AWS Step Functions Developer Guide

Release 1.0

Amazon Web Services

Dec 01, 2016

CHAPTER 1

Contents

1.1 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.

1.1.1 Overview of Step Functions

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

- Step Functions is based on the concepts of *tasks* and *state machines*.
- You define state machines using a JSON-based *language*.
- 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.

1.2 Getting Started with AWS Step Functions

This is a quick tutorial that introduces the basics of working with AWS Step Functions. In this tutorial, you will create a simple state machine that can run on its own, using a *Pass* state. The *Pass* state is the simplest

state and represents a "no-op".

Contents

- *Prerequisites*
- *Step 1: Create a State Machine*
- *Step 2: Start an Execution*

1.2.1 Prerequisites

To complete this tutorial you will need:

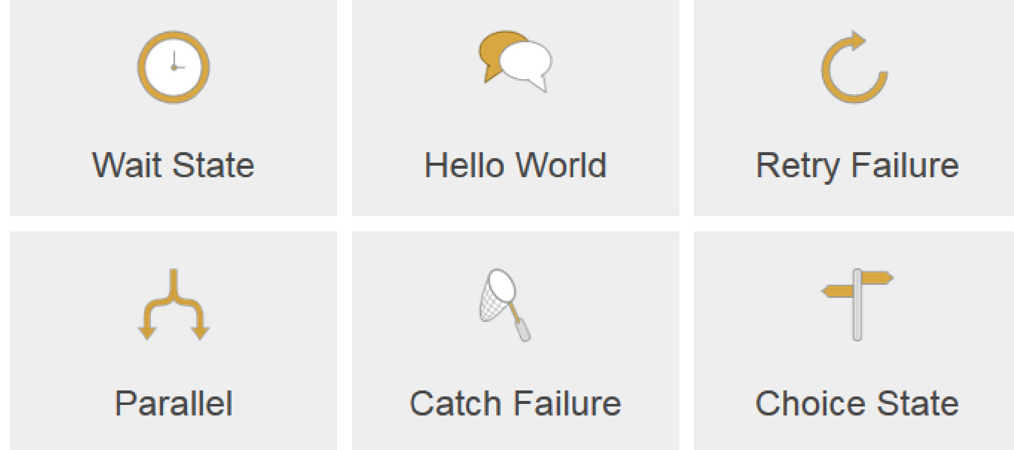
- An AWS account. If you haven't yet signed up for AWS, go to http://aws.amazon.com/ and click *Sign In to the Console*.

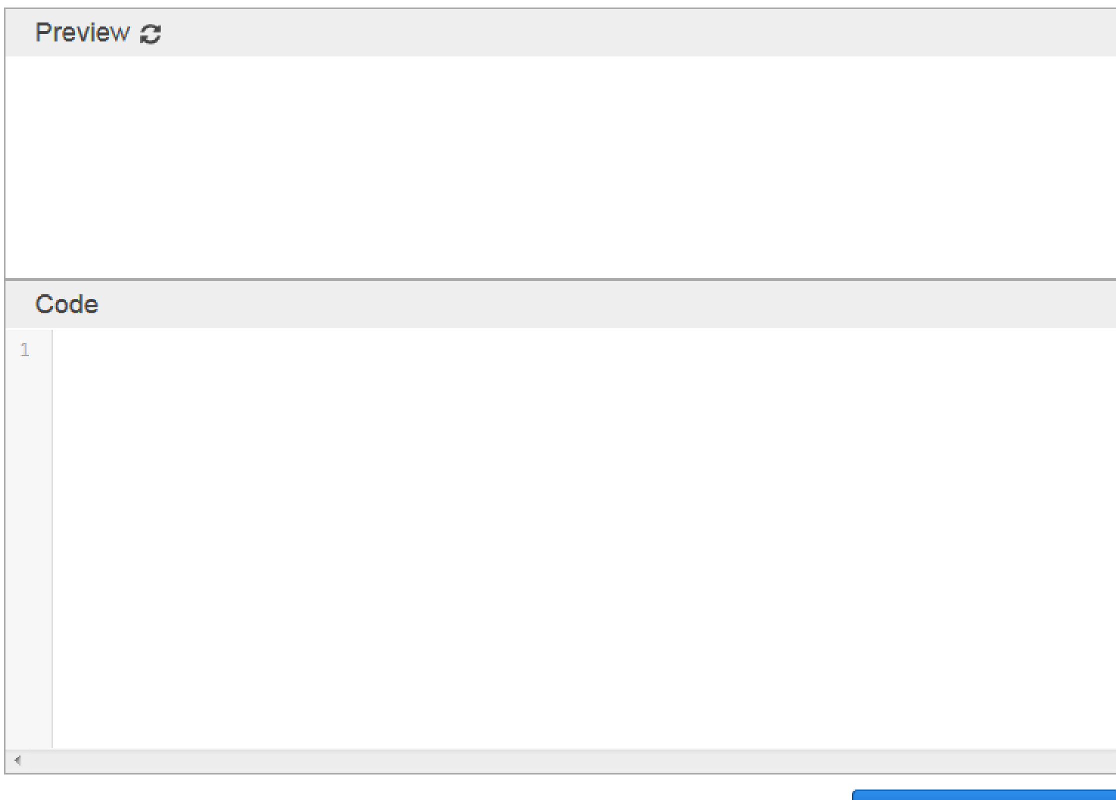
1.2.2 Step 1: Create a State Machine

First, you will create a state machine using the Step Functions console.

To create the state machine

1. Sign in to the Step Functions console.
2. Click *Get Started*. This will display the *Create State Machine* screen.


Dashboard
Create State Machine
Give a name to your state machine
Enter your name here
You can now create your own state machine with your own code or choose a blueprint below
Wait State
Hello World
Retry Failure
Parallel
Catch Failure
Choice State



3. In the box below *Give a name to your State Machine* type a name, such as “HelloStateMachine”.

Note: State machine names must be *unique* for your account and region.

4. Click the *Hello World* blueprint. The available blueprints enable you to begin with an example state machine template.

   The *Code* pane will be populated with the Amazon States Language description of the state machine:

```
{
  "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:REGION:ACCOUNT_ID:function:FUNCTION_
↪NAME",
      "End": true
    }
  }
}
```

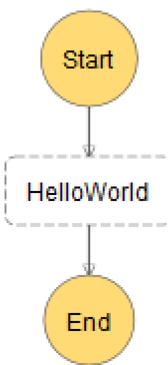
   The blueprint for this state machine consists of a single *Task* state named *HelloWorld*. In this tutorial

you are not going to execute a task, but will simply use a *Pass* state to inject results.

5. In the code, change the state's `Type` field value from `Task` to `Pass`, change the `Resource` field to a `Result` field and then change its value so you end up with something like this:

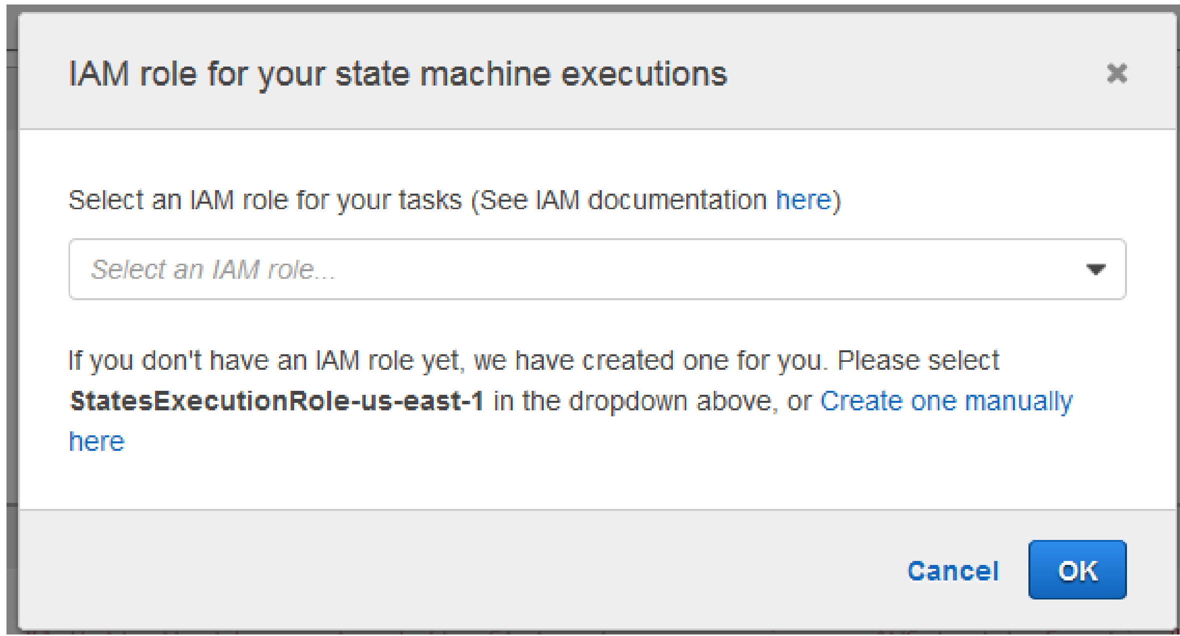
```
{
  "Comment": "A Hello World example of the Amazon States Language using␣
↪a Pass State",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Pass",
      "Result": "Hello, AWS Step Functions!",
      "End": true
    }
  }
}
```

6. Verify that the *Preview* pane displays the following graph of your state machine structure. If you don't see this, click the refresh icon in the *Preview* pane.



The graph in the *Preview* pane helps you to verify that your Amazon States Language code is describing your state machine correctly.

7. Click the *Create State Machine* button. An *IAM role for your state machine executions* dialog box appears.



8. In the *Select an IAM role for your tasks* list, select the role (`StatesExecutionRole-REGION`) that has been automatically created for you by Step Functions and then click *OK* to finish creating your state machine.

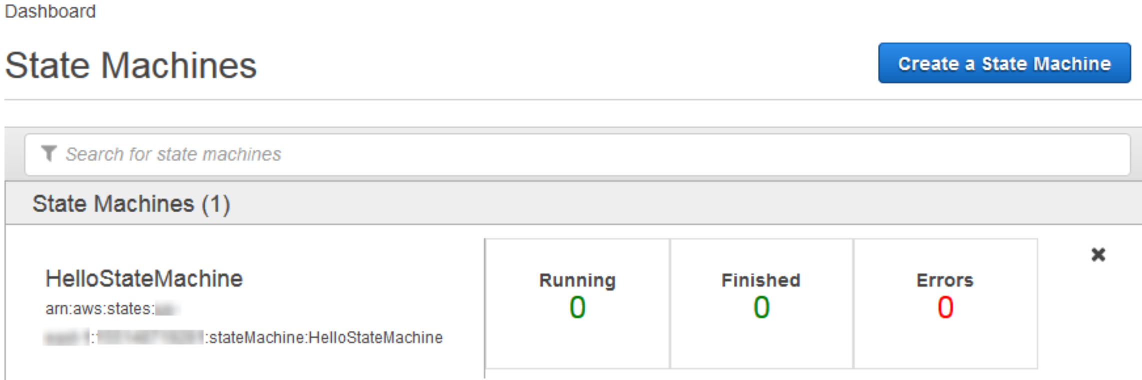
Note: If you delete the IAM role that has been automatically created for you, there is no way for Step Functions to re-create it for you at a later time. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), there is no way for Step Functions to restore its original settings at a later time. For more information on creating an IAM role by hand see *Creating IAM Roles for Use with AWS Step Functions*.

1.2.3 Step 2: Start an Execution

As soon as you finish creating a state machine, you will see an acknowledgment page. From here, you can start an execution using the *New execution* button. You can also start a new execution at any time from the state machine's detail page, which can be reached from the dashboard.

To start a new execution using the console

1. In the left pane, click *Dashboard*. There, you will see the state machine you just created.



2. Click the state machine to see its detail page.
3. From your state machine's detail page, click *New execution*. The *New execution* pane will open.

Dashboard 〉 HelloStateMachine 〉 new

HelloStateMachine

New execution

Enter your execution id here

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

Start Execution

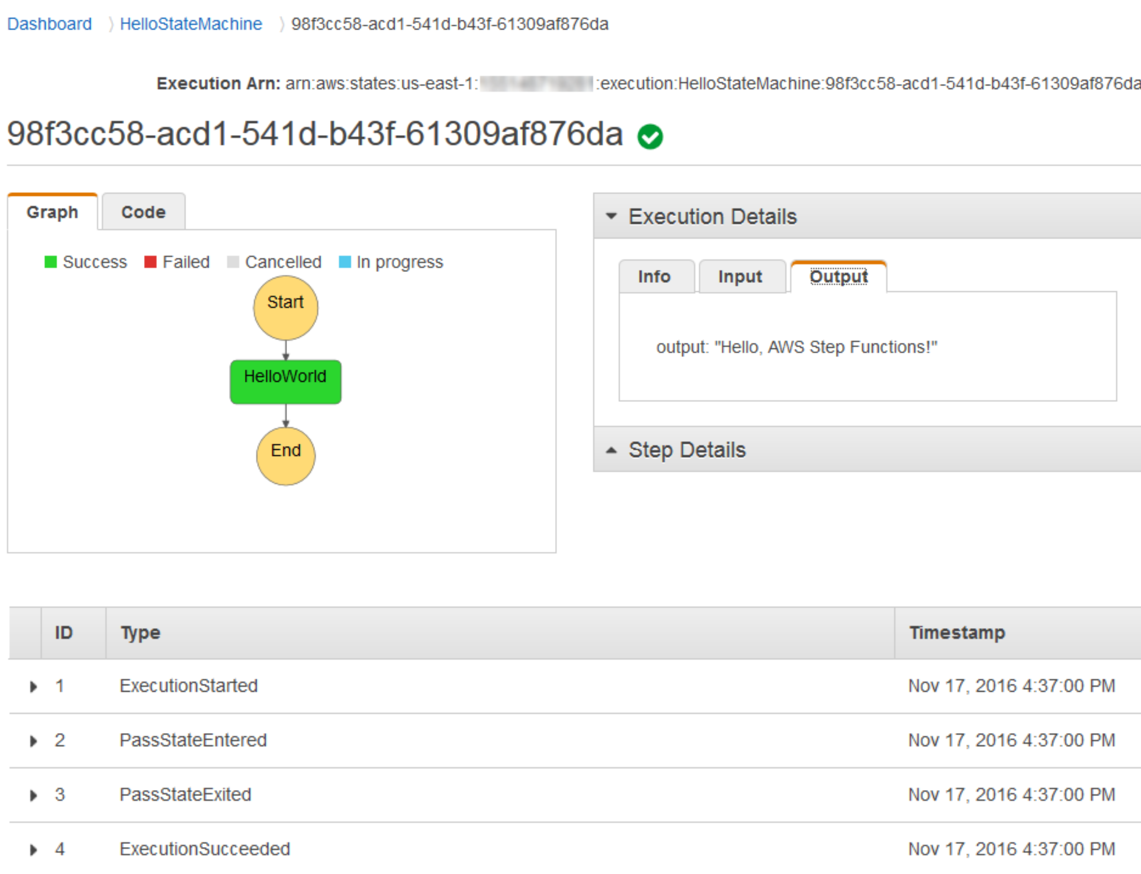
Tip: To help identify your execution, you can optionally enter an id for it. If you don't, a unique id will be generated for you. To set the id, use the box labelled *Enter your execution id here*.

4. Below the execution input area, click the *Start Execution* button to start the execution. A new

execution of your state machine will start, and a new page will appear, showing your open execution.

5. In the *Execution Details* section, click the *Info* tab to see the *Execution Status* and when the execution started and closed.

6. Click the *Output* tab to see the results of your execution:



Congratulations, you have implemented and executed your first state machine!

1.3 Tutorial: A Lambda State Machine

In this tutorial, you will create a state machine that uses an 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*, easy to write, and don't require you to deploy code to a server instance. You can just write code in the AWS Management Console or your favorite editor, and AWS handles the details of providing a computing environment and running your function.

Contents

• *Prerequisites*
• *Step 1: Create an IAM Role for Lambda*

- *Step 2: Create a Lambda Function for Hello World*
- *Step 3: Create a State Machine*
- *Step 4: Start an Execution*

1.3.1 Prerequisites

To complete this tutorial, you will need:

- An AWS account. If you haven't yet signed up for AWS, go to http://aws.amazon.com/ and click *Sign In to the Console*.

