# Table of Contents

What Is AWS Step Functions? .................................................................................................................. 1  
Overview of Step Functions .................................................................................................................. 1  
Supported Regions ................................................................................................................................... 1 
   About Amazon Web Services ................................................................................................................ 2  
Getting Started ........................................................................................................................................ 3  
   Step 1: Creating a State Machine ......................................................................................................... 3  
      To create the state machine .................................................................................................................. 3  
   Step 2: Starting a New Execution ......................................................................................................... 4  
      To start a new execution ....................................................................................................................... 4  
   Step 3: (Optional) Update a State Machine .......................................................................................... 5  
      To update a state machine .................................................................................................................... 5  
Next Steps ................................................................................................................................................ 6  
Tutorials .................................................................................................................................................. 7  
   Development Options ............................................................................................................................ 7  
      Step Functions Console ...................................................................................................................... 7  
      AWS SDKs ......................................................................................................................................... 7  
      HTTPS Service API ............................................................................................................................. 7  
      Development Environments ............................................................................................................... 8  
      Endpoints .......................................................................................................................................... 8  
      AWS CLI ............................................................................................................................................ 8  
Creating a Lambda State Machine ........................................................................................................... 8  
   Step 1: Creating an IAM Role for Lambda ............................................................................................. 9  
   Step 2: Creating a Lambda Function .................................................................................................... 9  
   Step 3: Testing the Lambda Function .................................................................................................. 10  
   Step 4: Creating a State Machine ....................................................................................................... 10  
   Step 5: Starting a New Execution ....................................................................................................... 12  
Creating a Lambda State Machine Using AWS CloudFormation ................................................................ 13  
   Step 1: Setting Up Your AWS CloudFormation Template .................................................................. 13  
   Step 2: Using the AWS CloudFormation Template to Create a Lambda State Machine ................. 15  
   Step 3: Starting a State Machine Execution ....................................................................................... 16  
Creating an Activity State Machine ........................................................................................................ 17  
   Step 1: Creating a New Activity .......................................................................................................... 17  
   Step 2: Creating a State Machine ....................................................................................................... 18  
   Step 3: Implementing a Worker ......................................................................................................... 19  
   Step 4: Starting an Execution ............................................................................................................ 21  
   Step 5: Running and Stopping the Worker ......................................................................................... 21  
Handling Error Conditions Using a State Machine .................................................................................. 22  
   Step 1: Creating an IAM Role for Lambda .......................................................................................... 22  
   Step 2: Creating a Lambda Function That Fails .............................................................................. 23  
   Step 3: Testing the Lambda Function ................................................................................................. 23  
   Step 4: Creating a State Machine with a Catch Field ..................................................................... 24  
   Step 5: Starting a New Execution ....................................................................................................... 26  
Starting a State Machine Execution Using CloudWatch Events .................................................................. 27  
   Step 1: Creating a State Machine ....................................................................................................... 27  
   Step 2: Creating a CloudWatch Events Rule ...................................................................................... 27  
Creating a Step Functions API Using API Gateway .............................................................................. 29  
   Step 1: Creating an IAM Role for API Gateway ............................................................................... 30  
   Step 2: Creating your API Gateway API ......................................................................................... 30  
   Step 3: Testing and Deploying the API Gateway API ....................................................................... 32  
Iterating a Loop Using Lambda .......................................................................................................... 34  
   Step 1: Create a Lambda Function to Iterate a Count ..................................................................... 34  
   Step 2: Test the Lambda Function ..................................................................................................... 35  
   Step 3: Create a State Machine ......................................................................................................... 36  
   Step 4: Start a New Execution ............................................................................................................ 39
## Table of Contents

- **How Step Functions Works** .................................................................................................................. 49
  - States ..................................................................................................................................... 49
  - Tasks .............................................................................................................................. 50
  - Activities .......................................................................................................................... 50
  - Creating an Activity ........................................................................................................... 51
  - Writing a Worker ............................................................................................................... 51
  - Example Activity Worker in Ruby ....................................................................................... 51
- **Transitions** ................................................................................................................................. 57
  - State Machine Data .............................................................................................................. 58
  - Data Format ....................................................................................................................... 58
  - State Machine Input/Output .................................................................................................. 59
  - State Input/Output .............................................................................................................. 59
- **Executions** ................................................................................................................................. 61
  - Error Handling ....................................................................................................................... 61
  - Error Names .......................................................................................................................... 61
  - Retrying After an Error .......................................................................................................... 62
  - Fallback States ..................................................................................................................... 63
  - Examples Using Retry and Using Catch ............................................................................. 64
- **Read Consistency** .......................................................................................................................... 67
  - Templates ............................................................................................................................... 68
  - Sample Projects .................................................................................................................... 68
  - Job Status Poller .................................................................................................................... 68
  - Task Timer ............................................................................................................................. 71
- **Amazon States Language** ............................................................................................................. 74
  - Example Amazon States Language Specification ...................................................................... 74
  - State Machine Structure ........................................................................................................ 75
  - States ..................................................................................................................................... 76
  - Common State Fields ........................................................................................................... 77
  - Pass ................................................................................................................................ 77
  - Task ................................................................................................................................ 78
  - Choice .............................................................................................................................. 80
  - Wait ................................................................................................................................ 83
  - Succeed ............................................................................................................................. 84
  - Fail ................................................................................................................................ 85
  - Parallel ............................................................................................................................... 85
- **Input and Output Processing** ......................................................................................................... 87
  - Paths ................................................................................................................................ 87
  - Reference Paths ................................................................................................................... 87
- **Errors** ...................................................................................................................................... 90
  - Error Representation .......................................................................................................... 90
  - Retrying After an Error ......................................................................................................... 90
  - Fallback States ..................................................................................................................... 92
- **Best Practices** .......................................................................................................................... 94
  - Use Timeouts to Avoid Stuck Executions ............................................................................. 94
  - Use ARNs Instead of Passing Large Payloads ...................................................................... 94
  - Avoid Reaching the History Limit ....................................................................................... 94
- **Limits** ...................................................................................................................................... 96
  - General Limits ....................................................................................................................... 96
  - Limits Related to Accounts .................................................................................................. 96
  - Limits Related to State Machine Executions ......................................................................... 97
<table>
<thead>
<tr>
<th>Limits Related to Task Executions</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limits Related to API Action Throttling</td>
<td>98</td>
</tr>
<tr>
<td>Limits Related to State Throttling</td>
<td>98</td>
</tr>
<tr>
<td>Monitoring and Logging</td>
<td>100</td>
</tr>
<tr>
<td>Monitoring Step Functions Using CloudWatch</td>
<td>100</td>
</tr>
<tr>
<td>Metrics that Report a Time Interval</td>
<td>101</td>
</tr>
<tr>
<td>Metrics that Report a Count</td>
<td>101</td>
</tr>
<tr>
<td>State Machine Metrics</td>
<td>101</td>
</tr>
<tr>
<td>Viewing Metrics for Step Functions</td>
<td>103</td>
</tr>
<tr>
<td>Setting Alarms for Step Functions</td>
<td>104</td>
</tr>
<tr>
<td>Logging Step Functions using CloudTrail</td>
<td>106</td>
</tr>
<tr>
<td>Step Functions Information in CloudTrail</td>
<td>106</td>
</tr>
<tr>
<td>Understanding Step Functions Log File Entries</td>
<td>107</td>
</tr>
<tr>
<td>Security</td>
<td>111</td>
</tr>
<tr>
<td>Authentication</td>
<td>111</td>
</tr>
<tr>
<td>Creating IAM Roles for AWS Step Functions</td>
<td>112</td>
</tr>
<tr>
<td>Create a Role for Step Functions</td>
<td>112</td>
</tr>
<tr>
<td>Creating Granular IAM Permissions for Non-Admin Users</td>
<td>112</td>
</tr>
<tr>
<td>Service-Level Permissions</td>
<td>113</td>
</tr>
<tr>
<td>State Machine-Level Permissions</td>
<td>113</td>
</tr>
<tr>
<td>Execution-Level Permissions</td>
<td>114</td>
</tr>
<tr>
<td>Activity-Level Permissions</td>
<td>114</td>
</tr>
<tr>
<td>Related Step Functions Resources</td>
<td>116</td>
</tr>
<tr>
<td>Document History</td>
<td>117</td>
</tr>
<tr>
<td>AWS Glossary</td>
<td>124</td>
</tr>
</tbody>
</table>
What Is AWS Step Functions?

AWS Step Functions is a web service that enables you to coordinate the components of distributed applications and microservices using visual workflows. You build applications from individual components that each perform a discrete function, or task, allowing you to scale and change applications quickly. Step Functions provides a reliable way to coordinate components and step through the functions of your application. Step Functions provides a graphical console to visualize the components of your application as a series of steps. It automatically triggers and tracks each step, and retries when there are errors, so your application executes in order and as expected, every time. Step Functions logs the state of each step, so when things do go wrong, you can diagnose and debug problems quickly.

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

You can run your tasks on the AWS Cloud, on your own servers, or on any system that has access to AWS. Step Functions can be accessed and used with the Step Functions console, the AWS SDKs, or an HTTP API. This guide shows you how to develop, test, and troubleshoot your own state machine using these methods.

Overview of Step Functions

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

- Step Functions is based on the concepts of tasks (p. 50) and state machines (p. 49).
- You define state machines using the JSON-based Amazon States Language (p. 74).
- The Step Functions console displays a graphical view of your state machine's structure, which provides you with a way to visually check your state machine's logic and monitor executions.

Supported Regions

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

- US East (Ohio)
- US East (N. Virginia)
- US West (Oregon)
- US West (N. California)
- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- Asia Pacific (Seoul)
- EU (Frankfurt)
- EU (Ireland)
- EU (London)
- Canada (Central)
- Asia Pacific (Singapore)
About Amazon Web Services

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

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

Topics

• Step 1: Creating a State Machine (p. 3)
• Step 2: Starting a New Execution (p. 4)
• Step 3: (Optional) Update a State Machine (p. 5)
• Next Steps (p. 6)

Step 1: Creating a State Machine

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

To create the state machine

1. Sign in to the Step Functions console, and then choose Get Started.
2. On the Create a state machine page, select Templates and then choose Hello world.

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

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

This JSON text defines a Pass state named HelloWorld. For more information, see State Machine Structure (p. 75).

3. Use the graph in the Visual Workflow pane to check that your Amazon States Language code describes your state machine correctly.
If you don’t see the graph, choose ☐ in the Visual Workflow pane.

4. Create or enter an IAM role.
   
   - To create a new IAM role for Step Functions, choose Create a role for me, and then choose I acknowledge that Step Functions will create an IAM role which allows access to my Lambda functions.
   
   - If you have previously created an IAM role for Step Functions (p. 112), choose I will provide an IAM role ARN and enter your existing IAM role ARN.

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

5. Choose Create State Machine.

---

**Step 2: Starting a New Execution**

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

**To start a new execution**

1. On the HelloWorld page, choose **New execution**.
   
   The **New execution** window is displayed.

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

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

3. Choose **Start Execution**.

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

4. (Optional) In the **Execution Details** section, choose the **Info** tab to view the **Execution Status** and the **Started** and **Closed** timestamps.

5. To view the results of your execution, choose the **Output** tab.
Step 3: (Optional) Update a State Machine

You can update your state machine for future executions.

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

To update a state machine

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

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

3. (Optional) Select a new IAM role from the **IAM role for executions** list.
   
   **Note**
   You can also select **Create new role** to create a new IAM role. For more information, see [Creating IAM Roles for AWS Step Functions](p. 112).

4. Choose **Save** and then **Execute**.
5. On the **New execution** page choose **Start Execution**.
6. To view the results of your execution, select the **HelloWorld** state in the **Visual workflow** and expand the **Output** section under **Step details**.
Note
The output text matches your newly updated state machine.

Next Steps

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

- Create a Lambda state machine (p. 8)
- Create a Lambda state machine using AWS CloudFormation (p. 13)
- Create an activity state machine (p. 17)
- Handle error conditions using a state machine (p. 22)
- Start a state machine using Amazon CloudWatch Events (p. 27)
- Create a Step Functions API using Amazon API Gateway (p. 29)
Tutorials

The following tutorials will help you get started working with AWS Step Functions. To complete these tutorials, you'll need an AWS account. If you don't have an AWS account, navigate to http://aws.amazon.com/ and choose Sign in to the Console.

Topics

- Development Options (p. 7)
- Creating a Lambda State Machine (p. 8)
- Creating a Lambda State Machine Using AWS CloudFormation (p. 13)
- Creating an Activity State Machine (p. 17)
- Handling Error Conditions Using a State Machine (p. 22)
- Starting a State Machine Execution Using CloudWatch Events (p. 27)
- Creating a Step Functions API Using API Gateway (p. 29)
- Iterating a Loop Using Lambda (p. 34)
- Continue as a New Execution (p. 40)

Development Options

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

Step Functions Console

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

The Creating a Lambda State Machine (p. 8) tutorial uses this technique to create a simple state machine, execute it, and view its results.

AWS SDKs

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

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

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

Note

Step Functions supports only an HTTPS endpoint.

HTTPS Service API

Step Functions provides service operations accessible through HTTPS requests. You can use these operations to communicate directly with Step Functions and to develop your own libraries in any language that can communicate with Step Functions through HTTPS.
You can develop state machines, workers, or state machine starters using the service API actions. You can also access visibility operations through the API actions to develop your own monitoring and reporting tools. For detailed information on API actions, see the AWS Step Functions API Reference.

Development Environments

You must set up a development environment appropriate to the programming language that you plan to use. For example, if you intend to develop for Step Functions with Java, you should install a Java development environment (such as the SDK for Java) on each of your development workstations. If you use Eclipse IDE for Java Development, you should also install the Toolkit for Eclipse. This Eclipse plug-in adds features useful for AWS development.

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

Endpoints

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

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

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

AWS CLI

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

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

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

Creating a Lambda State Machine

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

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

Topics
Step 1: Creating an IAM Role for Lambda

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

Lambda requires you to assign an IAM role when you create a Lambda function in the same way Step Functions requires you to assign an IAM role when you create a state machine.

To create a role for Lambda

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

To create a role (console)

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane of the IAM console, choose Roles. Then choose Create role.
3. Choose the AWS Service role type, and then choose Lambda.
4. Choose the Lambda use case. Use cases are defined by the service to include the trust policy required by the service. Then choose Next: Permissions.
5. Choose one or more permissions policies to attach to the role. Select the box next to the policy that assigns the permissions that you want the role to have, and then choose Next: Review.
6. Enter a Role name.
7. (Optional) For Role description, edit the description for the new service-linked role.
8. Review the role and then choose Create role.

Step 2: Creating a Lambda Function

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

To create the Lambda function

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

1. Log in to the Lambda console and choose Create a function.
2. In the Blueprints section, choose Author from scratch.
3. In the Basic information section, configure your Lambda function:
   a. For Name, type HelloFunction.
   b. For Role, select Choose an existing role.
   c. For Existing role, select the Lambda role that you created earlier (p. 9).
Step 3: Testing the Lambda Function

Test your Lambda function to see it in operation.

To test your Lambda function

1. On the Configure test event dialog box, type HelloFunction for Event name.
2. Replace the example data with the following:

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

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

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

Step 4: Creating a State Machine

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

To create the state machine

1. Log in to the Step Functions console and choose Create a state machine.
2. On the Create a state machine page, select Author from scratch and enter a Name your state machine, for example LambdaStateMachine.

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

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

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

3. Create or enter an IAM role.

   - To create a new IAM role for Step Functions, choose Create a role for me, and then choose I acknowledge that Step Functions will create an IAM role which allows access to my Lambda functions.
   
   - If you have previously created an IAM role for Step Functions (p. 112), choose I will provide an IAM role ARN and enter your existing IAM role ARN.

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

4. In the State machine definition pane, add the following state machine definition using the ARN of the Lambda function that you created earlier (p. 9), for example:

   ```json
   {
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda function",
   "StartAt": "HelloWorld",
   "States": {
   "HelloWorld": {
   "Type": "Task",
   "End": true
   }
   }
   }
   ```

   This is a description of your state machine using the Amazon States Language. It defines a single Task state named HelloWorld. For more information, see State Machine Structure (p. 75).

