
AWS Step Functions

Developer Guide



AWS Step Functions: Developer Guide

Copyright © 2017 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

Table of Contents

What is AWS Step Functions?	1
Overview of Step Functions	1
Supported Regions	1
About Amazon Web Services	2
Getting Started	3
Step 1: Creating the State Machine	3
To create the state machine	3
Step 2: Starting a new execution	4
To start a new execution	5
Next Steps	5
Tutorials	7
Development Options	7
Step Functions Console	7
AWS SDKs	7
HTTPS Service API	8
Development Environments	8
Endpoints	8
AWS CLI	8
Creating a Lambda State Machine	9
Step 1: Creating an IAM Role for Lambda	9
Step 2: Creating a Lambda Function	9
Step 3: Testing the Lambda Function	11
Step 4: Creating a State Machine	11
Step 5: Starting a New Execution	13
Creating a Lambda State Machine Using AWS CloudFormation	15
Step 1: Setting Up Your AWS CloudFormation Template	15
Step 2: Using the AWS CloudFormation Template to Create a Lambda State Machine	17
Step 3: Starting a State Machine Execution	19
Creating an Activity State Machine	20
Step 1: Creating a New Activity	21
Step 2: Creating a State Machine	21
Step 3: Implementing a Worker	22
Step 4: Starting an Execution	24
Step 5: Running and Stopping the Worker	24
Starting a State Machine Execution Using CloudWatch Events	25
Step 1: Creating a State Machine	25
Step 2: Creating a CloudWatch Events Rule	25
Handling Error Conditions Using a State Machine	27
Step 1: Creating an IAM Role for Lambda	28
Step 2: Creating a Lambda Function That Fails	28
Step 3: Testing the Lambda Function	29
Step 4: Creating a State Machine with a <code>catch</code> Field	30
Step 5: Starting a New Execution	32
Creating a Step Functions API Using API Gateway	33
Step 1: Creating an IAM Role for API Gateway	34
Step 2: Creating your API Gateway API	34
Step 3: Testing and Deploying the API Gateway API	37
How Step Functions Works	39
Blueprints	39
States	40
Tasks	40
Activities	41
Creating an Activity	41
Writing a Worker	41

Transitions	42
State Machine Data	42
Data Format	43
State Machine Input/Output	43
State Input/Output	43
Executions	45
Error Handling	45
Error Names	46
Retrying After an Error	46
Fallback States	48
Examples Using Retry and Using Catch	49
Amazon States Language	53
Example Amazon States Language Specification	53
State Machine Structure	54
States	55
Common State Fields	56
Pass	56
Task	57
Choice	60
Wait	62
Succeed	63
Fail	63
Parallel	64
Input and Output Processing	66
Paths	66
Reference Paths	66
Errors	68
Error Representation	69
Retrying After an Error	69
Fallback States	71
Limits	73
General Limits	73
Limits Related to Accounts	74
Limits Related to State Machine Executions	74
Limits Related to Task Executions	74
Limits Related to API Action Throttling	75
Monitoring and Logging	76
Monitoring Step Functions Using CloudWatch	76
Metrics that Report a Time Interval	77
Metrics that Report a Count	77
State Machine Metrics	77
Viewing Metrics for Step Functions	79
Setting Alarms for Step Functions	80
Logging Step Functions using AWS CloudTrail	82
Step Functions Information in CloudTrail	82
Understanding Step Functions Log File Entries	83
Security	87
Creating IAM Roles for Use with AWS Step Functions	87
Steps to Create a Role for Use with Step Functions	87
Document History	89

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 provides 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 on the AWS cloud, on your own servers, or on any system that has access to AWS. Step Functions can be accessed and used with 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 \(p. 40\)](#) and [state machines \(p. 40\)](#).
- You define state machines using the JSON-based [Amazon States Language \(p. 53\)](#).
- 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.

Supported Regions

Currently, Step Functions is supported only in the following regions:

- US East (Ohio)
- US East (N. Virginia)
- US West (Oregon)

- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- EU (Frankfurt)
- EU (Ireland)

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**.

Getting Started

This tutorial introduces you to the basics of working with AWS Step Functions. In this tutorial you'll create a simple, independently-running state machine using a `PASS` state. The `PASS` state is the simplest state which represents a *no-op* (an instruction with no operation).

Topics

- [Step 1: Creating the State Machine \(p. 3\)](#)
- [Step 2: Starting a new execution \(p. 4\)](#)
- [Next Steps \(p. 5\)](#)

Step 1: Creating the State Machine

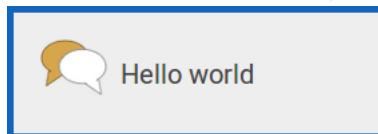
Step Functions offers various predefined state machines in the form of *blueprints*. Create your first state machine using the **Hello World** blueprint.

To create the state machine

1. Log in to the [Step Functions console](#) and choose **Get Started**.

The **Create a state machine** page is displayed.

2. Choose the **Hello world** blueprint.



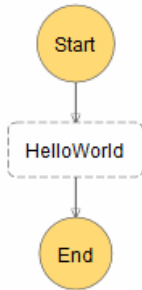
Step Functions fills in the name of the state machine automatically and populates the **Code** pane with the Amazon States Language description of the state machine.


```
{
  "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  
}
```

This JSON text defines a single `Pass` state named `HelloWorld`. For more information, see [State Machine Structure \(p. 54\)](#).

3. Verify that the **Visual Workflow** pane displays the following graph of your state machine structure. The graph helps you verify that your Amazon States Language code describes the state machine correctly.



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

4. Choose **Create State Machine**.

The **IAM role for your state machine executions** dialog box is displayed. Step Functions creates and selects an IAM role automatically.

IAM role for your state machine executions

Select an IAM role for your tasks (See IAM documentation [here](#))

StatesExecutionRole-us-east-1

If you don't have an IAM role, you can [create one manually here](#)

Cancel OK

Note

If you delete the IAM role that Step Functions creates, Step Functions can't re-create it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later. For more information about creating an IAM role manually, see [Creating IAM Roles for Use with AWS Step Functions \(p. 87\)](#).

5. Choose **OK**.

The state machine is created and an acknowledgement page is displayed.

Step 2: Starting a new execution

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

To start a new execution

1. On the acknowledgement page, choose **New execution**.

The **New execution** page is displayed.

HelloStateMachine

New execution

```
Enter your execution id here  
1 {  
2   "Comment": "Insert your JSON here"  
3 }
```

2. (Optional) To help identify your execution, you can enter an ID for it. To specify the ID, use the **Enter your execution id here** text box. If you don't enter an ID, Step Functions generates a unique ID automatically.
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, 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 Arn: arn:aws:states:us-east-1:123456789012:execution:HelloWorld:9d824c71-5fd9-a33e-4de5-9b6604c1144c

9d824c71-5fd9-a33e-4de5-9b6604c1144c ✓

Graph Code

■ Success ■ Failed ■ Cancelled ■ In Progress

```
graph TD; Start((Start)) --> HelloWorld[HelloWorld]; HelloWorld --> End((End));
```

Execution Details

Info Input Output

output: "Hello World!"

Step Details

ID	Type	Timestamp
▶ 1	ExecutionStarted	Jun 9, 2017 12:31:10 PM
▶ 2	PassStateEntered	Jun 9, 2017 12:31:10 PM
▶ 3	PassStateExited	Jun 9, 2017 12:31:10 PM
▶ 4	ExecutionSucceeded	Jun 9, 2017 12:31:10 PM

Next Steps

Now that you've created a simple state machine using a `PASS` state, you might want to try the following:

- [Create a Lambda state machine. \(p. 9\)](#)
- [Create a Lambda state machine using AWS CloudFormation. \(p. 15\)](#)
- [Create an activity state machine. \(p. 20\)](#)
- [Start a state machine using Amazon CloudWatch Events. \(p. 25\)](#)
- [Handle error conditions using a state machine. \(p. 27\)](#)
- [Create a Step Functions API using Amazon API Gateway. \(p. 33\)](#)

Tutorials

The following tutorials will help you get started working with AWS Step Functions. To complete these tutorials, you'll need an AWS account. If you haven't yet signed up for AWS, navigate to <http://aws.amazon.com/> and choose **Sign In to the Console**.

Topics

- [Development Options \(p. 7\)](#)
- [Creating a Lambda State Machine \(p. 9\)](#)
- [Creating a Lambda State Machine Using AWS CloudFormation \(p. 15\)](#)
- [Creating an Activity State Machine \(p. 20\)](#)
- [Starting a State Machine Execution Using CloudWatch Events \(p. 25\)](#)
- [Handling Error Conditions Using a State Machine \(p. 27\)](#)
- [Creating a Step Functions API Using API Gateway \(p. 33\)](#)

Development Options

You can implement your Step Functions state machines in a number of ways.

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 taking advantage of Lambda to supply code for your tasks and the Step Functions console to define your state machine using Amazon States Language.

The [Creating a Lambda State Machine \(p. 9\)](#) tutorial uses this technique to create a simple state machine, execute it, and view its results.

AWS SDKs

Step Functions is supported by SDKs for Java, .NET, Ruby, PHP, Python (boto 3), JavaScript, Go, and C++, providing 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 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 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 on API actions, see the [AWS Step Functions API Reference](#).

Development Environments

You must set up a development environment 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 Development, you should also install the AWS Toolkit for Eclipse. This Eclipse plug-in adds features useful for AWS development.

If your programming language requires a run-time 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 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, being 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 program using the AWS Step Functions API actions. For example, you can use the AWS CLI to create a new state machine and then list your state machines.

The Step Functions commands in AWS CLI allow 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 Command Line Interface 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.

Creating a Lambda State Machine

In this tutorial you'll create an AWS Step Functions state machine that uses a AWS Lambda function to implement a `Task` state. A `Task` state is a simple state that 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: Creating an IAM Role for Lambda \(p. 9\)](#)
- [Step 2: Creating a Lambda Function \(p. 9\)](#)
- [Step 3: Testing the Lambda Function \(p. 11\)](#)
- [Step 4: Creating a State Machine \(p. 11\)](#)
- [Step 5: Starting a New Execution \(p. 13\)](#)

Step 1: Creating an IAM Role for Lambda

Both Lambda and 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.

In the [Getting Started \(p. 3\)](#) tutorial, this was done automatically for Step Functions: an IAM role was created when you created the state machine. However, Lambda requires you to assign an IAM role when you create a Lambda function in the same way Step Functions required you to assign an IAM role when you create a state machine.

To create a role for use with Lambda

1. Log in to the [IAM console](#) and choose **Roles, Create New Role**.
2. On the **Set Role Name** page, type the **Role Name** and then choose **Next Step**.
3. On the **Select Role Type** page, select **AWS Lambda** from the list.

Note

The role is automatically provided with a trust relationship that allows Lambda to use the role.

4. On the **Attach Policy** page, choose **Next Step**.
5. On the **Review** page, choose **Create Role**.

The role appears in the list of roles in the IAM console.

Step 2: Creating a Lambda Function

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

To create the Lambda function

1. Log in to the [Lambda console](#).
2. If this is your first Lambda function, choose **Get Started Now**. Otherwise, choose **Create a Lambda function**.
3. On the **Select blueprint** page, choose the **Blank Function** blueprint.

Select blueprint ?

Blueprints are sample configurations of event sources and Lambda functions. Choose a blueprint that best aligns with your desired scenario and customize as needed, or skip this step if you want to author a Lambda function and configure an event source separately. Except where otherwise noted, blueprints are licensed under [CC0](#).