1.3.2 Step 1: Create an IAM Role for Lambda

Both Lambda and AWS Step Functions are capable of executing code and accessing AWS resources (such as data stored in Amazon S3 buckets.) To maintain security, you must grant Lambda and Step Functions access to those resources. In the first tutorial, that was done automatically for Step Functions—an IAM role was created when you created the state machine.

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 created a state machine. So you will create one for Lambda now.

To create a role for use with Lambda

1. Open the IAM console.
2. Choose *Roles* in the left pane, then click the *Create New Role* button to begin the role-creation process.
3. On *Set Role Name*, type a name for your role, such as `lambda-role`, and click *Next Step*.
4. On *Select Role Type*, choose *AWS Lambda* from the list.

   Note: This will cause you to skip over the *Establish Trust* step. The role is automatically provided with a trust relationship that allows Lambda to use the role.

5. On *Attach Policy*, don't attach any policy. Click *Next Step*.
6. On *Review*, you get a final chance to change the name and policy for your role. Click *Create Role*.

Your role should now appear in the list of roles in the IAM console.

1.3.3 Step 2: Create a Lambda Function for Hello World

The function that you create will take some input (a name) and will return a greeting that includes the input value.

To create the Lambda function for Hello World

1. Open the Lambda console.

   If you have never created a Lambda function before, you will be greeted with a screen with a single button *Get Started Now* that allows you to create your first Lambda function.

2. If this is your first Lambda function, then click the *Get Started Now* button. Otherwise, click the *Create a Lambda function* button. The first screen that appears is the *Select blueprint* screen. You won't need a blueprint because you will type the code for your function by hand.

3. In the left pane, click *Configure function*

4. On the screen that appears, type your function's *Name* and *Description*, and choose the *Runtime*, using the following data:

Field	Value
Name	`HelloFunction`
Description	`Say "Hello" to someone.`
Runtime	*Node.js 4.3*

5. Next, type the 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 that's passed in to your function. (You will provide input data for this function later, when you perform *Step 4: Start an Execution*.)

   The *callback* method can be used to return a value from your function. Here, you return the assembled greeting.

6. In the *Lambda function handler and role* section, open the *Role* drop-down list and select *Choose an existing role*.

7. Open the *Existing role* drop-down list and choose the Lambda role that you created earlier (`lambda-role`).

   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. Revisit *Step 1: Create an IAM Role for Lambda* to verify or edit the trust relationship for your role.

8. In the *Advanced settings* section, accept the default values. Note that you can set a *Timeout* value in seconds. This timeout value can be set to restrict how long your function can take to execute before it is considered a failure.

   The default value of `3` seconds is fine for your function.

9. Click *Next* to review your function, and then click *Create function* to finish creating your Lambda function.

Once your Lambda function has been created, its Amazon Resource Name (ARN) will be displayed in the top-right corner of the detail screen. It will look similar to this:

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

10. Copy your function's ARN to use later when creating your state machine.

Tip: For more information about AWS Lambda, see the Lambda Developer Guide.

Optional: Test Your Lambda Function

If you want to, you can test your Lambda function now to see it in operation—a good idea if you're developing a Lambda function to use in a real state machine!

To test your Lambda function

1. At the top of your Lambda function's detail screen, click the *Test* button.

   The *Input sample event* screen will appear, pre-loaded with sample data:

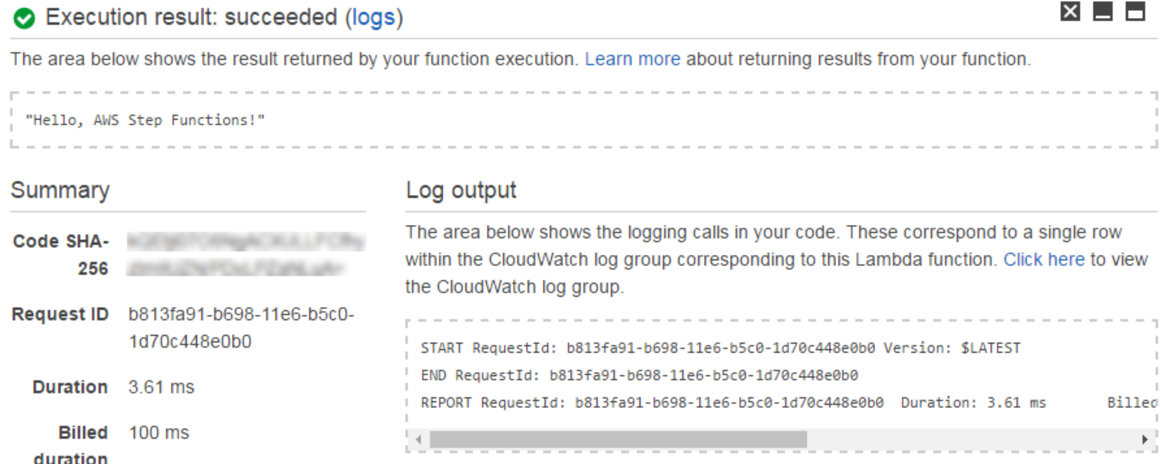
```
{
    "key3": "value3",
    "key2": "value2",
    "key1": "value1"
}
```

2. Replace the sample data with the following (or use your own name in place of "AWS Step Functions"):

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

Note: The "who" entry corresponds to the `event.who` field that's used in your Lambda function to complete the greeting. You will use this same input data when running the function as a Step Functions task.

3. Click *Save and test* to test your Lambda function with the data. The results of the test appear at the bottom of the screen, so you may have to scroll the window to see them:

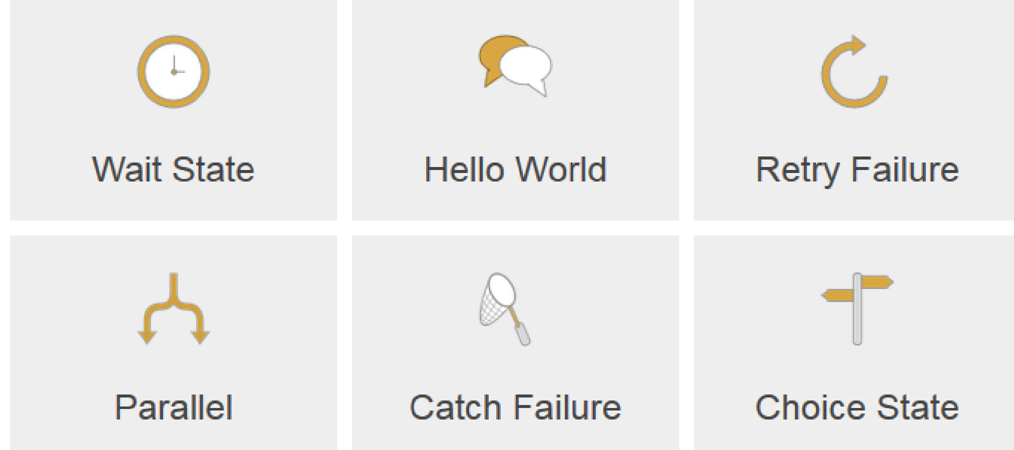


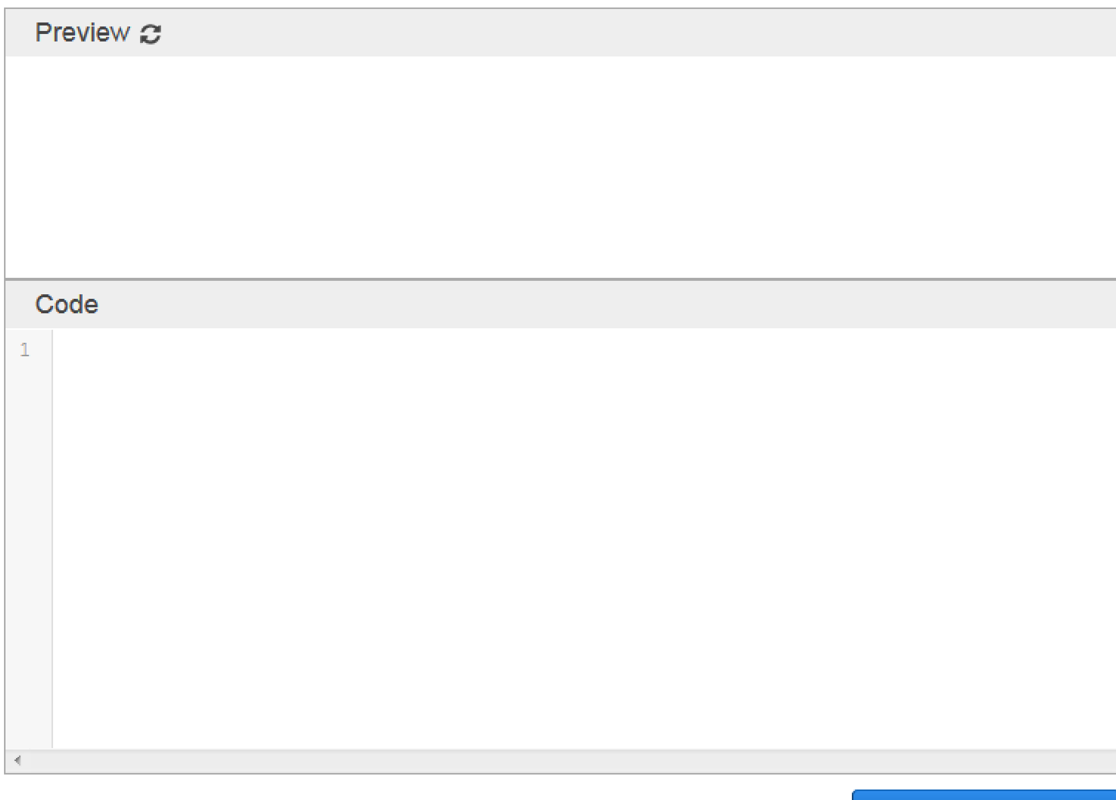
1.3.4 Step 3: Create a State Machine

Next, using the Step Functions console, you will create a state machine that contains a single *Task* state. You will put a reference to your Lambda function in the *Task* state, causing the Lambda function to be invoked when an execution of the state machine reaches that state.

To create the state machine

1. Open the Step Functions console.
2. Click *Get Started*. This will display the *Create State Machine* screen. (If you have already created a state machine using Step Functions, click *Dashboard* in the left pane then click *Create a State Machine*.


Dashboard
Create State Machine
Give a name to your state machine
Enter your name here
You can now create your own state machine with your own code or choose a blueprint below
Wait State
Hello World
Retry Failure
Parallel
Catch Failure
Choice State



3. In the box below *Give a name to your State Machine* type a name, such as "LambdaStateMachine".

Note: State machine names must be *unique* for your account and region.

4. Click the *Hello World* blueprint. The available blueprints enable you to begin with an example state machine template.

The *Code* pane will be populated with the Amazon States Language description of the state machine:

```
{
  "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:REGION:ACCOUNT_ID:function:FUNCTION_
↪NAME",
      "End": true
    }
  }
}
```

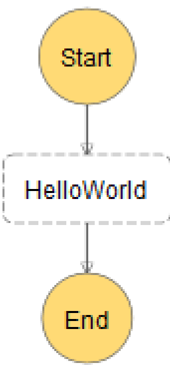
The state machine blueprint consists of a single *Task* state, called *HelloWorld*. The *Resource*

parameter will be used to reference the ARN of the Lambda function, *HelloFunction*, that you created earlier.

5. In the code, replace the value of the `Resource` field (`arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME`) with the ARN of the Lambda function that you saved previously. When you hover over the current value, you should see a pop-up window that allows you to select and auto-complete the new value from a list of Lambda ARN's. Delete the old value, then click the appropriate ARN to fill in the new. Your edited code should look similar to this:

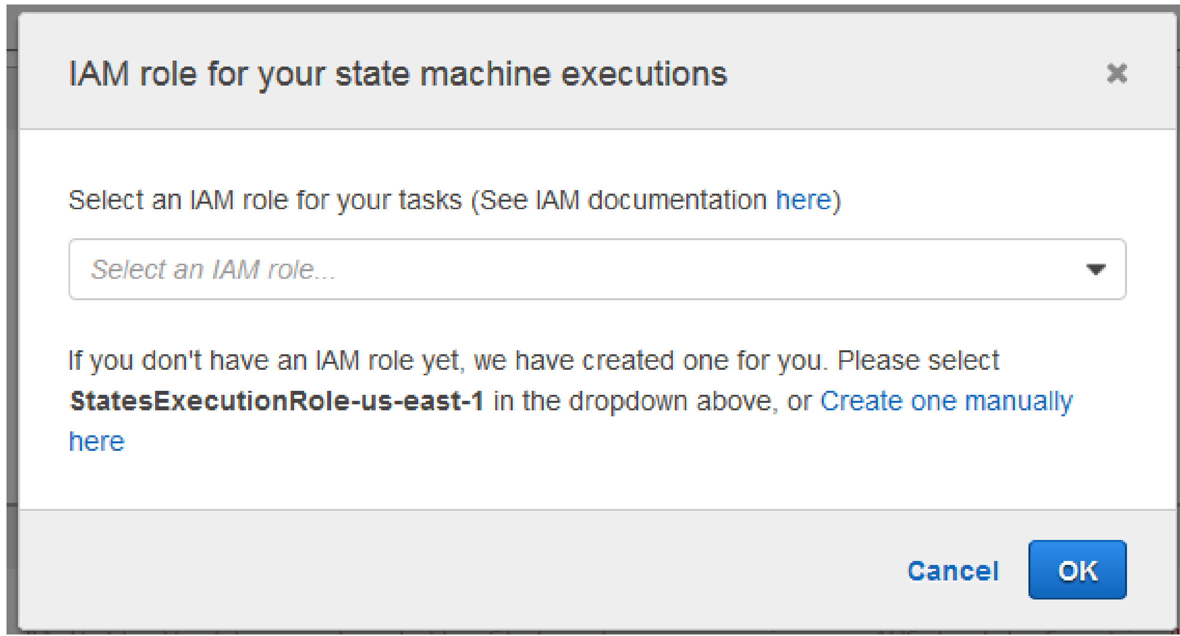
```
{
  "Comment": "A Hello World example of the Amazon States Language using␣
↪an AWS Lambda Function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:
↪HelloFunction",
      "End": true
    }
  }
}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

6. Verify that the *Preview* pane displays the following graph of your state machine. If you don't see the graph, click the refresh icon in the *Preview* pane.



The graph helps you verify that your Amazon States Language code describes your state machine correctly.

7. Click the *Create State Machine* button. A pop-up window will appear, titled *IAM role for your state machine executions*.



8. In the *Select an IAM role for your tasks* drop-down list, select the role (`StatesExecutionRole-REGION`) that has been automatically created for you by Step Functions and then click *OK* to finish creating your state machine.

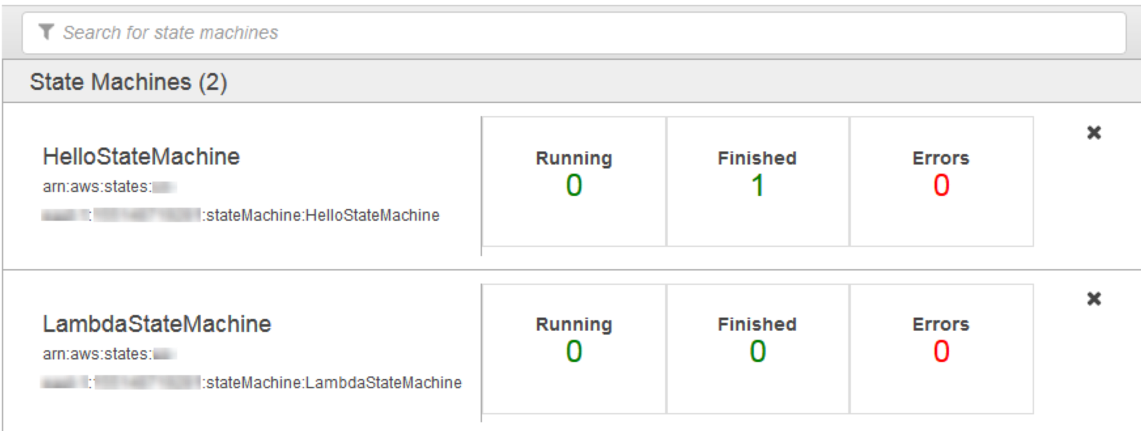
Note: If you delete the IAM role that has been automatically created for you, there is no way for Step Functions to re-create it for you at a later time. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), there is no way for Step Functions to restore its original settings at a later time. For more information on creating an IAM role by hand, see *Creating IAM Roles for Use with AWS Step Functions*.

1.3.5 Step 4: Start an Execution

As soon as you finish creating a state machine, you will see an acknowledgment page. From here, you can start a new execution of your state machine using the *New execution* button. You can also start a new execution at any time from the state machine's detail page, which can be reached from the dashboard.

To start a new execution using the console

1. In the left pane, click *Dashboard*. You will see the state machine you just created.



2. Click the state machine (*LambdaStateMachine*) to see its detail page.
3. From your state machine's detail page, click *New execution*. The *New execution* pane will open and display an execution input area.

Dashboard 〉 LambdaStateMachine 〉 new

LambdaStateMachine

New execution

Enter your execution id here