5. Use the graph in the Visual Workflow pane to check that your Amazon States Language code describes your state machine correctly.
If you don’t see the graph, choose ☰ in the **Visual Workflow** pane.

6. Choose **Create State Machine**.

### Step 5: Starting a New Execution

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

#### To start a new execution

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

   The **New execution** page is displayed.

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

   **Note**

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

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

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

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

4. Choose **Start Execution**.

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

5. To view the results of your execution, select the **HelloWorld** state in the **Visual workflow** and expand the **Output** section uneder **Step details**.
Creating a Lambda State Machine Using AWS CloudFormation

This tutorial shows you how to create a basic AWS Lambda function and start a state machine execution automatically. You will use the AWS CloudFormation console and a YAML template to create the stack (IAM roles, the Lambda function, and the state machine). You will then use the AWS Step Functions console to start the state machine execution. For more information, see Working with CloudFormation Templates and the AWS::StepFunctions::StateMachine resource in the AWS CloudFormation User Guide.

Topics
- Step 1: Setting Up Your AWS CloudFormation Template (p. 13)
- Step 2: Using the AWS CloudFormation Template to Create a Lambda State Machine (p. 15)
- Step 3: Starting a State Machine Execution (p. 16)

Step 1: Setting Up Your AWS CloudFormation Template

Before you use the example YAML template (p. 15), you should understand its separate parts.

To create an IAM role for Lambda

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

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

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

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

```
MyLambdaFunction:
  Type: "AWS::Lambda::Function"
  Properties:
    Handler: "index.handler"
    Role: !GetAtt [ LambdaExecutionRole, Arn ]
    Code:
      ZipFile: |
      exports.handler = (event, context, callback) => {
        callback(null, "Hello World!");
      };
    Runtime: "nodejs4.3"
    Timeout: "25"
```

To create an IAM role for the state machine execution

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

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

To create a Lambda state machine

Define the Lambda state machine.

```
MyStateMachine:
  Type: "AWS::StepFunctions::StateMachine"
  Properties:
    DefinitionString:
      !Sub
        |- |
        { "Comment": "A Hello World AWL example using an AWS Lambda function",
          "StartAt": "HelloWorld",
          "States": {
Step 2: Using the AWS CloudFormation Template to Create a Lambda State Machine

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

To create the Lambda state machine

1. Copy the following example YAML data to a file named MyStateMachine.yaml.

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

  MyLambdaFunction:
    Type: "AWS::Lambda::Function"
    Properties:
      Handler: "index.handler"
      Role: !GetAtt [ LambdaExecutionRole, Arn ]
      Code:
        ZipFile: |
          exports.handler = (event, context, callback) => {
            callback(null, "Hello World!");
          };
      Runtime: "nodejs4.3"
      Timeout: "25"

  StatesExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
        - Effect: "Allow"
          Principal:
            Service:
            - !Sub states.${AWS::Region}.amazonaws.com
          Action: "sts:AssumeRole"
        - Effect: "Allow"
          Principal:
            Service:
            - !Sub lambda.${AWS::Region}.amazonaws.com
          Action: "sts:AssumeRole"
        Path: "/"
        Policies:
        - PolicyName: StatesExecutionPolicy
```
Step 3: Starting a State Machine Execution

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

To start the state machine execution

1. Log in to the Step Functions console and choose the name of the state machine that you created using AWS CloudFormation.
3. The New execution page is displayed.
4. (Optional) To help identify your execution, you can specify an ID for it in the Enter an execution name box. If you don’t enter an ID, Step Functions generates a unique ID automatically.
Creating an Activity State Machine

You can coordinate task code in your state machine. This tutorial introduces you to creating an activity-based state machine using Java and AWS Step Functions.

To complete this tutorial you'll need the following:

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

Topics

- Step 1: Creating a New Activity (p. 17)
- Step 2: Creating a State Machine (p. 18)
- Step 3: Implementing a Worker (p. 19)
- Step 4: Starting an Execution (p. 21)
- Step 5: Running and Stopping the Worker (p. 21)

Step 1: Creating a New Activity

You must make Step Functions aware of the activity whose worker (a program) you want to create. Step Functions responds with an ARN that establishes an identity for the activity. Use this identity to coordinate the information passed between your state machine and worker.
Important
Ensure that your activity task is under the same AWS account as your state machine.

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

Step 2: Creating a State Machine
Create a state machine that will determine when your activity is invoked and when your worker should perform its primary work, collect its results, and return them.

To create the state machine
1. In the Step Functions console, choose State machines in the left navigation panel.
2. On the State machines page, choose Create state machine, select Author from scratch, and enter a name under Details (for example ActivityStateMachine).
   
   Note
   State machine names must be 1–80 characters in length, must be unique for your account and region, and must not contain any of the following:
   
   - Whitespace
   - Wildcard characters (\? *\)
   - Bracket characters (< > { } [ ] )
   - Special characters (!: ; , | ^ ~ $ # % & "')
   - Control characters (\\u0000 - \\ or \\ - \\).

   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. Create or enter an IAM role.
   
   - To create a new IAM role for Step Functions, choose Create a role for me, and then choose I acknowledge that Step Functions will create an IAM role which allows access to my Lambda functions.
   - If you have previously created an IAM role for Step Functions (p. 112), choose I will provide an IAM role ARN and enter your existing IAM role ARN.

   Note
   If you delete the IAM role that Step Functions creates, Step Functions can’t recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can’t restore its original settings later.
4. Under State machine definition, enter the following code, and include the ARN of the activity task that you created earlier (p. 18) in the Resource field, for example:
Step 3: Implementing a Worker

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

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

**Note**
For a more complete example of an activity worker, see Example Activity Worker in Ruby (p. 51). This example provides an implementation based on best practices, that can be used as a reference for your activity worker. The code implements a consumer-producer pattern with a configurable number of threads for pollers and activity workers.

**To implement the worker**

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

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

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

    public static void main(final String[] args) throws Exception {
        GreeterActivities greeterActivities = new GreeterActivities();
        ClientConfiguration clientConfiguration = new ClientConfiguration();
        clientConfiguration.setSocketTimeout((int) TimeUnit.SECONDS.toMillis(70));
        AWSStepFunctions client = AWSStepFunctionsClientBuilder.standard()
            .withRegion(Regions.US_EAST_1)
            .withCredentials(new EnvironmentVariableCredentialsProvider())
            .withClientConfiguration(clientConfiguration)
            .build();

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

            if (getActivityTaskResult.getTaskToken() != null) {
                try {
                    JsonNode json = Jackson.jsonNodeOf(getActivityTaskResult.getInput());
                    String greetingResult =
                        greeterActivities.getGreeting(json.get("who").textValue());
                    client.sendTaskSuccess(
                        new SendTaskSuccessRequest().withOutput(
                            greetingResult).withTaskToken(getActivityTaskResult.getTaskToken()));
                } catch (Exception e) {
                    client.sendTaskFailure(new SendTaskFailureRequest().withTaskToken(
                        getActivityTaskResult.getTaskToken()));
                } else {
                    Thread.sleep(1000);
                }
            } else {
                Thread.sleep(1000);
            }
        }
    }
}
```

**Note**

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

To give Step Functions sufficient time to process the request, setSocketTimeout is set to 70 seconds.

3. In the parameter list of the GetActivityTaskRequest().withActivityArn() constructor, replace the ACTIVITY_ARN value with the ARN of the activity task that you created earlier (p. 18).

Step 4: Starting an Execution

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

To start the execution

1. On the ActivityStateMachine page, choose Start execution.

   The New execution page is displayed.

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

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

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

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


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

5. In the Execution Details section, choose Info to view the Execution Status and the Started and Closed timestamps.

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

Step 5: Running and Stopping the Worker

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

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

To run and stop the worker

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

3. Compile the file:

   ```
   $ javac GreeterActivities.java
   ```

4. Run the file:

   ```
   $ java GreeterActivities
   ```

5. In the Step Functions console, navigate to the Execution Details page.

6. When the execution completes, choose Output to see the results of your execution.

7. Stop the worker.

### Handling Error Conditions Using a State Machine

In this tutorial, you create an AWS Step Functions state machine with a Catch field which uses an AWS Lambda function to respond with conditional logic based on error message type, a method called function error handling. For more information, see Function Error Handling in the AWS Lambda Developer Guide.

**Note**

You can also create state machines that Retry on timeouts or those that use Catch to transition to a specific state when an error or timeout occurs. For examples of these error handling techniques, see Examples Using Retry and Using Catch (p. 64).

**Topics**

- Step 1: Creating an IAM Role for Lambda (p. 22)
- Step 2: Creating a Lambda Function That Fails (p. 23)
- Step 3: Testing the Lambda Function (p. 23)
- Step 4: Creating a State Machine with a Catch Field (p. 24)
- Step 5: Starting a New Execution (p. 26)

### Step 1: Creating an IAM Role for Lambda

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

Lambda requires you to assign an IAM role when you create a Lambda function in the same way Step Functions requires you to assign an IAM role when you create a state machine.

**To create a role for Lambda**

1. Sign in to the IAM console and choose Roles, Create role.

2. On the Select type of trusted entity page, under AWS service, select Lambda from the list, and then choose Next: Permissions.

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

The IAM role appears in the list of roles.

**Step 2: Creating a Lambda Function That Fails**

Use a Lambda function to simulate an error condition.

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

**To create a Lambda function that fails**

1. Log in to the Lambda console and choose Create a function.
2. In the Blueprints section, type step-functions into the filter, and then choose the step-functions-error blueprint.
3. In the Basic information section, configure your Lambda function:
   a. For Name, type FailFunction.
   b. For Role, select Choose an existing role.
   c. For Existing role, select the Lambda role that you created earlier (p. 22).

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

4. The following code is displayed in the Lambda function code pane:

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

     const error = new CustomError('This is a custom error!');
     callback(error);
   }
   
   The context object returns the error message This is a custom error!.

5. Choose Create function.

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

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

**Step 3: Testing the Lambda Function**

Test your Lambda function to see it in operation.
To test your Lambda function

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

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

Step 4: Creating a State Machine with a Catch Field

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

To create the state machine

1. Log in to the Step Functions console and choose Create state machine.
2. On the Create a state machine page, select Templates and choose Catch failure.
3. Name your state machine, for example CatchStateMachine.

    Note
    State machine names must be 1–80 characters in length, must be unique for your account and region, and must not contain any of the following:
    
    - Whitespace
    - Wildcard characters (\? * )
    - Bracket characters (< > { } [ ])
    - Special characters (\: ;) , \\ | ^ ~ $ # % & ` ”)
    - Control characters (\u0000 - \u001f or \u007f - \u009f).

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

4. Create or enter an IAM role.

    - To create a new IAM role for Step Functions, choose Create a role for me, and then choose I acknowledge that Step Functions will create an IAM role which allows access to my Lambda functions.
    - If you have previously created an IAM role for Step Functions (p. 112), choose I will provide an IAM role ARN and enter your existing IAM role ARN.

    Note
    If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.
5. In the **Code** pane, add the ARN of the Lambda function that you created earlier (p. 23) to the **Resource** field, for example:

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

This is a description of your state machine using the Amazon States Language. It defines a single **Task** state named **CreateAccount**. For more information, see **State Machine Structure** (p. 75).

For more information about the syntax of the **Retry** field, see **Retrying After an Error** (p. 90).

**Note**

Unhandled errors in Lambda are reported as **Lambda.Unknown** in the error output. These include out-of-memory errors, function timeouts, and hitting the concurrent Lambda invoke limit. You can match on **Lambda.Unknown**, **States.ALL**, or **States.TaskFailed** to handle these errors. For more information about Lambda **Handled** and **Unhandled** errors, see **FunctionError** in the **AWS Lambda Developer Guide**.

6. Use the graph in the **Visual Workflow** pane to check that your Amazon States Language code describes your state machine correctly.
Step 5: Starting a New Execution

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

To start a new execution


   The New execution page is displayed.

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

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

3. Choose Start Execution.

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

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

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

5. To view your custom error message, select CreateAccount in the Visual workflow and expand the Output section.
Starting a State Machine Execution Using CloudWatch Events

You can execute a Step Functions state machine in response to an event pattern or on a schedule using Amazon CloudWatch Events. This tutorial shows how to set a state machine as a target for a CloudWatch Events rule that starts the execution of a state machine every 5 minutes.

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

Topics

- Step 1: Creating a State Machine (p. 27)
- Step 2: Creating a CloudWatch Events Rule (p. 27)

Step 1: Creating a State Machine

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

- To create a basic state machine, use the Getting Started (p. 3) tutorial.
- If you already have a state machine, proceed to the next step.

Step 2: Creating a CloudWatch Events Rule

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

To create the rule

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

   The Step 1: Create rule page is displayed.

2. In the Event source section, select Schedule and type 5 for Fixed rate of.
3. In the **Targets** section, choose **Add target** and from the list choose **Step Functions state machine**.

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

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

6. Type a **Name** for your rule (for example, statemachine-event), choose **Enabled** for **State**, and then choose **Create rule**.
A new execution of your state machine starts every 5 minutes.

Creating a Step Functions API Using API Gateway

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

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

Topics

- Step 1: Creating an IAM Role for API Gateway (p. 30)
- Step 2: Creating your API Gateway API (p. 30)
- Step 3: Testing and Deploying the API Gateway API (p. 32)
Step 1: Creating an IAM Role for API Gateway

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

To create a role for API Gateway

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

The IAM role appears in the list of roles.

5. Choose the name of your role and note the Role ARN, for example:

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

To attach a policy to the IAM role

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

Step 2: Creating your API Gateway API

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

To create the API

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

To create a resource

2. On the New Child Resource page, type execution for Resource Name, and then choose Create Resource.

To create a POST Method

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

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

1. For **Integration Type**, choose **AWS Service**.
2. For **AWS Region**, choose a region from the list.
   
   **Note**
   
   For regions that currently support Step Functions, see the [Supported Regions](p. 1).
3. For **AWS Service**, choose **Step Functions** from the list.
4. For **HTTP Method**, choose **POST** from the list.
   
   **Note**
   
   All Step Functions API actions use the HTTP POST method.
5. For **Action Type**, choose **Use action name**.
6. For **Action**, type **StartExecution**.
7. For **Execution Role**, type the role ARN of the IAM role that you created earlier (p. 30), for example:

```bash
arn:aws:iam::123456789012:role/APIGatewayToStepFunctions
```
8. Choose **Save**.

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

---

**Step 3: Testing and Deploying the API Gateway API**

To test the communication between API Gateway and Step Functions

1. On the **/execution - POST** - **Method Execution** page, choose **Test**.
2. On the /execution - POST - Method Test page, copy the following request parameters into the Request Body section using the ARN of an existing state machine (or create a new state machine (p. 3)), and then choose Test.