Welcome to AWS Lambda! You can get started on creating your first Lambda function by choosing one of the blueprints below. ✕

Select runtime ▼⌵<< < Viewing 1-3 of 3 > >>

Blank Function

Configure your function from scratch. Define the trigger and deploy your code by stepping through our wizard.

custom

step-functions-error-python

An AWS Lambda function that throws an error. AWS Step Functions state machines can be configured to handle

python2.7 ⬇

step-functions-error

An AWS Lambda function that throws an error. AWS Step Functions state machines can be configured to handle

nodejs6.10 ⬇

4. On the **Configure triggers** page, choose **Next**.
5. On the **Configure function** page, configure your Lambda function:
 - a. For **Name**, type `HelloFunction`.
 - b. For **Description**, type `Say "Hello" to someone`.
 - c. From the **Runtime** list, select **Node.js 6.10**.
6. In the **Lambda function code** section, copy the following code for the Lambda function:

```
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 will add input data for this function later, when you [start a new execution](#) (p. 13). The `callback` method returns the assembled greeting from your function.

7. In the **Lambda function handler and role** section, select **Choose an existing role** from the **Role** list and then select the Lambda role that you have created earlier from the **Existing role** list.

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. In the meanwhile, verify that `lambda.amazonaws.com` has access to the role. To verify or edit the trust relationship for your role, revisit [Step 1: Creating an IAM Role for Lambda](#) (p. 9).

You can use the **Timeout** value in the **Advanced settings** to specify how long your function can take to execute before failing.

8. Choose **Next**, review your function, and then choose **Create function**.
9. When your Lambda function is created, note its Amazon Resource Name (ARN), displayed in the upper-right corner of the page. For example:

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

Step 3: Testing the Lambda Function

Test your Lambda function to see it in operation.

To test the Lambda function

1. On your Lambda function page, choose **Test**.

The **Input test event** dialog box is displayed.

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 **Save and test** to test your Lambda function using the new data.

The results of the test are displayed at the bottom of the page.

✓ Execution result: succeeded ([logs](#))

The area below shows the result returned by your function execution. [Learn more](#) about returning results from your function.

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

Summary	Log output
Code SHA-256 [REDACTED]	The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.
Request ID b813fa91-b698-11e6-b5c0-1d70c448e0b0	<pre>START RequestId: b813fa91-b698-11e6-b5c0-1d70c448e0b0 Version: \$LATEST END RequestId: b813fa91-b698-11e6-b5c0-1d70c448e0b0 REPORT RequestId: b813fa91-b698-11e6-b5c0-1d70c448e0b0 Duration: 3.61 ms Billed</pre>
Duration 3.61 ms	
Billed duration 100 ms	

Step 4: Creating a State Machine

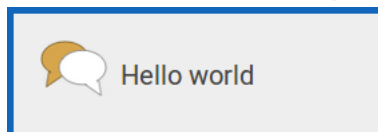
Use the [Step Functions console](#) to create a state machine with a single `Task` state. Add a reference to your Lambda function in the `Task` state. The Lambda function is invoked when an execution of the state machine reaches the `Task` state.

To create the state machine

1. Log in to the [Step Functions console](#) and choose **Get Started**.

The **Create a state machine** page is displayed.

2. Choose the **Hello World** blueprint.



3. In the box below **Name your state machine**, type a name, for example `LambdaStateMachine`.

Note

State machine names must be 1-80 characters in length, must be unique for your account and region, and must not contain any of the following:

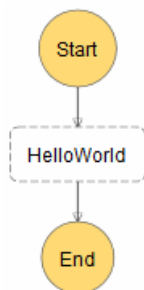
- Whitespace
- Whitespace characters (? *)
- Bracket characters (< > { } [])
- Special characters (: ; , \ | ^ ~ \$ # % & ` ")
- Control characters (\\u0000 - \\u001f or \\u007f - \\u009f).


4. In the **Code** pane, replace the example JSON text with the following, using a `Task` state and the ARN of the Lambda function that you have created earlier in the `Resource` field, 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 \(p. 54\)](#).

5. Verify that the **Visual Workflow** pane displays the following graph of your state machine structure. The graph helps you verify 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 **Create State Machine**.

The **IAM role for your state machine executions** dialog box is displayed. Step Functions creates and selects an IAM role automatically.

IAM role for your state machine executions ✕

Select an IAM role for your tasks (See IAM documentation [here](#))

StatesExecutionRole-us-east-1 ▼

If you don't have an IAM role, you can [create one manually here](#)

Cancel OK

Note

If you delete the IAM role that Step Functions creates, Step Functions can't re-create it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later. For more information about creating an IAM role manually, see [Creating IAM Roles for Use with AWS Step Functions \(p. 87\)](#).

7. Choose **OK**.

The state machine is created and an acknowledgement page is displayed.

Step 5: Starting a New Execution

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

To start a new execution

1. In the [Step Functions console](#), choose **Dashboard** and then choose the name of the state machine that you have created earlier.

Dashboard

State Machines

Delete

Copy to New

Create a State Machine

Search for state machines			
State Machines (3)			
<input type="radio"/> HelloStateMachine arn:aws:states:...:stateMachine:HelloStateMachine	Running 0	Finished 1	Errors 0
<input type="radio"/> LambdaStateMachine arn:aws:states:...:stateMachine:LambdaStateMachine	Running 0	Finished 1	Errors 0
<input type="radio"/> RetryStateMachine arn:aws:states:...:stateMachine:RetryStateMachine	Running 0	Finished 0	Errors 0

2. On the state machine's detail page, choose **New execution**.

The **New execution** page is displayed.

LambdaStateMachine

New execution

```
Enter your execution id here  
1 {  
2   "Comment": "Insert your JSON here"  
3 }
```

3. (Optional) To help identify your execution, you can enter an ID for it. To specify the ID, use the **Enter your execution id here** text box. If you don't enter an ID, Step Functions generates a unique ID automatically.
4. 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.


5. Choose **Start Execution**.

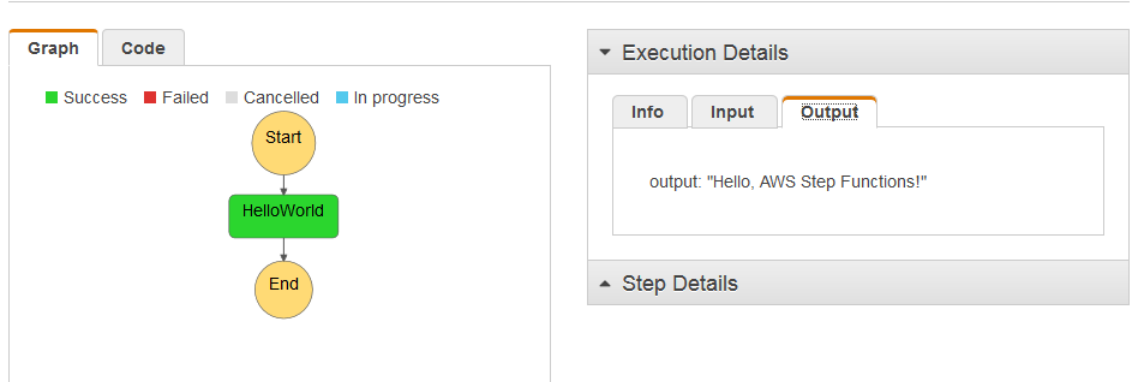
A new execution of your state machine starts, and a new page showing your running execution is displayed.

6. In the **Execution Details** section, choose the **Info** tab to view the **Execution Status** and the **Started** and **Closed** timestamps.
7. To view the results of your execution, choose the **Output** tab.

[Dashboard](#) > [LambdaStateMachine](#) > b4781ddd-7e3e-5fba-b7d2-2c42130aacf6

Execution Arn: arn:aws:states:us-east-1:██████████:execution:LambdaStateMachine:b4781ddd-7e3e-5fba-b7d2-2c42130aacf6

b4781ddd-7e3e-5fba-b7d2-2c42130aacf6 



The screenshot shows the AWS Step Functions console interface. On the left, the 'Graph' tab is active, displaying a state machine flow diagram. The flow starts at a yellow circle labeled 'Start', moves to a green rectangle labeled 'HelloWorld', and ends at a yellow circle labeled 'End'. A legend above the graph indicates that green represents 'Success', red represents 'Failed', grey represents 'Cancelled', and blue represents 'In progress'. On the right, the 'Execution Details' section is expanded, showing three tabs: 'Info', 'Input', and 'Output'. The 'Output' tab is selected, displaying the text 'output: "Hello, AWS Step Functions!"'. Below the output, there is a section for 'Step Details' which is currently collapsed.

ID	Type	Timestamp
▶ 1	ExecutionStarted	Nov 17, 2016 4:58:11 PM
▶ 2	TaskStateEntered	Nov 17, 2016 4:58:11 PM
▶ 3	LambdaFunctionScheduled	Nov 17, 2016 4:58:11 PM
▶ 4	LambdaFunctionStarted	Nov 17, 2016 4:58:11 PM
▶ 5	LambdaFunctionSucceeded	Nov 17, 2016 4:58:14 PM
▶ 6	TaskStateExited	Nov 17, 2016 4:58:14 PM
▶ 7	ExecutionSucceeded	Nov 17, 2016 4:58:14 PM

Creating a Lambda State Machine Using AWS CloudFormation

This tutorial shows you how to create a basic AWS Lambda function and start a state machine execution automatically. You will use the AWS CloudFormation console and a YAML *template* to create the *stack* (IAM roles, the Lambda function, and the state machine). You will then 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: Setting Up Your AWS CloudFormation Template \(p. 15\)](#)
- [Step 2: Using the AWS CloudFormation Template to Create a Lambda State Machine \(p. 17\)](#)
- [Step 3: Starting a State Machine Execution \(p. 19\)](#)

Step 1: Setting Up Your AWS CloudFormation Template

Before you use the [example YAML template \(p. 17\)](#), you should understand its separate parts.

To create an IAM role for Lambda

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

```
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 which prints the message `Hello World`.

```
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: "nodejs4.3"
    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.

```
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: "*"

```

To create a Lambda state machine

Define the Lambda state machine.

```
MyStateMachine:
  Type: "AWS::StepFunctions::StateMachine"
  Properties:
    DefinitionString:
      !Sub
      - |-
        {
          "Comment": "A Hello World AWL example using an AWS Lambda Function",
          "StartAt": "HelloWorld",
          "States": {
            "HelloWorld": {
              "Type": "Task",
              "Resource": "${lambdaArn}",
              "End": true
            }
          }
        }

```

```
    }  
  }  
}  
- {lambdaArn: !GetAtt [ MyLambdaFunction, Arn ]}  
RoleArn: !GetAtt [ StatesExecutionRole, Arn ]
```

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

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

To create the Lambda state machine

1. Copy the following example YAML data to a file named `MyStateMachine.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: "nodejs4.3"  
      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
```