```
{
    "Comment": "Insert your JSON here"
}
```

Start Execution

Tip: To help identify your execution, you can optionally enter an id for it. If you don't, a unique id will be generated for you. To set the id, use the box labelled *Enter you execution id here*.

---

4. In the execution input area, change the JSON to:

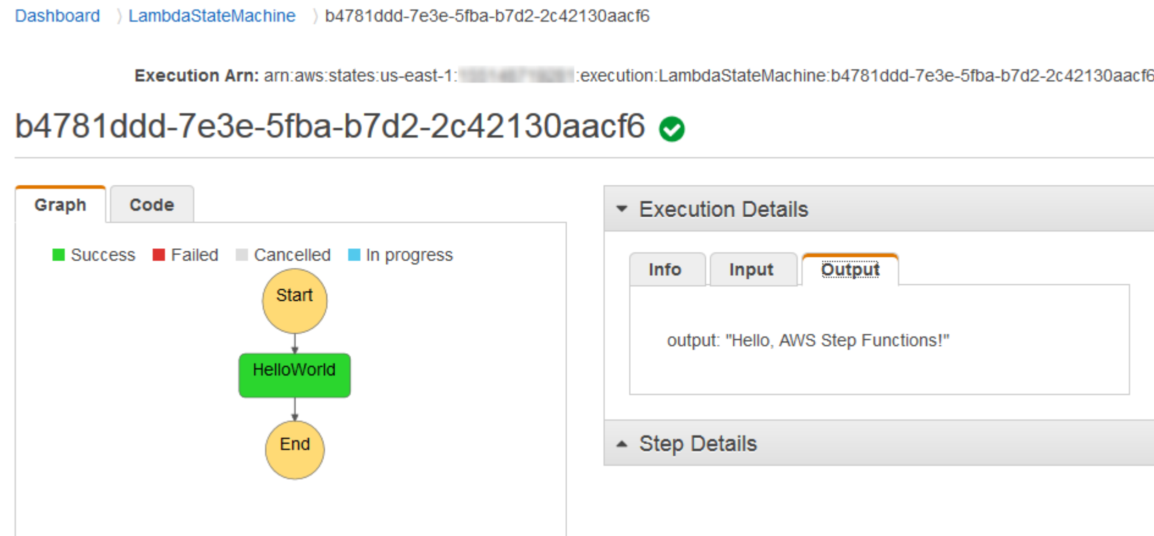
```
{
    "who" : "AWS Step Functions"
}
```

   You may recall that "who" was the key name that your *Lambda function* uses to get the name of the person/thing to greet. In this case, you have set its value to "AWS Step Functions", but you can use any name you want to.

5. Below the execution input area, click the *Start Execution* button to start the execution using the input you provided. A new execution will start, and a new page will appear, showing your open execution.

6. In the *Execution Details* section, click the *Info* tab to see the *Execution Status* and when the execution started and closed.

7. Once it completes, click the *Output* tab to see the results of your execution:



	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

Congratulations, you have implemented and executed your first state machine that uses a Lambda function!

1.4 Tutorial: An Activity State Machine

Although you can use AWS Lambda to perform tasks, as shown in *Tutorial: A Lambda State Machine*, you can also run task code on your own machines. This topic will provide you with a quick introduction to writing an activity-based state machine with Java and AWS Step Functions.

Contents

- *Prerequisites*
- *Step 1: Create a New Activity*
- *Step 2: Create a State Machine*
- *Step 3: Implement a Worker*
- *Step 4: Start the Execution*
- *Step 5: Run the Worker*
- *Step 6: Stop the Worker*

1.4.1 Prerequisites

To complete this tutorial, you will need:

- An Amazon Web Services account. If you haven't yet signed up for AWS, go to http://aws.amazon.com/ and click *Sign In to the Console*.
- The AWS SDK for Java. The example activity shown 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 Java Developer Guide.

1.4.2 Step 1: Create a New Activity

In this step, you let Step Functions know about the *activity* whose *worker* (implementation) you are about to write. Step Functions responds with an ARN that establishes an identity for the *activity*. You will utilize this identity when creating your state machine and implementing your worker in order to coordinate information passed between them.

1. Sign in to the Step Functions console.
2. Click *Tasks* in the left pane.
3. Click the *Create new activity* button to create a new activity. Give your activity a name, such as `get-greeting`.
4. Click the *Create Activity* button to complete creating the new activity.

   Your activity is represented by an Amazon Resource Name (ARN) that ends with the name that you gave to the task, for example:

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

5. Make a note of the ARN—you will need to refer to your activity's ARN in the state machine definition and in the worker code.

   Next, you will create the state machine.

1.4.3 Step 2: Create a State Machine

In this step you create a state machine that will determine when your *activity* is invoked, and so your *worker* should perform its primary work, collect the results and send them back.

1. In the Step Functions console click *Dashboard* in the left pane.
2. Click the *Create a State Machine* button.
3. In the box below *Give a name to your state machine*, type a name such as `HelloActivityStateMachine`.
4. In the Code pane, add the following code to the state machine:

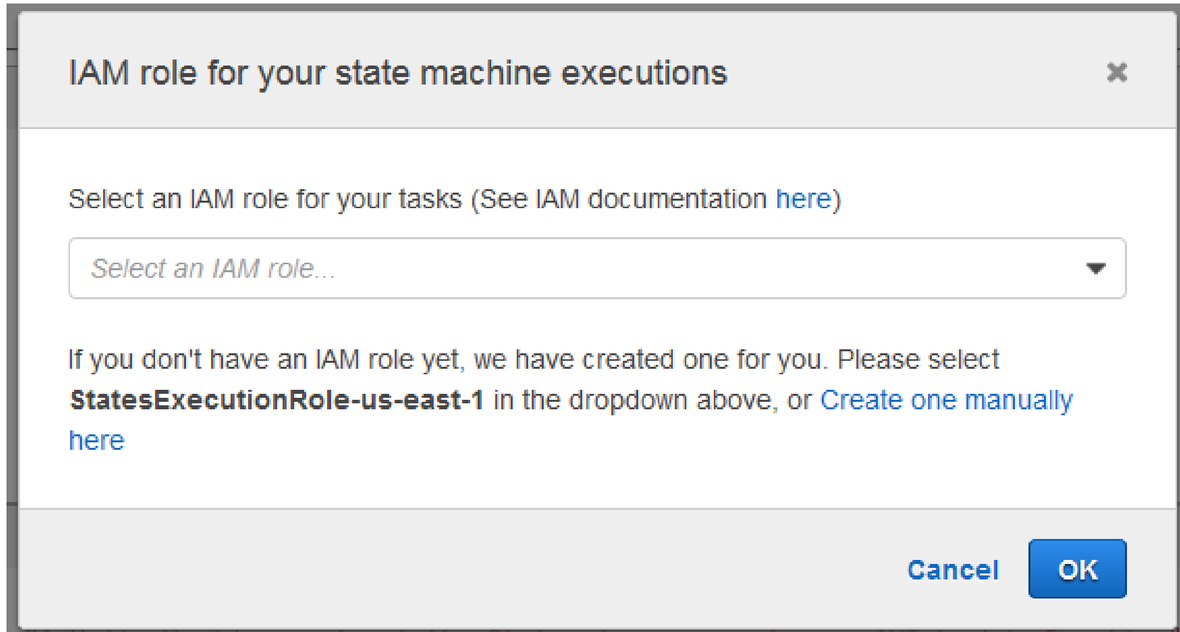
```
{
  "Comment": "An example using a Task state.",
  "StartAt": "getGreeting",
  "Version": "1.0",
  "TimeoutSeconds": 300,
  "States":
  {
    "getGreeting": {
      "Type": "Task",
      "Resource": "arn:aws:states:eu_central-1:123456789012:activity:get-
↪greeting",
```

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

This is a description of your state machine using the Amazon States Language. It defines a single activity named *get-greeting*. Your activity's ARN is specified in the *Resource* field.

Note: For more information about the state machine description language, see *Amazon States Language Overview*.

5. In the code you just entered, replace the value of the `Resource` field with the ARN of the activity that you created in *Step 1: Create a New Activity*.
6. Click the *Create State Machine* button. A pop-up window will appear, titled *IAM role for your state machine executions*.



7. In the *Select an IAM role for your tasks* drop-down list, select the role (`StatesExecutionRole-REGION`) that was automatically created for you by Step Functions, and then click *OK* to finish creating your state machine.

Note: If you delete the IAM role that has been automatically created for you, there is no way for Step Functions to re-create it for you at a later time. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), there is no way for Step Functions to restore its original settings at a later time. For more information on creating an IAM role by hand, see *Creating IAM Roles for Use with AWS Step Functions*.

When state machine creation is complete, you will be presented with your state machine's detail page. If you click the *Dashboard* link, you will see your state machine's ARN in the list of state machines that you have created. Clicking on the state machine will return you to its detail page.

1.4.4 Step 3: Implement a Worker

You now have an activity ARN and you have created a new state machine. Next, you will create a program known as an *worker*, which is responsible for: polling Step Functions for activities (using the "GetActivityTask" API call), performing the work of the activity (with code you write, for example, the `getGreeting()` method in the code below) and sending the results back (using the "SendTask*" API calls).

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

```
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.stepfunctions.AWSStepFunctionsClient;
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;


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();
        AWSStepFunctionsClient client = new AWSStepFunctionsClient(
                new BasicAWSCredentials(ACCESS_KEY, SECRET_KEY),
                new ClientConfiguration());

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

            if (getActivityTaskResult != null) {
                try {
                    JsonNode json = Jackson.
↪jsonNodeOf(getActivityTaskResult.getInput());
                    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);
            }
        }
    }


}
```

3. In the code, replace `ACCESS_KEY` and `SECRET_KEY` in the parameter list of `BasicAWSCredentials()` with references to your corresponding user security credentials for the account you are using.
4. In the code, replace the value of the parameter to `GetActivityTaskRequest().withActivityArn()` with the ARN of the activity that you created in *Step 1: Create a New Activity*.

1.4.5 Step 4: Start the Execution

In this step, you start an execution of the state machine so your *worker* can poll Step Functions, perform its work (using input you provide) and, when it is done, send back its results.

1. In the Step Functions console, click *Dashboard* in the left pane, and then click the state machine that you created in *Step 2: Create a State Machine* (ActivityStateMachine.) This will open its detail page.
2. On the state machine's detail page, click *New execution*. The *New execution* pane will open.

   Tip: To help identify your execution, you can enter an id for it. If you don't, a unique id will be generated for you.

3. Provide the following input to the state machine:

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

4. Click the *Start Execution* button to start the execution using the input that you provided. Your state machine will start executing, and a new page will appear, showing your open execution.

1.4.6 Step 5: Run the Worker

You must run your worker in order to have it poll your state machine for activities.

1. Open a terminal (command-line) window and navigate to the directory in which you created `GreeterActivities.java`.
2. Compile it:

```
$ javac GreeterActivities.java
```

3. Run it:

```
$ java GreeterActivities
```

4. Return to the Step Functions console where the *Execution Details* page should be displayed.
5. Once the execution completes, click the *Output* tab to see the results of your execution.

1.4.7 Step 6: Stop the Worker

1. Stop your worker that you started in *Step 5: Run the Worker*.

Note: If you don't stop your worker, it will continue to run and poll for activities. Because the execution is now stopped, your worker has no source of tasks and will generate a *SocketTimeoutException* each time it polls.

Congratulations, you have successfully created and run an activity-based state machine! You can provide any activity code you want, and run it as a task in your state machine.

1.5 Tutorial: Handle Errors Using Retry

This tutorial will show you how to handle error conditions with a state machine that uses a *Retry* field. To generate the error conditions, you will use an AWS Lambda function that automatically generates an error.

Contents

- *Prerequisites*
- *Step 1: Create an IAM Role for Lambda*
- *Step 2: Create a Lambda Function That Fails*
- *Step 3: Create a State Machine with a Retry Field*
- *Step 4: Start an Execution*

1.5.1 Prerequisites

To complete this tutorial, you will need:

- An AWS account. If you haven't yet signed up for AWS, go to http://aws.amazon.com/ and click *Sign In to the Console*.

1.5.2 Step 1: Create an IAM Role for Lambda

Note: If you followed the steps in *Tutorial: A Lambda State Machine* you have already performed this step and may skip to the next.

Both Lambda and AWS Step Functions are capable of executing code and accessing AWS resources (such as data stored in Amazon S3 buckets), so to maintain security, you must grant Lambda and Step Functions access to those resources. This is done automatically for Step Functions—an IAM role is created when you create a state machine (though you can define and use your own, if you wish.)

Lambda requires you to assign an IAM role when you create an AWS Lambda function in the same way Step Functions required you to assign an IAM role when you create a state machine. So you will create one for Lambda now.

To create a role for use with Lambda

1. Open the IAM console.
2. Choose *Roles* in the left pane, then click the *Create New Role* button to begin the role-creation process.
3. On *Set Role Name*, type a name for your role, such as `lambda-role`, and click *Next Step*.
4. On *Select Role Type*, choose *AWS Lambda* from the list.

   Note: This will cause you to skip over the *Establish Trust* step. The role is automatically provided with a trust relationship that allows Lambda to use the role.

   ---

5. On *Attach Policy*, don't attach any policy. Click *Next Step*.
6. On *Review*, you get a final chance to change the name and policy for your role. Click *Create Role*.

Your role should now appear in the list of roles in the IAM console.

1.5.3 Step 2: Create a Lambda Function That Fails

1. Open the Lambda console.

   If you have never created a Lambda function before, you will be greeted with a screen with a single button *Get Started Now* that allows you to create your first Lambda function.

2. If this is your first Lambda function, then click the *Get Started Now* button. Otherwise, click the *Create a Lambda function* button. The first screen that appears is the *Select blueprint* screen. You won't need a blueprint because you will type the code for your function by hand.
3. In the left pane, click *Configure function*.
4. On the screen that appears, type your function's *Name* and *Description*, and choose the *Runtime*, using the following data:

Field	Value
Name	`FailFunction`
Description	`Always generate an error.`
Runtime	*Node.js 4.3*

5. Next, enter the code for the Lambda function:

```
exports.handler = function(event, context) {
    context.fail("error");
};
```

The *context* object can be used to return an error from your function using the `fail` method. Here, you return a message consisting of just "error".

6. In the *Lambda function handler and role* section, open the *Role* drop-down list and select *Choose an existing role*.
7. Open the *Existing role* drop-down list and choose the Lambda role that you created earlier (`lambda-role`).

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. Revisit *Step 2: Create a Lambda Function That Fails* to verify or edit the trust relationship for your role.

8. In the *Advanced settings* section, accept the default values. Note that you can set a *Timeout* value in seconds. This timeout value can be set to restrict how long your function can take to execute before it is considered a failure.

   The default value of `3` seconds is fine for your function.
9. Click *Next* to review your function, and then click *Create function* to finish creating your Lambda function.

   Once your Lambda function has been created, its Amazon Resource Name (ARN) will be displayed in the top-right corner of the detail screen for your function. It will look similar to this:

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

10. Copy your function's ARN to use when creating your state machine.

Tip: For more information about AWS Lambda, see the Lambda Developer Guide.

Optional: Test Your Lambda Function

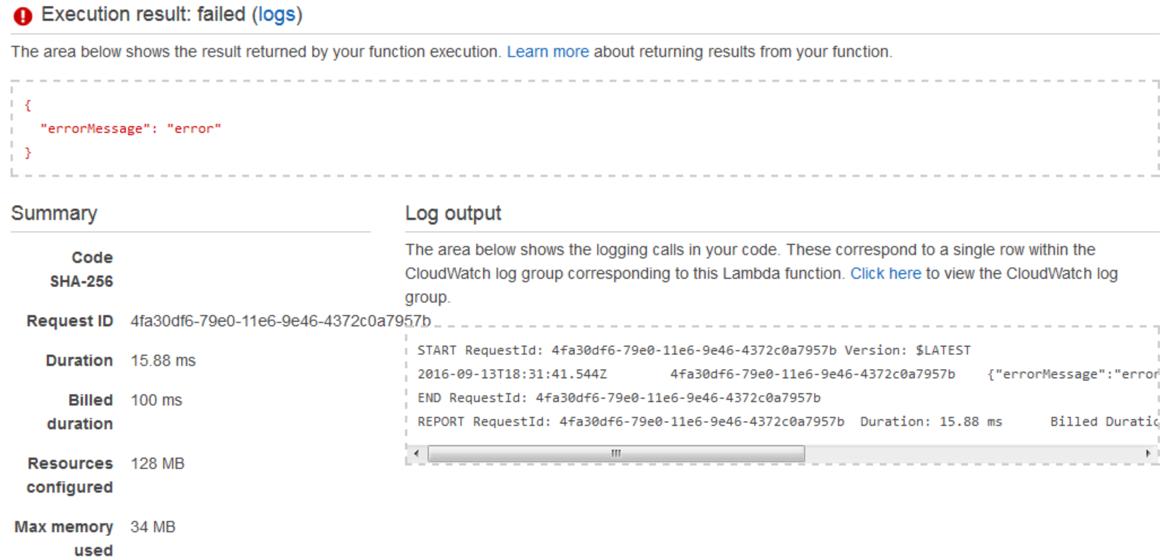
If you want to, you can test your Lambda function now to see it in operation—a good idea if you're developing a Lambda function to use in a real state machine!

To test your Lambda function

1. At the top of your Lambda function's detail screen, click the *Test* button.

   The *Input sample event* screen will appear, pre-loaded with sample data which, in this case, won't be used or changed.

2. Click *Save and test* to test your Lambda function. The results of the test appear at the bottom of the screen, so you may need to scroll the window to see them. (You should see an error message, which is the expected behavior.)



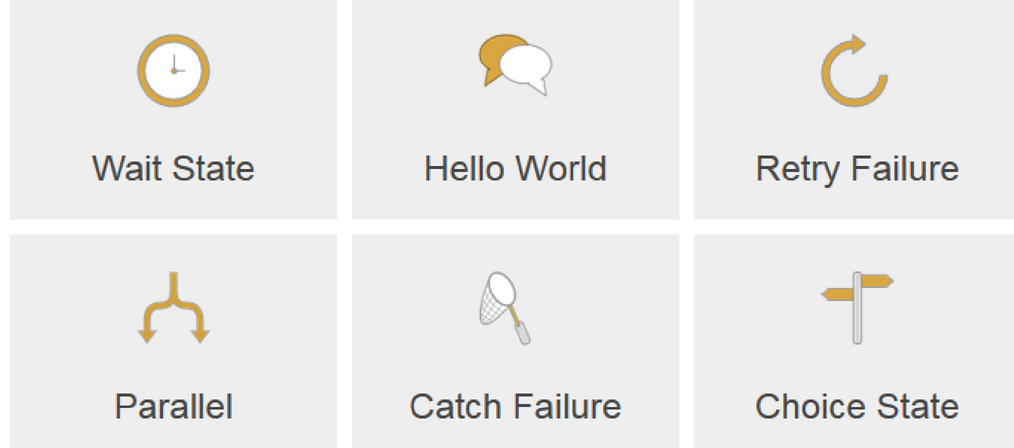
1.5.4 Step 3: Create a State Machine with a Retry Field

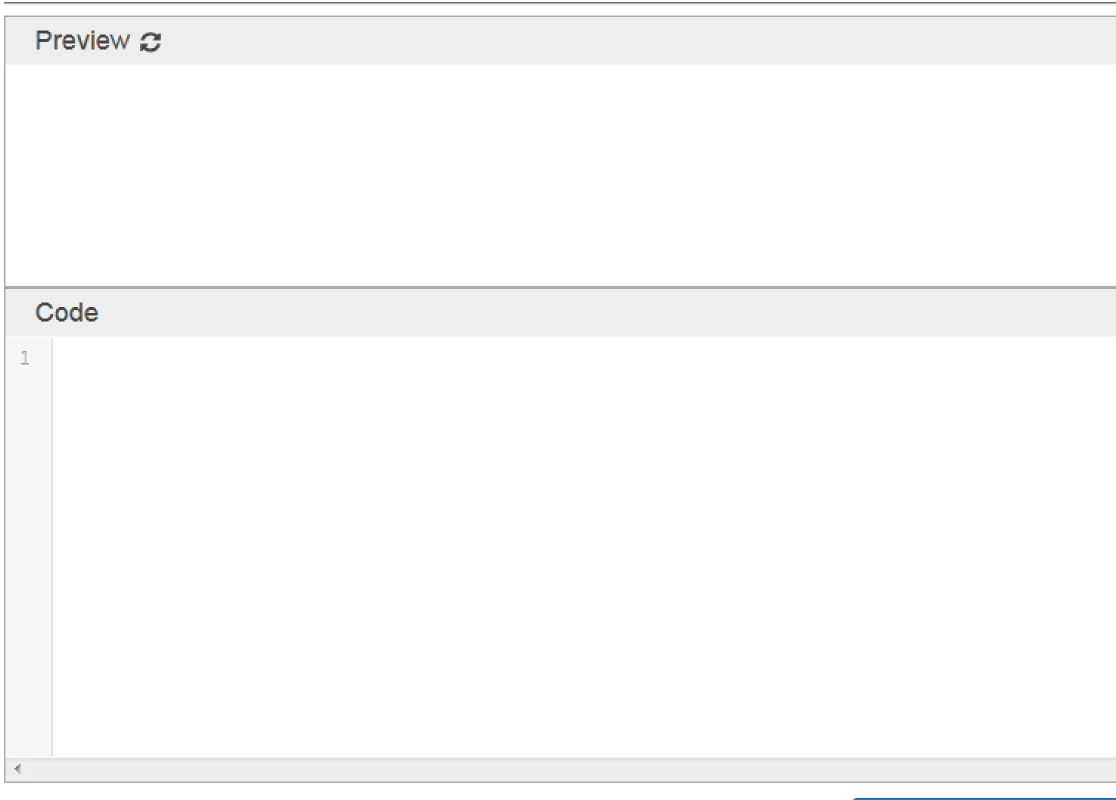
Next, using the Step Functions console, you will create a state machine that contains a *Task* state with a *Retry* field. You will put a reference to your Lambda function in the *Task* state, causing the Lambda function to be invoked, and fail, during execution. The function will be retried twice with an exponential backoff between retries.

To create the state machine

1. Open the Step Functions console.

2. Click *Get Started*. This will display the *Create State Machine* screen. (If you have already created a state machine using Step Functions, click *Dashboard* in the left pane and then click *Create a State Machine*.)


Dashboard
Create State Machine
Give a name to your state machine
Enter your name here
You can now create your own state machine with your own code or choose a blueprint below
Wait State
Hello World
Retry Failure
Parallel
Catch Failure
Choice State



3. In the box below *Give a name to your State Machine*, type a name, such as "RetryStateMachine".

Note: State machine names must be *unique* for your account and region.

4. Click the *Retry Failure* blueprint. The available blueprints enable you to begin with an example state machine template.

   The *Code* pane will be populated with the Amazon States Language description of the state machine.

   The state machine blueprint consists of a single *Task* state, named *HelloWorld*. The *Resource* parameter will be used to reference the ARN of the Lambda function, *FailFunction*, that you created earlier.

5. In the code, replace the value of the `Resource` field (`arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME`) with the ARN of the Lambda function that you saved previously. When you hover over the current value, you should see a pop-up window that allows you to select and auto-complete the new value from a list of Lambda ARN's. Delete the old value, then click the appropriate ARN to fill in the new.

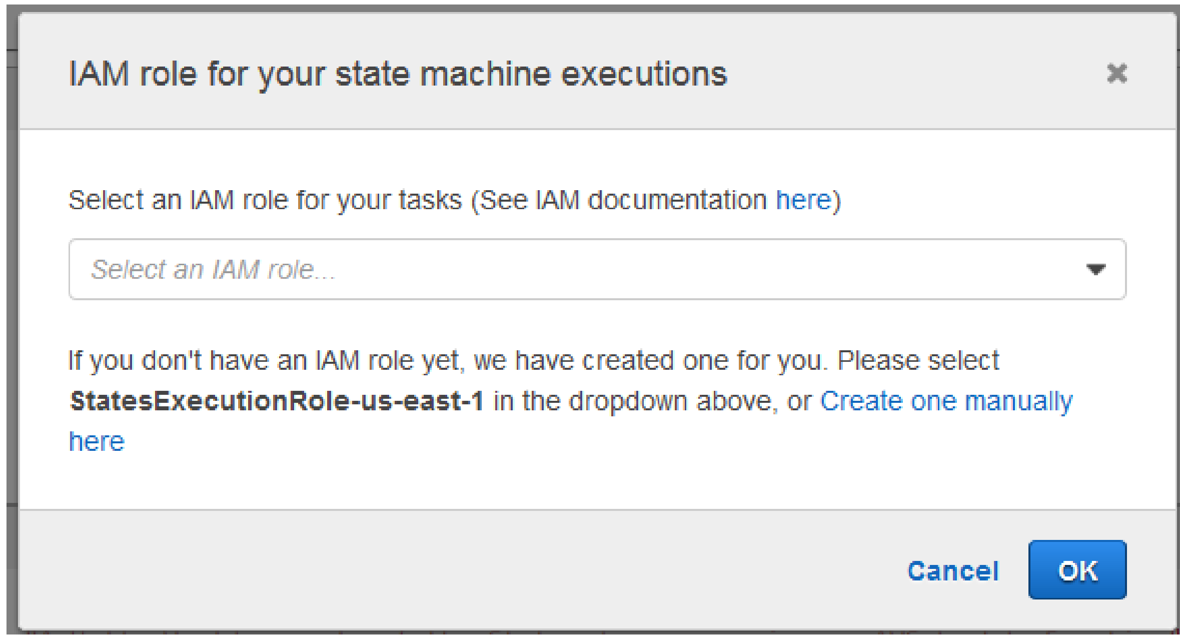
6. Check that the resulting code looks similar to this:

```
{
    "Comment": "A Retry 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
                },
                {
                    "ErrorEquals": ["States.TaskFailed"],
                    "IntervalSeconds": 30,
                    "MaxAttempts": 2,
                    "BackoffRate": 2.0
                },
                    "ErrorEquals": ["States.ALL"],
                    "IntervalSeconds": 5,
                    "MaxAttempts": 5,
                    "BackoffRate": 2.0
                }
            ],
            "End": true
        }
    }
}
```

Note: For more information on the syntax of the `Retry` field, see *Retrying After an Error*.

7. Click the *Create State Machine* button. A pop-up window will appear, titled *IAM role for your state machine executions*.



8. In the *Select an IAM role for your tasks* list, select the role (`StatesExecutionRole-REGION`) that was automatically created for you by Step Functions, and then click *OK* to finish creating your state machine.

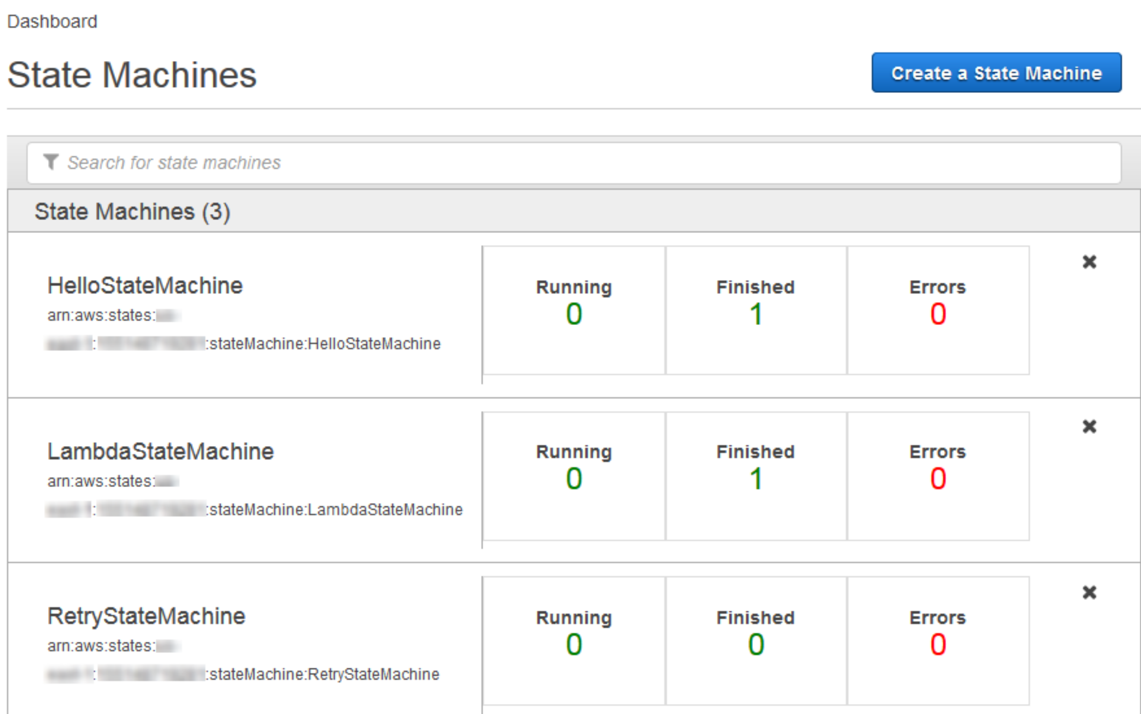
Note: If you delete the IAM role that has been automatically created for you, there is no way for Step Functions to re-create it for you at a later time. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), there is no way for Step Functions to restore its original settings at a later time. For more information on creating an IAM role by hand, see *Creating IAM Roles for Use with AWS Step Functions*.

1.5.5 Step 4: Start an Execution

As soon as you finish creating a state machine, you will see an acknowledgment page. From here, you can start a new execution of your state machine using the *New execution* button. You can also start a new execution at any time from the state machine's detail page, which can be reached from the dashboard.

To start a new execution using the console

1. In the left pane, click *Dashboard*. You will see the state machine you just created.



2. Click the state machine (RetryStateMachine) to see its detail page.
3. From your state machine's detail page, click *New execution*. The *Execution Input* pane will open.

Dashboard 〉 RetryStateMachine 〉 new

RetryStateMachine

New execution

Enter your execution id here

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

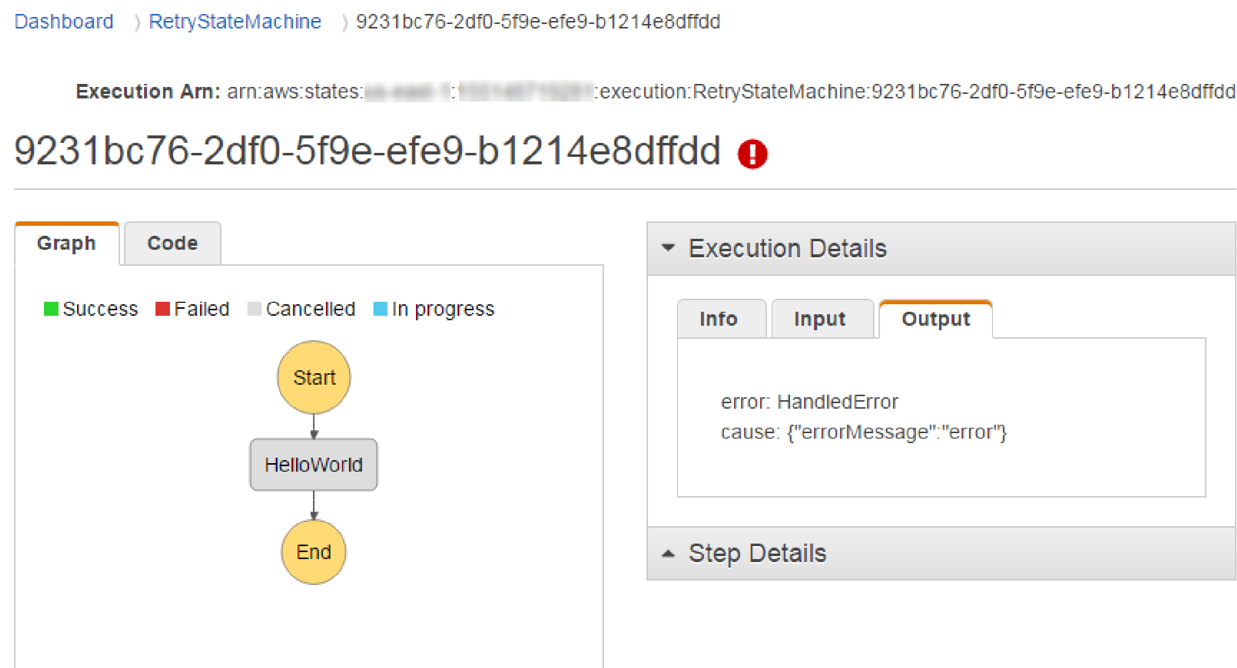
Start Execution

Tip: To help identify your execution, you can optionally enter an id for it. If you don't, a unique id will be generated for you. To set the id, use the box labelled *Enter your execution id here*.

4. Below the *Execution input* pane, click the *Start Execution* button to start the execution. A new execution will start, and a new page will appear, showing your open execution. Wait a few moments for the execution to end.

5. In the *Execution Details* section, click the *Info* tab to see the execution status and when the execution started and closed.

6. Click the *Output* tab to see the output returned by the state machine.



The bottom of the page (*Step Details*) shows that the Lambda function was tried once, and then retried two additional times, before the execution as a whole failed.

ID	Type	Timestamp
1	ExecutionStarted	Oct 27, 2016 12:14:15 PM
2	TaskStateEntered	Oct 27, 2016 12:14:15 PM
3	LambdaFunctionScheduled	Oct 27, 2016 12:14:15 PM
4	LambdaFunctionStarted	Oct 27, 2016 12:14:16 PM
5	LambdaFunctionFailed	Oct 27, 2016 12:14:17 PM
6	LambdaFunctionScheduled	Oct 27, 2016 12:14:18 PM
7	LambdaFunctionStarted	Oct 27, 2016 12:14:19 PM
8	LambdaFunctionFailed	Oct 27, 2016 12:14:19 PM
9	LambdaFunctionScheduled	Oct 27, 2016 12:14:21 PM
10	LambdaFunctionStarted	Oct 27, 2016 12:14:22 PM
11	LambdaFunctionFailed	Oct 27, 2016 12:14:22 PM
12	ExecutionFailed	Oct 27, 2016 12:14:22 PM

Note: If you don't see the ExecutionFailed event, refresh your web page to get a full list of history events.

You have successfully implemented and run your first state machine that uses a *Retry* field!

You can also create state machines that *Retry* on timeouts and 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 Catch*.

1.6 AWS Step Functions Concepts

To understand AWS Step Functions, you will need to be familiar with a number of important concepts. This section will describe, at a high level, how Step Functions operates.

1.6.1 Amazon States Language

Amazon States Language is a JSON-based, structured language used to define your state machine. Here is an example of a state machine specification using the Amazon States Language:

```
{
  "Comment": "An Amazon States Language 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
    }
  }
}
```

A state machine consists of a collection of *states*, that can do work (*Task* states), determine which states to transition to next (*Choice* states), stop an execution with an error (*Fail* states), and so on.

For an overview of Amazon States Language, see the *Amazon States Language Overview*.

For information on how to use Amazon States Language to create a state machine on the AWS Step Functions console, see *Create State Machine*.

1.6.2 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.

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

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

For example, here is a sample 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. (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*.

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* screen on the Step Functions console.

1.6.3 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*, which is a completely cloud-based task that runs on the Lambda service. Lambda functions can be written in JavaScript (which you can write using the AWS Management Console or upload to Lambda), or in Java or Python (uploaded 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* in the *Amazon States Language Overview*.

To see a list of your tasks, you can access the *Tasks* screen in the Step Functions console.

1.6.4 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 *exactly* match the name of a state specified in the state machine description (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 Overview* 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 (i.e. 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 (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.

1.6.5 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.

Contents

- *Data Format*
- *State Machine Input/Output*

• *State Input/Output*

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 *Filters*.

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* (i.e. 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*, have *InputPath*, *ResultPath* and *OutputPath* fields. The values of these fields are *paths*.

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 (i.e. `$`.) 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 added to the state's input (and, so its output.) 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]
}
```

But now, the state output data will be:

```
{
  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.

1.6.6 Executions

When a state machine runs and performs its tasks, we refer to this as a state machine "execution". Each AWS Step Functions state machine can be have multiple, simultaneous executions which can be initiated from the Step Functions console or by a program you write using the API, AWS SDKs and so on (see *Development Options*.) An execution is given input data in JSON text format and produces output in JSON text format, which you can also access using an application you write or from the Step Functions console.

For more information on initiating an execution from the Step Functions console, see *State Machine Details*.

1.6.7 Error Handling

Any state can encounter runtime errors. Errors can arise because of state machine definition issues (e.g. no matching rule in a *Choice* state), task failures (e.g. 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 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. [Required]

IntervalSeconds An integer that represents the number of seconds before the first retry attempt (default 1). [Optional]

MaxAttempts 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. [Optional]

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

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-states-1:123456789012:task:X",
  "Next": "Y",
  "Retry": [
    {
      "ErrorEquals": [ "ErrorA", "ErrorB" ],
      "IntervalSeconds": 1,
      "BackoffRate": 2,
      "MaxRetries": 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 ("MaxRetries") 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.

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

`Next` A string which must exactly match one of the state machine's state names. [Required]

`ResultPath` A *path* which determines what is sent as input to the state specified by the *Next* field. [Optional]

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.

Examples Using Retry and Catch

The state machines defined next 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.

Here is the definition of a Lambda function that will always fail, returning the message "error". In the state machine examples that follow, it is assumed that it is named "FailFunction":

```
exports.handler = function(event, context) {
    context.fail("error");
};
```

Here is the definition of a Lambda function that sleeps for 10 seconds. In the state machine examples that follow, it is assumed that it 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 (the default) to 10 seconds.

```
exports.handler = function(event, context) {
    setTimeout(function(){
            context.succeed(1);
    }, 10000);
};
```

Handle a Failure Using Retry

This state machine uses a *Retry* field to retry a function that fails and throws the errorName "HandledError". The function will be 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
    }
  }
}
```

Here is a variant that uses the predefined ErrorCode "States.TaskFailed", which will match any error that a Lambda function throws:

```
{
  "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
    }
  }
}
```

Handle a Failure Using Catch

This example uses a *Catch* field. When an error is thrown by the Lambda function, it will be caught and the state machine will transition 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
    }
  }
}
```

Here is a variant that uses the predefined ErrorCode "States.TaskFailed", which will match any error that a Lambda function throws:

```
{
  "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
    }
  }
}
```

Handle a Timeout With Retry

This state machine uses a *Retry* field to retry a function that times out. The function will be 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
    }
  }
}
```