```json
{
    "input": "{}
    "name": "MyExecution",
}
```

**Note**
For more information, see the StartExecution Request Syntax in the AWS Step Functions API Reference.

If you don't want to include the ARN of your state machine in the body of your API Gateway call, you can configure a body-mapping template, for example:

```json
{
    "input": "$util.escapeJavaScript($input.json('"'))",
}
```

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

```json
{
    "input": "$util.escapeJavaScript($input.json('"'))",
}
```

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

```json
{
    "startDate": 1486768956.878
}
```

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

**To deploy your API**

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

**To test your deployment**

1. On the Stages page of StartExecutionAPI, expand alpha, /, /execution, POST.
2. On the alpha - POST - /execution page, note the Invoke URL, for example:
Iterating a Loop Using Lambda

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

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

Before you begin, go through the Creating a Lambda State Machine (p. 8) tutorial to ensure you have created the necessary IAM role, and are familiar with using Lambda and Step Functions together.

Topics
- Step 1: Create a Lambda Function to Iterate a Count (p. 34)
- Step 2: Test the Lambda Function (p. 35)
- Step 3: Create a State Machine (p. 36)
- Step 4: Start a New Execution (p. 39)

Step 1: Create a Lambda Function to Iterate a Count

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

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

To create the Lambda function

1. Sign in to the Lambda console, and then choose Create function.
2. In the Create function section, choose Author from scratch.
3. In the Author from scratch section, configure your Lambda function, as follows:
   a. For Name, type Iterator.
   b. For Runtime, select Node.js 6.10.
Step 2: Test the Lambda Function

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

To test your Lambda function

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

```json
{
  "Comment": "Test my Iterator function",
  "iterator": {
    "count": 10,
    "index": 5,
    "step": 1
  }
}
```
Step 3: Create a State Machine

To create the state machine

1. Sign in to the Step Functions console, and then choose Create a state machine.
   
   **Important**
   
   Ensure that your state machine is under the same AWS account and region as the Lambda function you created earlier.

2. On the Create a state machine page, choose Author from scratch. For Give a name to your state machine, enter IterateCount.

   **Note**

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

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

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

3. Create or enter an IAM role.

   - To create a new IAM role for Step Functions, choose Create a role for me, and then choose I acknowledge that Step Functions will create an IAM role which allows access to my Lambda functions.
   - If you have previously created an IAM role for Step Functions (p. 112), choose I will provide an IAM role ARN and enter your existing IAM role ARN.
Note
If you delete the IAM role that Step Functions creates, Step Functions can't recreate it later. Similarly, if you modify the role (for example, by removing Step Functions from the principals in the IAM policy), Step Functions can't restore its original settings later.

4. The following code describes a state machine with the following states:
   - **ConfigureCount**: Sets the default values for count, index, and step.
     ```json
     "ConfigureCount": {
       "Type": "Pass",
       "Result": {
         "count": 10,
         "index": 0,
         "step": 1
       },
     }
     
   - **Iterator**: References your Lambda function you created earlier, passing in the values configured in **ConfigureCount**.
     ```json
     "Iterator": {
       "Type": "Task",
       "ResultPath": "$.iterator",
       "Next": "IsCountReached"
     },
     
   - **IsCountReached**: A choice state that will either run your sample work again or will go to **Done** based on a boolean returned from your **Iterator** Lambda function.
     ```json
     "IsCountReached": {
       "Type": "Choice",
       "Choices": [
         {
           "Variable": "$.iterator.continue",
           "BooleanEquals": true,
           "Next": "ExampleWork"
         }
       ],
       "Default": "Done"
     },
     
   - **ExampleWork**: A stub for the work you want to accomplish in your execution. In this example it is a **pass** state. In an actual implementation this would be a **task** state. See **Tasks** (p. 50).
   - **Done**: The end state of your execution.

In the **Code** pane, add the following state machine definition using the Amazon Resource Name of the Lambda function that you created earlier (p. 34).

```json
{
  "Comment": "Iterator State Machine Example",
  "StartAt": "ConfigureCount",
  "States": {
    "ConfigureCount": {
      "Type": "Pass",
      "Result": {
        "count": 10,
        "index": 0,
      },
    }
```

37
Be sure to update the Amazon Resource Name in the Iterator state above so that it references the Lambda you created earlier. For more information about the Amazon States Language, see State Machine Structure (p. 75).

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

This graph shows the logic expressed in the above state machine code.
Step 4: Start a New Execution

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

To start a new execution

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

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

3. Choose **Start Execution**.

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

4. (Optional) In the **Execution Details** section, choose the **Info** tab to view the **Execution Status** and the **Started** and **Closed** time stamps.

5. Once the count reaches the number configured in the `ConfigureCount` state in your state machine, the execution quits iterating and ends.

---

**Continue as a New Execution**

AWS Step Functions is designed to run workflows that have a finite duration and number of steps. Executions are limited to a duration of one year, and a maximum of 25,000 events (see Limits (p. 96)).
However, you can create a state machine that uses a Lambda function to start a new execution, before allowing the current execution to terminate. This enables you to have a state machine that can break large jobs into smaller workflows, or to have a state machine that runs indefinitely.

This tutorial builds on the concept of using an external Lambda function to modify your workflow, which was demonstrated in the Iterating a Loop Using Lambda (p. 34) tutorial. You'll use the same Lambda function (Iterator) to iterate a loop for a specific number of times. In addition, you'll create another Lambda function to start a new execution of your workflow, and to decrement a count each time it starts a new execution. By setting the number of executions in the input, this state machine will end and restart an execution a specified number of times.

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

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

<table>
<thead>
<tr>
<th>State</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigureCount</td>
<td>A Pass (p. 77) state that configures the count, index, and step values that are used by the Iterator Lambda function to step through iterations of work.</td>
</tr>
<tr>
<td>Iterator</td>
<td>A Task (p. 78) state that references the Iterator Lambda function.</td>
</tr>
<tr>
<td>IsCountReached</td>
<td>A Choice (p. 80) state that uses a Boolean value from the Iterator function to decide if the state machine should continue the example work, or move to the ShouldRestart choice state.</td>
</tr>
<tr>
<td>ExampleWork</td>
<td>In this example, ExampleWork is a Pass state that represents the Task state that would perform work in an actual implementation.</td>
</tr>
</tbody>
</table>
### Prerequisites

Before you begin, go through the Creating a Lambda State Machine (p. 8) tutorial to ensure you have created an initial IAM role, and that you are familiar with using Lambda and Step Functions together.

#### Topics

- Step 1: Create an Iterate Lambda Function to Iterate a Count (p. 42)
- Step 2: Create a Restart Lambda Function to Start a New Step Functions Execution (p. 44)
- Step 3: Create a State Machine (p. 45)
- Step 4: Update the IAM Policy (p. 47)
- Step 5: Run an Execution (p. 47)

---

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

**Note**  
If you have completed the Iterating a Loop Using Lambda (p. 34) tutorial, you can skip this step and use that Lambda function.

This section, and the Iterating a Loop Using Lambda (p. 34) tutorial, shows how you can use a Lambda function to track a count so that you can track the number of iterations of a loop in your state machine.

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

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

#### To create the Iterate Lambda function

1. Sign in to the [Lambda console](https://console.aws.amazon.com/lambda/), and then choose Create function.
2. In the Create function section, choose Author from scratch.
3. In the Author from scratch section, configure your Lambda function, as follows:
   a. For Name, type Iterator.
   b. For Runtime, select [Node.js 6.10](https://docs.aws.amazon.com/lambda/latest/dg/lambda-nodejs.html).
   c. For Role, select Choose an existing role.
   d. For Existing role, select the Lambda role that you created in the Creating a Lambda State Machine (p. 8) tutorial.
Note
If the IAM role that you created doesn’t appear in the list, the role might still need a few minutes to propagate to Lambda.

e. Choose Create function.

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

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

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

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

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

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

5. Choose Save.

**Test the Iterate Lambda Function**

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

**To test your Lambda function**

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

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

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

3. Choose Create.

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

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

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

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

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

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

2. In the Author from scratch section, configure your Lambda function, as follows:
   a. For Name, type Restart.
   b. For Runtime, select Node.js 6.10.
   c. For Role, select Choose an existing role.
   d. Under Existing role, select the role that includes the IAM policy you created previously.
   e. Choose Create function.

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

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

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

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

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

   exports.restart = function(event, context, callback) {
     let StateMachineArn = event.restart.StateMachineArn;
     event.restart.executionCount -= 1;
     event = JSON.stringify(event);

     let params = {
       input: event,
       stateMachineArn: StateMachineArn
     };

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

4. Choose **Save**.

## Step 3: Create a State Machine

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

**Example ShouldRestart Choice state**

This excerpt of your state machine shows the ShouldRestart Choice (p. 80) state. This state decides if you should restart the execution.

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

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

**Example Restart Task state**

This excerpt of your state machine shows the Restart Task (p. 78) state. This state uses the Lambda function you created earlier to restart the execution, and to decrement the count to track the remaining number of executions to start.

```
"Restart": {
  "Type": "Task",
  "Next": "Done"
},
```

1. In the Step Functions console, choose **Create a state machine**.
2. Select **Author from scratch**, and enter **ContinueAsNew** as your state machine name.
3. Under **IAM role for your state machine executions**, select the IAM role that you use for Lambda functions.
4. Paste the following into the **Code** pane.

**Example ContinueAsNew state machine**

```
{
  "Comment": "Continue-as-new State Machine Example",
  "StartAt": "ConfigureCount",
  "States": {
    "ConfigureCount": {
      "Type": "Pass",
      "Comment": "Configure Count State"
    },
    "ShouldRestart":{
      "Type": "Choice",
      "Choices": [
        {"Variable": ".\$.restart.executionCount", "NumericGreaterThan": 1, "Next": "Restart"}
      ],
    "Restart": {
      "Type": "Task",
      "Next": "Done"
    },
    "Done": {}
  }
}
```
5. Update the Resource string in the Restart and Iterator states to reference the respective Lambda functions you created earlier.


**Note**
Save the Amazon Resource Name of this state machine.
Step 4: Update the IAM Policy

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

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

**Note**

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

Step 5: Run an Execution

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

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

```json
{
    "restart": {
        "executionCount": 4
    }
}
```
3. Update the StateMachineArn field with the Amazon Resource Name for your ContinueAsNew state machine.

The Visual Workflow graph will display the first of the four executions. Before it completes, it will pass through the Restart state and start a new execution.
With this execution complete, you can go look at the next execution that's running. Select the **ContinueAsNew** link at the top to see the list of executions. You should see both the recently closed execution, and an ongoing execution that the **Restart** Lambda function kicked off.

Once all the executions are complete, you should see four successful executions in the list. The first execution started displays the name you chose, and subsequent executions have a generated name.
How Step Functions Works

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

Topics
- States (p. 49)
- Tasks (p. 50)
- Activities (p. 50)
- Transitions (p. 57)
- State Machine Data (p. 58)
- Executions (p. 61)
- Error Handling (p. 61)
- Read Consistency (p. 67)
- Templates (p. 68)
- Sample Projects (p. 68)

States

A finite state machine can express an algorithm as a number of states, their relationships, and their input and output. AWS Step Functions allows you to coordinate individual tasks by expressing your workflow as a finite state machine, written in the Amazon States Language. Individual states can make decisions based on their input, perform actions, and pass output to other states. In Step Functions you express your workflows in the Amazon States Language, and the Step Functions console provides a graphical representation of that state machine to help visualize your application logic.

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

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

States can perform a variety of functions in your state machine:
- Do some work in your state machine (a Task (p. 50) state).
- Make a choice between branches of execution (a Choice (p. 80) state)
- Stop an execution with a failure or success (a Fail (p. 85) or Succeed (p. 84) state)
- Simply pass its input to its output or inject some fixed data (a Pass (p. 77) state)
- Provide a delay for a certain amount of time or until a specified time/date (a Wait (p. 83) state)
- Begin parallel branches of execution (a Parallel (p. 85) state)

For example, here is a example state named HelloWorld which performs a Lambda function:

```
"HelloWorld": {
  "Type": "Task",
  "Next": "AfterHelloWorldState",
```

States share a number of common features:

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

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

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

For more information regarding the various states that you can define using Amazon States Language, see States (p. 76).

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

### Tasks

All work in your state machine is done by tasks. A task can be an activity or a Lambda function.

- An activity consists of program code that waits for an operator to perform an action or to provide input. You can host activities on Amazon EC2, on Amazon ECS, or even on mobile devices. Activities poll Step Functions using the `GetActivityTask` and `SendTaskSuccess`, `SendTaskFailure`, and `SendTaskHeartbeat` API actions.
- A Lambda function (p. 8) is a cloud-native task that runs on AWS Lambda. You can write Lambda functions in a variety of programming languages, using the AWS Management Console or by uploading code to Lambda.

Amazon States Language represents tasks by setting a state's type to Task and by providing the task with the ARN of the activity or Lambda function. For more information about specifying task types, see Task (p. 78) in Amazon States Language (p. 74).

To see a list of your tasks, use the Tasks page in the Step Functions console.

### Activities

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

**Topics**

- Creating an Activity (p. 51)
- Writing a Worker (p. 51)
- Example Activity Worker in Ruby (p. 51)
Creating an Activity

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

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

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

- Call `CreateActivity` with the activity name.
- Using the Step Functions console.

**Note**

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

Writing a Worker

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

```ruby
[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.

Example Activity Worker in Ruby

The following is an example activity worker written in Ruby. This provides an implementation based on best practices that can be used as a reference for your activity worker. The code implements a consumer-producer pattern with a configurable number of threads for pollers and activity workers. The poller threads are constantly long polling the activity task. Once an activity task is retrieved, it is passed through a bounded blocking queue for the activity thread to pick it up.

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

The following Ruby code is the main entry point and example usage for the Ruby activity worker that follows.
require_relative './lib/step_functions/activity'
credentials = Aws::SharedCredentials.new
 region = 'us-west-2'
activity_arn = 'ACTIVITY_ARN'

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

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

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>require_relative</td>
<td>Relative path to the activity worker code below.</td>
</tr>
<tr>
<td>region</td>
<td>AWS region of your activity.</td>
</tr>
<tr>
<td>workers_count</td>
<td>The number of threads for your activity worker. For most implementations, between 10 and 20 threads should be sufficient. The longer the activity takes to process, the more threads it may need. As an estimate, multiply the number of process activities per second by the 99th percentile activity processing latency, in seconds.</td>
</tr>
<tr>
<td>pollers_count</td>
<td>The number of threads for your pollers. Between 10 and 20 threads should be sufficient for most implementations.</td>
</tr>
<tr>
<td>heartbeat_delay</td>
<td>The delay in seconds between heartbeats.</td>
</tr>
<tr>
<td>input</td>
<td>Implementation logic of your activity.</td>
</tr>
</tbody>
</table>

The following is the Ruby activity worker used by the example above and referenced with ../lib/step_functions/activity.

```ruby
require 'set'
require 'json'
require 'thread'
require 'logger'
require 'aws-sdk'
module Validate
```
def self.positive(value)
    raise ArgumentError, 'Argument has to positive' if value <= 0
    value
end