AWS Step Functions Developer Guide
Step 2: Using the AWS CloudFormation
Template to Create a Lambda State Machine

```
        Action:
          - "lambda:InvokeFunction"
        Resource: "*"

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

2. Log in to the [AWS CloudFormation console](#) and choose **Create Stack**.
3. On the **Select Template** page, select **Upload a template to Amazon S3**. Choose your `MyStateMachine.yaml` file, and then choose **Next**.

Select Template

Select the template that describes the stack that you want to create. A stack is a group of related resources that you manage as a single unit.

Design a template Use AWS CloudFormation Designer to create or modify an existing template. [Learn more.](#)

Design template

Choose a template A template is a JSON/YAML-formatted text file that describes your stack's resources and their properties. [Learn more.](#)

Select a sample template

Upload a template to Amazon S3

Browse... MyStateMachine.yaml

Specify an Amazon S3 template URL

Cancel

Next

4. On the **Specify Details** page, for **Stack name**, type `MyStateMachine`, and then choose **Next**.

Specify Details

Specify a stack name and parameter values. You can use or change the default parameter values, which are defined in the AWS CloudFormation template. [Learn more.](#)

Stack name

Cancel Previous **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**.

i The following resource(s) require capabilities: [AWS::IAM::Role]
This template contains Identity and Access Management (IAM) resources that might provide entities access to make changes to your AWS account. Check that you want to create each of these resources and that they have the minimum required permissions. [Learn more.](#)

I acknowledge that AWS CloudFormation might create IAM resources.

Cancel Previous **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.

Filter: Active ▾ By Stack Name Showing 1 stack

	Stack Name	Created Time	Status	Description
<input checked="" type="checkbox"/>	MyStateMachine	2017-02-06 16:32:20 UTC-0800	CREATE_COMPLETE	A sample template for statemachine v

7. (Optional) To display the resources in your stack, select the stack and choose the **Resources** tab.

Overview	Outputs	Resources	Events	Template	Parameters	Tags	Stack Policy	Change Sets
Logical ID	Physical ID	Type	Status					
LambdaExecutionRole	HelloWorld-LambdaExecutionRole-	AWS::IAM::Role	CREATE_COMPLETE					
MyLambdaFunction	HelloWorld-MyLambdaFunction-	AWS::Lambda::Function	CREATE_COMPLETE					
MyStateMachine	arn:aws:states:us-west-2-:stateMachine:MyStateMachine-	AWS::StepFunctions::StateMachine	CREATE_COMPLETE					
StatesExecutionRole	HelloWorld-StatesExecutionRole-	AWS::IAM::Role	CREATE_COMPLETE					

Step 3: Starting a State Machine Execution

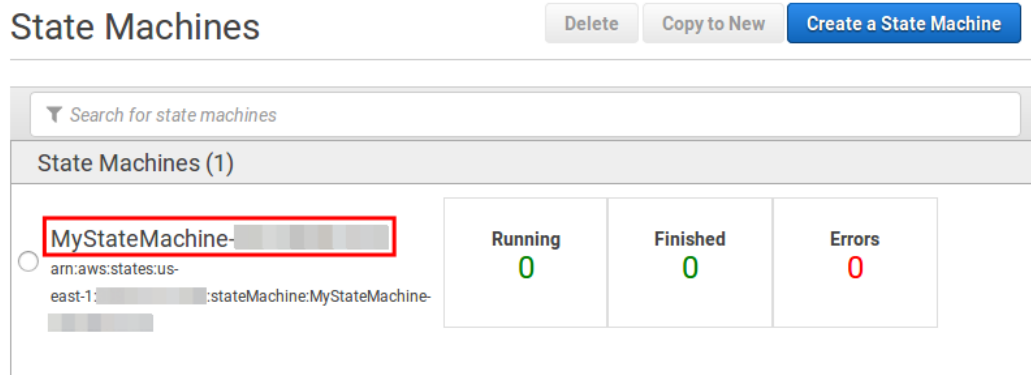
After you create your Lambda state machine, you can start an execution.

To start the state machine execution

1. Log in to the [Step Functions console](#).

On the **Dashboard** page, the `MyStateMachine` state machine and its ARN are displayed.

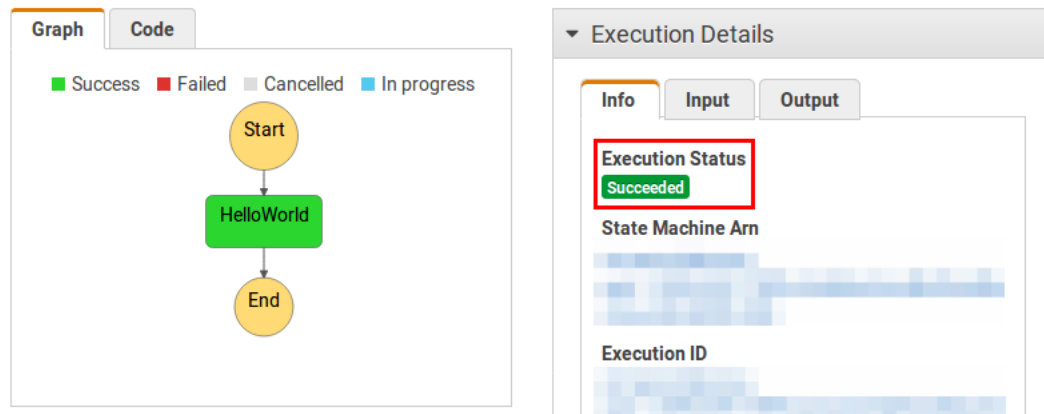
2. Choose the name of the state machine that you have created using AWS CloudFormation.



3. On the **MyStateMachine-ABCDEFGHIJK** page, choose **New execution**.

4. At the bottom of the **New execution** page, choose **Start Execution**.

The state machine is executed and Step Functions displays the **Succeeded** status.



Creating an Activity State Machine

You can run task code on a state machine. This tutorial introduces you to creating an activity-based state machine using Java and AWS Step Functions.

To complete this tutorial you'll need the following:

- The [AWS 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: Creating a New Activity](#) (p. 21)
- [Step 2: Creating a State Machine](#) (p. 21)
- [Step 3: Implementing a Worker](#) (p. 22)
- [Step 4: Starting an Execution](#) (p. 24)
- [Step 5: Running and Stopping the Worker](#) (p. 24)

Step 1: Creating 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.

To create the new activity task

1. Log in to the [Step Functions console](#) and choose **Tasks**.
2. Choose **Create new activity**.
3. On the **Tasks** page, type an **Activity Name**, for example `get-greeting`, and choose **Create Activity**.
4. When your activity task is created, note its Amazon Resource Name (ARN), displayed on the right-hand side of the page, for example:

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

Step 2: Creating a State Machine

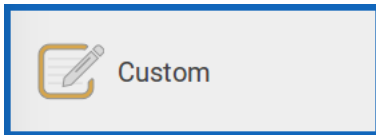
Create a state machine that will determine when your activity is invoked and when your worker should perform its primary work, collect its results, and return them.

To create the state machine

1. Log in to the [Step Functions console](#) and choose **Get Started**.

The **Create a state machine** page is displayed.

2. Choose the **Custom** blueprint.



3. In the box below **Name your state machine** type a name, for example `ActivityStateMachine`.

Note

State machine names must be 1-80 characters in length, must be unique for your account and region, and must not contain any of the following:

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

4. In the **Code** pane, add the following JSON text, using a `Task` state and the ARN of the activity task that you have created earlier 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 \(p. 54\)](#).

5. Choose **Create State Machine**.

The **IAM role for your state machine executions** dialog box is displayed. Step Functions creates and selects an IAM role automatically.

IAM role for your state machine executions ✕

Select an IAM role for your tasks (See IAM documentation [here](#))

StatesExecutionRole-us-east-1

If you don't have an IAM role, you can [create one manually here](#)

Cancel **OK**

Note

If you delete the IAM role that Step Functions creates, Step Functions can't re-create it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later. For more information about creating an IAM role manually, see [Creating IAM Roles for Use with AWS Step Functions \(p. 87\)](#).

6. Choose **OK**.

The state machine is created and an acknowledgement page is displayed.

Step 3: Implementing a Worker

Create a *worker*, a program which is responsible for the following:

- 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 code below).
- Returning the results using the `SendTask*` API actions.

To implement the worker

1. Create a new 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*. To give Step Functions sufficient time to process the request, `setSocketTimeout` is set 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 have created earlier.

Step 4: Starting 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.

To start the execution

1. In the [Step Functions console](#), choose **Dashboard** and then choose the name of the state machine that you have created earlier.

Dashboard

State Machines

Delete

Copy to New

Create a State Machine

Search for state machines			
State Machines (3)			
<input type="radio"/> HelloStateMachine arn:aws:states::: :stateMachine:HelloStateMachine	Running 0	Finished 1	Errors 0
<input type="radio"/> LambdaStateMachine arn:aws:states::: :stateMachine:LambdaStateMachine	Running 0	Finished 1	Errors 0
<input type="radio"/> RetryStateMachine arn:aws:states::: :stateMachine:RetryStateMachine	Running 0	Finished 0	Errors 0

2. On the state machine's detail page, choose **New execution**.

The **New execution** page is displayed.

3. (Optional) To help identify your execution, you can enter an ID for it. To specify the ID, use the **Enter your execution id here** text box. If you don't enter an ID, Step Functions generates a unique ID automatically.
4. In the execution input area, replace the example data with the following:

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

5. Choose **Start Execution**.

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

Step 5: Running and Stopping the Worker

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

After the execution completes, you should stop your worker. If you don't stop the worker, it will continue to run and poll for activities. When the execution is stopped, your worker has no source of tasks and generates a `SocketTimeoutException` during each poll.

To run and stop 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.

Starting 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 how to set a state machine as a target for a CloudWatch Events rule that starts the execution of a state machine every 5 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: Creating a State Machine \(p. 25\)](#)
- [Step 2: Creating a CloudWatch Events Rule \(p. 25\)](#)

Step 1: Creating 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 \(p. 3\)](#) tutorial.
- If you already have a state machine, proceed to the next step.

Step 2: Creating a CloudWatch Events Rule

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

To create the rule

1. Navigate to the [CloudWatch Events console](#), choose **Events**, and then choose **Create Rule**.

The **Step 1: Create rule** page is displayed.

- In the **Event source** section, select **Schedule** and type 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 ▼

Cron expression

[Learn more about CloudWatch Events schedules.](#)

▶ Show sample event(s)

- In the **Targets** section, choose **Add target** and 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* ▼

▶ 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

Use existing role

- 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 before, choose **Use existing role**.
- Choose **Configure details**.

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

- Type a **Name** for your rule, choose **Enabled** for **State**, and then choose **Create rule**.

Rule definition

Name*

Description

State Enabled

* Required Cancel Back Create rule

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.

Create rule Actions ▾

Status **All** Name

	Status	Name	Description
<input type="radio"/>	<input checked="" type="checkbox"/>	statemachine-event	CloudWatch Events

A new execution of your state machine starts every 5 minutes.

Handling Error Conditions Using a State Machine

This tutorial introduces you to handling error conditions using an AWS Step Functions state machine with a `catch` field. In this tutorial, you'll use an AWS Lambda function to respond with conditional logic based on error message type, a method 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. 49).

Topics

- [Step 1: Creating an IAM Role for Lambda](#) (p. 28)
- [Step 2: Creating a Lambda Function That Fails](#) (p. 28)
- [Step 3: Testing the Lambda Function](#) (p. 29)
- [Step 4: Creating a State Machine with a Catch Field](#) (p. 30)
- [Step 5: Starting a New Execution](#) (p. 32)

Step 1: Creating an IAM Role for Lambda

Both Lambda and 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.

In the [Getting Started \(p. 3\)](#) tutorial, this is done automatically for Step Functions: an IAM role is created when you create the state machine. However, Lambda requires you to assign an 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.

To create a role for use with Lambda

1. Log in to the [IAM console](#) and choose **Roles, Create New Role**.
2. On the **Set Role Name** page, type the **Role Name** and then choose **Next Step**.
3. On the **Select Role Type** page, select **AWS Lambda** from the list.

Note

The role is automatically provided with a trust relationship that allows Lambda to use the role.

4. On the **Attach Policy** page, choose **Next Step**.
5. On the **Review** page, choose **Create Role**.

The role appears in the list of roles in the IAM console.

Step 2: Creating a Lambda Function That Fails

Use a Lambda function to simulate an error condition.

To simulate a failing Lambda function

1. Log in to the [Lambda console](#).
2. If this is your first Lambda function, choose **Get Started Now**. Otherwise, choose **Create a Lambda function**.
3. On the **Select blueprint** page, type `step functions` into the **Filter**, and then choose the **step-functions-error** blueprint.

Select blueprint ?

Blueprints are sample configurations of event sources and Lambda functions. Choose a blueprint that best aligns with your desired scenario and customize as needed, or skip this step if you want to author a Lambda function and configure an event source separately. Except where otherwise noted, blueprints are licensed under [CC0](#).

Welcome to AWS Lambda! You can get started on creating your first Lambda function by choosing one of the blueprints below. ✕

Select runtime ⌵

step functions ⏪ < Viewing 1-3 of 3 > ⏩

<p>Blank Function</p> <p>Configure your function from scratch. Define the trigger and deploy your code by stepping through our wizard.</p> <hr/> <p>custom</p>	<p>step-functions-error-python</p> <p>An AWS Lambda function that throws an error. AWS Step Functions state machines can be configured to handle</p> <hr/> <p>python2.7 ⏴</p>	<p>step-functions-error</p> <p>An AWS Lambda function that throws an error. AWS Step Functions state machines can be configured to handle</p> <hr/> <p>nodejs6.10 ⏴</p>
---	---	---

4. On the **Configure triggers** page, choose **Next**.
5. On the **Configure function** page, type `FailFunction` for the **Name**.

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!`.

6. In the **Lambda function handler and role** section, select **Choose an existing role** from the **Role** list. Then select the Lambda role that you have created earlier from the **Existing role** list.

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. In the meanwhile, verify that `lambda.amazonaws.com` has access to the role. To verify or edit the trust relationship for your role, revisit [Step 1: Creating an IAM Role for Lambda \(p. 28\)](#).

You can use the **Timeout** value in the **Advanced settings** to specify how long your function can take to execute before failing.

7. Choose **Next**, review your function, and then choose **Create function**.
8. When your Lambda function is created, note its Amazon Resource Name (ARN), displayed in the upper-right corner of the page. For example:

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

Step 3: Testing the Lambda Function

Test your Lambda function to see it in operation.

To test your Lambda function

1. On your Lambda function page, choose **Test**.

The **Input test event** dialog box is displayed.

2. Choose **Save and test**.

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

Execution result: failed (logs)

The area below shows the result returned by your function execution. [Learn more](#) about returning results from your function.

