# Table of Contents

What is AWS Step Functions? ................................................................................................................ 1  
AWS SDK and Optimized integrations ................................................................................................. 1  
Standard and Express workflows ....................................................................................................... 1  
  Standard workflows .......................................................................................................................... 2  
  Express workflows ............................................................................................................................ 2  
Use cases ........................................................................................................................................ 2  
  Use case #1: Function orchestration ................................................................................................. 2  
  Use case #2: Branching .................................................................................................................... 3  
  Use case #3: Error handling ............................................................................................................ 3  
  Use case #4: Human in the loop ...................................................................................................... 4  
  Use case #5: Parallel processing ..................................................................................................... 4  
  Use case #6: Dynamic parallelism ................................................................................................. 4  
Service integrations ............................................................................................................................ 1  
Supported regions ............................................................................................................................... 7  
Is this your first time using Step Functions? ....................................................................................... 7  
Prerequisites .................................................................................................................................... 8  
  Create an AWS account .................................................................................................................. 8  
  Create an IAM user and group ........................................................................................................ 8  
  Create an access key for your IAM user ......................................................................................... 9  
Getting started .................................................................................................................................. 10  
  Step 1: Create a state machine ...................................................................................................... 10  
  Step 2: Start a new execution .......................................................................................................... 12  
  Step 3: Update a state machine ...................................................................................................... 12  
  Step 4: Clean up ............................................................................................................................ 13  
  Next steps ...................................................................................................................................... 14  
Use cases ........................................................................................................................................ 15  
  Data processing .............................................................................................................................. 15  
  Machine learning ............................................................................................................................. 16  
  Microservice orchestration ............................................................................................................. 17  
  IT and security automation ............................................................................................................ 17  
How Step Functions works ................................................................................................................ 19  
  Standard vs. Express Workflows ................................................................................................. 19  
    Synchronous and Asynchronous Express Workflows .................................................................... 20  
    Execution guarantees ..................................................................................................................... 21  
States .............................................................................................................................................. 21  
  Amazon States Language ............................................................................................................... 23  
  Pass .............................................................................................................................................. 28  
  Task .............................................................................................................................................. 29  
  Choice ......................................................................................................................................... 40  
  Wait .............................................................................................................................................. 44  
  Succeed ....................................................................................................................................... 46  
  Fail ............................................................................................................................................... 46  
  Parallel ....................................................................................................................................... 46  
  Map ............................................................................................................................................... 49  
Transitions ....................................................................................................................................... 53  
State Machine Data ............................................................................................................................ 53  
  Data Format ................................................................................................................................... 54  
  State Machine Input/Output ......................................................................................................... 54  
  State Input/Output .......................................................................................................................... 54  
Input and Output Processing .............................................................................................................. 55  
  Paths ............................................................................................................................................ 57  
  InputPath, Parameters and ResultSelector ................................................................................... 57  
  ItemsPath ..................................................................................................................................... 61  
  ResultPath .................................................................................................................................... 62
<table>
<thead>
<tr>
<th>Tutorials</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a Step Functions State Machine That Uses Lambda</td>
<td>137</td>
</tr>
<tr>
<td>Tutorial: Learn to use the AWS Step Functions Workflow Studio</td>
<td>126</td>
</tr>
<tr>
<td>Error handling</td>
<td>124</td>
</tr>
<tr>
<td>Configure input and output</td>
<td>118</td>
</tr>
<tr>
<td>Error handling</td>
<td>124</td>
</tr>
<tr>
<td>Configure input to a state</td>
<td>118</td>
</tr>
<tr>
<td>Configure output of a state</td>
<td>120</td>
</tr>
<tr>
<td>Error handling</td>
<td>124</td>
</tr>
<tr>
<td>Retry on errors</td>
<td>125</td>
</tr>
<tr>
<td>Catch errors</td>
<td>125</td>
</tr>
<tr>
<td>Timeouts</td>
<td>126</td>
</tr>
<tr>
<td>Heartbeat</td>
<td>126</td>
</tr>
<tr>
<td>Tutorial: Learn to use the AWS Step Functions Workflow Studio</td>
<td>126</td>
</tr>
<tr>
<td>Step 1: Navigate to Workflow Studio</td>
<td>127</td>
</tr>
<tr>
<td>Step 2: Create a state machine</td>
<td>127</td>
</tr>
<tr>
<td>Step 3: Review the auto-generated Amazon States Language definition</td>
<td>129</td>
</tr>
<tr>
<td>Step 4: Start a new execution</td>
<td>131</td>
</tr>
<tr>
<td>Step 5: Update your state machine</td>
<td>132</td>
</tr>
<tr>
<td>Step 6: Clean up</td>
<td>133</td>
</tr>
<tr>
<td>Known limitations</td>
<td>133</td>
</tr>
<tr>
<td>Workflow Studio does not support dynamic resource IDs</td>
<td>134</td>
</tr>
<tr>
<td>A task token is not added to the state definition when you select Wait for callback in Workflow Studio</td>
<td>135</td>
</tr>
<tr>
<td>Some service states are not available in Workflow Studio</td>
<td>135</td>
</tr>
<tr>
<td>Workflow Studio does not support Internet Explorer 11</td>
<td>136</td>
</tr>
<tr>
<td>Tutorials</td>
<td>137</td>
</tr>
<tr>
<td>Create a Step Functions State Machine That Uses Lambda</td>
<td>137</td>
</tr>
<tr>
<td>Step 1: Create a Lambda Function</td>
<td>138</td>
</tr>
<tr>
<td>Step 2: Test the Lambda Function</td>
<td>138</td>
</tr>
<tr>
<td>Step 3: Create a State Machine</td>
<td>139</td>
</tr>
</tbody>
</table>
### Quotas
- General quotas ................................................................. 485
- Quotas related to accounts ................................................. 486
- Quotas related to state throttling ....................................... 486
- Quotas related to API action throttling ............................... 487
- Quotas related to state machine executions ...................... 487
- Quotas related to task executions ...................................... 488
- Other quotas ...................................................................... 488
- Restrictions related to tagging ......................................... 489

### Logging and monitoring
- Amazon CloudWatch Metrics ........................................... 491
  - Metrics That Report a Time Interval ................................. 492
  - Metrics That Report a Count ........................................... 492
  - Execution Metrics .......................................................... 492
  - Activity Metrics ............................................................. 493
  - Lambda Function Metrics .............................................. 494
  - Service Integration Metrics ........................................... 494
  - Service Metrics ............................................................. 495
  - API Metrics ................................................................... 495
  - Viewing Metrics for Step Functions ................................. 496
  - Setting Alarms for Step Functions .................................... 497
- Amazon EventBridge Events ............................................. 499
  - EventBridge payloads ..................................................... 500
  - Step Functions event examples ....................................... 500
  - Routing a Step Functions event to EventBridge ................. 503
- Logging Step Functions Using AWS CloudTrail .................. 504
  - Step Functions Information in CloudTrail ....................... 504
  - Example: Step Functions Log File Entries ....................... 505
- Logging using CloudWatch Logs ....................................... 508
  - Configure logging ........................................................ 508
  - CloudWatch Logs payloads ........................................... 509
  - IAM Policies for logging to CloudWatch Logs ................. 509
  - Log levels ..................................................................... 510

### X-Ray
- Setup and configuration .................................................... 512
- Concepts .......................................................................... 514
- Service integrations ......................................................... 515
- Viewing the X-Ray console ............................................... 516
- Viewing X-Ray tracing information for Step Functions ....... 516
What is AWS Step Functions?

Step Functions is a serverless orchestration service that lets you combine AWS Lambda functions and other AWS services to build business-critical applications. Through Step Functions' graphical console, you see your application's workflow as a series of event-driven steps.

Step Functions is based on state machines and tasks. A state machine is a workflow. A task is a state in a workflow that represents a single unit of work that another AWS service performs. Each step in a workflow is a state.

With Step Functions' built-in controls, you examine the state of each step in your workflow to make sure that your application runs in order and as expected. Depending on your use case, you can have Step Functions call AWS services, such as Lambda, to perform tasks. You can create workflows that process and publish machine learning models. You can have Step Functions control AWS services, such as AWS Glue, to create extract, transform, and load (ETL) workflows. You also can create long-running, automated workflows for applications that require human interaction.

Topics
- AWS SDK and Optimized integrations (p. 1)
- Standard and Express workflows (p. 1)
- Use cases (p. 2)
- Service integrations (p. 1)
- Supported regions (p. 7)
- Is this your first time using Step Functions? (p. 7)

AWS SDK and Optimized integrations

To call other AWS services, you can use Step Functions's AWS SDK integrations, or you can use one of Step Functions's Optimized integrations.

- The AWS SDK integrations (p. 274) let you call any of the over two hundred AWS services directly from your state machine, giving you access to over nine thousand API actions.
- Step Functions's Optimized integrations (p. 299) have been customized to simplify usage in your state machines.

Standard and Express workflows

Step Functions has two workflow types. Standard workflows have exactly-once workflow execution and can run for up to one year. This means that each step in a Standard workflow will execute exactly-once. Express workflows, however, have at-least-once workflow execution and can run for up to five minutes. This means that it's possible that one or more steps in an Express Workflow can execute more than once, while each step in the workflow executes at-least-once.

Executions are instances where you run your workflow to perform tasks. Standard workflows are ideal for long-running, auditable workflows, as they show execution history and visual debugging. Express
workflows are ideal for high-event-rate workloads, such as streaming data processing and IoT data ingestion.

**Standard workflows**

- 2,000 per second execution rate
- 4,000 per second state transition rate
- Priced per state transition
- Shows execution history and visual debugging
- Supports all service integrations and patterns

**Express workflows**

- 100,000 per second execution rate
- Nearly unlimited state transition rate
- Priced per number and duration of executions
- Sends execution history to Amazon CloudWatch
- Supports all service integrations and most patterns

For more information about Standard and Express workflows, including Step Functions pricing, see the following:

- Standard vs. Express Workflows (p. 19)
- AWS Step Functions pricing

**Use cases**

Step Functions manages your application's components and logic, so you can write less code and focus on building and updating your application quickly. This section describes typical use cases for working with Step Functions.

**Use case #1: Function orchestration**

You create a workflow that runs a group of Lambda functions (steps) in a specific order. One Lambda function's output passes to the next Lambda function's input. The last step in your workflow gives a result. With Step Functions, you can see how each step in your workflow interacts with one other, so you can make sure that each step performs its intended function.

For a tutorial that shows you how to create a state machine with a group of functions, see the following:

- Getting started with AWS Step Functions (p. 10)
Use case #2: Branching

A customer requests a credit limit increase. Using a **Choice** (p. 40) state, you can have Step Functions make decisions based on the Choice state's input. If the request is more than your customer's pre-approved credit limit, you can have Step Functions send your customer's request to a manager for sign-off. If the request is less than your customer's pre-approved credit limit, you can have Step Functions approve the request automatically.

Use case #3: Error handling

**Retry**

In this use case, a customer requests a username. The first time, your customer's request is unsuccessful. Using a **Retry** statement, you can have Step Functions try your customer's request again. The second time, your customer's request is successful.

**Catch**

In a similar use case, a customer requests an unavailable username. Using a **Catch** statement, you have Step Functions suggest an available username. If your customer takes the available username, you can have Step Functions go to the next step in your workflow, which is to send a confirmation email. If your customer doesn't take the available username, you have Step Functions go to a different step in your workflow, which is to start the sign-up process over.

For more detailed examples of **Retry** and **Catch** statements, see the following:

- Error handling in Step Functions (p. 88)
Use case #4: Human in the loop

Using a banking app, one of your customers sends money to a friend. Your customer waits for a confirmation email. With a callback and a task token (p. 293), you have Step Functions tell Lambda to send your customer’s money and report back when your customer’s friend receives it. After Lambda reports back that your customer’s friend received the money, you can have Step Functions go to the next step in your workflow, which is to send your customer a confirmation email.

To see a sample project that shows a callback with a task token, see the following:

- Callback Pattern Example (Amazon SQS, Amazon SNS, Lambda) (p. 369)

Use case #5: Parallel processing

A customer converts a video file into five different display resolutions, so viewers can watch the video on multiple devices. Using a Parallel (p. 46) state, Step Functions inputs the video file, so Lambda can process it into the five display resolutions at the same time.

Use case #6: Dynamic parallelism

A customer orders three items, and you need to prepare each item for delivery. You check each item’s availability, gather each item, and then package each item for delivery. Using a Map (p. 49) state,
Step Functions has Lambda process each of your customer's items in parallel. Once all of your customer's items are packaged for delivery, Step Functions goes to the next step in your workflow, which is to send your customer a confirmation email with tracking information.

To see a sample project that shows dynamic parallelism using a Map state, see the following:
- Dynamically process data with a Map state (p. 379)

## Service integrations

Step Functions integrates with multiple AWS services. To combine Step Functions with these services, use the following service integration patterns:

### Request a response (default) (p. 292)

- Call a service, and let Step Functions progress to the next state after it gets an HTTP response.

### Run a job (.sync) (p. 292)

- Call a service, and have Step Functions wait for a job to complete.

### Wait for a callback with a task token (.waitForTaskToken) (p. 293)

- Call a service with a task token, and have Step Functions wait until the task token returns with a callback.

The table below shows the available service integrations and service integration patterns for Step Functions.

Standard Workflows and Express Workflows support the same integrations but do not support the same integration patterns. Express Workflows do not support Run a Job (.sync) or Wait for Callback (.waitForTaskToken). Optimized integrations pattern support is different for each integration. For more information, see Standard vs. Express Workflows (p. 19).

### Standard Workflows

<table>
<thead>
<tr>
<th>Service</th>
<th>Request Response (p. 292)</th>
<th>Run a Job (.sync) (p. 292)</th>
<th>Wait for Callback (.waitForTaskToken) (p. 293)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda (p. 301)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>AWS Batch (p. 303)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DynamoDB (p. 304)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon ECS/AWS Fargate (p. 307)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Amazon SNS (p. 309)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
## Service integrations

<table>
<thead>
<tr>
<th>Service</th>
<th>Request Response (p. 292)</th>
<th>Run a Job (.sync) (p. 292)</th>
<th>Wait for Callback (.waitForTaskToken) (p. 293)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon SQS (p. 311)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>AWS Glue (p. 312)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SageMaker (p. 313)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Amazon EMR (p. 320)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Amazon EMR on EKS (p. 327)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CodeBuild (p. 329)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Athena (p. 332)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Amazon EKS (p. 334)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>API Gateway (p. 343)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>AWS Glue DataBrew (p. 348)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Amazon EventBridge (p. 349)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>AWS Step Functions (p. 351)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>AWS SDK integrations</td>
<td>Over two hundred (p. 275)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Express Workflows

#### Supported service integrations

<table>
<thead>
<tr>
<th>Service</th>
<th>Request Response (p. 292)</th>
<th>Run a Job (.sync) (p. 292)</th>
<th>Wait for Callback (.waitForTaskToken) (p. 293)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized integrations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambda (p. 301)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Batch (p. 303)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DynamoDB (p. 304)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon ECS/AWS Fargate (p. 307)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon SNS (p. 309)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Supported regions

Most AWS regions support Step Functions. For a complete list of AWS regions where Step Functions is available, see the [AWS Region Table](#).

## Is this your first time using Step Functions?

If this is your first time using Step Functions, the following topics help you understand different parts of working with Step Functions, including how Step Functions combines with other AWS services:

- Tutorials for Step Functions (p. 137)
- Sample projects for Step Functions (p. 354)
- AWS Step Functions Data Science SDK for Python (p. 247)

---

<table>
<thead>
<tr>
<th>Service</th>
<th>Request Response (p. 292)</th>
<th>Run a Job (<code>.sync</code>) (p. 292)</th>
<th>Wait for Callback (<code>.waitForTaskToken</code>) (p. 293)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon SQS (p. 311)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Glue (p. 312)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SageMaker (p. 313)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EMR (p. 320)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EMR on EKS (p. 327)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeBuild (p. 329)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Athena (p. 332)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EKS (p. 334)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>API Gateway (p. 343)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Glue DataBrew (p. 348)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EventBridge (p. 349)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Step Functions (p. 351)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS SDK integrations</td>
<td>Over two hundred (p. 275)</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Prerequisites for Getting Started with AWS Step Functions

Before you get started with AWS Step Functions for the first time, complete the prerequisites that are listed on this page.

Topics
- Create an AWS account (p. 8)
- Create an IAM user and group (p. 8)
- Create an access key for your IAM user (p. 9)

Create an AWS account

When you sign up with AWS, you gain access to all of the services that AWS offers, including Step Functions. You’re only charged for the products and services that you use, and you can get started with AWS for free. For more information, see AWS Free Tier.

If you already have an AWS account, skip to the next prerequisite.

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account
2. Follow the online instructions.
   - Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

Create an IAM user and group

To access any AWS service, you must provide credentials. These credentials determine whether you have the correct permissions to use the resources that an AWS service contains.

When you first create an Amazon Web Services (AWS) account, you begin with a single sign-in identity. That identity has complete access to all AWS services and resources in the account. This identity is called the AWS account root user. When you sign in, enter the email address and password that you used to create the account.

Important
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. To view the tasks that require you to sign in as the root user, see Tasks that require root user credentials.

For more information about root user and IAM user credentials, see AWS Account Root User Credentials and IAM User Credentials.
To create an IAM user with administrator permissions

1. With your root user credentials, sign in to IAM console and create an IAM user with administrator permissions. For instructions, see Creating Your First IAM User and Administrators Group in the IAM User Guide.

2. Using your new IAM user credentials, sign in to the AWS Management Console. For more information, see How Users Sign in to Your Account.

For more information about IAM, see the following:

- AWS Identity and Access Management (IAM)
- Getting Started
- IAM User Guide

Create an access key for your IAM user

An access key is made up of two parts, an access key ID and a secret access key. Both keys sign programmatic requests over AWS. Similar to a username and password, both keys are used together.

To create an access key for your IAM user

- Sign in to the IAM console, and create an access key for your IAM user with administrator permissions. For instructions, see Programmatic Access.

  Important
  Protect your AWS account. Never send or share your credentials with anyone outside of your organization. No one who legitimately represents Amazon will ever ask you for your credentials.
Getting started with AWS Step Functions

In this tutorial, you learn the basics of working with Step Functions. You sign in to the Step Functions console, where you create a state machine that uses two Pass states (p. 21). You then start a new execution and review the execution details. You then change one of the Pass state’s result and view the changes. Finally, you perform a clean-up step, where you delete your state machine. At the end of this tutorial, you’ll know how to create, test, debug, and delete a state machine.

Make sure to complete the prerequisites for this tutorial (p. 8).

Topics

• Step 1: Create a state machine (p. 10)
• Step 2: Start a new execution (p. 12)
• Step 3: Update a state machine (p. 12)
• Step 4: Clean up (p. 13)
• Next steps (p. 14)

Step 1: Create a state machine

A state machine is a graphical representation of your workflow that you can use to examine the individual steps that define it.

For more information about state machine structure and common states, see the following:

• State machine structure (p. 24)
• Common states (p. 27)

To create a state machine

1. Sign in to the Step Functions console.
2. Choose Create state machine.
3. On the Choose authoring method page, choose Design your workflow visually.
4. Under Type, choose Standard.

Step Functions has two workflow types: Standard and Express. These workflows determine how Step Functions performs tasks, integrates with AWS services, and manages pricing. After you create a state machine, you can’t change its workflow type.

For a side-by-side comparison of both workflows, under Type, choose Help me decide.
5. Choose Next. This will open Workflow Studio.
Workflow Studio

6. Select the Flow panel, then drag a Pass state to the empty state labelled Drag first state here.

7. Under Definition, review the state machine's workflow.

8. In the code for this workflow, two Pass states are defined. One Pass state is named Hello, and the other Pass state is named World.

```json
{
   "Comment": "A Hello World example of the Amazon States Language using Pass states",
   "StartAt": "Hello",
   "States": {
      "Hello": {
         "Type": "Pass",
         "Next": "World"
      },
      "World": {
         "Type": "Pass",
         "Result": "World",
         "End": true
      }
   }
}
```

9. Choose Next. You can see the Amazon States Language workflow definition.

10. Choose Next. Give your workflow the name HelloWorld.

11. Under Permissions, select Create a new IAM role.

When you create a state machine, you select an IAM role that defines which resources the state machine has permission to access during its execution. Choose from the following options:

- **Create a new IAM role** – Select this option when you want Step Functions to create a new IAM role for you based on the definition of your state machine and its configuration details.

- **Choose an existing role** – Select this option if you previously created an IAM role for Step Functions and your state machine has the correct permissions.
- **Enter a role ARN** – Select this option if you know the ARN details for the IAM role that you want to use for Step Functions.

12. Choose **Create state machine**.

### Step 2: Start a new execution

State machine executions are instances where you run your workflow to perform tasks.

**To start a new execution**

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

2. **Optional** - On the **New execution** page, in the **execution ID** field, you can enter a name of your choice.

   Make sure that the execution name doesn’t contain any non-ASCII characters. If you don’t specify your own execution ID, Step Functions generates a unique execution ID for you.

3. On the **New execution** page, choose **Start execution**.

   After you choose **Start execution**, the Step Functions console directs you to a page that’s titled with your execution ID. On this page, you can review the results of your new execution. Under **Execution details**, you can see your execution ARN and a status to indicate whether your execution succeeded. You can also see the timestamps for when your execution started and ended.

4. To view the results of your execution, choose **Output**.

   The output is **World**.

### Step 3: Update a state machine

Change a **Pass** state's result, and update your state machine for future exceptions. Then view your changes in the visual workflow pane. An exception is an event that disrupts a step in your workflow.

When you update a state machine, your updates are **eventually consistent**. After a few seconds or minutes, all newly started executions will reflect your state machine’s updated definition and roleARN. All currently running executions will run to completion under the previous definition and roleARN before updating.

**To change a Pass state’s result**

1. On the page titled with your execution ID, choose **Edit state machine**.

2. On the **Edit Helloworld** page, in the code pane, update the second Result to **World has been updated**!

```json
{
  "Comment": "A Hello World example of the Amazon States Language using Pass states",
  "StartAt": "Hello",
  "States": {
    "Hello": {
      "Type": "Pass",
      "Result": "Hello",
      "Next": "World"
    },
    "World": {
      "Type": "Pass",
      "Result": "World has been updated!"
    }
  }
}
```
3. Choose Save, and then choose Start execution.

   After you choose Save, the following message appears:

   "The changes to your state machine may affect which resources it needs to access. To ensure your state machine has the right permissions, you might need to edit the current IAM role, create a new one, or select a different role."

   This message is standard. Choose Save anyway.


5. On the next page, in the visual workflow pane, examine the individual steps that define your workflow.

6. Optional - To export the graph of your workflow to an SVG or PNG file, choose Export.

7. To view the results of your execution, in the visual workflow pane, choose World, and then, under Step details, choose Output.

   The output is World has been updated!

---

**Step 4: Clean up**

In some cases, you might need to delete your state machine and execution role.

If you're done with this tutorial, delete your state machine and the execution role that Step Functions created for you.

**To delete your state machine**

1. From the navigation menu, choose State machines.
2. On the State machines page, under State machines, select Helloworld, and then choose Delete.

   After you choose Delete, the following message appears:

   “You are about to delete your state machine. Do you want to proceed?”

   This message is standard. Choose Delete state machine.

   A green status bar appears at the top of your screen. The green status bar tells you that your state machine is marked for deletion. Your state machine will be removed when all of its executions stop running.

**To delete your execution role**

1. Open the Roles page for IAM.
2. Choose the IAM role that Step Functions created for you: StepFunctions-Helloworld-role-EXAMPLE.
3. Choose Delete role.
4. Choose Yes, delete.

   Now that you completed this tutorial, you know how to create, test, debug, and delete a state machine.
Next steps

For more tutorials about working with Step Functions, try the following:

- Create a Lambda State Machine (p. 137)
- Create a Lambda State Machine Using AWS CloudFormation (p. 217)
- Create an Activity State Machine (p. 162)
- Handle Error Conditions Using a State Machine (p. 140)
- Start a State Machine Using Amazon CloudWatch Events (p. 148)
- Create a Step Functions API Using Amazon API Gateway (p. 152)
Use cases

AWS Step Functions lets you build visual workflows that help rapidly translate business requirements into applications. Step Functions manages state, checkpoints and restarts for you, and provides built-in capabilities to automatically deal with errors and exceptions. To better understand the capabilities Step Functions can provide you with, read through the following use cases:

Topics
- Data processing (p. 15)
- Machine learning (p. 16)
- Microservice orchestration (p. 17)
- IT and security automation (p. 17)

Data processing

As the volume of data grows, coming from increasingly diverse sources, organizations find they need to move quickly to process this data to ensure they make faster, well-informed business decisions. To process data at scale, organizations need to elastically provision resources to manage the information they receive from mobile devices, applications, satellites, marketing and sales, operational data stores, infrastructure, and more.

Step Functions provides the scalability, reliability, and availability needed to successfully manage your data processing workflows. You can manage millions of concurrent executions with Step Functions as it scales horizontally and provides fault-tolerant workflows. Process data faster using parallel executions like Step Functions’ Parallel (p. 46) state type, or dynamic parallelism using its Map (p. 49) state type. As part of your workflow, you can use the Map (p. 49) state to iterate over objects in a static data store like an Amazon S3 bucket. Step Functions also lets you easily retry failed executions, or choose a specific way to handle errors without the need to manage a complex process.

Depending upon your data processing needs, Step Functions directly integrates with other data processing services provided by AWS such as AWS Batch (p. 303) for batch processing, Amazon EMR (p. 320) for big data processing, AWS Glue (p. 312) for data preparation, Athena (p. 332) for data analysis, and AWS Lambda (p. 301) for compute.

Examples of the types of data processing workflows that customers use Step Functions to accomplish include:

File, video, and image processing
- Take a collection of video files and convert them to other sizes or resolutions that are ideal for the device they will be displayed on, such as mobile phones, laptops, or a television.
- Take a large collection of photos uploaded by users and convert them into thumbnails or various resolution images that can then be displayed on users’ websites.
- Take semi-structured data, such as a CSV file, and combine it with unstructured data, such as an invoice, to produce a business report that is sent to business stakeholders monthly.
- Take earth observing data collected from satellites, convert it into formats that align with each other and then add other data sources collected on earth for additional insight.
- Take the transportation logs from various modes of transportation for products and look for optimizations using Monte Carlo Simulations and then send reports back to the organizations and people that are relying on you to ship their goods.
Coordinate extract, transform and load (ETL) jobs:

- Combine sales opportunity records with marketing metric datasets through a series of data preparation steps using AWS Glue, and produce business intelligence reports that can be used across the organization.
- Create, start, and terminate an Amazon EMR cluster for big data processing.

Batch processing and High Performance Computing (HPC) workloads:

- Build a genomics secondary analysis pipeline that processes raw whole genome sequences into variant calls. Align raw files to a reference sequence, and call variants on a specified list of chromosomes using dynamic parallelism.
- Find efficiencies in the production of your next mobile device or other electronics by simulating various layouts using different electric and chemical compounds. Run large batch processing of your workloads through various simulations to get the optimal design.

Machine learning

Machine learning enables organizations to quickly analyze collected data to identify patterns, then make decisions with minimal human intervention. Machine learning starts with an initial set of data, known as training data. This training data helps to increase a machine learning model’s prediction accuracy, and serves as the foundation through which this model learns. Once the model is considered accurate enough to meet business needs, it’s deployed to production. The AWS Step Functions Data Science Software Development Kit (SDK) (p. 247) is an open-source library that allows you to easily create workflows that preprocess data, train and then publish your models using Amazon SageMaker and Step Functions.

Preprocessing existing data sets is how an organization often creates training data. This method adds information, such as by labeling objects in an image, annotating text or processing audio. To preprocess data you can use AWS Glue, or you can create a SageMaker notebook instance that runs the Jupyter Notebook app. Once your data is ready, it can be uploaded to Amazon S3 for easy access. As machine learning models are trained, you can make adjustments to each model’s parameters to improve accuracy until it’s ready for deployment.

Step Functions lets you to orchestrate end-to-end machine learning workflows on SageMaker. These workflows can include data preprocessing, post-processing, feature engineering, data validation, and model evaluation. Once the model has been deployed to production, you can refine and test new approaches to continually improve business outcomes. You can create production-ready workflows directly in Python, or you can use the Step Functions Data Science SDK to copy that workflow, experiment with new options, and place the refined workflow in production.

Some types of machine learning workflows that customers use Step Functions for include:

**Fraud Detection**

- Identify and prevent fraudulent transactions, such as credit fraud, from occurring.
- Detect and prevent account takeovers using trained machine learning models.
- Identify promotional abuse, including the creation of fake accounts, so you can quickly take action.

**Personalization and Recommendations**

- Recommend products to targeted customers based upon what is predicted to attract their interest.
- Predict whether a customer will upgrade their account from a free tier to a paid subscription.
Data Enrichment

- Use data enrichment as part of preprocessing to provide better training data for more accurate machine learning models.
- Annotate text and audio excerpts to add syntactical information, such as sarcasm and slang.
- Label additional objects in images to provide critical information for the model to learn from, such as whether an object is an apple, a basketball, a rock, or an animal.

Microservice orchestration

Microservice architecture breaks applications into loosely coupled services. Benefits include improved scalability, increased resiliency, and faster time to market. Each microservice is independent, making it easy to scale up a single service or function without needing to scale the entire application. Individual services are loosely coupled, letting independent teams focus on a single business process, without the need for them to understand the entire application. Microservices also let you choose which individual components suit your business needs, giving you the flexibility to change your selection without rewriting your entire workflow. Different teams can use the programming languages and frameworks of their choice to work with their microservice, and this microservice can still communicate with any other in the application through application programming interfaces (APIs).

Step Functions gives you several ways to manage your microservice workflows. For long-running workflows you can use Standard Workflows with the AWS Fargate integration to orchestrate applications running in containers. For short-duration, high-volume workflows that require an immediate response, Synchronous Express Workflows (p. 20) are ideal. These can be used for web-based or mobile applications, which often have workflows of short duration, and require the completion of a series of steps before they return a response. You can directly trigger a Synchronous Express Workflows from Amazon API Gateway, and the connection is held open until the workflow completes or timeouts. For short duration workflows that do not require an immediate response, Step Functions provides Asynchronous Express Workflows.

Examples of some API orchestrations that use Step Functions include:

**Synchronous or real-time workflows**

- Change a value in a record such as updating an employee’s last name and have the change immediately visible on the screen.
- Update an order during checkout, such as adding, removing, or changing the quantity of an item, then immediately reflect the update back to the customer.
- Run a quick processing job and immediately return the result back to the requester.

**Container Orchestration**

- Run jobs on Kubernetes with Amazon Elastic Kubernetes Service or on Amazon Elastic Container Service (ECS) with Fargate and integrate with other AWS services, such as sending notifications with Amazon SNS, as part of the same workflow.

IT and security automation

IT automation can help manage increasingly complex and time-consuming operations, such as upgrading and patching software, deploying security updates to address vulnerabilities, selecting infrastructure, synchronizing data, routing support tickets, and more. The automation of repetitive and time-consuming tasks can allow your organization to complete routine operations quickly and consistently on a large
Step Functions allows you to create workflows that automatically scale to meet the needs of your business without requiring manual intervention. In cases where an error occurs in your workflow, it often does not require manual intervention. Step Functions lets you automatically retry failed tasks (p. 90) and an exponential backoff (p. 93) that can manage errors in your workflow.

There can be situations where human intervention is required before the workflow can progress. For example, approving a substantial credit increase may require human approval. To manage this, you can define branching logic in Step Functions, so that only requests over a defined amount require human approval, while all other requests are automatically completed. In cases where human approval is required, Step Functions lets you pause the workflow at a specific step, wait for a response, and then continue the workflow once the response is received.

Some examples of the types of automation workflows that customers use Step Functions for include:

**IT automation**

- Auto-remediate incidents like opening an SSH port, low disk space, or when a public access is given to a Amazon S3 bucket.
- Automate the deployment of AWS CloudFormation StackSets

**Security automation**

- Automate the response to a scenario where an IAM user and user access key has been exposed.
- Auto-remediate security incident responses according to policy actions defined such as restricting actions to specific ARNs or applying other actions.
- Warn employees of phishing emails within seconds of receipt.

**Human Approval**

- Automate the training of machine learning model and then require manual approval of the model by a data scientist before then automatically deploying or rejecting the model based upon the response received.
- Automate the routing of customer feedback received based on sentiment analysis so that those with a negative sentiment are immediately escalated for manual review.
How Step Functions works

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

Topics
- Standard vs. Express Workflows (p. 19)
- States (p. 21)
- Transitions (p. 53)
- State Machine Data (p. 53)
- Input and Output Processing in Step Functions (p. 55)
- Executions in Step Functions (p. 73)
- Error handling in Step Functions (p. 88)
- Invoke AWS Step Functions from other services (p. 96)
- Read Consistency in Step Functions (p. 97)
- Tagging in Step Functions (p. 97)

Standard vs. Express Workflows

When you create a state machine, you can select a Type of either Standard (default) or Express. In both cases, you define your state machine using the Amazon States Language. Your state machine executions will behave differently, depending on which Type you select. The Type you choose cannot be changed after your state machine has been created.

Standard Workflows are ideal for long-running, durable, and auditable workflows. They can run for up to a year and you can retrieve the full execution history using the Step Functions API, up to 90 days after your execution completes. Standard Workflows employ an exactly-once model, where your tasks and states are never executed more than once unless you have specified retry behavior in ASL. This makes them suited to orchestrating non-idempotent actions, such as starting an Amazon EMR cluster or processing payments. Standard Workflows executions are billed according to the number of state transitions processed.

Express Workflows are ideal for high-volume, event-processing workloads such as IoT data ingestion, streaming data processing and transformation, and mobile application backends. They can run for up to five minutes. Express Workflows employ an at-least-once model, where there is a possibility that an execution might be run more than once. This makes them ideal for orchestrating idempotent actions such as transforming input data and storing via PUT in Amazon DynamoDB. Express Workflow executions are billed by the number of executions, the duration of execution, and the memory consumed.

Standard and Express Workflows can automatically start in response to events such as HTTP requests via Amazon API Gateway (fully-managed APIs at scale), IoT Rules and over 140 event sources in Amazon EventBridge.

Standard vs Express Workflows

<table>
<thead>
<tr>
<th></th>
<th>Standard Workflows</th>
<th>Express Workflows: Synchronous and Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum duration</td>
<td>1 year.</td>
<td>5 minutes.</td>
</tr>
</tbody>
</table>
Synchronous and Asynchronous Express Workflows

<table>
<thead>
<tr>
<th>Standard Workflows</th>
<th>Express Workflows: Synchronous and Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported execution start rate (p. 485)</td>
<td>Over 2,000 per second</td>
</tr>
<tr>
<td>Supported state transition rate (p. 485)</td>
<td>Over 4,000 per second per account</td>
</tr>
<tr>
<td>Pricing</td>
<td>Priced per state transition. A state transition is counted each time a step in your execution is completed.</td>
</tr>
<tr>
<td>Execution history</td>
<td>Executions can be listed and described with Step Functions APIs, and visually debugged through the console. They can also be inspected in CloudWatch Logs by enabling logging on your state machine.</td>
</tr>
</tbody>
</table>
| Execution semantics (p. 21) | Exactly-once workflow execution. | Asynchronous Express Workflows: At-least-once workflow execution. Synchro
| Service integrations (p. 271) | Supports all service integrations and patterns. | Supports all service integrations. Does not support Job-run (.sync) or Callback (.waitForTaskToken) patterns. |
| Step Functions activities | Supports Step Functions activities. | Does not support Step Functions activities. |

Synchronous and Asynchronous Express Workflows

There are two types of Express Workflows that you can choose, Asynchronous Express Workflows and Synchronous Express Workflows.

- **Asynchronous Express Workflows** return confirmation that the workflow was started, but do not wait for the workflow to complete. To get the result, you must poll the service's CloudWatch Logs (p. 508). Asynchronous Express Workflows can be used when you don't require immediate response output, such as messaging services, or data processing that other services don't depend on. Asynchronous Express Workflows can be started in response to an event, by a nested workflow in Step Functions, or by using the StartExecution API call.

- **Synchronous Express Workflows** start a workflow, wait until it completes, then return the result. Synchronous Express Workflows can be used to orchestrate microservices, and allow you to develop applications without the need to develop additional code to handle errors, retries, or execute parallel
tasks. Synchronous Express Workflows can be invoked from Amazon API Gateway, AWS Lambda, or by using the `StartSyncExecution` API call.

**Note**
If you run Step Functions Express Workflows synchronously from the console, the `StartSyncExecution` request elapses after 60 seconds. To run the Express Workflows synchronously for a duration of up to five minutes, make the `StartSyncExecution` requests using the AWS SDK or CLI instead of the Step Functions console.

Synchronous Express execution API calls do not contribute to the existing account capacity limits. Step Functions will provide capacity on demand and will automatically scale with sustained workload. Surges in workload may be throttled until capacity is available.

### Execution guarantees

<table>
<thead>
<tr>
<th>Standard Workflows</th>
<th>Asynchronous Express Workflows</th>
<th>Synchronous Express Workflows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exactly-once workflow execution</td>
<td>At-least-once workflow execution</td>
<td>At-most-once workflow execution</td>
</tr>
</tbody>
</table>

The execution state is internally persisted on every state transition. To guarantee that only one workflow with the same name can run, Step Functions will return an idempotent response when you start a Standard Workflow with the same name as an already running workflow. In this case, Step Functions will not start a new workflow. When the workflow completes, Step Functions will respond with an exception. After 90 days, the workflow data will be removed, and the name can then be reused.

No internally persisted state for workflow progress. If you attempt to start an Express Workflow with the same name more than once, each attempt causes a workflow to start concurrently. In rare cases, the internal state of a workflow can be lost, and the workflow will be automatically restarted from beginning. You should ensure your state machine logic is idempotent and should not be affected adversely by multiple concurrent executions of the same input.

After a workflow starts, Step Functions will wait and returns the result as part of the API response. If service exceptions occur, Step Functions will not restart from the beginning. You should ensure your state machine logic is idempotent and should not be affected adversely by multiple concurrent executions of the same input.

### States

Individual states can make decisions based on their input, perform actions, and pass output to other states. In AWS Step Functions you define your workflows in the Amazon States Language. The Step Functions console provides a graphical representation of that state machine to help visualize your application logic.
States are elements in your state machine. A state is referred to by its name, which can be any string, but which must be unique within the scope of the entire state machine.

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

- Do some work in your state machine (a Task (p. 29) state)
- Make a choice between branches of execution (a Choice (p. 40) state)
- Stop an execution with a failure or success (a Fail (p. 46) or Succeed (p. 46) state)
- Simply pass its input to its output or inject some fixed data (a Pass (p. 28) state)
- Provide a delay for a certain amount of time or until a specified time/date (a Wait (p. 44) state)
- Begin parallel branches of execution (a Parallel (p. 46) state)
- Dynamically iterate steps (a Map (p. 49) state)

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

```json
"HelloWorld": {
  "Type": "Task",
  "Next": "AfterHelloWorldState",
  "Comment": "Run the HelloWorld Lambda function"
}
```

States share many common features:

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

**Note**

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

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

After you have created and executed Standard Workflows, 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. For more information, see Viewing and debugging executions on the Step Functions console (p. 75)

After you have created and executed Express Workflows, and if logging is enabled, you can access information about the execution in Amazon CloudWatch Logs (p. 508).

**Topics**

- Amazon States Language (p. 23)
- Pass (p. 28)
- Task (p. 29)
- Choice (p. 40)
- Wait (p. 44)
- Succeed (p. 46)
- Fail (p. 46)
- Parallel (p. 46)
Amazon States Language

The Amazon States Language is a JSON-based, structured language used to define your state machine, a collection of states (p. 21), 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. 10).

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"
    },
    "DefaultState": {
      "Type": "Fail",
      "Error": "DefaultStateError",
      "Cause": "No Matches!"
    }
  },
  "NextState": {
```
“Type”: "Task",
"End": true
}
}
}

Topics
• State Machine Structure (p. 24)
• Intrinsic functions (p. 25)
• Common State Fields (p. 27)

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

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

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

TimeoutSeconds (Optional)
The maximum number of seconds an execution of the state machine can run. If it runs longer than
the specified time, the execution fails with a States.Timeout Error Name (p. 89).

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

States (Required)
An object containing a comma-delimited set of states.

The States field contains States (p. 21).

```
{
  "State1" : {
  },

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

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

The following is an example.

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

24
When an execution of this state machine is launched, the system begins with the state referenced in the `StartAt` field ("HelloWorld"). If this state has an "End": true field, the execution stops and returns a result. Otherwise, the system looks for a "Next": field and continues with that state next. This process repeats until the system reaches a terminal state (a state with "Type": "Succeed", "Type": "Fail", or "End": true), or a runtime error occurs.

The following rules apply to states within a state machine:

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

**Intrinsic functions**

The Amazon States Language provides several Intrinsics to allow basic operations without Task states. Intrinsics are constructs that look like functions in programming languages and can be used to help Payload Builders process the data going to and from Task Resources.

**States.Format**

Supports string construction from literal and interpolated values, and takes one or more arguments. The Value of the first argument must be a string, and may include zero or more instances of the character sequence `{}`. There must be as many remaining arguments in the Invocation as there are occurrences of `{}`. The interpreter returns the first-argument string with each `{}` replaced by the Value of the positionally-corresponding argument in the Intrinsic invocation.

Example: given the following input:

```
{
  "name": "Arnav",
  "template": "Hello, my name is {}.
}
```

**States.Format** can be used as follows:

```
States.Format('Hello, my name is {}.', $.name)
```

or

```
States.Format($.template, $.name)
```

Both result in an output of Hello, my name is Arnav.

**States.StringToJson**

Takes a single argument, a reference path to an escaped JSON string.
The interpreter applies a JSON parser returns its parsed JSON form. For example, given the following input:

```json
{
  "escapedJsonString": "{"foo": \"bar\"}"
}
```

You could use

```json
States.StringToJson($.escapedJsonString)
```

Which would return

```
{ "foo": "bar" }
```

**States.JsonToString**

Takes a single argument. This argument must be a Path. The interpreter returns a string which is a JSON text representing the data identified by the Path. For example, given the following input:

```json
{
  "unescapedJson": {
    "foo": "bar"
  }
}
```

You could use

```json
States.JsonToString($.unescapedJson)
```

Which would return

```
{"foo": \"bar\"}
```

**States.Array**

This intrinsic takes zero or more arguments. The interpreter returns a JSON array containing the Values of the arguments, in the order provided. For example, given the following input:

```json
{
  "Id": 123456
}
```

You could use

```
"BuildId.$": "States.Array($.Id)"
```

Which would return “BuildId”: [123456]

**Reserved characters in intrinsic functions**

The following characters are reserved for intrinsic functions, and must be escaped with a backslash (`) if you want them to appear in the Value: `{}`, and `\`.

If the character `\` needs to appear as part of the value without serving as an escape character, it must first be escaped with a backslash. Reserved characters for intrinsic invocations are escaped as follows:
• The literal string \' represents '.
• The literal string \{ represents {.
• The literal string \} represents }.
• The literal string \\ represents \\

In JSON, backslashes contained in a string literal value must be escaped with another backslash. The equivalent list for JSON is:

• The escaped string \\
' represents '.
• The escaped string \\
{ represents {.
• The escaped string \\
} represents }.
• The escaped string \\
\\ represents \\\n
**Note**
If an open escape backslash \ is found in the intrinsic invocation string, the interpreter will throw a runtime error.

### Fields that support intrinsic functions

The following table shows which fields support intrinsic functions for each state.

#### Fields that support intrinsic functions

<table>
<thead>
<tr>
<th>State</th>
<th>Pass</th>
<th>Task</th>
<th>Choice</th>
<th>Wait</th>
<th>Succeed</th>
<th>Fail</th>
<th>Parallel</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputPath</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters/</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ResultSelector</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ResultPath</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutputPath</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Comparison Operator&gt;Path</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TimeoutSecondsPath</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HeartbeatSecondsPath</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Common State Fields

**Type (Required)**

The state's type.

**Next**

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

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

Comment (Optional)

Holds a human-readable description of the state.

InputPath (Optional)

A path (p. 55) 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. 55).

OutputPath (Optional)

A path (p. 55) that selects a portion of the state's output to be passed to the next state. If omitted, it has the value $ which designates the entire input. For more information, see Input and Output Processing (p. 55).

Pass

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

In addition to the common state fields (p. 27), Pass states allow the following fields.

Result (Optional)

Refers to the output of a virtual task that is passed on to the next state. If you include the ResultPath field in your state machine definition, Result is placed as specified by ResultPath and passed on to the next state.

ResultPath (Optional)

Specifies where to place the output (relative to the input) of the virtual task specified in Result. The input is further filtered as specified by the OutputPath field (if present) before being used as the state's output. For more information, see Input and Output Processing (p. 55).

Parameters (Optional)

Creates a collection of key-value pairs that will be passed as input. You can specify Parameters as a static value or select from the input using a path. For more information, see InputPath, Parameters and ResultSelector (p. 57).

Pass State Example

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

"No-op": {
    "Type": "Pass",
    "Result": {
        "x-datum": 0.381018,
        "y-datum": 622.2269926397355
    },
    "ResultPath": ".coords",
    "Next": "End"
}
Suppose the input to this state is the following.

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

Then the output would be this.

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

---

**Task**

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

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

AWS Step Functions can invoke Lambda functions directly from a task state. A Lambda function is a cloud-native task that runs on AWS Lambda. You can write Lambda functions in a variety of programming languages, using the AWS Management Console or by uploading code to Lambda. For more information see ??? (p. 137).

**Note**

Step Functions can coordinate some AWS services directly from a task state. For more information see Working with other services (p. 271).

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

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

In addition to the common state fields (p. 27), Task states have the following fields.

**Resource (Required)**

A URI, especially an ARN that uniquely identifies the specific task to execute.

**Parameters (Optional)**

Used to pass information to the API actions of connected resources. The parameters can use a mix of static JSON and JsonPath. For more information, see Pass Parameters to a Service API (p. 296).

**ResultPath (Optional)**

Specifies where (in the input) to place the results of executing the task that's specified in Resource. The input is then filtered as specified by the OutputPath field (if present) before being used as the state's output. For more information, see Input and Output Processing (p. 55).

**ResultSelector (Optional)**

Pass a collection of key value pairs, where the values are static or selected from the result. For more information, see ResultSelector (p. 60).
**Task State**

**Retry (Optional)**

An array of objects, called Retriers, that define a retry policy if the state encounters runtime errors. For more information, see Examples using Retry and using Catch (p. 93).

**Catch (Optional)**

An array of objects, called Catchers, that define a fallback state. This state is executed if the state encounters runtime errors and its retry policy is exhausted or isn't defined. For more information, see Fallback States (p. 92).

**TimeoutSeconds (Optional)**

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

**TimeoutSecondsPath (Optional)**

If you want to provide a timeout value dynamically from the state input using a reference path, use TimeoutSecondsPath. When resolved, the reference path must select fields whose values are positive integers.

**HeartbeatSeconds (Optional)**

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

**HeartbeatSecondsPath (Optional)**

If you want to provide a heartbeat value dynamically from the state input using a reference path, use HeartbeatSecondsPath. When resolved, the reference path must select fields whose values are positive integers.

**Note**

A Task state cannot include both TimeoutSeconds and TimeoutSecondsPath

A Task state cannot include both HeartbeatSeconds and HeartbeatSecondsPath

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

**Task state timeouts and heartbeat intervals**

It's a good practice to set a timeout value and a heartbeat interval for long-running activities. This can be done by specifying the timeout and heartbeat values, or by setting them dynamically.

**Static timeout and heartbeat notification example**

When HelloWorld completes, the next state (here called NextState) will be run.

If this task fails to complete within 300 seconds, or doesn't send heartbeat notifications in intervals of 60 seconds, the task is marked as failed.

```
"ActivityState": { 
"Type": "Task", 
```
Dynamic task timeout and heartbeat notification example

In this example, when the AWS Glue job completes, the next state will be run.

If this task fails to complete within the interval set dynamically by the AWS Glue job, or doesn't send heartbeat notifications for the interval set by the AWS Glue job, the task is marked as failed.

```
"ActivityState": {
  "Type": "Task",
  "Resource": "arn:aws:states:::glue:startJobRun.sync",
  "Parameters": {
    "JobName": "myGlueJob"
  },
  "TimeoutSecondsPath": "$params.maxTime",
  "HeartbeatSecondsPath": "$params.heartbeat",
  "Next": "NextState"
}
```

Specifying Resource ARNs in Tasks

The Resource field's ARN is specified using the following pattern.

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

In this pattern:

- **partition** is the AWS Step Functions partition to use, most commonly `aws`.
- **service** indicates the AWS service used to execute the task, and is:
  - `states` for an activity (p. 32).
  - `lambda` for a Lambda function (p. 32).
- **region** is the **AWS Region** in which the Step Functions activity or state machine type or Lambda function has been created.
- **account** is your AWS account ID.
- **task_type** is the type of task to run. It is one of the following values:
  - **activity** – An activity (p. 32).
  - **function** – A Lambda function (p. 32).
  - **servicename** – The name of a supported connected service (see Optimized integrations for Step Functions (p. 299)).
- **name** is the registered resource name (activity name, Lambda function name, or service API action).

**Note**

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

**Task types**

The following task types are supported:

- **Activity** (p. 32)
Lambda functions (p. 32)
A supported AWS service (p. 271)

The following sections provide more detail about each task type.

**Activity**

Activities represent workers (processes or threads), implemented and hosted by you, that perform a specific task. They are supported only by Standard Workflows, not Express Workflows.

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. 31).

**Note**

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

For more information about creating an activity and implementing workers, see Activities (p. 33).

**Lambda functions**

Lambda tasks 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. 31).

For example:

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

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

**A supported AWS service**

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

Connected service Resource ARNs use the following syntax.

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

**Note**

To create a synchronous connection to a connected resource, append .sync to the APIname entry in the ARN. For more information, see Working with other services (p. 271).
For example:

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

## Activities

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

### Overview

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

**Note**

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

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

### APIs Related to Activity Tasks

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

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

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

Note
Polling for activity tasks with GetActivityTask can cause latency in some implementations.
See Avoid latency when polling for activity tasks (p. 269).

Waiting for an Activity Task to Complete

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

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

1. Create an activity by using the console, or by using CreateActivity. Make a note of the activity ARN.
2. Reference that ARN in an activity task state in your state machine definition and set TimeoutSeconds.
3. Implement an activity worker that polls for work by using GetActivityTask, referencing that activity ARN.
4. Use SendTaskHeartbeat periodically within the time you set in HeartbeatSeconds (p. 29) in your state machine task definition to keep the task from timing out.
5. Start an execution of your state machine.
6. Start your activity worker process.

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

Next Steps

For a more detailed look at creating state machines that use an activity workers, see:

• Creating an Activity State Machine Using Step Functions (p. 162)
• Example Activity Worker in Ruby (p. 34)

Example Activity Worker in Ruby

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

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

• For more information about the AWS SDK for Ruby, see the AWS SDK for Ruby API Reference.
• To download this code and related resources, see the step-functions-ruby-activity-worker repository on GitHub.
The following Ruby code is the main entry point for this example Ruby activity worker.

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>require_relative</td>
<td>Relative path to the following example activity worker code.</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,</td>
</tr>
<tr>
<td></td>
<td>between 10 and 20 threads should be sufficient. The longer the activity</td>
</tr>
<tr>
<td></td>
<td>takes to process, the more threads it might need. As an estimate, multiply</td>
</tr>
<tr>
<td></td>
<td>the number of process activities per second by the 99th percentile activity</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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, referenced with ../lib/step_functions/activity in your code.

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

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

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

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

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

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

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

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

AWS Step Functions Developer Guide
Task
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
def shutdown_worker(worker)
    worker.kill
end

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

def wait_activities_drained
    wait_condition { @sink.empty? }  
end

def wait_activities_completed
    wait_condition { @activities.empty? }  
end

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

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

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

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

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

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

    def stop
        @running = false
    end

    def kill

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

def wait
  @thread.join
end
end

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

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

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

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

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

class HeartbeatWorker < Worker
    def initialize(options = {})
        @states = options[:states]
        @activities = options[:activities]
        @heartbeat_delay = options[:heartbeat_delay]
        @max_retry = options[:max_retry]
        @logger = Logger.new(STDOUT)
    end
    def run
        sleep(@heartbeat_delay)
        @activities.each do |token|
            send_heartbeat(token)
        end
    end
    def send_heartbeat(token)
        StepFunctions.with_retries(max_retry: @max_retry) do
            begin
                @states.send_task_heartbeat(token)
            rescue => e
                @logger.error('Failed to send heartbeat for activity')
                @logger.error(e)
            end
        end
    end
end

Choice

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

In addition to most of the common state fields (p. 27), Choice states introduce the following additional fields.

**Choices (Required)**

An array of Choice Rules (p. 42) that determines which state the state machine transitions to next.
Default (Optional, Recommended)

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

Important
Choice states don’t support the End field. In addition, they use Next only inside their Choices field.

Choice State Example

The following is an example of a Choice state and other states that it transitions to.

Note
You must specify the $.type field. If the state input doesn’t contain the $.type field, the execution fails and an error is displayed in the execution history. You can only specify a string in the StringEquals field that matches a literal value. For example, "StringEquals": "Buy".

```
"ChoiceStateX": {
  "Type": "Choice",
  "Choices": [
    {
      "Not": {
        "Variable": "$.type",
        "StringEquals": "Private"
      },
      "Next": "Public"
    },
    {
      "Variable": "$.value",
      "NumericEquals": 0,
      "Next": "ValueIsZero"
    },
    {
      "And": ["ValueInTwenties"
        {
          "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"
  }
}
```
In this example, the state machine starts with the following input value.

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

Step Functions transitions to the ValueInTwenties state, based on the value field.

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

### Choice Rules

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

- A **comparison** – Two fields that specify an input variable to compare, the type of comparison, and the value to compare the variable to. Choice Rules support comparison between two variables. Within a Choice Rule, the value of Variable can be compared with another value from the state input by appending Path to name of supported comparison operators.
- 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 string is null.

```json
{
  "Variable": ".foo",
  "Null": true,
  "Next": "FirstMatchState"
}
```
The following example shows how the StringEquals rule is only evaluated when 
$.keyThatMightNotExist exists because of the preceding IsPresent Choice Rule.

"And": [
  {
    "Variable": "$.keyThatMightNotExist",
    "IsPresent": true
  },
  {
    "Variable": "$.keyThatMightNotExist",
    "StringEquals": "foo"
  }
]

The following example checks whether a pattern with a wildcard matches.

{  
  "Variable": "$.foo",
  "StringMatches": "log-*.txt"
}

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

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

The following example compares a variable with another value from the state input.

{  
  "Variable": "$.foo",
  "StringEqualsPath": "$.bar"
}

Step Functions examines each of the Choice Rules in the order listed in the Choices field. Then it transitions to the state specified in the Next field of the first Choice Rule in which the variable matches the value according to the comparison operator.

The following comparison operators are supported:

- And
- BooleanEquals, BooleanEqualsPath
- IsBoolean
- IsNull
- IsNumeric
- IsPresent
- IsString
- IsTimestamp
• Not
• NumericEquals, NumericEqualsPath
• NumericGreaterThan, NumericGreaterThanPath
• NumericGreaterThanEquals, NumericGreaterThanEqualsPath
• NumericLessThan, NumericLessThanPath
• NumericLessThanEquals, NumericLessThanEqualsPath
• Or
• StringEquals, StringEqualsPath
• StringGreaterThan, StringGreaterThanPath
• StringGreaterThanEquals, StringGreaterThanEqualsPath
• StringLessThan, StringLessThanPath
• StringLessThanEquals, StringLessThanEqualsPath
• StringMatches
• TimestampEquals, TimestampEqualsPath
• TimestampGreaterThan, TimestampGreaterThanPath
• TimestampGreaterThanEquals, TimestampGreaterThanEqualsPath
• TimestampLessThan, TimestampLessThanPath
• TimestampLessThanEquals, TimestampLessThanEqualsPath

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

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

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

To understand the behavior of string comparisons, see the Java compareTo documentation. The values of the And or 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.

String comparison against patterns with one or more wildcards (“*”) can be performed with the StringMatches comparison operator. The wildcard character is escaped by using the standard \ (Ex: “\*”). No characters other than “*” have any special meaning during matching.

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. 27), Wait states have one of the following fields.
Seconds

A time, in seconds, to wait before beginning the state specified in the `Next` field. You must specify time as a positive, integer value.

Timestamp

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

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

SecondsPath

A time, in seconds, to wait before beginning the state specified in the `Next` field, specified using a path (p. 55) from the state's input data.

You must specify an integer value for this field.

TimestampPath

An absolute time to wait until beginning the state specified in the `Next` field, specified using a path (p. 55) from the state's input data.

Note

You must specify exactly one of `Seconds`, `Timestamp`, `SecondsPath`, or `TimestampPath`. In addition, the maximum wait time that you can specify for Standard Workflows and Express workflows is one year and five minutes respectively.

Wait State Examples

The following `Wait` state introduces a 10-second delay into a state machine.

```
"wait_ten_seconds": {
   "Type": "Wait",
   "Seconds": 10,
   "Next": "NextState"
}
```

In the next example, the `Wait` state waits until an absolute time: March 14, 2016, at 1:59 PM UTC.

```
"wait_until": {
   "Type": "Wait",
   "Timestamp": "2016-03-14T01:59:00Z",
   "Next": "NextState"
}
```

You don't have to hard-code the wait duration. For example, given the following input data:

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

You can select the value of "expirydate" from the input using a reference path (p. 55) to select it from the input data.

```
"wait_until": {
}
**Succeed**

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

Because Succeed states are terminal states, they have no Next field, and don't need an End field, as shown in the following example.

```json
"SuccessState": {  
    "Type": "Succeed"
}
```

**Fail**

A Fail state ("Type": "Fail") stops the execution of the state machine and marks it as a failure, unless it is caught by a Catch block.

The Fail state only allows the use of Type and Comment fields from the set of common state fields (p. 27). 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 operational or diagnostic purposes.

Because Fail states always exit the state machine, they have no Next field and don't require an End field.

The following is an 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. 27), Parallel states include these additional fields.

- **Branches (Required)**
  An array of objects that specify state machines to execute in parallel. Each such state machine object must have fields named States and StartAt, whose meanings are exactly like those in the top level of a state machine.
**ResultPath (Optional)**

Specifies where (in the input) to place the output of the branches. The input is then filtered as specified by the `OutputPath` field (if present) before being used as the state's output. For more information, see Input and Output Processing (p. 55).

**ResultSelector (Optional)**

Pass a collection of key value pairs, where the values are static or selected from the result. For more information, see ResultSelector (p. 60).

**Retry (Optional)**

An array of objects, called Retriers, that define a retry policy in case the state encounters runtime errors. For more information, see Examples using Retry and using Catch (p. 93).

**Catch (Optional)**

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

**Parallel State Example**

```json
{
  "Comment": "Parallel Example.",
  "StartAt": "LookupCustomerInfo",
  "States": {
    "LookupCustomerInfo": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "LookupAddress",
          "States": {
            "LookupAddress": {
              "Type": "Task",
              "End": true
            }
          }
        },
        {
          "StartAt": "LookupPhone",
          "States": {
            "LookupPhone": {
              "Type": "Task",
              "End": true
            }
          }
        }
      ]
    }
  }
}
```
In this example, the `LookupAddress` and `LookupPhone` branches are executed in parallel. Here is how the visual workflow looks in the Step Functions console.

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

**Parallel State Input and Output Processing**

A `Parallel` state provides each branch with a copy of its own input data (subject to modification by the `InputPath` field). It generates output that is an array with one element for each branch, containing the output from that branch. There is no requirement that all elements be of the same type. The output array can be inserted into the input data (and the whole sent as the `Parallel` state's output) by using a `ResultPath` field in the usual way (see Input and Output Processing (p. 55)).