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

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

def self.with_retries(options = {}, &block)
    retries = 0
    base_delay_seconds = options[:base_delay_seconds] || 2
    max_retries = options[:max_retries] || 3
    begin
        block.call
        rescue => e
            puts e
            if retries < max_retries
                retries += 1
                sleep base_delay_seconds**retries
                retry
            end
            raise RetryError, 'All retries of operation had failed'
        end
    end
end

class Activity
    def initialize(options = {})
        @states = Aws::States::Client.new(
            credentials: Validate.required(options[:credentials]),
            region: Validate.required(options[:region]),
            http_read_timeout: Validate.positive(options[:http_read_timeout] || 60)
        )
        @activity_arn = Validate.required(options[:activity_arn])
        @heartbeat_delay = Validate.positive(options[:heartbeat_delay] || 60)
        @queue_max = Validate.positive(options[:queue_max] || 5)
        @pollers_count = Validate.positive(options[:pollers_count] || 1)
        @workers_count = Validate.positive(options[:workers_count] || 1)
        @max_retry = Validate.positive(options[:max_retry] || 3)
        @logger = Logger.new(STDOUT)
    end

def start(&block)
    @sink = SizedQueue.new(@queue_max)
    @activities = Set.new
    start_heartbeat_worker(@activities)
    start_workers(@activities, block, @sink)
    start_pollers(@activities, @sink)
    wait
end

def queue_size
    return 0 if @sink.nil?
    @sink.size
end

def activities_count

```
return 0 if @activities.nil?
  @activities.size
end

private

def start_pollers(activities, sink)
  @pollers = Array.new(@pollers_count) do
    PollerWorker.new(
      states: @states,
      activity_arn: @activity_arn,
      sink: sink,
      activities: activities,
      max_retry: @max_retry
    )
  end
  @pollers.each(&:start)
end

def start_workers(activities, block, sink)
  @workers = Array.new(@workers_count) do
    ActivityWorker.new(
      states: @states,
      block: block,
      sink: sink,
      activities: activities,
      max_retry: @max_retry
    )
  end
  @workers.each(&:start)
end

def start_heartbeat_worker(activities)
  @heartbeat_worker = HeartbeatWorker.new(
    states: @states,
    activities: activities,
    heartbeat_delay: @heartbeat_delay,
    max_retry: @max_retry
  )
  @heartbeat_worker.start
end

def wait
  sleep
rescue Interrupt
  shutdown
ensure
  Thread.current.exit
end

def shutdown
  stop_workers(@pollers)
  wait_workers(@pollers)
  wait_activities_drained
  stop_workers(@workers)
  wait_activities_completed
  shutdown_workers(@workers)
  shutdown_worker(@heartbeat_worker)
end

def shutdown_workers(workers)
  workers.each do |worker|
    shutdown_worker(worker)
  end
end
def shutdown_worker(worker)
    worker.kill
end

def wait_workers(workers)
    workers.each(&:wait)
end

def wait_activities_drained
    wait_condition { @sink.empty? }
end

def wait_activities_completed
    wait_condition { @activities.empty? }
end

def wait_condition(&block)
    loop do
        break if block.call
        sleep(1)
    end
end

def stop_workers(workers)
    workers.each(&:stop)
end

class Worker
    def initialize
        @logger = Logger.new(STDOUT)
        @running = false
    end

    def run
        raise 'Method run hasn\'t been implemented'
    end

    def process
        loop do
            begin
                break unless @running
                run
            rescue => e
                puts e
                @logger.error('Unexpected error had occurred')
                @logger.error(e)
            end
        end
    end

    def start
        if @thread.nil?
            @running = true
            @thread = Thread.new do
                process
            end
        end
    end

    def stop
        @running = false
    end

    def kill
        return if @thread.nil?
        @thread.kill
        @thread = nil
    end
end
end

def wait
    @thread.join
end
end

class PollerWorker < Worker
def initialize(options = {})
    @states = options[:states]
    @activity_arn = options[:activity_arn]
    @sink = options[:sink]
    @activities = options[:activities]
    @max_retry = options[:max_retry]
    @logger = Logger.new(STDOUT)
end

def run
    activity_task = StepFunctions.with_retries(max_retry: @max_retry) do
        begin
            @states.get_activity_task(activity_arn: @activity_arn)
            rescue => e
                @logger.error('Failed to retrieve activity task')
                @logger.error(e)
        end
    end
    return if activity_task.nil? || activity_task.task_token.nil?
    @activities.add(activity_task.task_token)
    @sink.push(activity_task)
end
end

class ActivityWorker < Worker
def initialize(options = {})
    @states = options[:states]
    @block = options[:block]
    @sink = options[:sink]
    @activities = options[:activities]
    @max_retry = options[:max_retry]
    @logger = Logger.new(STDOUT)
end

def run
    activity_task = @sink.pop
    result = @block.call(JSON.parse(activity_task.input))
    send_task_success(activity_task, result)
    rescue => e
        send_task_failure(activity_task, e)
    ensure
        @activities.delete(activity_task.task_token) unless activity_task.nil?
end

def send_task_success(activity_task, result)
    StepFunctions.with_retries(max_retry: @max_retry) do
        begin
            @states.send_task_success(
                task_token: activity_task.task_token,
                output: JSON.dump(result)
            )
        rescue => e
            @logger.error('Failed to send task success')
            @logger.error(e)
        end
    end
end
def send_task_failure(activity_task, error)
    StepFunctions.with_retries do
        begin
            @states.send_task_failure(
                task_token: activity_task.task_token,
                cause: error.message
            )
            rescue => e
                @logger.error('Failed to send task failure')
                @logger.error(e)
            end
        end
    end
end

class HeartbeatWorker < Worker
    def initialize(options = {})
        @states = options[:states]
        @activities = options[:activities]
        @heartbeat_delay = options[:heartbeat_delay]
        @max_retry = options[:max_retry]
        @logger = Logger.new(STDOUT)
    end

    def run
        sleep(@heartbeat_delay)
        @activities.each do |token|
            send_heartbeat(token)
        end
    end

    def send_heartbeat(token)
        StepFunctions.with_retries(max_retry: @max_retry) do
            begin
                @states.send_task_heartbeat(token)
            rescue => e
                @logger.error('Failed to send heartbeat for activity')
                @logger.error(e)
            end
            rescue => e
                @logger.error('Failed to send heartbeat for activity')
                @logger.error(e)
            end
        end
    end
end

Transitions

When an execution of a state machine is launched, the system begins with the state referenced in the top-level StartAt field. This field (a string) must exactly match, including case, the name of one of the states.

After executing a state, AWS Step Functions uses the value of the Next field to determine the next state to advance to.

Next fields also specify state names as strings, and must match the name of a state specified in the state machine description exactly (case-sensitive).
For example, the following state includes a transition to `NextState`:

```
"SomeState" : {
    ...
    "Next" : "NextState"
}
```

Most states permit only a single transition rule via the `Next` field. However, certain flow-control states (for example, a `Choice` state) allow you to specify multiple transition rules, each with its own `Next` field. The Amazon States Language (p. 74) provides details about each of the state types you can specify, including information about how to specify transitions.

States can have multiple incoming transitions from other states.

The process repeats until it reaches a terminal state (a state with "Type": Succeed, "Type": Fail, or "End": true), or a runtime error occurs.

The following rules apply to states within a state machine:

- States can occur in any order within the enclosing block, but the order in which they're listed doesn't affect the order in which they're run, which is determined by the contents of the states themselves.
- Within a state machine, there can be only one state designated as the `start` state, which is designated by the value of the `StartAt` field in the top-level structure.
- Depending on your state machine logic—for example, if your state machine has multiple branches of execution—you may have more than one `end` state.
- If your state machine consists of only one state, it can be both the `start` state and the `end` state.

## State Machine Data

State Machine data takes the following forms:

- The initial input into a state machine
- Data passed between states
- The output from a state machine

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

**Topics**

- Data Format (p. 58)
- State Machine Input/Output (p. 59)
- State Input/Output (p. 59)

## Data Format

State machine data is represented by JSON text, so you can provide values using any data type supported by JSON:

**Note**

- Numbers in JSON text format conform to JavaScript semantics. These numbers typically correspond to double-precision IEEE-854 values.
- The following is valid JSON text: stand-alone, quote-delimited strings; objects; arrays; numbers; Boolean values; and `null`.
State Machine Input/Output

You can give AWS Step Functions initial input data by passing it to a `StartExecution` action when you start an execution, or by passing initial data using the Step Functions console. Initial data is passed to the state machine's `StartAt` state. If no input is provided, the default is an empty object (`{}`).

The output of the execution is returned by the last state (terminal). This output appears as JSON text in the execution's result. You can retrieve execution results from the execution history using external callers (for example, in the `DescribeExecution` action). You can view execution results on the Step Functions console.

State Input/Output

Each state's input consists of JSON text from the preceding state or, for the `StartAt` state, the input into the execution. Certain flow-control states echo their input to their output.

In the following example, the state machine adds two numbers together:

1. Define the Lambda function.

```javascript
function Add(input) {
  var numbers = JSON.parse(input).numbers;
  var total = numbers.reduce(
    function(previousValue, currentValue, index, array) {
      return previousValue + currentValue;
    });
  return JSON.stringify({ result: total });
}
```

2. Define the state machine.

```json
{
  "Comment": "An example that adds two numbers together.",
  "StartAt": "Add",
  "Version": "1.0",
  "TimeoutSeconds": 10,
  "States":
    {
      "Add": {
        "Type": "Task",
        "End": true
      }
    }
}
```

3. Start an execution with the following JSON text:

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

The `Add` state receives the JSON text and passes it to the Lambda function.

The Lambda function returns the result of the calculation to the state.

The state returns the following value in its output:

59
Because `Add` is also the final state in the state machine, this value is returned as the state machine's output.

If the final state returns no output, then the state machine returns an empty object ({}).

**Filters**

Some states, such as Task (p. 78), have `InputPath`, `ResultPath`, and `OutputPath` fields. The values of these fields are path (p. 87).

- The `InputPath` field selects a portion of the state's input to pass into the state's processing logic (an activity, Lambda function, or so on). If the field is `null`, an empty object ({}) is passed.
  
  **Note**
  If you omit the `InputPath` field, the entire state input ($) is selected by default.

- The `ResultPath` field selects a portion of the state's input to overwrite with, or add to, result data from the state's processing logic.

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

  **Note**
  If you omit the optional `ResultPath` field, it defaults to $, which overwrites the entire input. However, you can select a portion of the input using the `OutputPath` field before the input is sent as the state's output.

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

  **Note**
  If you omit the optional `OutputPath` field, it defaults to $, which selects the entire input (as modified by the `ResultPath` field), and sends the input as the state's output.

In the following example, a state outputs the sum of its input values.

1. Define the `InputPath`, `ResultPath`, and `OutputPath`.

   ```json
   "InputPath": "$\cdot numbers",
   "ResultPath": "$\cdot sum",
   "OutputPath": "$"
   ``

2. Add state input data.

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

   The state output data is as follows:

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

3. Adjust the `OutputPath` without changing the state input data.
"InputPath": ".numbers",
"ResultPath": ".sum",
"OutputPath": ".sum"

The state output data changes as follows:

```
{ 7 }
```

You can separate the names of data members in your state machine data from the functions that process the data using the InputPath and ResultPath fields.

## Executions

A state machine execution occurs when a Step Functions state machine runs and performs its tasks. Each Step Functions state machine can have multiple simultaneous executions which you can initiate from the Step Functions console, or using the AWS SDKs, the Step Functions API actions, or the AWS CLI. An execution receives JSON input and produces JSON output.

For more information about the different ways of working with Step Functions, see Development Options (p. 7). For more information about initiating an execution from the Step Functions console, see To start a new execution (p. 4).

**Note**

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

## Error Handling

Any state can encounter runtime errors. Errors can happen for various reasons:

- State machine definition issues (for example, no matching rule in a Choice state).
- Task failures (for example, an exception in a Lambda function).
- Transient issues (for example, network partition events).

By default, when a state reports an error, Step Functions causes the execution to fail entirely.

## Error Names

Step Functions identifies errors in Amazon States Language using case-sensitive strings, known as error names. Amazon States Language defines a set of built-in strings that name well-known errors, all beginning with the States. prefix.

**States.ALL**

A wildcard that matches any known error name.

**States.Timeout**

A Task state either ran longer than the TimeoutSeconds value, or failed to send a heartbeat for a period longer than the HeartbeatSeconds value.
States.TaskFailed

A Task state failed during the execution.

States.Permissions

A Task state failed because it had insufficient privileges to execute the specified code.

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

Note
Unhandled errors in Lambda are reported as Lambda.Unknown in the error output. These include out-of-memory errors, function timeouts, and hitting the concurrent Lambda invoke limit. You can match on Lambda.Unknown, States.ALL, or States.TaskFailed to handle these errors. For more information about Lambda.Handled and Unhandled errors, see FunctionError in the AWS Lambda Developer Guide.

Retrying After an Error

Task and Parallel states can have a field named Retry, whose value must be an array of objects known as retriers. An individual retrier represents a certain number of retries, usually at increasing time intervals.

Note
Retries are treated as state transitions. For information on how state transitions affect billing, see Step Functions Pricing.

A retrier contains the following fields:

ErrorEquals (Required)

A non-empty array of strings that match error names. When a state reports an error, Step Functions scans through the retriers. When the error name appears in this array, it implements the retry policy described in this retrier.

IntervalSeconds (Optional)

An integer that represents the number of seconds before the first retry attempt (1 by default).

MaxAttempts (Optional)

A positive integer that represents the maximum number of retry attempts (3 by default). If the error recurs more times than specified, retries cease and normal error handling resumes. A value of 0 specifies that the error or errors are never retried.

BackoffRate (Optional)

The multiplier by which the retry interval increases during each attempt (2.0 by default).

This example of a Retry makes 2 retry attempts after waiting for 3 and 4.5 seconds.

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

The reserved name States.ALL that appears in a Retriever's ErrorEquals field is a wildcard that matches any error name. It must appear alone in the ErrorEquals array and must appear in the last retrier in the Retry array.

This example of a Retry field retries any error except States.Timeout.
"Retry": [ {
  "ErrorEquals": [ "States.Timeout" ],
  "MaxAttempts": 0
}, {
  "ErrorEquals": [ "States.ALL" ]
} ]

Complex Retry Scenarios

A retriever's parameters apply across all visits to the retriever in the context of a single-state execution. Consider the following Task state:

```
"X": {
  "Type": "Task",
  "Next": "Y",
  "Retry": [ {
    "ErrorEquals": [ "ErrorA", "ErrorB" ],
    "IntervalSeconds": 1,
    "BackoffRate": 2.0,
    "MaxAttempts": 2
  }, {
    "ErrorEquals": [ "ErrorC" ],
    "IntervalSeconds": 5
  } ],
  "Catch": [ {
    "ErrorEquals": [ "States.ALL" ],
    "Next": "Z"
  } ]
}
```

This task fails five times in succession, outputting these error names: ErrorA, ErrorB, ErrorC, ErrorB, and ErrorB. The following occurs as a result:

- The first two errors match the first retriever and cause waits of 1 and 2 seconds.
- The third error matches the second retriever and causes a wait of 5 seconds.
- The fourth error matches the first retriever and causes a wait of 4 seconds.
- The fifth error also matches the first retriever. However, it has already reached its limit of two retries (MaxAttempts) for that particular error (ErrorB), so it fails and execution is redirected to the Z state via the Catch field.

Fallback States

Task and Parallel states can have a field named Catch. This field's value must be an array of objects, known as catchers.

A catcher contains the following fields:

**ErrorEquals (Required)**

A non-empty array of Strings that match error names, specified exactly as they are with the retriever field of the same name.

**Next (Required)**

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

**ResultPath (Optional)**

A path (p. 87) that determines what input is sent to the state specified in the Next field.
When a state reports an error and either there is no Retry field, or if retries fail to resolve the error, Step Functions scans through the catchers in the order listed in the array. When the error name appears in the value of a catcher’s ErrorEquals field, the state machine transitions to the state named in the Next field.

The reserved name States.ALL that appears in a catcher’s ErrorEquals field is a wildcard that matches any error name. It must appear alone in the ErrorEquals array and must appear in the last catcher in the Catch array.

The following example of a Catch field transitions to the state named RecoveryState when a Lambda function outputs an unhandled Java exception. Otherwise, the field transitions to the EndState state:

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

**Note**

Each catcher can specify multiple errors to handle.

### Error Output

When Step Functions transitions to the state specified in a catch name, the object usually contains the field Cause. This field’s value is a human-readable description of the error. This object is known as the error output.

In this example, the first catcher contains a ResultPath field. This works similarly to a ResultPath field in a state’s top level, resulting in two possibilities:

- It takes the results of executing the state and overwrites a portion of the state’s input (or all of the state’s input).
- It takes the results and adds them to the input. In the case of an error handled by a catcher, the result of executing the state is the error output.

Thus, in this example, for the first catcher the error output is added to the input as a field named error-info (if there isn’t already a field with this name in the input). Then, the entire input is sent to RecoveryState. For the second catcher, the error output overwrites the input and only the error output is sent to EndState.

**Note**

If you don’t specify the ResultPath field, it defaults to $, which selects and overwrites the entire input.

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

### Examples Using Retry and Using Catch

The state machines defined in the following examples assume the existence of two Lambda functions: one that always fails and one that waits long enough to allow a timeout defined in the state machine to occur.

This is a definition of a Lambda function that always fails, returning the message error. In the state machine examples that follow, this Lambda function is named FailFunction.
exports.handler = (event, context, callback) => {
  callback("error");
};

This is a definition of a Lambda function that sleeps for 10 seconds. In the state machine examples that follow, this Lambda function is named `sleep10`.