```
{
  "errorMessage": "This is a custom error!",
  "errorType": "CustomError",
  "stackTrace": [
    "exports.handler (/var/task/index.js:9:34)"
  ]
}
```

Summary

Code SHA-256	
Request ID	82b8bd9f-1b23-11e7-8f94-2f30e0a3d77f
Duration	0.70 ms
Billed duration	100 ms
Resources configured	128 MB
Max memory used	15 MB

Log output

The area below shows the logging calls in your code. CloudWatch log group.

```
START RequestId: 82b8bd9f-1b23-11e7-8f94-2f30e0a3d77f
2017-04-06T23:48:18.921Z      82b8bd9f-1b23-11e7-8f94-2f30e0a3d77f
END RequestId: 82b8bd9f-1b23-11e7-8f94-2f30e0a3d77f
REPORT RequestId: 82b8bd9f-1b23-11e7-8f94-2f30e0a3d77f
```

Step 4: Creating a State Machine with a `Catch` Field

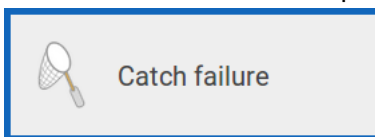
Use the [Step Functions console](#) to create a state machine that uses a `Task` with a `Catch` field. Add a reference to your Lambda function in the `Task` state. The Lambda function is invoked and fails during execution. Step Functions retries the function twice using exponential backoff between retries.

To create the state machine

1. Log in to the [Step Functions console](#) and choose **Get Started**.

The **Create a state machine** page is displayed.

2. Choose the **Catch Failure** blueprint.



3. In the box under **Name your State Machine**, type a name, for example `CatchStateMachine`.

Note

State machine names must be 1-80 characters in length, must be unique for your account and region, and must not contain any of the following:

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

4. In the **Code** pane, replace the example JSON text with the following, using a `Task` state and the ARN of the Lambda function that you have created earlier in 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
  }
}
```

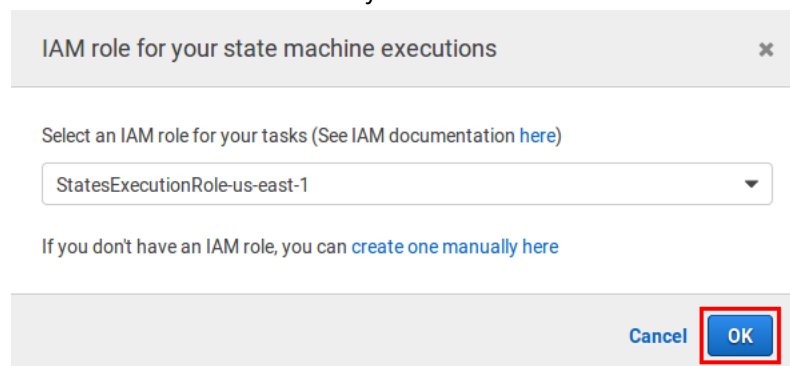
This is a description of your state machine using the Amazon States Language. It defines a single `Task` state named `CreateAccount`. For more information, see [State Machine Structure \(p. 54\)](#).

Note

For more information about the syntax of the `Retry` field, see [Retrying After an Error \(p. 69\)](#).

5. Choose **Create State Machine**.

The **IAM role for your state machine executions** dialog box is displayed. Step Functions creates and selects an IAM role automatically.



Note

If you delete the IAM role that Step Functions creates, Step Functions can't re-create it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later. For more information about creating an IAM role manually, see [Creating IAM Roles for Use with AWS Step Functions \(p. 87\)](#).

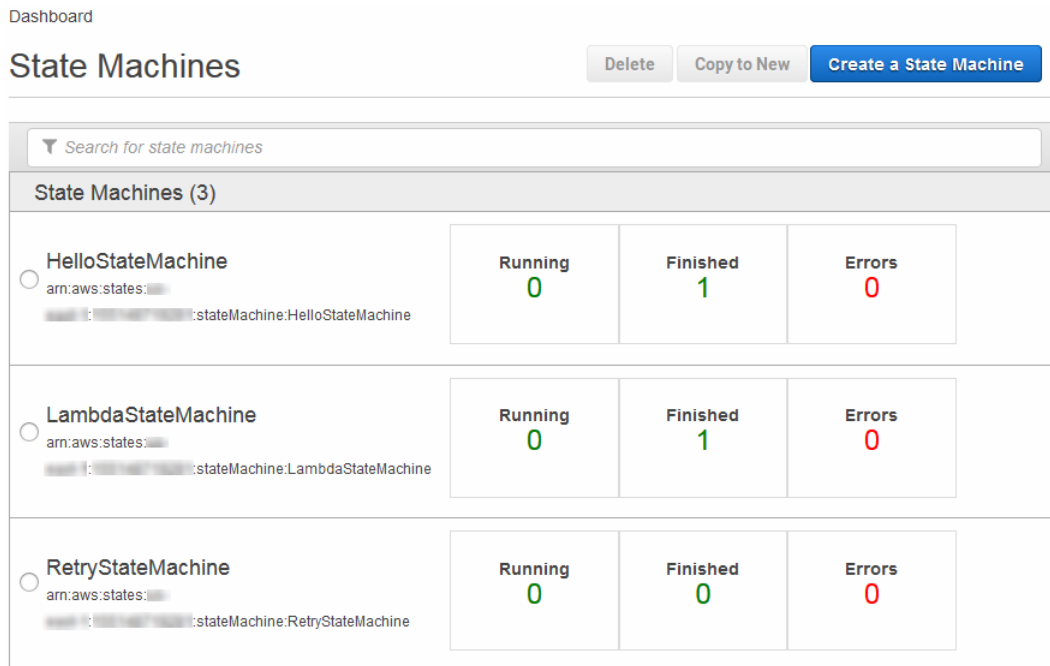
6. Choose **OK**.

Step 5: Starting a New Execution

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

To start a new execution

1. In the [Step Functions console](#), choose **Dashboard**, and then choose the name of the state machine that you have created earlier.



2. On the state machine's detail page, choose **New execution**.

The **New execution** page is displayed.



3. (Optional) To help identify your execution, you can enter an ID for it. To specify the ID, use the **Enter your execution id here** text box. If you don't enter an ID, Step Functions generates a unique ID automatically.
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 the **Info** tab to view the **Execution Status** and the **Started** and **Closed** timestamps.
6. To view the results of your execution, choose the **Output** tab.

Execution Arn: arn:aws:states:ap-northeast-1: :execution:CatchFailure:3b095a9c-0dff-f9a7-a420-e32235d9cb72

3b095a9c-0dff-f9a7-a420-e32235d9cb72 ✓

The screenshot displays the AWS Step Functions console interface. On the left, the 'Graph' tab shows a state machine flow starting with a 'Start' node, followed by a 'HelloWorld' task (green), which branches into three paths: 'CustomErrorFallback' (green), 'ReservedTypeFallback' (dashed), and 'CatchAllFallback' (dashed). All paths converge at an 'End' node. A legend indicates: Success (green), Failed (red), Cancelled (grey), and In progress (blue). On the right, the 'Execution Details' section is open, with the 'Output' tab selected, showing the text: 'output: "This is a fallback from a custom lambda function exception"'. Below this, the 'Step Details' section is partially visible.

ID	Type	Timestamp
▶ 1	ExecutionStarted	Mar 14, 2017 3:33:32 PM
▶ 2	TaskStateEntered	Mar 14, 2017 3:33:32 PM
▶ 3	LambdaFunctionScheduled	Mar 14, 2017 3:33:32 PM
▶ 4	LambdaFunctionStarted	Mar 14, 2017 3:33:33 PM
▶ 5	LambdaFunctionFailed	Mar 14, 2017 3:33:33 PM
▶ 6	TaskStateExited	Mar 14, 2017 3:33:33 PM
▶ 7	PassStateEntered	Mar 14, 2017 3:33:33 PM
▶ 8	PassStateExited	Mar 14, 2017 3:33:33 PM
▶ 9	ExecutionSucceeded	Mar 14, 2017 3:33:33 PM

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, so that, 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*.

Topics

- [Step 1: Creating an IAM Role for API Gateway \(p. 34\)](#)
- [Step 2: Creating your API Gateway API \(p. 34\)](#)
- [Step 3: Testing and Deploying the API Gateway API \(p. 37\)](#)

Step 1: Creating an IAM Role for API Gateway

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

To create the IAM role

1. Log in to the [AWS Identity and Access Management console](#).
2. On the **Roles** page, choose **Create New Role**.
3. On the **Set Role Name** page, type `APIGatewayToStepFunctions` for **Role Name**, and then choose **Next Step**.
4. On the **Select Role Type** page, under **Select Role Type**, select **Amazon API Gateway**.
5. On the **Attach Policy** page, choose **Next Step**.
6. On the **Review** page, note the **Role ARN**, for example:

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

7. Choose **Create Role**.

To attach a policy to the IAM role

1. On the **Roles** page, search for your role by name (`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: Creating your API Gateway API

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

To create the API

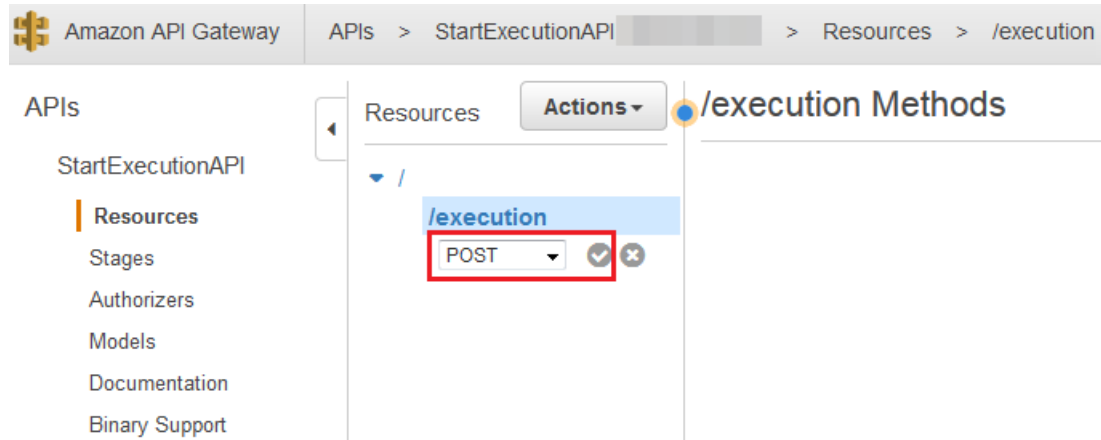
1. Navigate to the [Amazon API Gateway console](#).
2. On the **APIs** page, choose **Create API**.
3. On the **Create new API** page, type `StartExecutionAPI` for the **API name**, and then choose **Create API**.

To create a resource

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

To create a POST Method

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



To 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 the [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**, type `startExecution`.
7. For **Execution Role**, type [the role ARN of the IAM role \(p. 34\)](#) that you have created earlier, 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

AWS Service

AWS Subdomain

HTTP method

Action Type Use action name
 Use path override

Action

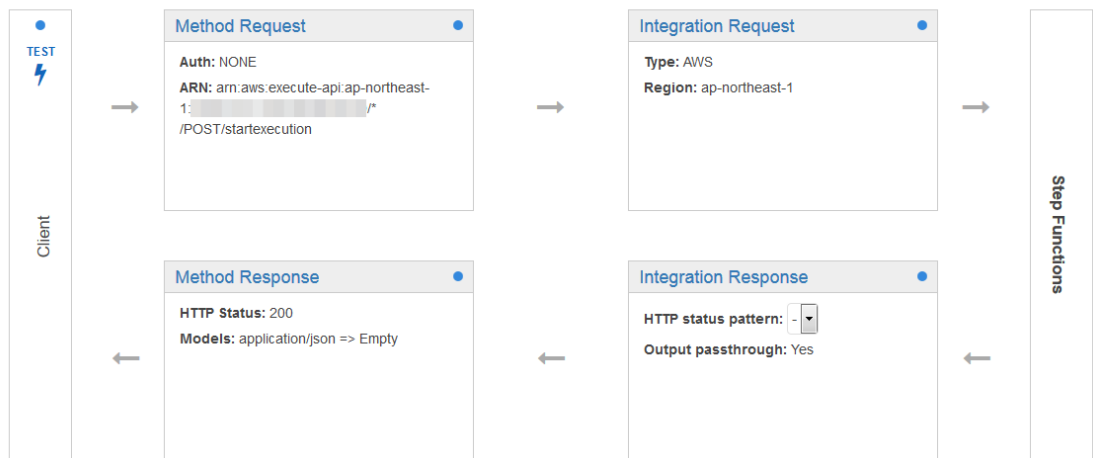
Execution role ⓘ

Content Handling ⓘ

8. Choose **Save**.

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

/startexecution - POST - Method Execution



Step 3: Testing and Deploying the API Gateway API

To 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 \(p. 3\)](#)), and then choose **Test**.

```
{
  "input": "{}",
  "name": "MyExecution",
  "stateMachineArn": "arn:aws:states:ap-
northeast-1:123456789012:stateMachine>HelloWorld"
}
```

Note

For more information, see the `StartExecution` [Request Syntax](#) in the *AWS Step Functions API Reference*.