```json
{
  "Comment": "Parallel Example.",
  "StartAt": "FunWithMath",
  "States": {
```
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 the Add and Subtract tasks would be the sum of and difference between the array elements 3 and 2, which is 5 and 1, while the output of the Parallel state would be an array.

\([5, 1]\)

**Error Handling**

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

**Note**

When a parallel state fails, invoked Lambda functions continue to run and activity workers processing a task token are not stopped.

- To stop long-running activities, use heartbeats to detect if its branch has been stopped by Step Functions, and stop workers that are processing tasks. Calling \(\text{SendTaskHeartbeat}, \text{SendTaskSuccess},\) or \(\text{SendTaskFailure}\) will throw an error if the state has failed. See Heartbeat Errors.
- Running Lambda functions cannot be stopped. If you have implemented a fallback, use a Wait state so that cleanup work happens after the Lambda function has finished.

**Map**

The Map state ("Type": "Map") can be used to run a set of steps for each element of an input array. While the Parallel (p. 46) state executes multiple branches of steps using the same input, a Map state will execute the same steps for multiple entries of an array in the state input.
For an introduction to using a Map state, see the Map State Tutorial (p. 144).

In addition to the common state fields (p. 27), Map states include these additional fields.

**Iterator (Required)**

The Iterator field's value is an object that defines a state machine which will process each element of the array.

**ItemsPath (Optional)**

The ItemsPath field's value is a reference path identifying where in the effective input the array field is found. For more information, see ItemsPath (p. 61).

States within an Iterator field can only transition to each other, and no state outside the Iterator field can transition to a state within it.

If any iteration fails, entire Map state fails, and all iterations are terminated.

**MaxConcurrency (Optional)**

The MaxConcurrency field's value is an integer that provides an upper bound on how many invocations of the Iterator may run in parallel. For instance, a MaxConcurrency value of 10 will limit your Map state to 10 concurrent iterations running at one time.

*Note*

Concurrent iterations may be limited. When this occurs, some iterations will not begin until previous iterations have completed. The likelihood of this occurring increases when your input array has more than 40 items.

To achieve a higher number of concurrency, consider using nested state machines that cascade Map states. For example, to achieve a concurrency of 1024 you could build a state machine that contains a Map state that iterates 32 times, then nest that state machine into the Map state of a higher level state machine that iterates 32 times.

The default value is 0, which places no quota on parallelism and iterations are invoked as concurrently as possible.

A MaxConcurrency value of 1 invokes the Iterator once for each array element in the order of their appearance in the input, and will not start a new iteration until the previous has completed.

**ResultPath (Optional)**

Specifies where (in the input) to place the output of the branches. The input is then filtered as specified by the OutputPath field (if present) before being used as the state's output. For more information, see Input and Output Processing (p. 55).

**ResultSelector (Optional)**

Pass a collection of key value pairs, where the values are static or selected from the result. For more information, see ResultSelector (p. 60).

**Retry (Optional)**

An array of objects, called Retriers, that define a retry policy in case the state encounters runtime errors. For more information, see Examples using Retry and using Catch (p. 93).

**Catch (Optional)**

An array of objects, called Catchers, that define a fallback state that is executed if the state encounters runtime errors and its retry policy is exhausted or isn't defined. For more information, see Fallback States (p. 92).

**Map State Example**

Consider the following input data for a Map state.
{  
  "ship-date": "2016-03-14T01:59:00Z",
  "detail": {  
    "delivery-partner": "UQS",
    "shipped": [  
      { "prod": "R31", "dest-code": 9511, "quantity": 1344 },
      { "prod": "S39", "dest-code": 9511, "quantity": 40 },
      { "prod": "R31", "dest-code": 9833, "quantity": 12 },
      { "prod": "R40", "dest-code": 9860, "quantity": 887 },
      { "prod": "R40", "dest-code": 9511, "quantity": 1220 }
    ]
  }
}

Given the previous input, the Map state in the following example will invoke a AWS Lambda function (ship-val) once for each item of the array in the shipped field.

"Validate-All": {  
  "Type": "Map",
  "InputPath": ".detail",
  "ItemsPath": ".shipped",
  "MaxConcurrency": 0,
  "Iterator": {  
    "StartAt": "Validate",
    "States": {  
      "Validate": {  
        "Type": "Task",
        "End": true
      }
    },
    "ResultPath": ".detail.shipped",
    "End": true
  }
}

Each iteration of the Map state will send an item in the array (selected with the ItemsPath (p. 61) field) as input to the Lambda function. For instance, the input to one invocation of Lambda would be the following.

{  
  "prod": "R31",
  "dest-code": 9511,
  "quantity": 1344
}

When complete, the output of the Map state is a JSON array where each item is the output of an iteration (in this case, the output of the ship-val Lambda function).

**Map State Example with Parameters**

Suppose that the ship-val Lambda function in the previous example also needs information about the shipment's courier as well as the items in the array for each iteration. You can include information from the input, along with information specific to the current iteration of the map state. Note the Parameters field in the following example.

"Validate-All": {  
  "Type": "Map",
  "InputPath": ".detail",
  "ItemsPath": ".shipped",
  "Parameters": {  
    "ship-date": "2016-03-14T01:59:00Z",
    "delivery-partner": "UQS"
  }
}
The Parameters block replaces the input to the iterations with a JSON node that contains both the current item data from the context object (p. 71), and the courier information from the delivery-partner field from the Map state input. The following is an example of input to a single iteration, that is passed to an invocation of the ship-val Lambda function.

```json
{
  "parcel": {
    "prod": "R31",
    "dest-code": 9511,
    "quantity": 1344
  },
  "courier": "UQS"
}
```

In the previous Map state example, the ResultPath (p. 62) field produces output the same as the input, but with the detail.shipped field overwritten by an array in which each element is the output of the ship-val Lambda function for each iteration.

For more information see the following.

- Using a Map State to Call Lambda Multiple Times (p. 144)
- Input and Output Processing in Step Functions (p. 55)
- ItemsPath (p. 61)
- Context Object Data for Map States (p. 71)

### Map State Input and Output Processing

For a map state, InputPath (p. 58) works as it does for other state types, selecting a subset of the input.

The input of a Map state must include a JSON array, and it will run the Iterator section once for each item in the array. You specify where in the input to find this array using the ItemsPath (p. 61) field. If not specified, the value of ItemsPath is $, and the Iterator section expects that the array is the only input. A Map state may also include an ItemsPath (p. 61) field, whose value must be a Reference Path (p. 57). The ItemsPath field selects where in the input to find the array to use for iterations. The Reference Path is applied to the effective input (after InputPath is applied) and must identify a field whose value is a JSON array.

The input to each iteration, by default, is a single element of the array field identified by the ItemsPath value, This may be overridden using the Parameters (p. 58) field.
When complete, the output of the Map state is a JSON array, where each item is the output of an iteration.

For more information, see the following.

- Map State Tutorial (p. 144)
- Map State Example with Parameters (p. 51)
- Input and Output Processing in Step Functions (p. 55)
- Context Object Data for Map States (p. 71)
- Dynamically process data with a Map state (p. 379)

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.

```json
"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. 23) provides details about each of the state types you can specify, including information about how to specify transitions.

States can have multiple incoming transitions from other states.

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

The following rules apply to states within a state machine:

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

State Machine Data

State machine data takes the following forms:
Data Format

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

Note

- Numbers in JSON text format conform to JavaScript semantics. These numbers typically correspond to double-precision IEEE-854 values.
- The following is valid JSON text: standalone, quote-delimited strings; objects; arrays; numbers; Boolean values; and null.
- The output of a state becomes the input into the next state. However, you can restrict states to working on a subset of the input data by using Input and Output Processing (p. 55).

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.

For Standard Workflows, 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.

For Express Workflows, if you have enabled logging, you can retrieve results from CloudWatch Logs. See Logging using CloudWatch Logs (p. 508) for more information.

You should also consider quotas related to your state machine. For more information, see:

- Quotas (p. 485)

State Input/Output

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

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

1. Define the AWS Lambda function.
function Add(input) {
  var numbers = JSON.parse(input).numbers;
  var total = numbers.reduce(
    function(previousValue, currentValue, index, array) {
      return previousValue + currentValue;
    },
    0);
  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.

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

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

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

For more information, see Input and Output Processing in Step Functions (p. 55).

## Input and Output Processing in Step Functions

A Step Functions execution receives a JSON text as input and passes that input to the first state in the workflow. Individual states receive JSON as input and usually pass JSON as output to the next state. Understanding how this information flows from state to state, and learning how to filter and manipulate this data, is key to effectively designing and implementing workflows in AWS Step Functions.

In the Amazon States Language, these fields filter and control the flow of JSON from state to state:

- `InputPath`
- `OutputPath`
• ResultPath
• Parameters
• ResultSelector

The following diagram shows how JSON information moves through a task state. InputPath selects which parts of the JSON input to pass to the task of the Task state (for example, an AWS Lambda function). ResultPath then selects what combination of the state input and the task result to pass to the output. OutputPath can filter the JSON output to further limit the information that’s passed to the output.

InputPath, Parameters, ResultSelector, ResultPath, and OutputPath each manipulate JSON as it moves through each state in your workflow.

Each can use paths (p. 57) to select portions of the JSON from the input or the result. A path is a string, beginning with $, that identifies nodes within JSON text. Step Functions paths use JsonPath syntax.

Tip
Use the data flow simulator in the Step Functions console to test JSON path syntax, to better understand how data is manipulated within a state, and to see how data is passed between states.

Topics
• Paths (p. 57)
• InputPath, Parameters and ResultSelector (p. 57)
• ItemsPath (p. 61)
• ResultPath (p. 62)
• OutputPath (p. 68)
• InputPath, ResultPath and OutputPath Example (p. 68)
• Context Object (p. 70)

Paths

In the Amazon States Language, a _path_ is a string beginning with `#` that you can use to identify components within JSON text. Paths follow JsonPath syntax. You can specify a path to access subsets of the input when specifying values for InputPath, ResultPath, and OutputPath. For more information see Input and Output Processing in Step Functions (p. 55).

**Note**

You can also specify a JSON node of the input or the context object by using paths within the Parameters field of a state definition. See Pass Parameters to a Service API (p. 296).

Reference Paths

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

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

For example, if state input data contains the following values:

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

The following reference paths would return the following.

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

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

**InputPath, Parameters and ResultSelector**

The InputPath, Parameters and ResultSelector fields provide a way to manipulate JSON as it moves through your workflow. InputPath can limit the input that is passed by filtering the JSON notation by using a path (see Paths (p. 57)). The Parameters field enables you to pass a collection of key-value pairs, where the values are either static values that you define in your state machine definition, or that are selected from the input using a path. The ResultSelector field provides a way to manipulate the state’s result before ResultPath is applied.

AWS Step Functions applies the InputPath field first, and then the Parameters field. You can first filter your raw input to a selection you want using InputPath, and then apply Parameters to
manipulate that input further, or add new values. You can then use the ResultSelector field to manipulate the state's output before ResultPath is applied.

**Tip**
Use the data flow simulator in the Step Functions console to test JSON path syntax, to better understand how data is manipulated within a state, and to see how data is passed between states.

### InputPath

Use InputPath to select a portion of the state input.

For example, suppose the input to your state includes the following.

```json
{
    "comment": "Example for InputPath.",
    "dataset1": {
        "val1": 1,
        "val2": 2,
        "val3": 3
    },
    "dataset2": {
        "val1": "a",
        "val2": "b",
        "val3": "c"
    }
}
```

You could apply the InputPath.

"InputPath": ".dataset2",

With the previous InputPath, the following is the JSON that is passed as the input.

```json
{
    "val1": "a",
    "val2": "b",
    "val3": "c"
}
```

**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 ]
```

### Parameters

This section describes the different ways you can use the Parameters field.

**Key-value pairs**

Use the Parameters field to create a collection of key-value pairs that are passed as input. The values of each can either be static values that you include in your state machine definition, or selected from either
the input or the context object with a path. For key-value pairs where the value is selected using a path, the key name must end in \$. 

For example, suppose you provide the following input.

```json
{
   "comment": "Example for Parameters.",
   "product": {
      "details": {
         "color": "blue",
         "size": "small",
         "material": "cotton"
      },
      "availability": "in stock",
      "sku": "2317",
      "cost": "$23"
   }
}
```

To select some of the information, you could specify these parameters in your state machine definition.

```json
"Parameters": {
   "comment": "Selecting what I care about.",
   "MyDetails": {
      "size.$": ".product.details.size",
      "exists.$": ".product.availability",
      "StaticValue": "foo"
   }
},
```

Given the previous input and the Parameters field, this is the JSON that is passed.

```json
{
   "comment": "Selecting what I care about.",
   "MyDetails": {
      "size": "small",
      "exists": "in stock",
      "StaticValue": "foo"
   }
},
```

In addition to the input, you can access a special JSON object, known as the context object. The context object includes information about your state machine execution. See Context Object (p. 70).

**Connected resources**

The Parameters field can also pass information to connected resources. For example, if your task state is orchestrating an AWS Batch job, you can pass the relevant API parameters directly to the API actions of that service. For more information, see:

- Pass Parameters to a Service API (p. 296)
- Working with other services (p. 271)

**Amazon S3**

Alternatively, if the Lambda function data you are passing between states might grow to more than 262,144 bytes, we recommend using Amazon S3 to store the data, and parse the Amazon Resource Name (ARN) of the bucket in the Payload parameter to get the bucket name and key value.
Alternatively, you can adjust your implementation to pass smaller payloads in your executions. For more information, see Use Amazon S3 ARNs instead of passing large payloads (p. 267).

**ResultSelector**

Use the ResultSelector field to manipulate a state's result before ResultPath is applied. The ResultSelector field lets you create a collection of key value pairs, where the values are static or selected from the state's result. The output of ResultSelector replaces the state's result and is passed to ResultPath.

ResultSelector is an optional field in the following states:

- Map (p. 49)
- Task (p. 29)
- Parallel (p. 46)

For example, Step Functions service integrations return metadata in addition to the payload in the result. ResultSelector can select portions of the result and merge them with the state input with ResultPath. In this example, we want to select just the `resourceType` and `ClusterId`, and merge that with the state input from an Amazon EMR createCluster.sync. Given the following:

```json
{
    "resourceType": "elasticmapreduce",
    "resource": "createCluster.sync",
    "output": {
        "SdkHttpMetadata": {
            "HttpHeaders": {
                "Content-Length": "1112",
                "Content-Type": "application/x-amz-JSON-1.1",
                "Date": "Mon, 25 Nov 2019 19:41:29 GMT",
                "x-amzn-RequestId": "1234-5678-9012"
            },
            "HttpStatusCode": 200
        },
        "SdkResponseMetadata": {
            "RequestId": "1234-5678-9012"
        },
        "ClusterId": "AKIAIOSFODNN7EXAMPLE"
    }
}
```

You can then select the `resourceType` and `ClusterId` using ResultSelector:

```json
"Create Cluster": {
    "Type": "Task",
    "Resource": "arn:aws:states:::elasticmapreduce:createCluster.sync",
    "Parameters": { <some parameters> },
    "ResultSelector": {
        "ClusterId.$": "$!.output.ClusterId",
        "ResourceType.$": "$!.resourceType"
    },
    "ResultPath": "$!.EMROutput",
    "Next": "Next Step"
}
```

With the given input, using ResultSelector produces:
ItemsPath

The ItemsPath field is used in a Map state (p. 49) to select an array in the input. A Map state is used to iterate steps for each item in an array contained in the input. By default, a Map state sets ItemsPath to $ selecting the entire input. If the input to the Map state is a JSON array it will run an iteration for each item in the array, passing that item to the iteration as input. For an example of this, see the Map State Tutorial (p. 144).

The ItemsPath field allows you to specify a location in the input to find the JSON array to use for iterations. The value of ItemsPath must be a Reference Path (p. 57), and it must identify a value that is a JSON array. For instance, consider input to a Map state that includes two arrays, like the following example.

```
{
  "ThingsPiratesSay": [
    { "say": "Avast!" },
    { "say": "Yar!" },
    { "say": "Walk the Plank!" }
  ],
  "ThingsGiantsSay": [
    { "say": "Fee!" },
    { "say": "Fi!" },
    { "say": "Fo!" },
    { "say": "Fum!" }
  ]
}
```

In this case, you could specify which array to use for Map state iterations by selecting a specific array with ItemsPath. The following state machine definition specifies the ThingsPiratesSay array in the input using ItemsPath, and will run an iteration of the SayWord pass state for each item in the ThingsPiratesSay array.

```
{
  "StartAt": "PiratesSay",
  "States": {
    "PiratesSay": {
      "Type": "Map",
      "ItemsPath": "$ ThingsPiratesSay",
      "Next": "SayWord"
    }
  }
}
```
When processing input, ItemsPath is applied after InputPath (p. 58). It operates on the effective input to the state, after InputPath has filtered the input.

For more information on Map states, see the following:

- Map State (p. 49)
- Map (p. 49)
- Map State Example (p. 50)
- Using a Map State to Call Lambda Multiple Times (p. 144)
- Map State Input and Output Processing (p. 52)
- Dynamically process data with a Map state (p. 379)

**ResultPath**

The output of a state can be a copy of its input, the result it produces (for example, output from a Task state's Lambda function), or a combination of its input and result. Use ResultPath to control which combination of these is passed to the state output.

The following state types can generate a result and can include ResultPath:

- Pass (p. 28)
- Task (p. 29)
- Parallel (p. 46)

Use ResultPath to combine a task result with task input, or to select one of these. The path you provide to ResultPath controls what information passes to the output.

**Note**

ResultPath is limited to using reference paths (p. 57), which limit scope so that it can identify only a single node in JSON. See Reference Paths (p. 57) in the Amazon States Language (p. 23).

These examples are based on the state machine and Lambda function described in the Creating a Step Functions State Machine That Uses Lambda (p. 137) tutorial. Work through that tutorial and test different outputs by trying various paths in a ResultPath field.

**Use ResultPath to:**
- Use ResultPath to Replace the Input with the Result (p. 63)
- Discard the Result and Keep the Original Input (p. 64)
- Use ResultPath to Include the Result with the Input (p. 64)
• Use ResultPath to Update a Node in the Input with the Result (p. 66)
• Use ResultPath to Include Both Error and Input in a Catch (p. 67)

**Tip**
Use the data flow simulator in the Step Functions console to test JSON path syntax, to better understand how data is manipulated within a state, and to see how data is passed between states.

**Use ResultPath to Replace the Input with the Result**
If you don’t specify a ResultPath, the default behavior is as if you had specified "ResultPath": 
"$". Because this tells the state to replace the entire input with the result, the state input is completely replaced by the result coming from the task result.

The following diagram shows how ResultPath can completely replace the input with the result of the task.

Using the state machine and Lambda function described in Creating a Step Functions State Machine That Uses Lambda (p. 137), if we pass the following input:

```json
{
    "comment": "This is a test of the input and output of a Task state.",
    "details": "Default example",
    "who": "AWS Step Functions"
}
```

The Lambda function provides the following result.

"Hello, AWS Step Functions!"

If ResultPath isn’t specified in the state, or if "ResultPath": "$" is set, the input of the state is replaced by the result of the Lambda function, and the output of the state is the following.
"Hello, AWS Step Functions!"

**Note**
ResultPath is used to include content from the result with the input, before passing it to the output. But, if ResultPath isn't specified, the default is to replace the entire input.

**Discard the Result and Keep the Original Input**

If you set ResultPath to null, it will pass the original input to the output. Using "ResultPath": null, the state's input payload will be copied directly to the output, with no regard for the result.

The following diagram shows how a null ResultPath will copy the input directly to the output.

**Use ResultPath to Include the Result with the Input**

The following diagram shows how ResultPath can include the result with the input.
Using the state machine and Lambda function described in the Creating a Step Functions State Machine That Uses Lambda (p. 137) tutorial, we could pass the following input:

```json
{
    "comment": "This is a test of the input and output of a Task state.",
    "details": "Default example",
    "who": "AWS Step Functions"
}
```

The result of the Lambda function is the following.

"Hello, AWS Step Functions!"

To preserve the input, insert the result of the Lambda function, and then pass the combined JSON to the next state, we could set ResultPath to the following.

```
"ResultPath": "$\cdot taskresult"
```

This includes the result of the Lambda function with the original input.

```json
{
    "comment": "This is a test of input and output of a Task state.",
    "details": "Default behavior example",
    "who": "AWS Step Functions",
    "taskresult": "Hello, AWS Step Functions!"
}
```

The output of the Lambda function is appended to the original input as a value for taskresult. The input, including the newly inserted value, is passed to the next state.

You can also insert the result into a child node of the input. Set the ResultPath to the following.
"ResultPath": "$.strings.lambdaresult"

Start an execution using the following input.

```json
{
  "comment": "An input comment.",
  "strings": {
    "string1": "foo",
    "string2": "bar",
    "string3": "baz"
  },
  "who": "AWS Step Functions"
}
```

The result of the Lambda function is inserted as a child of the `strings` node in the input.

```json
{
  "comment": "An input comment.",
  "strings": {
    "string1": "foo",
    "string2": "bar",
    "string3": "baz",
    "lambdaresult": "Hello, AWS Step Functions!"
  },
  "who": "AWS Step Functions"
}
```

The state output now includes the original input JSON with the result as a child node.

**Use ResultPath to Update a Node in the Input with the Result**

The following diagram shows how `ResultPath` can update the value of existing JSON nodes in the input with values from the task result.
Using the example of the state machine and Lambda function described in the Creating a Step Functions State Machine That Uses Lambda (p. 137) tutorial, we could pass the following input.

```json
{
    "comment": "This is a test of the input and output of a Task state.",
    "details": "Default example",
    "who": "AWS Step Functions"
}
```

The result of the Lambda function is the following.

Hello, AWS Step Functions!

Instead of preserving the input and inserting the result as a new node in the JSON, we can overwrite an existing node.

For example, just as omitting or setting "ResultPath": "$" overwrites the entire node, you can specify an individual node to overwrite with the result.

"ResultPath": "$.comment"

Because the comment node already exists in the state input, setting ResultPath to "$.comment" replaces that node in the input with the result of the Lambda function. Without further filtering by OutputPath, the following is passed to the output.

```json
{
    "comment": "Hello, AWS Step Functions!",
    "details": "Default behavior example",
    "who": "AWS Step Functions"
}
```

The value for the comment node, "This is a test of the input and output of a Task state.", is replaced by the result of the Lambda function: "Hello, AWS Step Functions!" in the state output.

**Use ResultPath to Include Both Error and Input in a Catch**

The Handling Error Conditions Using a Step Functions State Machine (p. 140) tutorial shows how to use a state machine to catch an error. In some cases, you might want to preserve the original input with the error. Use ResultPath in a Catch to include the error with the original input, instead of replacing it.

"Catch": [{
    "ErrorEquals": ["States.ALL"],
    "Next": "NextTask",
    "ResultPath": "$.error"
}]

If the previous Catch statement catches an error, it includes the result in an error node within the state input. For example, with the following input:

```json
{"foo": "bar"}
```

The state output when catching an error is the following.

```json
{
    "foo": "bar",
```
For more information about error handling, see the following:

- Error handling in Step Functions (p. 88)
- Handling Error Conditions Using a Step Functions State Machine (p. 140)

OutputPath

OutputPath enables you to select a portion of the state output to pass to the next state. This enables you to filter out unwanted information, and pass only the portion of JSON that you care about.

If you don't specify an OutputPath the default value is $. This passes the entire JSON node (determined by the state input, the task result, and ResultPath) to the next state.

Tip

Use the data flow simulator in the Step Functions console to test JSON path syntax, to better understand how data is manipulated within a state, and to see how data is passed between states.

For more information, see the following:

- Paths in the Amazon States Language (p. 57)
- InputPath, ResultPath and OutputPath Example (p. 68)
- Pass Static JSON as Parameters (p. 297)
- Input and Output Processing in Step Functions (p. 55)

InputPath, ResultPath and OutputPath Example

Any state other than a Fail state can include InputPath, ResultPath or OutputPath. These allow you to use a path to filter the JSON as it moves through your workflow.

You can also use the Parameters field to manipulate JSON as it moves through your workflow. For information about using Parameters, see InputPath, Parameters and ResultSelector (p. 57).

For example, start with the AWS Lambda function and state machine described in the Creating a Step Functions State Machine That Uses Lambda (p. 137) tutorial. Modify the state machine so that it includes the following InputPath, ResultPath, and OutputPath.

```json
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "InputPath": "$.lambda",
      "ResultPath": "$.data.lambdaresult",
      "OutputPath": "$.data",
      "End": true
    }
  }
}
```
Start an execution using the following input.

```json
{
  "comment": "An input comment.",
  "data": {
    "val1": 23,
    "val2": 17
  },
  "extra": "foo",
  "lambda": {
    "who": "AWS Step Functions"
  }
}
```

Assume that the comment and extra nodes can be discarded, but that we want to include the output of the Lambda function, and preserve the information in the data node.

In the updated state machine, the Task state is altered to process the input to the task.

```
"InputPath": ".lambda",
```

This line in the state machine definition limits the task input to only the lambda node from the state input. The Lambda function receives only the JSON object {"who": "AWS Step Functions"} as input.

```
"ResultPath": ".data.lambdaresult",
```

This ResultPath tells the state machine to insert the result of the Lambda function into a node named lambdaresult, as a child of the data node in the original state machine input. Without further processing with OutputPath, the output of the state now includes the result of the Lambda function with the original input.

```json
{
  "comment": "An input comment.",
  "data": {
    "val1": 23,
    "val2": 17,
    "lambdaresult": "Hello, AWS Step Functions!"
  },
  "extra": "foo",
  "lambda": {
    "who": "AWS Step Functions"
  }
}
```

But, our goal was to preserve only the data node, and include the result of the Lambda function. OutputPath filters this combined JSON before passing it to the state output.

```
"OutputPath": ".data",
```

This selects only the data node from the original input (including the lambdaresult child inserted by ResultPath) to be passed to the output. The state output is filtered to the following.

```json
{
  "val1": 23,
  "val2": 17,
  "lambdaresult": "Hello, AWS Step Functions!"
}
```
In this Task state:

1. InputPath sends only the lambda node from the input to the Lambda function.
2. ResultPath inserts the result as a child of the data node in the original input.
3. OutputPath filters the state input (which now includes the result of the Lambda function) so that it passes only the data node to the state output.

For more information, see Input and Output Processing in Step Functions (p. 55).

Context Object

The context object is an internal JSON structure that is available during an execution, and contains information about your state machine and execution. This allows your workflows access to information about their specific execution. You can access the context object from the following fields:

- InputPath
- OutputPath
- ItemsPath (in Map states)
- Variable (in Choice states)
- ResultSelector
- Variable to variable comparison operators

Context Object Format

The context object includes information about the state machine, state, execution, and task. This JSON object includes nodes for each type of data, and is in the following format.

```json
{
    "Execution": {
        "Id": "String",
        "Input": {},
        "Name": "String",
        "RoleArn": "String",
        "StartTime": "Format: ISO 8601"
    },
    "State": {
        "EnteredTime": "Format: ISO 8601",
        "Name": "String",
        "RetryCount": Number
    },
    "StateMachine": {
        "Id": "String",
        "Name": "String"
    },
    "Task": {
        "Token": "String"
    }
}
```

During an execution, the context object is populated with relevant data for the Parameters field from where it is accessed. The value for a Task field is null if the Parameters field is outside of a task state.

Content from a running execution includes specifics in the following format.

```json
{
    "Execution": {
        ...
    },
    "State": {
        ...
    },
    "StateMachine": {
        ...
    },
    "Task": {
        ...
    }
}
```
"Execution": {
    "Id": "arn:aws:states:us-east-1:123456789012:execution:stateMachineName:executionName",
    "Input": {
        "key": "value"
    },
    "Name": "executionName",
    "RoleArn": "arn:aws:iam::123456789012:role...",
    "StartTime": "2019-03-26T20:14:13.192Z"
},
"State": {
    "Name": "Test",
    "RetryCount": 3
},
"StateMachine": {
    "Id": "arn:aws:states:us-east-1:123456789012:stateMachine:stateMachineName",
    "Name": "name"
},
"Task": {
    "Token": "h7XRiCdLtd/83p1E0dMccox1zFhglsdkzp9mBVKZsp7d9yrT1W"
}
}

**Note**

For context object data related to `Map` states, see [Context Object Data for Map States](p. 71).

### Accessing the Context Object

To access the context object, first specify the parameter name by appending `.#$` to the end, as you do when selecting state input with a path. Then, to access context object data instead of the input, prepend the path with `$$.`. This tells AWS Step Functions to use the path to select a node in the context object.

This example task state uses a path to retrieve and pass the execution Amazon Resource Name (ARN) to an Amazon Simple Queue Service (Amazon SQS) message.

```json
{
    "Order Flight Ticket Queue": {
        "Type": "Task",
        "Resource": "arn:aws:states:::sqs:sendMessage",
        "Parameters": {
            "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/flight-purchase",
            "MessageBody": {
                "From": "YVR",
                "To": "SEA",
                "Execution.$": "$$.$$.Execution.Id"
            }
        },
        "Next": "NEXT_STATE"
    }
}
```

**Note**

For more information about using the task token when calling an integrated service, see [Wait for a Callback with the Task Token](p. 293).

### Context Object Data for Map States

There are two additional items available in the context object when processing a `Map` state (p. 49): `Index` and `Value`. The `Index` contains the index number for the array item that is being processed in the current iteration. Within a `Map` state, the context object includes the following.
These are available only in a Map state, and can be specified in the Parameters (p. 58) field, before the Iterator section.

**Note**

You must define parameters from the context object in the Parameters block of the main Map state, not within the states included in the Iterator section.

Given a state machine with a simple Map state, we can inject information from the context object as follows.

```json
"Map": {
  "Item": {
    "Index": Number,
    "Value": "String"
  }
}
```

If you execute the previous state machine with the following input, Index and Value are inserted in the output.

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

The output for the execution is the following.

```json
[
  {"ContextValue": {"who": "bob"},
```
Executions in Step Functions

A state machine execution occurs when an AWS Step Functions state machine runs and performs its tasks. Each Step Functions state machine can have multiple simultaneous executions, which you can initiate from the Step Functions console, or by using the AWS SDKs, the Step Functions API actions, or the AWS Command Line Interface (AWS CLI). An execution receives JSON input and produces JSON output. You can start a Step Functions execution in the following ways:

- Call the StartExecution API action.
- Start a new execution (p. 12) in the Step Functions console.
- Use Amazon EventBridge to start an execution (p. 149) in response to an event or on a schedule.
- Start an execution with Amazon API Gateway (p. 152).
- Start a nested workflow execution (p. 73) from a Task state.

For more information about the different ways of working with Step Functions, see Development Options (p. 208).

Start Workflow Executions from a Task State

AWS Step Functions can start workflow executions directly from a Task state of a state machine. This allows you to break your workflows into smaller state machines, and to start executions of these other state machines. By starting these new workflow executions you can:

- Separate higher level workflow from lower level, task-specific workflows.
- Avoid repetitive elements by calling a separate state machine multiple times.
- Create a library of modular reusable workflows for faster development.
- Reduce complexity and make it easier to edit and troubleshoot state machines.

Step Functions can start these workflow executions by calling its own API as an integrated service (p. 271). Simply call the StartExecution API action from your Task state and pass the necessary parameters. You can call the Step Functions API using any of the service integration patterns (p. 291). To start a new execution of a state machine, use a Task state similar to the following.

```json
{
    "Type": "Task",
    "Resource": "arn:aws:states:::states:startExecution",
}```

This Task state will start a new execution of the HelloWorld state machine, and will pass the JSON comment as input.

**Note**
The StartExecution API action quotas can limit the number of executions that you can start. Use the Retry on StepFunctions.ExecutionLimitExceeded to ensure your execution is started. See the following:

- Quotas related to API action throttling (p. 487)
- Error handling in Step Functions (p. 88)

### Associate Workflow Executions

To associate a started workflow execution with the execution that started it, pass the execution ID from the context object (p. 70) to the execution input. You can access the ID from the context object from your Task state in a running execution. Pass the execution ID by appending \$. to the parameter name, and referencing the ID in the context object with `$.Execution.Id`.

```
"AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$.Execution.Id"
```

You can use a special parameter named `AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID` when you start an execution. If included, this association provides links in the Step details section of the Step Functions console. When provided, you can easily trace the executions of your workflows from starting executions to their started workflow executions. Using the previous example, associate the execution ID with the started execution of the HelloWorld state machine, as follows.

```
```

For more information, see the following:

- Working with other services (p. 271)
Viewing and debugging executions on the Step Functions console

The Execution Details page on the Step Functions console is a dashboard that details information about past and in-progress state machine executions for Standard Workflows. For example, you can find the state machine’s Amazon States Language definition, its execution status, ARN, and total number of state transitions. On this page, you can also view the execution details for any state in the state machine.

Contents

- Execution Details page – Interface overview (p. 75)
  - Execution summary (p. 76)
  - View mode (p. 77)
  - Step details (p. 82)
- Tutorial: Examining state machine executions using the Step Functions console (p. 83)
  - Step 1: Create and test the required Lambda functions (p. 83)
  - Step 2: Create and execute the state machine (p. 85)
  - Step 3: View the state machine execution details (p. 87)
  - Step 4: Explore the different View modes (p. 87)

Execution Details page – Interface overview

You can find the details of all your in-progress and past state machine executions for Standard Workflows on the Execution Details page. If you specified an execution ID while starting your execution, this page is titled with that execution ID. Otherwise, it’s titled with the unique execution ID that Step Functions automatically generates for you.

Besides the execution metrics, the Execution Details page provides the following options for managing your state machine’s execution:

<table>
<thead>
<tr>
<th>Button</th>
<th>Choose this button to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>New execution page</td>
<td>Toggle between the new and old layout of the Execution Details page.</td>
</tr>
<tr>
<td>Edit state machine</td>
<td>Edit your state machine’s Amazon States Language definition.</td>
</tr>
<tr>
<td>Export</td>
<td>Export the execution details in JSON format to share it with others or perform offline analysis.</td>
</tr>
<tr>
<td>New execution</td>
<td>Start a new execution of your state machine.</td>
</tr>
<tr>
<td>Stop execution</td>
<td>Stop the ongoing execution of your state machine.</td>
</tr>
</tbody>
</table>

The Execution Details console page contains the following sections:
Execution summary

The **Execution summary** section appears at the top of the **Execution Details** page. This section provides you with the high-level execution details of your workflow. This information is divided between the following three tabs:
Details

Shows information, such as the execution's status, ARN, and timestamps for execution start and end time. You can also view the total count of State transitions that occurred while executing your state machine. In addition, you can view the links for X-Ray trace map and Amazon CloudWatch Execution Logs if you enabled tracing or logs for your state machine.

If your state machine execution was triggered by another state machine, you can view the link for the parent state machine on this tab.

Execution input & output

Shows the state machine execution input and output side-by-side.

Definition

Shows the state machine's Amazon States Language definition.

Note

If your state machine execution failed, the Execution Details page displays an error message. Choose Cause or Show step detail on the error message to view the reason for execution failure or the step that caused the error.

If you choose Show step detail, the Step detail (p. 82) section displays the Input & Output tab for the step that caused the error. Additionally, the erroneous step appears highlighted in the Graph view and Table view.

### View mode

The View mode section contains three different visualizations for your state machine. You can choose to view a graphic representation of the workflow, a table outlining the states in your workflow, or a list of the events associated with your state machine's execution:

**Note**
Choose a tab to view its contents.
Graph view

The **Graph view** mode displays a graphical representation of your workflow. A legend is included at the bottom that indicates the execution status of the state machine. It also contains buttons that let you zoom in, zoom out, center the full workflow, or view the workflow in full-screen mode.

From this view, you can choose any step in your workflow to view details about its execution in the **Step details (p. 82)** component. Further, the step that you choose in the **Graph view** gets selected in the **Table view**, as well as the other way around.

If your state machine contains a **Map** state, **Parallel** state, or both, you can view their names in the workflow in the **Graph view**. In addition, for the **Map** state, the **Graph view** lets you move across different iterations of the **Map** state execution data. For example, if your **Map** state has five iterations and you want to view the execution data for the third and fourth iterations, do the following:

1. Choose the **Map** state whose iteration data you want to view.
2. From **Map iteration viewer**, choose #2 from the dropdown list for third iteration because iterations are counted from zero. Similarly, choose #3 from the dropdown list for the fourth iteration.

   Alternatively, use the ▲ and ▼ arrow keys to move across different iterations of the **Map** state.
Note
If your state machine contains nested `Map` states, the dropdown lists for the parent and child `Map` state iterations will be displayed as shown in the following example:

<table>
<thead>
<tr>
<th>▼ Map iteration viewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoopOverStores #1 ▼ ▲ ▲ &gt; LoopOverFruits #2 ▼ ▲</td>
</tr>
</tbody>
</table>

3. (Optional) If one or more of your `Map` state iterations failed to execute, or the execution was stopped, you can view their data by choosing those iteration numbers under *Failed* or *Aborted* in the dropdown list.

Finally, you can use the **Export** and **Layout** buttons to export the workflow graph as an SVG or PNG image or switch between horizontal and vertical views of your workflow.

**Table view**

The **Table view** mode displays a tabular representation of the states in your workflow. In this **View mode**, you can see the details of each state that was executed in your workflow, including its name, the name of any resource it used (such as an AWS Lambda function), and if the state executed successfully.

From this view, you can choose any state in your workflow to view details about its execution in the **Step details (p. 82)** component. Further, the state that you choose in the **Table view** gets selected in the **Graph view**, as well as the other way around.

You can also limit the amount of data displayed in the **Table view** mode by applying filters to the view. You can create a filter for a specific property, such as execution status, or for a date and time range. For more information, see Tutorial: Examining state machine executions using the Step Functions console (p. 83).
### Table view

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Status</th>
<th>Resource</th>
<th>Duration</th>
<th>Timeline</th>
<th>Started After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>Parallel</td>
<td>✔️ Succeeded</td>
<td></td>
<td>32 sec</td>
<td></td>
<td>35 ms</td>
</tr>
<tr>
<td>#1</td>
<td>ParallelBranch</td>
<td>✔️ Succeeded</td>
<td></td>
<td>32 sec</td>
<td></td>
<td>147 ms</td>
</tr>
<tr>
<td>Lambda</td>
<td>Task</td>
<td>✔️ Succeeded</td>
<td>Lambda</td>
<td>31 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait</td>
<td>Wait</td>
<td>✔️ Succeeded</td>
<td></td>
<td>1 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Choice</td>
<td>Choice</td>
<td>✔️ Succeeded</td>
<td></td>
<td>0 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass</td>
<td>Pass</td>
<td>✔️ Succeeded</td>
<td></td>
<td>0 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0</td>
<td>ParallelBranch</td>
<td>✔️ Succeeded</td>
<td></td>
<td>9 sec</td>
<td></td>
<td>156 ms</td>
</tr>
<tr>
<td>Map</td>
<td>Map</td>
<td>✔️ Succeeded</td>
<td></td>
<td>134 ms</td>
<td></td>
<td>156 ms</td>
</tr>
<tr>
<td>#0</td>
<td>MapIteration</td>
<td>✔️ Succeeded</td>
<td></td>
<td>103 ms</td>
<td></td>
<td>156 ms</td>
</tr>
<tr>
<td>#1</td>
<td>MapIteration</td>
<td>✔️ Succeeded</td>
<td></td>
<td>113 ms</td>
<td></td>
<td>156 ms</td>
</tr>
<tr>
<td>#2</td>
<td>MapIteration</td>
<td>✔️ Succeeded</td>
<td></td>
<td>122 ms</td>
<td></td>
<td>156 ms</td>
</tr>
<tr>
<td>#3</td>
<td>MapIteration</td>
<td>✔️ Succeeded</td>
<td></td>
<td>134 ms</td>
<td></td>
<td>156 ms</td>
</tr>
<tr>
<td>Pass</td>
<td>Pass</td>
<td>✔️ Succeeded</td>
<td></td>
<td>0 ms</td>
<td></td>
<td>290 ms</td>
</tr>
<tr>
<td>FailAct</td>
<td>Parallel</td>
<td>▶️ Caught error</td>
<td></td>
<td>5 sec</td>
<td></td>
<td>302 ms</td>
</tr>
<tr>
<td>#0</td>
<td>ParallelBranch</td>
<td>▶️ Caught error</td>
<td></td>
<td>0 ms</td>
<td></td>
<td>405 ms</td>
</tr>
<tr>
<td>Task</td>
<td></td>
<td>✖️ Aborted</td>
<td></td>
<td>32 sec</td>
<td></td>
<td>405 ms</td>
</tr>
<tr>
<td>#1</td>
<td>ParallelBranch</td>
<td>▶️ Caught error</td>
<td></td>
<td>0 ms</td>
<td></td>
<td>419 ms</td>
</tr>
</tbody>
</table>

By default, this mode displays the **Name**, **Type**, **Status**, **Resource**, and **Started After** columns. However, you can configure the columns you want to view using the **Preferences** dialog box. The selections that you make on this dialog box persist for future state machine executions.

If you add the **Timeline** column, the execution duration of each state is shown with respect to the runtime for the entire execution. This is displayed as a color-coded, linear timeline. This can help you identify any performance-related issues with a specific state's execution. The color-coded segments for each state on the timeline help you identify the state's execution status, such as in-progress, failed, or aborted.

For example, if you've defined execution retries for a state in your state machine, these retries are shown in the timeline. Red segments represent the failed **Retry** attempts, while light gray segments represent the **BackoffRate** between each **Retry** attempt.
If your state machine contains a Map state, Parallel state, or both, you can view their names in the workflow in Table view. For Map and Parallel states, the Table view mode displays the execution data for their iterations and parallel branches as nodes inside a tree view. You can choose each node in these states to view their individual details in the Step details (p. 82) section. For example, you can review the data for a specific Map state iteration that caused the state to fail. Expand the node for the Map state, and then view the status for each iteration in the Status column.

Event view

The Event view mode displays the complete history for the selected execution as a list of events spanning multiple pages. Each page contains up to 25 events. This view also displays the total event count, which can help you determine if you exceeded the maximum count of 25,000 events.
By default, the results in the Event view mode are displayed in ascending order based on the Timestamp of the events. You can change the execution event history's sorting to descending order by clicking on the Timestamp column header.

In the Event view mode, each event is color-coded to indicate its execution status. For example, events that failed are colored red. To view additional details about an event, choose the ▶ next to the event ID. Once open, the event details show the input, output, and resource invocation for the event.

In addition, in the Event view mode, you can apply filters to a property to limit the execution event history results that are displayed. You can choose properties such as execution ID, or a date and time range. For more information, see Tutorial: Examining state machine executions using the Step Functions console (p. 83).

**Step details**

The Step details section opens up on the right when you choose a state in the Graph view or Table view. This section contains the following tabs, which provide you in-depth information about the selected state:

**Input & Output**

Shows the input and output details of the selected state. If there was an error in the input or output, it is indicated with a △ on the tab header. In addition, you can view the reason for the error in the Reason box.

You can choose the Advanced view toggle button to see the input and output data transfer path as the data passed through the selected state. This lets you identify how your input was processed as one or more of the fields, such as InputPath, Parameters, ResultSelector, OutputPath, and ResultPath, were applied to the data.

**Details**

Shows information, such as the state type, its execution status, and execution duration.

For Task states that use a resource, such as AWS Lambda, this tab provides links to the resource definition page and Amazon CloudWatch logs page for the resource invocation. It also shows values, if specified, for the Task state's TimeoutSeconds and HeartbeatSeconds fields.

For Map states, this tab shows you information regarding the total count of a Map state's iterations. Iterations are categorized as Failed, Aborted, Succeeded, or InProgress.

**Definition**

Shows the Amazon States Language definition corresponding to the selected state.

**Retry**

*Note*

This tab appears only if you've defined a Retry field in your state machine's Task or Parallel state.

Shows the initial and subsequent retry attempts of the selected state. For the Initial attempt and all the subsequent failed attempts, choose the ▶ next to the Status to view the Reason for failure that appears in a dropdown box. If the retry attempt succeeds, you can view the Output that appears in a dropdown box.

**Events**

Shows a filtered list of the events associated with the selected state in an execution. The information you see on this tab is a subset of the complete execution event history you see in the Event view (p. 81) mode.
Tutorial: Examining state machine executions using the Step Functions console

In this tutorial, you will learn how to inspect the execution information displayed on the Execution Details page and view the reason for a failed execution. Then, you'll learn how to access different iterations of a Map state execution. Finally, you'll learn how to configure the columns to display on the Table view and apply suitable filters to view only the information of interest.

The state machine used in this tutorial obtains the price of a set of fruits. To do this, the state machine uses three AWS Lambda functions which return a random list of four fruits, the price of each fruit, and the average cost of the fruits. The Lambda functions are defined to throw an error if the price of the fruits is less than or equal to a threshold value.

Contents

- Step 1: Create and test the required Lambda functions (p. 83)
- Step 2: Create and execute the state machine (p. 85)
- Step 3: View the state machine execution details (p. 87)
- Step 4: Explore the different View modes (p. 87)

Step 1: Create and test the required Lambda functions

1. Open the Lambda console and then perform steps 1 through 4 in the Step 1: Create a Lambda Function (p. 138) section. Make sure to name the Lambda function GetListOfFruits.
2. After your Lambda function is created, copy the function’s Amazon Resource Name (ARN) displayed in the upper-right corner of the page. To copy the ARN, click the icon. The following is an example ARN:

```text
arn:aws:lambda:us-east-1:123456789012:function:function-name
```
3. Copy the following code for the Lambda function into the Code source area of the GetListOfFruits page.

```javascript
function getRandomSubarray(arr, size) {
  var shuffled = arr.slice(0), i = arr.length, temp, index;
  while (i--) {
    index = Math.floor((i + 1) * Math.random());
    temp = shuffled[index];
    shuffled[index] = shuffled[i];
    shuffled[i] = temp;
  }
  return shuffled.slice(0, size);
}
exports.handler = async function(event, context) {
  const fruits = ['Abiu','Açaí','Acerola','Ackee','African
cucumber','Apple','Apricot','Avocado','Banana','Bilberry','Blackberry','Blackcurrant','Jostaberry','

  const errorChance = 45;
  const waitTime = Math.floor( 100 * Math.random() );
  await new Promise( r => setTimeout(() => r(), waitTime));
  const num = Math.floor( 100 * Math.random() );
  // const num = 51;
```

83
4. Choose **Deploy** and then choose **Test** to deploy the changes and see the output of your Lambda function.

5. Create the remaining two Lambda functions named **GetFruitPrice** and **CalculateAverage** with the following steps:

a. Copy the following code into the **Code source** area of the **GetFruitPrice** Lambda function:

```javascript
exports.handler = async function(event, context) {
    const errorChance = 0;
    const waitTime = Math.floor( 100 * Math.random() );
    await new Promise( r => setTimeout(() => r(), waitTime));
    const num = Math.floor( 100 * Math.random() );
    if (num <= errorChance) {
        throw(new Error('Error'));
    }
    return Math.floor(Math.random()*100)/10;
};
```

b. Copy the following code into the **Code source** area of the **CalculateAverage** Lambda function:

```javascript
function getRandomSubarray(arr, size) {
    var shuffled = arr.slice(0), i = arr.length, temp, index;
    while (i--) {
        index = Math.floor((i + 1) * Math.random());
        temp = shuffled[index];
        shuffled[index] = shuffled[i];
        shuffled[i] = temp;
    }
    return shuffled.slice(0, size);
}

const average = arr => arr.reduce( ( p, c ) => p + c, 0 ) / arr.length;
exports.handler = async function(event, context) {
    const errors = [
        "Error getting data from DynamoDB",
        "Error connecting to DynamoDB",
        "Network error",
        "MemoryError - Low memory"
    ]
    const errorChance = 0;
    const waitTime = Math.floor( 100 * Math.random() );
    await new Promise( r => setTimeout(() => r(), waitTime));
    const num = Math.floor( 100 * Math.random() );
    if (num <= errorChance) {
        throw(new Error(getRandomSubarray(errors, 1)[0]));
    }
};
```
Step 2: Create and execute the state machine

Use the Step Functions console to create a state machine that invokes the Lambda functions you created in Step 1 (p. 83). In this state machine, three Map states are defined. Each of these Map states contains a Task state to invoke each of the Lambda functions. Additionally, a Retry field is defined in each Task state with a number of retry attempts defined for each state. If a Task state encounters a runtime error, it's executed again up to the defined number of retry attempts for that Task.

1. Open the Step Functions console and choose Write your workflow in code.
   Important
   Ensure that your state machine is under the same AWS account and Region as the Lambda function you created earlier.
2. For Type, keep the default selection of Standard.
3. Copy the following Amazon States Language definition and paste it under Definition. Make sure to replace the ARNs shown with those of the Lambda functions that you previously created.
   ```json
   {
     "StartAt": "LoopOverStores",
     "States": {
       "LoopOverStores": {
         "Type": "Map",
         "Iterator": {
           "StartAt": "GetListOfFruits",
           "States": {
             "GetListOfFruits": {
               "Type": "Task",
               "Resource": "arn:aws:states:::lambda:invoke",
               "OutputPath": "$Payload",
               "Parameters": {
                 "Payload": {
                   "storeName.$": "$"
                 }
               },
               "Retry": [
                 {
                   "ErrorEquals": ["States.ALL"],
                   "IntervalSeconds": 2,
                   "MaxAttempts": 1,
                   "BackoffRate": 1.3
                 }
               ],
               "Next": "LoopOverFruits"
             }
           },
           "LoopOverFruits": {
             "Type": "Map",
             "Iterator": {
               "StartAt": "GetFruitPrice",
               "States": {
                 "GetFruitPrice": {
                   "Type": "Task",
                   "Resource": "arn:aws:states:::lambda:invoke",
                   "OutputPath": "$Payload",
                 }
               },
               "Next": "LoopOverFruits"
             }
           }
         }
       }
     }
   }
   ```
"Parameters": {
  "FunctionName": "arn:aws:lambda:us-east-1:123456789012:function:GetFruitPrice:$LATEST",
  "Payload": {
    "fruitName.$": "$"
  }
},
"Retry": [
  {
    "ErrorEquals": [
      "States.ALL"
    ],
    "IntervalSeconds": 2,
    "MaxAttempts": 3,
    "BackoffRate": 1.3
  }
],
"End": true
},
"ItemsPath": "$",
"End": true
},
"ItemsPath": ".stores",
"Next": "LoopOverStoreFruitsPrice",
"ResultPath": ".storesFruitsPrice"
},
"LoopOverStoreFruitsPrice": {
  "Type": "Map",
  "End": true,
  "Iterator": {
    "StartAt": "CalculateAverage",
    "States": {
      "CalculateAverage": {
        "Type": "Task",
        "Resource": "arn:aws:states:::lambda:invoke",
        "OutputPath": ".Payload",
        "Parameters": {
          "FunctionName": "arn:aws:lambda:us-east-1:123456789012:function:Calculate-average:$LATEST",
          "Payload.$": "$
        }
      },
      "Retry": [
        {
          "ErrorEquals": [
            "States.ALL"
          ],
          "IntervalSeconds": 2,
          "MaxAttempts": 2,
          "BackoffRate": 1.3
        }
      ],
      "End": true
    }
  }
},
"ItemsPath": ".storesFruitsPrice",
"ResultPath": ".storesPriceAverage",
"MaxConcurrency": 1
}
4. Enter a name for your state machine. Keep the default selections for the other options on the page, and then choose Create state machine.

5. Open the page titled with your state machine name. Perform steps 1 through 4 in the Step 4: Start a New Execution (p. 140) section, but use the following data as the execution input:

```json
{
    "stores": [
        "Store A",
        "Store B",
        "Store C",
        "Store D"
    ]
}
```

Step 3: View the state machine execution details

On the page titled with your execution ID, you can review the results of your execution and debug any errors.

1. (Optional) Choose from the tabs displayed on the Execution Details page to see the information present in each of them. For example, to view the state machine input and its execution output, choose Execution input & output on the Execution summary (p. 76) section.

2. If your state machine execution failed, choose Cause or Show step detail on the error message. Details about the error are displayed in the Step details (p. 82) section. Notice that the affected step, which is a Task state named GetListofFruits, that caused the error appears highlighted in the Graph view and Table view.

   Note
   Because the GetListofFruits step is defined inside a Map state, and the step failed to execute successfully, the Status of Map state step is displayed as Failed.

Step 4: Explore the different View modes

You can choose a preferred mode to view the state machine workflow or the execution event history. Some of the tasks that you can perform in these View modes are as follows:

**Graph view – Switch between different Map state iterations**

If your Map state has five iterations and you want to view the execution details for the iteration number three and four respectively, do the following:

1. Choose the Map state for which you want to view the different iteration data.

2. From Map iteration viewer, choose the iteration you want to view. Iterations are counted from zero. To choose the third iteration out of five, choose #2 from the dropdown list next to the Map state's name.

   Note
   If your state machine contains nested Map states, the dropdown lists for the parent and child Map state iterations will be displayed as following:

   ▼ Map iteration viewer

   | LoopOverStores | #1 ▼ ▲ | ▶ | LoopOverFruits | #2 ▼ ▲ |

87
3. (Optional) If one or more of your Map state iterations failed to execute, or the iteration stopped in an aborted state, you can view details about the failed iteration. To see these details, choose the affected iteration numbers under Failed or Aborted in the dropdown list.

**Table view – Switch between different Map state iterations**

If your Map state has five iterations and you want to view the execution details for the iteration number three and four, do the following:

1. Choose the Map state for which you want to view the different iteration data.
2. In the tree view display of the Map state iterations, choose the row for iteration named #2 for the iteration number three. Similarly, choose the row named #3 for the iteration number four.

**Table view – Configure the columns to display**

Choose . Then, in the Preferences dialog box, choose the columns you want to display under Select visible columns.

By default, this mode displays the Name, Type, Status, Resource, and Started After columns.

**Table view – Filter the results**

Limit the amount of information displayed by applying one or more filters based on a property, such as Status, or a date and time range. For example, to view the steps that failed execution, apply the following filter:

1. Choose Filter by properties or search by keyword, and then choose Status under Properties.
2. Under Operators, choose Status =.
3. Choose Status = Failed.
4. (Optional) Choose Clear filters to remove the applied filters.

**Event view – Filter the results**

Limit the amount of information displayed by applying one or more filters based on a property, such as Type, or a date and time range. For example, to view the Task state steps that failed execution, apply the following filter:

1. Choose Filter by properties or search by keyword, and then choose Type under Properties.
2. Under Operators, choose Type =.
3. Choose Type = TaskFailed.
4. (Optional) Choose Clear filters to remove the applied filters.

**Event view – Inspect a TaskFailed event detail**

Choose the next to the ID of a TaskFailed event to inspect its details, including input, output, and resource invocation that appear in a dropdown box.

---

**Error handling in Step Functions**

Any state can encounter runtime errors. Errors can happen for various reasons:
- State machine definition issues (for example, no matching rule in a Choice state)
- Task failures (for example, an exception in a Lambda function)
- Transient issues (for example, network partition events)

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

## Error names

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

### States.ALL

A wildcard that matches any known error name.

**Note**

This error type can't catch the States.DataLimitExceeded terminal error type and runtime error types. For more information about these error types, see States.DataLimitExceeded (p. 89) and States.Runtime (p. 90).

### States.BranchFailed

A branch of a Parallel state failed.

### States.DataLimitExceeded

A States.DataLimitExceeded exception will be thrown for the following:

- When the output of a connector is larger than payload size quota.
- When the output of a state is larger than payload size quota.
- When, after Parameters processing, the input of a state is larger than the payload size quota.

For more information on quotas, see Quotas (p. 485).

**Note**

This is a terminal error that can't be caught by the States.ALL error type.

### States.HeartbeatTimeout

A Task state failed to send a heartbeat for a period longer than the HeartbeatSeconds value.

**Note**

This error is only available inside the Catch and Retry fields.

### States.IntrinsicFailure

This error occurs when the attempt to invoke an intrinsic function within a payload template fails.

### States.NoChoiceMatched

This runtime error occurs if a Choice state fails to match the input with the conditions defined in the Choice Rule and no Default transition is specified.

### States.ParameterPathFailure

This error occurs when within a state's Parameters field, an attempt to replace a field whose name ends in .$ using a path fails.

### States.Permissions

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

A state's ResultPath field cannot be applied to the input the state received.

States.Runtime

An execution failed due to some exception that could not be processed. Often these are caused by errors at runtime, such as attempting to apply InputPath or OutputPath on a null JSON payload. A States.Runtime error is not retriable, and will always cause the execution to fail. A retry or catch on States.ALL will not catch States.Runtime errors.

States.TaskFailed

A Task state failed during the execution. When used in a retry or catch, States.TaskFailed acts as a wildcard that matches any known error name except for States.Timeout.

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

As a best practice, ensure production code can handle AWS Lambda service exceptions (Lambda.ServiceException and Lambda.SdkClientException). For more information, see Handle Lambda service exceptions (p. 268).

Note

Unhandled errors in Lambda are reported as Lambda.Unknown in the error output. These include out-of-memory errors and function timeouts. You can match on Lambda.Unknown, States.ALL, or States.TaskFailed to handle these errors. When Lambda hits the maximum number of invocations, the error is Lambda.TooManyRequestsException. For more information about Lambda Handled and Unhandled errors, see FunctionError in the AWS Lambda Developer Guide.

Retrying after an error

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

Note

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

A retrier contains the following fields.

ErrorEquals (Required)

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

IntervalSeconds (Optional)

An integer that represents the number of seconds before the first retry attempt (1 by default). IntervalSeconds has a maximum value of 999999999.

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. \texttt{MaxAttempts} has a maximum value of 99999999.

\textbf{BackoffRate (Optional)}

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

This example of a \texttt{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 \texttt{States.ALL} that appears in a retrier's \texttt{ErrorEquals} field is a wildcard that matches any error name. It must appear alone in the \texttt{ErrorEquals} array and must appear in the last retrier in the \texttt{Retry} array. The name \texttt{States.TaskFailed} also acts a wildcard and matches any error except for \texttt{States.Timeout}.

