle, the AWS Step Functions Developer Guide is available on GitHub. To leave feedback, choose the GitHub icon in the upper right-hand corner. 	March 2, 2018
Update	Added a topic describing other resources relating to Step Functions. See Related Information (p. 250).	February 20, 2018

Change	Description	Date Changed
New feature	- When you create a new state machine, you must acknowledge that AWS Step Functions will create an IAM role which allows access to your Lambda functions. - Updated the following tutorials to reflect the minor changes in the state machine creation workflow: - Getting Started (p. 13) - Creating a Lambda State Machine (p. 19) - Creating an Activity State Machine (p. 31) - Handling Error Conditions Using a State Machine (p. 35) - Iterating a Loop Using Lambda (p. 52)	February 19, 2018
Update	Added a topic that describes an example activity worker written in Ruby. This implementation can be used to create a Ruby activity worker directly, or as a design pattern for creating an activity worker in another language. See Example Activity Worker in Ruby (p. 96).	February 6, 2018
Update	Added a new tutorial describing a design pattern that uses a Lambda function to iterate a count. See Creating a Lambda State Machine (p. 19).	January 31, 2018
Update	Updated content on IAM permissions to include `DescribeStateMachineForExecution` and `UpdateStateMachine` APIs. See Creating Granular IAM Permissions for Non-Admin Users (p. 231).	January 26, 2018
Update	Added newly available regions: Canada (Central), Asia Pacific (Singapore). See Supported Regions (p. 1).	January 25, 2018
Update	Updated tutorials and procedures to reflect that IAM allows you to select *Step Functions* as a role.	January 24, 2018
Update	Added a new *Best Practices* topic that suggests not passing large payloads between states. See Use ARNs Instead of Passing Large Payloads (p. 203).	January 23, 2018
Update	Corrected procedures to match updated interface for creating a state machine: - Getting Started (p. 13) - Creating a Lambda State Machine (p. 19) - Creating an Activity State Machine (p. 31) - Handling Error Conditions Using a State Machine (p. 35)	January 17, 2018

Change	Description	Date Changed
New Feature	You can use *Sample Projects* to quickly provision state machines and all related AWS resources. See Sample Projects (p. 168), Available sample projects include: - Poll for Job Status (Lambda, AWS Batch) (p. 178) - Task Timer (p. 180) **Note** These sample projects and related documentation replace tutorials that described implementing the same functionality.	January 11, 2018
Update	Added a *Best Practices* section that includes information on avoiding stuck executions. See Best Practices for Step Functions (p. 203).	January 5, 2018
Update	Added a note on how retries can affect pricing: **Note** Retries are treated as state transitions. For information about how state transitions affect billing, see Step Functions Pricing.	December 8, 2017
Update	Added information related to resource names: **Note** Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.	December 6, 2017
Update	Improved security overview information and added a topic on granular IAM permissions. See Security in AWS Step Functions (p. 228) and Creating Granular IAM Permissions for Non-Admin Users (p. 231).	November 27, 2017
New Feature	You can update an existing state machine. See Update a state machine (p. 15).	November 15, 2017
Update	Added a note to clarify `Lambda.Unknown` errors and linked to the Lambda documentation in the following sections: - Error Names (p. 132) - Step 4: Create a State Machine with a Catch Field (p. 37) **Note** Unhandled errors in Lambda are reported as `Lambda.Unknown` in the error output. These include out-of-memory errors, function timeouts, and hitting the concurrent Lambda invoke limit. You can match on `Lambda.Unknown`, `States.ALL`, or `States.TaskFailed` to handle these errors. When Lambda hits the invocation limit, the error is `Lambda.TooManyRequestsException`. For more information about Lambda `Handled` and `Unhandled` errors, see `FunctionError` in the AWS Lambda Developer Guide.	October 17, 2017