**Note**
When you create this Lambda function in the Lambda console, remember to change the **Timeout** value in the **Advanced settings** section from 3 seconds (default) to 11 seconds.

exports.handler = (event, context, callback) => {
  setTimeout(function(){
    }, 11000);
};

### Handling a Failure Using Retry

This state machine uses a **Retry** field to retry a function that fails and outputs the error name `HandledError`. The function is retried twice with an exponential backoff between retries.

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Retry": [{
        "ErrorEquals": ["HandledError"],
        "IntervalSeconds": 1,
        "MaxAttempts": 2,
        "BackoffRate": 2.0
      }],
      "End": true
    }
  }
}
```

This variant uses the predefined error code `States.TaskFailed`, which matches any error that a Lambda function outputs.

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Retry": [{
        "ErrorEquals": ["States.TaskFailed"],
        "IntervalSeconds": 1,
        "MaxAttempts": 2,
        "BackoffRate": 2.0
      }],
      "End": true
    }
  }
}
```
Handling a Failure Using Catch

This example uses a Catch field. When a Lambda function outputs an error, the error is caught and the state machine transitions to the fallback state.

```json
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Catch": [{
        "ErrorEquals": ["HandledError"],
        "Next": "fallback"
      }],
      "End": true
    },
    "fallback": {
      "Type": "Pass",
      "Result": "Hello, AWS Step Functions!",
      "End": true
    }
  }
}
```

This variant uses the predefined error code States.TaskFailed, which matches any error that a Lambda function outputs.

```json
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Catch": [{
        "ErrorEquals": ["States.TaskFailed"],
        "Next": "fallback"
      }],
      "End": true
    },
    "fallback": {
      "Type": "Pass",
      "Result": "Hello, AWS Step Functions!",
      "End": true
    }
  }
}
```

Handling a Timeout Using Retry

This state machine uses a Retry field to retry a function that times out. The function is retried twice with an exponential backoff between retries.

```json
{
```

66
"Comment": "A Hello World example of the Amazon States Language using an AWS Lambda function",
"StartAt": "HelloWorld",
"States": {
  "HelloWorld": {
    "Type": "Task",
    "TimeoutSeconds": 2,
    "Retry": [ {
      "ErrorEquals": ["States.Timeout"],
      "IntervalSeconds": 1,
      "MaxAttempts": 2,
      "BackoffRate": 2.0
    } ],
    "End": true
  }
}

Handling a Timeout Using Catch

This example uses a Catch field. When a timeout occurs, the state machine transitions to the fallback state.

{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "TimeoutSeconds": 2,
      "Catch": [ {
        "ErrorEquals": ["States.Timeout"],
        "Next": "fallback"
      } ],
      "End": true
    },
    "fallback": {
      "Type": "Pass",
      "Result": "Hello, AWS Step Functions!",
      "End": true
    }
  }
}

Read Consistency

State machine updates in AWS Step Functions are eventually consistent. All StartExecution calls within a few seconds will use the updated definition and roleArn (the Amazon Resource Name for the IAM role). Executions started immediately after calling UpdateStateMachine might use the previous state machine definition and roleArn.

For more information, see:

- UpdateStateMachine — in the AWS Step Functions API Reference
- Step 3: (Optional) Update a State Machine (p. 5) — in the Getting Started (p. 3) section
Templates

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

Note
Choosing any of the templates overwrites the contents of the Code pane.

- **Hello world** – A state machine with a Pass state.
- **Wait state** – A state machine that demonstrates different ways of injecting a Wait state into a running state machine:
  - By waiting for a number of seconds.
  - By waiting for an absolute time (timestamp).
  - By specifying the Wait state's definition.
  - By using the state's input data.
- **Retry failure** – A state machine that retries a task after the task fails. This blueprint demonstrates how to handle multiple retries and various failure types.
- **Parallel** – A state machine that demonstrates how to execute two branches at the same time.
- **Catch failure** – A state machine that performs a different task after its primary task fails. This blueprint demonstrates how to call different tasks depending on the failure type.
- **Choice state** – A state machine that makes a choice: It either runs a Task state from a set of Task states or runs a Fail state after the initial state is complete.

Sample Projects

In the Step Functions console, you can choose one of the following state machine sample projects to automatically create the state machine Code, Visual Workflow, and all related AWS resources for the project. Each of the sample projects provisions a fully functional state machine, and creates the related resources for it to run. When you create a sample project, Step Functions uses AWS CloudFormation to create the related resources referenced by the state machine.

Topics
- Job Status Poller (p. 68)
- Task Timer (p. 71)

Job Status Poller

This sample project creates an AWS Batch job status poller. It implements an AWS Step Functions state machine that uses AWS Lambda to create a Wait state loop that checks on an AWS Batch job. This sample project creates and configures all resources so that your Step Functions workflow will submit an AWS Batch job, and will wait for that job to complete before ending successfully.

This sample project creates the state machine, two Lambda functions, an AWS Batch queue, and configures the related IAM permissions. For more information on the resources that are created with the Job Status Poller sample project, see:

- AWS CloudFormation User Guide
- AWS Batch User Guide
To create the **Job Status Poller** state machine and provision all resources:

1. Log in to the Step Functions console, and choose **Create a state machine**.
2. Select **Sample Projects** and choose **Job Status Poller**.

   The state machine **Code** and **Visual Workflow** are displayed.

   ![State Machine Diagram](image)

   **Note**
   The **Code** section in this state machine references the AWS resources that will be created for this sample project.

3. Choose **Create Sample Project**.

   The **Create Project Resources** window is displayed, listing the resources that will be created. For this sample project the resources include:

   - A SubmitJob Lambda function
   - A CheckJob Lambda function
   - A SampleJobQueue Batch Job Queue

   **Note**
   It can take up to 10 minutes as these resources and related IAM permissions are created. While the **Create Project Resources** window displays **Creating resources**, you can open the **Stack ID** link to see which resources are being provisioned.

Once complete, the **New execution** window is displayed, with example input similar to this:

```json
{
   "jobName": "my-job",
   "jobQueue": "arn:aws:batch:us-east-2:123456789012:job-queue/SampleJobQueue-4d9d696031e1449",
   "wait_time": 60
}
```
Starting an Execution

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

To start a new execution

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

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

3. Optionally, you can go to the newly-created state machine on the Step Functions Dashboard, select New execution, and enter the input code using the names or Amazon Resource Names of your newly created resources.

   For instance, the input for the above execution using only the resource names would be:

   ```json
   {
     "jobName": "my-job",
     "jobDefinition": "SampleJobDefinition-343f54b445d5312",
     "jobQueue": "SampleJobQueue-4d9d696031e1449",
     "wait_time": 60
   }
   ```

   **Note**
   wait_time instructs the Wait state to loop every sixty seconds.


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

5. (Optional) In the Execution Details section, choose Info to view the Execution Status and the Started and Closed timestamps.

6. To view the changing status of your AWS Batch job and the looping results of your execution, choose Output.
Task Timer

This sample project creates a task timer. It implements an AWS Step Functions state machine that implements a wait state, and uses a Lambda function to that sends an Amazon Simple Notification Service notification. A Wait state is a state type that waits for a trigger to perform a single unit of work.

This sample project creates the state machine, a Lambda function, an Amazon SNS topic, and configures the related IAM permissions. For more information on the resources that are created with the Task Timer sample project, see:

- AWS CloudFormation User Guide
- Amazon Simple Notification Service Developer Guide
- AWS Lambda Developer Guide
- IAM Getting Started Guide

To create the Task Timer state machine and provision all resources:

1. Log in to the Step Functions console, and choose Create a state machine.
2. Select Sample Projects and choose Task Timer.

The state machine Code and Visual Workflow are displayed.
3. Choose Create Sample Project.

The Create Project Resources window is displayed, listing the resources that will be created. For this sample project the resources include:

- A SendToSNS Lambda function
- A TaskTimerTopic Amazon SNS topic

Note
It can take up to 10 minutes as these resources and related IAM permissions are created. While the Create Project Resources window displays Creating resources, you can open the Stack ID: link to see which resources are being provisioned.

Once complete, the New execution window is displayed, with example input similar to this:

```json
{
  "topic": "arn:aws:sns:us-east-2:123456789012:StepFunctionsSample-TaskTimer-517b8680-e0ad-07cf-f65e6a5c638c0-SNSTopic-96RHT77RAKTS",
  "message": "HelloWorld",
  "timer_seconds": 10
}
```


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

5. (Optional) In the Execution Details section, choose Info to view the Execution Status and the Started and Closed timestamps.

6. To view the changing status of your AWS Batch job and the looping results of your execution, choose Output.
Amazon States Language

Amazon States Language is a JSON-based, structured language used to define your state machine, a collection of states (p. 49), that can do work (Task states), determine which states to transition to next (Choice states), stop an execution with an error (Fail states), and so on. For more information, see the Amazon States Language Specification and Statelint, a tool that validates Amazon States Language code.

To create a state machine on the Step Functions console using Amazon States Language, see Getting Started (p. 3).

Topics
- Example Amazon States Language Specification (p. 74)
- State Machine Structure (p. 75)
- States (p. 76)
- Input and Output Processing (p. 87)
- Errors (p. 90)

Example Amazon States Language Specification

```json
{
  "Comment": "An example of the Amazon States Language using a choice state.",
  "StartAt": "FirstState",
  "States": {
    "FirstState": {
      "Type": "Task",
      "Next": "ChoiceState"
    },
    "ChoiceState": {
      "Type": "Choice",
      "Choices": [
      {
        "Variable": ".foo",
        "NumericEquals": 1,
        "Next": "FirstMatchState"
      },
      {
        "Variable": ".foo",
        "NumericEquals": 2,
        "Next": "SecondMatchState"
      }
      ],
      "Default": "DefaultState"
    },
    "FirstMatchState": {
      "Type": "Task",
      "Next": "NextState"
    },
    "SecondMatchState": {
      "Type": "Task",
      "Next": "NextState"
    }
  }
}
```
State Machine Structure

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

- **Comment (Optional)**
  A human-readable description of the state machine.

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

- **TimeoutSeconds (Optional)**
  The maximum number of seconds an execution of the state machine may run; if it runs longer than the specified time, then the execution fails with an `States.Timeout` Error name (p. 90).

- **Version (Optional)**
  The version of Amazon States Language used in the state machine, default is "1.0".

- **States (Required)**
  This field's value is an object containing a comma-delimited set of states.

The `States` field contains a number of States (p. 76):

```
{
  "State1" : {
  },
  "State2" : {
  },
  ...
}
```

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

Here's an example:

```
{
  "Comment": "A Hello World example of the Amazon States Language using a Pass state",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Pass",
      "Result": "Hello World!"
    }
  }
}
```
When an execution of this state machine is launched, the system begins with the state referenced in the StartAt field ("HelloWorld"). If this state has an "End": true field, the execution stops and returns a result. Otherwise, the system looks for a "Next": field and continues with that state next. This process repeats until the system reaches a terminal state (a state with "Type": "Succeed", "Type": "Fail", or "End": true), or a runtime error occurs.

The following rules apply to states within a state machine:

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

States

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

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

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

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

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

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

Topics
- Common State Fields (p. 77)
- Pass (p. 77)
Common State Fields

**Type (Required)**

The state's type.

**Next**

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

**End**

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

**Comment (Optional)**

Holds a human-readable description of the state.

**InputPath (Optional)**

A path (p. 87) that selects a portion of the state's input to be passed to the state's task for processing. If omitted, it has the value `$` which designates the entire input. For more information, see Input and Output Processing (p. 87).

**OutputPath (Optional)**

A path (p. 87) that selects a portion of the state's input to be passed to the state's output. If omitted, it has the value `$` which designates the entire input. For more information, see Input and Output Processing (p. 87).

**Pass**

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

In addition to the common state fields (p. 77), Pass states allow the following fields:

**Result (Optional)**

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

**ResultPath (Optional)**

Specifies where (in the input) to place the "output" of the virtual task specified in Result. The input is further filtered as prescribed by the OutputPath field (if present) before being used as the state's output. For more information, see Input and Output Processing (p. 87).

Here is an example of a Pass state that injects some fixed data into the state machine, probably for testing purposes.
"No-op": {
  "Type": "Pass",
  "Result": {
    "x-datum": 0.381018,
    "y-datum": 622.2269926397355
  },
  "ResultPath": ".coords",
  "Next": "End"
}

Suppose the input to this state is:

{
  "georefOf": "Home"
}

Then the output would be:

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

Task

A Task state ("Type": "Task") represents a single unit of work performed by a state machine. In addition to the common state fields (p. 77), Task states have the following fields:

Resource (Required)

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

ResultPath (Optional)

Specifies where (in the input) to place the results of executing the task specified in Resource. The input is then filtered as prescribed by the OutputPath field (if present) before being used as the state's output. For more information, see path (p. 87).

Retry (Optional)

An array of objects, called Retriers, that define a retry policy in case the state encounters runtime errors. For more information, see Retrying After an Error (p. 90).

Catch (Optional)

An array of objects, called Catchers, that define a fallback state which is executed in case the state encounters runtime errors and its retry policy has been exhausted or is not defined. For more information, see Fallback States (p. 92).

TimeoutSeconds (Optional)

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

HeartbeatSeconds (Optional)

If more time than the specified seconds elapses between heartbeats from the task, then this state fails with an States.Timeout Error Name. Must be a positive, non-zero integer less than the
number of seconds specified in the `TimeoutSeconds` field. If not provided, the default value is 99999999.

A `Task` state must set either the `End` field to `true` if the state ends the execution, or must provide a state in the `Next` field that will be run upon completion of the `Task` state.

Here is an example:

```json
"ActivityState": {
  "Type": "Task",
  "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:

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

Where:

- `partition` is the AWS Step Functions partition to use, most commonly `aws`.
- `service` indicates the AWS service used to execute the task, and is either:
  - `states` for an activity (p. 80).
  - `lambda` for a Lambda function (p. 80).
- `region` is the AWS region in which the Step Functions activity/state machine type or Lambda function has been created.
- `account` is your AWS account id.
- `task_type` is the type of task to run. It will be one of the following values:
  - `activity` – an activity (p. 80).
  - `function` – a Lambda function (p. 80).
- `name` is the registered resource name (activity name or Lambda function name).

**Note**

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

### Task Types

The following task types are supported:

- `activity` (p. 80)
- `Lambda functions` (p. 80)
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 more information about these fields, see Specifying Resource ARNs in Tasks (p. 79).

**Note**

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

For more information about creating an activity and implementing workers, see Activities (p. 50).

**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 more information about these fields, see Specifying Resource ARNs in Tasks (p. 79).

For example:

```
"LambdaState": {
  "Type": "Task",
  "Next": "NextState"
}
```

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

**Choice**

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

In addition to the common state fields (p. 77), Choice states introduce the following additional fields:

**Choices** *(Required)*

An array of Choice Rules (p. 82) that determines which state the state machine transitions to next.

**Default** *(Optional, Recommended)*

The name of the state to transition to if none of the transitions in Choices is taken.

**Important**

Choice states do not support the End field. In addition, they use Next only inside their Choices field.
The following is an example of a Choice state and other states that it transitions to.

**Note**
You must specify the `.type` field. If the state input doesn't contain the `.type` field, the execution fails and an error is displayed in the execution history.