This example of a \texttt{Retry} field retries any error except \texttt{States.Timeout}.

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

\section*{Complex retry scenarios}

A retrier's parameters apply across all visits to the retrier in the context of a single-state execution.

Consider the following Task state.

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

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

- The first two errors match the first retrier and cause waits of 1 and 2 seconds.
- The third error matches the second retrier and causes a wait of 5 seconds.
• The fourth error matches the first retrier and causes a wait of 4 seconds.
• The fourth error also matches the first retrier. However, it has already reached its maximum of two retries (MaxAttempts) for that particular error (ErrorB), so that retrier fails and the execution is redirected to the Z state through the Catch field.
• The fifth error also matches the first retrier. However, it has already reached its maximum 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, Map and Parallel states can have a field named Catch. This field's value must be an array of objects, known as catchers.

A catcher contains the following fields.

**ErrorEquals (Required)**

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

**Next (Required)**

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

**ResultPath (Optional)**

A path (p. 55) 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 name States.TaskFailed also acts a wildcard and matches any error except for States.Timeout.

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.

```json
"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 \texttt{ResultPath} field. This works similarly to a \texttt{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 \texttt{error-info} (if there isn't already a field with this name in the input). Then, the entire input is sent to \texttt{RecoveryState}. For the second catcher, the error output overwrites the input and only the error output is sent to \texttt{EndState}.

\textbf{Note}
If you don't specify the \texttt{ResultPath} field, it defaults to $\$, which selects and overwrites the entire input.

When a state has both \texttt{Retry} and \texttt{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.

\section*{Cause payloads and service integrations}

A catcher returns a string payload as an output. When working with service integrations such as Amazon Athena or AWS CodeBuild, you may want to convert the \texttt{Cause} string to JSON. The following example of a Pass state with intrinsic functions shows how to convert a \texttt{Cause} string to JSON.

```
"Handle escaped JSON with JSONtoString": {
  "Type": "Pass",
  "Parameters": {
    "Cause.$": "States.StringToJson($$.Cause)"
  },
  "Next": "Pass State with Pass Processing"
},
```

\section*{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 Node.js Lambda function that always fails, returning the message \texttt{error}. In the state machine examples that follow, this Lambda function is named \texttt{FailFunction}. For information about creating a Lambda function, see Step 1: Create a Lambda Function (p. 138) section.

```javascript
exports.handler = (event, context, callback) => {
  callback("error");
};
```

This is a definition of a Node.js Lambda function that sleeps for 10 seconds. In the state machine examples that follow, this Lambda function is named \texttt{sleep10}.

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

```javascript
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.

```json
{
  "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.

```json
{
  "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
    }
  }
}
```

**Note**

As a best practice, tasks that reference a Lambda function should handle Lambda service exceptions. For more information, see Handle Lambda service exceptions (p. 268).

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
{
```

94
"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.

{
  "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 Task state that times out based on the timeout value specified in TimeoutSeconds. The Lambda function invocation in this Task state 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",
      "TimeoutSeconds": 2,
      "Retry": [ {
        "Type": "Task",
        "Catch": [ {
          "ErrorEquals": ["States.TaskFailed"],
          "Next": "fallback"
        } ],
        "End": true
      },
      "fallback": {
        "Type": "Pass",
        "Result": "Hello, AWS Step Functions!",
        "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.

```json
{
  "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
    }
  }
}
```

**Note**

You can preserve the state input and the error by using ResultPath. See Use ResultPath to Include Both Error and Input in a Catch (p. 67).

Invoke AWS Step Functions from other services

You can configure several other services to invoke state machines. You can only invoke state machines asynchronously. With asynchronous invocation, Step Functions returns a response immediately. Services that can be configured to invoke Step Functions include:

- AWS Lambda, using the StartExecution call.
- Amazon API Gateway
- Amazon EventBridge
- AWS CodePipeline
- AWS IoT Rules Engine
- AWS Step Functions

Step Functions invocations are governed by the StartExecution quota. For more information, see:
Read Consistency in Step Functions

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

For more information, see the following:

- \texttt{UpdateStateMachine} in the \textit{AWS Step Functions API Reference}
- Step 3: Update a state machine (p. 12) in Getting started with AWS Step Functions (p. 10)

Tagging in Step Functions

AWS Step Functions supports tagging of state machines (both Standard and Express) and activities. This can help you track and manage the costs associated with your resources, and provide better security in your AWS Identity and Access Management (IAM) policies. Tagging Step Functions resources allows them to be managed by AWS Resource Groups. For more information on Resource Groups, see the \textit{AWS Resource Groups User Guide}.

To review the restrictions related to resource tagging, see Restrictions related to tagging (p. 489).

Topics

- Tagging for Cost Allocation (p. 97)
- Tagging for Security (p. 98)
- Viewing and Managing Tags in the Step Functions Console (p. 98)
- Manage Tags with Step Functions API Actions (p. 99)

Tagging for Cost Allocation

To organize and identify your Step Functions resources for cost allocation, you can add metadata \texttt{tags} that identify the purpose of a state machine or activity. This is especially useful when you have many resources. You can use cost allocation tags to organize your AWS bill to reflect your own cost structure.

To do this, sign up to get your AWS account bill to include the tag keys and values. For more information, see Setting Up a Monthly Cost Allocation Report in the \textit{AWS Billing User Guide}.

For example, you could add tags that represent the cost center and purpose of your Step Functions resources, as follows.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateMachine1</td>
<td>Cost Center</td>
<td>34567</td>
</tr>
<tr>
<td></td>
<td>Application</td>
<td>Image processing</td>
</tr>
<tr>
<td>StateMachine2</td>
<td>Cost Center</td>
<td>34567</td>
</tr>
<tr>
<td></td>
<td>Application</td>
<td>Rekognition processing</td>
</tr>
<tr>
<td>Activity1</td>
<td>Cost Center</td>
<td>12345</td>
</tr>
</tbody>
</table>
This tagging scheme allows you to group two state machines performing related tasks in the same cost center, while tagging an unrelated activity with a different cost allocation tag.

**Tagging for Security**

IAM supports controlling access to resources based on tags. To control access based on tags, provide information about your resource tags in the condition element of an IAM policy.

For example, you could restrict access to all Step Functions resources that include a tag with the key environment and the value production.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "states:TagResource",
                "states:DeleteActivity",
                "states:DeleteStateMachine",
                "states:StopExecution"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/environment": "production"
                }
            }
        }
    ]
}
```

For more information, see Controlling Access Using Tags in the IAM User Guide.

**Viewing and Managing Tags in the Step Functions Console**

Step Functions allows you to view and manage the tags for your state machines in the Step Functions console. From the Details page of a state machine, select Tags. Here, you can view the existing tags associated with your state machine.

**Note**

To manage tags for activities, see Manage Tags with Step Functions API Actions (p. 99).

To add or delete tags that are associated with your state machine, select the Manage Tags button.

1. Browse to the details page of a state machine.
2. Select Tags, next to Executions and Definition.
3. Choose Manage tags.
   - To modify existing tags, edit the Key and Value.
   - To remove existing tags, choose Remove tag.
   - To add a new tag, choose Add tag and enter a Key and Value.
4. Choose Save.
Manage Tags with Step Functions API Actions

To manage tags using the Step Functions API, use the following API actions:

- ListTagsForResource
- TagResource
- UntagResource
Workflow Studio for AWS Step Functions is a low-code visual workflow designer for Step Functions that lets you create serverless workflows by orchestrating AWS services. Use drag and drop to create and edit workflows, control how input and output is filtered or transformed for each state, and configure error handling. As you create a workflow, Workflow Studio validates your work and auto-generates code. You can review the generated code, or export it for local development or AWS CloudFormation. When you are finished, you can save your workflow, run it, then examine the results in the Step Functions console. You can also use Workflow Studio to prototype new workflows and use your prototypes as the starting point for local development with the AWS Toolkit for Visual Studio Code.
To use Step Functions Workflow Studio, you will need an AWS account, and credentials that provide the correct permissions for any resources you want to use. For more information, see Prerequisites for Getting Started with AWS Step Functions (p. 8).

You can access Workflow Studio from the Step Functions console, when you create or edit a workflow in Step Functions.

Topics
- Interface overview (p. 101)
- Using Workflow Studio (p. 108)
- Configure inputs and outputs for your states (p. 118)
- Error handling (p. 124)
- Tutorial: Learn to use the AWS Step Functions Workflow Studio (p. 126)
- Known limitations when using Workflow Studio (p. 133)

Interface overview

Workflow Studio for AWS Step Functions is a low-code visual workflow designer for Step Functions that lets you create serverless workflows by orchestrating AWS services.

Get to know Workflow Studio with an overview of the interface components: The states browser, the canvas, and the inspector panel.

1. The States browser contains two panels. The Actions panel provides a list of AWS APIs that you can drag and drop into your workflow graph in the canvas. Each Action represents a Task (p. 29) state. The Flow panel provides a list of flow states that you can drag and drop into your workflow graph in the canvas.
2. The canvas is where you drag and drop states into your workflow graph, change the order of states, and select states to configure or view.

3. The Inspector panel is where you can choose the Form, which lets you view and edit the properties of any state you select, and the Definition, where you can view the Amazon States Language code for your workflow, with the currently selected state highlighted.

4. Info links open a panel with contextual information when you need help. These panels also include links to related topics in the Step Functions documentation.

States browser

The States browser is where you select states to drag and drop into your workflow graph. The Actions panel provides a list of AWS APIs, and the Flow panel provides a list of flow states. You can search all states in the States Browser using the search field at the top.

There are seven flow states that you can use to direct and control your workflow. All of them take input from the previous state, and many let you filter the input from the preceding state, and the output to the state that follows. The flow states are:
• **Choice (p. 40):** Add a choice between branches of execution to your workflow. In the **Configuration** tab of the Inspector, you can configure rules to determine which state the workflow will transition to.

• **Parallel (p. 46):** Add parallel branches of execution to your workflow.

• **Map (p. 49):** Dynamically iterate steps for each element of an input array. Unlike a **Parallel** flow state, a **Map** state will execute the same steps for multiple entries of an array in the state input.

• **Pass (p. 28):** Lets you pass its input to its output. (Optional) You can add fixed data into the output.

• **Wait (p. 44):** Have your workflow pause for a certain amount of time or until a specified time or date.

• **Succeed (p. 46):** Stops your workflow with a success.

• **Fail (p. 46):** Stops your workflow with a failure.

**Canvas**

After you choose a state to add to your workflow, drag it to the canvas and drop it into your workflow graph. You can also drag and drop states to move them to different places in your workflow. If your workflow is complex, you may not be able to view all of it in the canvas panel. Use the controls at the top of the canvas to zoom in or out. To view different parts of a workflow graph, you can drag the workflow graph in the canvas.

Drag a workflow state from the **Actions** or **Flow** panel and drop it into your workflow. A line shows where it will be placed in your workflow. The new workflow state has been added to your workflow, and its code is auto-generated.
To change the order of a state, you can drag it to a different place in your workflow.
Inspector

After you have added a state to your workflow, you will want to configure it. Choose the state you want to configure, and you will see its configuration options in the Inspector panel. You can also see the workflow code by choosing the **Definition** panel. The code associated with the state you have selected will be highlighted.
# Keyboard shortcuts

Workflow Studio supports the following keyboard shortcuts:

<table>
<thead>
<tr>
<th>Keyboard shortcut</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + Z</td>
<td>Undo the last operation</td>
</tr>
<tr>
<td>Ctrl + Shift + Z</td>
<td>Redo the last operation</td>
</tr>
</tbody>
</table>

```
"Process data": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:::function:payload",
  "OutputPath": "$$.Payload",
  "Parameters": {
    "Payload.$": "$"
  },
  "Retry": {
    "ErrorEquals": [
      "Lambda.ServiceException",
      "Lambda.AWSLambdaException",
      "Lambda.SdkClientException"
    ],
    "IntervalSeconds": 2,
    "MaxAttempts": 5,
    "BackoffRate": 2
  }
},

"Next": "Choice"

"Choice": {
  "Type": "Choice",
  "Choices": [
    {
      "Next": "Send alert"
    }
  ],
  "Default": "Run backup"

"Send alert": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sns:::topic:alert",
  "Parameters": {
    "Message.$": "$"
  },
  "End": true
}

"Run backup": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:::function:backup",
  "OutputPath": "$$.Payload",
  "Parameters": {
    "Payload.$": "$"
  },
  "Retry": {
    "ErrorEquals": [
      "Lambda.ServiceException",
      "Lambda.AWSLambdaException",
      "Lambda.SdkClientException"
    ]
  }
}  ```
Using Workflow Studio

Learn to create, edit and run workflows using Step Functions Workflow Studio. After your workflow is ready, you can export it. You can also use Workflow Studio for rapid prototyping.

Create a workflow

1. Sign in to the Step Functions console.
2. Choose Create state machine.
3. On the Choose authoring method page, choose Design your workflow visually.
4. Under Type, choose Standard or Express, and then choose Next. Standard is recommended for your first state machine.

5. Design your workflow using Workflow Studio, then choose Next.

   **Note**
   If you see errors in your workflow, you can stay in Workflow Studio and correct the errors, or you can proceed to the next step and correct the errors in code.

6. On the Review generated code page, you can view the generated Amazon States Language definition of your workflow. If you want to make changes, you can do this in the Definition area. You can also access code snippets under Generate code snippet.

7. Choose Next.

8. In the Name box, enter a name for your state machine, for example, `MyStateMachine`.

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

9. In Execution role, under the Permissions section, choose Create new role.
10. Choose Create state machine.

You can now run your new workflow.

**Design a workflow**

If you know the name of the state you want to add, use the search box at the top of the states browser to find that state in the Actions and Flow panels.

Otherwise, choose a state from the states browser and drag and drop it onto the canvas, placing it where you want in your workflow. You can also re-order states in your workflow by dragging them to a different location in your workflow. As you drag a state onto the canvas, a line appears wherever you can drop it in your workflow. After a state is dropped onto the canvas, its code is auto-generated and added inside your workflow definition.
After you drop a state onto the canvas, you can configure it using the **Form** panel on the right. This panel further contains the **Configuration**, **Input**, **Output**, and **Error Handling** tabs for each of the state or API action that you place on the canvas. For example, the **Configuration** tab for Lambda Invoke API action consists of the following sections:
1. The **State name** identifies the state. You can use your own name or accept the default generated name.

2. The **API** shows which API the action uses.

3. The **Integration type** dropdown list provides options to choose the type of service integrations available in Step Functions. The integration type you choose is used to call API actions of a specific AWS service from your workflow.

4. The **Function name** provides options to:
   - **Enter a function name**: You can enter your function name or its ARN.
   - **Get function name at runtime from state input**: You can use this option to dynamically get the function name from the state input based on the path you specify.
   - **Select function name**: You can directly select from the functions available in your account and region.

5. The **Payload** lets you select from the following:
• **Use state input as payload** You can use this option to pass the state's input as the payload provided to your Lambda function.

• **Enter your own payload** You can use this option to construct a JSON object to pass as the payload to your Lambda function. This JSON can include both static values and values selected from the state input.

• **No payload** You can use this option if you don't want to pass any payload to your Lambda function.

6. **Next state** lets you to select the state you want to transition to next.

7. (optional) The **Comment** field can be used to add your own comment. It will not affect the workflow, but can be used to annotate your workflow.

8. (optional) Some states will have an option to select **Wait for task to complete** or **Wait for callback**. When available, these options select one of the following service integration patterns (p. 291):

   • **No option selected**: Step Functions will use the **Request Response** (p. 292) integration pattern. Step Functions will wait for an HTTP response and then progress to the next state. Step Functions will not wait for a job to complete. When no options are available, the state will use this pattern.

   • **Wait for task to complete**: Step Functions will use the **Run a Job (.sync)** (p. 292) integration pattern.

   • **Wait for callback**: Step Functions will use the **Wait for a Callback with the Task Token** (p. 293) integration pattern.

Some states will have more generic configuration options. For example, the Amazon ECS **RunTask** state configuration contains an **API Parameters** field populated with placeholder values.
## Run configuration

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Input</th>
<th>Output</th>
<th>Error handling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State name</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run configuration</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### API

**Amazon ECS: RunTask**

#### API Parameters

JSON object containing the parameters to pass into this API. Contains sample values. Update the JSON with your own parameter values.

```json
{
    "LaunchType": "FARGATE",
    "TaskDefinition": "arn:aws:ecs:REGION:ACCOUNT_ID:task-definition/MyTaskDefinition:1"
}
```

Must be valid JSON. To get a value from your state's JSON input using a path, the key must end in ".$".

### Next state

- **Choice**

### Comment - optional

**Enter comment**

- **Wait for task to complete - optional**
  - Pause the execution at this state and monitor the task. Resume the execution once the task is complete

- **Wait for callback - optional**
  - Pause the execution at this state until the execution receives a callback from SendTaskSuccess or SendTaskFailure APIs with the task token.

**Delete state**
For these states, you can replace the placeholder values with configurations that are suited to your needs.

To delete a state, you can use backspace, right-click and choose **Delete state** or use the **Form** and choose **Delete state**.

As your workflow grows, it may not fit in the canvas. You can:

1. Use the controls on the side panels to resize or close the panels.
2. Use the controls at the top of the canvas to zoom the workflow graph in or out.
Run your workflow

After you create or edit your workflow with the Workflow Studio, you can run it and view its execution in the Step Functions console:

1. Choose your workflow in the Step Functions console.
2. Choose Start execution. The Start execution dialog box is displayed.
3. Choose Start execution to run the workflow.
4. You can see the execution results for each state in the Graph inspector. Choose each step to view details about its execution.

Edit your workflow

You can edit an existing workflow visually using Workflow Studio, or you can edit an existing workflow in code. To edit an existing workflow:

1. Choose the workflow you want to edit.
2. Choose Edit.

3. The state machine has two panes: a code pane and a visual workflow pane. Choose Workflow Studio in the visual workflow pane to edit your workflow with Workflow Studio. When you are done, choose Apply and exit to save your changes and exit.
Note
If you see errors in your workflow, you can stay in Workflow Studio and correct the errors, or you can proceed to the next step and correct the errors in code.

4. (Optional) Use the code pane to edit the workflow definition in code.
5. If you have added new resources to your workflow, in Permissions, choose Create new role or use a role that allows additional access to these resources.
6. Choose Save to save your updated workflow.

Export your workflow

You can export your workflow's Amazon States Language code, and also a graph of your workflow:

1. Choose your workflow in the Step Functions console.
2. (optional) Choose Edit.
3. Choose Definition.
4. In the workflow definition:
   - To export your workflow's code, select it, then copy it to its destination.
   - To export the graph of your workflow to an SVG or PNG file, choose Export and select the format you want use.

Prototype a workflow

You can use Workflow Studio to prototype new workflows that contain placeholder resources. To create a prototype:

1. Sign in to the Step Functions console.
2. On the Choose authoring method page, choose Design your workflow visually.
3. Under Type, choose Standard or Express, then choose Next.
4. Design your workflow using Workflow Studio. To include placeholder resources:
   a. Choose the state for which you want to include a placeholder resource:
      - For Lambda Invoke states, under Configuration, choose Function name, then choose Enter function name. Enter a custom name for your function.
      - For Amazon SQS Send Message states, under Configuration, choose Queue URL, then choose Enter queue URL. Enter a placeholder queue URL.
      - For all other states listed under Actions, you can use the default configuration.
   b. Choose Next.

   Note
   If you see errors in your workflow, you can stay in Workflow Studio and correct the errors, or you can proceed to the next step and correct the errors in code.
   c. On the Review generated code page, you can view the generated Amazon States Language definition of your workflow. If needed, make any changes under Definition, then choose Next.
   d. Specify your workflow settings, then choose Create state machine.

You now have created a new workflow with placeholder resources that can be used to prototype. You can export your workflow definition and the graph of your workflow.

- To export your workflow definition, select and copy it to your clipboard. You can paste this definition and use it as the starting point for local development with the AWS Toolkit for Visual Studio Code.
• To export the graph of your workflow to an SVG or PNG file, choose **Export** and choose the format you want to use.

## Configure inputs and outputs for your states

Each state makes a decision or performs an action based on input that it receives. In most cases, it then passes output to other states. In Workflow Studio, you can configure how a state filters and manipulates its input and output data in the **Input** and **Output** tabs of the **Form** panel. Use the **Info** links to access contextual help when configuring inputs and outputs.

For detailed information about how Step Functions processes input and output, see Input and Output Processing in Step Functions (p. 55).

### Configure input to a state

Each state receives input from the previous state as JSON. If you want to filter the input, you can use the **InputPath** (p. 58) under the **Input** tab in the **Inspector** to select this. The InputPath is a string, beginning with `$`, that identifies a specific JSON node. These are called reference paths (p. 57), and they follow JsonPath syntax.
To filter the input:

- Choose **Filter input with InputPath**.
- Enter a valid InputPath.

Your InputPath filter will be added to your workflow.

**Example 1**

For this example, the input to your state includes the following:

```json
{
   "comment": "Example for InputPath",
   "dataset1": {
      "val1": 1,
      "val2": 2,
      "val3": 3
   },
   "dataset2": {
      "val1": "a",
      "val2": "b",
      "val3": "c"
   }
}
```

To apply the InputPath choose **Filter input with InputPath**, then enter an appropriate reference path. If you enter `.dataset2.val1`, the following JSON is passed as input to the state:

```json
{"a"}
```

A reference path can also have a selection of values. If your reference is `{ "a": [1, 2, 3, 4] }` and you apply the reference path `.a[0:2]` as the InputPath, the following is the result:

```
[ 1, 2 ]
```

Parallel, Map and Pass flow states have an additional input filtering option called **Parameters** under their **Input** tab. This filter takes effect after the InputPath filter and can be used to construct a custom JSON object consisting of one or more key-value pairs. The values of each pair can either be static values, can be selected from the input or can be selected from the Context Object (p. 70) with a path.

**Note**

To specify that a parameter uses a reference path to point to a JSON node in the input, the parameter name must end with `.#`. 
Example 2

In this example, the following is the input to a Parallel state:

```json
{
"comment": "Example for Parameters",
"product": {
  "details": {
    "color": "blue",
    "size": "small",
    "material": "cotton"
  },
  "availability": "in stock",
  "sku": "2317",
  "cost": "$23"
}
}
```

To select part of this information and pass additional key-value pairs with a static value, you can specify the following in the Parameters field, under the Parallel state's Input tab.

```json
{
"comment": "Selecting what I care about.",
"MyDetails": {
  "size.$": ".product.details.size",
  "exists.$": ".product.availability",
  "StaticValue": "foo"
}
}
```

The following JSON will be the result:

```json
{
"comment": "Selecting what I care about.",
"MyDetails": {
  "size": "small",
  "exists": "in stock",
  "StaticValue": "foo"
}
}
```

Configure output of a state

Each state produces JSON output that can be filtered before it is passed to the next state. There are several filters available, and each affects the output in a different way. Output filters available for each state are listed under the Output tab in the Inspector. For task states, any output filters you select are processed in this order:

1. ResultSelector (p. 60) Use ResultSelector to manipulate the state's result. You can construct a new JSON object with parts of the result.
2. ResultPath (p. 62). Use ResultPath to select a combination of the state input and the task result to pass to the output.
3. OutputPath (p. 68). Use OutputPath to filter the JSON output to choose which information from the result will be passed to the next state.
Configure output of a state

Use ResultSelector

ResultSelector is an optional output filter for the following states:

- Task states, which are all states listed in the Actions panel of the States browser.
- Map states, in the Flow panel of the States browser.
- Parallel states, in the Flow panel of the States browser.

ResultSelector can be used to construct a custom JSON object consisting of one or more key-value pairs. The values of each pair can either be static values or selected from the state's result with a path.

**Note**
To specify that a parameter uses a path to reference a JSON node in the result, the parameter name must end with .$.

**Example 1**
In this example, you use ResultSelector to manipulate the response from the Amazon EMR CreateCluster API call for an Amazon EMR CreateCluster state. The following is the result from the Amazon EMR CreateCluster API call:

```json
{
    "resourceType": "elasticmapreduce",
    "resource": "createCluster.sync",
    "output": {
        "SdkHttpMetadata": {
            "HttpHeaders": {
                "Content-Length": "1112",
                "Content-Type": "application/x-amz-JSON-1.1",
                "Date": "Mon, 25 Nov 2019 19:41:29 GMT",
                "x-amzn-RequestId": "1234-5678-9012"
            },
            "HttpStatusCode": 200
        },
        "SdkResponseMetadata": {
            "SdkHttpMetadata": {
                "HttpHeaders": {
                    "Content-Length": "1112",
                    "Content-Type": "application/x-amz-JSON-1.1",
                    "Date": "Mon, 25 Nov 2019 19:41:29 GMT",
                    "x-amzn-RequestId": "1234-5678-9012"
                },
                "HttpStatusCode": 200
            }
        }
    }
}```
To select part of this information and pass an additional key-value pair with a static value, specify the following in the `ResultSelector` field, under the state's **Output** tab:

```json
{
  "result": "found",
  "ClusterId.$": "$.output.ClusterId",
  "ResourceType.$": "$.resourceType"
}
```

Using `ResultSelector` produces the following result:

```json
{
  "result": "found",
  "ClusterId": "AKIAIOSFODNN7EXAMPLE",
  "ResourceType": "elasticmapreduce"
}
```

### Use ResultPath

The output of a state can be a copy of its input, the result it produces, or a combination of its input and result. Use `ResultPath` to control which combination of these is passed to the state output. For more use cases of `ResultPath`, see [ResultPath](p. 62).

`ResultPath` is an optional output filter for the following states:

- **Task states**, which are all states listed in the **Actions** panel of the States browser.
- **Map states**, in the **Flow** panel of the States browser.
- **Parallel states**, in the **Flow** panel of the States browser.
- **Pass states**, in the **Flow** panel of the States browser.

`ResultPath` can be used to add the result into the original state input. The specified path indicates where to add the result.

**Example**

The following is the input to a given Task state:

```json
{
  "details": "Default example",
  "who": "AWS Step Functions"
}
```

The result of the Task state is the following:

```
Hello, AWS Step Functions
```

You can add this result to the state's input by applying `ResultPath` and entering a reference path (p. 57) that indicates where to add the result, such as `$.taskresult:`
With this `ResultPath`, the following is the JSON that is passed as the state's output.

```json
{
    "details": "Default example",
    "who": "AWS Step Functions",
    "taskresult": "Hello, AWS Step Functions!"
}
```

**Use OutputPath**

The `OutputPath` filter lets you filter out unwanted information, and pass only the portion of JSON that you care about. The `OutputPath` is a string, beginning with $, that identifies nodes within JSON text.

**Example**

A Lambda Invoke API call returns metadata in addition to the payload, which is the Lambda function's result. An example of the response from this API call is shown under the Output tab of the state's Form mode.

[Diagram of Lambda Invoke]
You can use `OutputPath` to filter out the additional metadata. By default, Lambda Invoke states created through the Workflow Studio contain a `.Payload OutputPath` filter. This default removes the additional metadata and returns an output equivalent to running the Lambda function directly.

The Lambda Invoke task result example and the `.Payload Output` path pass the following JSON as the output:

```json
{
    "foo": "bar",
    "colors": [
        "red",
        "blue",
        "green"
    ],
    "car": {
        "year": 2008,
        "make": "Toyota",
        "model": "Matrix"
    }
}
```

**Note**

Since the `OutputPath` filter is the last output filter to take effect, if you use additional output filters such as `ResultSelector` or `ResultPath`, you should modify the default `.Payload OutputPath` filter accordingly.

## Error handling

By default, when a state reports an error, Step Functions causes the execution to fail entirely. For actions and some flow states, you can configure how Step Functions handles errors. Even if you have configured error handling, some errors may still cause an execution to fail. For more information, see Error handling in Step Functions (p. 88). In Workflow Studio, configure error handling in the the Error handling panel.
Retry on errors

You can add one or more rules to action states and the Parallel flow state to retry the task when an error occurs. These rules are called *retriers*. To add a retrier, choose **Add new retrier**, then configure its options:

- (Optional) The **Comment** field can be used to add your own comment. It will not affect the workflow, but can be used to annotate your workflow.
- Choose an **Error** that will trigger the retrier, or enter your own custom error name.
- (Optional) Set an **Interval**. This is the time in seconds before Step Functions make its first retry. Additional retries will follow at intervals that you can configure with **Max attempts** and **Backoff rate**.
- (Optional) Set **Max attempts**. This is the maximum number retries before Step Functions will cause the execution to fail.
- (Optional) Set the **Backoff rate**. This is a multiplier that determines by how much the retry interval will increase with each attempt.

Catch errors

You can add one or more rules to action states and to the Parallel and Map flow state to catch an error. These rules are called *catchers*. To add a catcher, choose **Add new catcher**, then configure its options:
(Optional) The **Comment** field can be used to add your own comment. It will not affect the workflow, but can be used to annotate your workflow.

Choose an **Error** that will trigger the catcher, or enter your own custom error name.

Choose an **Fallback state**. This is the state that the workflow will move to next, after an error is caught.

(Optional) Add a **ResultPath** to add the error to the original state input. The ResultPath (p. 62) must be in valid JSONPath syntax. This will be sent to the fallback state.

### Timeouts

You can configure a timeout for action states to set the maximum number of seconds your state can run before it fails. Use timeouts to prevent stuck executions. To configure a timeout, enter the number of seconds your state should wait before the execution fails. For more information, see **TimeoutSeconds** in the Task (p. 29) section.

### Heartbeat

You can configure a heartbeat or periodic notification sent by your task. If you set a heartbeat interval, and your state doesn't send heartbeat notifications in the configured intervals, the task is marked as failed. To configure a heartbeat, set a positive, non-zero integer number of seconds. For more information, see **HeartBeatSeconds** in the Task (p. 29) section.

**Note**

Not all error handling options are available for all states. Lambda Invoke has one retrier configured by default.

### Tutorial: Learn to use the AWS Step Functions Workflow Studio

In this tutorial, you will learn the basics of working with Workflow Studio for AWS Step Functions. In Workflow Studio, you'll create a state machine containing multiple states, including Pass, Choice, Fail, Wait, and Parallel. You'll use the drag and drop feature to search for, select, and configure these states. Then, you'll view the auto-generated Amazon States Language JSON code for your workflow, exit Workflow Studio, run the state machine, and review the execution details.

In this tutorial, you'll also learn how to update the state machine and view the changes in the execution output. Finally, you'll perform a clean-up step and delete your state machine.

After you complete this tutorial, you'll know how to use Workflow Studio to create and configure a workflow. You'll also know how to update, execute, and delete your state machine.

**Note**

Before you start, make sure to complete the prerequisites for this tutorial (p. 8).

### Topics

- **Step 1**: Navigate to Workflow Studio (p. 127)
- **Step 2**: Create a state machine (p. 127)
- **Step 3**: Review the auto-generated Amazon States Language definition (p. 129)
- **Step 4**: Start a new execution (p. 131)
- **Step 5**: Update your state machine (p. 132)
- **Step 6**: Clean up (p. 133)
Step 1: Navigate to Workflow Studio

1. Sign in to the Step Functions console.
2. Choose Create state machine.
3. On the Choose authoring method page, choose Design your workflow visually.
4. For Type, keep the default selection of Standard.

Step Functions has two workflow types: Standard and Express. These types determine how Step Functions performs tasks, integrates with AWS services, and manages pricing. Once you create a state machine, you can't change its workflow type.

For a side-by-side comparison of both workflows, under Type, choose Help me decide.
5. Choose Next. This will open Workflow Studio.

Step 2: Create a state machine

In Workflow Studio, a state machine is a graphical representation of your workflow. You can use Workflow Studio to define, configure, and examine the individual steps of your workflow.

To create a state machine

1. In Workflow Studio, from the states browser on the left, choose the Flow panel. Then, drag a Pass state to the empty state labelled Drag first state here.
2. Drag a Choice state from the Flow panel and drop it below the Pass state.
3. For State name, replace the default name. For this tutorial, use the name IsHelloWorldExample.
4. Drag another Pass state and drop it to one branch of the Choice state. Then, drag a Fail state to the other branch of the Choice state.
5. Choose the Pass (1) state, and rename it to Yes. Rename the Fail state as No.
6. Specify the Choice state's branching logic using the boolean variable IsHelloWorldExample.
If IsHelloWorldExample is False, the workflow will enter the No state. Otherwise, the workflow will continue its execution flow in the Yes state’s branch.

To define the branching logic, do the following:

a. Choose the Choice state on the canvas, and then under Choice Rules choose ☑ in the Rule #1 box to define the first choice rule.

b. Choose Add conditions.

c. In the Conditions for rule #1 dialog box, enter $.IsHelloWorldExample under Variable.

d. Choose is equal to under Operator.

e. Choose Boolean constant under Value, and then choose true from the dropdown list.

f. Choose Save conditions.

g. Make sure the Then next state is: dropdown list has Yes selected.

h. Choose Add new choice rule, then choose Add conditions. In the Rule #2 box, define the second choice rule when the IsHelloWorldExample variable’s value is false by repeating substeps 6.c through 6.e. For step 6.e, choose false instead of true.

i. In the Rule #2 box, choose No from the Then next state is: dropdown list.

j. In the Default rule box, click ☑ to define the default choice rule, and then choose Yes from the dropdown list.

7. Add a Wait state after the Yes Pass state, and name it Wait 3 sec. Then, configure the wait time to be three seconds by doing the following steps:

a. Under Options, keep the default selection of Wait for a fixed interval.

b. Under Seconds, make sure Enter seconds is selected, and then enter 3 in the box.

8. After the Wait 3 sec state, add a Parallel state. Add two more Pass states in its two branches. Name the first Pass state Hello. Name the second Pass state World.

The completed workflow will look like this:
Step 3: Review the auto-generated Amazon States Language definition

As you drag and drop states from the Flow panel onto the canvas, Workflow Studio automatically composes the Amazon States Language (p. 23) definition of your workflow in real-time. You can edit this definition as required.

1. (Optional) Choose Definition on the Inspector panel and view the state machine's workflow.

The following example code shows the auto-generated Amazon States Language definition for the IsHelloWorldExample state machine. The Choice state that you added in Workflow Studio is used to determine the execution flow based on the branching logic you defined in Step 2 (p. 127).
```json
{
  "Comment": "A Hello World example of the Amazon States Language using Pass states",
  "StartAt": "Pass",
  "States": {
    "Pass": {
      "Type": "Pass",
      "Next": "IsHelloWorldExample",
      "Comment": "A Pass state passes its input to its output, without performing work. Pass states are useful when constructing and debugging state machines."
    },
    "IsHelloWorldExample": {
      "Type": "Choice",
      "Comment": "A Choice state adds branching logic to a state machine. Choice rules can implement 16 different comparison operators, and can be combined using And, Or, and Not\"",
      "Choices": [
        {
          "Variable": "$.IsHelloWorldExample",
          "BooleanEquals": false,
          "Next": "No"
        },
        {
          "Variable": "$.IsHelloWorldExample",
          "BooleanEquals": true,
          "Next": "Yes"
        }
      ],
      "Default": "Yes"
    },
    "No": {
      "Type": "Fail",
      "Cause": "Not Hello World"
    },
    "Yes": {
      "Type": "Pass",
      "Next": "Wait 3 sec"
    },
    "Wait 3 sec": {
      "Type": "Wait",
      "Seconds": 3,
      "Next": "Parallel"
    },
    "Parallel": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "Hello",
          "States": {
            "Hello": {
              "Type": "Pass",
              "End": true
            }
          }
        },
        {
          "StartAt": "World",
          "States": {
            "World": {
              "Type": "Pass",
              "End": true
            }
          }
        }
      ]
    }
  }
}
```
Step 4: Start a new execution

State machine executions are instances where you run your workflow to perform tasks.

2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name 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, enter input values for your execution in JSON format. Based on your input, the IsHelloWorldExample variable determines which state machine flow will be executed. For now, use the following input value:

   ```json
   {
     "IsHelloWorldExample": true
   }
   
   **Note**
   While specifying an execution input is optional, in this tutorial, it is mandatory to specify an execution input similar to the above example input. This input value is referenced in the Choice state when you run the state machine.

5. The Step Functions console directs you to a page that's titled with your execution ID. On this page, you can review the results of your new execution. Under Details, you can see your execution ARN.

```

2. Choose Next.
3. On the Review generated code page, review your state machine's Amazon States Language definition. If needed, you can make additional changes under Definition.
4. Choose Next.
5. Enter a name for your workflow. For example, enter HelloWorld.
6. Under Permissions, choose Create a new IAM role.

   When you create a state machine, you select an IAM role that defines which resources the state machine has permission to access during its execution. Choose one of the following options:

   - **Create a new IAM role** – Select this option when you want Step Functions to create a new IAM role for you based on the definition of your state machine and its configuration details.

   - **Choose an existing role** – Select this option if you previously created an IAM role for Step Functions and your state machine has the correct permissions.

   - **Enter a role ARN** – Select this option if you know the ARN for the IAM role that you want to use for Step Functions.

7. Choose Create state machine.
and a status which indicates if your execution succeeded. You can also see the timestamps for when your execution started and ended. To view the results of your execution, choose **Execution output**.

For this tutorial, if you entered an input value of "IsHelloWorldExample": true, you should see the following output:

```json
{
  "IsHelloWorldExample": true
},
{
  "IsHelloWorldExample": true
}
```

---

### Step 5: Update your state machine

When you update a state machine, your updates are **eventually consistent**. After a short amount of time, all newly started executions will reflect your state machine's updated definition and roleARN. All currently running executions will run to completion under the previous definition and roleARN before they update.

In this step, you'll update your state machine by adding a **Result** field in the **Pass** state named **World**.

1. On the page titled with your execution ID, choose **Edit state machine**.
2. On the **Edit HelloWorld** page, in the **Definition** area, choose **Workflow Studio**.
3. Choose the **Pass** state named **World** on the canvas, and then choose **Output**.
4. In the **Result** box, enter "**World has been updated!**".
5. Choose **Apply and exit** to save the change and return to the **Edit HelloWorld** page.
6. (Optional) In the **Definition** area, view the updated Amazon States Language definition of your workflow.

```json
{
  "Type": "Parallel",
  "End": true,
  "Branches": [
    {
      "StartAt": "Hello",
      "States": {
        "Hello": {
          "Type": "Pass",
          "End": true
        }
      }
    },
    {
      "StartAt": "World",
      "States": {
        "World": {
          "Type": "Pass",
          "Result": "World has been updated!",
          "End": true
        }
      }
    }
  ]
}
```

7. Choose **Save**, and then choose **Start execution**.
8. The Step Functions console displays the following message:
The changes to your state machine may affect which resources it needs to access. To ensure your state machine has the right permissions, you might need to edit the current IAM role, create a new one, or select a different role.

This message is standard. Choose Save anyway.

9. Choose Start execution. In the Start execution dialog box, identify your execution, and then enter the following input value in the Input area:

```json
{
  "IsHelloWorldExample": true
}
```

10. Choose Start Execution again.

11. (Optional) In the visual workflow panel, choose each step to view its details, input, and output under the Details, Step input and Step output tabs, respectively. For example, in the visual workflow panel, choose the World step, and then choose Step output. The output is World has been updated!

12. (Optional) To export the graph of your workflow to an SVG or PNG file, choose Export.

Step 6: Clean up

To delete your state machine

1. From the navigation menu, choose State machines.
2. On the State machines page, select HelloWorld, and then choose Delete.
3. The Step Functions console prompts you with the following message:

   You are about to delete your state machine. Do you want to proceed?

   This message is standard. Choose Delete state machine.

   If deletion is successful, a green status bar appears at the top of your screen. The green status bar tells you that your state machine is marked for deletion. Your state machine will be removed when all of its executions stop running.

To delete your execution role

1. Open the Roles page for IAM.
2. Choose the IAM role that Step Functions created for you: StepFunctions-HelloWorld-role-EXAMPLE.
3. Choose Delete role.
4. Choose Yes, delete.

Known limitations when using Workflow Studio

When working with Workflow Studio, there are several limitations that you should be aware of.
Workflow Studio does not support dynamic resource IDs

You cannot use dynamic ARNs in the Resource section of a Task (p. 29) state. Instead, you can enter the dynamic ARN in the appropriate Parameters field of your task state. For example, if you have the dynamic ARN `${my-input}`, you could add to the field as seen in the following:

Note
State machine definitions that contain AWS CloudFormation references cannot be saved. However, you can copy or export the definition for use elsewhere.
A task token is not added to the state definition when you select *Wait for callback* in Workflow Studio

When you select *Wait for callback*, Workflow Studio adds `.waitForTaskToken` to the resource, but no task token is added to the state definition. To add a task token:

1. On the *Configuration* tab, make sure the *Wait for a callback* option is selected.
2. Finish editing your state machine in Workflow Studio, then choose **Next** if you’re creating a new state machine, or choose **Apply and exit** if you’re editing an existing state machine.
3. Choose **Generate code snippet**, then choose the task state for which you want to use *Wait for callback*.
4. Choose **Wait for a callback with task token**, then copy the added text. This text will be similar to "TaskToken.$": "$.Task.Token".
5. Paste this text in the appropriate field in the editor.

You can now finish creating or updating your state machine.

Some service states are not available in Workflow Studio

Some service states are not currently available in Workflow Studio. Instead, after you create or edit your state machine in Workflow Studio, you can use code snippets (p. 298) to add the state to your state machine definition. Code snippets can be found in the drop down menu on your state machine’s code pane.
Workflow Studio does not support Internet Explorer 11

Internet Explorer 11 is not supported by Workflow Studio. If you are using Internet Explorer 11 and encounter issues using Workflow Studio, try using a different browser.
Tutorials for Step Functions

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

To complete these tutorials, you need an AWS account. If you don't have an AWS account, navigate to https://aws.amazon.com/ and choose Create an AWS Account.

Topics
- Creating a Step Functions State Machine That Uses Lambda (p. 137)
- Handling Error Conditions Using a Step Functions State Machine (p. 140)
- Using a Map State to Call Lambda Multiple Times (p. 144)
- Periodically Start a State Machine Execution Using EventBridge (p. 148)
- Starting a State Machine Execution in Response to Amazon S3 Events (p. 149)
- Creating a Step Functions API Using API Gateway (p. 152)
- Create a Step Functions State Machine Using AWS SAM (p. 157)
- Creating an Activity State Machine Using Step Functions (p. 162)
- Iterating a Loop Using Lambda (p. 167)
- Continuing as a New Execution (p. 172)
- Using Code Snippets to Create a State to Send an Amazon SNS message (p. 181)
- Deploying an Example Human Approval Project (p. 185)
- View X-Ray traces in Step Functions (p. 195)
- Gather Amazon S3 bucket info using AWS SDK service integrations (p. 201)

Creating a Step Functions State Machine That Uses Lambda

In this tutorial, you will create a single-step workflow using AWS Step Functions to invoke an AWS Lambda function.

Note
In Step Functions, a workflow is called a state machine, which is a series of event-driven steps. Each step in a workflow is called a state. A task state represents a unit of work that another AWS service, such as AWS Lambda, performs. A task state can call any AWS service or API. For more information, see:

- What is AWS Step Functions? (p. 1)
- Call other AWS services (p. 271)

Lambda is well-suited for task states, because Lambda functions are serverless and easy to write. You can write code in the AWS Management Console or your favorite editor. AWS handles the details of providing a computing environment for your function and running it.

Topics
- Step 1: Create a Lambda Function (p. 138)
- Step 2: Test the Lambda Function (p. 138)
- Step 3: Create a State Machine (p. 139)
- Step 4: Start a New Execution (p. 140)
Step 1: Create a Lambda Function

Your Lambda function receives event data and returns a greeting message.

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

1. Open the Lambda console and choose **Create function**.
2. On the **Create function** page, choose **Author from scratch**.
3. In the **Basic information** section, configure your Lambda function:
   a. For **Function name**, enter **HelloFunction**.
   b. For **Runtime**, choose **Node.js 14.x**.
   c. In **Change default execution role**, choose **Create a new role with basic Lambda permissions**.
   d. Choose **Create function**.
   e. After your Lambda function is created, copy the function's Amazon Resource Name (ARN) displayed in the upper-right corner of the page. To copy the ARN, click ⬤. The following is an example ARN:

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

4. Copy the following code for the Lambda function into the **Code source** section of the **HelloFunction** page.

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

   This code assembles a greeting using the **who** field of the input data, which is provided by the **event** object passed into your function. You add input data for this function later, when you start a new execution (p. 140). The **callback** method returns the assembled greeting from your function.

5. Choose **Deploy**.

Step 2: Test the Lambda Function

Test your Lambda function to see it in operation.

1. Choose **Test**.
2. In the **Configure test event** dialog box, enter **HelloEvent** in the **Event name** box.
3. 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 input the same input data when you run your state machine.

4. Choose **Create**.
5. On the **HelloFunction** page, choose **Test** to test your Lambda function using the new data.

   The results of the test are displayed in the **Execution results** tab.
6. Choose the **Execution results** tab to see the output.

### Step 3: Create a State Machine

Use the Step Functions console to create a state machine that invokes the Lambda function that you created earlier in Step 1 (p. 138).

1. Open the Step Functions console and 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 **Choose authoring method** page, choose **Design your workflow visually**.

3. For **Type**, retain the default selection, that is, **Standard**.

4. Choose **Next**. This will open Workflow Studio.

5. From the States browser on the left, choose the **Actions** panel.
   
   - Drag and drop the **AWS Lambda Invoke** API into the empty state labelled **Drag first state here**.

6. In the **Inspector** panel on the right, configure the Lambda function and its name:
   
   a. Choose **Configuration**, and then edit the **State name**, if required.
   b. In the **API Parameters** section, choose the Lambda function that you created earlier (p. 138) in the **Function name** dropdown list.
   c. Retain the default selection in the **Payload** dropdown list.

7. Choose **Next**.

8. On the **Review generated code** page, review the state machine's Amazon States Language (ASL) definition, which is automatically generated based on your selections in the **Actions** and **Inspector** panel.

9. Choose **Next**.

10. Enter a **Name for your state machine**, for example, *LambdaStateMachine*.

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

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

11. In **Execution role**, under the **Permissions** section, choose **Create new role**.
12. Choose **Create state machine**.

### Step 4: Start a New Execution

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

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

   The **Start execution** dialog box is displayed.
2. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name 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, choose the **Execution output** tab.

### Handling Error Conditions Using a Step Functions State Machine

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

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

**Note**
- You can also create state machines that **Retry** on timeouts or those that use **Catch** to transition to a specific state when an error or timeout occurs. For examples of these error handling techniques, see **Examples Using Retry and Using Catch** (p. 93).

**Topics**
- Step 1: Create a Lambda Function That Fails (p. 141)
- Step 2: Test the Lambda Function (p. 141)
Step 1: Create a Lambda Function That Fails

Use a Lambda function to simulate an error condition.

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

1. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
   Choose Create function.
2. Choose Use a blueprint, enter step-functions into the filter, and then choose the step-functions-error blueprint.
3. Choose Configure.
4. In the Basic information section, configure your Lambda function:
   a. For Name, enter FailFunction.
   b. For Role, choose Create a new role with basic Lambda permissions.
5. The following code is displayed in the Lambda function code pane.

```javascript
exports.handler = async (event, context) => {
  function CustomError(message) {
    this.name = 'CustomError';
    this.message = message;
  }
  CustomError.prototype = new Error();
  throw new CustomError('This is a custom error!');
};
```

The context object returns the error message This is a custom error!.
6. Choose Create function.

After your Lambda function is created, copy the function’s Amazon Resource Name (ARN) displayed in the upper-right corner of the page. To copy the ARN, click 📋. The following is an example ARN:

```
arn:aws:lambda:us-east-1:123456789012:function:FailFunction
```
7. Choose Deploy.

Step 2: Test the Lambda Function

Test your Lambda function to see it in operation.

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

The results of the test (the simulated error) are displayed in a new tab Execution results.
Step 3: Create a State Machine with a Catch Field

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

1. Open the Step Functions console and choose Create state machine.
2. On the Choose authoring method page, choose Write your workflow in code.
3. For Type, retain the default selection, that is, Standard.
4. In the Definition pane, paste the following code, but replace the ARN of the Lambda function that you created earlier (p. 141) in the Resource field.

```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. 24).

For more information about the syntax of the Retry field, see Examples using Retry and using Catch (p. 93).

**Note**

Unhandled errors in Lambda are reported as Lambda.Unknown in the error output. These include out-of-memory errors and function timeouts. You can match on Lambda.Unknown, States.ALL, or States.TaskFailed to handle these errors. When Lambda hits the
maximum number of invocations, the error is Lambda.TooManyRequestsException. For more information about Lambda Handled and Unhandled errors, see FunctionError in the AWS Lambda Developer Guide.

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

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

6. Choose Next.
7. Enter a Name for your state machine, such as Catchfailure.
8. In Permissions, choose Create new role.
9. Choose Create state machine.

**Step 4: Start a New Execution**

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

1. On the Catchfailure page, choose Start execution.

The Start execution dialog box is displayed.

2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name 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. Go to the Execution output tab to view the output of your workflow.

5. To view your custom error message, select CreateAccount in the Graph inspector pane and choose the Step output tab.

   **Note**
   You can preserve the state input with the error by using ResultPath. See Use ResultPath to Include Both Error and Input in a Catch (p. 67).
Using a Map State to Call Lambda Multiple Times

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

The Creating a Step Functions State Machine That Uses Lambda (p. 137) tutorial walks you through creating a state machine that calls a Lambda function. If you have completed that tutorial, skip to Step 4 (p. 146) and use the AWS Identity and Access Management (IAM) role and Lambda function that you previously created.

Topics

- Step 1: Create an IAM Role for Lambda (p. 144)
- Step 2: Create a Lambda Function (p. 144)
- Step 3: Test the Lambda Function (p. 145)
- Step 4: Create a State Machine (p. 146)
- Step 5: Start a New Execution (p. 147)

Step 1: Create an IAM Role for Lambda

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

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

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

To create a role (console)

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

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

Step 2: Create a Lambda Function

Your Lambda function receives input (a name) and returns a greeting that includes the input value.
Important
Ensure that your Lambda function is under the same AWS account and AWS Region as your state machine.

1. Open the Lambda console and choose Create function.
2. In the Blueprints section, choose Author from scratch.
3. In the Basic information section, configure your Lambda function:
   a. For Function name, enter HelloFunction.
   b. For Runtime, choose Node.js 12.x.
   c. For Role, select Choose an existing role.
   d. For Existing role, select the Lambda role that you created earlier (p. 144).
   
   Note
   If the IAM role that you created doesn't appear in the list, the role might still need a few minutes to propagate to Lambda.

   e. Choose Create function.

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

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

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

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

This code assembles a greeting using the who field of the input data, which is provided by the event object passed into your function. You add input data for this function later, when you start a new execution (p. 147). The callback method returns the assembled greeting from your function.

5. Choose Save.

Step 3: Test the Lambda Function

Test your Lambda function to see it in operation.

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

```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. Expand Details to see the output.
Step 4: Create a State Machine

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

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

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

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

3. In the State machine definition pane, add the following state machine definition using the ARN of the Lambda function that you created earlier (p. 145), as shown in the following example.

   ```json
   {
   "StartAt": "ExampleMapState",
   "States": {
   "ExampleMapState": {
   "Type": "Map",
   "Iterator": {
   "StartAt": "CallLambda",
   "States": {
   "CallLambda": {
   "Type": "Task",
   "End": true
   }
   }
   }
   }
   }
   
   This is a description of your state machine using the Amazon States Language. It defines a Map state named ExampleMapState that includes a Task state (CallLambda) that calls your Lambda function. For more information, see State Machine Structure (p. 24).

   **Note**
   You can also set up a Retry for Task states. As a best practice, ensure production code can handle Lambda service exceptions (Lambda ServiceException and Lambda.SdkClientException). For more information, see the following:
Choose Next.

4. Create or enter an IAM role:

   - To create an IAM role for Step Functions, select Create an IAM role for me, and enter a Name for your role.
   - If you have previously created an IAM role (p. 526) with the correct permissions for your state machine, select Choose an existing IAM role. Select a role from the list, or provide an ARN for that role.

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

5. Select Next.

Step 5: Start a New Execution

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


   The New execution page is displayed.

2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name 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": "bob"
     },
     {
       "who": "meg"
     },
     {
       "who": "joe"
     }
   ]
   ```

   Your Map state will iterate and run the CallLambda task state for each of these items in the input.

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


   A new execution of your state machine starts, and a new page showing your running execution is displayed.
Periodically Start a State Machine Execution Using EventBridge

You can execute an AWS Step Functions state machine in response to an event pattern or on a schedule using Amazon EventBridge. This tutorial shows you how to set a state machine as a target for an EventBridge rule that starts the periodic execution of a state machine based on the schedule you define.

**Topics**

- Step 1: Create a State Machine (p. 148)
- Step 2: Create an EventBridge Rule (p. 148)
- Example of Execution Input (p. 149)

**Step 1: Create a State Machine**

Before you can set an EventBridge target, you must create a state machine.

- To create a basic state machine, use the Getting Started (p. 10) tutorial.
- If you already have a state machine, proceed to the next step.

**Step 2: Create an EventBridge Rule**

After you create your state machine, you can create your EventBridge rule.

1. Navigate to the Amazon EventBridge console, and choose Create rule.

   Alternatively, on the Amazon EventBridge console, choose Rules, and then choose Create rules.
2. On the Step 1 Define rule detail page, enter a name and description for the rule.

   **Note**
   A rule can't have the same name as another rule in the same Region and on the same event bus.

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

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

For more information on using Map states, see the following:

- Map (p. 49)
- Map State Example (p. 50)
- Map State Input and Output Processing (p. 52)
- ItemsPath (p. 61)
3. For **Event bus**, choose the event bus that you want to associate with this rule. If you want this rule to match events that come from your account, choose **default**. When an AWS service in your account emits an event, it always goes to your account’s default event bus.

4. For **Rule type**, choose **Schedule**.

5. Choose **Next**.

6. On the **Step 2 Define schedule** page, for **Schedule pattern**, do one of the following:

   - To use a cron expression to define the schedule, choose **A fine-grained schedule that runs at a specific time, such as 8:00 a.m. PST on the first Monday of every month.** and enter the cron expression.

   - To use a rate expression to define the schedule, choose **A schedule that runs at a regular rate, such as every 10 minutes.** and enter the rate expression.

7. Choose **Next**.

8. On the **Step 3 Select target(s)** page, for **Target types**, choose **AWS service**.

9. For **Select a target**, choose **Step Functions state machine**.

10. For **State machine**, choose the state machine that you want to run periodically.

11. For **Execution role**, do one of the following:

   - To create an IAM role automatically, choose **Create a new role for this specific resource**.

   - To use an IAM role that you created earlier, choose **Use existing role** and choose the existing role from the drop-down list.

12. Choose **Next**.

13. (Optional) On the **Step 4 Configure tags** page, enter one or more tags for the rule.

14. Choose **Next**.

15. On the **Step 5 Review and create** page, review the details of the rule and choose **Create rule**.

### Example of Execution Input

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

```json
{
  "version": "0",
  "id": "6c540ad4-0671-9974-6511-756fb789fced",
  "detail-type": "Scheduled Event",
  "source": "aws.events",
  "account": "123456789012",
  "time": "2022-03-26T00:01:02Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:sm:::rule/my-state-machine"
  ],
  "detail": {}
}
```

### Starting a State Machine Execution in Response to Amazon S3 Events

You can use Amazon EventBridge to execute an AWS Step Functions state machine in response to an event or on a schedule.
This tutorial shows you how to configure a state machine as a target for an Amazon EventBridge rule. This rule will start an execution when files are added to an Amazon Simple Storage Service (Amazon S3) bucket.

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

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

**Topics**
- Prerequisite: Create a State Machine (p. 150)
- Step 1: Create a Bucket in Amazon S3 (p. 150)
- Step 2: Enable Amazon S3 Event Notification with EventBridge (p. 150)
- Step 3: Create an Amazon EventBridge Rule (p. 151)
- Step 4: Test the Rule (p. 151)
- Example of Execution Input (p. 152)

### Prerequisite: Create a State Machine

Before you can configure an Amazon EventBridge target, you must create a state machine.

- To create a basic state machine, use the Getting Started (p. 10) tutorial.
- If you already have a Helloworld state machine, proceed to the next step.

### Step 1: Create a Bucket in Amazon S3

Now that you have a Helloworld state machine, you need to first create an Amazon S3 bucket which stores your files. In Step 3 of this tutorial, you set up a rule so that when a file is added to this bucket, EventBridge triggers an execution of the state machine.

1. Navigate to the Amazon S3 console, and then choose Create bucket to create the bucket in which you want to store your files and trigger an Amazon S3 event rule.
2. Enter a **Bucket name**, such as `username-sfn-tutorial`.
   **Note**
   Bucket names must be unique across all existing bucket names in all AWS Regions in Amazon S3. Use your own `username` to make this name unique. You need to create all resources in the same AWS Region.
3. Choose Create bucket.

### Step 2: Enable Amazon S3 Event Notification with EventBridge

After you create the Amazon S3 bucket, configure it to send events to EventBridge whenever certain events happen in your S3 bucket, such as file uploads. In Step 3, you create an EventBridge rule to route events to a target, such as a Step Functions state machine.

1. Navigate to the Amazon S3 console.
2. In the **Buckets** list, choose the name of the bucket that you want to enable events for.
3. Choose **Properties**.
4. Navigate to the **Event Notifications** section and find the **Amazon EventBridge** subsection. Then choose **Edit**.
5. Under **Send notifications to Amazon EventBridge for all events in this bucket**, choose **On**.

   **Note**
   After you enable EventBridge, it takes around five minutes for the changes to take effect.

---

### Step 3: Create an Amazon EventBridge Rule

After you have a state machine, and have created the Amazon S3 bucket and configured it to send event notifications to EventBridge, create an EventBridge rule.

**Note**
You must configure EventBridge rule in the same AWS Region as the Amazon S3 bucket.

#### To create the rule

1. Navigate to the Amazon EventBridge console, choose **Create rule**.
2. Enter a **Name** for your rule (for example, **S3StepFunctions**) and optionally enter a **Description** for the rule.
3. In **Define pattern**, choose **Event pattern**.
4. In **Event matching pattern**, choose **Pre-defined pattern by service**.
5. In **Service provider**, choose **AWS**.
6. In **Service name**, choose **Simple Storage Service (S3)**.
7. For **Event type**, choose **Amazon S3 Event Notification**.
8. Choose **Specific operation(s)**, and then choose **Object Created**.
9. Choose **Specific bucket(s) by name** and enter the bucket name you created in Step 1 (**username-sfn-tutorial**) to store your files.
10. Retain the default selections in **Select event bus**.

#### To create the target

1. Choose **Step Functions state machine** in the **Target** dropdown list, and in the **State machine** list, choose the state machine from Step 1 (**Helloworld**).
2. Amazon EventBridge can create the IAM role that your event needs to run:
   - To create an IAM role automatically, choose **Create a new role for this specific resource**.
   - To use an IAM role that you created before, choose **Use existing role**.
3. Choose **Create**.

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

---

### Step 4: Test the Rule

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

1. Add a file to your Amazon S3 bucket.
Navigate to the Amazon S3 console, select the bucket you created to store files (username-sfn-tutorial), and then choose Upload.

2. Add a file (test.png in the following example), and then choose Upload.

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

3. Check the execution for your state machine.

Navigate to the Step Functions console and select the state machine used in your Amazon EventBridge rule (helloworld).

4. Select the most recent execution of that state machine and expand the Execution Input section.

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

Example of Execution Input

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

```json
{
  "version": "0",
  "id": "6c540ad4-0671-9974-6511-756fbd7771c3",
  "detail-type": "Object Created",
  "source": "aws.s3",
  "account": "123456789012",
  "time": "2022-02-19T01:36:58Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:s3:::username-sfn-tutorial"
  ],
  "detail": {
    "version": "0",
    "bucket": {
      "name": "username-sfn-tutorial"
    },
    "object": {
      "key": "test.png",
      "size": 800704,
      "etag": "f31d8546bb67845b4d3048cde533b937",
      "sequencer": "00621049BA9A8C712B"
    },
    "request-id": "79104EXAMPLEB723",
    "requester": "123456789012",
    "source-ip-address": "203.0.113.34",
    "reason": "PutObject"
  }
}
```

Creating a Step Functions API Using API Gateway

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

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

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

**Topics**
- Step 1: Create an IAM Role for API Gateway (p. 153)
- Step 2: Create your API Gateway API (p. 153)
- Step 3: Test and Deploy the API Gateway API (p. 156)

**Step 1: Create an IAM Role for API Gateway**

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

**To set up permissions for API Gateway**

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 API Gateway from the list, and then choose Next: Permissions.
3. On the Attached permissions policy page, choose Next: Tags.
4. (Optional) On the Add tags page, enter any tags you want to add, then choose Next: Review.
5. On the Review page, enter `APIGatewayToStepFunctions` for Role name, and then choose Create role.

The IAM role appears in the list of roles.

6. Choose the name of your role and note the Role ARN, as shown in the following 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: Create your API Gateway API**

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

**To create the API**

1. Open the Amazon API Gateway console.
2. On the REST API pane, choose Build.
3. Choose New API.
4. In the Settings section, enter `StartExecutionAPI` for the API name, and then choose Create API.
To create a resource

2. On the New Child Resource page, enter 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 check mark.

To configure the integration point for your method

1. On the /execution - POST - Setup page, for Integration Type, choose AWS Service.
2. For AWS Region, choose a Region from the list.
   
   **Note**
   For Regions that currently support Step Functions, see Supported Regions (p. 7).
3. For AWS Service, choose Step Functions from the list.
4. For HTTP Method, choose POST from the list.
   
   **Note**
   All Step Functions API actions use the HTTP POST method.
5. For Action Type, choose Use action name.
6. For Action, enter StartExecution.
7. For Execution Role, enter the role ARN of the IAM role that you created earlier (p. 153), as shown in the following example.