Handle a Timeout With Catch

This example uses a *Catch* field. When a timeout occurs, the state machine will transition 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
    }
  }
}
```

1.6.8 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 click the *Create New Role* button to begin the role creation process.
3. On *Set Role Name*, type a name for your role, such as `states-lambda-role`, and click *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*, click to select the *AWSLambdaRole* policy, then click *Next Step*. (If you are creating a state machine for a different purpose, please choose the appropriate policy here.)
6. On *Review*, click *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, click the name of the role that you just created (e.g. `states-lambda-role`) from the list. This will open the role's detail page.
8. Click the *Trust Relationships* tab and then click *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. Click *Update Trust Policy*.

1.7 Development Options

You have a number of options for implementing your state machine solutions with AWS Step Functions.

Contents

- *AWS Step Functions console*
- *AWS SDKs*
- *HTTPS Service API*
- *Development Environments*
- *Endpoints*
- *AWS CLI*

1.7.1 AWS Step Functions console

You can define a state machine completely within the AWS Step Functions console. By using Lambda to supply code for your tasks and by using the Step Functions console to write your state machine using Amazon States Language, even complex state machines can be written completely on the cloud, without the use of a local development environment.

Tutorial: A Lambda State Machine uses this technique to create a simple state machine, run it, and view its results.

1.7.2 AWS SDKs

AWS 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 in the programming language of your choice.

You can develop state machines, activities, or state machine starters using the API exposed by these libraries. Additionally, you can access visibility operations through these libraries so you can develop your own Step Functions monitoring and reporting tools.

If you plan on using Step Functions with other AWS services, it may be helpful to refer to the reference documentation for the current AWS SDKs. To download or discover more about the AWS SDKs, go to *Tools for Amazon Web Services*.

Note: Step Functions only supports an HTTPS endpoint.

1.7.3 HTTPS Service API

AWS Step Functions provides service operations that are accessible through HTTPS requests. You can use these operations to communicate directly with Step Functions, and you can use them 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 by using the service API. You can also access visibility operations through the API to develop your own monitoring and reporting tools. For detailed information about API operations, see the AWS Step Functions API.

1.7.4 Development Environments

You must set up a development environment appropriate to the programming language that you will use. For example, if you intend to develop for AWS 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 the Eclipse IDE for Java development, you might consider also installing the AWS Toolkit for Eclipse. The Toolkit is an Eclipse plug-in that adds features that are helpful for AWS development.

If your programming language requires a run-time environment, you must set up that environment on each computer on which these processes run.

1.7.5 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; any state machines and activities you have created in one region do not share any data or attributes with those in another. In other words, when you create a state machine or activity, it exists *only within the region you created it in*. For example, you could register a state machine named `STATES-Flows-1` in two different regions, but they will share no data or attributes with each other—each acts as a completely independent state machine.

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

1.7.6 AWS CLI

Many of the features of AWS Step Functions can be accessed from the AWS CLI. The AWS CLI provides an alternative to using the Step Functions console or, in some cases, to programming with the AWS Step Functions API. For example, you can use the AWS CLI to create a new state machine and you can list your state machines.

The AWS Step Functions commands in AWS CLI provide the ability to start and manage executions, poll for activities, record task heartbeats, and more! For a complete list of AWS Step Functions commands, with descriptions of the available arguments and examples showing their use, see the AWS Command Line Reference.

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

1.8 Implementing 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.

- *Creating an Activity*
- *Writing a Worker*

1.8.1 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.

1.8.2 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.

1.9 Console Reference

This section describes the AWS Step Functions interface provided by the Step Functions console. The Step Functions console provides an easy-to-use interface that can be used to generate Step Functions API requests on your behalf.

1.9.1 Dashboard

The AWS Step Functions dashboard on the Step Functions console is the default screen that you encounter if you have already created a state machine. If you haven't created a state machine yet, the first screen you will encounter is *Create State Machine*.

It consists of two parts—the *Create a State Machine* button, and a list of state machines that you have created.

Dashboard

State Machines

Create a State Machine

Search for state machines

State Machines (2)

State Machine	Running	Finished	Errors
HelloStateMachine	0	1	0
LambdaStateMachine	0	0	0

- *Create a State Machine Button*
- *State Machine List*

Create a State Machine Button

Clicking the *Create State Machine* button starts the *Create State Machine* process to create a new state machine.

State Machine List

Shows a list of state machines that you have created along with a number of statistics (the number of executions that are *Running*, *Finished*, or that resulted in *Errors*) for each state machine. You can click the state machine's ARN to show that state machine's *detail page*.

Delete a State Machine

In the State Machine list, click the X to the right of a state machine item to *delete* it.

Note: Once you delete a state machine, its name can't be used for any new state machine types.

1.9.2 Create State Machine

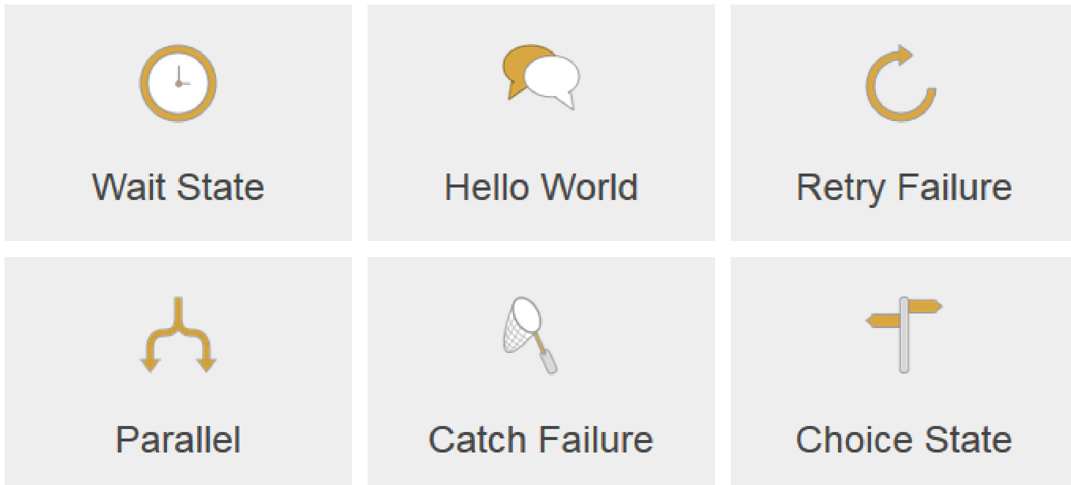
The *Create State Machine* screen allows you to create a new state machine. It is accessed from the *Dashboard* by clicking the *Create a State Machine* button.

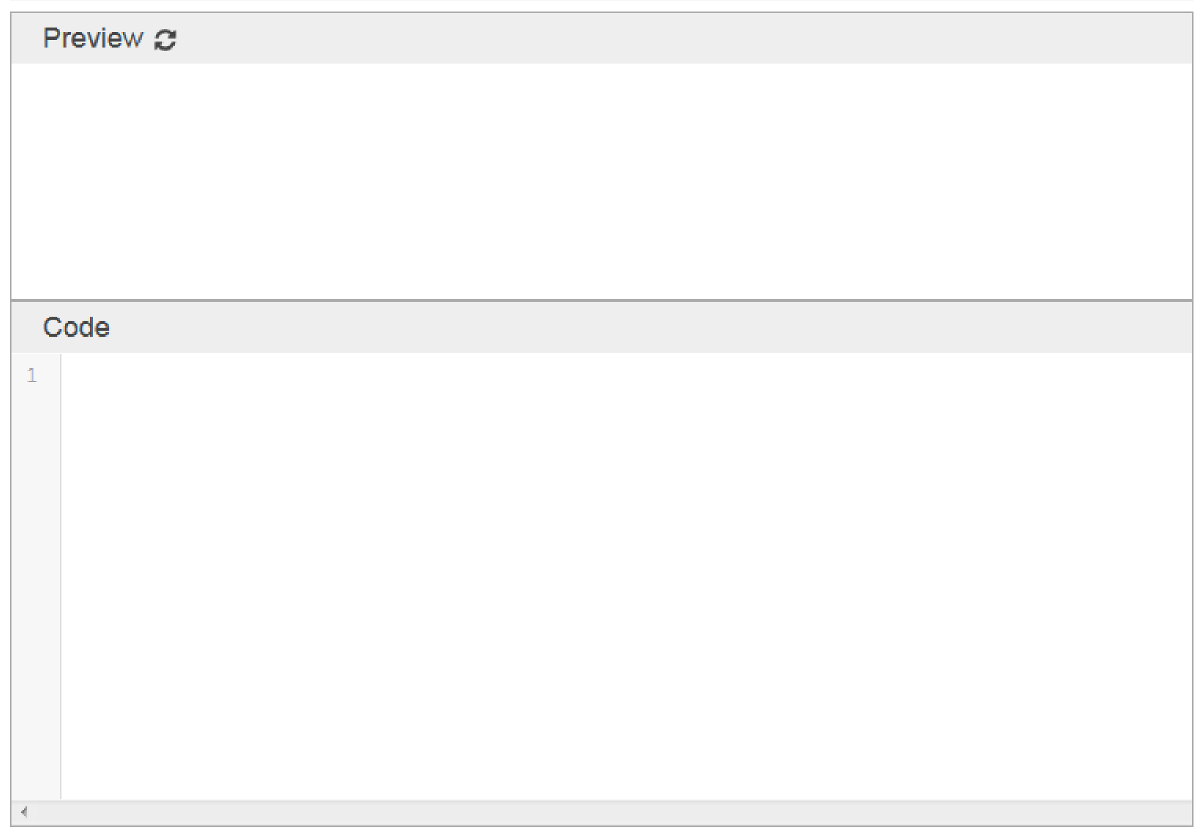
Dashboard 〉 Create State Machine

Give a name to your state machine

Enter your name here

You can now create your own state machine with your own code or choose a blueprint below





- *State Machine Name*
- *Blueprints*
- *Preview Pane*
- *Code Pane*

State Machine Name

Use the box below *Give a name to your State Machine* to specify the state machine name. The name must be unique for your account and region, and it must meet Amazon Resource Name (ARN) constraints.

Resource names can be from 1 – 80 characters in length, and must:

- Not contain whitespace.
- Not contain bracket characters `< > { } [ ]`
- Not contain any of `:` (colon), `;` (semi-colon), `,` (comma), / (slash), \ (backslash), `|` (vertical bar), `^` (caret), `~` (tilde), `$` (dollar sign), `#` (hash mark), `%` (percent sign), `&` (ampersand), `` ` `` (left-quote)

- Not contain any control characters (\u0000 – \u001f or \u007f – \u009f).
- Not contain the literal string “arn”

Blueprints

You can choose one of the state machine blueprints (or templates) to automatically fill the *Code* pane. Each of the templates is runnable, and can also be used as a basis for state machine code of your own.

To use a blueprint, click one of the following:

- *Wait State* – a state machine that demonstrates the different ways to inject a wait into a running state machine: for a number of seconds or until an absolute time (timestamp), specified either directly in the *Wait* state’s definition or from 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 many different tasks depending on the type of failure.
- *Choice State* – a state machine that makes a choice, running one of a set of *Task* states or a *Fail* state after the initial state is complete.

Caution: Choosing any of the options will *overwrite* the contents of the *Code* pane. *If you have hand-typed any code in the pane, it will be lost when you choose another blueprint.*

Preview Pane

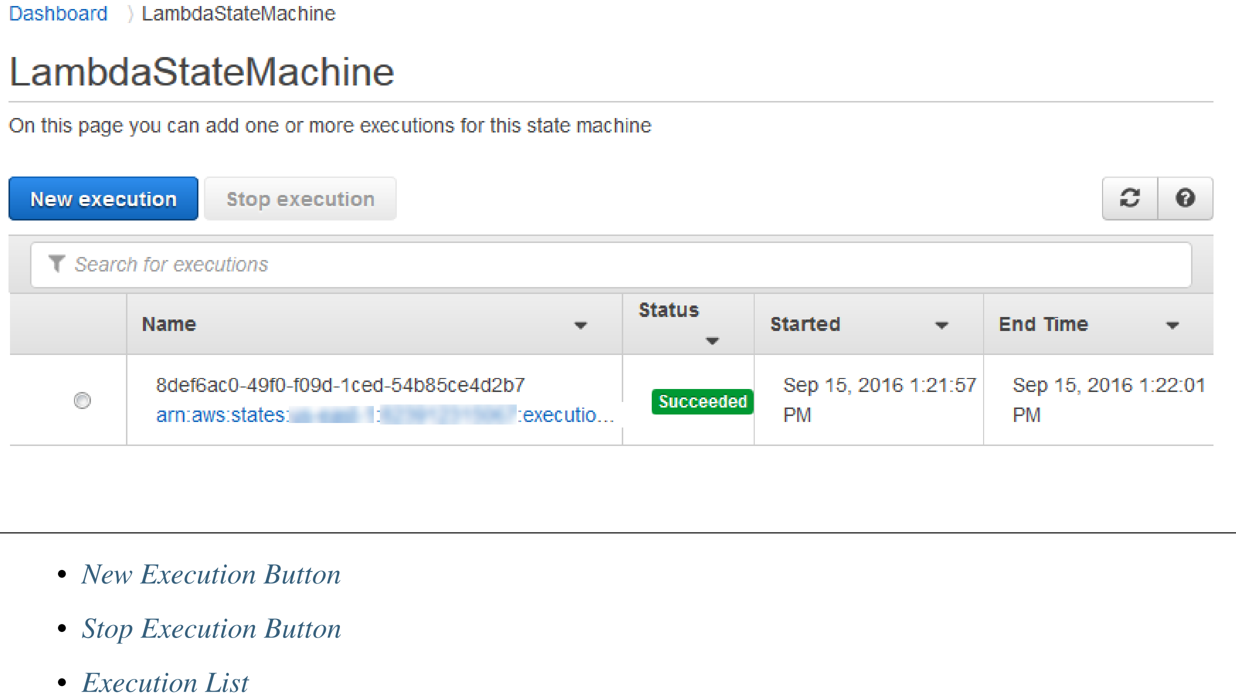
The *Preview* pane displays a graphical view of your state machine. Use this to verify that the state machine description in the *Code* pane accurately represents what you had in mind. If the preview graph does not appear, click the refresh icon next to *Preview*.

Code Pane

The *Code* pane allows you to create or edit a state machine description using JavaScript Object Notation (JSON). Any changes that you make in the *Code* pane will be reflected in the *Preview pane*.

1.9.3 State Machine Details

By clicking a state machine’s ARN on the *Dashboard*, you bring up its detail page, where you can view its executions, start new executions and stop running executions.



- *New Execution Button*
- *Stop Execution Button*
- *Execution List*

New Execution Button

Clicking the *New execution* button begins the process of starting a new execution of the state machine. A new pane will open, allowing you to provide the execution with a name (optional) and with input data in the form of JSON text.

Field	Description
Give a name to this execution	Enter the id of this execution. Providing an id is optional; if you don't provide one, a unique id will be automatically generated for you.
JSON input area	Enter input data in JSON text format. The input data should include field names (keys) that are used in your tasks to access the field data.

To accept the values that you entered in the *Execution's input* pane and start the execution, click the *Start Execution* button that appears below the pane.

Stop Execution Button

If you have a running execution selected in the execution list, click the *Stop execution* button to stop it prematurely.

Execution List

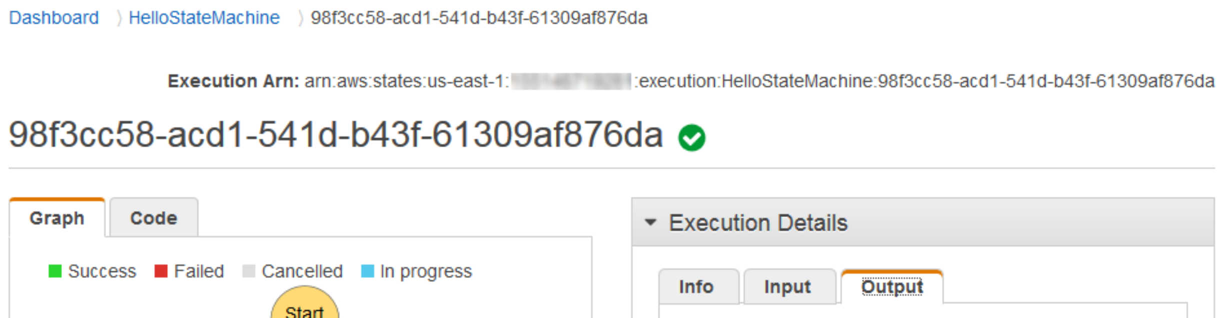
Below the *Stop execution* and *New execution* buttons is a list of executions. Each execution is listed with its *Name* (entered or generated when you click the *Start execution* button), *Status* value, indicating if the execution completed successfully or not, and the *Started* and *End Time* of the execution.

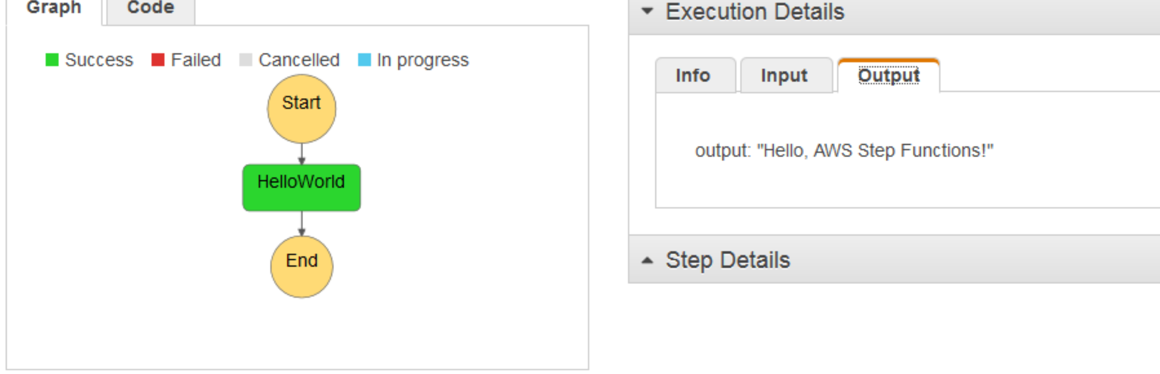
Clicking the execution's ID in the list brings you to the *Execution Details* page for that execution.

1.9.4 Execution Details

By clicking an execution ID on the *State Machine Details* screen, you will be presented with the execution's detail page.

The page shows a number of tabbed windows that provide different types of detail about the execution.





ID	Type	Timestamp
1	ExecutionStarted	Nov 17, 2016 4:37:00 PM
2	PassStateEntered	Nov 17, 2016 4:37:00 PM
3	PassStateExited	Nov 17, 2016 4:37:00 PM
4	ExecutionSucceeded	Nov 17, 2016 4:37:00 PM

- *Graph Tab*
- *Code Tab*
- *Info Tab*
- *Input Tab*
- *Output Tab*
- *Step Details Tab*

Graph Tab

Shows a graphical representation of the state machine that was executed.

Code Tab

Shows the code that defines the state machine, using Amazon States Language notation. For more information, see *Amazon States Language Overview*.

Info Tab

Lists general information about the execution:

Field	Description
Execution Status	The current status of this execution.
State Machine Arn	The Amazon Resource Name (ARN) of the state machine.
Execution ID	When displayed, the execution ID.
Started	The timestamp indicating when the execution started.
Closed	The timestamp indicating when the execution was closed.

Input Tab

Shows the input data, in JSON text format, that was provided when the execution was started.

Output Tab

Displays any output data generated by the execution.

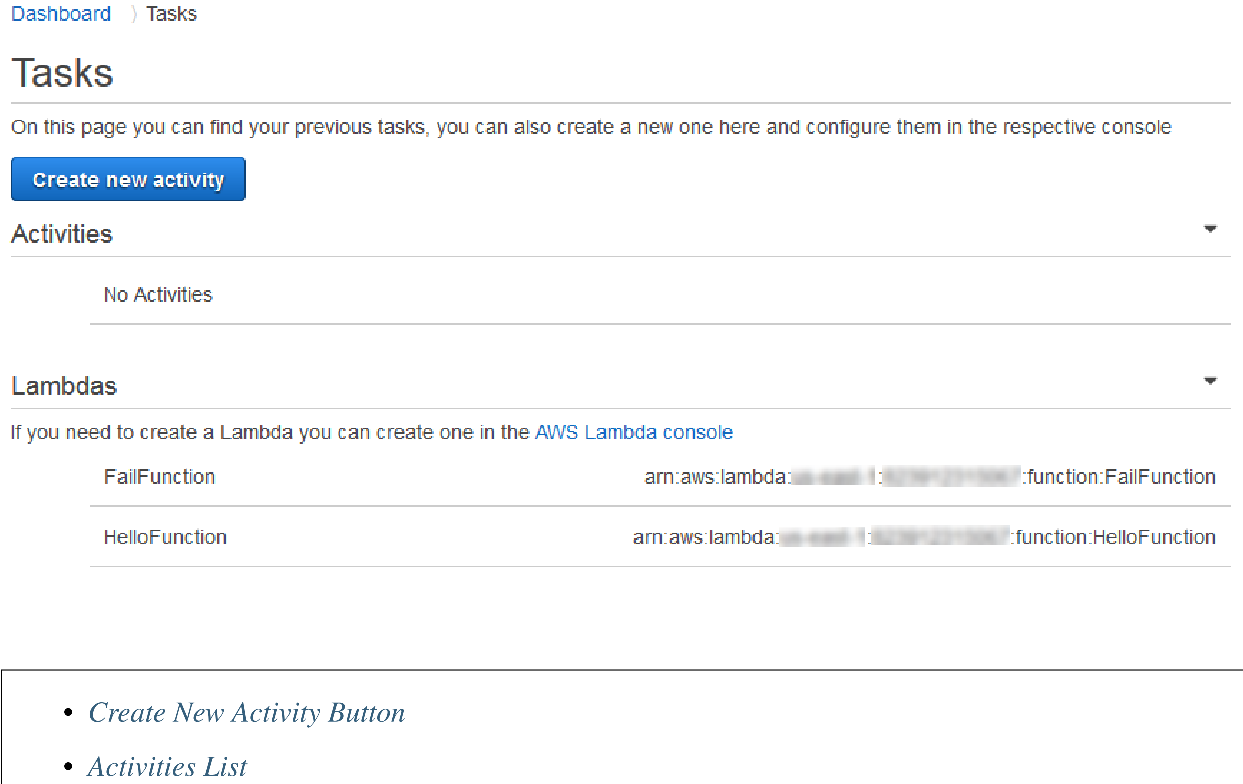
Step Details Tab

Lists the events (steps) that occurred over the course of this execution. The list items contain the following fields:

Field	Description
(not labeled)	Click the down/up arrow to expand/collapse details about the event which will be shown in the “Type” column.
ID	The event ID, which corresponds with the order in which the events occurred in the execution history.
Type	The event type, along with additional information when expanded (including, depending on the event type, the state name, input, output, resource ARN, etc.)
Timestamp	The time at which the event occurred.

1.9.5 Tasks

The *Tasks* menu item on the left pane of the AWS Step Functions console displays a list of any existing created tasks and provides you with the ability to create new tasks.



- *Create New Activity Button*
- *Activities List*
- *Lambdas List*

Create New Activity Button

Click the *Create new activity* button to start the process of creating a new activity. A *New Activity* pane is opened that allows you to set the name of your new activity, which must be unique for your account and region.

Once you enter a name for the activity, click the *Create Activity* button to the right of the *New Activity* pane. Your activity will be created and displayed in the created activity list.

Activities List

This part contains a list of the activities that you have created, each identified by a name and Amazon Resource Name (ARN).

Lambdas List

The bottom part of the page contains a list of the Lambda functions that you have created, each identified by a name and Amazon Resource Name (ARN). There is also a link to the Lambda console where you can create a new Lambda function.

1.10 Amazon States Language Overview

This section describes the syntax and some features of Amazon States Language, which is used to define state machines for AWS Step Functions.

1.10.1 State Machine Structure

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

Comment A human-readable description of the state machine. [Optional]

StartAt A string that must exactly match (case-sensitive) the name of one of the state objects. [Required]

TimeoutSeconds 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*. [Optional]

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

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

The *States* field contains a number of *states*:

```
{
    "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 an␣
↪AWS Lambda Function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
```

```
      "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 (i.e. a state with `"Type": "Succeed"`, `"Type": "Fail"`, or `"End": true`), or a runtime error occurs.

In the previous example, the state machine contains a single state (`"HelloWorld"`) and because it is a *Task* state, AWS Step Functions invokes the Lambda function pointed to by its *Resource* field. Assuming the Lambda function runs successfully, the execution stops successfully as well.

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.

1.10.2 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.

Common State Fields

`Type` The state's type. Can be any of the values listed in *State Types*. [Required]

`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 Holds a human-readable description of the state. [Optional]

InputPath A *Path* that selects a portion of the state's input to be passed to the state's task for processing. If omitted, it has the value `$` which designates the entire input. (See *Filters*). [Optional]

OutputPath A *Path* 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. (See *Filters*.) [Optional]

State Types

Task

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

In addition to the *common state fields*, *Task* states have the following fields:

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

ResultPath 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. (See *Paths*) [Optional]

Retry An array of objects, called Retriers, that define a retry policy in case the state encounters runtime errors. See *Retrying After an Error*. [Optional]

Catch 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. See *Fallback States*. [Optional]

TimeoutSeconds 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. [Optional]

HeartbeatSeconds 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. [Optional]

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*.
  - `lambda` for a *Lambda function*.
- *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*.
  - `function` – a *Lambda function*.
- *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*
- *Lambda Functions*

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*.

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 *Implementing Activities*.

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*.

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").

Wait

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

In addition to the *common state fields*, *Wait* states have one of the following fields:

Seconds A time, in seconds, to wait before beginning the state specified in the *Next* field.

Timestamp An absolute time to wait until before beginning the state specified in the *Next* field. This string must conform to the RFC3339 profile of ISO 8601, with the further restrictions that an uppercase "T" must be used to separate the date and time portions, and an uppercase "Z" must be used if no numeric time zone offset is present, e.g. "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* 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* 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* to select it from the input data:

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

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*, *Pass* states allow the following fields:

Result 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). [Optional]

ResultPath 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. (See *Paths*) [Optional]

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
  }
}
```

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.

Here's an 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*. In addition, the *Fail* state allows the following fields:

`Cause` Provides a custom failure string that can be used for operational or diagnostic purposes. [Optional]

`Error` Provides an error name that can be used for error handling (*Retry/Catch*), operational or diagnostic purposes. [Optional]

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"
}
```

Choice

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

In addition to the *common state fields*, *Choice* states introduce these additional fields:

`Choices` An array of *Choice Rules* that determine which state the state machine transitions to next. [Required]

`Default` The name of a state to transition to if none of the transitions in *Choices* is taken. [Optional, but *recommended*]

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 be used to separate the date and time portions, and an uppercase "Z" must be used if a numeric time zone offset is not present, e.g. "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. See the extended example at the outset of this section.