Change	Description	Date Changed
Update	Corrected and clarified IAM instructions and updated the screenshots in all tutorials (p. 17).	October 11, 2017
Update	• Added new screenshots for state machine execution results to reflect changes in the Step Functions console. Rewrote the Lambda instructions in the following tutorials to reflect changes in the Lambda console: • Creating a Lambda State Machine (p. 19) • Creating a Job Status Poller • Creating a Task Timer • Handling Error Conditions Using a State Machine (p. 35) • Corrected and clarified information about creating state machines in the following sections: • Getting Started (p. 13) • Creating an Activity State Machine (p. 31)	October 6, 2017
Update	Rewrote the IAM instructions in the following sections to reflect changes in the IAM console: • Creating IAM Roles for AWS Step Functions (p. 231) • Creating a Lambda State Machine (p. 19) • Creating a Job Status Poller • Creating a Task Timer • Handling Error Conditions Using a State Machine (p. 35) • Creating a Step Functions API Using API Gateway (p. 48)	October 5, 2017
Update	Rewrote the State Machine Data (p. 113) section.	September 28, 2017
New feature	The limits related to API action throttling (p. 208) are increased for all regions where Step Functions is available.	September 18, 2017
Update	• Corrected and clarified information about starting new executions in all tutorials. • Corrected and clarified information in the Limits Related to Accounts (p. 207) section.	September 14, 2017
Update	Corrected and clarified information in the Templates (p. 163) section.	September 1, 2017
Update	Rewrote the following tutorials to reflect changes in the Lambda console: • Creating a Lambda State Machine (p. 19) • Handling Error Conditions Using a State Machine (p. 35) • Creating a Job Status Poller	August 28, 2017
New feature	Step Functions is available in EU (London).	August 23, 2017
New feature	The visual workflows of state machines let you zoom in, zoom out, and center the graph.	August 21, 2017

<table>
<tr><th>Change</th><th>Description</th><th>Date Changed</th></tr>
<tr><td>New feature</td><td>

Important
An execution can't use the name of another execution for 90 days.

When you make multiple `StartExecution` calls with the same name, the new execution doesn't run and the following rules apply.

<table>
<tr><th>Input Type</th><th colspan="2">Execution State</th></tr>
<tr><td></td><td>Open</td><td>Closed</td></tr>
<tr><td>Identical</td><td><code>Success</code></td><td><code>ExecutionAlreadyExists</code></td></tr>
<tr><td>Different</td><td><code>ExecutionAlreadyExists</code></td><td><code>ExecutionAlreadyExists</code></td></tr>
</table>

For more information, see the `name` request parameter of the `StartExecution` API action in the *AWS Step Functions API Reference*.

</td><td>August 18, 2017</td></tr>
<tr><td>Update</td><td>Added information about an alternative way of passing the state machine ARN to the Creating a Step Functions API Using API Gateway (p. 48) tutorial.</td><td>August 17, 2017</td></tr>
<tr><td>Update</td><td>Added the new <i>Creating a Job Status Poller</i> tutorial.</td><td>August 10, 2017</td></tr>
<tr><td>New feature</td><td>

- Step Functions emits the `ExecutionThrottled` CloudWatch metric. For more information, see Monitoring Step Functions Using CloudWatch (p. 211).
- Added the Limits Related to State Throttling (p. 210) section.

</td><td>August 3, 2017</td></tr>
<tr><td>Update</td><td>Updated the instructions in the Step 1: Create an IAM Role for API Gateway (p. 48) section.</td><td>July 18, 2017</td></tr>
<tr><td>Update</td><td>Corrected and clarified information in the Choice (p. 102) section.</td><td>June 23, 2017</td></tr>
<tr><td>Update</td><td>

Added information about using resources under other AWS accounts to the following tutorials:

- Creating a Lambda State Machine (p. 19)
- Creating a Lambda State Machine Using AWS CloudFormation (p. 23)
- Creating an Activity State Machine (p. 31)
- Handling Error Conditions Using a State Machine (p. 35)

</td><td>June 22, 2017</td></tr>
<tr><td>Update</td><td>

Corrected and clarified information in the following sections:

- Getting Started (p. 13)
- Handling Error Conditions Using a State Machine (p. 35)
- States (p. 86)
- Error Handling (p. 132)

</td><td>June 21, 2017</td></tr>
<tr><td>Update</td><td>Rewrote all tutorials to match the Step Functions console refresh.</td><td>June 12, 2017</td></tr>
<tr><td>New feature</td><td>Step Functions is available in Asia Pacific (Sydney).</td><td>June 8, 2017</td></tr>
</table>

Change	Description	Date Changed
Update	Restructured the Amazon States Language (p. 87) section.	June 7, 2017
Update	Corrected and clarified information in the Creating an Activity State Machine (p. 31) section.	June 6, 2017
Update	Corrected the code examples in the Examples Using Retry and Using Catch (p. 135) section.	June 5, 2017
Update	Restructured this guide using AWS documentation standards.	May 31, 2017
Update	Corrected and clarified information in the Parallel (p. 107) section.	May 25, 2017
Update	Merged the Paths and Filters sections into the Input and Output Processing in Step Functions (p. 115) section.	May 24, 2017
Update	Corrected and clarified information in the Templates (p. 163) section.	May 16, 2017
Update	Corrected and clarified information in the Monitoring Step Functions Using CloudWatch (p. 211) section.	May 15, 2017
Update	Updated the `GreeterActivities.java` worker code in the Creating an Activity State Machine (p. 31) tutorial.	May 9, 2017
Update	Added an introductory video to the What Is AWS Step Functions? (p. 1) section.	April 19, 2017
Update	Corrected and clarified information in the following tutorials: • Getting Started (p. 13) • Creating a Lambda State Machine (p. 19) • Creating an Activity State Machine (p. 31) • Handling Error Conditions Using a State Machine (p. 35)	April 19, 2017
Update	Added information about Lambda templates to the Creating a Lambda State Machine (p. 19) and Handling Error Conditions Using a State Machine (p. 35) tutorials.	April 6, 2017
Update	Changed the "Maximum input or result data size" limit to "Maximum input or result data size for a task, state, or execution" (32,768 characters). For more information, see Limits Related to Task Executions (p. 207).	March 31, 2017
New feature	• Step Functions supports executing state machines by setting Step Functions as Amazon CloudWatch Events targets. • Added the Periodically Start a State Machine Execution Using CloudWatch Events (p. 40) tutorial.	March 21, 2017
New feature	• Step Functions allows Lambda function error handling as the preferred error handling method. • Updated the Handling Error Conditions Using a State Machine (p. 35) tutorial and the Error Handling (p. 132) section.	March 16, 2017

Change	Description	Date Changed
New feature	Step Functions is available in EU (Frankfurt).	March 7, 2017
Update	Reorganized the topics in the table of contents and updated the following tutorials: • Getting Started (p. 13) • Creating a Lambda State Machine (p. 19) • Creating an Activity State Machine (p. 31) • Handling Error Conditions Using a State Machine (p. 35)	February 23, 2017
New feature	• The **State Machines** page of the Step Functions console includes the **Copy to New** and **Delete** buttons. • Updated the screenshots to match the console changes.	February 23, 2017
New feature	• Step Functions supports creating APIs using API Gateway. • Added the Creating a Step Functions API Using API Gateway (p. 48) tutorial.	February 14, 2017
New feature	• Step Functions supports integration with AWS CloudFormation. • Added the Creating a Lambda State Machine Using AWS CloudFormation (p. 23) tutorial.	February 10, 2017
Update	Clarified the current behavior of the `ResultPath` and `OutputPath` fields in relation to `Parallel` states.	February 6, 2017
Update	• Clarified state machine naming restrictions in tutorials. • Corrected some code examples.	January 5, 2017
Update	Updated Lambda function examples to use the latest programming model.	December 9, 2016
New feature	The initial release of Step Functions.	December 1, 2016

AWS Glossary

For the latest AWS terminology, see the AWS Glossary in the *AWS General Reference*.