```
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: Test and Deploy the API Gateway API

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

To test the communication between API Gateway and Step Functions

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. 10)), 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, as shown in the following example.

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

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

```json
{
  "input": "$util.escapeJavaScript($input.json('"'))",
```
Create a Step Functions State Machine Using AWS SAM

In this guide, you download, build, and deploy a sample AWS SAM application that contains an AWS Step Functions state machine. This application creates a mock stock trading workflow which runs on a pre-defined schedule (note that the schedule is disabled by default to avoid incurring charges).
The following is a preview of commands that you run to create your sample application. For more details about each of these commands, see the sections later in this page

```
# Step 1 - Download a sample application. For this tutorial you
# will follow the prompts to select an AWS Quick Start Template
# called 'Step Functions Sample App (Stock Trader)'
sam init

# Step 2 - Build your application
cd project-directory
sam build

# Step 3 - Deploy your application
sam deploy --guided
```

Prerequisites

This guide assumes that you've completed the steps in the Installing the AWS SAM CLI for your OS. It assumes that you've done the following:

1. Created an AWS account.
2. Configured IAM permissions.
3. Installed Homebrew. Note: Homebrew is only a prerequisite for Linux and macOS.
4. Installed the AWS SAM CLI. Note: Make sure you have version 0.52.0 or later. You can check which version you have by executing the command `sam --version`. 
Step 1: Download a Sample AWS SAM Application

Command to run:

```
sam init
```

Follow the on-screen prompts to select the following:

1. **Template:** AWS Quick Start Templates
2. **Language:** Python, Ruby, NodeJS, Go, Java, or .NET
3. **Project name:** (name of your choice - default is `sam-app`)
4. **Quick start application:** Step Functions Sample App (Stock Trader)

What AWS SAM is doing:

This command creates a directory with the name you provided for the 'Project name' prompt (default is `sam-app`). The specific contents of the directory will depend on the language you choose.

Following are the directory contents when you choose one of the Python runtimes:

```markdown
### README.md
### functions
#   ### __init__.py
#   ### stock_buyer
#   #   ### __init__.py
#   #   ### app.py
#   #   ### requirements.txt
#   ### stock_checker
#   #   ### __init__.py
#   #   ### app.py
#   #   ### requirements.txt
#   ### stock_seller
#       ### __init__.py
#       ### app.py
#       ### requirements.txt
### statemachine
#   ### stock_trader.asl.json
### template.yaml
### tests
### unit
    ### __init__.py
    ### test_buyer.py
    ### test_checker.py
    ### test_seller.py
```

There are two especially interesting files that you can take a look at:

- `template.yaml`: Contains the AWS SAM template that defines your application's AWS resources.
- `statemachine/stockTrader.asl.json`: Contains the application's state machine definition, which is written in Amazon States Language (p. 23).

You can see the following entry in the `template.yaml` file, which points to the state machine definition file:

```
Properties:
```
It can be helpful to keep the state machine definition as a separate file instead of embedding it in the AWS SAM template. For example, tracking changes to the state machine definition is easier if you don’t include the definition in the template. You can use the Workflow Studio to create and maintain the state machine definition, and export the definition from the console directly to the Amazon States Language specification file without merging it into the template.

For more information about the sample application, see the README.md file in the project directory.

**Step 2: Build Your Application**

**Command to run:**

First change into the project directory (that is, the directory where the template.yaml file for the sample application is located; by default is sam-app), then run this command:

```
sam build
```

**Example output:**

```
Build Succeeded
Built Artifacts  : .aws-sam/build
Built Template   : .aws-sam/build/template.yaml

Commands you can use next
==========================
[*] Invoke Function: sam local invoke
[*] Deploy: sam deploy --guided
```

**What AWS SAM is doing:**

The AWS SAM CLI comes with abstractions for a number of Lambda runtimes to build your dependencies, and copies all build artifacts into staging folders so that everything is ready to be packaged and deployed. The sam build command builds any dependencies that your application has, and copies the build artifacts to folders under .aws-sam/build.

**Step 3: Deploy Your Application to the AWS Cloud**

**Command to run:**

```
sam deploy --guided
```

Follow the on-screen prompts. You can just respond with Enter to accept the default options provided in the interactive experience.

**What AWS SAM is doing:**

This command deploys your application to the AWS cloud. It takes the deployment artifacts you build with the sam build command, packages and uploads them to an Amazon S3 bucket created by AWS SAM CLI, and deploys the application using AWS CloudFormation. In the output of the deploy command you can see the changes being made to your AWS CloudFormation stack.
You can verify the example Step Functions state machine was successfully deployed by following these steps:

1. Sign in to the AWS Management Console and open the Step Functions console at https://console.aws.amazon.com/states/.
2. In the left navigation, choose State machines.
3. Find and choose your new state machine in the list. It will be named StockTradingStateMachine-<unique-hash>.
4. Choose the Definition tab.

You should now see a visual representation of your state machine. You can verify that the visual representation matches the state machine definition found in the statemachine/stockTrader.asl.json file of your project directory.

Troubleshooting

SAM CLI error: "no such option: --guided"

When executing sam deploy, you see the following error:

```
Error: no such option: --guided
```

This means that you are using an older version of the AWS SAM CLI that does not support the --guided parameter. To fix this, you can either update your version of AWS SAM CLI to 0.33.0 or later, or omit the --guided parameter from the sam deploy command.

SAM CLI error: "Failed to create managed resources: Unable to locate credentials"

When executing sam deploy, you see the following error:

```
Error: Failed to create managed resources: Unable to locate credentials
```

This means that you have not set up AWS credentials to enable the AWS SAM CLI to make AWS service calls. To fix this, you must set up AWS credentials. For more information, see Setting Up AWS Credentials in the AWS Serverless Application Model Developer Guide.

Clean Up

If you no longer need the AWS resources you created by running this tutorial, you can remove them by deleting the AWS CloudFormation stack that you deployed.

To delete the AWS CloudFormation stack created with this tutorial using the AWS Management Console, follow these steps:

2. In the left navigation pane, choose **Stacks**.
3. In the list of stacks, choose **sam-app** (or the name of stack you created).
4. Choose **Delete**.

When done, the status of the of the stack will change to **DELETE_COMPLETE**.

Alternatively, you can delete the AWS CloudFormation stack by executing the following AWS CLI command:

```bash
aws cloudformation delete-stack --stack-name sam-app --region region
```

**Verify Deleted Stack**

For both methods of deleting the AWS CloudFormation stack, you can verify it was deleted by going to the https://console.aws.amazon.com/cloudformation, choosing **Stacks** in the left navigation pane, and choosing **Deleted** in the dropdown to the right of the search text box. You should see your stack name **sam-app** (or the name of the stack you created) in the list of deleted stacks.

## Creating an Activity State Machine Using Step Functions

This tutorial shows you how to create an activity-based state machine using Java and AWS Step Functions. Activities allow you to control worker code that runs somewhere else from your state machine. For an overview, see Activities (p. 33) in How Step Functions works (p. 19).

To complete this tutorial, you need the following:

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

### Topics

- **Step 1: Create an Activity** (p. 162)
- **Step 2: Create a State Machine** (p. 163)
- **Step 3: Implement a Worker** (p. 164)
- **Step 4: Start an Execution** (p. 166)
- **Step 5: Run and Stop the Worker** (p. 166)

### Step 1: Create an Activity

You must make Step Functions aware of the **activity** whose **worker** (a program) you want to create. Step Functions responds with an Amazon Resource Name(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.
Step 2: Create a State Machine

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

1. In the Step Functions console, in the navigation pane on the left, choose State machines.
2. On the State machines page, choose Create state machine, and then choose Author with code snippets. For Type, choose Standard, and then enter a name for your state machine (for example, ActivityStateMachine).

   **Note**
   State machine, execution, and activity names must be 1–80 characters in length, must be unique for your account and AWS Region, and must not contain any of the following:
   - Whitespace
   - Wildcard characters (? *)
   - Bracket characters (< > { } [ ])
   - Special characters (: ; , | ^ ~ $ # % & ` ")
   - Control characters (\u0000 - \u000f or \u001f - \u007f).

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

Under State machine definition, enter the following code, and include the ARN of the activity task that you created earlier (p. 162) in the Resource field, as shown in the following example.

```json
{
  "Comment": "An example using a Task state.",
  "StartAt": "getGreeting",
  "Version": "1.0",
  "TimeoutSeconds": 300,
  "States":
  {
    "getGreeting": {
      "Type": "Task",
      "End": true
    }
  }
}
```

This is a description of your state machine using the Amazon States Language. It defines a single Task state named getGreeting. For more information, see State Machine Structure (p. 24).

3. Use the graph in the Visual Workflow pane to check that your Amazon States Language code describes your state machine correctly.
Step 3: Implement a Worker

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

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

Note
For a more complete example of an activity worker, see Example Activity Worker in Ruby (p. 34). This example provides an implementation based on best practices, which you can use as a reference for your activity worker. The code implements a consumer-producer pattern with a configurable number of threads for pollers and activity workers.

To implement the worker

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

```java
import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.EnvironmentVariableCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.stepfunctions.AWSStepFunctions;
```
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.setSocketTimeout((int) TimeUnit.SECONDS.toMillis(70));

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

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

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

    Note
    The EnvironmentVariableCredentialsProvider class in this example assumes
    that the AWS_ACCESS_KEY_ID (or AWS_ACCESS_KEY) and AWS_SECRET_KEY (or
    AWS_SECRET_ACCESS_KEY) environment variables are set. For more information about
    providing the required credentials to the factory, see AWSCredentialsProvider in the AWS
    SDK for Java API Reference and Set Up AWS Credentials and Region for Development in the
    AWS SDK for Java Developer Guide.
    By default the AWS SDK will wait up to 50 seconds to receive data from the server for any
    operation. The GetActivityTask operation is a long-poll operation that will wait up to
60 seconds for the next available task. To prevent receiving a SocketTimeoutException error, set SocketTimeout to 70 seconds.

3. In the parameter list of the GetActivityTaskRequest().withActivityArn() constructor, replace the ACTIVITY_ARN value with the ARN of the activity task that you created earlier (p. 162).

**Step 4: Start an Execution**

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

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

   The *New execution* page is displayed.

2. (Optional) To identify your execution, you can specify a name for it in the *Name* box. By default, Step Functions generates a unique execution name 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"
   }
   ```

4. Choose *Start Execution*.

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

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

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

**Step 5: Run and Stop the Worker**

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

1. On the command line, navigate to the directory in which you created GreeterActivities.java.

2. To use the AWS SDK, add the full path of the lib and third-party directories to the dependencies of your build file and to your Java CLASSPATH. For more information, see *Downloading and Extracting the SDK* in the *AWS SDK for Java Developer Guide*.

3. Compile the file.

   ```
   $ javac GreeterActivities.java
   ```

4. Run the file.

   ```
   $ java GreeterActivities
   ```

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

6. When the execution completes, choose *Output* to see the results of your execution.
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 quotas for AWS Step Functions, AWS Lambda, or other AWS services.

Before you begin, go through the Creating a Step Functions State Machine That Uses Lambda (p. 137) tutorial to ensure you are familiar with using Lambda and Step Functions together.

Topics
- Step 1: Create a Lambda Function to Iterate a Count (p. 167)
- Step 2: Test the Lambda Function (p. 168)
- Step 3: Create a State Machine (p. 169)
- Step 4: Start a New Execution (p. 171)

Step 1: Create a Lambda Function to Iterate a Count

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

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

To create the Lambda function

1. Sign in to the Lambda console, and then choose Create function.
2. On the Create function page, choose Author from scratch.
3. In the Basic information section, configure your Lambda function, as follows:
   a. For Function name, enter Iterator.
   b. For Runtime, choose Node.js 14.x.
   c. In Change default execution role, choose Create a new role with basic Lambda permissions.
   d. Choose Create function.
   e. After your Lambda function is created, copy the function's Amazon Resource Name (ARN) displayed in the upper-right corner of the page. To copy the ARN, click COPY. The following is an example ARN:

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

4. Copy the following code for the Lambda function into the Code source section of the Iterator page.

```javascript
exports.handler = function iterator (event, context, callback) {
```
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 Deploy.

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. Choose Test.
2. In the Configure test event dialog box, enter TestIterator in the Event name box.
3. Replace the example data with the following.

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

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

4. Choose Create.
5. On the Iterator page, choose Test to test your Lambda function.

The results of the test are displayed in the Execution results tab.

6. Choose the Execution results tab to see the output.

```
{
    "index": 6,
    "step": 1,
    "count": 10,
    "continue": true
}
```
Note
If you set index to 9 for this test, the index increments to 10, and continue is false.

Step 3: Create a State Machine

Use the Step Functions console to create a state machine with the following states to invoke the Lambda function that you created earlier in Step 1 (p. 167).

- **ConfigureCount** – Sets the default values for count, index, and step.
- **Iterator** – References the Lambda function you created earlier, passing in the values configured in ConfigureCount.
- **IsCountReached** – A choice state that either runs your sample work again or goes to Done, based on a Boolean value returned from your Iterator Lambda function.
- **ExampleWork** – A stub for the work you want to accomplish in your execution. In this example, it's a Pass state. In an actual implementation, this would be a Task state.
- **Done** – The end state of your execution.

1. Open 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 **Choose authoring method** page, choose **Write your workflow in code**.
3. For **Type**, retain the default selection, that is, **Standard**.
4. In the **Definition** pane, paste the following code, but replace the ARN of the Lambda function that you created earlier (p. 167) in the Resource field.

```
{
  "Comment": "Iterator State Machine Example",
  "StartAt": "ConfigureCount",
  "States": {
    "ConfigureCount": {
      "Type": "Pass",
      "Result": {
        "count": 10,
        "index": 0,
        "step": 1
      },
      "ResultPath": ".\$.iterator",
      "Next": "Iterator"
    },
    "Iterator": {
      "Type": "Task",
      "ResultPath": ".\$.iterator",
      "Next": "IsCountReached"
    },
    "IsCountReached": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$\$.iterator.continue",
          "BooleanEquals": true,
          "Next": "ExampleWork"
        }
      ]
    }
  }
}
```
Be sure to update the ARN in the Iterator state above, so that it references the Lambda function that you created earlier (p. 167).

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

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

For more information about the Amazon States Language, see State Machine Structure (p. 24).

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

6. Choose Next.
7. Enter a Name for your state machine, such as IterateCount.

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

- Whitespace
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.

8. In Permissions, choose Create new role.
9. Choose Create state machine.

Step 4: Start a New Execution

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


The Start execution dialog box is displayed.

2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name 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 Details tab, view the Execution Status and the timestamps for Started and End Time of the execution.

When the count reaches the number specified in the ConfigureCount state in your state machine, the execution quits iterating and ends.

---

**Continuing as a New Execution**

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

AWS Step Functions is designed to run workflows that have a finite duration and number of steps. Executions have a maximum duration of one year, and a maximum of 25,000 events (see Quotas (p. 485)).

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

This tutorial builds on the concept of using an external Lambda function to modify your workflow, which was demonstrated in the Iterating a Loop Using Lambda (p. 167) tutorial. You use the same Lambda function (Iterator) to iterate a loop for a specific number of times. In addition, you create another Lambda function to start a new execution of your workflow, and to decrement a count each time it starts a new execution. By setting the number of executions in the input, this state machine ends and restarts an execution a specified number of times.
The state machine you'll create implements the following states.

<table>
<thead>
<tr>
<th>State</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigureCount</td>
<td>A Pass (p. 28) state that configures the count, index, and step values that the Iterator Lambda function uses to step through iterations of work.</td>
</tr>
<tr>
<td>Iterator</td>
<td>A Task (p. 29) state that references the Iterator Lambda function.</td>
</tr>
<tr>
<td>IsCountReached</td>
<td>A Choice (p. 40) 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>
<tr>
<td>ShouldRestart</td>
<td>A Choice (p. 40) state that uses the executionCount value to decide if it should end one execution and start another, or simply end.</td>
</tr>
<tr>
<td>Restart</td>
<td>A Task (p. 29) state that uses a Lambda function to start a new execution of your state machine. Like the Iterator function, this function also decrements a count. It passes that value to the input of the new execution.</td>
</tr>
</tbody>
</table>
Prerequisites

Before you begin, go through the Creating a Step Functions State Machine That Uses Lambda (p. 137) 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. 174)
• Step 2: Create a Restart Lambda Function to Start a New Step Functions Execution (p. 176)
• Step 3: Create a State Machine (p. 176)
• Step 4: Update the IAM Policy (p. 179)
• Step 5: Run an Execution (p. 179)

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

Note

If you have completed the Iterating a Loop Using Lambda (p. 167) tutorial, you can skip this step and use that Lambda function.

This section and the Iterating a Loop Using Lambda (p. 167) tutorial show how you can use a Lambda function to track a count so that you can track the number of iterations of a loop in your state machine.

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

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

Create the Iterate Lambda function

1. Open the Lambda console, and then choose Create function.
2. In the Create function section, choose Author from scratch.
3. In the Author with code snippets section, configure your Lambda function, as follows:
   a. For Name, enter Iterator.
   b. For Runtime, choose Node.js 10.15.
   c. For Role, select Choose an existing role.
   d. For Existing role, choose the Lambda role that you created in the Creating a Step Functions State Machine That Uses Lambda (p. 137) 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 value of `continue`. The value of `continue` is `true` if `index` is less than `count`.

5. Choose **Save**.

**Test the Iterate Lambda Function**

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

**To test your Lambda function**

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

```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`. When the index is not less than the count, it returns `continue` as `false`. For this test, the index has already incremented to 5. The results should increment the index to 6 and set `continue` to `true`.

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

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

```json
{
    "index": 6,
    "step": 1,
    "count": 10,
    "continue": true
}
```
Step 2: Create a Restart Lambda Function to Start a New Step Functions Execution

1. Open the Lambda console, and then choose Create function.
2. In the Author with code snippets section, configure your Lambda function, as follows:
   a. For Name, enter Restart.
   b. For Runtime, choose Node.js 10.15.
   c. For Role, select Choose an existing role.
   d. Under Existing role, choose the role that includes the IAM policy you created previously.
   e. Choose Create function.

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

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

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

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

```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. 40) state. This state determines whether you should restart the execution.

```json
"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. 29) 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.

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

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

Example ContinueAsNew state machine

```json
{
  "Comment": "Continue-as-new State Machine Example",
  "StartAt": "ConfigureCount",
  "States": {
    "ConfigureCount": {
      "Type": "Pass",
      "Result": {
        "count": 100,
        "index": -1,
        "step": 1
      },
      "ResultPath": "$\.iterator",
      "Next": "Iterator"
    },
    "Iterator": {
      "Type": "Task",
      "ResultPath": "$\.iterator",
      "Next": "IsCountReached"
    },
    "IsCountReached": {
      "Type": "Choice",
      "Next": "Done"
    }
  }
}
```


```

"Choices": [
  {
    "Variable": "$\.iterator.continue",
    "BooleanEquals": true,
    "Next": "ExampleWork"
  }
],
"Default": "ShouldRestart"
),
"ExampleWork": {
  "Comment": "Your application logic, to run a specific number of times",
  "Type": "Pass",
  "Result": {
    "success": true
  },
  "ResultPath": "$\.result",
  "Next": "Iterator"
},
"ShouldRestart": {
  "Type": "Choice",
  "Choices": [
    {
      "Variable": "$\.restart.executionCount",
      "NumericGreaterThan": 0,
      "Next": "Restart"
    }
  ],
  "Default": "Done"
},
"Restart": {
  "Type": "Task",
  "Next": "Done"
},
"Done": {
  "Type": "Pass",
  "End": true
}
]
```

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

5. Choose Next.

6. Create or enter an IAM role:

   - To create an IAM role for Step Functions, select Create an IAM role for me, and enter a Name for your role.
   - If you have previously created an IAM role (p. 526) with the correct permissions for your state machine, select Choose an existing IAM role. Select a role from the list, or provide an ARN for that role.

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

7. Choose Create state machine.

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

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

```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 ARN for your ContinueAsNew state machine.

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

When all the executions are complete, you should see four successful executions in the list. The first execution that was started displays the name you chose, and subsequent executions have a generated name.
Using Code Snippets to Create a State to Send an Amazon SNS message

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

For more information about how Step Functions integrates with other AWS services directly from the Amazon States Language, see:
- Working with other services (p. 271)
- Code Snippets (p. 298)
- Pass Parameters to a Service API (p. 296)

Topics
- (Optional) Prerequisites (p. 181)
- Step 1: Generate a Code Snippet (p. 181)
- Step 2: Update Your State Machine Definition (p. 183)
- Step 3: Start an Execution (p. 185)

(Optional) Prerequisites

If you're using an existing role in AWS Identity and Access Management (IAM), you'll need to attach SNS:Publish permissions. You can use the AmazonSNSFullAccess AWS-managed policy to attach full permissions for Amazon SNS.

To attach permissions to a role
1. Open the Roles page.
2. Choose a role.
3. Choose Attach policies.
4. Type AmazonSNSFullAccess in the Filter policies text field.
5. Select the AmazonSNSFullAccess role.
6. Choose Attach policy.

Step 1: Generate a Code Snippet

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

2. Choose Author with code snippets. For Type, choose Standard, then enter a name for your state machine.

The default HelloWorld state machine is displayed in the State machine definition.
3. For **Generate Code Snippet**, choose **Amazon SNS: Publish a message**.

   The **Generate SNS Publish task state** window is displayed.

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

   Use the format [+][country code][subscriber number including area code]. For example: +12065550123.

5. Under **Message**, choose **Enter message**, and then enter some text to send as an SMS message.

   **Note**
   
   You can also choose **Specify message at runtime with state input**. This option enables you to use a reference path to select a message from the input of your state machine execution. For more information, see:

   - Input and Output Processing in Step Functions (p. 55)
   - Reference Paths (p. 57)
   - Pass State Input as Parameters Using Paths (p. 297)

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

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

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

Note
Under the Task state options section, you can also configure Retry, Catch, and TimeoutSeconds options. See Error handling in Step Functions (p. 88).

Step 2: Update Your State Machine Definition

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

1. After you have reviewed the code in the Preview section, choose Copy to clipboard.
2. Place your cursor after the closing bracket of the HelloWorld state in your state machine definition.
Enter a comma, press **Enter** to start a new line, and then paste your code snippet into your state machine definition.

3. Change the last line of the **Amazon SNS: Publish a message** state from "**Next**: "NEXT_STATE" to "**End**: true.

4. Change the last line of the **HelloWorld** state from "**End**: true to "**Next**: "Amazon SNS: Publish a message".

5. Choose in the **Visual Workflow** pane. Check the visual workflow to ensure your new state is included.

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

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

8. Create or enter an IAM role:
   - To create an IAM role for Step Functions, select **Create an IAM role for me**, and enter a **Name** for your role.
   - If you have previously created an IAM role (p. 526) with the correct permissions for your state machine, select **Choose an existing IAM role**. Select a role from the list, or provide an ARN for that role.

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

9. Choose **Create state machine**.

### Step 3: Start an Execution

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

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

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

   Choose **Start execution**.

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

### Deploying an Example Human Approval Project

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

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

- Amazon API Gateway resources
- An AWS Lambda function
- An AWS Step Functions state machine
- An Amazon Simple Notification Service email topic
- Related AWS Identity and Access Management roles and permissions
Note
You will need to provide a valid email address that you have access to when you create the AWS CloudFormation stack.

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

Topics
• Step 1: Create an AWS CloudFormation Template (p. 186)
• Step 2: Create a Stack (p. 186)
• Step 3: Approve the Amazon SNS Subscription (p. 187)
• Step 4: Run an Execution (p. 187)
• AWS CloudFormation Template Source Code (p. 188)

Step 1: Create an AWS CloudFormation Template

1. Copy the example code from the AWS CloudFormation Template Source Code (p. 188) section.

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

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

Step 2: Create a Stack

1. Log into the AWS CloudFormation console.
2. Choose Create Stack.
3. Under Choose a template, select Upload a template to Amazon S3 and then Choose File.
4. Browse to the human-approval.yaml file you created earlier that includes the template source code (p. 188).
5. Choose Open and then Next.
6. Under Specify Details enter a Stack name.
7. Under Parameters enter a valid Email address and choose Next.
8. On the Options page, scroll down and choose Next.
9. On the Review page, choose I acknowledge that AWS CloudFormation might create IAM resources and then choose Create.

   AWS CloudFormation begins to create your stack and displays the CREATE_IN_PROGRESS status. When the process is complete, AWS CloudFormation displays the CREATE_COMPLETE status.
10. (Optional) To display the resources in your stack, select the stack and choose the Resources tab.
Step 3: Approve the Amazon SNS Subscription

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

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

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

   ![Subscription confirmed!](image)

Step 4: Run an Execution

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

   ```json
   {
     "Comment": "Testing the human approval tutorial."
   }
   ``
6. Choose **Start execution**.
The **ApprovalTest** execution starts, and pauses at the **Lambda Callback** task.

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

   The message includes separate URLs for **Approve** and **Reject**.

8. Choose the **Approve** URL.

   The workflow continues based on your choice.

---

**AWS CloudFormation Template Source Code**

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

```yaml
AWSTemplateFormatVersion: "2010-09-09"
Description: "AWS Step Functions Human based task example. It sends an email with an HTTP URL for approval."
```
Parameters:
  Email:
    Type: String
    AllowedPattern: "^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"
    ConstraintDescription: Must be a valid email address.

Resources:
  # Begin API Gateway Resources
  ExecutionApi:
    Type: "AWS::ApiGateway::RestApi"
    Properties:
      Name: "Human approval endpoint"
      Description: "HTTP Endpoint backed by API Gateway and Lambda"
      FailOnWarnings: true
  ExecutionResource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref ExecutionApi
      ParentId: !GetAtt "ExecutionApi.RootResourceId"
      PathPart: execution
      ExecutionMethod:
        Type: "AWS::ApiGateway::Method"
        Properties:
          AuthorizationType: NONE
          HttpMethod: GET
          Integration:
            Type: AWS
            IntegrationHttpMethod: POST
            Uri: !Sub "arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/
            ${LambdaApprovalFunction.Arn}/invocations"
          IntegrationResponses:
            - StatusCode: 302
              ResponseParameters:
                method.response.header.Location: "integration.response.body.headers.Location"
          RequestTemplates:
            application/json: |
            #if($foreach.hasNext),#end
            "body" : $input.json('$'),
            "method": "$context.httpMethod",
            "headers": {
              #foreach($header in $input.params().header.keySet())
                "$header": "$util.escapeJavaScript($input.params().header.get($header))"
              #if($foreach.hasNext),#end
            }" #if($foreach.hasNext),#end
            "query": {
              #foreach($queryParam in $input.params().querystring.keySet())
                "$queryParam": "$util.escapeJavaScript($input.params().querystring.get($queryParam))"
              #if($foreach.hasNext),#end
            }
          ResourceId: !Ref ExecutionResource
          RestApiId: !Ref ExecutionApi
          MethodResponses:
- StatusCode: 302
  ResponseParameters:
    method.response.header.Location: true

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

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

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

ApiDeployment:
  Type: "AWS::ApiGateway::Deployment"
  DependsOn:
    - ExecutionMethod
  Properties:
    RestApiId: !Ref ExecutionApi
    StageName: DummyStage

# End API Gateway Resources

# Begin
# Lambda that will be invoked by API Gateway
LambdaApprovalFunction:
  Type: 'AWS::Lambda::Function'
  Properties:
    Code:
      Fn::Sub: |
        const AWS = require('aws-sdk');
        var redirectToStepFunctions = function(lambdaArn, statemachineName, executionName, callback) {
          const lambdaArnTokens = lambdaArn.split(":");

        

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

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

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

callback(null, {
    statusCode: 302,
    headers: {
        Location: url
    }
});

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

    const stepfunctions = new AWS.StepFunctions();
    stepfunctions.sendTaskSuccess({
        output: JSON.stringify(message),
        taskToken: event.query.taskToken
    }).promise().then(function(data) {
        redirectToStepFunctions(context.invokedFunctionArn, statemachineName, executionName, callback);
    }).catch(function(err) {
        console.error(err, err.stack);
        callback(err);
    });

    Description: Lambda function that callback to AWS Step Functions
    FunctionName: LambdaApprovalFunction
    Handler: index.handler
    Role: !GetAtt "LambdaApiGatewayIAMRole.Arn"
    Runtime: nodejs12.x
    LambdaApiGatewayInvoke:
    Type: "AWS::Lambda::Permission"
    Properties:
    Action: "lambda:InvokeFunction"
```
FunctionName: !GetAtt "LambdaApprovalFunction.Arn"
Principal: "apigateway.amazonaws.com"
SourceArn: !Sub "arn:aws:execute-api:${AWS::Region}:${AWS::AccountId}:${ExecutionApi}/*"

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

# End Lambda that will be invoked by API Gateway

# Begin state machine that publishes to Lambda and sends an email with the link for approval
HumanApprovalLambdaStateMachine:
  Type: AWS::StepFunctions::StateMachine
  Properties:
    RoleArn: !GetAtt LambdaStateMachineExecutionRole.Arn
    DefinitionString:
      Fn::Sub: |
      {
        "StartAt": "Lambda Callback",
        "TimeoutSeconds": 3600,
        "States": { 
          "Lambda Callback": {
            "Type": "Task",
            "Resource": "arn:aws:lambda:${AWS::Region}:${AWS::AccountId}:function:LambdaHumanApprovalSendEmailFunction",
            "Parameters": {
              "FunctionName": "${LambdaHumanApprovalSendEmailFunction.Arn}"
           }
          },
          "ManualApprovalChoiceState": {
            "Type": "Choice",
            "Choices": [
              { "Variable": "$.Status", 
            }
"StringEquals": "Approved! Task approved by ${Email}"
},
{  "Variable": "$.Status",
  "StringEquals": "Rejected! Task rejected by ${Email}"
  }
},
"ApprovedPassState": {
  "Type": "Pass",
  "End": true
},
"RejectedPassState": {
  "Type": "Pass",
  "End": true
}
}

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

LambdaHumanApprovalSendEmailFunction:
  Type: "AWS::Lambda::Function"
  Properties:
    Handler: "index.lambda_handler"
    Role: !GetAtt LambdaSendEmailExecutionRole.Arn
    Runtime: "nodejs12.x"
    Timeout: "25"
    Code:
      ZipFile:
        Fn::Sub: |
          console.log('Loading function');
          const AWS = require('aws-sdk');
          exports.lambda_handler = (event, context, callback) => {
            console.log('event= ' + JSON.stringify(event));
            console.log('context= ' + JSON.stringify(context));

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

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

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

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

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

            const approveEndpoint = apigwEndpint + "/execution?action=approve&ex=" +
                executionName + "&asm=StdStringMachineName + "&taskToken=" + encodeURIComponent(taskToken);
            console.log('approveEndpoint= ' + approveEndpoint);

            const rejectEndpoint = apigwEndpint + "/execution?action=reject&ex=" +
                executionName + "&asm=StdStringMachineName + "&taskToken=" + encodeURIComponent(taskToken);
```javascript
console.log('rejectEndpoint= ' + rejectEndpoint);

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

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

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

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

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

LambdaSendEmailExecutionRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
          Action: "sts:AssumeRole"
    Policies:
      - PolicyName: CloudWatchLogsPolicy
```
View X-Ray traces in Step Functions

In this tutorial, you will learn how to use X-Ray to trace errors that occur when running a state machine. You can use AWS X-Ray to visualize the components of your state machine, identify performance bottlenecks, and troubleshoot requests that resulted in an error. In this tutorial, you will create several Lambda functions that randomly produce errors, which you can then trace and analyze using X-Ray.

The Creating a Step Functions State Machine That Uses Lambda (p. 137) tutorial walks you through creating a state machine that calls a Lambda function. If you have completed that tutorial, skip to Step 2 (p. 197) and use the AWS Identity and Access Management (IAM) role that you previously created.

Topics

- Step 1: Create an IAM Role for Lambda (p. 195)
- Step 2: Create a Lambda Function (p. 196)
- Step 3: Create two more Lambda functions (p. 197)
- Step 4: Create a State Machine (p. 197)
- Step 5: Start a New Execution (p. 199)

Step 1: Create an IAM Role for Lambda

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

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

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

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

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

Step 2: Create a Lambda Function

Your Lambda function will randomly throw errors or time out, producing example data to view in X-Ray.

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

1. Open the Lambda console and choose Create function.
2. In the Create function section, choose Author from scratch.
3. In the Basic information section, configure your Lambda function:
   a. For Function name, enter TestFunction1.
   b. For Runtime, choose Node.js 12.x.
   c. For Role, select Choose an existing role.
   d. For Existing role, select the Lambda role that you created earlier (p. 195).

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

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

   arn:aws:lambda:us-east-1:123456789012:function:TestFunction1

4. Copy the following code for the Lambda function into the Function code section of the TestFunction1 page.

```javascript
function getRandomSeconds(max) {
    return Math.floor(Math.random() * Math.floor(max)) * 1000;
}
function sleep(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}
exports.handler = async (event, context) => {
    if(getRandomSeconds(4) === 0) {
```
Step 3: Create two more Lambda functions

Create two more Lambda functions.

1. Repeat Step 2 to create two more Lambda functions. For the next function, in Function name, enter TestFunction2. For the last function, in Function name, enter TestFunction3.

2. In the Lambda console, check that you now have three Lambda functions, TestFunction1, TestFunction2, and TestFunction3.

Step 4: Create a State Machine

Use the Step Functions console to create a state machine with three Task states. Each Task state will reference one of your three Lambda functions.

1. Open the Step Functions console and choose Create a state machine.

2. On the Define state machine page, choose Author with code snippets. For Type, choose Standard.

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

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

3. In the State machine definition pane, replace the example with the following state machine definition. For each Task task state, use the ARN of the corresponding Lambda function that you created earlier (p. 196), as shown in the following example.

```json
{
    "StartAt": "CallTestFunction1",
    "States": {
        "CallTestFunction1": {
            "Type": "Task",
            "Catch": [
                {
```
This is a description of your state machine using the Amazon States Language. It defines three Task states named CallTestFunction1, CallTestFunction2 and CallTestFunction3. Each calls one of your three Lambda functions. For more information, see State Machine Structure (p. 24).

Choose Next.

4. Enter a Name, for example, TraceFunctions.

5. Create or enter an IAM role:
   - To create an IAM role for Step Functions, select Create an IAM role for me, and enter a Name for your role.
Step 5: Start a New Execution

After you create your state machine, start an execution.

1. On the TraceFunctions page, choose Start execution.

   The New execution page is displayed.

2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name 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. Run several (at least three) executions.

4. After the executions have finished, follow the X-Ray trace map link. You can view the trace while an execution is still running, but you may want to see the execution results before viewing the X-Ray trace map.

5. View the service map to identify where errors are occurring, connections with high latency, or traces for requests that were unsuccessful. In this example, you can see how much traffic each function is receiving. TestFunction2 was called more often than TestFunction3, and TestFunction1 was called more than twice as often as TestFunction2.
The service map indicates the health of each node by coloring it based on the ratio of successful calls to errors and faults:

- **Green** for successful calls
- **Red** for server faults (500 series errors)
- **Yellow** for client errors (400 series errors)
- **Purple** for throttling errors (429 Too Many Requests)

You can also choose a service node to view requests for that node, or an edge between two nodes to view requests that traveled that connection.

6. View the X-Ray trace map to see what has happened for each execution. The Timeline view shows a hierarchy of segments and subsegments. The first entry in the list is the segment, which represents all data recorded by the service for a single request. Below the segment are subsegments. This example shows subsegments recorded by the Lambda functions.

For more information on understanding X-Ray traces and using X-Ray with Step Functions, see the AWS X-Ray and Step Functions (p. 512)
Gather Amazon S3 bucket info using AWS SDK service integrations

This tutorial shows you how to deploy an AWS SDK integration (p. 274) project that will create one Standard state machine and one Express state machine, each with the same definition. These will gather information about your Amazon S3 buckets, then list your buckets along with version information for each bucket in the current region.

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

- An AWS Step Functions Standard state machine
- An AWS Step Functions Express state machine
- Related AWS Identity and Access Management roles and permissions

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

Topics

- Step 1: Create an AWS CloudFormation Template (p. 201)
- Step 2: Create a Stack (p. 201)
- Step 3: View your state machine in Workflow Studio (p. 202)
- Step 4: Run a Standard state machine execution (p. 203)
- Step 5: Run an Express state machine execution (p. 204)
- AWS CloudFormation Template Source Code (p. 204)

Step 1: Create an AWS CloudFormation Template

1. Copy the example code from the AWS CloudFormation Template Source Code (p. 204) section.

   ![Rights Reserved.](Copy)

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

   For this example the file is called `aws-sdk.yaml`.

Step 2: Create a Stack

1. Log into the AWS CloudFormation console.
2. Choose Create Stack.
3. Under **Choose a template**, select **Upload a template to Amazon S3** and then **Choose File**.
4. Browse to the `aws-sdk.yaml` file you created earlier that includes the template source code (p. 204).
5. Choose **Open** and then **Next**.
6. Under **Specify Details** enter a **Stack name**.
7. On the **Options** page, scroll down and choose **Next**.
8. On the **Review** page, choose:
   - I acknowledge that AWS CloudFormation might create IAM resources.
   - I acknowledge that AWS CloudFormation might create IAM resources with custom names.
   - I acknowledge that AWS CloudFormation might require the following capability: **CAPABILITY_AUTO_EXPAND**
9. Choose **Create**. AWS CloudFormation begins to create your stack and displays the **CREATE_IN_PROGRESS** status. When the process is complete, AWS CloudFormation displays the **CREATE_COMPLETE** status.
10. (Optional) To display the resources in your stack, select the stack and choose the **Resources** tab.

### Step 3: View your state machine in Workflow Studio

1. Sign in to the **Step Functions console**.
2. Choose one of your new state machines, either **Gather-S3-Bucket-Info-Standard** or **Gather-S3-Bucket-Info-Express**.

   ![State machines](image)

3. Choose **Edit**.
   
   The definition of your state machine is shown.
4. Choose **Workflow Studio**. You can view and edit your state machine in Step Functions Workflow Studio.
Step 4: Run a Standard state machine execution

1. If you are not already signed in, sign in to the Step Functions console.
2. On the State machines page, choose Gather-S3-Bucket-Info-Standard.
3. Choose Start execution.
4. (Optional) enter some input for the execution.

```json
{
    "Comment": "Testing the AWS SDK tutorial."
}
```
5. Choose Start execution.

The Gather-S3-Bucket-Info-Standard execution starts.
6. After the execution finishes, you can view the results in the Execution output tab.
Step 5: Run an Express state machine execution

1. If you are not already signed in, sign in to the Step Functions console.
2. On the State machines page, choose Gather-S3-Bucket-Info-Express.
3. Choose Start execution.
4. (Optional) enter some input for the execution.

```json
{
  "Comment": "Testing the AWS SDK Express tutorial."
}
```

5. Choose Start execution.

The Gather-S3-Bucket-Info-Express execution starts.

6. After the execution finishes, you can view the results in the Execution output section of the Details.

AWS CloudFormation Template Source Code

Use this AWS CloudFormation template to deploy an example of an AWS SDK integration for AWS Step Functions.

```yaml
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: MIT-0
AWSTemplateFormatVersion: "2010-09-09"
Transform: AWS::Serverless-2016-10-31
Description: "An example of using Aws-SDK service integrations to gather info on S3 buckets"
```
Resources:
  ApplicationRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: "Allow"
            Principal:
              Service:
                - states.amazonaws.com
            Action:
              - "sts:AssumeRole"
        Policies:
          - PolicyName: AppPolicy
            PolicyDocument:
              Version: 2012-10-17
              Statement:
                - Effect: Allow
                  Action:
                    - xray:PutTraceSegments
                    - xray:PutTelemetryRecords
                    - xray:GetSamplingRules
                    - xray:GetSamplingTargets
                    - s3:ListAllMyBuckets
                    - s3:GetBucket
                    - s3:GetBucketVersioning
                    - s3:GetBucketLocation
                  Resource: '*'

StateMachineGetS3BucketInfoStandard:
  Type: AWS::Serverless::StateMachine
  Properties:
    Type: "STANDARD"
    Name: "Gather-S3-Bucket-Info-Standard"
    Role: !GetAtt ApplicationRole.Arn
  Definition:
    Comment: An example of using Aws-SDK service integrations to gather info on S3 buckets
    StartAt: ListBuckets
    States:
      ListBuckets:
        Type: Task
        Parameters: {}
        Resource: arn:aws:states::aws-sdk:s3:listBuckets
        Next: Map
        Retry:
          - ErrorEquals:
            - States.ALL
          BackoffRate: 1
          IntervalSeconds: 1
          MaxAttempts: 3
      Map:
        Type: Map
        Iterator:
          StartAt: GetBucketLocation
          States:
            GetBucketLocation:
              Type: Task
              Parameters:
                Bucket.$: "$.Name"
              Resource: arn:aws:states::aws-sdk:s3:getBucketLocation
              ResultPath: "$.BucketLocationInfo"
Next: Is In Current Region?
Retry:
- ErrorEquals:
  - States.ALL
BackoffRate: 1
IntervalSeconds: 1
MaxAttempts: 3

Is In Current Region?:
Type: Choice
Choices:
- And:
  - Variable: "$.BucketLocationInfo.LocationConstraint"
    IsPresent: true
  - Variable: "$.BucketLocationInfo.LocationConstraint"
    StringEquals: ca-central-1
Next: GetBucketVersioning
Default: Add Unknown Version Info

Add Unknown Version Info:
Type: Pass
End: true
ResultPath: "$.BucketVersioningInfo"

GetBucketVersioning:
Type: Task
End: true
Parameters:
  Bucket: "$.Name"
ResultPath: "$.BucketVersioningInfo"
Retry:
- ErrorEquals:
  - States.ALL
BackoffRate: 1
IntervalSeconds: 1
MaxAttempts: 3
End: true
ItemsPath: "$.Buckets"

StateMachineGetS3BucketInfoExpress:
Type: AWS::Serverless::StateMachine
Properties:
  Type: "EXPRESS"
  Name: "Gather-S3-Bucket-Info-Express"
  Role: !GetAtt ApplicationRole.Arn
  Definition:
    Comment: An example of using Aws-SDK service integrations to gather info on S3 buckets
    StartAt: ListBuckets
    States:
      ListBuckets:
        Type: Task
        Parameters: {}
        Next: Map
        Retry:
          - ErrorEquals:
            - States.ALL
            BackoffRate: 1
            IntervalSeconds: 1
            MaxAttempts: 3
        Map:
          Type: Map
          Iterator:
            StartAt: GetBucketLocation
            States:
GetBucketLocation:
  Type: Task
  Parameters:
    Bucket.$: "$.Name"
  ResultPath: "$.BucketLocationInfo"
  Next: Is In Current Region?
  Retry:
    - ErrorEquals:
      - States.ALL
    - BackoffRate: 1
    - IntervalSeconds: 1
    - MaxAttempts: 3
  Is In Current Region?:
    Type: Choice
    Choices:
      - And:
        - Variable: "$.BucketLocationInfo.LocationConstraint"
          IsPresent: true
        - Variable: "$.BucketLocationInfo.LocationConstraint"
          StringEquals: ca-central-1
        - Next: GetBucketVersioning
    Default: Add Unknown Version Info
  Add Unknown Version Info:
    Type: Pass
    End: true
    ResultPath: "$.BucketVersioningInfo"
    Result:
      Status: Unknown
  GetBucketVersioning:
    Type: Task
    End: true
    Parameters:
      Bucket.$: "$.Name"
    ResultPath: "$.BucketVersioningInfo"
    Retry:
      - ErrorEquals:
        - States.ALL
      - BackoffRate: 1
      - IntervalSeconds: 1
      - MaxAttempts: 3
    End: true
    ItemsPath: "$.Buckets"
Developer tools

The following resources contain additional information about building serverless workflows and working with state machines:

- AWS CDK
- AWS Toolkit for VS Code

The topics below contain information that teach you how to create, test, and debug state machines.

Topics
- Development options (p. 208)
- AWS Step Functions and AWS SAM (p. 215)
- Creating a Lambda state machine for Step Functions using AWS CloudFormation (p. 217)
- Creating a Lambda State Machine for Step Functions Using the AWS CDK (p. 225)
- Creating an API Gateway REST API with Synchronous Express State Machine Using the AWS CDK (p. 234)
- AWS Step Functions Data Science SDK for Python (p. 247)
- Testing Step Functions State Machines Locally (p. 247)

Development options

You can implement your AWS Step Functions state machines in several ways, such as using the console, the SDKs, or a local version of Step Functions for testing and development.

Topics
- Step Functions console (p. 208)
- AWS SDKs (p. 209)
- Standard and Express workflows (p. 209)
- HTTPS service API (p. 209)
- Development environments (p. 209)
- Endpoints (p. 210)
- AWS CLI (p. 210)
- Step Functions Local (p. 210)
- AWS Serverless Application Model and Step Functions (p. 211)
- Definition format support (p. 211)

Step Functions console

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

The Creating a Lambda State Machine (p. 137) tutorial uses this technique to create a simple state machine, execute it, and view its results.

Data flow simulator

You can design, implement and debug workflows in the Step Functions console. You can also control the flow of data through your workflow by learning about and using JsonPath input and output processing. Use the data flow simulator in the Step Functions console to learn how information flows from state to state, and to understand how to filter and manipulate data. This tool simulates each of the fields (p. 55) that Step Functions uses to process data, such as InputPath, Parameters, ResultSelector, OutputPath, and ResultPath.

AWS SDKs

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

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

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

Note

Step Functions only supports HTTPS endpoints.

Standard and Express workflows

When you create a new state machine, you must select a Type of either Standard or Express. In both cases, you define your state machine using the Amazon States Language. Your state machine executions will behave differently, depending on which Type you select. The Type you choose can’t be changed after your state machine is created.

See Logging using CloudWatch Logs (p. 508) for more information.

HTTPS service API

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

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

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

Development environments

You must set up a development environment that is compatible with the programming language that you plan to use.
For example, to develop for Step Functions using Java, you must install a Java development environment, such as the AWS SDK for Java, on each of your development workstations. If you use Eclipse IDE for Java Developers, you should also install the AWS Toolkit for Eclipse. This Eclipse plugin adds features that are useful for developing on AWS.

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

Endpoints

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

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

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

AWS CLI

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

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

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

Step Functions Local

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

The local version of Step Functions can invoke AWS Lambda functions, both in AWS and when running locally. You can also coordinate other supported AWS services (p. 271). For more information, see Testing Step Functions State Machines Locally (p. 247).

AWS Toolkit for Visual Studio Code

You can use VS Code to interact with remote state machines and develop state machines locally. You can create or update state machines, list existing state machines, and execute or download a state machine. VS Code also lets you create new state machines from templates, see a visualization of your state machine, and provides code snippets, code completion, and code validation.

For more information, see the AWS Toolkit for Visual Studio Code User Guide
AWS Step Functions Developer Guide
AWS Serverless Application Model and Step Functions

AWS Serverless Application Model and Step Functions

Step Functions is integrated with the AWS Serverless Application Model, which lets you integrate workflows with Lambda functions, APIs and events to create serverless applications.

You can also use the AWS SAM CLI in conjunction with the AWS Toolkit for Visual Studio Code as part of an integrated experience.

For more information, see AWS Step Functions and AWS SAM (p. 215).

Definition format support

Step Functions offers a variety of tools that lets you provide your state machine definitions in different formats. An Amazon States Language (ASL) definition that specifies the details of your state machine can be provided as either a string, or as a serialized object using JSON or YAML.

**Note**

YAML allows single line comments. Any YAML comments provided in the state machine definition portion of a template will not be carried forward into the created resource's definition. Instead, you can use the `Comment` property within the state machine definition. For more information, see the State Machine Structure (p. 24) page.

The following table shows which tools support ASL-based definitions.

Definition format support by tool

<table>
<thead>
<tr>
<th>Tool</th>
<th>JSON</th>
<th>YAML</th>
<th>Stringified Amazon States Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step Functions Console (p. 208)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPS Service API (p. 209)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>AWS CLI (p. 210)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Step Functions Local (p. 210)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>AWS SAM (p. 211)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>AWS CloudFormation (p. 217)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Note**

AWS CloudFormation and AWS SAM also allow you to upload your state machine definitions to Amazon S3 in JSON or YAML format, and to provide the definition's Amazon S3 location.
in the template. This can improve the readability of your templates when your state machine definition is complex. For more information see the AWS::StepFunctions::StateMachine S3Location page.

The following example AWS CloudFormation templates show how you can provide the same state machine definition using different input formats.

JSON with Definition

```json
{
   "AWSTemplateFormatVersion": "2010-09-09",
   "Description": "AWS Step Functions sample template.",
   "Resources": {
      "MyStateMachine": {
         "Type": "AWS::StepFunctions::StateMachine",
         "Properties": {
            "RoleArn": {
               "Fn::GetAtt": [ "StateMachineRole", "Arn" ]
            },
            "TracingConfiguration": {
               "Enabled": true
            },
            "Definition": {
               "StartAt": "HelloWorld",
               "States": {
                  "HelloWorld": {
                     "Type": "Pass",
                     "End": true
                  }
               }
            }
         }
      },
      "StateMachineRole": {
         "Type": "AWS::IAM::Role",
         "Properties": {
            "AssumeRolePolicyDocument": {
               "Version": "2012-10-17",
               "Statement": [
                  {
                     "Action": ["sts:AssumeRole"],
                     "Effect": "Allow",
                     "Principal": {
                        "Service": ["states.amazonaws.com"]
                     }
                  }
               ]
            },
            "ManagedPolicyArns": [],
            "Policies": [
               {
                  "PolicyName": "StateMachineRolePolicy",
                  "PolicyDocument": {
                     "Statement": [
                        {
                           "Action": ["lambda:InvokeFunction"],
                           "Resource": "*",
                           "Effect": "Allow"
                        }
                     ]
                  }
               }
            ]
         }
      }
   }
}
```
AWS Step Functions Developer Guide
Definition format support

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "AWS Step Functions sample template.",
  "Resources": {
    "MyStateMachine": {
      "Type": "AWS::StepFunctions::StateMachine",
      "Properties": {
        "RoleArn": {
          "Fn::GetAtt": [ "StateMachineRole", "Arn" ]
        },
        "TracingConfiguration": {
          "Enabled": true
        },
        "DefinitionString": "{
          "StartAt": "HelloWorld",
          "States": {
            "HelloWorld": {"Type": "Pass", "End": true}
          }
        }
      }
    },
    "StateMachineRole": {
      "Type": "AWS::IAM::Role",
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
            {"Action": ["sts:AssumeRole"],
             "Effect": "Allow",
             "Principal": {
               "Service": ["states.amazonaws.com"]
             }
          ],
          "ManagedPolicyArns": [],
          "Policies": [
            {
              "PolicyName": "StateMachineRolePolicy",
              "PolicyDocument": {
                "Statement": [
                  {"Action": ["lambda:InvokeFunction"],
                   "Resource": "*"}
                ]
              }
            }
          ]
        }
      }
    }
  }
}
```

JSON with DefinitionString

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "AWS Step Functions sample template.",
  "Resources": {
    "MyStateMachine": {
      "Type": "AWS::StepFunctions::StateMachine",
      "Properties": {
        "RoleArn": {
          "Fn::GetAtt": [ "StateMachineRole", "Arn" ]
        },
        "TracingConfiguration": {
          "Enabled": true
        },
        "DefinitionString": "{
          "StartAt": "HelloWorld",
          "States": {
            "HelloWorld": {"Type": "Pass", "End": true}
          }
        }
      }
    },
    "StateMachineRole": {
      "Type": "AWS::IAM::Role",
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
            {"Action": ["sts:AssumeRole"],
             "Effect": "Allow",
             "Principal": {
               "Service": ["states.amazonaws.com"]
             }
          ],
          "ManagedPolicyArns": [],
          "Policies": [
            {
              "PolicyName": "StateMachineRolePolicy",
              "PolicyDocument": {
                "Statement": [
                  {"Action": ["lambda:InvokeFunction"],
                   "Resource": "*"}
                ]
              }
            }
          ]
        }
      }
    }
  }
}
```

213
"Effect": "Allow"
}
}
}
}
}

"Outputs": {
"StateMachineArn": {
"Value": {
"Ref": "MyStateMachine"
}
}
}

YAML with Definition

AWSTemplateFormatVersion: 2010-09-09
Description: AWS Step Functions sample template.
Resources:
  MyStateMachine:
    Type: 'AWS::StepFunctions::StateMachine'
    Properties:
      RoleArn: !GetAtt
        - StateMachineRole
        - Arn
      TracingConfiguration:
        Enabled: true
      Definition:
        # This is a YAML comment. This will not be preserved in the state machine
        # resource's definition.
        Comment: This is an ASL comment. This will be preserved in the state machine
        # resource's definition.
        StartAt: HelloWorld
        States:
          HelloWorld:
            Type: Pass
            End: true
  StateMachineRole:
    Type: 'AWS::IAM::Role'
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          - Action:
            - 'sts:AssumeRole'
            Effect: Allow
            Principal:
              Service:
                - states.amazonaws.com
        ManagedPolicyArns: []
      Policies:
        - PolicyName: StateMachineRolePolicy
          PolicyDocument:
            Statement:
              - Action:
                - 'lambda:InvokeFunction'
                Resource: "*"
                Effect: Allow

Outputs:
  StateMachineArn:
YAML with DefinitionString

```
AWSTemplateFormatVersion: 2010-09-09
Description: AWS Step Functions sample template.
Resources:
  MyStateMachine:
    Type: "AWS::StepFunctions::StateMachine"
    Properties:
      RoleArn: !GetAtt
        - StateMachineRole
        - Arn
      TracingConfiguration:
        Enabled: true
      DefinitionString: |
        {
          "StartAt": "HelloWorld",
          "States": {
            "HelloWorld": {
              "Type": "Pass",
              "End": true
            }
          }
        }
  StateMachineRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          - Action:
            - 'sts:AssumeRole'
            Effect: Allow
            Principal:
              Service:
                - states.amazonaws.com
          ManagedPolicyArns: []
        Policies:
          - PolicyName: StateMachineRolePolicy
            PolicyDocument:
              Statement:
                - Action:
                  - 'lambda:InvokeFunction'
                  Resource: "*"
                  Effect: Allow

Outputs:
  StateMachineArn:
    Value:
      Ref: MyStateMachine
```

AWS Step Functions and AWS SAM

You can use the AWS SAM CLI in conjunction with the AWS Toolkit for Visual Studio Code as part of an integrated experience to create state machines locally. You can build a serverless application with AWS SAM, then build out your state machine in the VS Code IDE. Then you can validate, package, and deploy your resources. Optionally, you can also publish to the AWS Serverless Application Repository.
Why use Step Functions with AWS SAM?

When you use Step Functions with AWS SAM you can:

- Get started using a AWS SAM sample template.
- Build your state machine into your serverless application.
- Use variable substitution to substitute ARNs into your state machine at the time of deployment.
- Specify your state machine's role using AWS SAM policy templates.
- Initiate state machine executions with API Gateway, EventBridge events, or on a schedule within your AWS SAM template.

Step Functions integration with the AWS SAM specification

You can use the AWS SAM Policy Templates to add permissions to your state machine. With these permissions, you can orchestrate Lambda functions and other AWS resources to form complex and robust workflows.

Step Functions integration with the SAM CLI

Step Functions is integrated with the AWS SAM CLI. Use this to quickly develop a state machine into your serverless application.

Try the Create a Step Functions State Machine Using AWS SAM (p. 157) tutorial to learn how to use AWS SAM to create state machines.

Supported AWS SAM CLI functions include:

<table>
<thead>
<tr>
<th>CLI Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sam init</td>
<td>Initializes a Serverless Application with an AWS SAM template. Can be used with a SAM template for Step Functions.</td>
</tr>
<tr>
<td>sam validate</td>
<td>Validates an AWS SAM template.</td>
</tr>
<tr>
<td>sam package</td>
<td>Packages an AWS SAM application. It creates a ZIP file of your code and dependencies, and then uploads it to Amazon S3. It then returns a copy of your AWS SAM template, replacing references to local artifacts with the Amazon S3 location where the command uploaded the artifacts.</td>
</tr>
<tr>
<td>sam deploy</td>
<td>Deploys an AWS SAM application.</td>
</tr>
<tr>
<td>sam publish</td>
<td>Publish an AWS SAM application to the AWS Serverless Application Repository. This command takes a packaged AWS SAM template and publishes the application to the specified region.</td>
</tr>
</tbody>
</table>

Note

When using AWS SAM local, you can emulate Lambda and API Gateway locally. However, you can't emulate Step Functions locally using AWS SAM.

You can learn more about using Step Functions with AWS SAM with the following resources:
Creating a Lambda state machine for Step Functions using AWS CloudFormation

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

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

Topics

- Step 1: Set up your AWS CloudFormation template (p. 217)
- Step 2: Use the AWS CloudFormation template to create a Lambda State Machine (p. 221)
- Step 3: Start a State Machine execution (p. 224)

Step 1: Set up your AWS CloudFormation template

Before you use the example templates (p. 221), you should understand how to declare the different parts of an AWS CloudFormation template.

Topics

- To create an IAM role for Lambda (p. 217)
- To create a Lambda function (p. 218)
- To create an IAM role for the state machine execution (p. 219)
- To create a Lambda state machine (p. 220)

To create an IAM role for Lambda

Define the trust policy associated with the IAM role for the Lambda function. The following examples define a trust policy using either YAML or JSON.

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 for a Lambda function that will print the message Hello World.

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

YAML

MyLambdaFunction:
  Type: "AWS::Lambda::Function"
  Properties:
    Handler: "index.handler"
    Role: !GetAtt [ LambdaExecutionRole, Arn ]
  Code:
    ZipFile: |
      exports.handler = (event, context, callback) => {
        callback(null, "Hello World!");
      };
  Runtime: "nodejs12.x"
  Timeout: "25"

JSON

"MyLambdaFunction": {
  "Type": "AWS::Lambda::Function",
  "Properties": {
    "Handler": "index.handler",
    "Role": {
      "Fn::GetAtt": [
        "LambdaExecutionRole",
        "Arn"
      ]
    },
    "Code": {
      "ZipFile": "exports.handler = (event, context, callback) => {
        callback(null, "Hello World!");
      };
    },
    "Runtime": "nodejs12.x",
    "Timeout": "25"
}
To create an IAM role for the state machine execution

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

**YAML**

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

**JSON**

```json
"StatesExecutionRole": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": [
              {"Fn::Sub": "states.
#{AWS::Region}.amazonaws.com"
            }
          ],
          "Action": "sts:AssumeRole"
        }
      ],
      "Path": "/",
      "Policies": [
        {"PolicyName": "StatesExecutionPolicy",
         "PolicyDocument": {
           "Version": "2012-10-17",
           "Statement": [
             {"Effect": "Allow",
              "Action": [
```
To create a Lambda state machine

Define the Lambda state machine.

**YAML**

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

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

**JSON**

```
"MyStateMachine": {
  "Type": "AWS::StepFunctions::StateMachine",
  "Properties": {
    "DefinitionString": {
      "Fn::Sub": [
        "{"n \"Comment\": \\
        \"A Hello World example using an AWS Lambda function\",
        \"StartAt\": \\
        \"HelloWorld\",n \"States\": {
        \"HelloWorld\": {
          \"Type\": \"Task\",
          \"Resource\": \\
          \"${lambdaArn}\",
          \"End\": true
        }
      }
    },
    lambdaArn: {
      "Fn::GetAtt": [
        "MyLambdaFunction",
        "Arn"
      ]
    },
    RoleArn: {
      "Fn::GetAtt": [
        "StatesExecutionRole",
        "Arn"
      ]
    }
  }
}
```
Step 2: Use the AWS CloudFormation template to create a Lambda State Machine

Once you understand the components of the AWS CloudFormation template, you can put them together and use the template to create an AWS CloudFormation stack.

To create the Lambda state machine

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

YAML

```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: "nodejs12.x"
      Timeout: "25"

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

221
Step 2: Use the AWS CloudFormation template to create a Lambda State Machine

```json
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "An example template with an IAM role for a Lambda state machine.",
    "Resources": {
        "LambdaExecutionRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            },
                            "Action": "sts:AssumeRole"
                        }
                    ]
                }
            }
        },
        "MyLambdaFunction": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Handler": "index.handler",
                "Role": {
                    "Fn::GetAtt": [
                        "LambdaExecutionRole",
                        "Arn"
                    ]
                },
                "Code": {
                    "ZipFile": "exports.handler = (event, context, callback) => {
                             callback(null, \"Hello World\"\n                           )\n                           ;\n                      }"
                }
            }
        }
    }
}
```
Step 2: Use the AWS CloudFormation template to create a Lambda State Machine

```
{
  "Runtime": "nodejs12.x",
  "Timeout": "25"
}

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

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

"RoleArn": {
  "Fn::GetAtt": [
    "MyLambdaFunction",
    "Arn"
  ]
}
```
2. Open the AWS CloudFormation console and choose Create Stack.
3. On the Select Template page, choose Upload a template to Amazon S3. Choose your MyStateMachine file, and then choose Next.
4. On the Specify Details page, for Stack name, enter MyStateMachine, and then choose Next.
5. On the Options page, choose Next.
6. On the Review page, choose I acknowledge that AWS CloudFormation might create IAM resources. and then choose Create.