```json
"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",
    "Next": "NextState"
  },
  "ValueIsZero": {
    "Type": "Task",
    "Next": "NextState"
  },
  "ValueInTwenties": {
    "Type": "Task",
    "Next": "NextState"
  },
  "DefaultState": {
    "Type": "Fail",
    "Cause": "No Matches!"
  }
}
```

In this example the state machine starts with the following input value:

```json
{
}
```
Step Functions transitions to the ValueInTwenties state, based on the value field.

If there are no matches for the Choice state’s Choices, the state provided in the Default field runs instead. If the Default state isn’t specified, the execution fails with an error.

**Choice Rules**

A Choice state must have a Choices field whose value is a non-empty array, whose every element is an object called a Choice Rule. A Choice Rule contains the following:

- A **comparison** – Two fields that specify an input variable to compared, the type of comparison, and the value to compare the variable to.
- A **Next field** – The value of this field must match a state name in the state machine.

The following example checks whether the numerical value is equal to 1:

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

The following example checks whether the string is equal to MyString:

```json
{
    "Variable": "$.foo",
    "StringEquals": "MyString",
    "Next": "FirstMatchState"
}
```

The following example checks whether the string is greater than MyStringABC:

```json
{
    "Variable": "$.foo",
    "StringGreaterThan": "MyStringABC",
    "Next": "FirstMatchState"
}
```

The following example checks whether the timestamp is equal to 2001-01-01T12:00:00Z:

```json
{
    "Variable": "$.foo",
    "TimestampEquals": "2001-01-01T12:00:00Z",
    "Next": "FirstMatchState"
}
```

Step Functions examines each of the Choice Rules in the order listed in the Choices field 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:

- And
For each of these operators, the corresponding value must be of the appropriate type: string, number, Boolean, or timestamp. Step Functions doesn't attempt to match a numeric field to a string value. However, because timestamp fields are logically strings, it is possible that a field considered to be a timestamp can be matched by a StringEquals comparator.

**Note**
For interoperability, don't assume that numeric comparisons work with values outside the magnitude or precision that the IEEE 754-2008 binary64 data type represents. In particular, integers outside of the range $[-2^{53}+1, 2^{53}-1]$ might fail to compare in the expected way. Timestamps (for example, 2016-08-18T17:33:00Z) must conform to RFC3339 profile ISO 8601, with further restrictions:

- An uppercase T must separate the date and time portions.
- An uppercase Z must denote that a numeric time zone offset isn't present.

To understand the behavior of string comparisons, see the Java compareTo documentation. The values of the And and Or operators must be non-empty arrays of Choice Rules that must not themselves contain Next fields. Likewise, the value of a Not operator must be a single Choice Rule that must not contain Next fields.

You can create complex, nested Choice Rules using And, Not, and Or. However, the Next field can appear only in a top-level Choice Rule.

**Wait**

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

In addition to the common state fields (p. 77), Wait states have one of the following fields:

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

An absolute time to wait until before beginning the state specified in the `Next` field.

Timestamps must conform to the RFC3339 profile of ISO 8601, with the further restrictions that an uppercase `Z` must separate the date and time portions, and an uppercase `Z` must denote that a numeric time zone offset is not present, for example, `2016-08-18T17:33:00Z`.

**SecondsPath**

A time, in seconds, to wait before beginning the state specified in the `Next` field, specified using a path (p. 87) from the state's input data.

**TimestampPath**

An absolute time to wait until before beginning the state specified in the `Next` field, specified using a path (p. 87) 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:

```json
"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.

```json
"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:

```json
{
  "expirydate": "2016-03-14T01:59:00Z"
}
```

You can select the value of "expirydate" from the input using a reference path (p. 87) to select it from the input data:

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

**Succeed**

A `Succeed` state ("Type": "Succeed") stops an execution successfully. The `Succeed` state is a useful target for `Choice` state branches that don't do anything but stop the execution.

Because `Succeed` states are terminal states, they have no `Next` field, nor do they have need of an `End` field, for example:
Fail

A Fail state ("Type": "Fail") stops the execution of the state machine and marks it as a failure.

The Fail state only allows the use of Type and Comment fields from the set of common state fields (p. 77). In addition, the Fail state allows the following fields:

**Cause (Optional)**

Provides a custom failure string that can be used for operational or diagnostic purposes.

**Error (Optional)**

Provides an error name that can be used for error handling (Retry/Catch), operational or diagnostic purposes.

Because Fail states always exit the state machine, they have no Next field nor do they require an End field.

For example:

```json
"FailState": {
  "Type": "Fail",
  "Cause": "Invalid response.",
  "Error": "ErrorA"
}
```

Parallel

The Parallel state ("Type": "Parallel") can be used to create parallel branches of execution in your state machine.

In addition to the common state fields (p. 77), Parallel states introduce these additional fields:

**Branches (Required)**

An array of objects that specify state machines to execute in parallel. Each such state machine object must have fields named States and StartAt whose meanings are exactly like those in the top level of a state machine.

**ResultPath (Optional)**

Specifies where (in the input) to place the output of the branches. The input is then filtered as prescribed by the OutputPath field (if present) before being used as the state's output. For more information, see Input and Output Processing (p. 87).

**Retry (Optional)**

An array of objects, called Retriers that define a retry policy in case the state encounters runtime errors. For more information, see Retrying After an Error (p. 90).

**Catch (Optional)**

An array of objects, called Catchers that define a fallback state which is executed in case the state encounters runtime errors and its retry policy has been exhausted or is not defined. For more information, see Fallback States (p. 92).
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:

```json
"LookupCustomerInfo": {
  "Type": "Parallel",
  "Branches": [
    {
      "StartAt": "LookupAddress",
      "States": {
        "LookupAddress": {
          "Type": "Task",
          "End": true
        }
      }
    },
    {
      "StartAt": "LookupPhone",
      "States": {
        "LookupPhone": {
          "Type": "Task",
          "End": true
        }
      }
    }
  ],
  "Next": "NextState"
}
```

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

Each branch must be self-contained. A state in one branch of a Parallel state must not have a Next field that targets a field outside of that branch, nor can any other state outside the branch transition into that branch.

**Parallel State Output**

A Parallel state provides each branch with a copy of its own input data (subject to modification by the InputPath field). It generates output which is an array with one element for each branch containing the output from that branch. There is no requirement that all elements be of the same type. The output array can be inserted into the input data (and the whole sent as the Parallel state's output) by using a ResultPath field in the usual way (see Input and Output Processing (p. 87)).

Here is another example:

```json
"FunWithMath": {
  "Type": "Parallel",
  "Branches": [
    {
      "StartAt": "Add",
      "States": {
        "Add": {
          "Type": "Task",
          "End": true
        }
      }
    }
  ]
}
```
If the FunWithMath state was given the array [3, 2] as input, then both the Add and Subtract states receive that array as input. The output of Add would be 5, that of Subtract would be 1, and the output of the Parallel state would be an array:

```
[ 5, 1 ]
```

**Error Handling**

If any branch fails, due to either an unhandled error or by transitioning to a Fail state, the entire Parallel state is considered to have failed and all its branches are stopped. If the error is not handled by the Parallel state itself, Step Functions will stop the execution with an error.

---

**Input and Output Processing**

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

**Paths**

In Amazon States Language, a path is a string beginning with # that you can use to identify components within JSON text. Paths follow JsonPath syntax.

**Reference Paths**

A reference path is a path whose syntax is limited in such a way that it can identify only a single node in a JSON structure:

- You can access object fields using only dot (.) and square bracket ([ ]) notation.
- The operators @ .. , : ? * aren't supported.

For example, state input data contains the following values:

```
{
  "foo": 123,
  "bar": ["a", "b", "c"],
  "car": {
    "cdr": true
  }
}
```
In this case, the following reference paths would return:

```
$.foo => 123
$.bar => ["a", "b", "c"]
$.car.cdr => true
```

Certain states use paths and reference paths to control the flow of a state machine or configure a state's settings or options.

### Paths in InputPath, ResultPath, and OutputPath Fields

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

- For **InputPath** and **OutputPath**, you must use a path (p. 87) that follows the JsonPath syntax.
- For **ResultPath**, you must use a reference path (p. 87).

#### InputPath

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

**Note**

A path can yield a selection of values. Consider the following example:

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

If you apply the path `.a[0:2]`, the following is the result:

```
[ 1, 2 ]
```

#### ResultPath

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

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

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

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

  **Note**
  ResultPath field values must be reference paths (p. 87).

**OutputPath**

• If the OutputPath matches an item in the state's input, only that input item is selected. This input item becomes the state's output.
• If the OutputPath doesn't match an item in the state's input, an exception specifies an invalid path. For more information, see Errors (p. 90).
• If the OutputPath has the default value of $, this matches the entire input completely. In this case, the entire input is passed to the next state.

  **Note**
  For more information about the effect ResultPath has on the input for those states that allow it, see ResultPath (p. 88).
• If the OutputPath is null, JSON text representing an empty object {} is sent to the next state.

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

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

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

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

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

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

You can slightly modify the OutputPath:

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

As before, you use the following state input data:

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

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

**Note**
Retries are treated as state transitions. For information on how state transitions affect billing, see Step Functions Pricing.

A Retrier contains the following fields:

- **ErrorEquals (Required)**
  A non-empty array of Strings that match Error Names. When a state reports an error, Step Functions scans through the Retriers and, when the Error Name appears in this array, it implements the retry policy described in this Retrier.

- **IntervalSeconds (Optional)**
  An integer that represents the number of seconds before the first retry attempt (default 1).
MaxAttempts (Optional)

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

BackoffRate (Optional)

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

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

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

The reserved name States.ALL appearing in a Retrier's ErrorEquals field is a wildcard that matches any Error Name. It must appear alone in the ErrorEquals array and must appear in the last Retrier in the Retry array.

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

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

Complex Retry Scenarios

A Retrier's parameters apply across all visits to that Retrier in the context of a single state execution. This is best illustrated by an example; consider the following Task state:

```
"X": {  
  "Type": "Task",  
  "Next": "Y",  
  "Retry": [  
    {  
      "ErrorEquals": [ "ErrorA", "ErrorB" ],  
      "IntervalSeconds": 1,  
      "BackoffRate": 2.0,  
      "MaxAttempts": 2  
    },  
    {  
      "ErrorEquals": [ "ErrorC" ],  
      "IntervalSeconds": 5  
    }  
  ],  
  "Catch": [  
    
  ]
```
Suppose that this task fails five successive times, outputting Error Names "ErrorA", "ErrorB", "ErrorC", "ErrorB", and "ErrorB". The first two errors match the first retrier, and cause waits of one and two seconds. The third error matches the second retrier, and causes a wait of five seconds. The fourth error matches the first retrier and causes a wait of four seconds. The fifth error also matches the first retrier, but it has already reached its limit of two retries ("MaxAttempts") for that particular error ("ErrorB") so it fails and execution is redirected to the "Z" state via the "Catch" field.

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

Note
You can generate custom error names (such as ErrorA and ErrorB above) using either an activity (p. 80) or Lambda functions (p. 80). For more information, see Handling Error Conditions Using a State Machine (p. 22).

Fallback States

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

A Catcher contains the following fields:

**ErrorEquals (Required)**

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

**Next (Required)**

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

**ResultPath (Optional)**

A path (p. 87) which determines what is sent as input to the state specified by the Next field.

When a state reports an error and either there is no Retry field, or retries have failed to resolve the error, AWS Step Functions scans through the Catchers in the order listed in the array, and when the Error Name appears in the value of a Catcher's ErrorEquals field, the state machine transitions to the state named in the Next field.

The reserved name States.ALL appearing in a Catcher's ErrorEquals field is a wildcard that matches any Error Name. It must appear alone in the ErrorEquals array and must appear in the last Catcher in the Catch array.

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

```
"Catch": [  
  {  
    "ErrorEquals": [ "java.lang.Exception" ],  
    "ResultPath": "$.error-info",  
    "Next": "RecoveryState"  
  },  
  {  
    "ErrorEquals": [ "States.ALL" ],  
```
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 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.
Best Practices for Step Functions

The following best practices for implementing Step Functions workflows can help you optimize the performance of your implementations.

Topics
- Use Timeouts to Avoid Stuck Executions (p. 94)
- Use ARNs Instead of Passing Large Payloads (p. 94)
- Avoid Reaching the History Limit (p. 94)

Use Timeouts to Avoid Stuck Executions

By default, the Amazon States Language doesn't set timeouts in state machine definitions. Without an explicit timeout, Step Functions often relies solely on a response from an activity worker to know that a task is complete. If something goes wrong and TimeoutSeconds isn't specified, an execution is stuck waiting for a response that will never come.

To avoid this, specify a reasonable timeout limit when you create a task in your state machine. For example:

```
"ActivityState": {
    "Type": "Task",
    "TimeoutSeconds": 300,
    "HeartbeatSeconds": 60,
    "Next": "NextState"
  }
```

For more information, see Task (p. 78) in the Amazon States Language documentation.

Use ARNs Instead of Passing Large Payloads

Executions that pass large payloads of data between states can be terminated. If the data you are passing between states might grow to over 32 K, use Amazon Simple Storage Service (Amazon S3) to store the data, and pass the Amazon Resource Name instead of the raw data. Alternatively, adjust your implementation so that you pass smaller payloads in your executions.

For more information, see:
- Amazon Simple Storage Service Developer Guide
- Amazon Resource Names (ARNs)

Avoid Reaching the History Limit

AWS Step Functions has a hard limit of 25,000 entries in the execution history. To avoid reaching this limit for long-running executions, implement a pattern that uses an AWS Lambda function that can start a new execution of your state machine to split ongoing work across multiple workflow executions.
For more information, see the Continue as a New Execution (p. 40) tutorial.
AWS Step Functions places limits on the sizes of certain state machine parameters, such as the number of API actions that you can make during a certain time period or the number of state machines that you can define. Although these limits are designed to prevent a misconfigured state machine from consuming all of the resources of the system, they aren't hard limits.

**Note**
If a particular stage of your state machine execution or activity execution takes too long, you can configure a state machine timeout to cause a timeout event.

### General Limits

<table>
<thead>
<tr>
<th>Limit</th>
<th>Description</th>
</tr>
</thead>
</table>
| State machine name            | State machine names must be 1–80 characters in length, must be unique for your account and region, and must not contain any of the following:  
   - Whitespace  
   - Wildcard characters (? * )  
   - Bracket characters (< > { } [ ] )  
   - Special characters ( : ; , \ | ^ ~ $ # % & ` " )  
   - Control characters ( \u0000 - \u001f or \u007f - \u009f ). |

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

### Limits Related to Accounts

<table>
<thead>
<tr>
<th>Limit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of registered activities</td>
<td>10,000</td>
</tr>
</tbody>
</table>
Limits Related to State Machine Executions

<table>
<thead>
<tr>
<th>Limit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of registered state machines</td>
<td>10,000</td>
</tr>
<tr>
<td>Maximum number of API actions</td>
<td>Beyond infrequent spikes, applications may be throttled if they make a large number of API actions in a very short period of time.</td>
</tr>
<tr>
<td>Maximum request size</td>
<td>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.</td>
</tr>
</tbody>
</table>

Limits Related to Task Executions

<table>
<thead>
<tr>
<th>Limit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum task execution time</td>
<td>1 year (constrained by execution time limit)</td>
</tr>
<tr>
<td>Maximum time Step Functions keeps a task in the queue</td>
<td>1 year (constrained by execution time limit)</td>
</tr>
<tr>
<td>Maximum open activities</td>
<td>1,000 per execution. This limit includes both activities that have been scheduled and those being processed by workers.</td>
</tr>
<tr>
<td>Maximum input or result data size for a task, state, or execution</td>
<td>32,768 characters. This limit affects tasks (activity or Lambda function), state or execution result data, and input data when scheduling a task, entering a state, or starting an execution.</td>
</tr>
</tbody>
</table>
Limits Related to API Action Throttling

Some Step Functions API actions are throttled using a token bucket scheme to maintain service bandwidth.

**Note**
Throttling limits are per account, per region. AWS Step Functions may increase both the bucket size and refill rate at any time. Do not rely on these throttling rates to limit your costs.

<table>
<thead>
<tr>
<th>API Name</th>
<th>Bucket Size</th>
<th>Refill Rate per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateActivity</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>CreateStateMachine</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>DeleteActivity</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>DeleteStateMachine</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>DescribeActivity</td>
<td>200</td>
<td>1</td>
</tr>
<tr>
<td>DescribeExecution</td>
<td>200</td>
<td>2</td>
</tr>
<tr>
<td>DescribeStateMachine</td>
<td>200</td>
<td>1</td>
</tr>
<tr>
<td>DescribeStateMachineForExecution</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>GetActivityTask</td>
<td>1,000</td>
<td>25</td>
</tr>
<tr>
<td>GetExecutionHistory</td>
<td>250</td>
<td>5</td>
</tr>
<tr>
<td>ListActivities</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>ListExecutions</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>ListStateMachines</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>SendTaskFailure</td>
<td>1,000</td>
<td>25</td>
</tr>
<tr>
<td>SendTaskHeartbeat</td>
<td>1,000</td>
<td>25</td>
</tr>
<tr>
<td>SendTaskSuccess</td>
<td>1,000</td>
<td>25</td>
</tr>
<tr>
<td>StartExecution</td>
<td>500</td>
<td>25</td>
</tr>
<tr>
<td>StopExecution</td>
<td>500</td>
<td>25</td>
</tr>
<tr>
<td>UpdateStateMachine</td>
<td>200</td>
<td>1</td>
</tr>
</tbody>
</table>

Limits Related to State Throttling

Step Functions state transitions are throttled using a token bucket scheme to maintain service bandwidth.

**Note**
Throttling on the StateTransition service metric is reported as ExecutionThrottled in CloudWatch. For more information, see the [ExecutionThrottled CloudWatch metric](p. 101).
## Limits Related to State Throttling

<table>
<thead>
<tr>
<th>Service Metric</th>
<th>Bucket Size</th>
<th>Refill Rate per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateTransition</td>
<td>800</td>
<td>400</td>
</tr>
</tbody>
</table>
Monitoring and Logging

This section provides information about monitoring and logging Step Functions.

Topics

• Monitoring Step Functions Using CloudWatch (p. 100)
• Logging Step Functions using CloudTrail (p. 106)

Monitoring Step Functions Using CloudWatch

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS Step Functions and your AWS solutions. You should collect as much monitoring data from the AWS services that you use so that you can more easily debug any multi-point failures. Before you start monitoring Step Functions, you should create a monitoring plan that answers the following questions:

• What are your monitoring goals?
• What resources will you monitor?
• How often will you monitor these resources?
• What monitoring tools will you use?
• Who will perform the monitoring tasks?
• Who should be notified when something goes wrong?

The next step is to establish a baseline for normal Step Functions performance in your environment. To do this, measure performance at various times and under different load conditions. As you monitor Step Functions, you should consider storing historical monitoring data. Such data can give you a baseline to compare against current performance data, to identify normal performance patterns and performance anomalies, and to devise ways to address issues.

For example, with Step Functions, you can monitor how many activities or Lambda tasks fail due to a heartbeat timeout. When performance falls outside your established baseline, you might have to change your heartbeat interval.

To establish a baseline you should, at a minimum, monitor the following metrics:

• ActivitiesStarted
• ActivitiesTimedOut
• ExecutionsStarted
• ExecutionsTimedOut
• LambdaFunctionsStarted
• LambdaFunctionsTimedOut

The following sections describe metrics that Step Functions provides to CloudWatch. You can use these metrics to track your state machines and activities and to set alarms on threshold values. You can view metrics using the AWS Management Console.
Some of the Step Functions CloudWatch metrics are *time intervals*, always measured in milliseconds. These metrics generally correspond to stages of your execution for which you can set state machine, activity, and Lambda function timeouts, with descriptive names.

For example, the `ActivityRunTime` metric measures the time it takes for an activity to complete after it begins to execute. You can set a timeout value for the same time period.

In the CloudWatch console, you can get the best results if you choose **average** as the display statistic for time interval metrics.

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

In the CloudWatch console, you can get the best results if you choose **sum** as the display statistic for count metrics.

The following metrics are available for Step Functions state machines:

### Execution Metrics

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

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ExecutionTime</strong></td>
<td>The interval, in milliseconds, between the time the execution starts and the time it closes.</td>
</tr>
<tr>
<td><strong>ExecutionThrottled</strong></td>
<td>The number of <code>StateEntered</code> events and retries that have been throttled. This is related to StateTransition throttling. For more information, see <strong>Limits Related to State Throttling</strong> in the <strong>AWS Step Functions Developer Guide</strong>.</td>
</tr>
<tr>
<td><strong>ExecutionsAborted</strong></td>
<td>The number of aborted or terminated executions.</td>
</tr>
<tr>
<td><strong>ExecutionsFailed</strong></td>
<td>The number of failed executions.</td>
</tr>
<tr>
<td><strong>ExecutionsStarted</strong></td>
<td>The number of started executions.</td>
</tr>
<tr>
<td><strong>ExecutionsSucceeded</strong></td>
<td>The number of successfully completed executions.</td>
</tr>
</tbody>
</table>
### State Machine Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecutionsTimedOut</td>
<td>The number of executions that time out for any reason.</td>
</tr>
</tbody>
</table>

### Dimension for Step Functions Execution Metrics

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateMachineArn</td>
<td>The ARN of the state machine for the execution in question.</td>
</tr>
</tbody>
</table>

### Activity Metrics

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

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivityRunTime</td>
<td>The interval, in milliseconds, between the time the activity starts and the time it closes.</td>
</tr>
<tr>
<td>ActivityScheduleTime</td>
<td>The interval, in milliseconds, for which the activity stays in the schedule state.</td>
</tr>
<tr>
<td>ActivityTime</td>
<td>The interval, in milliseconds, between the time the activity is scheduled and the time it closes.</td>
</tr>
<tr>
<td>ActivitiesFailed</td>
<td>The number of failed activities.</td>
</tr>
<tr>
<td>ActivitiesHeartbeatTimedOut</td>
<td>The number of activities that time out due to a heartbeat timeout.</td>
</tr>
<tr>
<td>ActivitiesScheduled</td>
<td>The number of scheduled activities.</td>
</tr>
<tr>
<td>ActivitiesStarted</td>
<td>The number of started activities.</td>
</tr>
<tr>
<td>ActivitiesSucceeded</td>
<td>The number of successfully completed activities.</td>
</tr>
<tr>
<td>ActivitiesTimedOut</td>
<td>The number of activities that time out on close.</td>
</tr>
</tbody>
</table>

### Dimension for Step Functions Activity Metrics

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivityArn</td>
<td>The ARN of the activity.</td>
</tr>
</tbody>
</table>

### Lambda Function Metrics

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

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LambdaFunctionRunTime</td>
<td>The interval, in milliseconds, between the time the Lambda function starts and the time it closes.</td>
</tr>
</tbody>
</table>
### Metric Description

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LambdaFunctionScheduleTime</code></td>
<td>The interval, in milliseconds, for which the Lambda function stays in the schedule state.</td>
</tr>
<tr>
<td><code>LambdaFunctionTime</code></td>
<td>The interval, in milliseconds, between the time the Lambda function is scheduled and the time it closes.</td>
</tr>
<tr>
<td><code>LambdaFunctionsFailed</code></td>
<td>The number of failed Lambda functions.</td>
</tr>
<tr>
<td><code>LambdaFunctionsHeartbeatTimedOut</code></td>
<td>The number of Lambda functions that time out due to a heartbeat timeout.</td>
</tr>
<tr>
<td><code>LambdaFunctionsScheduled</code></td>
<td>The number of scheduled Lambda functions.</td>
</tr>
<tr>
<td><code>LambdaFunctionsStarted</code></td>
<td>The number of started Lambda functions.</td>
</tr>
<tr>
<td><code>LambdaFunctionsSucceeded</code></td>
<td>The number of successfully completed Lambda functions.</td>
</tr>
<tr>
<td><code>LambdaFunctionsTimedOut</code></td>
<td>The number of Lambda functions that time out on close.</td>
</tr>
</tbody>
</table>

### Dimension for Step Functions Lambda Function Metrics

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LambdaFunctionArn</code></td>
<td>The ARN of the Lambda function.</td>
</tr>
</tbody>
</table>

### Viewing Metrics for Step Functions

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

   ![CloudWatch Metrics](image)

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

   - **Execution Metrics**
• Activity Function Metrics
• Lambda Function Metrics

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

To sort your metrics by Metric Name or StateMachineArn, use the column headings.

To view graphs for a metric, choose the box next to the metric on the list. You can change the graph parameters using the time range controls above the graph view.

You can choose custom time ranges using relative or absolute values (specific days and times). You can also use the drop-down list to display values as lines, stacked areas, or numbers (values).

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

The metric’s details are displayed.

For more information about working with CloudWatch metrics, see Using Amazon CloudWatch Metrics in the Amazon CloudWatch User Guide.

Setting Alarms for Step Functions

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

To set an alarm on a metric

1. Open the AWS Management Console and navigate to CloudWatch.
2. Choose Metrics and on the All Metrics tab, choose States.
If you ran any executions recently, you will see up to three types of metrics:

- **Execution Metrics**
- **Activity Function Metrics**
- **Lambda Function Metrics**

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

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

5. Choose ▲ next to a metric on the list.

The **Create Alarm** dialog box is displayed.
6. Enter the values for the **Alarm threshold** and **Actions** and then choose **Create Alarm**.

For more information about setting and using CloudWatch alarms, see [Creating Amazon CloudWatch Alarms](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CreateNotifications.html) in the *Amazon CloudWatch User Guide*.

### Logging Step Functions using CloudTrail

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

To learn more about CloudTrail, including how to configure and enable it, see the AWS *CloudTrail User Guide*.

### Step Functions Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API actions 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:
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 actions, so they do not appear in any specific order.

CreateActivity

The following example shows a CloudTrail log entry that demonstrates the CreateActivity action:

```json
{
   "eventVersion": "1.04",
   "userIdentity": {
      "type": "IAMUser",
      "principalId": "AIDAJYDLDBVI4EXAMPLE",
      "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": {
```

Every log entry contains information about who generated the request. The user identity information in the log helps you determine the following:

- Whether the request was made with root or IAM user credentials
- Whether the request was made with temporary security credentials for a role or federated user
- Whether the request was made by another AWS service

For more information, see the `userIdentity` element in the *AWS CloudTrail User Guide*.

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

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

You can also aggregate Step Functions log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket. For more information, see Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts.
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": {
    "creationDate": "Oct 28, 2016 1:18:07 AM"
  },
  "requestID": "3da6370c-9cac-11e6-aed5-5b57d226e9ef",
  "eventID": "84a0441dcfa06-4691-a60a-aab9e46d689c",
  "eventType": "AwsApiCall",
  "recipientAccountIds": ["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": "2016-10-28T01:18:07Z",
  "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": {
    "creationDate": "Oct 28, 2016 1:18:07 AM"
  },
  "requestID": "3da6370c-9cac-11e6-aed5-5b57d226e9ef",
  "eventID": "84a0441dcfa06-4691-a60a-aab9e46d689c",
  "eventType": "AwsApiCall",
  "recipientAccountIds": ["123456789012"]
}
```
DeleteStateMachine

The following example shows a CloudTrail log entry that demonstrates the DeleteStateMachine action:

```json
{
  "eventVersion": "1.04",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDAJAKS5MNKNAEXAMPLE",
    "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",
  "requestParameters": null,
  "responseElements": null,
  "requestID": "2cf23f3c-9cac-11e6-aed5-5b57d226e9ef",
  "eventName": "DeleteStateMachine",
  "recipientAccountId": "123456789012"
}
```

StartExecution

The following example shows a CloudTrail log entry that demonstrates the StartExecution action:

```json
{
  "eventVersion": "1.04",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDAJYDLDBVBI4EXAMPLE",
    "arn": "arn:aws:iam::123456789012:user/test-user",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI5Z75K4ZEXAMPLE",
    "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": {
  },
  "responseElements": null,
  "requestID": "490374ea-9cac-11e6-aed5-5b57d226e9ef",
  "eventTime": "2016-10-28T01:18:27Z",
  "eventSource": "states.amazonaws.com",
  "eventName": "DeleteStateMachine",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "10.61.88.189",
  "userAgent": "Coral/Netty",
  "requestParameters": null,
  "responseElements": null,
  "requestID": "e5eb9a3d-13bc-4fa1-9531-232d1914d263",
  "eventName": "StartExecution",
  "recipientAccountId": "123456789012"
}
```
StopExecution

The following example shows a CloudTrail log entry that demonstrates the StopExecution action:

```json
```
Security

This section provides information about Step Functions security and authentication.

Topics

- Authentication (p. 111)
- Creating IAM Roles for AWS Step Functions (p. 112)
- Creating Granular IAM Permissions for Non-Admin Users (p. 112)

Step Functions uses IAM to control access to other AWS services and resources. For an overview of how IAM works, see Overview of Access Management in the IAM User Guide. For an overview of security credentials, see AWS Security Credentials in the Amazon Web Services General Reference.

Authentication

You can access AWS as any of the following types of identities:

- **AWS account root user** – When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account root user and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

- **IAM user** – An IAM user is an identity within your AWS account that has specific custom permissions (for example, permissions to create a state machine in Step Functions). You can use an IAM user name and password to sign in to secure AWS webpages like the AWS Management Console, AWS Discussion Forums, or the AWS Support Center.

  In addition to a user name and password, you can also generate access keys for each user. You can use these keys when you access AWS services programmatically, either through one of the several SDKs or by using the AWS Command Line Interface (CLI). The SDK and CLI tools use the access keys to cryptographically sign your request. If you don't use AWS tools, you must sign the request yourself. Step Functions supports Signature Version 4, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 Signing Process in the AWS General Reference.

- **IAM role** – An IAM role is an IAM identity that you can create in your account that has specific permissions. It is similar to an IAM user, but it is not associated with a specific person. An IAM role enables you to obtain temporary access keys that can be used to access AWS services and resources. IAM roles with temporary credentials are useful in the following situations:

  - **Federated user access** – Instead of creating an IAM user, you can use existing user identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as federated users. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated Users and Roles in the IAM User Guide.
• **AWS service access** – You can use an IAM role in your account to grant an AWS service permissions to access your account’s resources. For example, you can create a role that allows Amazon Redshift to access an Amazon S3 bucket on your behalf and then load data from that bucket into an Amazon Redshift cluster. For more information, see Creating a Role to Delegate Permissions to an AWS Service in the IAM User Guide.

• **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM Role to Grant Permissions to Applications Running on Amazon EC2 Instances in the IAM User Guide.

### Creating IAM Roles for AWS Step Functions

AWS Step Functions can execute code and access AWS resources (such as invoking an AWS Lambda function). To maintain security, you must grant Step Functions access to those resources by using an IAM role.

The Tutorials (p. 7) in this guide enable you to take advantage of automatically generated IAM roles that are valid for the region in which you create the state machine. To create your own IAM role for a state machine, follow the steps in this section.

**Create a Role for Step Functions**

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

**To create a role for Step Functions**

1. Sign in to the IAM console, and then choose Roles, Create role.
2. On the Select type of trusted entity page, under AWS service, select Step Functions from the list and then choose Next: Permissions.
3. On the Attached permissions policy page, choose Next: Review.
4. On the Review page, type StepFunctionsLambdaRole for Role Name, and then choose Create role.

The IAM role appears in the list of roles.

For more information about IAM permissions and policies, see Access Management in the IAM User Guide.

### Creating Granular IAM Permissions for Non-Admin Users

The default managed policies in IAM, such as ReadOnly, don’t fully cover all types of Step Functions permissions. This section describes these different types of permissions and provides some example configurations.
AWS Step Functions has four categories of permissions. Depending on what access you want to provide to a user, you can control access by using permissions in these categories.

Service-Level Permissions (p. 113)

Apply to components of the API that do not act on a specific resource.

State Machine-Level Permissions (p. 113)

Apply to all API components that act on a specific state machine.

Execution-Level Permissions (p. 114)

Apply to all API components that act on a specific execution.

Activity-Level Permissions (p. 114)

Apply to all API components that act on a specific activity or on a particular instance of an activity.