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

```
{
  "executionArn": "arn:aws:states:ap-
northeast-1:123456789012:execution>HelloWorld:MyExecution",
  "startDate": 1486768956.878
}
```

Note

You can view the execution by choosing your state machine on the [AWS Step Functions console](#).

To deploy your API

1. On the **Resources** page of `StartExecutionAPI`, choose **Actions, Deploy API**.
2. In the **Deploy API** dialog box, select **[New Stage]** from the **Deployment stage** list, type `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.ap-northeast-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:ap-northeast-1:123456789012:stateMachine>HelloWorld"}' https://
a1b2c3d4e5.execute-api.ap-northeast-1.amazonaws.com/alpha/execution
```

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

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

How Step Functions Works

To understand AWS Step Functions, you will need to be familiar with a number of important concepts. This section describes how Step Functions works.

Topics

- [Blueprints \(p. 39\)](#)
- [States \(p. 40\)](#)
- [Tasks \(p. 40\)](#)
- [Activities \(p. 41\)](#)
- [Transitions \(p. 42\)](#)
- [State Machine Data \(p. 42\)](#)
- [Executions \(p. 45\)](#)
- [Error Handling \(p. 45\)](#)

Blueprints

In the [Step Functions console](#), you can choose one of the following state machine blueprints to automatically fill the **Code** pane. Each of the blueprints is fully functional and you can use any blueprint as the template for your own state machine.

Important

Choosing any of the blueprints overwrites the contents of the **Code** pane.

- **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.
- **Hello World** – A state machine with a single task.
- **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, running a `Task` state from a set of `Task` states or running a `Fail` state after the initial state is complete.

States

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.

Note

An instance of a state exists until the end of its execution.

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

- Do some work in your state machine (a [Task \(p. 40\)](#) state).
- Make a choice between branches of execution (a [Choice \(p. 60\)](#) state)
- Stop an execution with a failure or success (a [Fail \(p. 63\)](#) or [Succeed \(p. 63\)](#) state)
- Simply pass its input to its output or inject some fixed data (a [Pass \(p. 56\)](#) state)
- Provide a delay for a certain amount of time or until a specified time/date (a [Wait \(p. 62\)](#) state)
- Begin parallel branches of execution (a [Parallel \(p. 64\)](#) state)

For example, here is an example state named `HelloWorld` which performs a 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 a number of 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.

For more information regarding the various states that you can define using Amazon States Language, see [States \(p. 55\)](#).

Once you have created a state machine and 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 in the [Step Functions console](#).

Tasks

All work in your state machine is done by *tasks*. A task can be:

- An Activity, which can consist of any code in any language. Activities can be hosted on EC2, ECS, mobile devices—basically anywhere. Activities must poll AWS Step Functions using the `GetActivityTask` and `SendTask*` API calls. Ultimately, an activity can even be a human task—a task that waits for a human to perform some action and then continues.
- A [Lambda function \(p. 9\)](#), which is a completely cloud-based task that runs on Λ . You can write Lambda functions in JavaScript (using the AWS Management Console or by uploading code to Lambda), in Java, or in Python (by uploading code to Lambda).

Tasks are represented in Amazon States Language by setting a state's type to `TASK` and providing it with the ARN of the created activity or Lambda function. For details about how to specify different task types, see [Task \(p. 57\)](#) in the [Amazon States Language \(p. 53\)](#).

To see a list of your tasks, you can access the **Tasks** page in the [Step Functions console](#).

Activities

Activities are an AWS Step Functions concept that refers to a task to be performed by a *worker* that can be hosted on EC2, ECS, mobile devices—basically anywhere.

Topics

- [Creating an Activity \(p. 41\)](#)
- [Writing a Worker \(p. 41\)](#)

Creating an Activity

Activities are referred to by name. An activity's name can be any string that adheres to the following rules:

- It must be between 0 – 80 characters in length.
- It must be unique within your AWS account and region.

Activities can be created with Step Functions in any of the following ways:

- Call [CreateActivity](#) with the activity name.
- Using the [Step Functions console](#).

Note

Activities are not versioned and are expected to always be backwards compatible. If you must make a backwards-incompatible change to an activity definition, then a *new* activity should be created with Step Functions using a unique name.

Writing a Worker

Workers can be implemented in any language that can make AWS Step Functions API calls. Workers should repeatedly poll for work by implementing the following pseudo-code algorithm:

```
[taskToken, jsonInput] = GetActivityTask();
try {
    // Do some work...
    SendTaskSuccess(taskToken, jsonOutput);
} catch (Exception e) {
```

```
    SendTaskFailure(taskToken, reason, errorCode);  
}
```

Sending Heartbeat Notifications

States that have long-running activities should provide a heartbeat timeout value to verify that the activity is still running successfully.

If your activity has a heartbeat timeout value, the worker which implements it must send heartbeat updates to Step Functions. To send a heartbeat notification from a worker, use the [SendTaskHeartbeat](#) action.

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 \(p. 53\)](#) 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, which is determined by the contents of the states themselves.
- 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 to a state machine
- Data passed between states
- The output from a state machine.

This topic describes how state machine data is formatted and used in AWS Step Functions.

Topics

- [Data Format \(p. 43\)](#)
- [State Machine Input/Output \(p. 43\)](#)
- [State Input/Output \(p. 43\)](#)

Data Format

State machine data is represented by JSON text, so values can be provided using any data type supported by JSON: objects, arrays, numbers, strings, boolean values, and `null`.

Note that:

- Numbers in JSON text format conform to JavaScript semantics, typically corresponding to double-precision [IEEE-854](#) values.
- Stand-alone quote-delimited strings, booleans, and numbers are valid JSON text.
- The output of a state becomes the input to the next state. However, states can be restricted to working on a subset of the input data by using [Input and Output Processing \(p. 66\)](#).

State Machine Input/Output

AWS Step Functions can be given initial input data when you start an execution, by passing it to [StartExecution](#) 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 `terminal` (last) state that is reached, and is provided as JSON text in the execution's result. Execution results can be retrieved from the execution history by external callers (for example, in [DescribeExecution](#)) and can be viewed in the [Step Functions console](#).

State Input/Output

Each state's input consists of JSON text received from the preceding state or, in the case of the `startAt` state, the input to the execution.

A state's output must be given as JSON text. Certain flow-control states simply echo their input to their output.

For example, here is a state machine that adds two numbers together:

```
{
  "Comment": "An example that adds two numbers.",
  "StartAt": "Add",
  "Version": "1.0",
  "TimeoutSeconds": 10,
  "States": {
    "Add": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Add",
    }
  }
}
```

```
        "End": true
      }
    }
  }
```

which uses this 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 });
}
```

If an execution is started with the JSON text:

```
{ "numbers": [3, 4] }
```

The `Add` state receives the JSON text and passes it to the lambda function, which returns the result of the calculation to the state. The state then returns this value in its output:

```
{ "result": 7 }
```

Since `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 (`{}`).

Filters

Some states, such as [Task \(p. 57\)](#), have `InputPath`, `ResultPath`, and `OutputPath` fields. The values of these fields are [path \(p. 66\)](#).

The `InputPath` field selects a portion of the state's input to be passed to the state's processing logic (an Activity, Lambda function, or so on). If the `InputPath` field is omitted, the entire state input is selected by default (`$`). If it is `null`, an empty object `{}` is passed.

The `ResultPath` field selects a portion of the state's input to be overwritten by, or added to, with result data from the state's processing logic. The `ResultPath` field is optional and, if omitted, defaults to `$`, which overwrites the entire input. However, before the input is sent as the state's output, a portion can be selected with the `OutputPath` field.

The `OutputPath` field is also optional and, if omitted, defaults to `$`, which selects the entire input (as modified by the `ResultPath`), sending it as the state's output.

The `ResultPath` field's value can be `null`, which causes any output from your state's processing logic to be discarded instead of being added to the state's input. In this scenario, the state's output is identical to the state's input, given the default value for the `OutputPath` field.

If the `OutputPath` field's value is `null`, and empty object `{}` is sent as the state's output.

Here is an example. Given the following `ResultPath` field in a state that outputs the sum of its input values:

```
"InputPath": "$.numbers",
"ResultPath": "$.sum"
"OutputPath": "$"
```

With the following state input data:


```
{  
  "numbers": [3, 4]  
}
```

The state output data will have the following structure and values:

```
{  
  "numbers": [3, 4],  
  "sum": 7  
}
```

Let's change the `OutputPath` in our example slightly...

```
"InputPath": "$.numbers",  
"ResultPath": "$.sum"  
"OutputPath": "$.sum"
```

As before, with the following state input data:

```
{  
  "numbers": [3, 4]  
}
```

However, now the state output data is 7:

```
{  
  7  
}
```

By using the `InputPath` and `ResultPath` fields in this way, you can design separation between the names of data members in your state machine data, and the functions that process it.

Executions

A state machine *execution* occurs when a 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 using the AWS SDKs, the Step Functions API actions, or the AWS CLI. An execution receives JSON input and produces JSON output.

For more information about the different ways of working with Step Functions, see [Development Options \(p. 7\)](#). For more information about initiating an execution from the Step Functions console, see [To start a new execution \(p. 5\)](#).

Note

The Step Functions console displays a maximum of 1,000 executions per state machine. If you have more than 1,000 executions, use the Step Functions API actions, or the AWS CLI to display all of your executions.

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, Step Functions causes the execution to fail entirely.

Error Names

Step Functions identifies errors in Amazon States Language using case-sensitive strings, known as *error names*. 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 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.

Note

States can report errors with other names. However, these must not begin with the `states.` prefix.

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.

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-states-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`, `ErrorA`, 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.

Note

When the system transitions to another state, all retrier parameters are reset, regardless of how the transition happens.

You can generate custom error names such as (`ErrorA`, `ErrorB`) using an [Activity \(p. 59\)](#), but not using [Lambda Functions \(p. 59\)](#). For more information, see the response syntax for the [Invoke API action](#) in the *AWS Lambda Developer Guide*.

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 retriever 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 \(p. 66\)](#) 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  
    }  
  }  
}
```

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  
  }  
}  
}
```


Amazon States Language

Amazon States Language is a JSON-based, structured language used to define your state machine, a collection of [states](#) (p. 40), 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. 3).

Topics

- [Example Amazon States Language Specification](#) (p. 53)
- [State Machine Structure](#) (p. 54)
- [States](#) (p. 55)
- [Input and Output Processing](#) (p. 66)
- [Errors](#) (p. 68)

Example Amazon States Language Specification

```
{
  "Comment": "An Amazon States Language state machine example 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",
    "Cause": "No Matches!"
  },
  "NextState": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FUNCTION_NAME",
    "End": true
  }
}
}
```

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 (case-sensitive) the name of one of the state objects.

timeoutSeconds (Optional)

The maximum number of seconds an execution of the state machine may run; if it runs longer than the specified time, then the execution fails with an `States.Timeout` [Error name \(p. 69\)](#).

version (Optional)

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

states (Required)

This field's value is an object containing a comma-delimited set of states.

The `states` field contains a number of [States \(p. 55\)](#):

```
{
  "State1" : {
  },
}
```

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

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

Here's 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, which is determined by the contents of the states themselves.
- 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 to be an `end` (or `terminal`) state. 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.