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*, *Parallel* states introduce these additional fields:

`Branches` 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. [Required]

`ResultPath` 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. (See *Paths*) [Optional]

`Retry` An array of objects, called Retriers that define a retry policy in case the state encounters runtime errors. See *Retrying After an Error*. [Optional]

`Catch` 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. See *Fallback States*. [Optional]

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 *Paths*).

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.

1.10.3 Paths

A *path* is a string used to select a member or subtree of an object defined in JSON text, typically used with state input or output data.

When declaring a path, the top level of an object is referred to with the `$` symbol, and its elements are accessed using dot (`.`) notation. Nested elements can be selected using further dots (`$.x.y`) to select sub-levels of the structure.

For example, given this data:

```
{
    "foo": 123,
    "bar": ["a", "b", "c"],
    "car": {
        "cdr": true,
        "tires": 4
    }
}
```

The following table shows what values are obtained by applying various paths to the data:

Path	Value
$.foo	123
$.bar	["a", "b", "c"]
$.car.cdr	true

Paths are frequently used to extract elements from the state's input data for use in the state's fields, for example, to make a *Choice* about the next state to run, or to select a *Wait* time. Paths are also used to refer to parts of a tree in which to put output data.

Reference Paths

A *reference path* is a *path* that must be an unambiguous reference to a single node in a structure. Because of this, a reference path's syntax is limited:

- Object fields can only be accessed using the dot (".") notation.
- The operators @ .. , : ? and [] are not supported.

Paths in InputPath, ResultPath and OutputPath Fields

Paths are used in the *InputPath*, *ResultPath* and *OutputPath* fields of certain states to specify how part of the state's input is used and what is sent as output to the next state.

InputPath

The *InputPath* field selects a portion of the state's input to be passed to the state's task for processing. If omitted, it is considered to have the value of `$` which represents the whole input. If `null`, the input is discarded (not sent to the state's task) so the task receives JSON text representing an empty object `{}`.

Note that a path can yield a selection of values. For example, given:

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

if the path `$.a[0..1]` is applied, the result will be:

```
[ 1, 2 ]
```

ResultPath

Ordinarily, if a state executes a task, the results of that task are sent along as the state's output (which becomes the input to the next task.) If a state does not execute a task, the state's own input is sent, unmodified, as its output. However, when a path is specified in the value of a state's *ResultPath* and *OutputPath* fields, some different scenarios are possible.

The *ResultPath* takes the results of executing the state's task and puts this in the input. After that, the *OutputPath* selects a portion of that input to send as the state's output (more on that later). The *ResultPath* may add the results of executing the state's task to the input, overwrite an existing part, or overwrite (that is, replace) the entire input:

1. If the *ResultPath* matches an item in the state's input, then only that input item is overwritten with the results of executing the state's task, and the entire input, thus modified, is available to become the state's output.
2. If the *ResultPath* does not match an item in the state's input, then an item is added to the input, containing the results of executing the state's task, and the expanded input becomes available to be the state's output.
3. If the *ResultPath* is `$` (the default if it is omitted) this matches the input in its entirety. In this case, the results of executing the state overwrite the input entirely, and this is what is available to be passed along.
4. 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.

OutputPath

1. If the *OutputPath* matches an item in the state's input, then only that input item is selected and it becomes the state's output.
2. If the *OutputPath* does not match an item in the state's input, then an exception is thrown complaining of an invalid path. (See *Errors*)
3. If the *OutputPath* has a value of `$` (the default), this matches the input in its entirety. In this case, the entire input is passed along to the next state. (But see the explanation of *ResultPath* above for the effect it has on the input for those states which allow it.)
4. If the *OutputPath* is null, a JSON text representing an empty object `{}` is sent to the next state.

Here is an example of how *InputPath*, *ResultPath* and *OutputPath* fields work in practice. Let's say the current state has an input of

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

Further, let's say the state has the following *InputPath*, *ResultPath* and *OutputPath* fields:

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

So the state's task will receive just the `"numbers"` object from the input. Now, let's suppose this task returns `{ 7 }`, then the output of this state would be

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

But now, the state output data will be:

```
{
  7
}
```

1.10.4 Errors

Any state can encounter runtime errors. Errors can arise because of state machine definition issues (e.g. no matching rule in a *Choice* state), task failures (e.g. 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 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. [Required]

IntervalSeconds An integer that represents the number of seconds before the first retry attempt (default 1). [Optional]

MaxAttempts 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. [Optional]

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

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,
      "MaxRetries": 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 (“MaxRetries”) 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.

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

Next A string which must exactly match one of the state machine’s state names. [Required]

ResultPath A *path* which determines what is sent as input to the state specified by the *Next* field. [Optional]

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.

1.10.5 Filters

Some states, such as *Task*, have *InputPath*, *ResultPath* and *OutputPath* fields. The values of these fields are *paths*.

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 (i.e. `$`.) 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 added to the state's input (and, so its output.) 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]
}
```

But now, the state output data will be:

```
{
  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.

More information is available in the Amazon States Language specification.

You might also be interested in Statelint, a tool to validate your Amazon States Language code.

1.11 AWS Step Functions Limits

AWS Step Functions places limits on the sizes of certain state machine parameters, such as the number of API calls that can be made during a certain period or the number of state machines that can be defined. These limits are designed to prevent an erroneous state machine from consuming all of the resources of the system, but are not hard limits.

Contents

- *General Account Limits for AWS Step Functions*
- *Limits on Executions*
- *Limits on Task Executions*
- *AWS Step Functions Throttling Limits*

1.11.1 General Account Limits for AWS Step Functions

- **Maximum number of state machines and activities:** 10,000
- **API call limit:** 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.

1.11.2 Limits on Executions

- **Maximum open executions:** 1,000,000
- **Maximum execution time:** 1 year
- **Maximum execution history size:** 25,000 events
- **Execution idle time limit:** 1 year (constrained by execution time limit)
- **Execution history retention time limit:** 90 days

  After this time, the execution history can no longer be retrieved or viewed. There is no further limit to the number of closed executions that are retained by AWS Step Functions.

If your use case requires you to go beyond these limits, you can use features Step Functions provides to continue executions.

Note: You can configure state machine timeouts to cause a timeout event to occur if a particular stage of your state machine takes too long.

1.11.3 Limits on Task Executions

- **Maximum task execution time:** 1 year (constrained by execution time limit)
- **Maximum time AWS Step Functions will keep 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.
- **Maximum input/result data size:** 32,000 characters

  This limit affects activity or execution result data, input data when scheduling activities or executions, and input sent with an execution signal.

Note: You can configure default activity timeouts during activity creation that will cause a timeout event to occur if a particular stage of your activity execution takes too long.

1.11.4 AWS Step Functions Throttling Limits

In addition to the service limits described previously, certain Step Functions API calls are throttled to maintain service bandwidth, using a token bucket scheme.

Throttling limits are per account / region.

Throttling Limits

API name	Bucket size	Refill rate / s
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

1.12 Monitoring AWS Step Functions

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS Step Functions and your AWS solutions. You should collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. Before you start monitoring Step Functions, however, you should create a monitoring plan that includes answers to the following questions:

- What are you 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, by measuring performance at various times and under different load conditions. As you monitor Step Functions, you should consider storing historical monitoring data. This stored data will give you a baseline to compare against current performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

For example, with Step Functions, you can monitor how many activities or lambda tasks failed 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 items:

- **ExecutionsStarted**
- **ExecutionsTimedOut**
- **ActivititesStarted**
- **ActivititesTimedOut**
- **LambdaFunctionsStarted**
- **LambdaFunctionsTimedOut**

The following sections describe metrics that Step Functions provides for CloudWatch that you can use to track your state machines and activities and set alarms on threshold values that you choose. You can view metrics using the AWS Management Console.

- *Metrics that Report a Time Interval*
- *Metrics that Report a Count*
- *State Machine Metrics*
- *Viewing Metrics for Step Functions*

- *Setting Alarms*

1.12.1 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, lambda function and activity timeouts, and have similar names.

For example, the **ActivityRunTime** metric measures the time it took for the activity to complete after it began executing, which is the same time period for which you can set a timeout value.

Note: In the Cloud Watch console, you will get the best results if you choose *average* as the *statistic* to display for time interval metrics.

1.12.2 Metrics that Report a Count

Some of the Step Functions CloudWatch metrics report results as a *count*. For example, **ExecutionsFailed**, records the number of state machine executions which failed.

Note: In the Cloud Watch console, you will get the best results if you choose *sum* as the *statistic* to display for count metrics.

1.12.3 State Machine Metrics

The following metrics are available for Step Functions state machines:

Execution Metrics

Dimension	Description
StateMachineArn	The ARN of the state machine for the execution in question.

Metric	Description
ExecutionTime	The time interval, in milliseconds, between the time the execution started and the time it closed.
ExecutionsStarted	The number of executions started.
ExecutionsSucceeded	The number of executions that completed successfully.
ExecutionsFailed	The number of executions that failed.
ExecutionsAborted	The number of executions that were aborted/terminated.
ExecutionsTimedOut	The number of executions that timed out for any reason.

Activity Metrics

Dimension	Description
ActivityArn	The ARN of the activity.

Metric	Description
ActivityScheduleTime	The time interval, in milliseconds, that the activity stayed in the schedule state.
ActivityRunTime	The time interval, in milliseconds, between the time the activity was started and when it was closed.
ActivityTime	The time interval, in milliseconds, between the time the activity was scheduled and when it was closed.
ActivitiesScheduled	The number of activities that were scheduled.
ActivitiesStarted	The number of activities that were started.
ActivitiesSucceeded	The number of activities that completed successfully.
ActivitiesFailed	The number of activities that failed.
ActivitiesTimedOut	The number of activities that were timed out on close.
ActivitiesHeartbeat-TimedOut	The number of activities that were timed out due to a heartbeat timeout.

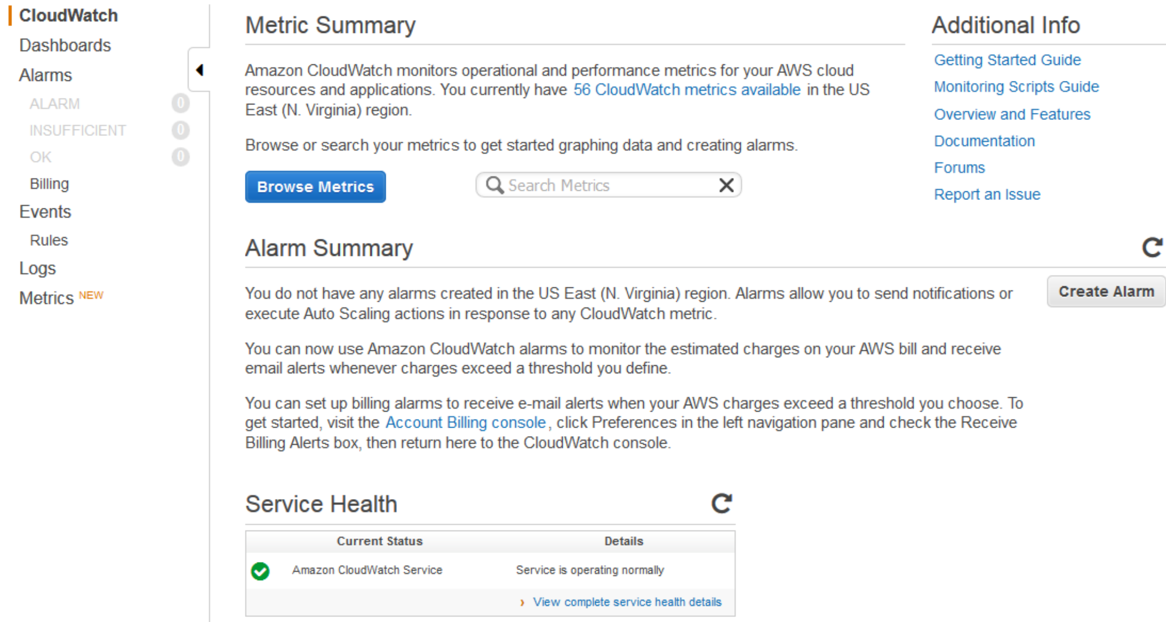
Lambda Function Metrics

Dimension	Description
LambdaFunctionArn	The ARN of the lambda function.

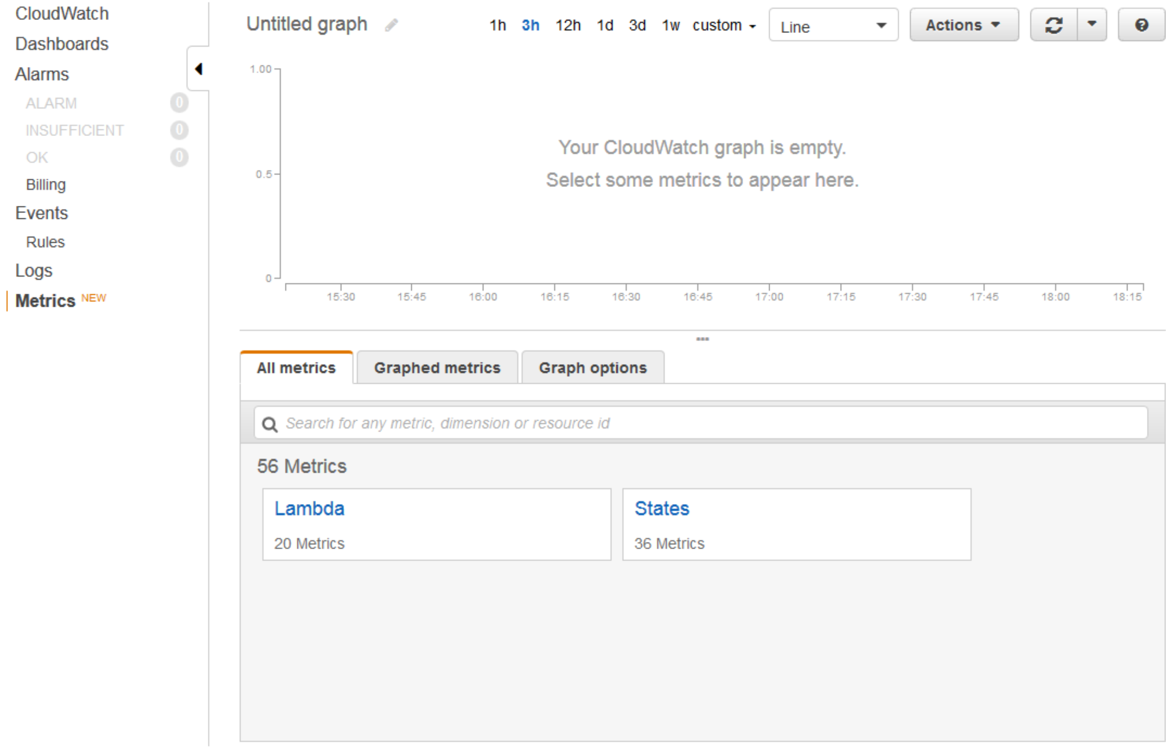
Metric	Description
LambdaFunctionSched-uleTime	The time interval, in milliseconds, that the activity stayed in the schedule state.
LambdaFunctionRun-Time	The time interval, in milliseconds, between the time the lambda function was started and when it was closed.
LambdaFunctionTime	The time interval, in milliseconds, between the time the lambda function was scheduled and when it was closed.
LambdaFunctionsSched-uled	The number of lambda functions that were scheduled.
LambdaFunctionsStarted	The number of lambda functions that were started.
LambdaFunctionsSuc-ceeded	The number of lambda functions that completed successfully.
LambdaFunctionsFailed	The number of lambda functions that failed.
LambdaFunctionsTimed-Out	The number of lambda functions that were timed out on close.
LambdaFunctionsHeart-beatTimedOut	The number of lambda functions that were timed out due to a heartbeat timeout.