AWS CloudFormation begins to create the MyStateMachine stack and displays the CREATE_IN_PROGRESS status. When the process is complete, AWS CloudFormation displays the CREATE_COMPLETE status.

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

---

**Step 3: Start a State Machine execution**

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

**To start the state machine execution**

1. Open the Step Functions console and choose the name of the state machine that you created using AWS CloudFormation.

The New execution page is displayed.

3. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

4. Choose **Start Execution**.

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

5. (Optional) In the **Execution Details**, review the **Execution Status** and the **Started** and **Closed** timestamps.

6. To view the results of your execution, choose **Output**.

---

**Creating a Lambda State Machine for Step Functions Using the AWS CDK**

This tutorial shows you how to create a AWS Step Functions state machine containing an AWS Lambda function using the AWS Cloud Development Kit (CDK). The AWS CDK is an Infrastructure as Code (IAC) framework that lets you define AWS infrastructure using a full-fledged programming language. You can write an app in one of the CDK's supported languages containing one or more stacks. Then, you can synthesize it to an AWS CloudFormation template and deploy it to your AWS account. We'll use this method to define a AWS Step Functions state machine containing a AWS Lambda function, then use the AWS Management Console to initiate execution.

Before you begin this tutorial, you must set up your AWS CDK development environment as described in **Getting Started With the AWS CDK - Prerequisites** Then, install the AWS CDK with the following command at the AWS CLI:

```
npm install -g aws-cdk
```

This tutorial produces the same result as the section called “Creating a Lambda State Machine Using AWS CloudFormation” (p. 217). However, in this tutorial, the AWS CDK doesn't require you to create any IAM roles; the AWS CDK does it for you. The AWS CDK version also includes a Succeed step to illustrate how to add additional steps to your state machine.

**Topics**

- Step 1: Set Up Your AWS CDK Project (p. 225)
- Step 2: Use the AWS CDK to Create a Lambda State Machine (p. 228)
- Step 3: Start a State Machine Execution (p. 233)
- Step 4: Clean Up (p. 233)
- Next steps (p. 234)

**Step 1: Set Up Your AWS CDK Project**

First, create a directory for your new AWS CDK app and initialize the project.

**Note**

Be sure to name the directory `step`. The AWS CDK application template uses the name of the directory to generate names for source files and classes. If you use a different name, your app will not match this tutorial.

**TypeScript**

```
mkdir step
```
### Step 1: Set Up Your AWS CDK Project

#### JavaScript

```bash
cd step
cdk init --language typescript
```

#### Python

```bash
mkdir step
cd step
cdk init --language python
```

After the project has been initialized, activate the project's virtual environment and install the AWS CDK's baseline dependencies.

```bash
source .venv/bin/activate
python -m pip install -r requirements.txt
```

#### Java

```bash
mkdir step
cd step
cdk init --language java
```

#### C#

```bash
mkdir step
cd step
cdk init --language csharp
```

Next, install the construct library modules for AWS Lambda and AWS Step Functions.

#### TypeScript

```bash
npm install @aws-cdk/aws-lambda @aws-cdk/aws-stepfunctions @aws-cdk/aws-stepfunctions-tasks
```

#### JavaScript

```bash
npm install @aws-cdk/aws-lambda @aws-cdk/aws-stepfunctions @aws-cdk/aws-stepfunctions-tasks
```

#### Python

```bash
python -m pip install aws-cdk.aws-lambda aws-cdk.aws-stepfunctions
python -m pip install aws-cdk.aws-stepfunctions-tasks
```

#### Java

Edit the project's `pom.xml` file to add the following dependencies inside the existing `<dependencies>` container.
Maven automatically installs these dependencies the next time you build your app. To build, issue `mvn compile` or use your Java IDE’s **Build** command.

**C#**

```csharp
dotnet add src\Step package Amazon.CDK.AWS.Lambda
dotnet add src\Step package Amazon.CDK.AWS.Stepfunctions
dotnet add src\Step package Amazon.CDK.AWS.Stepfunctions.Tasks
```

You may also install the indicated packages using the Visual Studio NuGet GUI, available via **Tools > NuGet Package Manager > Manage NuGet Packages for Solution**.

Once you’ve installed these modules, you can use them in your AWS CDK app by importing the following packages:

**TypeScript**

```typescript
@aws-cdk/aws-lambda
@aws-cdk/aws-stepfunctions
@aws-cdk/aws-stepfunctions-tasks
```

**JavaScript**

```javascript
@aws-cdk/aws-lambda
@aws-cdk/aws-stepfunctions
@aws-cdk/aws-stepfunctions-tasks
```

**Python**

```python
aws_cdk.aws_lambda
aws_cdk.aws_stepfunctions
aws_cdk.aws_stepfunctions_tasks
```

**Java**

```java
software.amazon.awscdk.services.lambda
software.amazon.awscdk.services.stepfunctions
software.amazon.awscdk.services.stepfunctions.tasks
```

**C#**

```csharp
Amazon.CDK.AWS.Lambda
```
Step 2: Use the AWS CDK to Create a Lambda State Machine

First, we'll present the individual pieces of code that define the Lambda function and the Step Functions state machine. Then, we'll explain how to put them together in your AWS CDK app. Finally, you'll see how to synthesize and deploy these resources.

To create a Lambda function

The following AWS CDK code defines the Lambda function, providing its source code inline.

**TypeScript**

```typescript
const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
  code: lambda.Code.fromInline(`
exports.handler = (event, context, callback) => {
  callback(null, "Hello World!");
};
`),
  runtime: lambda.Runtime.NODEJS_12_X,
  handler: "index.handler",
  timeout: cdk.Duration.seconds(25)
});
```

**JavaScript**

```javascript
const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
  code: lambda.Code.fromInline(`
exports.handler = (event, context, callback) => {
  callback(null, "Hello World!");
};
`),
  runtime: lambda.Runtime.NODEJS_12_X,
  handler: "index.handler",
  timeout: cdk.Duration.seconds(25)
});
```

**Python**

```python
hello_function = lambda_.Function(self, "MyLambdaFunction",
    code=lambda_.Code.from_inline("exports.handler = (event, context, callback) => {
        callback(null, "Hello World!");
    }"),
    runtime=lambda_.Runtime.NODEJS_12_X,
    handler="index.handler",
    timeout=cdk.Duration.seconds(25))
```

**Java**

```java
Function helloFunction = Function.Builder.create(this, "MyLambdaFunction")
  .code(Code.fromInline("exports.handler = (event, context, callback) => { callback(null, "Hello World!");"))
```
Step 2: Use the AWS CDK to Create a Lambda State Machine

You can see in this short example code:

• The function’s logical name, MyLambdaFunction.
• The source code for the function, embedded as a string in the source code of the AWS CDK app.
• Other function attributes, such as the runtime to be used (Node 12.x), the function’s entry point, and a timeout.

To create a Lambda state machine

Our state machine has two states: our Lambda function task, and a success state. The function requires that we create a Step Functions the section called “Task” (p. 29) that invokes our function. This task state is used as the first step in our state machine. The success state is added to the state machine using that task’s next() method. The following code both invokes the function MyLambdaTask, then uses the next() method to set a success state of GreetedWorld.

TypeScript

```typescript
const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
  definition: new tasks.LambdaInvoke(this, "MyLambdaTask", {
    lambdaFunction: helloFunction
  }).next(new sfn.Succeed(this, "GreetedWorld"))
});
```

JavaScript

```javascript
const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
  definition: new tasks.LambdaInvoke(this, "MyLambdaTask", {
    lambdaFunction: helloFunction
  }).next(new sfn.Succeed(this, "GreetedWorld"))
});
```

Python

```python
state_machine = sfn.StateMachine(self, "MyStateMachine",
    definition=tasks.LambdaInvoke(self, "MyLambdaTask",
        lambda_function=hello_function).next(
            sfn.Succeed(self, "GreetedWorld")))
```
To build and deploy the AWS CDK app

In your newly created AWS CDK project, edit the file that contains the stack's definition to look like the following example code. You'll recognize the definitions of the Lambda function and the Step Functions state machine from previous sections.

**TypeScript**

Update `lib/step-stack.ts` with the following code:

```typescript
import * as cdk from '@aws-cdk/core';
import * as lambda from '@aws-cdk/aws-lambda';
import * as sfn from '@aws-cdk/aws-stepfunctions';
import * as tasks from '@aws-cdk/aws-stepfunctions-tasks';

export class StepStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
      code: lambda.Code.fromInline(`
exports.handler = (event, context, callback) => {
  callback(null, "Hello World!");
};
`),
      runtime: lambda.Runtime.NODEJS_12_X,
      handler: "index.handler",
      timeout: cdk.Duration.seconds(3)
    });

    const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
      definition: new tasks.LambdaInvoke(this, 'MyLambdaTask', {
        lambdaFunction: helloFunction
      }).next(new sfn.Succeed(this, "GreetedWorld"))
    });
  }
}
```

**JavaScript**

Update `lib/step-stack.js` with the following code.
const cdk = require('@aws-cdk/core');
const lambda = require('@aws-cdk/aws-lambda');
const sfn = require('@aws-cdk/aws-stepfunctions');
const tasks = require('@aws-cdk/aws-stepfunctions-tasks');

class StepStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
      code: lambda.Code.fromInline(`
        exports.handler = (event, context, callback) => {
            callback(null, "Hello World!");
        };
      `),
      runtime: lambda.Runtime.NODEJS_12_X,
      handler: "index.handler",
      timeout: cdk.Duration.seconds(25)
    });

    const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
      definition: new tasks.LambdaInvoke(this, "MyLambdaTask", {
        lambdaFunction: helloFunction
      }).next(new sfn.Succeed(this, "GreetedWorld"))
    });
  }
}

module.exports = { StepStack }

Python

Update step/step_stack.py with the following code.

```python
from aws_cdk import core as cdk
from aws_cdk import aws_lambda as lambda_
from aws_cdk import aws_stepfunctions as sfn
from aws_cdk import aws_stepfunctions_tasks as tasks

class StepStack(cdk.Stack):
    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        hello_function = lambda_.Function(self, "MyLambdaFunction",
            code=lambda_.Code.from_inline("""""""
                exports.handler = (event, context, callback) => {
                    callback(null, "Hello World!");
                }""""),
            runtime=lambda_.Runtime.NODEJS_12_X,
            handler="index.handler",
            timeout=cdk.Duration.seconds(25))

        state_machine = sfn.StateMachine(self, "MyStateMachine",
            definition=tasks.LambdaInvoke(self, "MyLambdaTask",
                lambda_function=hello_function).next(sfn.Succeed(self, "GreetedWorld")))
```

Java

Update src/main/java/com.myorg/StepStack.java with the following code.

```java
from aws_cdk import core as cdk
from aws_cdk import aws_lambda as lambda_
from aws_cdk import aws_stepfunctions as sfn
from aws_cdk import aws_stepfunctions_tasks as tasks

class StepStack(cdk.Stack):
    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        hello_function = lambda_.Function(self, "MyLambdaFunction",
            code=lambda_.Code.from_inline("""""""
                exports.handler = (event, context, callback) => {
                    callback(null, "Hello World!");
                }""""),
            runtime=lambda_.Runtime.NODEJS_12_X,
            handler="index.handler",
            timeout=cdk.Duration.seconds(25))

        state_machine = sfn.StateMachine(self, "MyStateMachine",
            definition=tasks.LambdaInvoke(self, "MyLambdaTask",
                lambda_function=hello_function).next(sfn.Succeed(self, "GreetedWorld")))
```
package com.myorg;
import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.core.Duration;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.stepfunctions.StateMachine;
import software.amazon.awscdk.services.stepfunctions.Succeed;
import software.amazon.awscdk.services.stepfunctions.tasks.LambdaInvoke;

public class StepStack extends Stack {
    public StepStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public StepStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);

        Function helloFunction = Function.Builder.create(this, "MyLambdaFunction")
            .code(Code.fromInline("exports.handler = (event, context, callback) => {
                callback(null, 'Hello World!');
            }
            "))
            .runtime(Runtime.NODEJS_12_X)
            .handler("index.handler")
            .timeout(Duration.seconds(25))
            .build();

        StateMachine stateMachine = StateMachine.Builder.create(this, "MyStateMachine")
            .definition(LambdaInvoke.Builder.create(this, "MyLambdaTask")
                .lambdaFunction(helloFunction)
                .build())
            .next(new Succeed(this, "GreetedWorld"));
    }
}

C#

Update scr/Step/StepStack.cs with the following code.

using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.StepFunctions;
using Amazon.CDK.AWS.StepFunctions.Tasks;

namespace Step {
    public class StepStack : Stack {
        internal StepStack(Construct scope, string id, IStackProps props = null) :
            base(scope, id, props) {
            var helloFunction = new Function(this, "MyLambdaFunction", new
                FunctionProps {
                    Code = Code.FromInline("exports.handler = (event, context, callback) => {
                        callback(null, 'Hello World!');
                    }
                    "),
                    Runtime = Runtime.NODEJS_12_X,
                
            
        }
    }
}
Save the source file. Then, run the `cdk synth` command in the app's main directory. The AWS CDK runs the app and synthesizes an AWS CloudFormation template from it. The AWS CDK then displays the template.

To actually deploy the Lambda function and the Step Functions state machine to your AWS account, issue `cdk deploy`. You'll be asked to approve the IAM policies the AWS CDK has generated.

### Step 3: Start a State Machine Execution

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

**To start the state machine execution**

1. Open the [Step Functions console](https://console.aws.amazon.com/stepsfunction/home) and choose the name of the state machine that you created using the AWS CDK.
2. On the `MyStateMachine-ABCDEFGHIJKLMNOPQRSTUVWXYZ` page, choose **New execution**.

   The **New execution** page is displayed.

3. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

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

4. Choose **Start Execution**.

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

5. (Optional) In the **Execution Details**, review the **Execution Status** and the **Started** and **Closed** timestamps.

6. To view the results of your execution, choose **Output**.

### Step 4: Clean Up

After you've tested your state machine, we recommend that you remove both your state machine and the related Lambda function to free up resources in your AWS account. Run the `cdk destroy` command in your app's main directory to remove your state machine.
Next steps

To learn more about developing AWS infrastructure using the AWS CDK, see the AWS CDK Developer Guide.

For information about writing AWS CDK apps in your language of choice, see:

TypeScript
   Working with the AWS CDK in TypeScript

JavaScript
   Working with the AWS CDK in JavaScript

Python
   Working with the AWS CDK in Python

Java
   Working with the AWS CDK in Java

C#
   Working with the AWS CDK in C#

For more information on the AWS Construct Library modules used in this tutorial, see the AWS CDK API Reference overviews below.

- aws-lambda
- aws-stepfunctions
- aws-stepfunctions-tasks

Creating an API Gateway REST API with Synchronous Express State Machine Using the AWS CDK

This tutorial shows you how to create an API Gateway REST API with Synchronous Express State Machine as the backend integration using the AWS Cloud Development Kit (CDK). This tutorial will use the StepFunctionsRestApi construct to connect the State Machine to the API Gateway. The StepFunctionsRestApi construct will set up a default input/output mapping and the API Gateway REST API, with required permissions and an HTTP "ANY" method. The AWS CDK is an Infrastructure as Code (IAC) framework that lets you define AWS infrastructure using a full-fledged programming language. You write an app in one of the CDK's supported languages, containing one or more stacks, then synthesize it to an AWS CloudFormation template and deploy it to your AWS account. We'll use it to define an API Gateway REST API, which is integrated with Synchronous Express State Machine as the backend, then use the AWS Management Console to initiate execution.

Before embarking on this tutorial, set up your AWS CDK development environment as described in Getting Started With the AWS CDK - Prerequisites, then install the AWS CDK by issuing:

```bash
npm install -g aws-cdk
```
Step 1: Set Up Your AWS CDK Project

First, create a directory for your new AWS CDK app and initialize the project.

TypeScript

```bash
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language typescript
```

JavaScript

```bash
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language javascript
```

Python

```bash
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language python
```

After the project has been initialized, activate the project’s virtual environment and install the AWS CDK’s baseline dependencies.

```bash
source .venv/bin/activate
cpython -m pip install -r requirements.txt
```

Java

```bash
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language java
```

C#

```bash
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language csharp
```

Go

```bash
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
```
cdk init --language go

**Note**
Be sure to name the directory stepfunctions-rest-api. The AWS CDK application template uses the name of the directory to generate names for source files and classes. If you use a different name, your app will not match this tutorial.

Now install the construct library modules for AWS Step Functions and Amazon API Gateway.

**TypeScript**

```bash	npm install @aws-cdk/aws-stepfunctions @aws-cdk/aws-apigateway
```

**JavaScript**

```bash	npm install @aws-cdk/aws-stepfunctions @aws-cdk/aws-apigateway
```

**Python**

```bash
python -m pip install aws-cdk.aws-stepfunctions
python -m pip install aws-cdk.aws-apigateway
```

**Java**

Edit the project's pom.xml to add the following dependencies inside the existing `<dependencies>` container.

```xml
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>stepfunctions</artifactId>
    <version>${cdk.version}</version>
</dependency>
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>apigateway</artifactId>
    <version>${cdk.version}</version>
</dependency>
```

Maven automatically installs these dependencies the next time you build your app. To build, issue `mvn compile` or use your Java IDE's **Build** command.

**C#**

```bash
dotnet add src\StepfunctionsRestApi package Amazon.CDK.AWS.Stepfunctions
dotnet add src\StepfunctionsRestApi package Amazon.CDK.AWS.APIGateway
```

You may also install the indicated packages using the Visual Studio NuGet GUI, available via **Tools > NuGet Package Manager > Manage NuGet Packages for Solution**.

Once you have installed the modules, you can use them in your AWS CDK app by importing the following packages.

**TypeScript**

```bash
@aws-cdk/aws-stepfunctions
```
Step 2: Use the AWS CDK to create an API Gateway REST API with Synchronous Express State Machine backend integration

First, we'll present the individual pieces of code that define the Synchronous Express State Machine and the API Gateway REST API, then explain how to put them together into your AWS CDK app. Then you'll see how to synthesize and deploy these resources.

**Note**
The State Machine that we will show here will be a simple State Machine with a Pass state.

**To create an Express State Machine**

This is the AWS CDK code that defines a simple state machine with a Pass state.

**TypeScript**

```typescript
const machineDefinition = new sfn.Pass(this, 'PassState', {
  result: {value: "Hello!"},
})
```
Step 2: Use the AWS CDK to create an API Gateway REST API with Synchronous Express State Machine backend integration

```javascript
const stateMachine = new stepfunctions.StateMachine(this, 'MyStateMachine', {
  definition: machineDefinition,
  stateMachineType: stepfunctions.StateMachineType.EXPRESS,
});
```

**JavaScript**

```javascript
const machineDefinition = new sfn.Pass(this, 'PassState', {
  result: {value: "Hello"},
});

const stateMachine = new stepfunctions.StateMachine(this, 'MyStateMachine', {
  definition: machineDefinition,
  stateMachineType: stepfunctions.StateMachineType.EXPRESS,
});
```

**Python**

```python
machineDefinition = sfn.Pass(self, "PassState",
    result = sfn.Result("Hello"))

state_machine = sfn.StateMachine(self, 'MyStateMachine',
    definition = machine_definition,
    state_machine_type = sfn.StateMachineType.EXPRESS)
```

**Java**

```java
Pass machineDefinition = Pass.Builder.create(this, "PassState")
    .result(Result.fromString("Hello"))
    .build();

StateMachine stateMachine = StateMachine.Builder.create(this, "MyStateMachine")
    .definition(machineDefinition)
    .stateMachineType(StateMachineType.EXPRESS)
    .build();
```

**C#**

```csharp
var machineDefinition = new Pass(this, "PassState", new PassProps
{
    Result = Result.FromString("Hello")
});

var stateMachine = new StateMachine(this, "MyStateMachine", new StateMachineProps
{
    Definition = machineDefinition,
    StateMachineType = StateMachineType.EXPRESS
});
```

**Go**

```go
var machineDefinition = awsstepfunctions.NewPass(stack, jsii.String("PassState"),
    &awsstepfunctions.PassProps
    {
        Result: awsstepfunctions.NewResult(jsii.String("Hello")),
    })

var stateMachine = awsstepfunctions.NewStateMachine(stack, jsii.String("StateMachine"),
    &awsstepfunctions.StateMachineProps
    {
```
Definition: machineDefinition,
StateMachineType: awssstepfunctions.StateMachineType_EXPRESS,
}

You can see in this short snippet:

• The machine definition named PassState, which is a Pass State.
• The State Machine's logical name, MyStateMachine.
• The machine definition is used as the State Machine definition.
• The State Machine Type is set as EXPRESS because StepFunctionsRestApi will only allow a Synchronous Express state machine.

To create the API Gateway REST API using StepFunctionsRestApi construct

We will use StepFunctionsRestApi construct to create the API Gateway REST API with required permissions and default input/output mapping.

TypeScript

const api = new apigateway.StepFunctionsRestApi(this, 'StepFunctionsRestApi', { stateMachine: stateMachine });

JavaScript

const api = new apigateway.StepFunctionsRestApi(this, 'StepFunctionsRestApi', { stateMachine: stateMachine });

Python

api = apigw.StepFunctionsRestApi(self, "StepFunctionsRestApi", state_machine = state_machine)

Java

StepFunctionsRestApi api = StepFunctionsRestApi.Builder.create(this, "StepFunctionsRestApi")
    .stateMachine(stateMachine)
    .build();

C#

var api = new StepFunctionsRestApi(this, "StepFunctionsRestApi", new StepFunctionsRestApiProps
{    StateMachine = stateMachine
});

Go

awsapigateway.NewStepFunctionsRestApi(stack, jsii.String("StepFunctionsRestApi"),
  &awsapigateway.StepFunctionsRestApiProps
  {
Step 2: Use the AWS CDK to create an API Gateway REST API with Synchronous Express State Machine backend integration

To build and deploy the AWS CDK app

In the AWS CDK project you created, edit the file containing the definition of the stack to look like the code below. You'll recognize the definitions of the Step Functions state machine and the API Gateway from above.

TypeScript

Update `lib/stepfunctions-rest-api-stack.ts` to read as follows.

```typescript
import * as cdk from '@aws-cdk/core';
import * as stepfunctions from '@aws-stepfunctions';
import * as apigateway from '@aws-cdk/aws-apigateway';

export class StepfunctionsRestApiStack extends cdk.Stack {
    constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);

        const machineDefinition = new stepfunctions.Pass(this, 'PassState', {
            result: {value:"Hello!"},
        })

        const stateMachine = new stepfunctions.StateMachine(this, 'MyStateMachine', {
            definition: machineDefinition,
            stateMachineType: stepfunctions.StateMachineType.EXPRESS,
        });

        const api = new apigateway.StepFunctionsRestApi(this, 'StepFunctionsRestApi', { stateMachine: stateMachine });
    }
}
```

JavaScript

Update `lib/stepfunctions-rest-api-stack.js` to read as follows.

```javascript
const cdk = require('@aws-cdk/core');
const stepfunctions = require('@aws-stepfunctions');
const apigateway = require('@aws-cdk/aws-apigateway');

class StepfunctionsRestApiStack extends cdk.Stack {
    constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);

        const machineDefinition = new stepfunctions.Pass(this, "PassState", {
            result: {value:"Hello!"},
        })

        const stateMachine = new stepfunctions.StateMachine(this, 'MyStateMachine', {
            definition: machineDefinition,
            stateMachineType: stepfunctions.StateMachineType.EXPRESS,
        });

        const api = new apigateway.StepFunctionsRestApi(this, 'StepFunctionsRestApi', { stateMachine: stateMachine });
    }
}
```
Step 2: Use the AWS CDK to create an API Gateway REST API with Synchronous Express State Machine backend integration

Python

Update stepfunctions_rest_api/stepfunctions_rest_api_stack.py to read as follows.

```python
from aws_cdk import core as cdk
from aws_cdk import aws_stepfunctions as sfn
from aws_cdk import aws_apigateway as apigw

class StepfunctionsRestApiStack(cdk.Stack):
    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        machine_definition = sfn.Pass(self, "PassState",
            result = sfn.Result("Hello"))

        state_machine = sfn.StateMachine(self, 'MyStateMachine',
            definition = machine_definition,
            state_machine_type = sfn.StateMachineType.EXPRESS)

        api = apigw.StepFunctionsRestApi(self,
            "StepFunctionsRestApi",
            state_machine = state_machine)
```

Java

Update src/main/java/com.myorg/StepfunctionsRestApiStack.java to read as follows.

```java
package com.myorg;

import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.stepfunctions.Pass;
import software.amazon.awscdk.services.stepfunctions.StateMachine;
import software.amazon.awscdk.services.stepfunctions.StateMachineType;
import software.amazon.awscdk.services.apigateway.StepFunctionsRestApi;

class StepfunctionsRestApiStack extends Stack {
    public StepfunctionsRestApiStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public StepfunctionsRestApiStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);

        Pass machineDefinition = Pass.Builder.create(this, "PassState")
            .result(Result.fromString("Hello"))
            .build();

        StateMachine stateMachine = StateMachine.Builder.create(this, "MyStateMachine")
            .definition(machineDefinition)
            .stateMachineType(StateMachineType.EXPRESS)
            .build();
```
Step 2: Use the AWS CDK to create an API Gateway REST API with Synchronous Express State Machine backend integration

```csharp
using Amazon.CDK;
using Amazon.CDK.AWS.All;
using Amazon.CDK.AWS.APIGateway;

namespace StepfunctionsRestApi
{
    public class StepfunctionsRestApiStack : Stack
    {
        internal StepfunctionsRestApi(Construct scope, string id, IStackProps props = null) : base(scope, id, props)
        {
            var machineDefinition = new Pass(this, "PassState", new PassProps
            {
                Result = Result.FromString("Hello")
            });

            var stateMachine = new StateMachine(this, "MyStateMachine", new StateMachineProps
            {
                Definition = machineDefinition,
                StateMachineType = StateMachineType.EXPRESS
            });

            var api = new StepFunctionsRestApi(this, "StepFunctionsRestApi", new StepFunctionsRestApiProps
            {
                StateMachine = stateMachine
            });
        }
    }
}
```

```go
package main

import (
    "github.com/aws/aws-cdk-go/awscdk"
    "github.com/aws/aws-cdk-go/awscdk/awsapigateway"
    "github.com/aws/aws-cdk-go/awscdk/awsstepfunctions"
    "github.com/aws/constructs-go/constructs/v8"
    "github.com/aws/jsii-runtime-go"
)

type StepfunctionsRestApiGoStackProps struct {
    awscdk.StackProps
}
```
```go
func NewStepfunctionsRestApiGoStack(scope constructs.Construct, id string, props *StepfunctionsRestApiGoStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)
    // The code that defines your stack goes here
    var machineDefinition = awsstepfunctions.NewPass(stack, jsii.String("PassState")),
    &awsstepfunctions.PassProps {
        Result: awsstepfunctions.NewResult(jsii.String("Hello")),
    }
        Definition: machineDefinition,
        StateMachineType: awsstepfunctions.StateMachineType_EXPRESS,
    });
    awsapigateway.NewStepFunctionsRestApi(stack, jsii.String("StepFunctionsRestApi")),
    &awsapigateway.StepFunctionsRestApiProps {
        StateMachine = stateMachine,
    }
    return stack
}
func main() {
    app := awscdk.NewApp(nil)
    NewStepfunctionsRestApiGoStack(app, "StepfunctionsRestApiGoStack", &StepfunctionsRestApiGoStackProps{
        awscdk.StackProps {
            Env: env(),
        },
    })
    app.Synth(nil)
}
```

// env determines the AWS environment (account+region) in which our stack is to
// be deployed. For more information see: https://docs.aws.amazon.com/cdk/latest/guide/
// environments.html
func env() *awscdk.Environment {
    // If unspecified, this stack will be "environment-agnostic".
    // Account/Region-dependent features and context lookups will not work, but a
    // single synthesized template can be deployed anywhere.
    return nil
}

// Uncomment if you know exactly what account and region you want to deploy
// the stack to. This is the recommendation for production stacks.
// return &awscdk.Environment {
//    Account: jsii.String("123456789012"),
//    Region: jsii.String("us-east-1"),
//}

// Uncomment to specialize this stack for the AWS Account and Region that are
// implied by the current CLI configuration. This is recommended for dev
// stacks.
// return &awscdk.Environment{
```
Save the source file, then issue `cdk synth` in the app's main directory. The AWS CDK runs the app and synthesizes an AWS CloudFormation template from it, then displays the template.

To actually deploy the Amazon API Gateway and the AWS Step Functions state machine to your AWS account, issue `cdk deploy`. You'll be asked to approve the IAM policies the AWS CDK has generated. The policies being created will look something like this:

**IAM Statement Changes**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Effect</th>
<th>Action</th>
<th>Principal</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{SfnDemoCdkStack-StateMachine-apiRole.Arni}</code></td>
<td>Allow</td>
<td>sts:AssumeRole</td>
<td>Service:apigateway.amazonaws.com</td>
</tr>
<tr>
<td><code>{StateMachine}</code></td>
<td>Allow</td>
<td>states:StartSyncExecution</td>
<td>AWS: <code>{SfnDemoCdkStack-StateMachineRole}</code></td>
</tr>
<tr>
<td><code>{StepFunctions-rest-api/CloudWatchRole.Arn}</code></td>
<td>Allow</td>
<td>sts:AssumeRole</td>
<td>Service:apigateway.amazonaws.com</td>
</tr>
</tbody>
</table>

**IAM Policy Changes**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Managed Policy ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{StepFunctions-rest-api/CloudWatchRole}</code></td>
<td>arn:${AWS::Partition}::iam:service-role/AmazonAPIGatewayLogs</td>
</tr>
</tbody>
</table>

(Note: There may be security-related changes not in this list. See https://)
To test the deployed API Gateway using API Gateway console

1. Open the Amazon API Gateway console and sign in.
2. Choose your REST API named, StepFunctionsRestApi.
3. In the Resources pane, choose the method you want to test. For the purpose of this tutorial, this will be the ANY method.

4. In the Method Execution pane, in the Client box, choose TEST.
5. Choose POST from the Method Drop-down menu. Type values in the Request Body. The console includes these values in the method request in default application/json form. For the purpose of this tutorial type the following into the Request Body:
6. Choose **Test**. The following information will be displayed:

- **Request** is the resource's path that was called for the method.
- **Status** is the response's HTTP status code.
- **Latency** is the time between the receipt of the request from the caller and the returned response.
- **Response Body** is the HTTP response body.
- **Response Headers** are the HTTP response headers.
- **Logs** are the simulated Amazon CloudWatch Logs entries that would have been written if this method were called outside of the API Gateway console.

**Note**
Although the CloudWatch Logs entries are simulated, the results of the method call are real.

The **Response Body** output should be something like this:

"Hello"

**Tip**
Try the API Gateway with different Methods and an invalid input to see the error output. You may want to change the state machine to look for a particular key and during testing provide the wrong key to fail the State Machine execution and generate an error message in the **Response Body** output.

**To test the deployed API using cURL**

1. Open a terminal window.
2. Copy the following cURL command and paste it into the terminal window, replacing `<api-id>` with your API's ID and `<region>` with the region where your API is deployed.

```bash
curl -X POST
'https://<api-id>.execute-api.<region>.amazonaws.com/prod' 
-d '{"key":"Hello"}' 
-H 'Content-Type: application/json'
```

The **Response Body** output should be something like this:

"Hello"

**Tip**
Try the API Gateway with different Methods and an invalid input to see the error output. You may want to change the state machine to look for a particular key and during testing provide the wrong key to fail the State Machine execution and generate an error message in the **Response Body** output.

**Step 4: Clean Up**

When you're done trying out your API Gateway, you can tear down both the state machine and the API Gateway using the AWS CDK. Issue `cdk destroy` in your app's main directory.
AWS Step Functions Data Science SDK for Python

The AWS Step Functions Data Science SDK is an open source library for data scientists. With this SDK, you can create workflows that process and publish machine learning models using SageMaker and Step Functions. You can also create multi-step machine learning workflows in Python that orchestrate AWS infrastructure at scale, without having to provision and integrate the AWS services separately.

The AWS Step Functions Data Science SDK provides a Python API that can create and invoke Step Functions workflows. You can manage and execute these workflows directly in Python, as well as Jupyter notebooks.

In addition to creating production-ready workflows directly in Python, the AWS Step Functions Data Science SDK allows you to copy that workflow, experiment with new options, and then put the refined workflow into production.

For more information about the AWS Step Functions Data Science SDK, see the following:

- Project on Github
- SDK documentation
- The following Example notebooks, which are available in Jupyter notebook instances in the SageMaker console and the related GitHub project:
  - hello_world_workflow.ipynb
  - machine_learning_workflow_abalone.ipynb
  - training_pipeline_pytorch_mnist.ipynb

Testing Step Functions State Machines Locally

AWS Step Functions Local is a downloadable version of Step Functions that lets you develop and test applications using a version of Step Functions running in your own development environment. The local version of Step Functions can invoke AWS Lambda functions, both in AWS and while running locally. You can also coordinate other supported AWS services (p. 271).

While running Step Functions Local, you can use one of the following ways to invoke service integrations:

- Configuring local endpoints for AWS Lambda and other services. For information about the supported endpoints, see Setting Configuration Options for Step Functions Local (p. 249).
- Making calls directly to an AWS service from Step Functions Local.
- Mocking the response from service integrations. For information about using mocked service integrations, see Using Mocked Service Integrations (p. 255).

AWS Step Functions Local is available as a JAR package or a self-contained Docker image that runs on Microsoft Windows, Linux, macOS, and other platforms that support Java or Docker.

**Warning**

The downloadable version of AWS Step Functions is intended to be used only for testing and should never be used to process sensitive information.

The following topics describe how you can set up Step Functions Local using Docker and JAR file, and run Step Functions Local to work with AWS Lambda, AWS Serverless Application Model(AWS SAM) CLI Local, or other supported services.

**Topics**

- Setting Up Step Functions Local (Downloadable Version) and Docker (p. 248)
The Step Functions Local Docker image enables you to get started with Step Functions Local quickly by using a Docker image with all the needed dependencies. The Docker image enables you to include Step Functions Local in your containerized builds and as part of your continuous integration testing.

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

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

To start the downloadable version of Step Functions on Docker, run the following Docker run command.

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

To interact with AWS Lambda or other supported services, you need to configure your credentials and other configuration options first. For more information, see the following topics:

- Setting Configuration Options for Step Functions Local (p. 249)
- Credentials and configuration for Docker (p. 250)

## Setting Up Step Functions Local (Downloadable Version) - Java Version

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

### To set up and run Step Functions on your computer

1. Download Step Functions using the following links.

<table>
<thead>
<tr>
<th>Download Links</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>.tar.gz</td>
<td>.tar.gz.md5</td>
</tr>
<tr>
<td>.zip</td>
<td>.zip.md5</td>
</tr>
</tbody>
</table>

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

```
$ java -jar StepFunctionsLocal.jar -v
```
4. (Optional) View a listing of available commands.

    $ java -jar StepFunctionsLocal.jar -h

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

    java -jar StepFunctionsLocal.jar

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

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

**Note**
By default, Step Functions Local uses a local test account and credentials, and the AWS Region is set to US East (N. Virginia). To use Step Functions Local with AWS Lambda, or other supported services, you must configure your credentials and Region.
If you use Express workflows with Step Functions Local, the execution history will be stored in a log file. It is not logged to CloudWatch Logs. The log file path will be based on the CloudWatch Logs log group ARN provided when you create the local state machine. The log file will be stored in /aws/states/log-group-name/$#(execution_arn).log relative to the location where you are running Step Functions Local. For example, if the execution ARN is:

    arn:aws:states:us-east-1:123456789012:express:test:example-ExpressLogGroup-wJalrXUtnFEMI

the log file will be:


### Setting Configuration Options for Step Functions Local

When you start AWS Step Functions Local by using the JAR file, you can set configuration options by using the AWS Command Line Interface (AWS CLI), or by including them in the system environment. For Docker, you must specify these options in a file that you reference when starting Step Functions Local.

#### Configuration Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Command Line</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>-account, --aws-account</td>
<td>AWS_ACCOUNT_ID</td>
</tr>
<tr>
<td>Region</td>
<td>-region, --aws-region</td>
<td>AWS_DEFAULT_REGION</td>
</tr>
<tr>
<td>Wait Time Scale</td>
<td>-waitTimeScale, --wait-time-scale</td>
<td>WAIT_TIME_SCALE</td>
</tr>
<tr>
<td>Option</td>
<td>Command Line</td>
<td>Environment</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Lambda Endpoint</td>
<td>-lambdaEndpoint, --lambda-endpoint</td>
<td>LAMBDA_ENDPOINT</td>
</tr>
<tr>
<td>Batch Endpoint</td>
<td>-batchEndpoint, --batch-endpoint</td>
<td>BATCH_ENDPOINT</td>
</tr>
<tr>
<td>DynamoDB Endpoint</td>
<td>-dynamodbEndpoint, --dynamodb-endpoint</td>
<td>DYNAMODB_ENDPOINT</td>
</tr>
<tr>
<td>ECS Endpoint</td>
<td>-ecsEndpoint, --ecs-endpoint</td>
<td>ECS_ENDPOINT</td>
</tr>
<tr>
<td>Glue Endpoint</td>
<td>-glueEndpoint, --glue-endpoint</td>
<td>GLUE_ENDPOINT</td>
</tr>
<tr>
<td>SageMaker Endpoint</td>
<td>-sagemakerEndpoint, --sagemaker-endpoint</td>
<td>SAGE_MAKER_ENDPOINT</td>
</tr>
<tr>
<td>SQS Endpoint</td>
<td>-sqsEndpoint, --sqs-endpoint</td>
<td>SQS_ENDPOINT</td>
</tr>
<tr>
<td>SNS Endpoint</td>
<td>-snsEndpoint, --sns-endpoint</td>
<td>SNS_ENDPOINT</td>
</tr>
<tr>
<td>Step Functions Endpoint</td>
<td>-stepFunctionsEndpoint, --step-functions-endpoint</td>
<td>STEP_FUNCTIONS_ENDPOINT</td>
</tr>
</tbody>
</table>

### Credentials and configuration for Docker

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

This file contains your credentials and other configuration options. The following can be used as a template when creating the `aws-stepfunctions-local-credentials.txt` file.

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

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

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

**Note**

It is recommended to use the special DNS name `host.docker.internal`, which resolves to the internal IP address that the host uses, such as `http://host.docker.internal:8000`. For more information, see Docker documentation for Mac and Windows at Networking features in Docker Desktop for Mac and Networking features in Docker Desktop for Windows respectively.
Running Step Functions Local on Your Computer

Use the local version of Step Functions to configure, develop and test state machines on your computer.

Run a HelloWorld state machine locally

After you run Step Functions locally with the AWS Command Line Interface (AWS CLI), you can start a state machine execution.

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

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

**Note**
The role-arn is not used for Step Functions Local, but you must include it with the proper syntax. You can use the Amazon Resource Name (ARN) from the previous example.

If you successfully create the state machine, Step Functions responds with the creation date and the state machine ARN.

```
{
  "creationDate": 1548454198.202,
}
```

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

```
```

Step Functions Local with AWS SAM CLI Local

You can use the local version of Step Functions with a local version of AWS Lambda. To configure this, you must install and configure AWS SAM.

For information about configuring and running AWS SAM, see the following:

- Set Up AWS SAM
- Start AWS SAM CLI Local

When Lambda is running on your local system, you can start Step Functions Local. From the directory where you extracted your Step Functions local JAR files, start Step Functions Local and use the --lambda-endpoint parameter to configure the local Lambda endpoint.

```
java -jar StepFunctionsLocal.jar --lambda-endpoint http://127.0.0.1:3001 command
```
For more information about running Step Functions Local with AWS Lambda, see Testing Step Functions and AWS SAM CLI Local (p. 252).

Testing Step Functions and AWS SAM CLI Local

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

For more information, see the following topics:

- Testing Step Functions State Machines Locally (p. 247)
- Set Up AWS SAM

Topics

- Step 1: Set Up AWS SAM (p. 252)
- Step 2: Test AWS SAM CLI Local (p. 252)
- Step 3: Start AWS SAM CLI Local (p. 253)
- Step 4: Start Step Functions Local (p. 253)
- Step 5: Create a State Machine That References Your AWS SAM CLI Local Function (p. 254)
- Step 6: Start an Execution of Your Local State Machine (p. 254)

Step 1: Set Up AWS SAM

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

1. Install the AWS SAM CLI.

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

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

   1. Initialize the Application
   2. Test the Application Locally

   This creates a `sam-app` directory, and builds an environment that includes a Python-based Hello World Lambda function.

Step 2: Test AWS SAM CLI Local

Now that you have installed AWS SAM and created the Hello World Lambda function, you can test the function. In the `sam-app` directory, enter the following command:

```bash
sam local start-api
```

This launches a local instance of your Lambda function. You should see output similar to the following:

```
```
Open a browser and enter the following:

http://127.0.0.1:3000/hello

This will output a response similar to the following:

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

Enter CTRL+C to end the Lambda API.

**Step 3: Start AWS SAM CLI Local**

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

```
sam local start-lambda
```

This starts AWS SAM CLI Local and provides the endpoint to use, similar to the following output:

```
2019-01-29 15:33:32 Starting the Local Lambda Service. You can now invoke your Lambda Functions defined in your template through the endpoint.
```

**Step 4: Start Step Functions Local**

**JAR File**

If you're using the `.jar` file version of Step Functions Local, start Step Functions and specify the Lambda endpoint. In the directory where you extracted the `.jar` files, enter the following command:

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

When Step Functions Local starts, it checks the environment, and then the credentials configured in your `~/.aws/credentials` file. By default, it starts using a fictitious user ID, and is listed as region `us-east-1`.

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

**Docker**

If you're using the Docker version of Step Functions Local, launch Step Functions with the following command:

```
```

253
docker run -p 8083:8083 amazon/aws-stepfunctions-local

For information about installing the Docker version of Step Functions, see Setting Up Step Functions Local (Downloadable Version) and Docker (p. 248).

**Note**

You can specify the endpoint through the command line or by setting environment variables if you launch Step Functions from the .jar file. For the Docker version, you must specify the endpoints and credentials in a text file. See Setting Configuration Options for Step Functions Local (p. 249).

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

Once Step Functions Local is running, create a state machine that references the `HelloWorldFunction` that you initialized in Step 1: Set Up AWS SAM (p. 252).

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

This will create a state machine and provide an Amazon Resource Name (ARN) that you can use to start an execution.

```json
{
  "creationDate": 1548805711.403,
}
```

**Step 6: Start an Execution of Your Local State Machine**

Once you have created a state machine, start an execution. You'll need to reference the endpoint and state machine ARN when using the following `aws stepfunctions` command:

```bash
```

This starts an execution named `test` of your `HelloWorld` state machine.

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

Now that Step Functions is running locally, you can interact with it using the AWS CLI. For example, to get information about this execution, use the following command:
Calling `describe-execution` for an execution provides more complete details, similar to the following output:

```
{
    "status": "SUCCEEDED",
    "startDate": 1549056334.073,
    "name": "test",
    "stopDate": 1549056351.276,
    "output": "{"statusCode": 200, "body": "{\"message\": \"hello world\", \"location\": \"72.21.198.64\"}"",
    "input": "{}"
}
```

Using Mocked Service Integrations

In Step Functions Local, you can test the execution paths of your state machines without actually calling integrated services by using mocked service integrations. To configure your state machines to use mocked service integrations, you create a mock configuration file. In this file, you define the desired output of your service integrations as mocked responses and the executions which use your mocked responses to simulate an execution path as test cases.

By providing the mock configuration file to Step Functions Local, you can test service integration calls by running state machines that use the mocked responses specified in the test cases instead of making actual service integration calls.

**Note**

If you don’t specify mocked service integration responses in the mock configuration file, Step Functions Local will invoke the AWS service integration using the endpoint you configured while setting up Step Functions Local. For information about configuring endpoints for Step Functions Local, see Setting Configuration Options for Step Functions Local (p. 249).

**Topics**

- Key concepts in this topic (p. 255)
- Step 1: Specify Mocked Service Integrations in a Mock Configuration File (p. 256)
- Step 2: Provide the Mock Configuration File to Step Functions Local (p. 260)
- Step 3: Run Mocked Service Integration Tests (p. 261)
- Configuration File for Mocked Service Integrations (p. 263)

**Key concepts in this topic**

This topic uses several concepts which are defined in the following list:

- Mocked Service Integrations - Refers to Task states configured to use mocked responses instead of performing actual service calls.
- Mocked Responses - Refers to mock data that Task states can be configured to use.
- Test Cases - Refers to state machine executions configured to use mocked service integrations.
- Mock Configuration File - Refers to mock configuration file that contains JSON, which defines mocked service integrations, mocked responses, and test cases.
Step 1: Specify Mocked Service Integrations in a Mock Configuration File

You can test Step Functions AWS SDK and optimized service integrations using Step Functions Local. The following image shows the state machine defined in the State machine definition tab:

To do this, you must create a mock configuration file containing sections as defined in Introducing structure of mock configuration (p. 263).

1. Create a file named `MockConfigFile.json` to configure tests with mocked service integrations.

The following example shows a mock configuration file referencing a state machine with two defined states named `LambdaState` and `SQSState`.

Mock configuration file example

The following is an example of a mock configuration file which demonstrates how to mock responses from invoking a Lambda function (p. 301) and sending a message to Amazon SQS (p. 311). In this example, the `LambdaSQSIntegration` (p. 264) state machine contains three test cases named HappyPath, RetryPath, and HybridPath which mock the Task states named LambdaState and SQSState. These states use the `MockedLambdaSuccess`, `MockedSQSSuccess`, and `MockedLambdaRetry` mocked service responses. These mocked service responses are defined in the `MockedResponses` section of the file.

```json
{
  "StateMachines":{
    "LambdaSQSIntegration":{
      "TestCases":{
        "HappyPath":{
          "LambdaState":"MockedLambdaSuccess",
          "SQSState":"MockedSQSSuccess"
        },
        "RetryPath":{
          "LambdaState":"MockedLambdaRetry",
          "SQSState":"MockedSQSSuccess"
        },
        "HybridPath":{
          "LambdaState":"MockedLambdaSuccess"
        }
      }
    }
  },
  "MockedResponses":{
    "MockedLambdaSuccess":{
      "0":{}
    }
  }
}
```
The following is an example of a state machine definition called LambdaSQSIntegration, which defines two service integration task states named LambdaState and SQSState. LambdaState contains a retry policy based on States.ALL.
You can run the LambdaSQSIntegration state machine definition referenced in the mock configuration file using one of the following test cases:

- **HappyPath** - This test mocks the output of LambdaState and SQSState using MockedLambdaSuccess and MockedSQSSuccess respectively.
  - The LambdaState will return the following value:

    ```json
    "0":{
      "Return":{
        "StatusCode":200,
        "Payload":{
          "StatusCode":200,
          "body":"Hello from Lambda!"
        }
      }
    }
    ```

  - The SQSState will return the following value:

    ```json
    "0":{
      "Return":{
        "MD5OfMessageBody":"3bc6e8e-7b85-4375-b0bc-1a59812c6e51",
        "MessageId":"3bc6e8e-8b51-4375-b0bc-1a59812c6e51"
      }
    }
    ```

- **RetryPath** - This test mocks the output of LambdaState and SQSState using MockedLambdaRetry and MockedSQSSuccess respectively. In addition, LambdaState is configured to perform four retry attempts. The mocked responses for these attempts are defined and indexed in the MockedLambdaRetry state.
  - The initial attempt ends with a task failure containing a cause and error message as shown in the following example:
AWS Step Functions Developer Guide  
Using Mocked Service Integrations

```json
"0":{
  "Throw": {
    "Error": "Lambda.ResourceNotReadyException",
    "Cause": "Lambda resource is not ready."
  }
}
```

- The first and second retry attempts end with a task failure containing a cause and error message as shown in the following example:

```json
"1-2":{
  "Throw": {
    "Error": "Lambda.TimeoutException",
    "Cause": "Lambda timed out."
  }
}
```

- The third retry attempt ends with a task success containing state result from Payload section in the mocked Lambda response.

```json
"3":{
  "Return": {
    "StatusCode": 200,
    "Payload": {
      "StatusCode": 200,
      "body": "Hello from Lambda!"
    }
  }
}
```

**Note**
- For states with a retry policy, Step Functions Local will exhaust the retry attempts set in the policy until it receives a success response. This means that you must denote mocks for retries with consecutive attempt numbers and should cover all the retry attempts before returning a success response.
- If you do not specify a mocked response for a specific retry attempt, for example, retry "3", the state machine execution will fail.
- HybridPath - This test mocks the output of LambdaState. After LambdaState runs successfully and receives mocked data as a response, SQSState performs an actual service call to the resource specified in production.

For information about how to start test executions with mocked service integrations, see Step 3: Run Mocked Service Integration Tests (p. 261).

2. Make sure that the mocked responses' structure conforms to the structure of actual service responses you receive when you make integrated service calls. For information about the structural requirements for mocked responses, see Configuring mocked service integrations (p. 266).

In the previous example mock configuration file, the mocked responses defined in MockedLambdaSuccess and MockedLambdaRetry conform to the structure of actual responses that are returned from calling HelloFromLambda.

**Important**
AWS service responses can vary in structure between different services. Step Functions Local doesn't validate if mocked response structures conform to actual service response structures. You must ensure that your mocked responses conform to actual responses before testing. To review the structure of service responses, you can either perform the actual service calls using Step Functions or view the documentation for those services.
Step 2: Provide the Mock Configuration File to Step Functions Local

You can provide the mock configuration file to Step Functions Local in one of the following ways:

**Docker**

**Note**

If you're using the Docker version of Step Functions Local, you can provide the mock configuration file using an environment variable only. In addition, you must mount the mock configuration file onto the Step Functions Local container at the initial server boot-up.

Mount the mock configuration file onto any directory within the Step Functions Local container. Then, set an environment variable named `SFN_MOCK_CONFIG` that contains the path to the mock configuration file in the container. This method enables the mock configuration file to be named anything as long as the environment variable contains the file path and name.

The following command shows the format to start the Docker image.

```bash
docker run -p 8083:8083
--mount type=bind,readonly,source={absolute path to mock config file},destination=/home/StepFunctionsLocal/MockConfigFile.json
-e SFN_MOCK_CONFIG="/home/StepFunctionsLocal/MockConfigFile.json" amazon/aws-stepfunctions-local
```

The following example uses the command to start the Docker image.

```bash
docker run -p 8083:8083
--mount type=bind,readonly,source=/Users/admin/Desktop/workplace/MockConfigFile.json,destination=/home/StepFunctionsLocal/MockConfigFile.json
-e SFN_MOCK_CONFIG="/home/StepFunctionsLocal/MockConfigFile.json" amazon/aws-stepfunctions-local
```

**JAR File**

Use one of the following ways to provide the mock configuration file to Step Functions Local:

- Place the mock configuration file in the same directory as `StepFunctionsLocal.jar`. When using this method, you must name the mock configuration file `MockConfigFile.json`.

- In the session running Step Functions Local, set an environment variable named `SFN_MOCK_CONFIG` to the full path of the mock configuration file. This method enables the mock configuration file to be named anything as long as the environment variable contains its file path and name. In the following example, the `SFN_MOCK_CONFIG` variable is set to point at a mock configuration file named `EnvSpecifiedMockConfig.json`, located in the `/home/workspace` directory.

```bash
export SFN_MOCK_CONFIG="/home/workspace/EnvSpecifiedMockConfig.json"
```

**Note**

- If you do not provide the environment variable `SFN_MOCK_CONFIG` to Step Functions Local, by default, it will attempt to read a mock configuration file named `MockConfigFile.json` in the directory from which you launched Step Functions Local.

- If you place the mock configuration file in the same directory as `StepFunctionsLocal.jar` and set the environment variable `SFN_MOCK_CONFIG`, Step Functions Local will read the file specified by the environment variable.
**Step 3: Run Mocked Service Integration Tests**

After you create and provide a mock configuration file to Step Functions Local, run the state machine configured in the mock configuration file using mocked service integrations. Then check the execution results using an API action.

1. Create a state machine based on the previously mentioned definition in the mock configuration file (p. 256).

   ```bash
   aws stepfunctions create-state-machine \
   ```

2. Run the state machine using mocked service integrations.

   To use the mock configuration file, make a `StartExecution` API call on a state machine configured in the mock configuration file. To do this, append the suffix, `#test_name`, to the state machine ARN used by `StartExecution`. `test_name` is a test case, which is configured for the state machine in the same mock configuration file.

   The following command is an example that uses the `LambdaSQSIntegration` state machine and mock configuration. In this example, the `LambdaSQSIntegration` state machine is executed using the `HappyPath` test defined in Step 1: Specify Mocked Service Integrations in a Mock Configuration File (p. 256). The `HappyPath` test contains the configuration for the execution to handle mock service integration calls that `LambdaState` and `SQSState` states make using the `MockedLambdaSuccess` and `MockedSQSSuccess` mocked service responses.

   ```bash
   aws stepfunctions start-execution \
   --endpoint http://localhost:8083 \n   --name executionWithHappyPathMockedServices \n   --state-machine arn:aws:states:us-east-1:123456789012:stateMachine:LambdaSQSIntegration#HappyPath
   ```

3. View the state machine execution response.

   The response to calling `StartExecution` using a mocked service integration test is same as the response to calling `StartExecution` normally, which returns the execution ARN and start date.

   The following is an example response to calling `StartExecution` using the mocked service integration test:

   ```json
   {
   "startDate":"2022-01-28T15:03:16.981000-05:00",
   "executionArn": "arn:aws:states:us-east-1:123456789012:execution:LambdaSQSIntegration:executionWithHappyPathMockedServices"
   }
   ```

4. Check the execution's results by making a `ListExecutions`, `DescribeExecution`, or `GetExecutionHistory` API call.
The following example demonstrates parts of a response to calling GetExecutionHistory using the execution ARN from the example response shown in step 2. In this example, the output of LambdaState and SQSState is the mock data defined in MockedLambdaSuccess and MockedSQSSuccess in the mock configuration file (p. 256). In addition, the mocked data is used the same way that data returned by performing actual service integration calls would be used. Also, in this example, the output from LambdaState is passed onto SQSState as input.