Service-Level Permissions

This permission level applies to all API actions that do not act on a specific resource. These include `CreateStateMachine`, `CreateActivity`, `ListStateMachines`, and `ListActivities`.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:ListStateMachines",
        "states:ListActivities",
        "states:CreateStateMachine",
        "states:CreateActivity"
      ],
      "Resource": [
        "arn:aws:states::*:*::*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::*:role/my-execution-role"
      ]
    }
  ]
}
```

State Machine-Level Permissions

This permission level applies to all API actions that act on a specific state machine. These API require the ARN of the state machine as part of the request, such as `DeleteStateMachine`, `DescribeStateMachine`, `StartExecution`, and `ListExecutions`.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DeleteStateMachine",
        "states:DescribeStateMachine",
        "states:StartExecution",
        "states:ListExecutions"
      ],
      "Resource": [
        "arn:aws:states:my-state-machine:*:*"
      ]
    }
  ]
}
```
Execution-Level Permissions

This permission level applies to all the API actions that act on a specific execution. These API operations require the ARN of the execution as part of the request, such as DescribeExecution, GetExecutionHistory, and StopExecution.

Activity-Level Permissions

This permission level applies to all the API actions that act on a specific activity or on a particular instance of it. These API operations require the ARN of the activity or the token of the instance as part of the request, such as DeleteActivity, DescribeActivity, GetActivityTask, SendTaskSuccess, SendTaskFailure, and SendTaskHeartbeat.
"arn:aws:states:*:*:activity:ActivityPrefix"
## Related Step Functions Resources

The following table lists related resources that you might find useful as you work with this service.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS Step Functions API Reference</td>
<td>Descriptions of API actions, parameters, and data types and a list of errors that the service returns.</td>
</tr>
<tr>
<td>AWS Step Functions Command Line Reference</td>
<td>Descriptions of the AWS CLI commands that you can use to work with AWS Step Functions.</td>
</tr>
<tr>
<td>Product information for Step Functions</td>
<td>The primary web page for information about Step Functions.</td>
</tr>
<tr>
<td>Discussion Forums</td>
<td>A community-based forum for developers to discuss technical questions related to Step Functions and other AWS services.</td>
</tr>
<tr>
<td>AWS Premium Support Information</td>
<td>The primary web page for information about AWS Premium Support, a one-on-one, fast-response support channel to help you build and run applications on AWS infrastructure services.</td>
</tr>
</tbody>
</table>
# Document History

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

**Latest documentation update:** May 5, 2018

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>New feature</td>
<td>AWS Step Functions is now available the US West (N. California) and Asia Pacific (Seoul) regions. See Supported Regions (p. 1) for a list of supported regions.</td>
<td>May 5, 2018</td>
</tr>
<tr>
<td>Update</td>
<td>Updated procedures and images to match changes to the interface.</td>
<td>April 25, 2018</td>
</tr>
<tr>
<td>Update</td>
<td>Added a new tutorial that shows how to start a new execution to continue your work. See <em>Continue as a New Execution</em> (p. 40). This tutorial describes a design pattern that can help avoid some service limitations. See, Avoid Reaching the History Limit (p. 94).</td>
<td>April 19, 2018</td>
</tr>
<tr>
<td>Update</td>
<td>Improved introduction to states documentation by adding conceptual information about state machines. See States (p. 49).</td>
<td>March 9, 2018</td>
</tr>
<tr>
<td>Update</td>
<td>In addition to HTML, PDF, and Kindle, the AWS Step Functions Developer Guide is available on GitHub. To leave feedback, choose the GitHub icon in the upper right-hand corner.</td>
<td>March 2, 2018</td>
</tr>
<tr>
<td>Update</td>
<td>Added a topic describing other resources relating to Step Functions.</td>
<td>February 20, 2018</td>
</tr>
<tr>
<td>New feature</td>
<td>• When you create a new state machine, you must acknowledge that AWS Step Functions will create an IAM role which allows access to your Lambda functions.</td>
<td>February 19, 2018</td>
</tr>
<tr>
<td></td>
<td>• Updated the following tutorials to reflect the minor changes in the state machine creation workflow:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Getting Started (p. 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Lambda State Machine (p. 8)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating an Activity State Machine (p. 17)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handling Error Conditions Using a State Machine (p. 22)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Iterating a Loop Using Lambda (p. 34)</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Added a topic that describes an example activity worker written in Ruby. This implementation can be used to create a Ruby activity worker directly, or as a design pattern for creating an activity worker in another language.</td>
<td>February 6, 2018</td>
</tr>
<tr>
<td></td>
<td>See <em>Example Activity Worker in Ruby</em> (p. 51).</td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Update</td>
<td>Added a new tutorial describing a design pattern that uses a Lambda function to iterate a count.</td>
<td>January 31, 2018</td>
</tr>
<tr>
<td></td>
<td>See Creating a Lambda State Machine (p. 8).</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Updated content on IAM permissions to include DescribeStateMachineForExecution and UpdateStateMachine APIs.</td>
<td>January 26, 2018</td>
</tr>
<tr>
<td></td>
<td>See Creating Granular IAM Permissions for Non-Admin Users (p. 112).</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Added newly available regions: Canada (Central), Asia Pacific (Singapore).</td>
<td>January 25, 2018</td>
</tr>
<tr>
<td></td>
<td>See Supported Regions (p. 1).</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Updated tutorials and procedures to reflect that IAM allows you to select Step Functions as a role.</td>
<td>January 24, 2018</td>
</tr>
<tr>
<td>Update</td>
<td>Added a new Best Practices topic that suggests not passing large payloads between states.</td>
<td>January 23, 2018</td>
</tr>
<tr>
<td></td>
<td>See Use ARNs Instead of Passing Large Payloads (p. 94).</td>
<td></td>
</tr>
<tr>
<td>New Feature</td>
<td>Corrected procedures to match updated interface for creating a state machine:</td>
<td>January 17, 2018</td>
</tr>
<tr>
<td></td>
<td>• Getting Started (p. 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Lambda State Machine (p. 8)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating an Activity State Machine (p. 17)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handling Error Conditions Using a State Machine (p. 22)</td>
<td></td>
</tr>
<tr>
<td>New Feature</td>
<td>You can use Sample Projects to quickly provision state machines and all related AWS resources.</td>
<td>January 11, 2018</td>
</tr>
<tr>
<td></td>
<td>See Sample Projects (p. 68),</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Available sample projects include:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Job Status Poller (p. 68)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Task Timer (p. 71)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>These sample projects and related documentation replace tutorials that described implementing the same functionality.</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Added a Best Practices section that includes information on avoiding stuck executions. See Best Practices for Step Functions (p. 94).</td>
<td>January 5, 2018</td>
</tr>
<tr>
<td>Update</td>
<td>Added a note on how retries can affect pricing:</td>
<td>December 8, 2017</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Retries are treated as state transitions. For information on how state transitions affect billing, see Step Functions Pricing.</td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
</tbody>
</table>
| Update       | Added information related to resource names:  
**Note**  
Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.                                                                 | December 6, 2017    |
| Update       | Improved security overview information and added a topic on granular IAM permissions. See Security (p. 111) and Creating Granular IAM Permissions for Non-Admin Users (p. 112).                                                                                              | November 27, 2017   |
| New Feature  | You can update an existing state machine. See Update a State Machine (p. 5).                                                                                                                                                                                                     | November 15, 2017   |
| Update       | Added a note to clarify `Lambda.Unknown` errors and linked to the Lambda documentation in the following sections:  
- Error Names (p. 61)  
- To create the state machine (p. 24)  
**Note**  
Unhandled errors in Lambda are reported as `Lambda.Unknown` in the error output. These include out-of-memory errors, function timeouts, and hitting the concurrent Lambda invoke limit. You can match on `Lambda.Unknown`, `States.ALL`, or `States.TaskFailed` to handle these errors. For more information about Lambda Handled and Unhandled errors, see FunctionError in the AWS Lambda Developer Guide. | October 17, 2017    |
| Update       | Corrected and clarified IAM instructions and updated the screenshots in all tutorials (p. 7).                                                                                                                                                                                   | October 11, 2017    |
| Update       | Added new screenshots for state machine execution results to reflect changes in the Step Functions console. Rewrote the Lambda instructions in the following tutorials to reflect changes in the Lambda console:  
- Creating a Lambda State Machine (p. 8)  
- Creating a Job Status Poller  
- Creating a Task Timer  
- Handling Error Conditions Using a State Machine (p. 22)  
- Corrected and clarified information about creating state machines in the following sections:  
  - Getting Started (p. 3)  
  - Creating an Activity State Machine (p. 17) | October 6, 2017     |
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date Changed</th>
</tr>
</thead>
</table>
| Update | Rewrote the IAM instructions in the following sections to reflect changes in the IAM console:  
  - Creating IAM Roles for AWS Step Functions (p. 112)  
  - Creating a Lambda State Machine (p. 8)  
  - Creating a Job Status Poller  
  - Creating a Task Timer  
  - Handling Error Conditions Using a State Machine (p. 22)  
  - Creating a Step Functions API Using API Gateway (p. 29) | October 5, 2017 |
| Update | Rewrote the State Machine Data (p. 58) section. | September 28, 2017 |
| New feature | The limits related to API action throttling (p. 98) are increased for all regions where Step Functions is available. | September 18, 2017 |
| Update |  
  - Corrected and clarified information about starting new executions in all tutorials.  
  - Corrected and clarified information in the Limits Related to Accounts (p. 96) section. | September 14, 2017 |
| Update | Corrected and clarified information in the Templates (p. 68) section. | September 1, 2017 |
| Update | Rewrote the following tutorials to reflect changes in the Lambda console:  
  - Creating a Lambda State Machine (p. 8)  
  - Handling Error Conditions Using a State Machine (p. 22)  
  - Creating a Job Status Poller | August 28, 2017 |
| New feature | Step Functions is available in EU (London). | August 23, 2017 |
| New feature | The visual workflows of state machines let you zoom in, zoom out, and center the graph. | August 21, 2017 |
| New feature | **Important**  
  An execution can't use the name of another execution for 90 days.  
  When you make multiple StartExecution calls with the same name, the new execution doesn't run and the following rules apply.  
  | August 18, 2017 |
| |  
  **Input Type** | **Execution State** |  
  Open | Closed |  
  Identical | Success | ExecutionAlreadyExists |  
  Different | ExecutionAlreadyExists |  
  For more information, see the name request parameter of the StartExecution API action in the AWS Step Functions API Reference. |  
<p>| Update | Added information about an alternative way of passing the state machine ARN to the Creating a Step Functions API Using API Gateway (p. 29) tutorial. | August 17, 2017 |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>Added the new <em>Creating a Job Status Poller</em> tutorial.</td>
<td>August 10, 2017</td>
</tr>
</tbody>
</table>
| New feature | • Step Functions emits the *ExecutionThrottled* CloudWatch metric. For more information, see *State Machine Metrics* (p. 101).  
• Added the *Limits Related to State Throttling* (p. 98) section.                                                                                           | August 3, 2017 |
| Update   | Updated the instructions in the *To create a role for API Gateway* (p. 30) section.                                                                                                                     | July 18, 2017 |
| Update   | Corrected and clarified information in the *Choice* (p. 80) section.                                                                                                                                       | June 23, 2017 |
| Update   | Added information about using resources under other AWS accounts to the following tutorials:  
• *Creating a Lambda State Machine* (p. 8)  
• *Creating a Lambda State Machine Using AWS CloudFormation* (p. 13)  
• *Creating an Activity State Machine* (p. 17)  
• *Handling Error Conditions Using a State Machine* (p. 22)                                                                                     | June 22, 2017 |
| Update   | Corrected and clarified information in the following sections:  
• *Getting Started* (p. 3)  
• *Handling Error Conditions Using a State Machine* (p. 22)  
• *States* (p. 76)  
• *Error Handling* (p. 61)                                                                                                                                 | June 21, 2017 |
<p>| Update   | Rewrote all tutorials to match the Step Functions console refresh.                                                                                                                                              | June 12, 2017 |
| New feature | Step Functions is available in Asia Pacific (Sydney).                                                                                                                                                     | June 8, 2017 |
| Update   | Restructured the <em>Amazon States Language</em> (p. 74) section.                                                                                                                                                 | June 7, 2017 |
| Update   | Corrected and clarified information in the <em>Creating an Activity State Machine</em> (p. 17) section.                                                                                                              | June 6, 2017 |
| Update   | Corrected the code examples in the <em>Examples Using Retry and Using Catch</em> (p. 64) section.                                                                                                                    | June 5, 2017 |
| Update   | Restructured this guide using AWS documentation standards.                                                                                                                                                   | May 31, 2017  |
| Update   | Corrected and clarified information in the <em>Parallel</em> (p. 85) section.                                                                                                                                    | May 25, 2017  |
| Update   | Merged the <em>Paths and Filters</em> sections into the <em>Input and Output Processing</em> (p. 87) section.                                                                                                              | May 24, 2017  |
| Update   | Corrected and clarified information in the <em>Templates</em> (p. 68) section.                                                                                                                                   | May 16, 2017  |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>Corrected and clarified information in the Monitoring Step Functions Using CloudWatch (p. 100) section.</td>
<td>May 15, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Updated the GreeterActivities.java worker code in the Creating an Activity State Machine (p. 17) tutorial.</td>
<td>May 9, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Added an introductory video to the What Is AWS Step Functions? (p. 1) section.</td>
<td>April 19, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Corrected and clarified information in the following tutorials:</td>
<td>April 19, 2017</td>
</tr>
<tr>
<td></td>
<td>- Getting Started (p. 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Creating a Lambda State Machine (p. 8)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Creating an Activity State Machine (p. 17)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Handling Error Conditions Using a State Machine (p. 22)</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Added information about Lambda templates to the Creating a Lambda State Machine (p. 8) and Handling Error Conditions Using a State Machine (p. 22) tutorial.</td>
<td>April 6, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Changed the &quot;Maximum input or result data size&quot; limit to &quot;Maximum input or result data size for a task, state, or execution&quot; (32,768 characters). For more information, see Limits Related to Task Executions (p. 97).</td>
<td>March 31, 2017</td>
</tr>
<tr>
<td>New feature</td>
<td>Step Functions supports executing state machines by setting Step Functions as Amazon CloudWatch Events targets.</td>
<td>March 21, 2017</td>
</tr>
<tr>
<td>New feature</td>
<td>Added the Starting a State Machine Execution Using CloudWatch Events (p. 27) tutorial.</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>Step Functions allows Lambda function error handling as the preferred error handling method.</td>
<td>March 16, 2017</td>
</tr>
<tr>
<td>New feature</td>
<td>Updated the Handling Error Conditions Using a State Machine (p. 22) tutorial and the Error Handling (p. 61) section.</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>Step Functions is available in EU (Frankfurt).</td>
<td>March 7, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Reorganized the topics in the table of contents and updated the following tutorials:</td>
<td>February 23, 2017</td>
</tr>
<tr>
<td></td>
<td>- Getting Started (p. 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Creating a Lambda State Machine (p. 8)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Creating an Activity State Machine (p. 17)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Handling Error Conditions Using a State Machine (p. 22)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>The State Machines page of the Step Functions console includes the Copy to New and Delete buttons.</td>
<td>February 23, 2017</td>
</tr>
<tr>
<td>New feature</td>
<td>Updated the screenshots to match the console changes.</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>Step Functions supports creating APIs using API Gateway.</td>
<td>February 14, 2017</td>
</tr>
<tr>
<td></td>
<td>Added the Creating a Step Functions API Using API Gateway (p. 29) tutorial.</td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>New feature</td>
<td>• Step Functions supports integration with AWS CloudFormation.</td>
<td>February 10, 2017</td>
</tr>
<tr>
<td></td>
<td>• Added the Creating a Lambda State Machine Using AWS CloudFormation (p. 13) tutorial.</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Clarified the current behavior of the ResultPath and OutputPath fields in relation to Parallel states.</td>
<td>February 6, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>• Clarified state machine naming restrictions in tutorials.</td>
<td>January 5, 2017</td>
</tr>
<tr>
<td></td>
<td>• Corrected some code examples.</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Updated Lambda function examples to use the latest programming model.</td>
<td>December 9, 2016</td>
</tr>
<tr>
<td>New feature</td>
<td>The initial release of Step Functions.</td>
<td>December 1, 2016</td>
</tr>
</tbody>
</table>
AWS Glossary

For the latest AWS terminology, see the AWS Glossary in the AWS General Reference.