1.12.4 Viewing Metrics for Step Functions

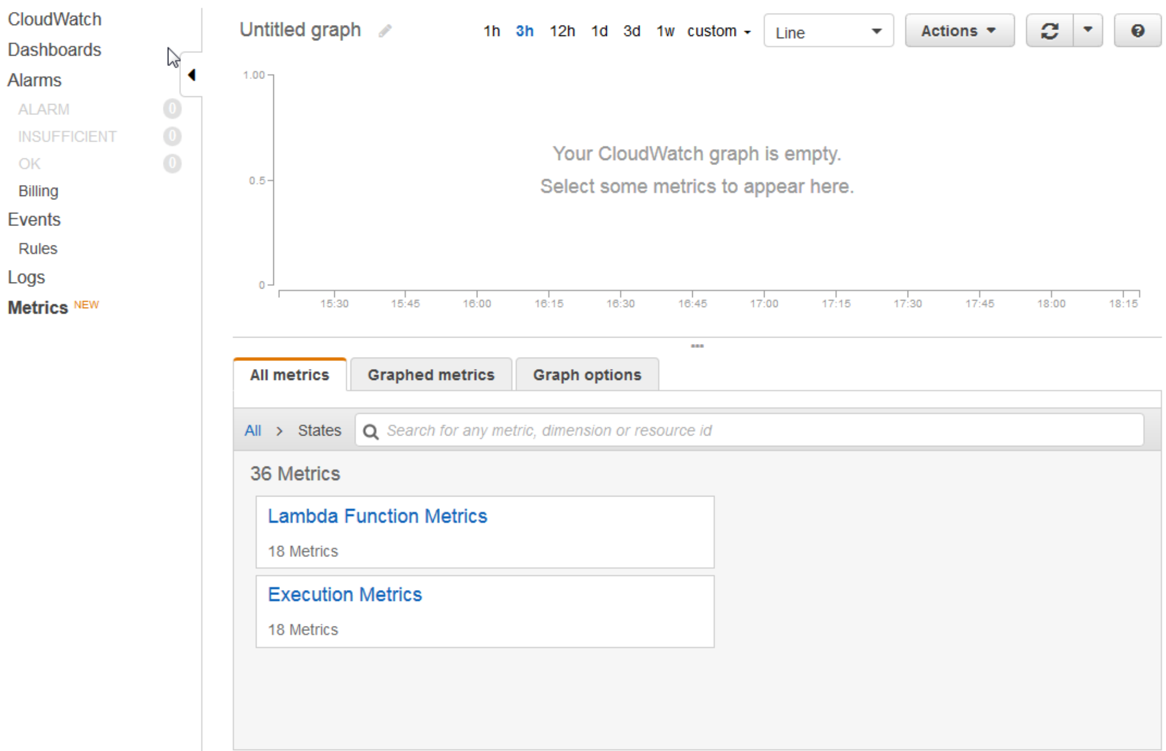
1. Open the AWS Management Console and navigate to CloudWatch.



2. In the navigation pane, click *Metrics*.

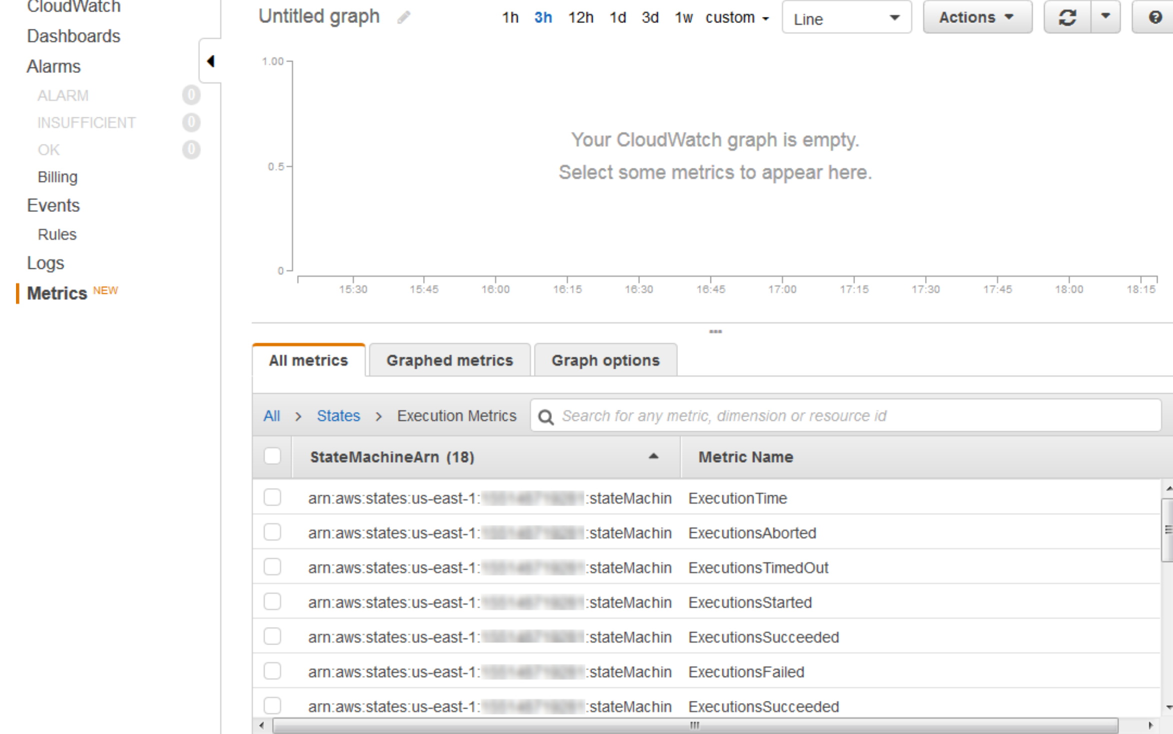


3. In the area beneath the *All Metrics* tab, click on *States*.



If you have run any executions recently, you will see up to three types of metrics presented: *Lambda Function Metrics*, *Activity Function Metrics* and *Execution Metrics*.

4. Click on one of the metric types to see a list of the metrics.



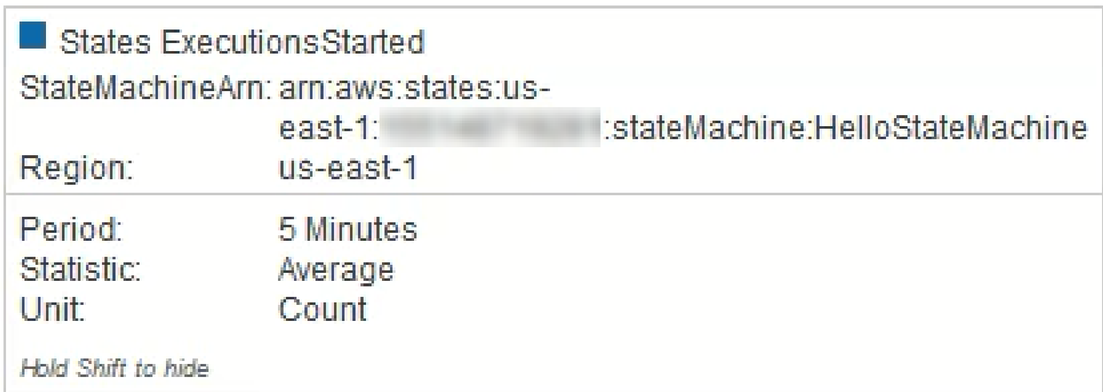
You can use the interactive headings above the columns in the list to sort your metrics by **StateMachineArn** or **Metric Name**.

You can view graphs for metrics by clicking the boxes next to the metric row in the list. Change the graph parameters using the time range controls above the graph view. You can choose custom time ranges specified in relative or absolute values (i.e. specific days and times). You can also use the drop-down box to display values as lines, stacked areas, or numbers (values).

For details about a metric on the graph, place your cursor over the metric color code which appears below the graph.

ExecutionsAborted ExecutionsStarted ExecutionsSucceeded ExecutionsTimedOut

A detail of the metric will be shown.



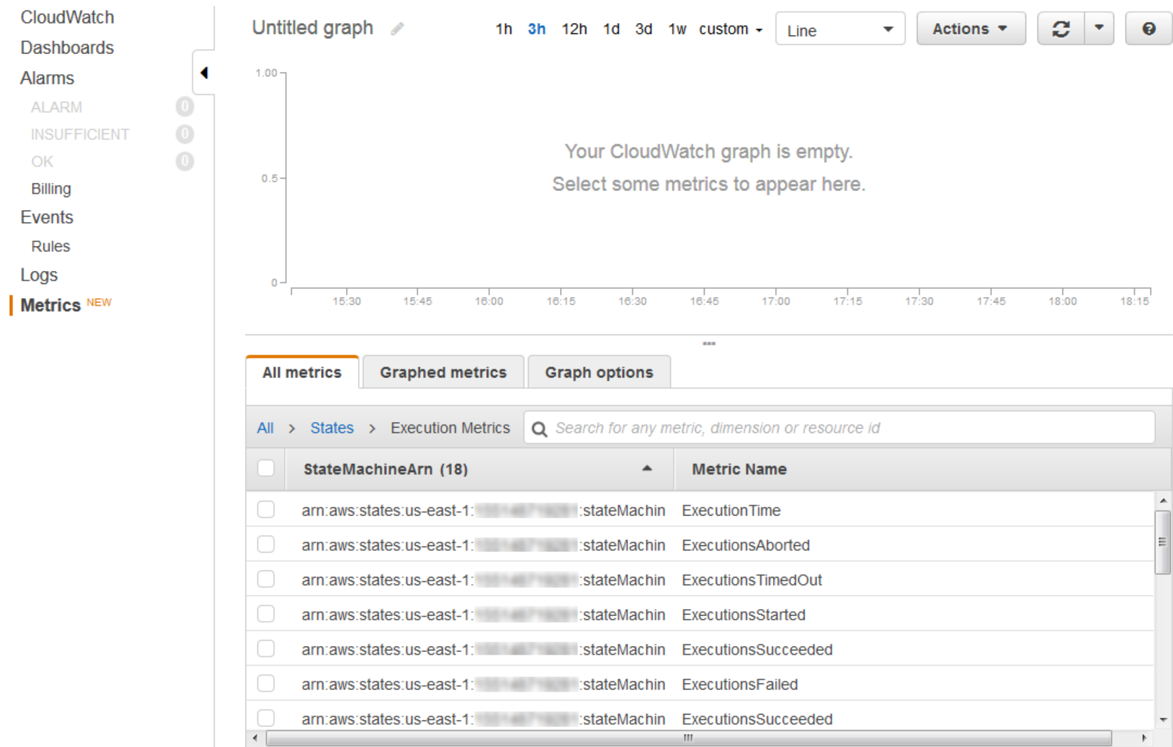
For more information about working with CloudWatch metrics, see Viewing, Graphing, and Publishing Metrics in the CloudWatch Developer Guide.

1.12.5 Setting Alarms

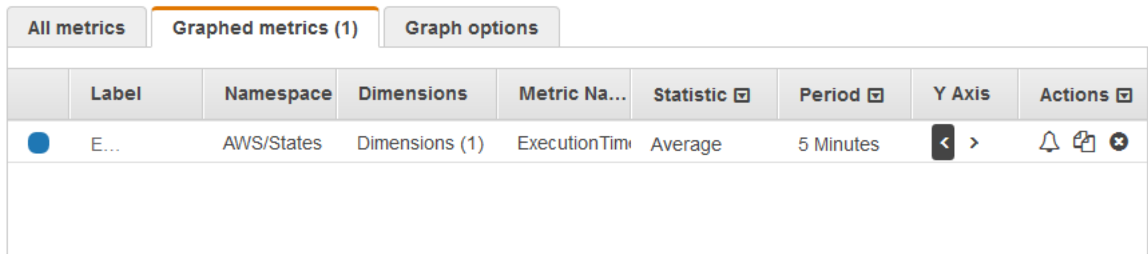
You can use CloudWatch alarms to perform actions such as notifying you when an alarm threshold is reached. For example, you can set an alarm to send a notification to an Amazon SNS topic or to send an email when the **StateMachinesFailed** metric rises above a certain threshold.

To set an alarm on any of your metrics

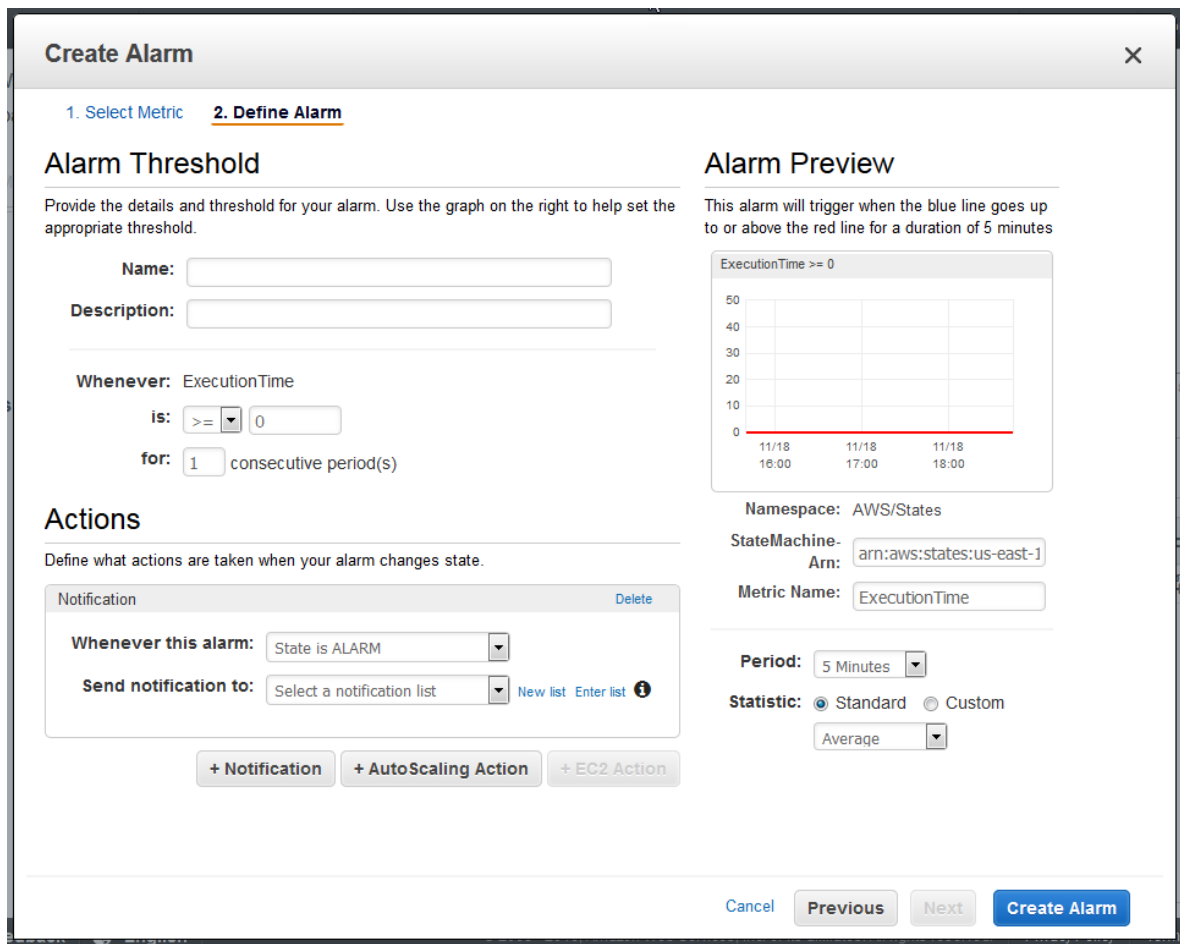
1. In the list of metrics under the *All metrics* tab, choose a metric by clicking its box to view its graph.



2. Click on the *Graphed metrics* tab to view the metrics shown in the graph.



3. Under the *Actions* column for a metric, click on the bell icon. The *Create Alarm* screen will open.



4. On the *Create Alarm* screen, enter the alarm threshold value, period parameters, and actions to take.

For more information about setting and using CloudWatch alarms, see Creating Amazon CloudWatch Alarms in the CloudWatch Developer Guide.

1.13 Logging AWS Step Functions API Calls with 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.

1.13.1 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 *CloudTrail Event Reference*.

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.

1.13.2 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.
↪3a0801c5-37"
    },
    "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"
```

```
}
```

1.14 Document History

This topic lists major changes to the *AWS Step Functions Developer Guide* over the course of its history.

- **Latest documentation update:** Dec 01, 2016

December 1, 2016 General release.

1.15 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 click **Create a Free Account**.