```json
{
  "events": [ 
    ..., 
    {
      "timestamp": "2021-12-02T19:48:00.000+00:00",
      "type": "TaskStateEntered",
      "id": 2,
      "previousEventId": 0,
      "stateEnteredEventDetails": {
        "name": "LambdaState",
        "input": "{}",
        "inputDetails": {
          "truncated": false
        }
      }
    }, ...
    ..., 
    {
      "timestamp": "2021-11-25T23:39:10.587000+00:00",
      "type": "LambdaFunctionSucceeded",
      "id": 5,
      "previousEventId": 4,
      "lambdaFunctionSucceededEventDetails": {
        "output": "{"statusCode":200,"body":"Hello from Lambda\n\n\n"}",
        "outputDetails": {
          "truncated": false
        }
      }
    }, ...
    ..., 
    {
      "timestamp": "2021-12-02T19:49.464000+00:00",
      "type": "TaskStateEntered",
      "id": 7,
      "previousEventId": 6,
      "stateEnteredEventDetails": {
        "name": "SQSState",
        "input": "{"statusCode":200,"body":"Hello from Lambda\n\n\n"}",
        "inputDetails": {
          "truncated": false
        }
      }
    }, ...
    ...
    {
      "timestamp": "2021-11-25T23:39:10.652000+00:00",
      "type": "TaskSucceeded",
      "id": 10,
      "previousEventId": 9,
    }
  ]
}```
"taskSucceededEventDetails": {  
  "resourceType": "sqs",  
  "resource": "sendMessage",  
  "output": "{"MD5OfMessageBody":"3bcb6e8e-7h85-4375-b0bc-1a59812c6e51","MessageId":"3bcb6e8e-8b51-4375-b0bc-1a59812c6e51""},  
  "outputDetails": {    
    "truncated": false  
  }  
},
...
]

Configuration File for Mocked Service Integrations

To use mocked service integrations, you must first create a mock configuration file named `MockConfigFile.json` containing your mock configurations. Then provide Step Functions Local with the mock configuration file. This configuration file defines test cases, which contain mock states that use mocked service integration responses. The following section contains information about the structure of mock configuration that includes the mock states and mocked responses:

**Topics**
- Introducing structure of mock configuration (p. 263)
- Configuring mocked service integrations (p. 266)

Introducing structure of mock configuration

A mock configuration is a JSON object containing the following top-level fields:

- **StateMachines** - The fields of this object represent state machines configured to use mocked service integrations.
- **MockedResponse** - The fields of this object represent mocked responses for service integration calls.

The following is an example of a mock configuration file which includes a `StateMachine` definition and `MockedResponse`.

```json
{
  "StateMachines":{
    "LambdaSQSIntegration":{
      "TestCases":{
        "HappyPath":{
          "LambdaState":"MockedLambdaSuccess",
          "SQSState":"MockedSQSSuccess"
        },
        "RetryPath":{
          "LambdaState":"MockedLambdaRetry",
          "SQSState":"MockedSQSSuccess"
        },
        "HybridPath":{
          "LambdaState":"MockedLambdaSuccess"
        }
      }
    }
  },
  "MockedResponses":{
    "MockedLambdaSuccess":{
      "0":{
        ...
      }
    }
  }
}
```
Mock configuration field reference

The following sections explain the top-level object fields that you must define in your mock configuration.

- StateMachines (p. 264)
- MockedResponses (p. 265)

StateMachines

The StateMachines object defines which state machines will use mocked service integrations. The configuration for each state machine is represented as a top-level field of StateMachines. The
field name is the name of the state machine and value is an object containing a single field named TestCases, whose fields represent test cases of that state machine.

The following syntax shows a state machine with two test cases:

```json
"MyStateMachine": {
  "TestCases": {
    "HappyPath": {
      ... },
    "SadPath": {
      ...
    }
  }
}
```

**TestCases**

The fields of TestCases represent individual test cases for the state machine. The name of each test case must be unique per state machine and the value of each test case is an object specifying a mocked response to use for Task states in the state machine.

The following example of a TestCase links two Task states to two MockedResponses:

```json
"HappyPath": {
  "SomeTaskState": "SomeMockedResponse",
  "AnotherTaskState": "AnotherMockedResponse"
}
```

**MockedResponses**

MockedResponses is an object containing multiple mocked response objects with unique field names. A mocked response object defines the successful result or error output for each invocation of a mocked Task state. You specify the invocation number using individual integer strings, such as "0", "1", "2", and "3" or an inclusive range of integers, such as "0-1", "2-3".

When you mock a Task, you must specify a mocked response for every invocation. A response must contain a single field named Return or Throw whose value is the result or error output for the mocked Task invocation. If you do not specify a mocked response, the state machine execution will fail.

The following is an example of a MockedResponse with Throw and Return objects. In this example, the first three times the state machine is run, the response specified in "0-2" is returned, and the fourth time the state machine runs, the response specified in "3" is returned.

```json
"SomeMockedResponse": {
  "0-2": {
    "Throw": {
      ...
    },
  },
  "3": {
    "Return": {
      ...
    }
  }
}
```

**Note**

If you are using a Map state, and want to ensure predictable responses for the Map state, set the value of maxConcurrency to 1. If you set a value greater than 1, Step Functions Local will run
multiple iterations concurrently, which will cause the overall execution order of states across iterations to be unpredictable. This may further cause Step Functions Local to use different mocked responses for iteration states from one execution to the next.

**Return**

Return is represented as a field of the `MockedResponse` objects. It specifies the successful result of a mocked Task state.

The following is an example of a `Return` object that contains a mocked response for calling `Invoke` on a Lambda function:

```json
"Return": {
    "StatusCode": 200,
    "Payload": {
        "StatusCode": 200,
        "body": "Hello from Lambda!"
    }
}
```

**Throw**

Throw is represented as a field of the `MockedResponse` objects. It specifies the error output (p. 88) of a failed Task. The value of Throw must be an object containing an `Error` and `Cause` fields with string values. In addition, the string value you specify in `Error` field in the `MockConfigFile.json` must match the errors handled in the `Retry` and `Catch` sections of your state machine.

The following is an example of a `Throw` object that contains a mocked response for calling `Invoke` on a Lambda function:

```json
"Throw": {
    "Error": "Lambda.TimeoutException",
    "Cause": "Lambda timed out."
}
```

**Configuring mocked service integrations**

You can mock any service integration using Step Functions Local. However, Step Functions Local doesn’t enforce the mocks to be the same as the real APIs. A mocked Task will never call the service endpoint. If you do not specify a mocked response, a Task will attempt to call the service endpoints. In addition, Step Functions Local will automatically generate a task token when you mock a Task using the `.waitForTaskToken`. 
Use timeouts to avoid stuck executions

By default, the Amazon States Language doesn't specify timeouts for 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 the TimeoutSeconds field isn't specified for an Activity or Task state, an execution is stuck waiting for a response that will never come.

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

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

For more information, see Task (p. 29) in the Amazon States Language documentation.

**Note**
You can set a timeout for your state machine using the TimeoutSeconds field in your Amazon States Language definition. For more information, see State Machine Structure (p. 24).

Use Amazon S3 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 262,144 bytes, use Amazon Simple Storage Service (Amazon S3) to store the data, and parse the Amazon Resource Name (ARN) of the bucket in the Payload parameter to get the bucket name and key value. Alternatively, adjust your implementation so that you pass smaller payloads in your executions.

For example, a state machine can pass input to an AWS Lambda function, like in the following example.
Avoid reaching the history quota

AWS Step Functions has a hard quota of 25,000 entries in the execution event history. To avoid reaching this quota 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 Continuing as a New Execution (p. 172) tutorial.

Handle Lambda service exceptions

AWS Lambda can occasionally experience transient service errors. In this case, invoking Lambda results in a 500 error, such as ServiceException, AWSLambdaException, or SdkClientException. As a best practice, proactively handle these exceptions in your state machine to Retry invoking your Lambda function, or to Catch the error.

Lambda errors are reported as Lambda.ErrorName. To retry a Lambda service exception error, you could use the following Retry code.

"Retry": [ 
  "ErrorEquals": [ "Lambda.ServiceException", "Lambda.AWSLambdaException", "Lambda.SdkClientException"],
]
Avoid latency when polling for activity tasks

The `GetActivityTask` API is designed to provide a `taskToken` exactly once. If a `taskToken` is dropped while communicating with an activity worker, a number of `GetActivityTask` requests can be blocked for 60 seconds waiting for a response until `GetActivityTask` times out.

If you only have a small number of polls waiting for a response, it's possible that all requests will queue up behind the blocked request and stop. However, if you have a large number of outstanding polls for each activity Amazon Resource Name (ARN), and some percentage of your requests are stuck waiting, there will be many more that can still get a `taskToken` and begin to process work.

For production systems, we recommend at least 100 open polls per activity ARN's at each point in time. If one poll gets blocked, and a portion of those polls queue up behind it, there are still many more requests that will receive a `taskToken` to process work while the `GetActivityTask` request is blocked.

To avoid these kinds of latency problems when polling for tasks:

- Implement your pollers as separate threads from the work in your activity worker implementation.
- Have at least 100 open polls per activity ARN at each point in time.

**Note**
Scaling to 100 open polls per ARN can be expensive. For example, 100 Lambda functions polling per ARN is 100 times more expensive than having a single Lambda function with 100 polling threads. To both reduce latency and minimize cost, use a language that has asynchronous I/O, and implement multiple polling threads per worker. For an example activity worker where the poller threads are separate from the work threads, see Example Activity Worker in Ruby (p. 34).

For more information on activities and activity workers see Activities (p. 33).

Choosing Standard or Express Workflows

AWS Step Functions offers Standard Workflows as the default workflow type, with the option to choose Express Workflows.
You can choose Standard Workflows when you need long-running, durable, and auditable workflows, or Express Workflows for high-volume, event processing workloads. Your state machine executions will behave differently, depending on which Type you select. The Type you choose cannot be changed after your state machine has been created.

For detailed information on the differences between Standard and Express Workflows, see Standard vs. Express Workflows (p. 19).

Amazon CloudWatch Logs resource policy size restrictions

CloudWatch Logs resource policies are limited to 5120 characters. When CloudWatch Logs detects that a policy approaches this size limit, it automatically enables log groups that start with `/aws/vendedlogs/`.

When you create a state machine with logging enabled, Step Functions must update your CloudWatch Logs resource policy with the log group you specify. To avoid reaching the CloudWatch Logs resource policy size limit, prefix your CloudWatch Logs log group names with `/aws/vendedlogs/`. When you create a log group in the Step Functions console, the log group names are prefixed with `/aws/vendedlogs/states`. For more information, see Enabling Logging from Certain AWS Services.
Using AWS Step Functions with other services

Learn about coordinating other AWS services with AWS Step Functions.

Topics
- Call other AWS services (p. 271)
- AWS SDK service integrations (p. 274)
- Service Integration Patterns (p. 291)
- Pass Parameters to a Service API (p. 296)
- Code Snippets (p. 298)
- Optimized integrations for Step Functions (p. 299)

Call other AWS services

AWS Step Functions integrates with AWS services, letting you call each service’s API actions from your workflow. You can use Step Functions’ AWS SDK integrations (p. 274) to call any of the over two hundred AWS services directly from your state machine, giving you access to over nine thousand API actions. Or you can use Step Functions’ Optimized integrations (p. 299), each of which has been customized to provide special functionality for your workflow. Some API actions are available in both types of integration. In this case, it’s recommended that you use the Optimized integration.

You coordinate these services directly from a Task state in the Amazon States Language. For example, using Step Functions, you can call other services to:
- Invoke an AWS Lambda function.
- Run an AWS Batch job and then perform different actions based on the results.
- Insert or get an item from Amazon DynamoDB.
- Run an Amazon Elastic Container Service (Amazon ECS) task and wait for it to complete.
- Publish to a topic in Amazon Simple Notification Service (Amazon SNS).
- Send a message in Amazon Simple Queue Service (Amazon SQS).
- Manage a job for AWS Glue or Amazon SageMaker.
- Build workflows for executing Amazon EMR jobs.
- Launch an AWS Step Functions workflow execution.

Optimized integrations

Optimized integrations have been customized by Step Functions to provide special functionality for a workflow context. For example, Lambda Invoke (p. 301) converts its API output from an escaped JSON to a JSON object. AWS BatchSubmitJob (p. 303) lets you pause execution until the job is complete. The first set of optimized integrations was released in 2018, and there are now over fifty APIs.

AWS SDK integrations

AWS SDK integrations work exactly like a standard API call using the AWS SDK. They provide the ability to call over nine thousand APIs across the more than two hundred AWS services directly from your state machine definition. AWS SDK integrations were released in 2021.
# Integration pattern support

Standard Workflows and Express Workflows support the same integrations but do not support the same integration patterns. Express Workflows do not support Run a Job (.sync) or Wait for Callback (.waitForTaskToken). Optimized integrations pattern support is different for each integration. For more information, see Standard vs. Express Workflows (p. 19).

## Standard Workflows

### Supported service integrations

<table>
<thead>
<tr>
<th>Optimized integrations</th>
<th>Service</th>
<th>Request Response (p. 292)</th>
<th>Run a Job (.sync) (p. 292)</th>
<th>Wait for Callback (.waitForTaskToken) (p. 293)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda (p. 301)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Batch (p. 303)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DynamoDB (p. 304)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon ECS/AWS Fargate (p. 307)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon SNS (p. 309)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon SQS (p. 311)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Glue (p. 312)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SageMaker (p. 313)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EMR (p. 320)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EMR on EKS (p. 327)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeBuild (p. 329)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Athena (p. 332)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EKS (p. 334)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>API Gateway (p. 343)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Glue DataBrew (p. 348)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EventBridge (p. 349)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Step Functions (p. 351)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
## AWS Step Functions Developer Guide

### Integration pattern support

<table>
<thead>
<tr>
<th>Service</th>
<th>Request Response (p. 292)</th>
<th>Run a Job (.sync) (p. 292)</th>
<th>Wait for Callback (.waitForTaskToken) (p. 293)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS SDK integrations</td>
<td>Over two hundred (p. 275)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Express Workflows

#### Supported service integrations

<table>
<thead>
<tr>
<th>Service</th>
<th>Request Response (p. 292)</th>
<th>Run a Job (.sync) (p. 292)</th>
<th>Wait for Callback (.waitForTaskToken) (p. 293)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized integrations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambda (p. 301)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Batch (p. 303)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DynamoDB (p. 304)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon ECS/AWS Fargate (p. 307)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon SNS (p. 309)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon SQS (p. 311)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Glue (p. 312)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SageMaker (p. 313)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EMR (p. 320)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EMR on EKS (p. 327)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeBuild (p. 329)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Athena (p. 332)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EKS (p. 334)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>API Gateway (p. 343)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Glue DataBrew (p. 348)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EventBridge (p. 349)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Step Functions (p. 351)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
AWS SDK service integrations

AWS Step Functions integrates with AWS services, letting you call each service's API actions from your workflow. You can use Step Functions' AWS SDK integrations (p. 274) to call almost any AWS service's API actions from your state machine. You can also use Step Functions's Optimized integrations (p. 299), each of which has been customized to provide special functionality for your workflow.

Some services or SDKs may not be available as AWS SDK integrations, either temporarily or permanently. Recently released services may not have SDK interactions available until a later update. Some services require customized configuration, such as specifying a customer-specific endpoint, which may be more suitable for an optimized integration. Other SDKs are unsuitable for use in a workflow, such as those for streaming audio or video. Finally, some services may be withheld until they pass certain internal validations performed by Step Functions.

Topics
- Using AWS SDK service integrations (p. 274)
- Supported AWS SDK service integrations (p. 275)
- Unsupported API actions for supported services (p. 290)
- Deprecated AWS SDK service integrations (p. 291)

Using AWS SDK service integrations

To use AWS SDK integrations, you specify the service name and API call and, optionally, a service integration pattern (p. 291).

Note
The API action will always be camel case, and parameter names will be Pascal case. For example, you could use Step Functions' startSyncExecution API action and specify the parameter StateMachineArn.

You can call AWS SDK services directly from the Amazon States Language in the Resource field of a task state. To do this, use the following syntax:

```
arn:aws:states:::aws-sdk:serviceName:apiAction.[serviceIntegrationPattern]
```

For example, for Amazon EC2, you might use `arn:aws:states:::aws-sdk:ec2:describeInstances`. This would return output as defined for the Amazon EC2 describeInstances API call.

If an AWS SDK integration encounters an error, the resulting Error field will be composed of the service name and the error name, separated by a period character: `ServiceName.ErrorName`. Both the service name and error name are in Pascal case. You can also see the service name, in lowercase, in the Task state's Resource field. You can find the potential error names in the target service's API reference documentation.

For example, you might use the `arn:aws:states:::aws-sdk:acmpca:deleteCertificateAuthority` AWS SDK integration. The AWS Certificate Manager Private Certificate Authority API Reference indicates that the deleteCertificateAuthority API action can result in a ResourceNotFoundException, for example. To handle this error, you would thus...
specify the Error AcmPca.ResourceNotFoundException in your Task state's Retriers or Catchers. For more information on error handling in Step Functions, see Error handling in Step Functions (p. 88).

Step Functions can't autogenerate IAM policies for AWS SDK integrations. After you create your state machine, you will need to navigate to the IAM console and configure your role policies. See IAM Policies for integrated services (p. 531) for more information.

See the Gather Amazon S3 bucket info using AWS SDK service integrations (p. 201) tutorial for an example of how to use AWS SDK integrations.

## Supported AWS SDK service integrations

The following table lists the AWS SDK service integrations supported by Step Functions. The Task state resource column lists the syntax to call a specific API action when using the integration for the service specified in the Service name column on the left. The Date supported column provides information about the dates on which the API actions were supported for each of the services. In addition, the Exception prefix column on the right, lists the exception prefixes for each service integration. These exception prefixes are present in the exceptions that are generated when you erroneously perform an AWS SDK service integration with Step Functions.

<table>
<thead>
<tr>
<th>Service name</th>
<th>Task state resource</th>
<th>Date supported</th>
<th>Exception prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS Audit Manager</td>
<td>arn:aws:states:::aws-sdk:auditmanager:[apiAction]</td>
<td>September 30, 2021</td>
<td>AuditManager</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------------------------------------------------------</td>
<td>----------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>----------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------------------------------------------------------</td>
<td>-----------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>AWS Ground Station</td>
<td>arn:aws:states:::aws-sdk:groundstation:[apiAction]</td>
<td>September 30, 2021</td>
<td>GroundStation</td>
</tr>
<tr>
<td>AWS IoT 1-Click</td>
<td>arn:aws:states:::aws-sdk:iot1clickprojects:[apiAction]</td>
<td>September 30, 2021</td>
<td>Iot1ClickProjects</td>
</tr>
<tr>
<td>AWS IoT Things Graph</td>
<td>arn:aws:states:::aws-sdk:iotthingsgraph:[apiAction]</td>
<td>September 30, 2021</td>
<td>IoTThingsGraph</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>----------------------------------------------</td>
<td>--------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>AWS License Manager</td>
<td><code>arn:aws:states:::aws-sdk:licensemanager:</code> + [apiAction]</td>
<td>September 30, 2021</td>
<td>LicenseManager</td>
</tr>
</tbody>
</table>
## Supported services

<table>
<thead>
<tr>
<th>Service name</th>
<th>Task state resource</th>
<th>Date supported</th>
<th>Exception prefix</th>
</tr>
</thead>
</table>
## Supported services

<table>
<thead>
<tr>
<th>Service name</th>
<th>Task state resource</th>
<th>Date supported</th>
<th>Exception prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>----------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Amazon Chime Meetings</td>
<td>arn:aws:states:::aws-sdk:chimesdkmeetings:[apiAction]</td>
<td>April 19, 2022</td>
<td>ChimeSdkMeetings</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td>--------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>Amazon Comprehend Medical</td>
<td>arn:aws:states:::aws-sdk:comprehendmedical:[apiAction]</td>
<td>September 30, 2021</td>
<td>ComprehendMedical</td>
</tr>
<tr>
<td>Amazon Connect Contact Lens</td>
<td>arn:aws:states:::aws-sdk:connectcontactlens:[apiAction]</td>
<td>September 30, 2021</td>
<td>ConnectContactLens</td>
</tr>
<tr>
<td>Amazon Data Lifecycle Manager</td>
<td>arn:aws:states:::aws-sdk:dlm:[apiAction]</td>
<td>September 30, 2021</td>
<td>Dlm</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-----------------------------------------------------------------</td>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Amazon FSx</td>
<td>arn:aws:states:::aws-sdk:fsx:[apiAction]</td>
<td>September 30, 2021</td>
<td>FSx</td>
</tr>
<tr>
<td>Amazon Fraud Detector</td>
<td>arn:aws:states:::aws-sdk:frauddetector:[apiAction]</td>
<td>September 30, 2021</td>
<td>FraudDetector</td>
</tr>
<tr>
<td>Amazon GameLift</td>
<td>arn:aws:states:::aws-sdk:gamelift:[apiAction]</td>
<td>September 30, 2021</td>
<td>GameLift</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td>----------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>---------------------</td>
<td>----------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Amazon Location Service</td>
<td><code>arn:aws:states:::aws-sdk:location:</code></td>
<td>September 30, 2021</td>
<td>Location</td>
</tr>
<tr>
<td>Amazon MQ</td>
<td><code>arn:aws:states:::aws-sdk:mq:</code></td>
<td>September 30, 2021</td>
<td>Mq</td>
</tr>
<tr>
<td>Amazon Macie</td>
<td><code>arn:aws:states:::aws-sdk:macie:</code></td>
<td>September 30, 2021</td>
<td>Macie2</td>
</tr>
<tr>
<td>Amazon Managed Blockchain</td>
<td><code>arn:aws:states:::aws-sdk:managedblockchain:</code></td>
<td>September 30, 2021</td>
<td>ManagedBlockchain</td>
</tr>
<tr>
<td>Amazon Managed Grafana</td>
<td><code>arn:aws:states:::aws-sdk:grafana:</code></td>
<td>April 19, 2022</td>
<td>Grafana</td>
</tr>
<tr>
<td>Amazon Managed Service for Prometheus</td>
<td><code>arn:aws:states:::aws-sdk:amp:</code></td>
<td>September 30, 2021</td>
<td>Amp</td>
</tr>
<tr>
<td>Amazon MSK Connect</td>
<td><code>arn:aws:states:::aws-sdk:kafkaconnect:</code></td>
<td>April 19, 2022</td>
<td>KafkaConnect</td>
</tr>
<tr>
<td>Amazon Managed Workflows for Apache Airflow</td>
<td><code>arn:aws:states:::aws-sdk:mwaa:</code></td>
<td>September 30, 2021</td>
<td>Mwaa</td>
</tr>
<tr>
<td>Amazon MemoryDB for Redis</td>
<td><code>arn:aws:states:::aws-sdk:memorydb:</code></td>
<td>April 19, 2022</td>
<td>MemoryDB</td>
</tr>
<tr>
<td>Amazon Personalize Events</td>
<td><code>arn:aws:states:::aws-sdk:personalizeevents:</code></td>
<td>September 30, 2021</td>
<td>PersonalizeEvents</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>---------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Amazon Pinpoint Email Service</td>
<td>arn:aws:states:::aws-sdk:pinpointemail:[apiAction]</td>
<td>September 30, 2021</td>
<td>PinpointEmail</td>
</tr>
<tr>
<td>Amazon QLDB Session</td>
<td>arn:aws:states:::aws-sdk:qldbsession:[apiAction]</td>
<td>September 30, 2021</td>
<td>QldbSession</td>
</tr>
<tr>
<td>Amazon S3 on Outposts</td>
<td>arn:aws:states:::aws-sdk:s3outposts:[apiAction]</td>
<td>September 30, 2021</td>
<td>S3Outposts</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td>------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Service name</td>
<td>Task state resource</td>
<td>Date supported</td>
<td>Exception prefix</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Amazon S3 Control</td>
<td>arn:aws:states:::aws-sdk:s3control:[apiAction]</td>
<td>September 30, 2021</td>
<td>S3Control</td>
</tr>
</tbody>
</table>

1 (p. 275) For information about unsupported API actions for AWS Application Discovery Service integration, see Unsupported API actions for supported services (p. 290).

2 (p. 276) For information about unsupported API actions for AWS CodeDeploy integration, see Unsupported API actions for supported services (p. 290).

3 (p. 277) For information about unsupported API actions for AWS Direct Connect integration, see Unsupported API actions for supported services (p. 290).

4 (p. 277) For information about unsupported API action for AWS Elemental MediaPackage integration, see Unsupported API actions for supported services (p. 290).

5 (p. 278) For information about unsupported API actions for AWS IoT integration, see Unsupported API actions for supported services (p. 290).

6 (p. 278) For information about unsupported API action for AWS IoT Core Device Advisor integration, see Unsupported API actions for supported services (p. 290).
### Unsupported API actions for supported services

The following table lists the unsupported API actions for AWS SDK service integrations. The right column contains the API actions that are currently not supported for the service listed in the left column.

<table>
<thead>
<tr>
<th>Service name</th>
<th>Unsupported API action</th>
</tr>
</thead>
</table>
| AWS Application Discovery Service   | • DescribeExportConfigurations
                                         • ExportConfigurations |
| AWS CodeDeploy                      | • BatchGetDeploymentInstances
                                         • GetDeploymentInstance
                                         • ListDeploymentInstances
                                         • SkipWaitTimeForInstanceTermination |
| Amazon Comprehend Medical            | DetectEntities                                              |

7 (p. 279) For information about unsupported API action for AWS Lambda integration, see Unsupported API actions for supported services (p. 290).

8 (p. 280) For information about unsupported API action for Amazon Relational Database Service integration, see Unsupported API actions for supported services (p. 290).

9 (p. 280) For information about unsupported API actions for AWS Security Token Service integration, see Unsupported API actions for supported services (p. 290).

10 (p. 280) For information about unsupported API action for AWS Shield integration, see Unsupported API actions for supported services (p. 290).

11 (p. 283) For information about unsupported API action for Amazon Comprehend Medical integration, see Unsupported API actions for supported services (p. 290).

12 (p. 283) For information about unsupported API action for Amazon EMR integration, see Unsupported API actions for supported services (p. 290).

13 (p. 284) For information about unsupported API action for Amazon Elastic File System integration, see Unsupported API actions for supported services (p. 290).

14 (p. 284) For information about unsupported API action for Amazon Elastic Transcoder integration, see Unsupported API actions for supported services (p. 290).

15 (p. 285) For information about unsupported API action for Amazon Kinesis integration, see Unsupported API actions for supported services (p. 290).

16 (p. 285) For information about unsupported API action for Amazon Lex Runtime V2 integration, see Unsupported API actions for supported services (p. 290).

17 (p. 288) For information about unsupported API action for Amazon Simple Storage Service integration, see Unsupported API actions for supported services (p. 290).

18 (p. 289) For information about unsupported API actions for Amazon S3 Control integration, see Unsupported API actions for supported services (p. 290)
### Deprecated AWS SDK service integrations

The following AWS SDK service integrations are now deprecated:

- AWS Mobile
- Amazon Macie

#### Service Integration Patterns

AWS Step Functions integrates with services directly in the Amazon States Language. You can control these AWS services using three service integration patterns:

- Call a service and let Step Functions progress to the next state immediately after it gets an HTTP response.
• Call a service and have Step Functions wait for a job to complete.
• Call a service with a task token and have Step Functions wait until that token is returned with a payload.

Each of these service integration patterns is controlled by how you create a URI in the "Resource" field of your task definition (p. 29).

Ways to Call an Integrated Service
• Request Response (p. 292)
• Run a Job (.sync) (p. 292)
• Wait for a Callback with the Task Token (p. 293)

For information about configuring AWS Identity and Access Management (IAM) for integrated services, see IAM Policies for integrated services (p. 531).

Request Response

When you specify a service in the "Resource" string of your task state, and you only provide the resource, Step Functions will wait for an HTTP response and then progress to the next state. Step Functions will not wait for a job to complete.

The following example shows how you can publish an Amazon SNS topic.

"Send message to SNS":{
  "Type":"Task",
  "Resource":"arn:aws:states:::sns:publish",
  "Parameters":{
    "Message":"Hello from Step Functions!"
  },
  "Next":"NEXT_STATE"
}

This example references the Publish API of Amazon SNS. The workflow progresses to the next state after calling the Publish API.

Run a Job (.sync)

For integrated services such as AWS Batch and Amazon ECS, Step Functions can wait for a request to complete before progressing to the next state. To have Step Functions wait, specify the "Resource" field in your task state definition with the .sync suffix appended after the resource URI.

For example, when submitting an AWS Batch job, use the "Resource" field in the state machine definition as shown in this example.

"Manage Batch task": {
  "Type": "Task",
  "Resource": "arn:aws:states:::batch:submitJob.sync",
  "Parameters": {
    "JobName": "testJob",
    "JobQueue": "arn:aws:batch:us-east-2:123456789012:job-queue/testQueue"
  },
  "Next": "NEXT_STATE"
Having the `.sync` portion appended to the resource Amazon Resource Name (ARN) means that Step Functions waits for the job to complete. After calling AWS Batch `submitJob`, the workflow pauses. When the job is complete, Step Functions progresses to the next state. For more information, see the AWS Batch sample project: Manage a Batch Job (AWS Batch, Amazon SNS) (p. 355).

If a task using this (.sync) service integration pattern is aborted, and Step Functions is unable to cancel the task, you might incur additional charges from the integrated service. A task can be aborted if:

- The state machine execution is stopped.
- A different branch of a Parallel state fails with an uncaught error.
- An iteration of a Map state fails with an uncaught error.

Step Functions will make a best-effort attempt to cancel the task. For example, if a Step Functions `states:startExecution.sync` task is aborted, it will call the Step Functions `StopExecution` API action. However, it is possible that Step Functions will be unable to cancel the task. Reasons for this include, but are not limited to:

- Your IAM execution role lacks permission to make the corresponding API call.
- A temporary service outage occurred.

To see a list of what integrated services support waiting for a job to complete (.sync), see Optimized integrations for Step Functions (p. 299).

**Note**
Service integrations that use the .sync pattern require additional IAM permissions. For more information, see IAM Policies for integrated services (p. 531).

**Wait for a Callback with the Task Token**

Callback tasks provide a way to pause a workflow until a task token is returned. A task might need to wait for a human approval, integrate with a third party, or call legacy systems. For tasks like these, you can pause Step Functions indefinitely, and wait for an external process or workflow to complete. For these situations Step Functions allows you to pass a task token to the AWS SDK service integrations, and also to some Optimized integrations integrated services. The task will pause until it receives that task token back with a `SendTaskSuccess` or `SendTaskFailure` call.

If a Task state using the callback task token times out, a new random token is generated. You can access the task tokens from the context object (p. 71).

**Note**
A task token must contain at least one character, and cannot exceed 1024 characters.

To use `.waitForTaskToken` with an AWS SDK integration, the API you use must have a parameter field in which to place the task token.

**Note**
You must pass task tokens from principals within the same AWS account. The tokens won't work if you send them from principals in a different AWS account.

To see a list of what integrated services support waiting for a task token (.waitForTaskToken), see Optimized integrations for Step Functions (p. 299).

**Topics**
- Task Token Example (p. 294)
Task Token Example

In this example, a Step Functions workflow needs to integrate with an external microservice to perform a credit check as a part of an approval workflow. Step Functions publishes an Amazon SQS message that includes a task token as a part of the message. An external system integrates with Amazon SQS, and pulls the message off the queue. When that's finished, it returns the result and the original task token. Step Functions then continues with its workflow.

```
"Send message to SQS": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
  "Parameters": {
    "QueueUrl": "https://sqs.us-east-2.amazonaws.com//ACCOUNT_ID/myQueue",
    "MessageBody": {
      "Input": "$",  
      "TaskToken.$": "$$.Task.Token"
    }
  },
  "Next": "NEXT_STATE"
}
```

This tells Step Functions to pause and wait for the task token. When you specify a resource using .waitForTaskToken, the task token can be accessed in the "Parameters" field of your state definition with a special path designation ($$.Task.Token). The initial $$ designates that the path accesses the context object (p. 295), and gets the task token for the current task in a running execution.
When it’s complete, the external service calls `SendTaskSuccess` or `SendTaskFailure` with the `taskToken` included. Only then does the workflow continue to the next state.

**Note**
To avoid waiting indefinitely if a process fails to send the task token with `SendTaskSuccess` or `SendTaskFailure`, see [Configure a Heartbeat Timeout for a Waiting Task](p. 296).

### Get a Token from the Context Object

The context object is an internal JSON object that contains information about your execution. Like state input, it can be accessed with a path from the "Parameters" field during an execution. When accessed from within a task definition, it includes information about the specific execution, including the task token.

```json
{
  "Execution": {
    "Id": "arn:aws:states:us-east-1:123456789012:execution:stateMachineName:executionName",
    "Input": {
      "key": "value"
    },
    "Name": "executionName",
    "RoleArn": "arn:aws:iam::123456789012:role...",
    "StartTime": "2019-03-26T20:14:13.192Z"
  },
  "State": {
    "Name": "Test",
    "RetryCount": 3
  },
  "StateMachine": {
    "Id": "arn:aws:states:us-east-1:123456789012:stateMachine:stateMachineName",
    "Name": "name"
  },
  "Task": {
    "Token": "h7XRiCdLtd/83p1E0dMccoxlzFhglsdkzpK9mBVKZap?d9yrTlW"
  }
}
```

You can access the task token by using a special path from inside the "Parameters" field of your task definition. To access the input or the context object, you first specify that the parameter will be a path by appending a `.` to the parameter name. The following specifies nodes from both the input and the context object in a "Parameters" specification.

```json
"Parameters": {
  "Input.": ",
  "TaskToken.": ",
}
```

In both cases, appending `.=` to the parameter name tells Step Functions to expect a path. In the first case, `"="` is a path that includes the entire input. In the second case, `"=".specifies that the path will access the context object, and `"=".Task.Token sets the parameter to the value of the task token in the context object of a running execution.

In the Amazon SQS example, `.waitForTaskToken` in the "Resource" field tells Step Functions to wait for the task token to be returned. The "TaskToken.": "".Task.Token" parameter passes that token as a part of the Amazon SQS message.

```json
"Send message to SQS": {
  "Type": "Task",
}
```
"Resource": "arn:aws:states:::sqs:sendMessage/waitForTaskToken",
"Parameters": {
  "QueueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/myQueue",
  "MessageBody": {
    "Message": "Hello from Step Functions!",
    "TaskToken.$": "$.Task.Token"
  }
},
"Next": "NEXT_STATE"

For more information about the context object, see Context Object (p. 70) in the Input and Output Processing (p. 55) section in this guide.

Configure a Heartbeat Timeout for a Waiting Task

A task that is waiting for a task token will wait until the execution reaches the one year service quota (see, Quotas related to state throttling (p. 486)). To avoid stuck executions you can configure a heartbeat timeout interval in your state machine definition. Use the HeartbeatSeconds (p. 29) field to specify the timeout interval.

{  
  "StartAt": "Push to SQS",
  "States": {
    "Push to SQS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage/waitForTaskToken",
      "HeartbeatSeconds": 600,
      "Parameters": {
        "MessageBody": { "myTaskToken.$": "$.Task.Token" },
        "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/push-based-queue"
      },
      "ResultPath": "$.SQS",
      "End": true
    }
  }
}

In this state machine definition, a task pushes a message to Amazon SQS and waits for an external process to call back with the provided task token. The "HeartbeatSeconds": 600 field sets the heartbeat timeout interval to 10 minutes. The task will wait for the task token to be returned with one of these API actions:

- SendTaskSuccess
- SendTaskFailure
- SendTaskHeartbeat

If the waiting task doesn't receive a valid task token within that 10-minute period, the task fails with a States.Timeout error name.

For more information, see the callback task sample project Callback Pattern Example (Amazon SQS, Amazon SNS, Lambda) (p. 369).

Pass Parameters to a Service API

Use the "Parameters" field in a Task state to control what parameters are passed to a service API.
Pass Static JSON as Parameters

You can include a JSON object directly in your state machine definition to pass as a parameter to a resource.

For example, to set the `RetryStrategy` parameter for the `SubmitJob` API for AWS Batch, you could include the following in your parameters.

```json
"RetryStrategy": {
  "attempts": 5
}
```

You can also pass multiple parameters with static JSON. As a more complete example, the following are the "Resource" and "Parameters" fields of the specification of a task that publishes to an Amazon SNS topic.

```json
"Resource": "arn:aws:states:::sns:publish",
"Parameters": {
  "Message": "test message",
  "MessageAttributes": {
    "my attribute no 1": {
      "DataType": "String",
      "StringValue": "value of my attribute no 1"
    },
    "my attribute no 2": {
      "DataType": "String",
      "StringValue": "value of my attribute no 2"
    }
  }
}
```

Pass State Input as Parameters Using Paths

You can pass portions of the state input into parameters by using paths (p. 57). A path is a string, beginning with $, that's used to identify components within JSON text. Step Functions paths use JsonPath syntax.

To specify that a parameter use a path to reference a JSON node in the input, end the parameter name with $. For example, if you have text in your state input in a node named `message`, you could pass that to a parameter by referencing the input JSON with a path.

Using the following state input.

```json
{
  "comment": "A message in the state input",
  "input": {
    "message": "foo",
    "otherInfo": "bar"
  },
  "data": "example"
}
```

You could pass the message `foo` as a parameter using the following.

```json
"Parameters": {"Message.$": ".input.message"},
```
Pass Context Object Nodes as Parameters

In addition to static content, and nodes from the state input, you can pass nodes from the context object as parameters. The context object is dynamic JSON data that exists during a state machine execution. It includes information about your state machine and the current execution. You can access the context object using a path in the "Parameters" field of a state definition.

For more information about the context object and how to access that data from a "Parameters" field, see the following:

- Context Object (p. 70)
- Accessing the Context Object (p. 71)
- Get a Token from the Context Object (p. 295)

Code Snippets

In AWS Step Functions, code snippets are a way to easily configure the options for a new state in your state machine definition. When you edit or create a state machine, the top of the code pane includes a Generate code snippet menu. Selecting an option from the Generate code snippet menu opens a window to configure parameters specific to that state, and generates Amazon States Language code based on the options you choose.

For example, if you choose the AWS Batch: Manage a job code snippet, you can configure the following:

- **Batch job name** – You can either specify the job name, or specify it at runtime using a path.
- **Batch job definition** – You can select the Amazon Resource Name (ARN) of an existing AWS Batch job in your account, enter the job definition, or choose to specify it at runtime using a path.
- **Batch job queue** – You can select the ARN of an existing AWS Batch job queue in your account, enter the job queue definition, or choose to specify it at runtime using a path.
- **Run synchronously** – Selecting this option configures Step Functions to wait until the AWS Batch job completes before continuing to the next state.

**Note**
For more information about specifying service parameters, see Pass Parameters to a Service API (p. 296).

After you configure your AWS Batch options, you can specify error handling options for your state, such as Retry, Catch, and TimeoutSeconds.

For more information, see Error names (p. 89) in Amazon States Language.

To learn more about Step Functions service integrations, see the following:

- Using AWS Step Functions with other services (p. 271)
- Optimized integrations for Step Functions (p. 299)
- Using Code Snippets (p. 181)
Optimized integrations for Step Functions

The following topics include the supported APIs, parameters, and request/response syntax in the Amazon States Language for coordinating other AWS services. The topics also provide example code. You can call Optimized integrations services directly from the Amazon States Language in the Resource field of a Task state.

You can use three service integration patterns:

- Default response (p. 292)
- Wait for a job to complete (.sync). (p. 292)
- Wait for a task token (.waitForTaskToken). (p. 293)

Standard Workflows and Express Workflows support the same integrations but do not support the same integration patterns. Express Workflows do not support Run a Job (.sync) or Wait for Callback (.waitForTaskToken). Optimized integrations pattern support is different for each integration. For more information, see Standard vs. Express Workflows (p. 19).

Standard Workflows

**Supported service integrations**

<table>
<thead>
<tr>
<th>Service</th>
<th>Request Response (p. 292)</th>
<th>Run a Job (.sync) (p. 292)</th>
<th>Wait for Callback (.waitForTaskToken) (p. 293)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized integrations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambda (p. 301)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>AWS Batch (p. 303)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DynamoDB (p. 304)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon ECS/AWS Fargate (p. 307)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Amazon SNS (p. 309)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Amazon SQS (p. 311)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>AWS Glue (p. 312)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SageMaker (p. 313)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Amazon EMR (p. 320)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Amazon EMR on EKS (p. 327)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CodeBuild (p. 329)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Athena (p. 332)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
## Optimized integrations

<table>
<thead>
<tr>
<th>Service</th>
<th>Request Response (p. 292)</th>
<th>Run a Job (.sync) (p. 292)</th>
<th>Wait for Callback (.waitForTaskToken) (p. 293)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon EKS (p. 334)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>API Gateway (p. 343)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>AWS Glue DataBrew (p. 348)</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Amazon EventBridge (p. 349)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Step Functions (p. 351)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>AWS SDK integrations Over two hundred (p. 275)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Express Workflows

### Supported service integrations

<table>
<thead>
<tr>
<th>Service</th>
<th>Request Response (p. 292)</th>
<th>Run a Job (.sync) (p. 292)</th>
<th>Wait for Callback (.waitForTaskToken) (p. 293)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized integrations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambda (p. 301)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Batch (p. 303)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DynamoDB (p. 304)</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Amazon ECS/AWS Fargate (p. 307)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon SNS (p. 309)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Amazon SQS (p. 311)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Glue (p. 312)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SageMaker (p. 313)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EMR (p. 320)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon EMR on EKS (p. 327)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeBuild (p. 329)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Athena (p. 332)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Invoke Lambda with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

**How the Optimized Lambda integration is different than the Lambda AWS SDK integration**

- The Payload field of the response is parsed from escaped Json to Json.
- If the response contains the field FunctionError, the task fails.

For more information about managing state input, output, and results, see Input and Output Processing in Step Functions (p. 55).

Supported AWS Lambda APIs:

- **Invoke**
  - **Request Syntax**
  - **Supported Parameters**
    - ClientContext
    - FunctionName
    - InvocationType
    - Qualifier
    - Payload
  - **Response syntax**

The following includes a Task state that invokes a Lambda function.
The following includes a Task state that implements the callback (p. 293) service integration pattern.

```json
{
  "StartAt":"GetManualReview",
  "States":{
    "GetManualReview":{
      "Type":"Task",
      "Resource":"arn:aws:states:::lambda:invoke.waitForTaskToken",
      "Parameters":{
        "FunctionName":"get-model-review-decision",
        "Payload":{
          "model.$":"$.new_model",
          "token.$":"$$\$.Task.Token"
        },
        "Qualifier":"prod-v1"
      },
      "End":true
    }
  }
}
```

When you invoke a Lambda function, the execution will wait for the function to complete. However, it is possible to call Lambda asynchronously using the InvocationType parameter, as seen in the following example:

```json
{
  "Comment": "A Hello World example of the Amazon States Language using Pass states",
  "StartAt": "Hello",
  "States": {
    "Hello": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Parameters": {
        "FunctionName": "echo",
        "InvocationType": "Event"
      },
      "End": true
    }
  }
}
```

When the Task result is returned, the function output is nested inside a dictionary of metadata. For example:

```json
{
}
```
Alternatively, you can invoke a Lambda function by specifying a function ARN directly in the "Resource" field. When you invoke a Lambda function in this way, you can't specify .waitTaskToken, and the task result contains only the function output.

```json
{
    "StartAt":"CallFunction",
    "States":{
        "CallFunction":{
            "Type":"Task",
            "End":true
        }
    }
}
```

You can invoke a specific Lambda function version or alias by specifying those options in the ARN in the Resource field. See the following in the Lambda documentation:

- AWS Lambda versioning
- AWS Lambda aliases

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

Manage AWS Batch with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

Supported AWS Batch APIs:

**How the Optimized AWS Batch integration is different than the AWS BatchAWS SDK integration**

- The Run a Job (.sync) (p. 292) integration pattern is available.
Note that there are no optimizations for the Request Response (p. 292) or Wait for a Callback with the Task Token (p. 293) integration patterns.

**Note**
Parameters in Step Functions are expressed in **PascalCase**, even when the native service API is **camelCase**.

- **SubmitJob**
  - **Request syntax**
  - **Supported parameters**:
    - **ArrayProperties**
    - **ContainerOverrides**
    - **DependsOn**
    - **JobDefinition**
    - **JobName**
    - **JobQueue**
    - **Parameters**
    - **RetryStrategy**
    - **Timeout**
  - **Response syntax**

The following includes a **Task** state that submits an AWS Batch job and waits for it to complete.

```
{
  "StartAt": "BATCH_JOB",
  "States": {
    "BATCH_JOB": {
      "Type": "Task",
      "Resource": "arn:aws:states:::batch:submitJob.sync",
      "Parameters": {
        "JobDefinition": "preprocessing",
        "JobName": "PreprocessingBatchJob",
        "JobQueue": "SecondaryQueue",
        "Parameters.$": ".batchjob.parameters",
        "ContainerOverrides": {
          "ResourceRequirements": [
            {
              "Type": "VCPU",
              "Value": "4"
            }
          ]
        }
      }
    },
    "End": true
  }
}
```

For information on how to configure IAM when using Step Functions with other AWS services, see **IAM Policies for integrated services (p. 531)**.

### Call DynamoDB APIs with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- **Working with other services (p. 271)**
Pass Parameters to a Service API (p. 296)

Note
There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 262,144 bytes of data as a UTF-8 encoded string when you send to, or receive data from, another service. See Quotas related to state machine executions (p. 487).

How the optimized DynamoDB integration is different than the DynamoDB AWS SDK integration

• There is no optimization for the Request Response (p. 292) integration pattern.
• The Wait for a Callback with the Task Token (p. 293) integration pattern is not supported.
• Only GetItem, PutItem, and UpdateItem API actions are available through optimized integration. Other API actions, such as CreateTable are available using the DynamoDB AWS SDK integration.

Supported Amazon DynamoDB APIs and syntax:

• GetItem
  • Request syntax
  • Supported parameters:
    • Key
    • TableName
    • AttributesToGet
    • ConsistentRead
    • ExpressionAttributeNames
    • ProjectionExpression
    • ReturnConsumedCapacity
  • Response syntax
• PutItem
  • Request syntax
  • Supported parameters:
    • Item
    • TableName
    • ConditionalOperator
    • ConditionExpression
    • Expected
    • ExpressionAttributeNames
    • ExpressionAttributeValues
    • ReturnConsumedCapacity
    • ReturnItemCollectionMetrics
    • ReturnValues
  • Response syntax
• DeleteItem
  • Request syntax
  • Supported parameters:
    • Key
    • TableName
The following is a Task state that retrieves a message from DynamoDB.

```json
"Read Next Message from DynamoDB": {
  "Type": "Task",
  "Resource": "arn:aws:states:::dynamodb:getItem",
  "Parameters": {
    "TableName": "TransferDataRecords-DDBTable-3J41R5L5EAGT",
    "Key": {
      "MessageId": {"S.$": ".List[0]"}
    }
  },
  "ResultPath": ".DynamoDB",
  "Next": "Send Message to SQS"
},
```

**Note**
You cannot pass a map or list to DynamoDB inside a map.

To see this state in a working example, see the Transfer Data Records (Lambda, DynamoDB, Amazon SQS) (p. 361) sample project.

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
Manage Amazon ECS or Fargate Tasks with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

How the Optimized Amazon ECS/Fargate integration is different than the Amazon ECS or Fargate AWS SDK integration

- The .sync integration pattern is available.

- *ecs:runTask* can return an HTTP 200 response, but have a non-empty *Failures* field as follows:
  - **Request Response**: return the response and do not fail the task. This is the same as no optimization.
  - **Run a Job**: if a non-empty *Failures* field is encountered, the task is failed with an *AmazonECS.Unknown* error.
  - **Task Token**: if a non-empty *Failures* field is encountered, the task is failed with an *AmazonECS.Unknown* error.

Supported Amazon ECS/Fargate APIs and syntax:

**Note**

Parameters in Step Functions are expressed in *PascalCase*, even when the native service API is *camelCase*.

- **RunTask** starts a new task using the specified task definition.
  - **Request syntax**
  - Supported parameters:
    - *Cluster*
    - *Group*
    - *LaunchType*
    - *NetworkConfiguration*
    - *Overides*
    - *PlacementConstraints*
    - *PlacementStrategy*
    - *PlatformVersion*
    - *PropagateTags*
    - *TaskDefinition*
  - **Response syntax**

**Note**

For the *Overides* parameter, Step Functions does not support *executionRoleArn* or *taskRoleArn* as *ContainerOverides*.  

307
Passing Data to an Amazon ECS Task

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

You can use overrides to override the default command for a container, and pass input to your Amazon ECS tasks. See ContainerOverride. In the example, we have used JsonPath to pass values to the Task from the input to the Task state.

The following includes a Task state that runs an Amazon ECS task and waits for it to complete.

```json
{
  "StartAt": "Run an ECS Task and wait for it to complete",
  "States": {
    "Run an ECS Task and wait for it to complete": {
      "Type": "Task",
      "Resource": "arn:aws:states:::ecs:runTask.sync",
      "Parameters": {
        "Cluster": "cluster-arn",
        "TaskDefinition": "job-id",
        "Overrides": {
          "ContainerOverrides": [
            {
              "Name": "container-name",
              "Command.$": "$\.commands"
            }
          ]
        }
      }
    },
    "End": true
  }
}
```

The "Command.$": "$.commands" line in ContainerOverrides passes the commands from the state input to the container.

For the previous example, each of the commands will be passed as a container override if the input to the execution is the following.

```json
{
  "commands": [
    "test command 1",
    "test command 2",
    "test command 3"
  ]
}
```

The following includes a Task state that runs an Amazon ECS task, and then waits for the task token to be returned. See Wait for a Callback with the Task Token (p. 293).

```json
{
  "StartAt": "Manage ECS task",
  "States": {
    "Manage ECS task": {
      "Type": "Task",
      "Resource": "arn:aws:states:::ecs:runTask.waitForTaskToken",
```
"Parameters":{
  "LaunchType": "FARGATE",
  "Cluster": "cluster-arn",
  "TaskDefinition": "job-id",
  "Overrides":{
    "ContainerOverrides":{
      "Name": "container-name",
      "Environment":{
        "Name": "TASK_TOKEN_ENV_VARIABLE",
        "Value.$": "$$.Task.Token"
      }
    }
  },
  "End":true
}

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

**Call Amazon SNS with Step Functions**

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

**How the Optimized Amazon SNS integration is different than the Amazon SNS AWS SDK integration**

There are no optimizations for the Request Response (p. 292) or Wait for a Callback with the Task Token (p. 293) integration patterns.

Supported Amazon SNS APIs:

**Note**

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 262,144 bytes of data as a UTF-8 encoded string when you send to, or receive data from, another service. See Quotas related to state machine executions (p. 487).

- Publish
  - Request syntax
- Supported Parameters
  - Message
  - MessageAttributes
  - MessageStructure
  - PhoneNumber
  - Subject
  - TargetArn
  - TopicArn
- Response syntax
The following includes a Task state that publishes to an Amazon Simple Notification Service (Amazon SNS) topic.

```json
{
    "StartAt": "Publish to SNS",
    "States": {
        "Publish to SNS": {
            "Type": "Task",
            "Resource": "arn:aws:states:::sns:publish",
            "Parameters": {
                "Message.$": ".input.message",
                "MessageAttributes": {
                    "my_attribute_no_1": {
                        "DataType": "String",
                        "StringValue": "value of my_attribute_no_1"
                    },
                    "my_attribute_no_2": {
                        "DataType": "String",
                        "StringValue": "value of my_attribute_no_2"
                    }
                }
            },
            "End": true
        }
    }
}
```

**Passing dynamic values.** You can modify the above example to dynamically pass an attribute from this JSON payload:

```json
{
    "input": {
        "message": "Hello world"
    },
    "SNSDetails": {
        "attribute1": "some value",
        "attribute2": "some other value"
    }
}
```

Append the `. $` to the `StringValue` field:

```json
"MessageAttributes": {
    "my_attribute_no_1": {
        "DataType": "String",
        "StringValue.$": ".SNSDetails.attribute1"
    },
    "my_attribute_no_2": {
        "DataType": "String",
        "StringValue.$": ".SNSDetails.attribute2"
    }
}
```

The following includes a Task state that publishes to an Amazon SNS topic, and then waits for the task token to be returned. See [Wait for a Callback with the Task Token (p. 293)].

```json
{
    "StartAt": "Send message to SNS",
    "States": {
        "Send message to SNS": {
            "Type": "Task",
```
Call Amazon SQS with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

**How the Optimized Amazon SQS integration is different than the Amazon SQS AWS SDK integration**

There are no optimizations for the Request Response (p. 292) or Wait for a Callback with the Task Token (p. 293) integration patterns.

Supported Amazon SQS APIs:

**Note**

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 262,144 bytes of data as a UTF-8 encoded string when you send to, or receive data from, another service. See Quotas related to state machine executions (p. 487).

- **SendMessage**

  Supported parameters:
  - DelaySeconds
  - MessageAttribute
  - MessageBody
  - MessageDeduplicationId
  - MessageGroupId
  - QueueUrl
  - Response syntax

The following includes a Task state that sends an Amazon Simple Queue Service (Amazon SQS) message.

```json
{
  "StartAt": "Send to SQS",
  "States": {
    "Send to SQS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage",
      "Parameters": {
        "Message": {
          "Input.$": "$",
          "TaskToken.$": "$$:Task.Token"
        }
      },
      "End": true
    }
  }
}
```
"Parameters": {
    "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/myQueue",
    "MessageBody.$": "$.input.message",
    "MessageAttributes": {
        "my attribute no 1": {
            "DataType": "String",
            "StringValue": "attribute1"
        },
        "my attribute no 2": {
            "DataType": "String",
            "StringValue": "attribute2"
        }
    }
},
"End": true
}

The following includes a Task state that publishes to an Amazon SQS queue, and then waits for the task token to be returned. See Wait for a Callback with the Task Token (p. 293).

{
    "StartAt":"Send message to SQS",
    "States":{
        "Send message to SQS":{
            "Type":"Task",
            "Resource":"arn:aws:states:::sqs:sendMessage.waitForTaskToken",
            "Parameters":{
                "QueueUrl": "https://sqs.us-east-1.amazonaws.com/123456789012/myQueue",
                "MessageBody":{
                    "Input.$": ",
                    "TaskToken.$": "$.$.Task.Token"
                }
            },
            "End":true
        }
    }
}

To learn more about receiving messages in Amazon SQS, see Receive and Delete Your Message in the Amazon Simple Queue Service Developer Guide.

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

Manage AWS Glue Jobs with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

How the Optimized AWS Glue integration is different than the AWS Glue AWS SDK integration

- The Run a Job (.sync) (p. 292) integration pattern is available.
- The JobName field is extracted from the request and inserted into the response, which normally only contains JobRunID.
Supported AWS Glue API:

- **StartJobRun**

The following includes a Task state that starts an AWS Glue job.

```json
"Glue StartJobRun": {
  "Type": "Task",
  "Resource": "arn:aws:states:::glue:startJobRun.sync",
  "Parameters": {
    "JobName": "GlueJob-JTrR05198qMG"
  },
  "Next": "ValidateOutput"
},
```

For information on how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services (p. 531)](#).

**Manage SageMaker with Step Functions**

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

**How the Optimized SageMaker integration is different than the SageMaker AWS SDK integration**

- The Run a Job (.sync) (p. 292) integration pattern is supported.
- There are no optimizations for the Request Response (p. 292) integration pattern.
- The Wait for a Callback with the Task Token (p. 293) integration pattern is not supported.

**Supported SageMaker APIs and syntax:**

- **CreateEndpoint**
  - Request syntax
  - Supported parameters:
    - EndpointConfigName
    - EndpointName
    - Tags
  - Response syntax
- **CreateEndpointConfig**
  - Request syntax
  - Supported parameters:
    - EndpointConfigName
    - KmsKeyId
    - ProductionVariants
    - Tags
  - Response syntax
- **CreateHyperParameterTuningJob**
• Request syntax
• Supported parameters:
  • HyperParameterTuningJobConfig
  • HyperParameterTuningJobName
  • Tags
  • TrainingJobDefinition
  • WarmStartConfig
• Response syntax
• CreateLabelingJob
• Request syntax
• Supported parameters:
  • HumanTaskConfig
  • InputConfig
  • LabelAttributeName
  • LabelCategoryConfigS3Uri
  • LabelingJobAlgorithmsConfig
  • LabelingJobName
  • OutputConfig
  • RoleArn
  • StoppingConditions
  • Tags
• Response syntax
• CreateModel
• Request syntax
• Supported parameters:
  • Containers
  • EnableNetworkIsolation
  • ExecutionRoleArn
  • ModelName
  • PrimaryContainer
  • Tags
  • VpcConfig
• CreateProcessingJob
• Request syntax
• Supported parameters:
  • AppSpecification
  • Environment
  • ExperimentConfig
  • NetworkConfig
  • ProcessingInputs
  • ProcessingJobName
  • ProcessingOutputConfig
  • ProcessingResources
  • RoleArn
  • StoppingCondition
• Tags
• Response syntax
• CreateTrainingJob
  • Request syntax
  • Supported parameters:
    • AlgorithmSpecification
    • HyperParameters
    • InputDataConfig
    • OutputDataConfig
    • ResourceConfig
    • RoleArn
    • StoppingCondition
    • Tags
    • TrainingJobName
    • VpcConfig
  • Response syntax
• CreateTransformJob
  • Note
    AWS Step Functions will not automatically create a policy for CreateTransformJob. You must attach an inline policy to the created role. For more information, see this example IAM policy: CreateTrainingJob (p. 540).
  • Request syntax
  • Supported parameters:
    • BatchStrategy
    • Environment
    • MaxConcurrentTransforms
    • MaxPayloadInMB
    • ModelName
    • Tags
    • TransformInput
    • TransformJobName
    • TransformOutput
    • TransformResources
  • Response syntax
• UpdateEndpoint
  • Request syntax
  • Supported parameters:
    • EndpointConfigName
    • EndpointName
  • Response syntax

SageMaker Transform Job Example

The following includes a Task state that creates an Amazon SageMaker transform job, specifying the Amazon S3 location for DataSource and TransformOutput.
SageMaker Training Job Example

The following includes a Task state that creates an Amazon SageMaker training job.

```json
{  
  "SageMaker CreateTransformJob": {  
    "Type": "Task",  
    "Resource": "arn:aws:states:::sagemaker:createTransformJob.sync",  
    "Parameters": {  
      "ModelName": "SageMakerCreateTransformJobModel-9iFBKsYt19vr",  
      "TransformInput": {  
        "CompressionType": "None",  
        "ContentType": "text/csv",  
        "DataSource": {  
          "S3DataSource": {  
            "S3DataType": "S3Prefix",  
            "S3Uri": "s3://my-s3bucket-example-1/TransformJobDataInput.txt"  
          }  
        },  
        "TransformOutput": {  
          "S3OutputPath": "s3://my-s3bucket-example-1/TransformJobOutputPath"  
        },  
        "TransformResources": {  
          "InstanceCount": 1,  
          "InstanceType": "ml.m4.xlarge"  
        },  
        "TransformJobName": "sfn-binary-classification-prediction"  
      },  
      "Next": "ValidateOutput"  
    }  
  },  
  "SageMaker CreateTrainingJob": {  
    "Type": "Task",  
    "Resource": "arn:aws:states:::sagemaker:createTrainingJob.sync",  
    "Parameters": {  
      "TrainingJobName": "search-model",  
      "ResourceConfig": {  
        "InstanceCount": 4,  
        "InstanceType": "ml.c4.8xlarge",  
        "VolumeSizeInGB": 20  
      },  
      "HyperParameters": {  
        "mode": "batch_skipgram",  
        "epochs": "5",  
        "min_count": "5",  
        "sampling_threshold": "0.0001",  
        "learning_rate": "0.025",  
        "window_size": "5",  
        "vector_dim": "300",  
        "negative_samples": "5",  
        "batch_size": "11"  
      },  
      "AlgorithmSpecification": {  
        "TrainingImage": "...",  
        "TrainingInputMode": "File"  
      },  
      "OutputDataConfig": {  
        "S3OutputPath": "s3://bucket-name/doc-search/model"  
      },  
      "StoppingCondition": {  
        "MaxRuntimeInSeconds": 100000  
      }  
    }  
  }
}
SageMaker Labeling Job Example

The following includes a Task state that creates an Amazon SageMaker labeling job.