States

States are top-level elements within a state machine's `states` field, and can take a number of different roles in your state machine depending on their type.

```
"FirstState" : {  
  "Type" : "Task",  
  ...  
}
```

States are identified by their name, which must be unique within the state machine specification, but otherwise can be any valid string in JSON text format. Each state also contains a number of fields with options that vary according to the contents of the state's required `Type` field.

Note

State machine names must be 1-80 characters in length, must be unique for your account and region, and must not contain any of the following:

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

Topics

- [Common State Fields \(p. 56\)](#)
- [Pass \(p. 56\)](#)
- [Task \(p. 57\)](#)
- [Choice \(p. 60\)](#)
- [Wait \(p. 62\)](#)
- [Succeed \(p. 63\)](#)
- [Fail \(p. 63\)](#)
- [Parallel \(p. 64\)](#)

Common State Fields

Type (Required)

The state's type.

Next

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

End

Designates this state as a terminal state (it 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`, do not support or use the `End` field.

Comment (Optional)

Holds a human-readable description of the state.

InputPath (Optional)

A [path \(p. 66\)](#) 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 \(p. 66\)](#).

OutputPath (Optional)

A [path \(p. 66\)](#) 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 \(p. 66\)](#).

Pass

A `Pass` state (`"Type": "Pass"`) simply passes its input to its output, performing no work. `Pass` states are useful when constructing and debugging state machines.

In addition to the [common state fields \(p. 56\)](#), `Pass` states allow the following fields:

Result (Optional)

Treated as the output of a virtual task to be passed on to the next state, and filtered as prescribed 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 prescribed by the `OutputPath` field (if present) before being used as the state's output. For more information, see [Input and Output Processing \(p. 66\)](#).

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:

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

Then the output would be:

```
{
  "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.

In addition to the [common state fields \(p. 56\)](#), `Task` states have the following fields:

Resource (Required)

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

ResultPath (Optional)

Specifies where (in the input) to place the results of executing the task specified in `Resource`. The input is then filtered as prescribed by the `OutputPath` field (if present) before being used as the state's output. For more information, see [path \(p. 66\)](#).

Retry (Optional)

An array of objects, called Retriers, that define a retry policy in case the state encounters runtime errors. For more information, see [Retrying After an Error \(p. 69\)](#).

Catch (Optional)

An array of objects, called Catchers, that define a fallback state which is executed in case the state encounters runtime errors and its retry policy has been exhausted or is not defined. For more information, see [Fallback States \(p. 71\)](#).

TimeoutSeconds (Optional)

If the task runs longer than the specified seconds, then this state fails with a `States.Timeout` Error Name. Must be a positive, non-zero integer. If not provided, the default value is 99999999.

HeartbeatSeconds (Optional)

If more time than the specified seconds elapses between heartbeats from the task, then this state fails with an `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.

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 will be run upon completion of the `Task` state.

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` 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 does not send heartbeat notifications in intervals of 60 seconds, then 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>
```

Where:

- `partition` is the AWS Step Functions partition to use, most commonly `aws`.
- `service` indicates the AWS service used to execute the task, and is either:
 - `states` for an [Activity \(p. 59\)](#).
 - `lambda` for a [Lambda function \(p. 59\)](#).
- `region` is the [AWS region](#) in which the Step Functions activity/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 will be one of the following values:
 - `activity` – an [Activity \(p. 59\)](#).
 - `function` – a [Lambda function \(p. 59\)](#).
- `name` is the registered resource name (activity name or Lambda function name).

Note

Step Functions does not support referencing ARNs across partitions (For example: "aws-cn" cannot invoke tasks in the "aws" partition, and vice versa);

Task Types

The following task types are supported:

- [Activity \(p. 59\)](#)
- [Lambda Functions \(p. 59\)](#)

The following sections will provide more detail about each 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 details about any of these fields, see [Specifying Resource ARNs in Tasks \(p. 58\)](#).

Note

activities must be created with Step Functions (using a [CreateActivity](#), API call, or the [Step Functions console](#)) before their first use.

For more information about creating an activity and implementing workers, see [Activities \(p. 41\)](#).

Lambda Functions

Lambda functions execute a function using AWS Lambda. To specify a Lambda function, use the 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 details about any of these fields, see [Specifying Resource ARNs in Tasks \(p. 58\)](#).

For example:

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

```
}
```

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

Choice

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

In addition to the [common state fields \(p. 56\)](#), Choice states introduce these additional fields:

choices (Required)

An array of [Choice Rules \(p. 61\)](#) that determine which state the state machine transitions to next.

Default (Optional, Recommended)

The name of a state to transition to if none of the transitions in `choices` is taken.

Important

Choice states do not support the `End` field. Also, they use `Next` only inside their `choices` field.

Here is an example of a Choice state, with some other states that it transitions to:

```
"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-states-1:123456789012:function:Bar",
  "Next": "NextState"
},
"DefaultState": {
  "Type": "Fail",
  "Cause": "No Matches!"
}
```

In the example, suppose the state machine is started with an input value of:

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

Then AWS Step Functions will transition to the "ValueInTwenties" state, based on the "value" field.

If there are no matches for the `Choice` state's `Choices`, then the state provided in the `Default` field is run instead. If there is no `Default` state provided, then the execution will fail with an error.

Choice Rules

A `Choice` state must have a `Choices` field whose value is a non-empty array, each element of which is a object called a Choice Rule. A Choice Rule contains a comparison (two fields that specify an input variable to be compared, the type of comparison and the value with which to compare it) and a `Next` field, whose value must match a state name in the state machine.

For example:

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

Step Functions looks at each of the Choice Rules in the order listed in the `Choices` field, and 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:

- `StringEquals`
- `StringLessThan`
- `StringGreaterThan`
- `StringLessThanEquals`
- `StringGreaterThanEquals`
- `NumericEquals`
- `NumericLessThan`

- NumericGreaterThan
- NumericLessThanEquals
- NumericGreaterThanEquals
- BooleanEquals
- TimestampEquals
- TimestampLessThan
- TimestampGreaterThan
- TimestampLessThanEquals
- TimestampGreaterThanEquals
- And
- Or
- Not

For each of these operators, the corresponding value must be of the appropriate type: String, number, boolean, or Timestamp (see below). Step Functions will not attempt to match a numeric field to a string value. However, since Timestamp fields are logically strings, it is possible that a field that is thought of as a time-stamp could be matched by a "StringEquals" comparator.

Note that for interoperability, numeric comparisons should not be assumed to work with values outside the magnitude or precision representable using the IEEE 754-2008 "binary64" data type. In particular, integers outside of the range $[-2^{53}+1, 2^{53}-1]$ might fail to compare in the expected way.

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`.

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 itself contain `Next` fields. Using `And`, `Or` and `Not`, you can create complex, nested Choice Rules. However, the `Next` field can only appear 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 \(p. 56\)](#), `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 before 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 \(p. 66\)](#) from the state's input data.

TimestampPath

An absolute time to wait until before beginning the state specified in the `Next` field, specified using a [path \(p. 66\)](#) from the state's input data.

Note

You must specify exactly one of `Seconds`, `Timestamp`, `SecondsPath`, or `TimestampPath`.

For example, the following `wait` state introduces a ten 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"  
}
```

The wait duration does not have to be hard-coded. 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 \(p. 66\)](#) 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, nor do they have need of 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. 56). 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 nor do they 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. 56), `Parallel` states introduce 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 prescribed by the `OutputPath` field (if present) before being used as the state's output. For more information, see [Input and Output Processing](#) (p. 66).

Retry (Optional)

An array of objects, called `Retriers` that define a retry policy in case the state encounters runtime errors. For more information, see [Retrying After an Error](#) (p. 69).

Catch (Optional)

An array of objects, called `Catchers` that define a fallback state which is executed in case the state encounters runtime errors and its retry policy has been exhausted or is not defined. For more information, see [Fallback States](#) (p. 71).

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:

```
"LookupCustomerInfo": {
  "Type": "Parallel",
```

```

"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
      }
    }
  }
],
"Next": "NextState"
}

```

In this example, the `LookupAddress` and `LookupPhone` branches are executed in parallel.

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 Output

A `Parallel` state provides each branch with a copy of its own input data (subject to modification by the `InputPath` field). It generates output which 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 \(p. 66\)](#)).

Here is another example:

```

"FunWithMath": {
  "Type": "Parallel",
  "Branches": [
    {
      "StartAt": "Add",
      "States": {
        "Add": {
          "Type": "Task",
          "Resource": "arn:aws:swf:::task:Add",
          "End": true
        }
      }
    },
    {
      "StartAt": "Subtract",
      "States": {
        "Subtract": {
          "Type": "Task",
          "Resource": "arn:aws:swf:::task:Subtract",
          "End": true
        }
      }
    }
  ]
}

```

```
    }  
  }  
},  
"Next": "NextState"  
}
```

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, due to either 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 will stop the execution with an error.

Input and Output Processing

In this section you will learn how to use paths and reference paths for input and output processing.

Paths

In Amazon States Language, a *path* is a string beginning with `$` that you can use to identify components within a JSON text. Paths follow [JsonPath](#) syntax.

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.

For example, state input data contains the following values:

```
{  
  "foo": 123,  
  "bar": ["a", "b", "c"],  
  "car": {  
    "cdr": true  
  }  
}
```

In this case, the following reference paths would return:

```
$.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.

Paths in InputPath, ResultPath, and OutputPath Fields

To specify how to use part of the state's input and what to send as output to the next state, you can use `InputPath`, `OutputPath`, and `ResultPath`:

- For `InputPath` and `OutputPath`, you must use a [path \(p. 66\)](#) that follows the [JsonPath](#) syntax.
- For `ResultPath`, you must use a [reference path \(p. 66\)](#).

InputPath

The `InputPath` field selects a portion of the state's input to pass to the state's task for processing. If you omit the field, it gets the `$` value, representing the entire input. If you use `null`, the input is discarded (not sent to the state's task) and the task receives JSON text representing an empty object `{}`.

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 ]
```

ResultPath

Usually, if a state executes a task, the task results are sent along as the state's output (which becomes the input for the next task).

If a state doesn't execute a task, the state's own input is sent, unmodified, as its output. However, when you specify a path in the value of a state's `ResultPath` and `OutputPath` fields, different scenarios become possible.

The `ResultPath` takes the results of executing the state's task and places them in the input. Next, the `OutputPath` selects a portion of the input to send as the state's output. The `ResultPath` might add the results of executing the state's task to the input, overwrite an existing part, or overwrite the entire input:

- If the `ResultPath` matches an item in the state's input, only that input item is overwritten with the results of executing the state's task. The entire modified input becomes available to the state's output.
- If the `ResultPath` doesn't match an item in the state's input, an item is added to the input. The item contains the results of executing the state's task. The expanded input becomes available to the state's output.
- If the `ResultPath` has the default value of `$`, it matches the entire input. In this case, the results of the state execution overwrite the input entirely and the input becomes available to pass along.
- If the `ResultPath` is `null`, the results of executing the state are discarded and the input is untouched.

Note

`ResultPath` field values must be [reference paths \(p. 66\)](#).

OutputPath

- If the `OutputPath` matches an item in the state's input, only that input item is selected. This input item becomes the state's output.
- If the `OutputPath` doesn't match an item in the state's input, an exception specifies an invalid path. For more information, see [Errors \(p. 68\)](#).

- If the `OutputPath` has the default value of `$`, this matches the entire input completely. In this case, the entire input is passed to the next state.

Note

For more information about the effect `ResultPath` has on the input for those states that allow it, see [ResultPath \(p. 67\)](#).

- If the `OutputPath` is `null`, a JSON text representing an empty object `{ }` is sent to the next state.

The following example demonstrates how `InputPath`, `ResultPath`, and `OutputPath` fields work in practice. Consider the following input for the current state:

```
{
  "title": "Numbers to add",
  "numbers": { "val1": 3, "val2": 4 }
}
```

In addition, the state has the following `InputPath`, `ResultPath`, and `OutputPath` fields:

```
"InputPath": "$.numbers",
"ResultPath": "$.sum",
"OutputPath": "$"
```

The state's task receives only the `numbers` object from the input. In turn, if this task returns 7, the output of this state is as follows:

```
{
  "title": "Numbers to add",
  "numbers": { "val1": 3, "val2": 4 }
  "sum": 7
}
```

You can slightly modify the `OutputPath`:

```
"InputPath": "$.numbers",
"ResultPath": "$.sum",
"OutputPath": "$.sum"
```

As before, you use the following state input data:

```
{
  "numbers": { "val1": 3, "val2": 4 }
}
```

However, now the state output data is 7:

```
{
  7
}
```

Errors

Any state can encounter runtime errors. Errors can arise because of state machine definition issues (for example, no matching rule in a `Choice` state), task failures (for example, an exception thrown by a

Lambda function) or because of transient issues, such as network partition events. When a state reports an error, the default course of action for AWS Step Functions is to fail the execution entirely.

Error Representation

Errors are identified in Amazon States Language by case-sensitive strings, called Error Names. Amazon States Language defines a set of built-in strings naming well-known errors, all of which begin with the prefix "States.":

Predefined Error Codes

States.ALL

A wild-card that matches any Error Name.

States.Timeout

A `Task` state either ran longer than the "TimeoutSeconds" value, or failed to send a heartbeat for a time 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 may report errors with other names, which must not begin with the prefix "States."

Retrying After an Error

`Task` and `Parallel` states may have a field named `Retry`, whose value must be an array of objects, called Retriers. An individual Retrier represents a certain number of retries, usually at increasing time intervals.

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 and, 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 (default 1).

MaxAttempts (Optional)

A positive integer, representing the maximum number of retry attempts (default 3). If the error recurs more times than specified, retries cease and normal error handling resumes. A value of 0 is permitted and indicates that the error or errors should never be retried.

BackoffRate (Optional)

A number that is the multiplier by which the retry interval increases on each attempt (default 2.0).

Here is an example of a `Retry` field that will make 2 retry attempts after waits of 3 and 4.5 seconds:

```
"Retry" : [  
  {
```

```
    "ErrorEquals": [ "States.Timeout" ],
    "IntervalSeconds": 3,
    "MaxAttempts": 2,
    "BackoffRate": 1.5
  }
]
```

The reserved name `States.ALL` appearing 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.

Here is an example of a `Retry` field that will retry any error except for `States.Timeout`:

```
"Retry" : [
  {
    "ErrorEquals": [ "States.Timeout" ],
    "MaxAttempts": 0
  },
  {
    "ErrorEquals": [ "States.ALL" ]
  }
]
```

Complex Retry Scenarios

A Retrier's parameters apply across all visits to that Retrier in the context of a single state execution. This is best illustrated by an example; 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"
    }
  ]
}
```

Suppose that this task fails five successive times, throwing Error Names "ErrorA", "ErrorB", "ErrorC", "ErrorB" and "ErrorB". The first two errors match the first retrier and cause waits of one and two seconds. The third error matches the second retrier and causes a wait of five seconds. The fourth error matches the first retrier and causes a wait of four seconds. The fifth error also matches the first retrier, but 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.

Note that once the system transitions to another state, no matter how, all Retrier parameters are reset.

Note

You can generate custom error names (such as `ErrorA` and `ErrorB` above) using either an [Activity \(p. 59\)](#) or [Lambda Functions \(p. 59\)](#). For more information, see [Handling Error Conditions Using a State Machine \(p. 27\)](#).

Fallback States

`Task` and `Parallel` states may have a field named `Catch`, whose value must be an array of objects, called `Catchers`.

A `Catcher` contains the following fields:

`ErrorEquals` (Required)

A non-empty array of Strings that match Error Names, specified exactly as with the `Retrier` field of the same name.

`Next` (Required)

A string which must exactly match one of the state machine's state names.

`ResultPath` (Optional)

A [path \(p. 66\)](#) which determines what is sent as input to the state specified by the `Next` field.

When a state reports an error and either there is no `Retry` field, or retries have failed to resolve the error, AWS Step Functions scans through the `Catchers` in the order listed in the array, and 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` appearing 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.

Here is an example of a `Catch` field that will transition to the state named "RecoveryState" when a Lambda function throws an unhandled Java Exception, and otherwise to the "EndState" state.

```
"Catch": [
  {
    "ErrorEquals": [ "java.lang.Exception" ],
    "ResultPath": "$.error-info",
    "Next": "RecoveryState"
  },
  {
    "ErrorEquals": [ "States.ALL" ],
    "Next": "EndState"
  }
]
```

Each `Catcher` can specify multiple errors to handle.

When AWS Step Functions transitions to the state specified in a `Catcher`, it sends along as input a JSON text that is different than what it would normally send to the next state when there was no error. This JSON text represents an object containing a field `Error` whose value is a string containing the error name. The object will also, usually, contain a field `Cause` that has a human-readable description of the error. We refer to this object as the Error Output.

In this example, the first `Catcher` contains a `ResultPath` field. This works in a similar fashion to a `ResultPath` field in a state's top level—it takes the results of executing the state and overwrites a portion

of the state's input, or all of the state's input, or 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.

So in the example, for the first Catcher the Error Output will be added to the input as a field named `error-info` (assuming there is not already a field by that name in the input) and the entire input will be sent to `RecoveryState`. For the second Catcher, the Error Output will overwrite the input and so just the Error Output will be sent to `EndState`. When not specified, the `ResultPath` field defaults to `$` which selects, and so overwrites, the entire input.

When a state has both Retry and Catch fields, Step Functions uses any appropriate Retriers first and only applies the matching Catcher transition if the retry policy fails to resolve the error.

Limits

AWS Step Functions places limits on the sizes of certain state machine parameters, such as the number of API calls 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, they 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 \(p. 73\)](#)
- [Limits Related to Accounts \(p. 74\)](#)
- [Limits Related to State Machine Executions \(p. 74\)](#)
- [Limits Related to Task Executions \(p. 74\)](#)
- [Limits Related to API Action Throttling \(p. 75\)](#)

General Limits

Limit	Description
State machine name	State machine names must be 1-80 characters in length, must be unique for your account and region, and must not contain any of the following: <ul style="list-style-type: none">• Whitespace• Whitespace characters (? *)• Bracket characters (< > { } [])• Special characters (: ; , \ ^ ~ \$ # % & ` ")• Control characters (\\u0000 - \\u001f or \\u007f - \\u009f).

Limits Related to Accounts

Limit	Description
Maximum number of state machines and activities	10,000
Maximum number of API calls	Beyond infrequent spikes, applications may be throttled if they make a large number of API calls 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.

Limits Related to State Machine Executions

Limit	Description
Maximum open executions	1,000,000
Maximum execution time	1 year
Maximum execution history size	25,000 events
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.
Maximum executions displayed in Step Functions console	The Step Functions console displays a maximum of 1,000 executions per state machine. If you have more than 1,000 executions, use the Step Functions API actions or the AWS CLI to display all of your executions.

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 open activities	1,000 per execution. This limit includes both activities that have been scheduled and those being processed by workers.

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 region.

API Name	Bucket Size	Refill Rate per Second
CreateActivity	100	1
CreateStateMachine	100	1
DeleteActivity	100	1
DeleteStateMachine	100	1
DescribeActivity	200	1
DescribeExecution	200	1
DescribeStateMachine	200	1
GetActivityTask	1000	10
GetExecutionHistory	50	1
ListActivities	100	1
ListExecutions	100	1
ListStateMachines	100	1
SendTaskFailure	1000	10
SendTaskHeartbeat	1000	10
SendTaskSuccess	1000	10
StartExecution	100	2
StopExecution	100	2

Monitoring and Logging

This section provides information about monitoring and logging Step Functions.

Topics

- [Monitoring Step Functions Using CloudWatch \(p. 76\)](#)
- [Logging Step Functions using AWS CloudTrail \(p. 82\)](#)

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, you should 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 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 \(p. 77\)](#)
- [Metrics that Report a Count \(p. 77\)](#)
- [State Machine Metrics \(p. 77\)](#)
- [Viewing Metrics for Step Functions \(p. 79\)](#)
- [Setting Alarms for Step Functions \(p. 80\)](#)

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.

State Machine Metrics

The following metrics are available for Step Functions state machines:

Execution Metrics

The `AWS/States` namespace includes the following metrics for Step Functions executions:

Metric	Description
<code>ExecutionsAborted</code>	The number of aborted or terminated executions.
<code>ExecutionsFailed</code>	The number of failed executions.
<code>ExecutionsStarted</code>	The number of started executions.
<code>ExecutionsSucceeded</code>	The number of successfully completed executions.
<code>ExecutionTime</code>	The interval, in milliseconds, between the time the execution starts and the time it closes.

Metric	Description
ExecutionsTimedOut	The number of executions that time out for any reason.

Dimension for Step Functions Execution Metrics

Dimension	Description
StateMachineArn	The 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.

Metric	Description
LambdaFunctionScheduleTime	The interval, in milliseconds, for which the Lambda function stays in the schedule state.
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.
LambdaFunctionsHeartbeatTimedOut	The number of Lambda functions that time out due to a heartbeat timeout.
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.

Viewing Metrics for Step Functions

1. Open the AWS Management Console and navigate to **CloudWatch**.
2. Choose **Metrics** and on the **All Metrics** tab, choose **States**.

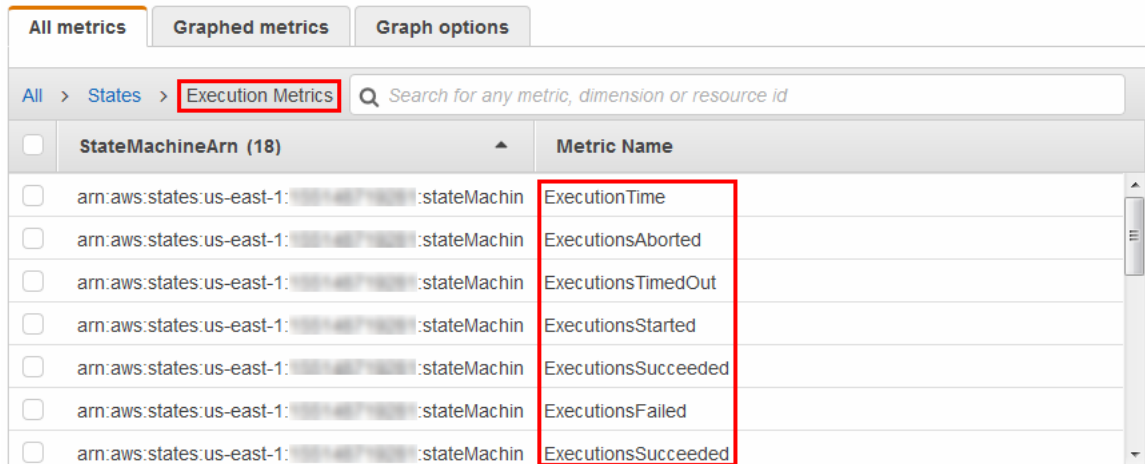
The screenshot shows the AWS CloudWatch console interface. On the left sidebar, the 'Metrics' option is highlighted with a red box. The main area displays an empty graph titled 'Untitled graph' with a message: 'Your CloudWatch graph is empty. Select some metrics to appear here.' Below the graph, there are tabs for 'All metrics', 'Graphed metrics', and 'Graph options'. Under the 'All metrics' tab, there is a search bar and a list of 56 metrics. Two categories are visible: 'Lambda' with 20 metrics and 'States' with 36 metrics. The 'States' category is highlighted with a red box.

If you have run 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.



- 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 drop-down list to display values as lines, stacked areas, or numbers (values).

- To view the details about a graph, hover over the metric color code which appears below the graph.

■ ExecutionsAborted ■ ExecutionsStarted ■ ExecutionsSucceeded ■ ExecutionsTimedOut

The metric's details are displayed.