```json
{
    "StartAt": "SageMaker CreateLabelingJob",
    "TimeoutSeconds": 3600,
    "States": {
```
"SageMaker CreateLabelingJob": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sagemaker:createLabelingJob.sync",
  "Parameters": {
    "HumanTaskConfig": {
      "AnnotationConsolidationConfig": {
      },
      "NumberOfHumanWorkersPerDataObject": 1,
      "TaskDescription": "Classify the following text",
      "TaskKeywords": [
        "tc",
        "Labeling"
      ],
      "TaskTimeLimitInSeconds": 300,
      "TaskTitle": "Classify short bits of text",
      "UiConfig": {
        "UiTemplateS3Uri": "s3://s3bucket-example/TextClassification.template"
      },
    },
    "InputConfig": {
      "DataAttributes": {
        "ContentClassifiers": [
          "FreeOfPersonallyIdentifiableInformation",
          "FreeOfAdultContent"
        ]
      },
      "DataSource": {
        "S3DataSource": {
          "ManifestS3Uri": "s3://s3bucket-example/manifest.json"
        }
      }
    },
    "LabelAttributeName": "Categories",
    "LabelCategoryConfigS3Uri": "s3://s3bucket-example/labelcategories.json",
    "LabelingJobName": "example-job-name",
    "OutputConfig": {
      "S3OutputPath": "s3://s3bucket-example/output"
    },
    "RoleArn": "arn:aws:iam::123456789012:role/service-role/AmazonSageMaker-ExecutionRole",
    "StoppingConditions": {
      "MaxHumanLabeledObjectCount": 10000,
      "MaxPercentageOfInputDatasetLabeled": 100
    }
  },
  "Next": "ValidateOutput"
},
"ValidateOutput": {
  "Type": "Choice",
  "Choices": [
    {
      "Not": {
        "Variable": "$.LabelingJobArn",
        "StringEquals": ""
      },
      "Next": "Succeed"
    }
  ],
  "Default": "Fail"
},
"Succeed": {  
  "Type": "Succeed"  
},  
"Fail": {  
  "Type": "Fail",  
  "Error": "InvalidOutput",  
  "Cause": "Output is not what was expected. This could be due to a service outage or a 
  misconfigured service integration." 
}  
}  

SageMaker Processing Job Example

The following includes a Task state that creates an Amazon SageMaker processing job.

```json
{
  "StartAt": "SageMaker CreateProcessingJob Sync",  
  "TimeoutSeconds": 3600,  
  "States": {
    "SageMaker CreateProcessingJob Sync": {  
      "Type": "Task",  
      "Resource": "arn:aws:states:::sagemaker:createProcessingJob.sync",  
      "Parameters": {
        "AppSpecification": {
          "ImageUri": "737474898029.dkr.ecr.sa-east-1.amazonaws.com/sagemaker-scikit-
          learn:0.20.0-cpu-py3" 
        },  
        "ProcessingResources": {
          "ClusterConfig": {
            "InstanceCount": 1,  
            "InstanceType": "ml.t3.medium",  
            "VolumeSizeInGB": 10  
          }  
        },  
        "RoleArn": "arn:aws:iam::123456789012:role/SM-003-
CreateProcessingJobAPIExecutionRole",  
        "ProcessingJobName.$": "$id"  
      },  
      "Next": "ValidateOutput"  
    },  
    "ValidateOutput": {  
      "Type": "Choice",  
      "Choices": [
        {  
          "Not": {  
            "Variable": "$.ProcessingJobArn",  
            "StringEquals": ""  
          },  
          "Next": "Succeed"  
        },  
        "Default": "Fail"  
      },  
      "Succeed": {  
        "Type": "Succeed"  
      },  
      "Fail": {  
        "Type": "Fail",  
        "Error": "InvalidConnectorOutput",  
        "Cause": "Connector output is not what was expected. This could be due to a service 
        outage or a misconfigured connector."  
      }  
    }  
  }  
}
```
For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

Call Amazon EMR with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

How the Optimized Amazon EMR integration is different than the Amazon EMR AWS SDK integration

The Optimized Amazon EMR service integration has a customized set of APIs that wrap the underlying Amazon EMR APIs, described below. Because of this, it differs significantly from the Amazon EMR AWS SDK service integration. In addition, the Run a Job (.sync) (p. 292) integration pattern is supported.

To integrate AWS Step Functions with Amazon EMR, you use the provided Amazon EMR service integration APIs. The service integration APIs are similar to the corresponding Amazon EMR APIs, with some differences in the fields that are passed and in the responses that are returned.

Step Functions does not terminate an Amazon EMR cluster automatically if execution is stopped. If your state machine stops before your Amazon EMR cluster has terminated, your cluster may continue running indefinitely, and can accrue additional charges. To avoid this, ensure that any Amazon EMR cluster you create is terminated properly. For more information, see:

- Control Cluster Termination in the Amazon EMR User Guide.
- The Service Integration Patterns Run a Job (.sync) (p. 292) section.

Note

As of emr-5.28.0, you can specify the parameter StepConcurrencyLevel when creating a cluster to allow multiple steps to run in parallel on a single cluster. You can use the Step Functions Map and Parallel states to submit work in parallel to the cluster.

The availability of Amazon EMR service integration is subject to the availability of Amazon EMR APIs. Please check the Amazon EMR documentation for limitations in special regions.

The following table describes the differences between each service integration API and its corresponding Amazon EMR API.

Amazon EMR Service Integration APIs and Corresponding Amazon EMR APIs

<table>
<thead>
<tr>
<th>Amazon EMR Service Integration API</th>
<th>Corresponding EMR API</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>createCluster</td>
<td>runJobFlow</td>
<td>createCluster uses the same request syntax as runJobFlow, except for the following:</td>
</tr>
<tr>
<td>Creates and starts running a cluster (job flow).</td>
<td></td>
<td>- The field Instances.KeepJobFlowAliveWhenNoSteps is mandatory, and must have the Boolean value TRUE.</td>
</tr>
<tr>
<td>Amazon EMR is linked directly to a unique type of IAM role known as a service-linked</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Amazon EMR Service Integration API

<table>
<thead>
<tr>
<th>Amazon EMR Service Integration API</th>
<th>Corresponding EMR API</th>
<th>Differences</th>
</tr>
</thead>
</table>
| role. For `createCluster` and `createCluster.sync` to work, you must have configured the necessary permissions to create the service-linked role `AWSServiceRoleForEMRCleanup`. For more information about this, including a statement you can add to your IAM permissions policy, see Using the Service-Linked Role for Amazon EMR. | | • The field `Steps` is not allowed.  
• The field `Instances.InstanceFleets[index].Name` should be provided and must be unique if the optional `modifyInstanceFleetByName` connector API is used.  
• The field `Instances.InstanceGroups[index].Name` should be provided and must be unique if the optional `modifyInstanceGroupByName` API is used. |
| **createCluster.sync**  
Creates and starts running a cluster (job flow). | **runJobFlow**  
The same as `createCluster`, but waits for the cluster to reach the `WAITING` state. | |
| **setClusterTerminationProtection**  
Locks a cluster (job flow) so the EC2 instances in the cluster cannot be terminated by user intervention, an API call, or a job-flow error. | **setTerminationProtection**  
Request uses this: | |
| | | {  
"ClusterId": "string"  
}  
Amazon EMR uses this: | |
| | | {  
"JobFlowIds": ["string"]  
} |
<table>
<thead>
<tr>
<th>Amazon EMR Service Integration API</th>
<th>Corresponding EMR API</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>terminateCluster</code></td>
<td><code>terminateJobFlows</code></td>
<td>Request uses this:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;ClusterId&quot;: &quot;string&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Amazon EMR uses this:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;JobFlowIds&quot;: [&quot;string&quot;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td><code>terminateCluster.sync</code></td>
<td><code>terminateJobFlows</code></td>
<td>The same as</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>terminateCluster</code>, but waits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for the cluster to terminate.</td>
</tr>
<tr>
<td><code>addStep</code></td>
<td><code>addJobFlowSteps</code></td>
<td>Request uses the key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;ClusterId&quot;. Amazon EMR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>uses &quot;JobFlowId&quot;. Request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>uses a single step.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;Step&quot;: &lt;&quot;StepConfig object&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Amazon EMR uses this:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;Steps&quot;: [&lt;StepConfig objects&gt;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Response is this:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;StepId&quot;: &quot;string&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Amazon EMR returns this:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;StepIds&quot;: [&lt;strings&gt;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td><code>addStep.sync</code></td>
<td><code>addJobFlowSteps</code></td>
<td>The same as <code>addStep</code>, but waits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for the step to complete.</td>
</tr>
<tr>
<td>Amazon EMR Service Integration API</td>
<td>Corresponding EMR API</td>
<td>Differences</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>cancelStep</code></td>
<td><code>cancelSteps</code></td>
<td>Request uses this:</td>
</tr>
</tbody>
</table>
|                                   |                       | ```
|                                   | {  
|                                   |   "StepId": "string"
|                                   | }
|                                   |                       | Amazon EMR uses this: |
|                                   |                       | ```
|                                   | {  
|                                   |   "StepIds": ["strings"]
|                                   | }
|                                   |                       | Response is this: |
|                                   |                       | ```
|                                   | {  
|                                   |   "CancelStepsInfo": 
|                                   |   <CancelStepsInfo object>
|                                   | }
|                                   |                       | Amazon EMR uses this: |
|                                   |                       | ```
|                                   | {  
|                                   |   "CancelStepsInfoList": 
|                                   |   [<CancelStepsInfo objects>]
|                                   | }

<table>
<thead>
<tr>
<th><code>modifyInstanceFleetByName</code></th>
<th><code>modifyInstanceFleet</code></th>
<th>Request is the same as for <code>modifyInstanceFleet</code>, except for the following:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>modifyInstanceFleetByName</code></td>
<td><code>modifyInstanceFleet</code></td>
<td>• The field <code>Instance.InstanceFleetId</code> is not allowed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• At runtime the <code>InstanceFleetId</code> is determined automatically by the service integration by calling <code>ListInstanceFleets</code> and parsing the result.</td>
</tr>
</tbody>
</table>

`cancelStep` Cancels a pending step in a running cluster.

`modifyInstanceFleetByName` Modifies the target On-Demand and target Spot capacities for the instance fleet with the specified `InstanceFleetName`. 
**Amazon EMR Service Integration API**

<table>
<thead>
<tr>
<th>correspond</th>
<th>request is this:</th>
</tr>
</thead>
</table>
| modifyInstanceGroupByName | {  
| modif... |      |
| and config... | of an instance group.  
| modifyInstanceGroups |     |
| Amazon EMR uses a list: | {  
| Within the InstanceGroupModifyConfig object, the field InstanceGroupId is not allowed.  
| A new field, InstanceGroupName, has been added. At runtime the InstanceGroupId is determined automatically by the service integration by calling ListInstanceGroups and parsing the result. |  

```
The following includes a Task state that creates a cluster.

"Create_Cluster": {
  "Type": "Task",
  "Resource": "arn:aws:states:::elasticmapreduce:createCluster.sync",
  "Parameters": {
    "Name": "MyWorkflowCluster",
    "VisibleToAllUsers": true,
    "ReleaseLabel": "emr-5.28.0",
    "Applications": [
      {"Name": "Hive"}
    ],
    "ServiceRole": "EMR_DefaultRole",
    "JobFlowRole": "EMR_EC2_DefaultRole",
    "LogUri": "s3n://aws-logs-123456789012-us-east-1/elasticmapreduce/",
    "Instances": {
      "KeepJobFlowAliveWhenNoSteps": true,
      "InstanceFleets": [
        {"InstanceFleetType": "MASTER",
        "Name": "MASTER",
```
The following includes a Task state that enables termination protection.

```
"Enable_Termination_Protection": {
  "Type": "Task",
  "Resource": "arn:aws:states:::elasticmapreduce:setClusterTerminationProtection",
  "Parameters": {
    "ClusterId.$": "$ClusterId",
    "TerminationProtected": true
  },
  "End": true
}
```

The following includes a Task state that submits a step to a cluster.

```
"Step_One": {
  "Type": "Task",
  "Resource": "arn:aws:states:::elasticmapreduce:addStep.sync",
  "Parameters": {
    "ClusterId.$": "$ClusterId",
    "Step": {
      "Name": "The first step",
      "ActionOnFailure": "CONTINUE",
      "HadoopJarStep": {
        "Jar": "command-runner.jar",
        "Args": [
          "hive-script",
          "--run-hive-script",
          "--args",
          "-f",
          "s3://<region>.elasticmapreduce.samples/cloudfront/code/Hive_CloudFront.q",
          "-d",
          "INPUT=s3://<region>.elasticmapreduce.samples",
          "-d",
          "OUTPUT=s3://<mybucket>/MyHiveQueryResults/"
        ]
      }
    }
  },
  "End": true
}
```
The following includes a Task state that cancels a step.

```
"Cancel_Step_One": {
    "Type": "Task",
    "Resource": "arn:aws:states:::elasticmapreduce:cancelStep",
    "Parameters": {
        "ClusterId.$": "$ClusterId",
        "StepId.$": "$AddStepsResult.StepId"
    },
    "End": true
}
```

The following includes a Task state that terminates a cluster.

```
"Terminate_Cluster": {
    "Type": "Task",
    "Resource": "arn:aws:states:::elasticmapreduce:terminateCluster.sync",
    "Parameters": {
        "ClusterId.$": "$ClusterId"
    },
    "End": true
}
```

The following includes a Task state that scales a cluster up or down for an instance group.

```
"ModifyInstanceGroupByName": {
    "Type": "Task",
    "Resource": "arn:aws:states:::elasticmapreduce:modifyInstanceGroupByName",
    "Parameters": {
        "ClusterId": "j-1234567890123",
        "InstanceGroupName": "MyCoreGroup",
        "InstanceGroup": {
            "InstanceCount": 8
        }
    },
    "End": true
}
```

The following includes a Task state that scales a cluster up or down for an instance fleet.

```
"ModifyInstanceFleetByName": {
    "Type": "Task",
    "Resource": "arn:aws:states:::elasticmapreduce:modifyInstanceFleetByName",
    "Parameters": {
        "ClusterId": "j-1234567890123",
        "InstanceFleetName": "MyCoreFleet",
        "InstanceFleet": {
            "TargetOnDemandCapacity": 8,
            "TargetSpotCapacity": 0
        }
    },
    "End": true
}
```

For information on how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services](p. 531).
Call Amazon EMR on EKS with AWS Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

**How the Optimized Amazon EMR on EKS integration is different than the Amazon EMR on EKS AWS SDK integration**

- The Run a Job (.sync) (p. 292) integration pattern is supported.
- There are no optimizations for the Request Response (p. 292) integration pattern.
- The Wait for a Callback with the Task Token (p. 293) integration pattern is not supported.

To integrate AWS Step Functions with Amazon EMR on EKS, use the Amazon EMR on EKS service integration APIs. The service integration APIs are the same as the corresponding Amazon EMR on EKS APIs, but not all APIs support all integration patterns, as shown in the following table.

<table>
<thead>
<tr>
<th>API</th>
<th>Request response</th>
<th>Run a job (.sync)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateVirtualCluster</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DeleteVirtualCluster</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>StartJobRun</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Supported Amazon EMR on EKS APIs:

**Note**

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 262,144 bytes of data as a UTF-8 encoded string when you send to, or receive data from, another service. See Quotas related to state machine executions (p. 487).

- CreateVirtualCluster
  - Request syntax
  - Supported parameters
  - Response syntax
- DeleteVirtualCluster
  - Request syntax
  - Supported parameters
  - Response syntax
- StartJobRun
  - Request syntax
  - Supported parameters
  - Response syntax

The following includes a Task state that creates a virtual cluster.

```
"Create_Virtual_Cluster": {
```
The following includes a Task state that submits a job to a virtual cluster and waits for it to complete.

```json
"Submit_Job": {
  "Type": "Task",
  "Parameters": {
    "Name": "MyJobName",
    "VirtualClusterId.$": "$VirtualClusterId",
    "ExecutionRoleArn": "arn:aws:iam::<accountId>:role/job-execution-role",
    "ReleaseLabel": "emr-6.2.0-latest",
    "JobDriver": {
      "SparkSubmitJobDriver": {
        "EntryPoint": "s3://<mybucket>/jobs/trip-count.py",
        "EntryPointArguments": [
          "60"
        ],
        "SparkSubmitParameters": "--conf spark.driver.cores=2 --conf spark.executor.instances=10 --conf spark.kubernetes.pyspark.pythonVersion=3 --conf spark.executor.memory=10G --conf spark.driver.memory=10G --conf spark.executor.cores=1 --conf spark.dynamicAllocation.enabled=false"
      }
    },
    "ConfigurationOverrides": {
      "ApplicationConfiguration": [
        {
          "Classification": "spark-defaults",
          "Properties": {
            "spark.executor.instances": "2",
            "spark.executor.memory": "2G"
          }
        }
      ],
      "MonitoringConfiguration": {
        "PersistentAppUI": "ENABLED",
        "CloudWatchMonitoringConfiguration": {
          "LogGroupName": "MyLogGroupName",
          "LogStreamNamePrefix": "MyLogStreamNamePrefix"
        },
        "S3MonitoringConfiguration": {
          "LogUri": "s3://<mylogsbucket>"
        }
      }
    }
  },
  "End": true
}
```
The following includes a Task state that deletes a virtual cluster and waits for the deletion to complete.

```
"Delete_Virtual_Cluster": {
  "Type": "Task",
  "Resource": "arn:aws:states:::emr-containers:deleteVirtualCluster.sync",
  "Parameters": {
    "Id.$": "$.VirtualClusterId"
  },
  "End": true
}
```

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

### Call AWS CodeBuild with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

**How the Optimized CodeBuild integration is different than the CodeBuild AWS SDK integration**

- The Run a Job (.sync) (p. 292) integration pattern is supported.
- After you call StopBuild or StopBuildBatch, the build or build batch is not immediately deletable until some internal work is completed within CodeBuild to finalize the state of the build or builds. If you attempt to use BatchDeleteBuilds or DeleteBuildBatch during this period, the build or build batch may not be deleted. The optimized service integrations for BatchDeleteBuilds and DeleteBuildBatch include an internal retry to simplify the use case of deleting immediately after stopping.

The AWS Step Functions service integration with AWS CodeBuild enables you to use Step Functions to trigger, stop, and manage builds, and to share build reports. Using Step Functions, you can design and run continuous integration pipelines for validating your software changes for applications.

Not all APIs support all integration patterns, as shown in the following table.

<table>
<thead>
<tr>
<th>API</th>
<th>Request Response</th>
<th>Run a Job (.sync)</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartBuild</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>StopBuild</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>BatchDeleteBuilds</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>BatchGetReports</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>StartBuildBatch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>StopBuildBatch</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RetryBuildBatch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DeleteBuildBatch</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Note
Parameters in Step Functions are expressed in PascalCase, even when the native service API is camelCase.

Supported CodeBuild APIs and syntax:

- **StartBuild**
  - Request syntax
  - Supported parameters:
    - **ProjectName**
    - **ArtifactsOverride**
    - **BuildspecOverride**
    - **CacheOverride**
    - **CertificateOverride**
    - **ComputeTypeOverride**
    - **EncryptionKeyOverride**
    - **EnvironmentTypeOverride**
    - **EnvironmentVariablesOverride**
    - **GitCloneDepthOverride**
    - **GitSubmodulesConfigOverride**
    - **IdempotencyToken**
    - **ImageOverride**
    - **ImagePullCredentialsTypeOverride**
    - **InsecureSslOverride**
    - **LogsConfigOverride**
    - **PrivilegedModeOverride**
    - **QueuedTimeoutInMinutesOverride**
    - **RegistryCredentialOverride**
    - **ReportBuildStatusOverride**
    - **SecondaryArtifactsOverride**
    - **SecondarySourcesOverride**
    - **SecondarySourcesVersionOverride**
    - **ServiceRoleOverride**
    - **SourceAuthOverride**
    - **SourceLocationOverride**
    - **SourceTypeOverride**
    - **SourceVersion**
    - **TimeoutInMinutesOverride**
  - Response syntax

- **StopBuild**
  - Request syntax
  - Supported parameters:
    - **Id**
  - Response syntax

- **BatchDeleteBuilds**
  - Request syntax
  - Supported parameters:
- **Ids**
- **Response syntax**
- **BatchGetReports**
  - **Request syntax**
  - **Supported parameters:**
    - ReportArns
    - **Response syntax**
- **StartBuildBatch**
  - **Request syntax**
  - **Supported parameters:**
    - ProjectName
    - ArtifactsOverride
    - BuildBatchConfigOverride
    - BuildspecOverride
    - BuildTimeoutInMinutesOverride
    - CacheOverride
    - CertificateOverride
    - ComputeTypeOverride
    - DebugSessionEnabled
    - EncryptionKeyOverride
    - EnvironmentTypeOverride
    - EnvironmentVariablesOverride
    - GitCloneDepthOverride
    - GitSubmodulesConfigOverride
    - IdempotencyToken
    - ImageOverride
    - ImagePullCredentialsTypeOverride
    - InsecureSslOverride
    - LogsConfigOverride
    - PrivilegedModeOverride
    - QueuedTimeoutInMinutesOverride
    - RegistryCredentialOverride
    - ReportBuildBatchStatusOverride
    - SecondaryArtifactsOverride
    - SecondarySourcesOverride
    - SecondarySourcesVersionOverride
    - ServiceRoleOverride
    - SourceAuthOverride
    - SourceLocationOverride
    -SourceTypeOverride
    - SourceVersion
    - **Response syntax**
- **StopBuildBatch**
  - **Request syntax**
  - **Supported parameters:**
• Id
• Response syntax
• RetryBuildBatch
• Request syntax
• Supported parameters:
  • Id
  • IdempotencyToken
  • RetryType
• Response syntax
• DeleteBuildBatch
• Request syntax
• Supported parameters:
  • Id
• Response syntax

**Note**
You can use the JSONPath recursive descent (..) operator for BatchDeleteBuilds. This returns an array, and enables you to turn the Arn field from StartBuild into a plural Ids parameter, as shown in the following example.

```json
"BatchDeleteBuilds": {
  "Type": "Task",
  "Resource": "arn:aws:states:::codebuild:batchDeleteBuilds",
  "Parameters": {
    "Ids.$": ".Build.Arn"
  },
  "Next": "MyNextState"
},
```

For information on how to configure IAM when using Step Functions with other AWS services, see [IAM Policies for integrated services (p. 531)](https://aws.amazon.com/).  

**Call Athena with Step Functions**

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

• Working with other services (p. 271)
• Pass Parameters to a Service API (p. 296)

**How the Optimized Athena integration is different than the Athena AWS SDK integration**

• The Run a Job (.sync) (p. 292) integration pattern is supported.
• There are no optimizations for the Request Response (p. 292) integration pattern.
• The Wait for a Callback with the Task Token (p. 293) integration pattern is not supported.

The AWS Step Functions service integration with Amazon Athena enables you to use Step Functions to start and stop query execution, and get query results. Using Step Functions, you can run ad-hoc or scheduled data queries, and retrieve results targeting your S3 data lakes. Athena is serverless, so there is no infrastructure to set up or manage, and you pay only for the queries you run.
To integrate AWS Step Functions with Amazon Athena, you use the provided Athena service integration APIs.

The service integration APIs are the same as the corresponding Athena APIs. Not all APIs support all integration patterns, as shown in the following table.

<table>
<thead>
<tr>
<th>API</th>
<th>Request Response</th>
<th>Run a Job (.sync)</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartQueryExecution</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>StopQueryExecution</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>GetQueryExecution</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>GetQueryResults</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Supported Amazon Athena APIs:

**Note**

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 262,144 bytes of data as a UTF-8 encoded string when you send to, or receive data from, another service. See Quotas related to state machine executions (p. 487).

- **StartQueryExecution**
  - Request syntax
  - Supported parameters:
    - ClientRequestToken
    - QueryExecutionContext
    - QueryString
    - ResultConfiguration
    - WorkGroup
  - Response syntax
- **StopQueryExecution**
  - Request syntax
  - Supported parameters:
    - QueryExecutionId
- **GetQueryExecution**
  - Request syntax
  - Supported parameters:
    - QueryExecutionId
  - Response syntax
- **GetQueryResults**
  - Request syntax
  - Supported parameters:
    - MaxResults
    - NextToken
    - QueryExecutionId
  - Response syntax

The following includes a Task state that starts an Athena query.
"Start an Athena query": {
  "Type": "Task",
  "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
  "Parameters": {
    "QueryString": "SELECT * FROM "myDatabase"."myTable" limit 1",
    "WorkGroup": "primary",
    "ResultConfiguration": {
      "OutputLocation": "s3://athenaQueryResult"
    }
  },
  "Next": "Get results of the query"
}

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

Call Amazon EKS with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

How the Optimized Amazon EKS integration is different than the Amazon EKS AWS SDK integration

- The Run a Job (.sync) (p. 292) integration pattern is supported.
- There are no optimizations for the Request Response (p. 292) integration pattern.
- The Wait for a Callback with the Task Token (p. 293) integration pattern is not supported.

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

Step Functions provides two types of service integration APIs for integrating with Amazon Elastic Kubernetes Service. One lets you use the Amazon EKS APIs to create and manage an Amazon EKS cluster. The other lets you interact with your cluster using the Kubernetes API and run jobs as part of your application's workflow. You can use the Kubernetes API integrations with Amazon EKS clusters created using Step Functions, with Amazon EKS clusters created by the eksctl tool or the Amazon EKS console, or similar methods. The Step Functions EKS integration supports only Kubernetes APIs with public endpoint access. For more information, see Creating an Amazon EKS cluster in the Amazon EKS User Guide.

Step Functions does not terminate an Amazon EKS cluster automatically if execution is stopped. If your state machine stops before your Amazon EKS cluster has terminated, your cluster may continue running indefinitely, and can accrue additional charges. To avoid this, ensure that any Amazon EKS cluster you create is terminated properly. For more information, see:

- Deleting a cluster in the Amazon EKS User Guide.
- Run a Job (.sync) (p. 292) in Service Integration Patterns.

Note
There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 262,144 bytes of data as a UTF-8 encoded string when you send to, or receive data from, another service. See Quotas related to state machine executions (p. 487).
Kubernetes API integrations

Step Functions supports the following Kubernetes APIs:

**RunJob**

The `eks:runJob` service integration allows you to run a job on your Amazon EKS cluster. The `eks:runJob.sync` variant allows you to wait for the job to complete, and, optionally retrieve logs.

Your Kubernetes API server must grant permissions to the IAM role used by your state machine. For more information, see Permissions (p. 342).

For the **Run a Job** (.sync) pattern, the status of the job is determined by polling. Step Functions initially polls at a rate of approximately 1 poll per minute. This rate eventually slows to approximately 1 poll every 5 minutes. If you require more frequent polling, or require more control over the polling strategy, you can use the `eks:call` integration to query the status of the job.

The `eks:runJob` integration is specific to `batch/v1` Kubernetes Jobs. For more information, see Jobs in the Kubernetes documentation. If you want to manage other Kubernetes resources, including custom resources, use the `eks:call` service integration. You can use Step Functions to build polling loops, as demonstrated in the the section called “Poll for Job Status (Lambda, AWS Batch)” (p. 364) sample project.

Supported parameters include:

- **ClusterName**: The name of the Amazon EKS cluster you want to call.
  - Type: String
  - Required: yes

- **CertificateAuthority**: The Base64-encoded certificate data required to communicate with your cluster. You can obtain this value from the Amazon EKS console or by using the Amazon EKS DescribeCluster API.
  - Type: String
  - Required: yes

- **Endpoint**: The endpoint URL for your Kubernetes API server. You can obtain this value from the Amazon EKS console or by using the Amazon EKS DescribeCluster API.
  - Type: String
  - Required: yes

- **Namespace**: The namespace in which to run the job. If not provided, the namespace `default` is used.
  - Type: String
  - Required: no

- **Job**: The definition of the Kubernetes Job. See Jobs in the Kubernetes documentation.
  - Type: JSON or String
  - Required: yes

- **LogOptions**: A set of options to control the optional retrieval of logs. Only applicable if the Run a Job (.sync) service integration pattern is used to wait for the completion of the job.
  - Type: JSON
  - Required: no

Logs are included in the response under the key `logs`. There may be multiple pods within the job, each with multiple containers.

```json
{
  ...
  "logs": {
    "pods": {
```
• Log retrieval is performed on a best-effort basis. If there is an error retrieving a log, in place of the log field there will be the fields error and cause.

• LogOptions.RetrieveLogs: Enable log retrieval after the job completes. By default, logs are not retrieved.
  • Type: Boolean
  • Required: no

• LogOptions.RawLogs: If RawLogs is set to true, logs will be returned as raw strings without attempting to parse them into JSON. By default, logs are deserialized into JSON if possible. In some cases such parsing can introduce unwanted changes, such as limiting the precision of numbers containing many digits.
  • Type: Boolean
  • Required: no

• LogOptions.LogParameters: The Kubernetes API's Read Log API supports query parameters to control log retrieval. For example, you can use tailLines or limitBytes to limit the size of retrieved logs and remain within the Step Functions data size quota. For more information, see the Read Log section of the Kubernetes API Reference.
  • Type: Map of String to List of Strings
  • Required: no
  • Example:

    "LogParameters": {
      "tailLines": [ "6" ]
    }

The following example includes a Task state that runs a job, waits for it to complete, then retrieves the job's logs:

```json
{
  "StartAt": "Run a job on EKS",
  "States": {
    "Run a job on EKS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:runJob.sync",
      "Parameters": {
        "ClusterName": "MyCluster",
        "CertificateAuthority": "ANPAJ2UCCREDFPCEEXAMPLE",
        "Endpoint": "https://AKIAIOSFODNN7EXAMPLE.yl4.us-east-1.eks.amazonaws.com",
        "LogOptions": {
          "RetrieveLogs": true
        },
        "Job": {
          "apiVersion": "batch/v1",
          "kind": "Job",
          "metadata": {
            "name": "example-job"
          }
        }
      }
    }  
  }
}  
```
Call

The eks:call service integration allows you to use the Kubernetes API to read and write Kubernetes resource objects via a Kubernetes API endpoint.

Your Kubernetes API server must grant permissions to the IAM role used by your state machine. For more information, see Permissions (p. 342).

For more information about the available operations, see the Kubernetes API Reference.

Supported parameters for Call include:

- **ClusterName**: The name of the Amazon EKS cluster you want to call.
  - **Type**: String
  - **Required**: Yes
- **CertificateAuthority**: The Base64-encoded certificate data required to communicate with your cluster. You can obtain this value from the Amazon EKS console or by using the Amazon EKS DescribeCluster API.
  - **Type**: String
  - **Required**: Yes
- **Endpoint**: The endpoint URL for your Kubernetes API server. You can find this value on the Amazon EKS console or by using Amazon EKS' DescribeCluster API.
  - **Type**: String
  - **Required**: Yes
- **Method**: The HTTP method of your request. One of: GET, POST, PUT, DELETE, HEAD, or PATCH.
  - **Type**: String
  - **Required**: Yes
- **Path**: The HTTP path of the Kubernetes REST API operation.
• Type: String
• Required: Yes

**QueryParameters**: The HTTP query parameters of the Kubernetes REST API operation.

• Type: Map of String to List of Strings
• Required: No
• Example:

```json
"QueryParameters": {
    "labelSelector": [ "job-name=example-job" ]
}
```

• RequestBody: The HTTP message body of the Kubernetes REST API operation.

• Type: JSON or String
• Required: No

The following includes a **Task** state that uses `eks:call` to list the pods belonging to the job `example-job`.

```json
{
    "StartAt": "Call EKS",
    "States": {
        "Call EKS": {
            "Type": "Task",
            "Resource": "arn:aws:states:::eks:call",
            "Parameters": {
                "ClusterName": "MyCluster",
                "CertificateAuthority": "ANPAJ2UCER6DPCEXAMPLE",
                "Endpoint": "https://444455566666.yl4.us-east-1.eks.amazonaws.com",
                "Method": "GET",
                "Path": "/api/v1/namespaces/default/pods",
                "QueryParameters": {
                    "labelSelector": [ "job-name=example-job" ]
                }
            },
            "End": true
        }
    }
}
```

The following includes a **Task** state that uses `eks:call` to delete the job `example-job`, and sets the **propagationPolicy** to ensure the job's pods are also deleted.

```json
{
    "StartAt": "Call EKS",
    "States": {
        "Call EKS": {
            "Type": "Task",
            "Resource": "arn:aws:states:::eks:call",
            "Parameters": {
                "ClusterName": "MyCluster",
                "CertificateAuthority": "ANPAJ2UCER6DPCEXAMPLE",
                "Endpoint": "https://444455566666.yl4.us-east-1.eks.amazonaws.com",
                "Method": "DELETE",
                "Path": "/apis/batch/v1/namespaces/default/jobs/example-job",
                "QueryParameters": {
                    "propagationPolicy": [ "Foreground"
                }
            },
            "End": true
        }
    }
}
```
Supported Amazon EKS APIs

Supported Amazon EKS APIs and syntax include:

- **CreateCluster**
  - Request syntax
  - Response syntax

  When an Amazon EKS cluster is created using the `eks:createCluster` service integration, the IAM role is added to the Kubernetes RBAC authorization table as the administrator (with system:masters permissions). Initially, only that IAM entity can make calls to the Kubernetes API server. For more information, see:
  - Managing users or IAM roles for your cluster in the *Amazon EKS User Guide*
  - The Permissions (p. 342) section

  Amazon EKS uses service-linked roles which contain the permissions Amazon EKS requires to call other services on your behalf. If these service-linked roles do not exist in your account already, you must add the `iam:CreateServiceLinkedRole` permission to the IAM role used by Step Functions. For more information, see Using Service-Linked Roles in the *Amazon EKS User Guide*.

  The IAM role used by Step Functions must have `iam:PassRole` permissions to pass the cluster IAM role to Amazon EKS. For more information, see *Amazon EKS cluster IAM role* in the *Amazon EKS User Guide*.

- **DeleteCluster**
  - Request syntax
  - Response syntax

  You must delete any Fargate profiles or node groups before deleting a cluster.

- **CreateFargateProfile**
  - Request syntax
  - Response syntax

  Amazon EKS on Fargate may not be available in all regions. For information on region availability, see the section on Fargate in the *Amazon EKS User Guide*.

  The IAM role used by Step Functions must have `iam:PassRole` permissions to pass the pod execution IAM role to Amazon EKS. For more information, see *Pod execution role* in the *Amazon EKS User Guide*.

- **DeleteFargateProfile**
  - Request syntax
  - Response syntax

- **CreateNodegroup**
Amazon EKS uses service-linked role whichs contain the permissions Amazon EKS requires to call other services on your behalf. If these service-linked roles do not exist in your account already, you must add the `iam:CreateServiceLinkedRole` permission to the IAM role used by Step Functions. For more information, see Using Service-Linked Roles in the Amazon EKS User Guide.

The IAM role used by Step Functions must have `iam:PassRole` permissions to pass the node IAM role to Amazon EKS. For more information, see Using Service-Linked Roles in the Amazon EKS User Guide.

The following includes a Task that creates an Amazon EKS cluster.

```
{
  "StartAt": "CreateCluster.sync",
  "States": {
    "CreateCluster.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:createCluster.sync",
      "Parameters": {
        "Name": "MyCluster",
        "ResourcesVpcConfig": {
          "SubnetIds": [
            "subnet-053e7c47012341234",
            "subnet-027c6a6b12341234"
          ],
          "RoleArn": "arn:aws:iam::123456789012:role/MyEKSClusterRole"
        }
      }
    },
    "End": true
  }
}
```

The following includes a Task state that deletes an Amazon EKS cluster.

```
{
  "StartAt": "DeleteCluster.sync",
  "States": {
    "DeleteCluster.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:deleteCluster.sync",
      "Parameters": {
        "Name": "MyCluster"
      },
      "End": true
    }
  }
}
```

The following includes a Task state that creates a Fargate profile.

```
{
  "StartAt": "CreateFargateProfile.sync",
  "States": {
    "CreateFargateProfile.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:createFargateProfile.sync",
      "Parameters": {
        "Name": "MyFargateProfile"
      }
    },
    "End": true
  }
}
```
The following includes a Task state that deletes a Fargate profile.

```
"CreateFargateProfile.sync": {
  "Type": "Task",
  "Resource": "arn:aws:states:::eks:createFargateProfile.sync",
  "Parameters": {
    "ClusterName": "MyCluster",
    "FargateProfileName": "MyFargateProfile",
    "PodExecutionRoleArn": "arn:aws:iam::123456789012:role/MyFargatePodExecutionRole",
    "Selectors": [{
      "Namespace": "my-namespace",
      "Labels": { "my-label": "my-value" }
    }]
  },
  "End": true
}
```

The following includes a Task state that creates a node group.

```
"DeleteFargateProfile.sync": {
  "StartAt": "DeleteFargateProfile.sync",
  "States": {
    "DeleteFargateProfile.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:deleteFargateProfile.sync",
      "Parameters": {
        "ClusterName": "MyCluster",
        "FargateProfileName": "MyFargateProfile"
      },
      "End": true
    }
  }
}
```

The following includes a Task state that deletes a node group.

```
"CreateNodegroup.sync": {
  "StartAt": "CreateNodegroup.sync",
  "States": {
    "CreateNodegroup.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:createNodegroup.sync",
      "Parameters": {
        "ClusterName": "MyCluster",
        "NodegroupName": "MyNodegroup",
        "NodeRole": "arn:aws:iam::123456789012:role/MyNodeInstanceRole",
        "Subnets": ["subnet-09fb51df01234", "subnet-027cfe4b1234"]
      },
      "End": true
    }
  }
}
```

The following includes a Task state that deletes a node group.

```
"DeleteNodegroup.sync": {
  "StartAt": "DeleteNodegroup.sync",
  "States": {
    "DeleteNodegroup.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:deleteNodegroup.sync",
      "Parameters": {
```
Permissions

When an Amazon EKS cluster is created using the `eks:createCluster` service integration, the IAM role is added to the Kubernetes RBAC authorization table as the administrator, with `system:masters` permissions. Initially, only that IAM entity can make calls to the Kubernetes API server. For example, you will not be able to use `kubectl` to interact with your Kubernetes API server, unless you assume the same role as your Step Functions state machine, or if you configure Kubernetes to grant permissions to additional IAM entities. For more information, see Managing users or IAM roles for your cluster in the Amazon EKS User Guide.

You can add permission for additional IAM entities, such as users or roles, by adding them to the `aws-auth` ConfigMap in the kube-system namespace. If you are creating your cluster from Step Functions, use the `eks:call` service integration.

The following includes a Task state that creates an `aws-auth` ConfigMap and grants `system:masters` permission to the IAM user `arn:aws:iam::123456789012:user/my-user` and the IAM role `arn:aws:iam::123456789012:role/my-role`.

```json
{
    "StartAt": "Add authorized user",
    "States": {
        "Add authorized user": {
            "Type": "Task",
            "Resource": "arn:aws:states:::eks:call",
            "Parameters": {
                "ClusterName": "MyCluster",
                "CertificateAuthority": "LS0tLS1CRUd...UtLS0tLQo=",
                "Endpoint": "https://444455556666.yl4.us-east-1.eks.amazonaws.com",
                "Method": "POST",
                "Path": "/api/v1/namespaces/kube-system/configmaps",
                "RequestBody": {
                    "apiVersion": "v1",
                    "kind": "ConfigMap",
                    "metadata": {
                        "name": "aws-auth",
                        "namespace": "kube-system"
                    },
                    "data": {
                        "mapUsers": "[{ "userarn": "arn:aws:iam::123456789012:user/my-user",
                        "username": "my-user", "groups": [ "system:masters" ] } ]",
                        "mapRoles": "[{ "rolearn": "arn:aws:iam::123456789012:role/my-role",
                        "username": "my-role", "groups": [ "system:masters" ] } ]"
                    }
                }
            }
        }
    },
    "End": true
}
```

**Note**
You may see the ARN for an IAM role displayed in a format that includes the path `/service-role/`, such as `arn:aws:iam::123456789012:role/service-role/my-role`. This `service-role` path token should not be included when listing the role in `aws-auth`.  

342
When your cluster is first created the `aws-auth` ConfigMap will not exist, but will be added automatically if you create a Fargate profile. You can retrieve the current value of `aws-auth`, add the additional permissions, and PUT a new version. It is usually easier to create `aws-auth` before the Fargate profile.

If your cluster was created outside of Step Functions, you can configure kubectl to communicate with your Kubernetes API server. Then, create a new `aws-auth` ConfigMap using `kubectl apply -f aws-auth.yaml` or edit one that already exists using `kubectl edit -n kube-system configmap/aws-auth`. For more information, see:

- Create a kubeconfig for Amazon EKS in the Amazon EKS User Guide.
- Managing users or IAM roles for your cluster in the Amazon EKS User Guide.

If your IAM role does not have sufficient permissions in Kubernetes, the `eks:call` or `eks:runJob` service integrations will fail with the following error:

```
Error:
EKS.401

Cause:
{
  "ResponseBody": {
    "kind": "Status",
    "apiVersion": "v1",
    "metadata": {},
    "status": "Failure",
    "message": "Unauthorized",
    "reason": "Unauthorized",
    "code": 401
  },
  "StatusCode": 401,
  "StatusText": "Unauthorized"
}
```

### Call API Gateway with Step Functions

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

#### How the Optimized API Gateway integration is different than the API Gateway AWS SDK integration

- `apigateway:invoke:` has no equivalent in the AWS SDK service integration. Instead, the Optimized API Gateway service calls your API Gateway endpoint directly.

You use Amazon API Gateway to create, publish, maintain, and monitor HTTP and REST APIs. To integrate with API Gateway, you define a Task state in Step Functions that directly calls an API Gateway HTTP or API Gateway REST endpoint, without writing code or relying on other infrastructure. A Task state definition includes all the necessary information for the API call. You can also select different authorization methods.
Note
Step Functions supports the ability to call HTTP endpoints through API Gateway, but does not currently support the ability to call generic HTTP endpoints.

API Gateway feature support

The Step Functions API Gateway integration supports some, but not all API Gateway features. For a more detailed list of supported features, see the following.

- Supported by both the Step Functions API Gateway REST API and API Gateway HTTP API integrations:
  - **Authorizers**: IAM (using Signature Version 4), No Auth, Lambda Authorizers (request-parameter based and token-based with custom header)
  - **API types**: Regional
  - **API management**: API Gateway API domain names, API stage, Path, Query Parameters, Request Body

- Supported by the Step Functions API Gateway HTTP API integration but not the Step Functions API Gateway REST API integration:
  - Edge-optimized APIs

- Unsupported by the Step Functions API Gateway integration:
  - **Authorizers**: Amazon Cognito, Native Open ID Connect / OAuth 2.0, Authorization header for token-based Lambda authorizers
  - **API types**: Private
  - **API management**: Custom domain names

For more information about API Gateway and its HTTP and REST APIs, see the following.

- The [Amazon API Gateway concepts page](#).
- Choosing between HTTP APIs and REST APIs in the API Gateway developer guide.

Request format

When you create your Task state definition, Step Functions validates the parameters, builds the necessary URL to perform the call, then calls the API. The response includes the HTTP status code, headers and response body. The request format has both required and optional parameters.

Required request parameters

- **ApiEndpoint**
  - **Type**: String
  - The hostname of an API Gateway URL. The format is `<API ID>.execute-api.<region>.amazonaws.com`.

  The API ID can only contain a combination of the following alphanumeric characters: 0123456789abcdefghijklmnopqrstuvwxyz

  Method
  - **Type**: Enum
  - The HTTP method, which must be one of the following:
    - GET
    - POST
    - PUT
    - DELETE
Optional request parameters

- **Headers**
  - **Type:** JSON
  - HTTP headers allow a list of values associated with the same key.
- **Stage**
  - **Type:** String
  - The name of the stage where the API is deployed to in API Gateway. It's optional for any HTTP API that uses the default stage.
- **Path**
  - **Type:** String
  - Path parameters that are appended after the API endpoint.
- **QueryParameters**
  - **Type:** JSON
  - Query strings allow a list of values associated with the same key.
- **RequestBody**
  - **Type:** JSON or String
  - The HTTP Request body. Its type can be either a JSON object or String. RequestBody is only supported for PATCH, POST, and PUT HTTP methods.
- **AllowNullValues**
  - **Type:** BOOLEAN
  - Setting AllowNullValues to true will allow you to pass null values such as the following:

  ```json
  { 
    "NewPet": { 
      "type": "turtle", 
      "price": 123, 
      "name": null 
    } 
  }
  ```

- **AuthType**
  - **Type:** JSON
  - The authentication method. The default method is NO_AUTH. The allowed values are:
    - NO_AUTH
    - IAM_ROLE
    - RESOURCE_POLICY

  See **Authentication and authorization** for more information.

**Note**

For security considerations, the following HTTP header keys are not currently permitted:

- Anything prefixed with X-Forwarded, X-Amz or X-Amzn.
- Authorization
- Connection
The following code example shows how to invoke API Gateway using Step Functions.

```json
{
  "Type": "Task",
  "Resource": "arn:aws:states:::apigateway:invoke",
  "Parameters": {
    "ApiEndpoint": "example.execute-api.us-east-1.amazonaws.com",
    "Method": "GET",
    "Headers": {
      "key": ["value1", "value2"]
    },
    "Stage": "prod",
    "Path": "bills",
    "QueryParameters": {
      "billId": ["123456"]
    },
    "RequestBody": {},
    "AuthType": "NO_AUTH"
  }
}
```

**Authentication and authorization**

You can use the following authentication methods:

- **No authorization**: Call the API directly with no authorization method.
- **IAM role**: With this method, Step Functions assumes the role of the state machine, signs the request with Signature Version 4 (SigV4), then calls the API.
- **Resource policy**: Step Functions authenticates the request, and then calls the API. You must attach a resource policy to the API which specifies the following:
  1. The state machine that will invoke API Gateway.

    **Important**
    You must specify your state machine to limit access to it. If you do not, then any state machine that authenticates its API Gateway request with Resource policy authentication to your API will be granted access.

  2. That Step Functions is the service calling API Gateway: "Service": "states.amazonaws.com".
  3. The resource you want to access, including:
    - The **region**.
- The `account-id` in the specified region.
- The `api-id`.
- The `stage-name`.
- The `HTTP-VERB` (method).
- The `resource-path-specifier`.

For an example resource policy, see IAM policies for Step Functions and API Gateway (p. 573).

For more information on the resource format, see Resource format of permissions for executing API in API Gateway in the API Gateway Developer Guide.

**Note**
Resource policies are only supported for the REST API.

### Service integration patterns

The API Gateway integration supports two service integration patterns:

- **Request Response** (p. 292), which is the default integration pattern. It lets Step Functions progress to the next step immediately after it receives an HTTP response.
- **Wait for a Callback with the Task Token** (p. 293) (`.waitForTaskToken`), which waits until a task token is returned with a payload. To use the `.waitForTaskToken` pattern, append `.waitForTaskToken` to the end of the `Resource` field of your task definition as shown in the following example:

```json
{
  "Type": "Task",
  "Resource": "arn:aws:states:::apigateway:invoke.waitForTaskToken",
  "Parameters": {
    "ApiEndpoint": "example.execute-api.us-east-1.amazonaws.com",
    "Method": "POST",
    "Headers": {
      "TaskToken.$": "States.Array($$.Task.Token)"
    },
    "Stage": "prod",
    "Path": "bills/add",
    "QueryParameters": {},
    "RequestBody": {
      "billId": "my-new-bill"
    },
    "AuthType": "IAM_ROLE"
  }
}
```

### Output format

The following output parameters are provided:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResponseBody</td>
<td>JSON or String</td>
<td>The response body of the API call.</td>
</tr>
<tr>
<td>Headers</td>
<td>JSON</td>
<td>The response headers.</td>
</tr>
<tr>
<td>StatusCode</td>
<td>Integer</td>
<td>The HTTP status code of the response.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>StatusText</td>
<td>String</td>
<td>The status text of the response.</td>
</tr>
</tbody>
</table>

An example response:

```json
{
  "ResponseBody": {
    "myBills": [],
  },
  "Headers": {
    "key": ["value1", "value2"]
  },
  "StatusCode": 200,
  "StatusText": "OK"
}
```

**Error handling**

When an error occurs, an error and cause is returned as follows:

- If the HTTP status code is available, then the error will be returned in the format `ApiGateway.<HTTP Status Code>`.
- If the HTTP status code is not available, then the error will be returned in the format `ApiGateway.<Exception>`.

In both cases, the cause is returned as a string.

The following example shows a response where an error has occurred:

```json
{
  "error": "ApiGateway.403",
  "cause": "{"message":"Missing Authentication Token"}"
}
```

**Note**

A status code of 2XX indicates success, and no error will be returned. All other status codes or thrown exceptions will result in an error.

For more information see:

- Amazon API Gateway concepts in the API Gateway Developer Guide.
- IAM policies for Step Functions and Amazon API Gateway (p. 573)
- A sample project that shows how to Make a call to API Gateway (p. 444)

Amazon API Gateway concepts in the API Gateway Developer Guide.

**Manage AWS Glue DataBrew Jobs with Step Functions**

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:
You can use the DataBrew integration to add data cleaning and data normalization steps into your analytics and machine learning workflows.

Supported DataBrew API:

- **StartJobRun**

The following includes a Task state that starts a request-response DataBrew job.

```
"DataBrew StartJobRun": {
  "Type": "Task",
  "Resource": "arn:aws:states:::databrew:startJobRun",
  "Parameters": {
    "Name": "sample-proj-job-1"
  },
  "Next": "NEXT_STATE"
},
```

The following includes a Task state that starts a sync DataBrew job.

```
"DataBrew StartJobRun": {
  "Type": "Task",
  "Resource": "arn:aws:states:::databrew:startJobRun.sync",
  "Parameters": {
    "Name": "sample-proj-job-1"
  },
  "Next": "NEXT_STATE"
},
```

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

**Call EventBridge with Step Functions**

Step Functions can control certain AWS services directly from the Amazon States Language. For more information about working with AWS Step Functions and its integrations, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

**How the Optimized EventBridge integration is different than the EventBridge AWS SDK integration**

- The execution ARN and the state machine ARN are automatically appended to the Resources field of each PutEventsRequestEntry.
- If the response from PutEvents contains a non-zero FailedEntryCount then the Task state fails with the error EventBridge.FailedEntry.

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
Step Functions provides a service integration API for integrating with Amazon EventBridge. This lets you build event-driven applications by sending custom events directly from Step Functions workflows.

To use the `PutEvents` API, you will need to create an EventBridge rule in your account that matches the specific pattern of the events you will send. For example, you could:

- Create a Lambda function in your account that receives and prints an event that matches an EventBridge rule.
- Create an EventBridge rule in your account on the default event bus that matches a specific event pattern and targets the Lambda function.

For more information, see:

- Wait for a Callback with the Task Token (p. 293) in Service Integration Patterns.

**Note**

There is a quota for the maximum input or result data size for a task in Step Functions. This restricts you to 262,144 bytes of data as a UTF-8 encoded string when you send to, or receive data from, another service. See Quotas related to state machine executions (p. 487).

**Supported EventBridge API**

Supported EventBridge API and syntax include:

- **PutEvents**
  - Request syntax
  - Supported parameter:
    - Entries
  - Response syntax

The following includes a Task that sends a custom event:

```json
{
  "Type": "Task",
  "Resource": "arn:aws:states:::events:putEvents",
  "Parameters": {
    "Entries": [
      {
        "Detail": {
          "Message": "MyMessage"
        },
        "DetailType": "MyDetailType",
        "EventBusName": "MyEventBus",
        "Source": "my.source"
      }
    ]
  },
  "End": true
}
```

**Error handling**

The `PutEvents` API accepts an array of entries as input, then returns an array of result entries. As long as the `PutEvents` action was successful, `PutEvents` will return an HTTP 200 response, even if one or more entries failed. `PutEvents` returns the number of failed entries in the `FailedEntryCount` field.
Step Functions checks whether the FailedEntryCount is greater than zero. If it is greater than zero, Step Functions fails the state with the error EventBridge.FailedEntry. This lets you use the built-in error handling of Step Functions on task states to catch or retry when there are failed entries, rather than needing to use an additional state to analyze the FailedEntryCount from the response.

**Note**
If you have implemented idempotency and can safely retry on all entries, you can use Step Functions' retry logic. Step Functions does not remove successful entries from the PutEvents input array before retrying. Instead, it retries with the original array of entries.

**Manage AWS Step Functions Executions as an Integrated Service**

Step Functions integrates with its own API as a service integration. This allows Step Functions to start a new execution of a state machine directly from the task state of a running execution. When building new workflows, use nested workflow executions (p. 73) to reduce the complexity of your main workflows and to reuse common processes.

**How the Optimized Step Functions integration is different than the Step Functions AWS SDK integration**

- The Run a Job (.sync) (p. 292) integration pattern is available.

Note that there are no optimizations for the Request Response (p. 292) or Wait for a Callback with the Task Token (p. 293) integration patterns.

For more information, see the following:

- Start Executions from a Task (p. 73)
- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)

**Supported Step Functions APIs and syntax:**

- StartExecution
- Request Syntax
- Supported Parameters
  - Input
  - Name
  - StateMachineArn
- Response syntax

The following includes a Task state that starts an execution of another state machine and waits for it to complete.

```json
{
  "Type": "Task",
  "Parameters": {
    "Input": {
      "Comment": "Hello world!"
    },
    "Name": "ExecutionName"
  }
}
```
The following includes a Task state that starts an execution of another state machine.

```json
{
  "Type": "Task",
  "Resource": "arn:aws:states:::states:startExecution",
  "Parameters":{
    "Input":{
      "Comment": "Hello world!"
    },
    "Name": "ExecutionName"
  },
  "End":true
}
```

The following includes a Task state that implements the callback (p. 293) service integration pattern.

```json
{
  "Type": "Task",
  "Resource": "arn:aws:states:::states:startExecution.waitForTaskToken",
  "Parameters":{
    "Input":{
      "Comment": "Hello world!",
      "token.$": "$$\.Task\.Token"
    },
    "Name": "ExecutionName"
  },
  "End":true
}
```

To associate a nested workflow execution with the parent execution that started it, pass a specially named parameter that includes the execution ID pulled from the context object (p. 70). When starting a nested execution, use a parameter named `AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID`. Pass the execution ID by appending `.$$` to the parameter name, and referencing the ID in the context object with `$$\.Execution\.Id`. For more information, see Accessing the Context Object (p. 71).