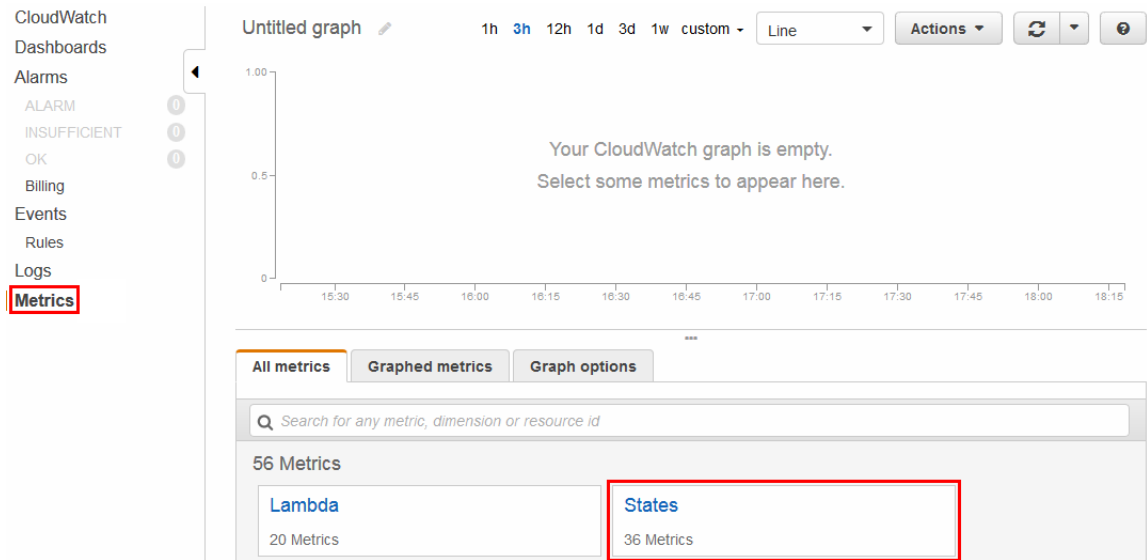
For more information about working with CloudWatch metrics, see [Using Amazon CloudWatch Metrics](#) in the *Amazon CloudWatch User Guide*.

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

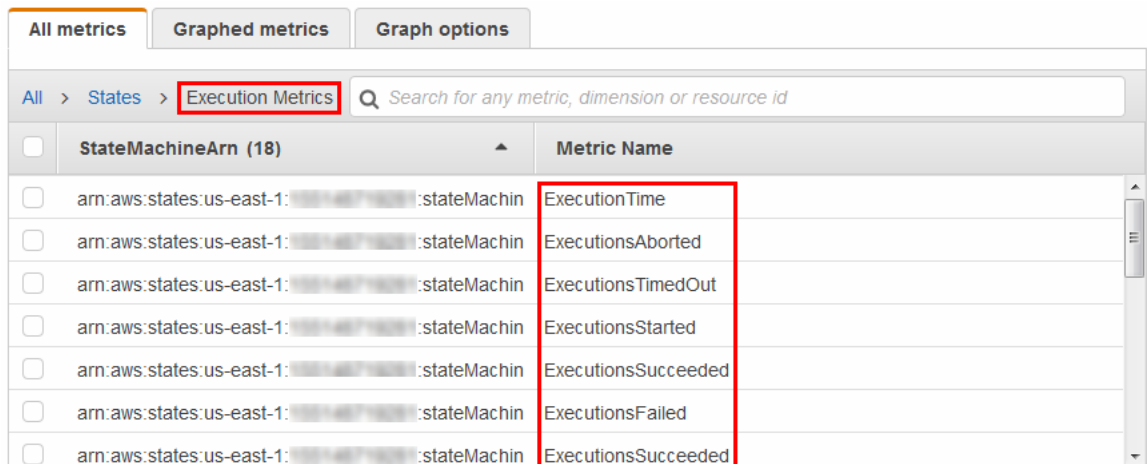
1. Open the AWS Management Console and navigate to **CloudWatch**.
2. Choose **Metrics** and on the **All Metrics** tab, choose **States**.




If you have run any executions recently, you will see up to three types of metrics:

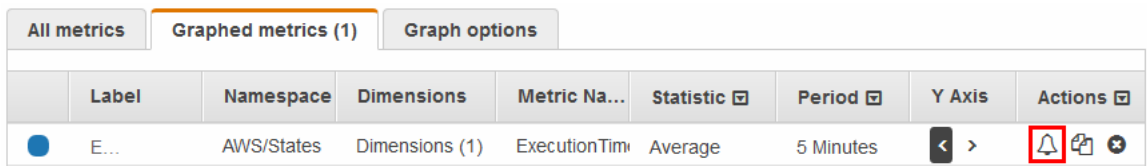
- Execution Metrics
- Activity Function Metrics
- Lambda Function Metrics

3. Choose a metric type to see a list of metrics.



4. Choose a metric and then choose **Graphed metrics**.

5. Choose  next to a metric on the list.



The **Create Alarm** dialog box is displayed.

Create Alarm [X]

1. Select Metric 2. Define Alarm

Alarm Threshold

Provide the details and threshold for your alarm. Use the graph on the right to help set the appropriate threshold.

Name:

Description:

Whenever: ExecutionTime

is:

for: consecutive period(s)

Actions

Define what actions are taken when your alarm changes state.

Notification Delete

Whenever this alarm:

Send notification to: [New list](#) [Enter list](#) ⓘ

Alarm Preview

This alarm will trigger when the blue line goes up to or above the red line for a duration of 5 minutes

ExecutionTime >= 0

Namespace: AWS/States

StateMachine Arn:

Metric Name:

Period:

Statistic: Standard Custom

6. Enter the values for the **Alarm threshold** and **Actions** and then choose **Create Alarm**.

For more information about setting and using CloudWatch alarms, see [Creating Amazon CloudWatch Alarms](#) in the *Amazon CloudWatch User Guide*.

Logging Step Functions using AWS CloudTrail

AWS Step Functions is integrated with AWS CloudTrail, a service that captures specific API calls and delivers log files to an Amazon S3 bucket that you specify. With the information collected by CloudTrail, you can determine what request was made to Step Functions, the IP address from which the request was made, who made the request, when it was made, and so on.

To learn more about CloudTrail, including how to configure and enable it, see the AWS [CloudTrail User Guide](#).

Step Functions Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to specific Step Functions actions are tracked in CloudTrail log files. Step Functions actions are written, together with other AWS service records. CloudTrail determines when to create and write to a new file based on a time period and file size.

The following actions are supported:

- [CreateActivity](#)
- [CreateStateMachine](#)
- [DeleteActivity](#)
- [DeleteStateMachine](#)
- [StartExecution](#)
- [StopExecution](#)

Every log entry contains information about who generated the request. The user identity information in the log 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 [userIdentity](#) element in the *AWS CloudTrail User Guide*.

You can store your log files in your S3 bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted with [Amazon S3 server-side encryption](#).

If you want to be notified upon log file delivery, you can configure CloudTrail to publish Amazon SNS notifications when new log files are delivered. For more information, see [Configuring Amazon SNS Notifications for CloudTrail](#).

You can also aggregate Step Functions log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket. For more information, see [Receiving CloudTrail Log Files from Multiple Regions](#) and [Receiving CloudTrail Log Files from Multiple Accounts](#).

Understanding Step Functions Log File Entries

CloudTrail log files contain one or more log entries. Each entry lists multiple JSON-formatted events. A log entry 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. The log entries are not an ordered stack trace of the public API calls, so they do not 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-4cf4-8523-376469410c25",  
    "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

This section provides information about Step Functions security.

Topics

- [Creating IAM Roles for Use with AWS Step Functions \(p. 87\)](#)

Creating IAM Roles for Use with AWS Step Functions

AWS Step Functions is capable of executing code and accessing AWS resources (such as data stored in Amazon S3 buckets), so to maintain security, you must grant Step Functions access to those resources. You do this for Step Functions with an IAM role.

In the tutorials for Step Functions in this document, you made use of automatically generated IAM roles that were valid for the region in which you created the state machine. If you wish to create your own IAM role for use with your state machine, this section outlines the steps needed to do that.

Steps to Create a Role for Use with Step Functions

In this example, you will create an IAM role with permission to invoke a Lambda function.

1. Open the [IAM console](#).
2. Choose **Roles** in the left pane, then choose **Create New Role**.
3. On **Set Role Name**, type a name for your role, such as `states-lambda-role`, and choose **Next Step**.
4. On **Select Role Type**, choose `AWS_SWF` from the list.

Note

Currently, there is no AWS service role registered with the IAM console for the Step Functions service. You must select one of the existing role policies and manually modify it after the role is created.

5. On **Attach Policy**, choose the **AWSLambdaRole** policy, and then choose **Next Step**. If you are creating a state machine for a different purpose, please choose the appropriate policy here.
6. On **Review**, choose **Create Role**. You always get a final chance to change the name and policy for your role.

Next, you will edit the trust relationship for the Step Functions role you created.

7. From the [IAM console](#), choose the name of the role that you just created (`states-lambda-role`) from the list. This will open the role's detail page.
8. Choose the **Trust Relationships** tab and then choose **Edit Trust Relationship**. You will see a trust relationship such as:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "swf.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

9. Under the **Principal** section, replace `swf.amazonaws.com` with `states.REGION.amazonaws.com` (where REGION is AWS region you are working in), resulting in the following trust relationship (for example):

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "states.us-east-1.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

10. Choose **Update Trust Policy**.

For more information about IAM permissions and policies, see [Access Management](#) in the IAM User Guide.

Document History

This topic lists major changes to the *AWS Step Functions Developer Guide*.

Latest documentation update: June 21, 2017

Change	Description	Date Changed
Update	Corrected and clarified information in the following sections: <ul style="list-style-type: none"> • Getting Started (p. 3) • Handling Error Conditions Using a State Machine (p. 27) • States (p. 55) • Error Handling (p. 45) 	June 21, 2017
Update	Rewrote all tutorials to match the Step Functions console refresh.	June 12, 2017
New feature	Step Functions is available in Asia Pacific (Sydney).	June 8, 2017
Update	Restructured the Amazon States Language (p. 53) section.	June 7, 2017
Update	Corrected and clarified information in the Creating an Activity State Machine (p. 20) section.	June 6, 2017
Update	Corrected the code examples in the Examples Using Retry and Using Catch (p. 49) section.	June 5, 2017
Update	Restructured this guide using AWS documentation standards.	May 31, 2017
Update	Corrected and clarified information in the Parallel (p. 64) section.	May 25, 2017
Update	Merged the Paths and Filters sections into the Input and Output Processing (p. 66) section.	May 24, 2017
Update	Corrected and clarified information in the Blueprints (p. 39) section.	May 16, 2017
Update	Corrected and clarified information in the Monitoring Step Functions Using CloudWatch (p. 76) section.	May 15, 2017
Update	Updated the <code>GreeterActivities.java</code> worker code in the Creating an Activity State Machine (p. 20) tutorial.	May 9, 2017
Update	Added an introductory video to the What is AWS Step Functions? (p. 1) section.	April 19, 2017

Change	Description	Date Changed
Update	Corrected and clarified information in the following tutorials: <ul style="list-style-type: none"> • Getting Started (p. 3) • Creating a Lambda State Machine (p. 9) • Creating an Activity State Machine (p. 20) • Handling Error Conditions Using a State Machine (p. 27) 	April 19, 2017
Update	Added information about Lambda blueprints to the Creating a Lambda State Machine (p. 9) and Handling Error Conditions Using a State Machine (p. 27) 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. 74) .	March 31, 2017
New feature	<ul style="list-style-type: none"> • Step Functions supports executing state machines by setting Step Functions as Amazon CloudWatch Events targets. • Added the new tutorial Starting a State Machine Execution Using CloudWatch Events (p. 25). 	March 21, 2017
New feature	<ul style="list-style-type: none"> • Step Functions allows Lambda function error handling as the preferred error handling method. • Updated the Handling Error Conditions Using a State Machine (p. 27) tutorial and the Error Handling (p. 45) section. 	March 16, 2017
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: <ul style="list-style-type: none"> • Getting Started (p. 3) • Creating a Lambda State Machine (p. 9) • Creating an Activity State Machine (p. 20) • Handling Error Conditions Using a State Machine (p. 27) 	February 23, 2017
New feature	<ul style="list-style-type: none"> • The State Machines page of the Step Functions console includes the Copy to New and Delete buttons. • Updated the screenshots to match the console changes. 	February 23, 2017
New feature	<ul style="list-style-type: none"> • Step Functions supports creating APIs using API Gateway. • Added the new tutorial Creating a Step Functions API Using API Gateway (p. 33). 	February 14, 2017
New feature	<ul style="list-style-type: none"> • Step Functions supports integration with AWS CloudFormation. • Added the new tutorial Creating a Lambda State Machine Using AWS CloudFormation (p. 15). 	February 10, 2017
Update	Clarified the current behavior of the <code>ResultPath</code> and <code>OutputPath</code> fields in relation to <code>Parallel</code> states.	February 6, 2017
Update	<ul style="list-style-type: none"> • Clarified state machine naming restrictions in tutorials. • Corrected some code examples. 	January 5, 2017

Change	Description	Date Changed
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