```json
{
  "Type": "Task",
  "Resource": "arn:aws:states:::states:startExecution.sync",
  "Parameters":{
    "Input":{
      "Comment": "Hello world!",
      "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$\.Execution\.Id"
    },
    "Name": "ExecutionName"
  },
  "End":true
}
```

Nested state machines return the following:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>startExecution.sync</td>
<td>String</td>
</tr>
<tr>
<td>Resource</td>
<td>Output</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>startExecution.sync:2</td>
<td>JSON</td>
</tr>
</tbody>
</table>

Both will wait for the nested state machine to complete, but they return different Output formats. For example, if you create a Lambda function that returns the object `{ "MyKey": "MyValue" }`, you would get the following responses:

For `startExecution.sync`:

```json
{
   "<other fields>
   "Output": "{ "MyKey": "MyValue" }"
}
```

For `startExecution.sync:2`:

```json
{
   "<other fields>
   "Output": {
      "MyKey": "MyValue"
   }
}
```

For information on how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
Sample projects for Step Functions

In the AWS Step Functions console, you can choose one of the following state machine sample projects to automatically create the state machine **Code**, **Visual Workflow**, and all related AWS resources for the project.

Each of the sample projects provisions a fully functional state machine, and creates the related resources for it to run. When you create a sample project, Step Functions uses AWS CloudFormation to create the related resources referenced by the state machine.

**Topics**

- Manage a Batch Job (AWS Batch, Amazon SNS) (p. 355)
- Manage a Container Task (Amazon ECS, Amazon SNS) (p. 357)
- Transfer Data Records (Lambda, DynamoDB, Amazon SQS) (p. 361)
- Poll for Job Status (Lambda, AWS Batch) (p. 364)
- Task Timer (Lambda, Amazon SNS) (p. 367)
- Callback Pattern Example (Amazon SQS, Amazon SNS, Lambda) (p. 369)
- Manage an Amazon EMR Job (p. 372)
- Start a Workflow within a Workflow (Step Functions, Lambda) (p. 377)
- Dynamically process data with a Map state (p. 379)
- Train a Machine Learning Model (p. 384)
- Tune a Machine Learning Model (p. 388)
- Process High-Volume Messages from Amazon SQS (Express Workflows) (p. 395)
- Selective Checkpointing Example (Express Workflows) (p. 399)
- Build an AWS CodeBuild Project (CodeBuild, Amazon SNS) (p. 406)
- Preprocess data and train a machine learning model (p. 409)
- Lambda orchestration example (p. 414)
- Start an Athena query (p. 419)
- Execute multiple queries (Amazon Athena, Amazon SNS) (p. 423)
- Query large datasets (Amazon Athena, Amazon S3, AWS Glue, Amazon SNS) (p. 429)
- Keep data up to date (Amazon Athena, Amazon S3, AWS Glue) (p. 434)
- Manage an Amazon EKS cluster (p. 438)
- Make a call to API Gateway (p. 444)
- Call a microservice running on Fargate using API Gateway integration (p. 447)
- Send a custom event to EventBridge (p. 452)
- Invoke Synchronous Express Workflows (p. 455)
- Run ETL/ELT workflows using Amazon Redshift (Lambda, Amazon Redshift Data API) (p. 459)
- Use Step Functions and AWS Batch with error handling (p. 475)
- Fan out an AWS Batch job (p. 478)
- AWS Batch with Lambda (p. 481)
Manage a Batch Job (AWS Batch, Amazon SNS)

This sample project demonstrates how to submit an AWS Batch job, and then send an Amazon SNS notification based on whether that job succeeds or fails. Deploying this sample project creates an AWS Step Functions state machine, an AWS Batch job, and an Amazon SNS topic.

In this project, Step Functions uses a state machine to call the AWS Batch job synchronously. It then waits for the job to succeed or fail, and it sends an Amazon SNS topic with a message about whether the job succeeded or failed.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Manage a Batch Job.

The state machine Code and Visual Workflow are displayed.

3. Choose Next.

The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

- An AWS Batch job
- An Amazon SNS topic

4. Choose Deploy Resources.

Note
It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.

Start a New Execution

1. On the New execution page, enter an execution name (optional), and then choose Start Execution.
2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

3. Optionally, you can go to the newly created state machine on the Step Functions dashboard, and then choose New execution.

4. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

**Example State Machine Code**

The state machine in this sample project integrates with AWS Batch and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls AWS Batch and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the Resource field, and by passing Parameters to the service API.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
    "Comment": "An example of the Amazon States Language for notification on an AWS Batch job completion",
    "StartAt": "Submit Batch Job",
    "TimeoutSeconds": 3600,
    "States": {
        "Submit Batch Job": {
            "Type": "Task",
            "Resource": "arn:aws:states:::batch:submitJob.sync",
            "Parameters": {
                "JobName": "BatchJobNotification",
                "JobQueue": "arn:aws:batch:us-east-1:123456789012:job-queue/BatchJobQueue-7049d367474b4dd",
                "JobDefinition": "arn:aws:batch:us-east-1:123456789012:job-definition/BatchJobDefinition-74d55ec34c4643c:1"
            },
            "Next": "Notify Success",
            "Catch": [
                {
                    "ErrorEquals": [ "States.ALL" ],
                    "Next": "Notify Failure"
                }
            ]
        },
        "Notify Success": {
            "Type": "Task",
            "Resource": "arn:aws:states:::sns:publish",
            "Parameters": {
                "Message": "Batch job submitted through Step Functions succeeded",
                "TopicArn": "arn:aws:sns:us-east-1:123456789012:batchjobnotificationintemplate-SNSTopic-1J757CVBQ2KHMM"
            },
            "End": true
        },
        "Notify Failure": {
            "Type": "Task",
            "Resource": "arn:aws:states:::sns:publish",
            "Parameters": {
                "Message": "Batch job submitted through Step Functions failed",
                "TopicArn": "arn:aws:sns:us-east-1:123456789012:batchjobnotificationintemplate-SNSTopic-1J757CVBQ2KHMM"
            }
        }
    }
}
```
IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": ["sns:Publish"],
         "Effect": "Allow"
      },
      {
         "Action": ["batch:SubmitJob", "batch:DescribeJobs", "batch:TerminateJob"],
         "Resource": "*",
         "Effect": "Allow"
      },
      {
         "Action": ["events:PutTargets", "events:PutRule", "events:DescribeRule"],
         "Resource": ["arn:aws:events:ap-northeast-1:123456789012:rule/StepFunctionsGetEventsForBatchJobsRule"],
         "Effect": "Allow"
      }
   ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

Manage a Container Task (Amazon ECS, Amazon SNS)

This sample project demonstrates how to run an AWS Fargate task, and then send an Amazon SNS notification based on whether that job succeeds or fails. Deploying this sample project will create an AWS Step Functions state machine, a Fargate cluster, and an Amazon SNS topic.
In this project, Step Functions uses a state machine to call the Fargate task synchronously. It then waits for the task to succeed or fail, and it sends an Amazon SNS topic with a message about whether the job succeeded or failed.

**Create the State Machine and Provision Resources**

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Manage a container task**.
   
   The state machine **Code** and **Visual Workflow** are displayed.

![State Machine Diagram](image)

3. Choose **Next**.
   
   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project the resources include:
   - A Fargate cluster
   - An Amazon SNS topic
4. Choose **Deploy Resources**.
   
   **Note**
   
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

**Start a New Execution**

1. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
2. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.
   
   **Note**
   
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. Optionally, you can go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.
4. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.
Example State Machine Code

The state machine in this sample project integrates with AWS Fargate and Amazon SNS by passing parameters directly to those resources. Browse through this example state machine to see how Step Functions uses a state machine to call the Fargate task synchronously, waits for the task to succeed or fail, and sends an Amazon SNS topic with a message about whether the job succeeded or failed.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
  "Comment": "An example of the Amazon States Language for notification on an AWS Fargate task completion",
  "StartAt": "Run Fargate Task",
  "TimeoutSeconds": 3600,
  "States": {
    "Run Fargate Task": {
      "Type": "Task",
      "Resource": "arn:aws:states:::ecs:runTask.sync",
      "Parameters": {
        "LaunchType": "FARGATE",
        "TaskDefinition": "arn:aws:ecs:ap-northeast-1:123456789012:task-definition/FargateTaskNotification-ECSTaskDefinition-13YOJ8ZELY5Q:1",
        "NetworkConfiguration": {
          "AwsvpcConfiguration": {
            "Subnets": [
              "subnet-07e1ad3abcfc6758",
              "subnet-04782e7f34ae3efdb"
            ],
            "AssignPublicIp": "ENABLED"
          }
        }
      },
      "Next": "Notify Success",
      "Catch": [
        {
          "ErrorEquals": [ "States.ALL" ],
          "Next": "Notify Failure"
        }
      ]
    },
    "Notify Success": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "Message": "AWS Fargate Task started by Step Functions succeeded",
        "TopicArn": "arn:aws:sns:ap-northeast-1:123456789012:FargateTaskNotification-SNSTopic-1XYW5YDS50VOM7C"
      },
      "End": true
    },
    "Notify Failure": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sns:publish",
      "Parameters": {
        "Message": "AWS Fargate Task started by Step Functions failed",
        "TopicArn": "arn:aws:sns:ap-northeast-1:123456789012:FargateTaskNotification-SNSTopic-1XYW5YDS50VOM7C"
      },
      "End": true
    }
  }
}
```
IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. It's a best practice to include only those permissions that are necessary in your IAM policies.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": ["sns:Publish"],
         "Resource": 
           "arn:aws:sns:ap-northeast-1:123456789012:FargateTaskNotification-SNSTopic-1XYW5YD5V0W7C"
      },
      "Effect": "Allow",
      {
         "Action": ["ecs:RunTask"],
         "Resource": 
           "arn:aws:ecs:ap-northeast-1:123456789012:task-definition/FargateTaskNotification-ECSTaskDefinition-13YOJT8ZL5Q:1"
      },
      "Effect": "Allow",
      {
         "Action": ["ecs:StopTask", "ecs:DescribeTasks"],
         "Resource": "*",
         "Effect": "Allow"
      },
      {
         "Action": ["events:PutTargets", "events:PutRule", "events:DescribeRule"],
         "Resource": 
           "arn:aws:events:ap-northeast-1:123456789012:rule/StepFunctionsGetEventsForECSTaskRule"
      },
      "Effect": "Allow"
   ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
Transfer Data Records (Lambda, DynamoDB, Amazon SQS)

This sample project demonstrates how to read values from an Amazon DynamoDB table and send them to Amazon SQS using AWS Step Functions. Deploying this sample project will create a Step Functions state machine, a DynamoDB table, an AWS Lambda function, and an Amazon SQS topic.

In this project, Step Functions uses the Lambda function to populate the DynamoDB table, uses a for loop to read each of the entries, and then sends each entry to Amazon SQS.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Transfer Data Records.

   The state machine Code and Visual Workflow are displayed.

   ![State Machine Diagram]

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

3. Choose Next.

   The Deploy resources page is displayed, listing the resources that will be created. For this sample project the resources include:
   - A Lambda function for seeding the DynamoDB table
   - An Amazon SQS queue
   - A DynamoDB table

4. Choose Deploy Resources.
Note
It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.

Start a New Execution

1. On the New execution page, enter an execution name (optional) and choose Start Execution.
2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

   Note
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
3. Optionally, you can go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.
4. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details

Example State Machine Code

The state machine in this sample project integrates with DynamoDB and Amazon SQS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls DynamoDB and Amazon SQS by connecting to the Amazon Resource Name (ARN) in the Resource field, and by passing Parameters to the service API.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
   "Comment": "An example of the Amazon States Language for reading messages from a DynamoDB table and sending them to SQS",
   "StartAt": "Seed the DynamoDB Table",
   "TimeoutSeconds": 3600,
   "States": {
      "Seed the DynamoDB Table": {
         "Type": "Task",
         "ResultPath": ".List",
         "Next": "For Loop Condition"
      },
      "For Loop Condition": {
         "Type": "Choice",
         "Choices": [
            { "Not": {
               "Variable": ".List[0]",
               "StringEquals": "DONE"
            },
            "Next": "Read Next Message from DynamoDB"
         }
      }
   }
}
```
IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. It’s a best practice to include only those permissions that are necessary in your IAM policies.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:GetItem"
            ],
            "Resource": ["arn:aws:dynamodb:ap-northeast-1:123456789012:table/TransferDataRecords-DDBTable-3I41R5L5EAGT"
        }
    ]
}
```

For more information about passing parameters and managing results, see the following:

- Pass Parameters to a Service API (p. 296)
- ResultPath (p. 62)
Poll for Job Status (Lambda, AWS Batch)

This sample project creates an AWS Batch job poller. It implements an AWS Step Functions state machine that uses AWS Lambda to create a `Wait` state loop that checks on an AWS Batch job.

This sample project creates and configures all resources so that your Step Functions workflow will submit an AWS Batch job, and will wait for that job to complete before ending successfully.

**Note**
You can also implement this pattern without using a Lambda function. For information about controlling AWS Batch directly, see Using AWS Step Functions with other services (p. 271).

This sample project creates the state machine, two Lambda functions, and an AWS Batch queue, and configures the related IAM permissions.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create state machine.
2. Choose Run a sample project, and then choose Job Poller.

   The state machine Definition and Visual Workflow are displayed.
The Definition section in this state machine references the AWS resources that will be created for this sample project.

3. Choose Next.

The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

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

4. Choose Deploy resources.

Note
It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.

Starting an Execution

After all the resources are provisioned and deployed, the Start execution dialog box is displayed with example input similar to the following.

```json
{
    "jobName": "my-job",
    "jobDefinition": "arn:aws:batch:us-east-1:123456789012:job-definition/SampleJobDefinition-343f54b445d5312:1",
    "jobQueue": "arn:aws:batch:us-east-1:123456789012:job-queue/SampleJobQueue-4d9d696031e1449",
    "wait_time": 60
}
```

Note
wait_time instructs the Wait state to loop every 60 seconds.
To start a new execution

1. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name 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.

2. Choose **Start Execution**.

3. (Optional) After the execution is complete, choose individual states on the **Graph inspector**, and then choose the **Step input** and **Step output** tabs to view each state's input and output respectively.

For example, to view the changing status of your AWS Batch job and the looping results of your execution, choose **Step output**.

---

**Example State Machine Code**

The state machine in this sample project integrates with AWS Lambda to submit an AWS Batch job. Browse through this example state machine to see how Step Functions controls Lambda and AWS Batch.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
   "Comment": "An example of the Amazon States Language that runs an AWS Batch job and monitors the job until it completes."
   "StartAt": "Submit Job",
   "States": {
      "Submit Job": {
         "Type": "Task",
         "ResultPath": "$.guid",
         "Next": "Wait X Seconds"
      }
   }
}```
Task Timer (Lambda, Amazon SNS)

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

Note
This sample project implements an AWS Lambda function to send an Amazon Simple Notification Service (Amazon SNS) notification. You can also send an Amazon SNS notification directly from the Amazon States Language. See Using AWS Step Functions with other services (p. 271).

This sample project creates the state machine, a Lambda function, and an Amazon SNS topic, and configures the related AWS Identity and Access Management (IAM) permissions. For more information about the resources that are created with the Task Timer sample project, see the following:
For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

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

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

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Run a sample project**, and then choose **Task Timer**.
   
   The state machine **Definition** and **Visual Workflow** are displayed.

3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

   - A SendToSNS Lambda function
   - A TaskTimerTopic Amazon SNS topic

4. Choose **Deploy resources**.

   **Note**
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

   When complete, the **Start execution** dialog box is displayed, with example input similar to the following.

   ```
   {
   "topic": "arn:aws:sns:us-east-2:123456789012:StepFunctionsSample-TaskTimer-517b8680-e0ad-07cf-feee-65aa5fc63ac0-SNSTopic-96RHT77RAKTS",
   "message": "HelloWorld",
   "timer_seconds": 10
   }
   ```

5. Choose **Start Execution**.
A new execution of your state machine starts, and a new page showing your running execution is displayed.

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

7. To view the status, input, or output of each step in your execution, select the step in the **Visual workflow** and review the **Step details**.

---

### Callback Pattern Example (Amazon SQS, Amazon SNS, Lambda)

This sample project demonstrates how to have AWS Step Functions pause during a task, and wait for an external process to return a task token that was generated when the task started.

When this sample project is deployed and an execution is started, the following steps occur:

1. Step Functions passes a message that includes a task token to an Amazon Simple Queue Service (Amazon SQS) queue.
2. Step Functions then pauses, waiting for that token to be returned.
3. The Amazon SQS queue triggers an AWS Lambda function that calls `SendTaskSuccess` with that same task token.
4. When the task token is received, the workflow continues.
5. The "Notify Success" task publishes an Amazon Simple Notification Service (Amazon SNS) message that the callback was received.

To learn how to implement the callback pattern in Step Functions, see *Wait for a Callback with the Task Token* (p. 293).
For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

Create State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Callback Pattern Example.
   
   The state machine Code and Visual Workflow are displayed.

3. Choose Create Sample Project.
   
   The Create Project Resources page is displayed, listing the resources that will be created. For this sample project the resources include:
   
   - An Amazon SQS message queue.
   - A Lambda function, that calls the Step Functions API action SendTaskSuccess.
   - An Amazon SNS topic to notify success or failure when the workflow can continue.

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

   When complete, the New execution page is displayed.
4. (Optional) Enter an execution name, and sample input.
Lambda Callback Example

To see how the components of this sample project work together, see the resources that were deployed in your AWS account. For example, here is the Lambda function that calls Step Functions with the task token.

```javascript
console.log('Loading function');
const aws = require('aws-sdk');
exports.lambda_handler = (event, context, callback) => {
    const stepfunctions = new aws.StepFunctions();
    // Code to call Step Functions
};
```
Manage an Amazon EMR Job

This sample project demonstrates Amazon EMR and AWS Step Functions integration.

It shows how to create an Amazon EMR cluster, add multiple steps and run them, and then terminate the cluster.

Important
Amazon EMR does not have a free pricing tier. Running the sample project will incur costs. You can find pricing information on the Amazon EMR pricing page. The availability of Amazon EMR service integration is subject to the availability of Amazon EMR APIs. Because of this, this sample project might not work correctly in some AWS Regions. See the Amazon EMR documentation for limitations in special Regions.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Manage an EMR Job.

The state machine Code and Visual Workflow are displayed.
3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project the resources include an Amazon S3 Bucket.

4. Choose **Deploy Resources**.

   **Note**
   It can take up to 10 minutes for these resources and related AWS Identity and Access Management (IAM) permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

---

**Start a New Execution**

1. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.

2. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name 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. (Optional) You can go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.

4. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.

---

**Example State Machine Code**

The state machine in this sample project integrates with Amazon EMR by passing parameters directly to those resources. Browse through this example state machine to see how Step Functions uses a state machine to call the Amazon EMR task synchronously, waits for the task to succeed or fail, and terminates the cluster.
For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
  "Comment": "An example of the Amazon States Language for running jobs on Amazon EMR",
  "StartAt": "Create an EMR cluster",
  "States": {
    "Create an EMR cluster": {
      "Type": "Task",
      "Resource": "arn:<PARTITION>:states::elasticmapreduce:createCluster.sync",
      "Parameters": {
        "Name": "ExampleCluster",
        "VisibleToAllUsers": true,
        "ReleaseLabel": "emr-5.26.0",
        "Applications": [
          { "Name": "Hive " }
        ],
        "ServiceRole": "<EMR_SERVICE_ROLE>",
        "JobFlowRole": "<EMR_EC2_INSTANCE_PROFILE>",
        "LogUri": "s3://<EMR_LOG_S3_BUCKET>/logs/",
        "Instances": {
          "KeepJobFlowAliveWhenNoSteps": true,
          "InstanceFleets": [
            { "Name": "MyMasterFleet",
              "InstanceFleetType": "MASTER",
              "TargetOnDemandCapacity": 1,
              "InstanceTypeConfigs": [
                { "InstanceType": "m5.xlarge"
              }
            ],
            { "Name": "MyCoreFleet",
              "InstanceFleetType": "CORE",
              "TargetOnDemandCapacity": 1,
              "InstanceTypeConfigs": [
                { "InstanceType": "m5.xlarge"
              }
            ]
          ],
          "ResultPath": ".cluster",
          "Next": "Run first step"
        }
      },
      "Run first step": {
        "Type": "Task",
        "Resource": "arn:<PARTITION>:states::elasticmapreduce:addStep.sync",
        "Parameters": {
          "ClusterId.$": ".cluster.ClusterId",
          "Step": {
            "Name": "My first EMR step",
            "ActionOnFailure": "CONTINUE",
            "HadoopJarStep": {
              "Jar": "command-runner.jar",
              "Args": ["<COMMAND_ARGUMENTS>"
            }
          },
          "Retry": [
            { }
```
IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. It's a best practice to include only those permissions that are necessary in your IAM policies.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elasticmapreduce:RunJobFlow",
                "elasticmapreduce:DescribeCluster",
                "elasticmapreduce:TerminateJobFlows"
            ],
            "Resource": "*"
        }
    ]
}
```
The following policy ensures that `addStep` has sufficient permissions.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:AddJobFlowSteps",
        "elasticmapreduce:DescribeStep",
        "elasticmapreduce:CancelSteps"
      ],
      "Resource": "arn:aws:elasticmapreduce::*:*:cluster/*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
Start a Workflow within a Workflow (Step Functions, Lambda)

This sample project demonstrates how to use an AWS Step Functions state machine to launch other state machine executions. See Start Workflow Executions from a Task State (p. 73).

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Start a Workflow Within a Workflow.

The state machine Code and Visual Workflow are displayed.

3. Choose Next.

The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

- An additional Step Functions state machine
- A Lambda function for callback

4. Choose Deploy Resources.

Note
It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.

Start a New Execution

1. On the New execution page, enter an execution name (optional), and then choose Start Execution.
2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

3. Optionally, you can go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.

4. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example State Machine Code

The state machine in this sample project integrates another state machine and AWS Lambda by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions calls the StartExecution API action for the other state machine. It launches two instances of the other state machine in parallel: one using the Run a Job (.sync) (p. 292) pattern and one using the Wait for a Callback with the Task Token (p. 293) pattern.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
    "Comment": "An example of combining workflows using a Step Functions StartExecution task state with various integration patterns.",
    "StartAt": "Start new workflow and continue",
    "States": {
        "Start new workflow and continue": {
            "Comment": "Start an execution of another Step Functions state machine and continue",
            "Type": "Task",
            "Resource": "arn:aws:states:::states:startExecution",
            "Parameters": {
                "Input": {
                    "NeedCallback": false,
                    "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
                }
            }
        },
        "Next": "Start in parallel"
    },
    "Start in parallel": {
        "Comment": "Start two executions of the same state machine in parallel",
        "Type": "Parallel",
        "End": true,
        "Branches": [
            {
                "StartAt": "Start new workflow and wait for completion",
                "States": {
                    "Start new workflow and wait for completion": {
                        "Comment": "Start an execution of the same 'NestingPatternAnotherStateMachine' and wait for its completion",
                        "Type": "Task",
                        "Resource": "arn:aws:states:::states:startExecution.sync",
                        "Parameters": {
                            "Input": {
                                "NeedCallback": false,
                                "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
                            }
                        }
                    }
                }
            }
        ]
    }
}
```
Dynamically process data with a Map state

This sample project demonstrates dynamic parallelism using a Map state. This sample project creates the following:

- Two AWS Lambda functions
- An Amazon Simple Queue Service (Amazon SQS) queue
- An Amazon Simple Notification Service (Amazon SNS) topic
- An Amazon DynamoDB table
- An AWS Step Functions state machine

In this project, Step Functions uses an AWS Lambda function to pull messages off an Amazon SQS queue, and pass a JSON array of those message to a Map state. For each message in the queue, the state machine writes the message to DynamoDB, invokes the other Lambda function to remove the message from Amazon SQS, and then publishes the message to the Amazon SNS topic.

For more information on Map states and Step Functions service integrations, see the following:

- Map (p. 49)
- Using AWS Step Functions with other services (p. 271)
Create the state machine and provision resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Dynamically process data with a Map state.

   The state machine Code and Visual Workflow are displayed.

3. Choose Next.

   The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

   • An Amazon SQS queue
   • An Amazon SNS topic
   • A DynamoDB table
   • Two Lambda functions
   • A Step Functions state machine

4. Choose Deploy Resources.

   **Note**
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.
Once the resources of the sample project are deployed, you need to add items to the Amazon SQS queue and subscribe to the Amazon SNS topic before you start an execution of the state machine.

**Subscribe to the Amazon SNS topic**

1. Open the Amazon SNS console.
2. Choose Topics and choose the topic that was created by the Map state sample project.
   
   The name will be similar to `MapSampleProj-SNSTopic-1CQO4HQ3IR1KN`.
3. Under Subscriptions, choose Create subscription.
   
   The Create subscription page is displayed, listing the Topic ARN for the topic.
4. Under Protocol, choose Email.
5. Under Endpoint, enter an email address to subscribe to the topic.
6. Choose Create subscription.

   **Note**
   You must confirm the subscription in your email before it is active.
7. Open the Subscription Confirmation email in the related account and open the Confirm subscription URL.

   The Subscription confirmed! page is displayed.

**Add messages to the Amazon SQS queue**

1. Open the Amazon SQS console.
2. Choose the queue that was created by the Map state sample project.
   
   The name will be similar to `MapSampleProj-SQSQueue-1UDIC9VZDORN7`.
3. In the Queue Actions list, select Send a Message.
4. On the Send a Message window, enter a message and choose Send Message.
5. Choose Send Another Message.
   
   Continue entering messages until you have several in the Amazon SQS queue.
6. Choose Close.

**Start a new execution**

**Note**

Queues in Amazon SNS are eventually consistent. For best results, wait a few minutes between populating your queue and running an execution of your state machine.

1. Open the Step Functions console.
2. On the State machines page, choose the MapStateMachine state machine that was created by the sample project and choose Start execution.
3. On the New execution page, enter an execution name (optional), and then choose Start Execution.
4. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

5. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.

6. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example state machine code

The state machine in this sample project integrates with Amazon SQS, Amazon SNS, and Lambda by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls Lambda, DynamoDB, Amazon SNS by connecting to the Amazon Resource Name (ARN) in the Resource field, and by passing Parameters to the service API.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
   "Comment": "An example of the Amazon States Language for reading messages from an SQS queue and iteratively processing each message.",
   "StartAt": "Read messages from SQS Queue",
   "States": {
      "Read messages from SQS Queue": {
         "Type": "Task",
         "Resource": "arn:aws:states:::lambda:invoke",
         "OutputPath": "$.Payload",
         "Parameters": {
            "FunctionName": "MapSampleProj-ReadFromSQSQueueLambda-1MY3M63RMJVA9"
         },
         "Next": "Are there messages to process?"
      },
      "Are there messages to process?": {
         "Type": "Choice",
         "Choices": [
            {
               "Variable": 
```
"TableName": "MapSampleProj-DDBTable-YJDJ1MKIN6C5",
"ReturnConsumedCapacity": "TOTAL",
"Item": {
  "MessageId": {
    "S.$": ".MessageDetails.MessageId"
  },
  "Body": {
    "S.$": ".MessageDetails.Body"
  }
},
"Next": "Remove message from SQS queue"
},
"Remove message from SQS queue": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "InputPath": ".MessageDetails",
  "ResultPath": null,
  "Parameters": {
    "FunctionName": "MapSampleProj-DeleteFromSQSQueueLambda-198J2839ZO5K2",
    "Payload": {
      "ReceiptHandle.$": ".ReceiptHandle"
    }
  },
  "Next": "Publish message to SNS topic"
},
"Publish message to SNS topic": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sns:publish",
  "InputPath": ".MessageDetails",
  "Parameters": {
    "Subject": "Message from Step Functions!",
    "Message.$": ".Body",
    "TopicArn": "arn:aws:sns:us-east-1:012345678910:MapSampleProj-SNSTopic-1CQO4HQ3IR1KN"
  }
},
"Finish": {
  "Type": "Succeed"
}

IAM example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:us-east-1:012345678901:function:MapSampleProj-ReadFromSQSQueueLambda-1MY3M63MKJVA9"
      ]
    }
  ]
}```
Train a Machine Learning Model

This sample project demonstrates how to use SageMaker and AWS Step Functions to train a machine learning model and how to batch transform a test dataset. This sample project creates the following:

- An AWS Lambda function
- An Amazon Simple Storage Service (Amazon S3) bucket
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

In this project, Step Functions uses a Lambda function to seed an Amazon S3 bucket with a test dataset. It then trains a machine learning model and performs a batch transform, using the SageMaker service integration (p. 313).

For more information about SageMaker and Step Functions service integrations, see the following:

- Using AWS Step Functions with other services (p. 271)
- Manage SageMaker with Step Functions (p. 313)

Note
This sample project may incur charges.
For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see SageMaker Pricing.
Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Train a machine learning model.

The state machine Code and Visual Workflow are displayed.

3. Choose Next.

The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

- A Lambda function
- An Amazon S3 bucket
- A Step Functions state machine
- Related IAM roles

4. Choose Deploy Resources.

Note
It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.

Start a New Execution

1. Open the Step Functions console.
2. On the State machines page, choose the TrainAndBatchTransformStateMachine state machine that was created by the sample project, and then choose Start execution.
3. On the New execution page, enter an execution name (optional), and then choose Start Execution.
4. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.
Example State Machine Code

The state machine in this sample project integrates with SageMaker and AWS Lambda by passing parameters directly to those resources, and uses an Amazon S3 bucket for the training data source and output.

Browse through this example state machine to see how Step Functions controls Lambda and SageMaker.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
  "StartAt": "Generate dataset",
  "States": {
    "Generate dataset": {
      "Type": "Task",
      "Next": "Train model (XGBoost)"
    },
    "Train model (XGBoost)": {
      "Resource": "arn:aws:states:::sagemaker:createTrainingJob.sync",
      "Parameters": {
        "AlgorithmSpecification": {
          "TrainingImage": "433757028032.dkr.ecr.us-west-2.amazonaws.com/xgboost:latest",
          "TrainingInputMode": "File"
        },
        "OutputDataConfig": {
          "S3OutputPath": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/models"
        },
        "StoppingCondition": {
          "MaxRuntimeInSeconds": 86400
        },
        "ResourceConfig": {
          "InstanceCount": 1,
          "InstanceType": "ml.m4.xlarge",
          "VolumeSizeInGB": 30
        },
        "RoleArn": "arn:aws:iam::123456789012:role/TrainAndBatchTransform-SageMakerAPIExecutionRole-Y9IX3DLF6EUO",
        "InputDataConfig": {
          "DataSource": {
            "S3DataSource": {
              "S3DataDistributionType": "ShardedByS3Key",
              "S3DataType": "S3Prefix",
              "S3Uri": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/csv/train.csv"
            }
          },
          "ChannelName": "train"
        }
    }
  }
}
```
"ContentType": "text/csv"
},
"HyperParameters": {
  "objective": "reg:logistic",
  "eval_metric": "rmse",
  "num_round": "5"
},
"TrainingJobName.$": "$.Execution.Name"
},
"Type": "Task",
"Next": "Save Model"
},
"Save Model": {
  "Parameters": {
    "PrimaryContainer": {
      "Image": "433757028032.dkr.ecr.us-west-2.amazonaws.com/xgboost:latest",
      "Environment": {},
      "ModelDataUrl.$": "$.ModelArtifacts.S3ModelArtifacts"
    },
    "ExecutionRoleArn": "arn:aws:iam::123456789012:role/TrainAndBatchTransform-SageMakerAPIExecutionRole-Y9IX3DLF6EUO",
    "ModelName.$": "$.TrainingJobName"
  },
  "Resource": "arn:aws:states:::sagemaker:createModel",
  "Type": "Task",
  "Next": "Batch transform"
},
"Batch transform": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sagemaker:createTransformJob.sync",
  "Parameters": {
    "ModelName.$": "$.Execution.Name",
    "TransformInput": {
      "CompressionType": "None",
      "ContentType": "text/csv",
      "DataSource": {
        "S3DataSource": {
          "S3DataType": "S3Prefix",
          "S3Uri": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/csv/test.csv"
        }
      }
    },
    "TransformOutput": {
      "S3OutputPath": "s3://trainandbatchtransform-s3bucket-1jn1le6gadwfz/output"
    },
    "TransformResources": {
      "InstanceCount": 1,
      "InstanceType": "ml.m4.xlarge"
    },
    "TransformJobName.$": "$.Execution.Name"
  },
  "End": true
}
IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": [
            "cloudwatch:PutMetricData",
            "logs:CreateLogStream",
            "logs:PutLogEvents",
            "logs:CreateLogGroup",
            "logs:DescribeLogStreams",
            "s3:GetObject",
            "s3:PutObject",
            "s3:ListBucket",
            "ecr:GetAuthorizationToken",
            "ecr:BatchCheckLayerAvailability",
            "ecr:GetDownloadUrlForLayer",
            "ecr:BatchGetImage"
         ],
         "Resource": "*",
         "Effect": "Allow"
      }
   ]
}
```

The following policy allows the Lambda function to seed the Amazon S3 bucket with sample data.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": ["s3:PutObject"],
         "Resource": "arn:aws:s3:::trainandbatchtransform-s3bucket-1jn1le6gadwfz/*",
         "Effect": "Allow"
      }
   ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

Tune a Machine Learning Model

This sample project demonstrates using SageMaker to tune the hyperparameters of a machine learning model, and to batch transform a test dataset. This sample project creates the following:

- Three AWS Lambda functions
- An Amazon Simple Storage Service (Amazon S3) bucket
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles
In this project, Step Functions uses a Lambda function to seed an Amazon S3 bucket with a test dataset. It then creates a hyperparameter tuning job using the SageMaker service integration (p. 313). It then uses a Lambda function to extract the data path, saves the tuning model, extracts the model name, and then runs a batch transform job to perform inference in SageMaker.

For more information about SageMaker and Step Functions service integrations, see the following:

- Using AWS Step Functions with other services (p. 271)
- Manage SageMaker with Step Functions (p. 313)

Note
This sample project may incur charges.
For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see SageMaker Pricing.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Tune a machine learning model.
   
   The state machine Code and Visual Workflow are displayed.

3. Choose Next.
   
   The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

- Three Lambda functions
- An Amazon S3 bucket
- A Step Functions state machine
- Related IAM roles

4. Choose Deploy Resources.
   
   Note
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.

Start a New Execution

1. Open the Step Functions console.
2. On the State machines page, choose the HyperparamTuningAndBatchTransformStateMachine state machine that was created by the sample project and choose Start execution.
3. On the New execution page, enter an execution name (optional), and then choose Start Execution.
4. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

5. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.

6. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

---

### Example State Machine Code

The state machine in this sample project integrates with SageMaker and AWS Lambda by passing parameters directly to those resources, and uses an Amazon S3 bucket for the training data source and output.

Browse through this example state machine to see how Step Functions controls Lambda and SageMaker.

For more information about how AWS Step Functions can control other AWS services, see [Using AWS Step Functions with other services](#).

```json
{
   "StartAt": "Generate Training Dataset",
   "States": {
      "Generate Training Dataset": {
         "Type": "Task",
         "Next": "HyperparameterTuning (XGBoost)"
      },
      "HyperparameterTuning (XGBoost)": {
         "Resource": "arn:aws:states:::sagemaker:createHyperParameterTuningJob.sync",
         "Parameters": {
            "HyperParameterTuningJobName.$": "$\text{.body.jobName}",
            "HyperParameterTuningJobConfig": {
               "Strategy": "Bayesian",
               "HyperParameterTuningJobObjective": {
                  "Type": "Minimize",
                  "MetricName": "validation:rmse"
               },
               "ResourceLimits": {
                  "MaxNumberOfTrainingJobs": 2,
                  "MaxParallelTrainingJobs": 2
               },
               "ParameterRanges": {
                  "ContinuousParameterRanges": [{
                     "Name": "alpha",
                     "MinValue": "0",
                     "MaxValue": "1000",
                     "ScalingType": "Auto"
                  },
                  {
                     "Name": "gamma",
                     "MinValue": "0",
                     "MaxValue": "5",
                     "ScalingType": "Auto"
                  }
               }
            }
         }
      }
   }
}
```
"IntegerParameterRanges": [{
    "Name": "max_delta_step",
    "MinValue": "0",
    "MaxValue": "10",
    "ScalingType": "Auto"
},
{
    "Name": "max_depth",
    "MinValue": "0",
    "MaxValue": "10",
    "ScalingType": "Auto"
}
]
],
"TrainingJobDefinition": {
    "AlgorithmSpecification": {
        "TrainingImage": "433757028032.dkr.ecr.us-west-2.amazonaws.com/xgboost:latest",
        "TrainingInputMode": "File"
    },
    "OutputDataConfig": {
        "S3OutputPath": "s3://stepfunctionssample-sagemak-bucketformodelanddata-80fbmdlcs9f/models"
    },
    "StoppingCondition": {
        "MaxRuntimeInSeconds": 86400
    },
    "ResourceConfig": {
        "InstanceCount": 1,
        "InstanceType": "ml.m4.xlarge",
        "VolumeSizeInGB": 30
    },
    "RoleArn": "arn:aws:iam::012345678912:role/StepFunctionsSample-SageMakerAPIExecutionRole-1MNHIYS5CGG0G",
    "InputDataConfig": [{
        "DataSource": {
            "S3DataSource": {
                "S3DataDistributionType": "FullyReplicated",
                "S3DataType": "S3Prefix",
                "S3Uri": "s3://stepfunctionssample-sagemak-bucketformodelanddata-80fbmdlcs9f/csv/train.csv"
            }
        },
        "ChannelName": "train",
        "ContentType": "text/csv"
    },
    {
        "DataSource": {
            "S3DataSource": {
                "S3DataDistributionType": "FullyReplicated",
                "S3DataType": "S3Prefix",
                "S3Uri": "s3://stepfunctionssample-sagemak-bucketformodelanddata-80fbmdlcs9f/csv/validation.csv"
            }
        },
        "ChannelName": "validation",
        "ContentType": "text/csv"
    }],
    "StaticHyperParameters": {
        "precision_dtype": "float32",
        "num_round": "2"
    }
}
"Type": "Task",
"Next": "Extract Model Path"
},
"Extract Model Path": {
"Type": "Task",
"Next": "HyperparameterTuning - Save Model"
},
"HyperparameterTuning - Save Model": {
"Parameters": {
"PrimaryContainer": {
"Image": "433757028032.dkr.ecr.us-west-2.amazonaws.com/xgboost:latest",
"Environment": {},
"ModelDataUrl.$": "$\$.body.modelDataUrl"
},
"ExecutionRoleArn": "arn:aws:iam::012345678912:role/StepFunctionsSample-SageM-SageMakerAPIExecutionRol-1MNH1VS5CGGOG",
"ModelName.$": "$\$.body.bestTrainingJobName"
},
"Resource": "arn:aws:states:::sagemaker:createModel",
"Type": "Task",
"Next": "Extract Model Name"
},
"Extract Model Name": {
"Type": "Task",
"Next": "Batch transform"
},
"Batch transform": {
"Type": "Task",
"Resource": "arn:aws:states:::sagemaker:createTransformJob.sync",
"Parameters": {
"ModelName.$": "$\$.body.jobName",
"TransformInput": {
"CompressionType": "None",
"ContentType": "text/csv",
"DataSource": {
"S3DataSource": {
"S3DataType": "S3Prefix",
"S3Uri": "s3://stepfunctionssample-sagemak-bucketformodelanddata-80fblmdlcs9f/csv/test.csv"
}
}
},
"TransformOutput": {
"S3OutputPath": "s3://stepfunctionssample-sagemak-bucketformodelanddata-80fblmdlcs9f/csv/"
},
"TransformResources": {
"InstanceCount": 1,
"InstanceType": "ml.m4.xlarge"
},
"TransformJobName.$": "$\$.body.jobName"
},
"End": true
}

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
IAM Examples

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

The following IAM policy is attached to the state machine, and allows the state machine execution to access necessary SageMaker, Lambda, and Amazon S3 resources.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": [
            "sagemaker:CreateHyperParameterTuningJob",
            "sagemaker:DescribeHyperParameterTuningJob",
            "sagemaker:StopHyperParameterTuningJob",
            "sagemaker:ListTags",
            "sagemaker:CreateModel",
            "sagemaker:CreateTransformJob",
            "iam:PassRole"
         ],
         "Resource": "*",
         "Effect": "Allow"
      },
      {
         "Action": ["lambda:InvokeFunction"],
         "Resource": [
            "arn:aws:lambda:us-west-2:012345678912:function:StepFunctionsSample-SageMa-
            LambdaForDataGeneration-1TF67BUE5A12U",
            "arn:aws:lambda:us-west-2:012345678912:function:StepFunctionsSample-SageM-
            LambdaToExtractModelPath-V0R37CVARUS9",
            "arn:aws:lambda:us-west-2:012345678912:function:StepFunctionsSample-SageM-
            LambdaToExtractModelName-8FUOB30S5EM"
         ],
         "Effect": "Allow"
      },
      {
         "Action": ["events:PutTargets", "events:PutRule", "events:DescribeRule"],
         "Resource": [
            "arn:aws:events:*:*:rule/StepFunctionsGetEventsForSageMakerTrainingJobsRule",
            "arn:aws:events:*:*:rule/StepFunctionsGetEventsForSageMakerTransformJobsRule",
            "arn:aws:events:*:*:rule/StepFunctionsGetEventsForSageMakerTuningJobsRule"
         ],
         "Effect": "Allow"
      }
   ]
}
```

The following IAM policy is referenced in the TrainingJobDefinition and HyperparameterTuning fields of the HyperparameterTuning state.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Resource": "*",
         "Effect": "Allow"
      }
   ]
}
```
The following IAM policy allows the Lambda function to seed the Amazon S3 bucket with sample data.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:PutObject"
      ],
      "Resource": "arn:aws:s3:::stepfunctionssample-sagemak-bucketformodelanddata-80fbmldcs9f/*",
      "Effect": "Allow"
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
Process High-Volume Messages from Amazon SQS (Express Workflows)

This sample project demonstrates how to use an AWS Step Functions Express Workflow to process messages or data from a high-volume event source, such as Amazon Simple Queue Service (Amazon SQS). Because Express Workflows can be started at a very high rate, they are ideal for high-volume event processing or streaming data workloads.

Here are two commonly used methods to execute your state machine from an event source:

• **Configure an Amazon CloudWatch Events rule to start a state machine execution whenever the event source emits an event.** For more information, see Creating a CloudWatch Events Rule That Triggers on an Event.

• **Map the event source to a Lambda function, and write function code to execute your state machine.** The AWS Lambda function is invoked each time your event source emits an event, in turn starting a state machine execution. For more information see Using AWS Lambda with Amazon SQS.

This sample project uses the second method to start an execution each time the Amazon SQS queue sends a message. You can use a similar configuration to trigger Express Workflows execution from other event sources, such as Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and Amazon Kinesis.

For more information about Express Workflows and Step Functions service integrations, see the following:

• Standard vs. Express Workflows (p. 19)
• Using AWS Step Functions with other services (p. 271)
• Quotas (p. 485)

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Run a sample project, and then choose Process high-volume messages from Amazon SQS.

   The state machine Code and Visual Workflow are displayed.
3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:
   
   - A Step Functions state machine
   - An Amazon SQS queue
   - A Lambda function

4. Choose **Deploy Resources**.

   **Note**
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

**Trigger Execution**

1. Open the **Amazon SQS console**.
2. Select the queue that was created by the sample project.

   The name will be similar to **Example-SQSQueue-wJalrXUtnFEMI**.
3. In the **Queue Actions** list, select **Send a Message**.
4. Use the copy button to copy the following message, and on the **Send a Message** window, enter it, and choose **Send Message**.
Note

In this sample message, the `input:` line has been formatted with line breaks to fit the page. Use the copy button or otherwise ensure that it is entered as a single line with no breaks.

```json
{
    "input": "QW5kIGxpa2UgdGhlIGJhc2VsZXNzIGZyZnJpYyBvZiB0aGlzIHZpc2lvbiwgVGhlIGNsb3VkLWNhcHBlZCB0b3d1cnMnIHRoZW91cyBvYnNlcmluZw==",
    "Comment": "An example of using Express workflows to run text processing for each message sent from an SQS queue.",
    "StartAt": "Decode base64 string",
    "States": {
        "Decode base64 string": {
            "Type": "Task",
            "Resource": "arn:<PARTITION>:states:::lambda:invoke",
            "Input": "QW5kIHRoZXJlIHRoZXJlIHRoZXJlIHRoZXJlIHR3byBJ4oCZzbGwg2FsayBUyBzdlGlsCBteSBiZWF0aW5nIG1pbmQu"
        }
    }
}
```

5. Choose Close.
6. Open the Step Functions console.
7. Go to your Amazon CloudWatch Logs log group and inspect the logs. The name of the log group will look like `example-ExpressLogGroup-wJalrXUtFEMI`.

### Example Lambda Function Code

The following is Lambda function code that shows how the initiating Lambda function starts a state machine execution using the AWS SDK.

```python
import boto3

def lambda_handler(event, context):
    message_body = event['Records'][0]['body']
    client = boto3.client('stepfunctions')
    response = client.start_execution(
        stateMachineArn='${ExpressStateMachineArn}',
        input=message_body
    )
```

### Example State Machine Code

The Express Workflow in this sample project consists of a set of Lambda functions for text processing.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).
IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [ "lambda:InvokeFunction" ],
      "Resource": [ "arn:aws:lambda:us-east-1:123456789012:function:example-Base64DecodeLambda-wJalrXUtFEMI",
                    "arn:aws:lambda:us-east-1:123456789012:function:example-StringCleanerLambda-je7MtGbClwBF",
                    "arn:aws:lambda:us-east-1:123456789012:function:example-TokenizerCounterLambda-wJalrXUtFEMI",
                    "arn:aws:lambda:us-east-1:123456789012:function:example-GenerateStatsLambda-je7MtGbClwBF"
      ]
    }
  ]
}
```
The following policy ensures that there are sufficient permissions for CloudWatch Logs.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": [
            "logs:CreateLogDelivery",
            "logs:GetLogDelivery",
            "logs:UpdateLogDelivery",
            "logs:DeleteLogDelivery",
            "logs:ListLogDeliveries",
            "logs:PutResourcePolicy",
            "logs:DescribeResourcePolicies",
            "logs:DescribeLogGroups"
         ],
         "Resource": ["*"]
      },
      {
         "Effect": "Allow"
      }
   ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

## Selective Checkpointing Example (Express Workflows)

This sample project demonstrates how to combine Standard and Express Workflows by running a mock e-commerce workflow that does selective checkpointing. Deploying this sample project creates a Standard workflows state machine, a nested Express Workflows state machine, an AWS Lambda function, an Amazon Simple Queue Service (Amazon SQS) queue, and an Amazon Simple Notification Service (Amazon SNS) topic.

For more information about Express Workflows, nested workflows, and Step Functions service integrations, see the following:

- Standard vs. Express Workflows (p. 19)
- Start Workflow Executions from a Task State (p. 73)
- Using AWS Step Functions with other services (p. 271)

## Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Run a sample project, and then choose Selective checkpointing example.

   The Standard Workflows state machine Code and Visual Workflow are displayed.
After the sample project is deployed, you can view the state machine **Code** and **Visual Workflow** of the nested Express Workflow.

3. Choose **Next**.

The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

- A Step Functions state machine
- An Amazon SQS queue
- A Lambda function

4. Choose **Deploy Resources**.

   **Note**
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

After the resources of the sample project are deployed do the following.

**Start a New Execution**

1. Open the **Step Functions console**.
2. On the **State machines** page, choose the state machine that was created by the sample project and select **Start execution**.
3. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
4. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.
Note
Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

5. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose Start execution.
6. When an execution is complete, you can open the Step Functions console.
7. Go to your CloudWatch Logs log group and inspect the logs. The name of the log group will look like example-ExpressLogGroup-wJalrXUtnFEMI.

Example State Machine Code for the Parent (Standard Workflows)

The state machine in this sample project integrates with Amazon SQS, Amazon SNS, and Step Functions Express Workflows.

Browse through this example state machine to see how Step Functions processes input from Amazon SQS and Amazon SNS, and then uses a nested Express Workflows state machine to update backend systems.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
    "Comment": "An example of combining standard and express workflows to run a mock e-commerce workflow that does selective checkpointing."
    "StartAt": "Approve Order Request",
    "States": {
        "Approve Order Request": {
            "Type": "Task",
            "Resource": "arn:<PARTITION>:states:::sqs:sendMessage.waitForTaskToken",
            "Parameters": {
                "QueueUrl": "<SQS_QUEUE_URL>",
                "MessageBody": {
                    "MessageTitle": "Order Request received. Pausing workflow to wait for manual approval. ",
                    "TaskToken.$": "$.Task.Token"
                }
            },
            "Next": "Notify Order Success",
            "Catch": [
                {
                    "ErrorEquals": [
                        "States.ALL"
                    ],
                    "Next": "Notify Order Failure"
                }
            ],
            "TaskToken.$": "$.Task.Token"
        }
    }
}
```

401
"Notify Order Failure": {
  "Type": "Task",
  "Resource": "arn:<PARTITION>:states:::sns:publish",
  "Parameters": {
    "Message": "Order not approved. Order failed.",
    "TopicArn": "<SNS_ARN>"
  },
  "End": true
},
"Process Payment": {
  "Type": "Task",
  "Resource": "arn:<PARTITION>:states:::sqs:sendMessage.waitForTaskToken",
  "Parameters": {
    "QueueUrl": "<SQS_QUEUE_URL>",
    "MessageBody": {
      "MessageTitle": "Payment sent to third-party for processing. Pausing workflow to wait for response.",
      "TaskToken.$": "$$.Task.Token"
    }
  },
  "Next": "Notify Payment Success",
  "Catch": [
    {
      "ErrorEquals": [
        "States.ALL"
      ],
      "Next": "Notify Payment Failure"
    }
  ]
},
"Notify Payment Success": {
  "Type": "Task",
  "Resource": "arn:<PARTITION>:states:::sns:publish",
  "Parameters": {
    "Message": "Payment processing succeeded. Resuming workflow.",
    "TopicArn": "<SNS_ARN>"
  },
  "Next": "Workflow to Update Backend Systems"
},
"Notify Payment Failure": {
  "Type": "Task",
  "Resource": "arn:<PARTITION>:states:::sns:publish",
  "Parameters": {
    "Message": "Payment processing failed.",
    "TopicArn": "<SNS_ARN>"
  },
  "End": true
},
"Workflow to Update Backend Systems": {
  "Comment": "Starting an execution of an Express workflow to handle backend updates. Express workflows are fast and cost-effective for steps where checkpointing isn't required.",
  "Type": "Task",
  "Resource": "arn:<PARTITION>:states:::states:startExecution.sync",
  "Parameters": {
    "StateMachineArn": "<UPDATE_DATABASE_EXPRESS_STATE_MACHINE_ARN>",
    "Input": {
      "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$\#.Execution.Id"
    }
  },
  "Next": "Ship the Package"
},
"Ship the Package": {
  "Type": "Task",
  "Resource": "arn:<PARTITION>:states:::sns:publish",
  "Parameters": {
    "Message": "Package shipped. Resume workflow.",
    "TopicArn": "<SNS_ARN>"
  },
  "End": true
}
Example IAM Role for the Parent State Machine

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

Amazon SNS policy:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": ["sns:Publish"],
      "Effect": "Allow"
    }
  ]
}
```

Amazon SQS policy:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": ["sqs:SendMessage"],
      "Resource": "arn:aws:sqs:us-east-1:123456789012:Checkpoint-SQSQueue-je7MtGbClwBF",
      "Effect": "Allow"
    }
  ]
}
```

States execution policy:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": ["states:StartExecution", "states:DescribeExecution", "states:StopExecution"],
      "Resource": "*"
    }
  ]
}
```
Example State Machine Code for the Nested State Machine (Express Workflows)

The state machine in this sample project updates backend information when called by the parent state machine.

Browse through this example state machine to see how Step Functions updates the different components of the mock e-commerce backend systems.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```
"Effect": "Allow"
},
{
"Action": [
"events:PutTargets",
"events:PutRule",
"events:DescribeRule"
],
"Resource": "arn:aws:events:us-east-1:123456789012:rule/StepFunctionsGetEventsForStepFunctionsExecutionRule",
"Effect": "Allow"
}
```


Example IAM Role for Child State Machine

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
```
The following policy ensures that there are sufficient permissions for CloudWatch Logs.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "logs:CreateLogDelivery",
        "logs:GetLogDelivery",
        "logs:UpdateLogDelivery",
        "logs:DeleteLogDelivery",
        "logs:ListLogDeliveries",
        "logs:PutResourcePolicy",
        "logs:DescribeResourcePolicies",
        "logs:DescribeLogGroups"
      ],
      "Resource": [
        "*",
      ],
      "Effect": "Allow"
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

**Build an AWS CodeBuild Project (CodeBuild, Amazon SNS)**

This sample project demonstrates how to use AWS Step Functions to build an AWS CodeBuild project, run tests, and then send an Amazon SNS notification.

**Create the State Machine and Provision Resources**

1. Open the Step Functions console, and then choose Create a state machine.
2. Choose Sample Projects, and then choose Start a CodeBuild build.

   The state machine Code and Visual Workflow are displayed.
3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:
   
   - A CodeBuild project
   - An Amazon SNS topic

4. Choose **Deploy Resources**.

   **Note**
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

### Start a New Execution

1. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
2. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name 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. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose **Start execution**.

4. When an execution is complete, select states on the **Visual workflow**, and browse the Input and **Output** under **Step details**.

## Example State Machine Code

The state machine in this sample project integrates with CodeBuild and Amazon SNS.

Browse through this example state machine to see how Step Functions uses a state machine to build a CodeBuild project, and then sends an Amazon SNS topic with a message about whether the job succeeded or failed.

For more information about how Step Functions can control other AWS services, see *Using AWS Step Functions with other services* (p. 271).

```json
{
    "Comment": "An example of using CodeBuild to run tests, get test results and send a notification.",
    "StartAt": "Trigger CodeBuild Build",
    "States": {
        "Trigger CodeBuild Build": {
            "Type": "Task",
            "Resource": "arn:aws:states:::codebuild:startBuild.sync",
            "Parameters": {
                "ProjectName": "CodeBuildProject-Dtw1jBhEYGDf"
            },
            "Next": "Get Test Results"
        },
        "Get Test Results": {
            "Type": "Task",
            "Resource": "arn:aws:states:::codebuild:batchGetReports",
            "Parameters": {
                "ReportArns.$": "$/.Build.ReportArns"
            },
            "Next": "All Tests Passed?"
        },
        "All Tests Passed?": {
            "Type": "Choice",
            "Choices": [
                {
                    "Variable": ".Reports[0].Status",
                    "StringEquals": "SUCCEEDED",
                    "Next": "Notify Success"
                }
            ],
            "Default": "Notify Failure"
        },
        "Notify Success": {
            "Type": "Task",
            "Resource": "arn:aws:states:::sns:publish",
            "Parameters": {
                "Message": "CodeBuild build tests succeeded",
            },
            "End": true
        },
        "Notify Failure": {
            "Type": "Task",
            "Resource": "arn:aws:states:::sns:publish",
```
Preprocess data and train a machine learning model

This sample project demonstrates how to use SageMaker and AWS Step Functions to preprocess data and train a machine learning model. This sample project creates the following:

- An AWS Lambda function
- An Amazon Simple Storage Service (Amazon S3) bucket
- An AWS Step Functions state machine
- Related AWS Identity and Access Management (IAM) roles

In this project, Step Functions uses a Lambda function to seed an Amazon S3 bucket with a test dataset and a Python script for data processing. It then trains a machine learning model and performs a batch transform, using the SageMaker service integration (p. 313).

For more information about SageMaker and Step Functions service integrations, see the following:

- Using AWS Step Functions with other services (p. 271)
- Manage SageMaker with Step Functions (p. 313)

**Note**
This sample project may incur charges.
For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see SageMaker Pricing.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Preprocess data and train a machine learning model.

The state machine Code and Visual Workflow are displayed.
3. Choose Next.

The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

- A Lambda function
- An Amazon S3 bucket
- A Step Functions state machine
- Related IAM roles

4. Choose Deploy Resources.

Note
It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.
Start a New Execution

1. Open the Step Functions console.
2. On the State machines page, choose the FeatureTransformStateMachine state machine that was created by the sample project, and then choose Start execution.
3. On the New execution page, enter an execution name (optional), and then choose Start Execution.
4. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

5. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.
6. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example State Machine Code

The state machine in this sample project integrates with SageMaker and AWS Lambda by passing parameters directly to those resources, and uses an Amazon S3 bucket for the training data source and output.

Browse through this example state machine to see how Step Functions controls Lambda and SageMaker.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
  "StartAt": "Generate dataset",
  "States": {
    "Generate dataset": {
      "Type": "Task",
      "Next": "Standardization: x' = (x - x#) / #"
    },
    "Standardization: x' = (x - x#) / #": {
      "Resource": "arn:aws:states:::sagemaker:createProcessingJob.sync",
      "Parameters": {
        "ProcessingResources": {
          "ClusterConfig": {
            "InstanceCount": 1,
            "InstanceType": "ml.m5.xlarge",
            "VolumeSizeInGB": 10
          }
        },
        "ProcessingInputs": [
          {
            "InputName": "input-1",
            "S3Input": {
              "S3Url": "s3://featuretransform-bucketforcodeanddata-1jn1le6gadwfz/input/raw.csv",
              "LocalPath": "/opt/ml/processing/input",
              "S3DataType": "S3Prefix",
              "S3InputMode": "File",
```
"S3DataDistributionType": "FullyReplicated",
"S3CompressionType": "None"
}
},
{  
"InputName": "code",
"S3Input": {  
  "S3Uri": "s3://featuretransform-bucketforcodeanddata-1jn1le6gadwfz/code/transform.py",
  "LocalPath": "/opt/ml/processing/input/code",
  "S3DataType": "S3Prefix",
  "S3InputMode": "File",
  "S3DataDistributionType": "FullyReplicated",
  "S3CompressionType": "None"
}
]
},
"ProcessingOutputConfig": {
  "Outputs": [
    {
      "OutputName": "train_data",
      "S3Output": {
        "S3Uri": "s3://featuretransform-bucketforcodeanddata-1jn1le6gadwfz/train",
        "LocalPath": "/opt/ml/processing/output/train",
        "S3UploadMode": "EndOfJob"
      }
    }
  ]
},
"AppSpecification": {
  "ImageUri": "737474898029.dkr.ecr.sa-east-1.amazonaws.com/sagemaker-scikit-learn:0.20.0-cpu-py3",
  "ContainerEntrypoint": [  
    "python3",
    "/opt/ml/processing/input/code/transform.py"
  ]
},
"StoppingCondition": {
  "MaxRuntimeInSeconds": 300
},
"RoleArn": "arn:aws:iam::1234567890:role/SageMakerAPIExecutionRole-AIDACKECV5Q6C2EXAMPLE",
"ProcessingJobName.$": "$$.Execution.Name"
},
"Type": "Task",
"Next": "Train model (XGBoost)"
},
"Train model (XGBoost)": {
  "Resource": "arn:aws:states:::sagemaker:createTrainingJob.sync",
  "Parameters": {
    "AlgorithmSpecification": {
      "TrainingImage": "855470959533.dkr.ecr.sa-east-1.amazonaws.com/xgboost:latest",
      "TrainingInputMode": "File"
    },
    "OutputDataConfig": {
      "S3OutputPath": "s3://featuretransform-bucketforcodeanddata-1jn1le6gadwfz/models"
    },
    "StoppingCondition": {
      "MaxRuntimeInSeconds": 86400
    },
    "ResourceConfig": {
      "InstanceCount": 1,
      "InstanceType": "ml.m5.xlarge",
      "VolumeSizeInGB": 30
    }
  }
}
"RoleArn": "arn:aws:iam::1234567890:role/SageMakerAPIExecutionRole-AIDACKCEVSQ6C2EXAMPLE",
"InputDataConfig": [
    {
        "DataSource": {
            "S3DataSource": {
                "S3DataDistributionType": "ShardedByS3Key",
                "S3DataType": "S3Prefix",
                "S3Uri": "s3://featuretransform-bucketforcodeanddata-1jn1le6gadwzf"
            }
        },
        "ChannelName": "train",
        "ContentType": "text/csv"
    }
],
"HyperParameters": {
    "objective": "reg:logistic",
    "eval_metric": "rmse",
    "num_round": "5"
},
"TrainingJobName.$": "$.Execution.Name"
】,
"Type": "Task",
"End": true
}

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

{  
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "cloudwatch:PutMetricData",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:DescribeLogStreams",
                "s3:GetObject",
                "s3:PutObject",
                "s3:ListBucket",
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage"
            ],
            "Resource": "+",
            "Effect": "Allow"
        }
    ]
}

The following policy allows the Lambda function to seed the Amazon S3 bucket with sample data.
Lambda orchestration example

This sample project demonstrates how to integrate AWS Lambda functions in Step Functions state machines. This sample project creates the following:

- Five Lambda functions
- An Amazon Simple Queue Service queue
- An Amazon Simple Notification Service topic
- Related AWS Identity and Access Management (IAM) roles

In this project, Step Functions uses Lambda functions to check a stock price and determine a buy or sell trading recommendation. The user is then provided this recommendation and can choose whether to buy or sell the stock. The result of the trade is returned using an SNS topic.

For more information about Step Functions service integrations, see the following:

- Using AWS Step Functions with other services (p. 271)
- IAM policies for:
  - AWS Lambda (p. 533)
  - Amazon Simple Queue Service (p. 538)
  - Amazon Simple Notification Service (p. 538)

**Note**
This sample project may incur charges.
For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the free tier, see [Pricing](#).

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create state machine.
2. On the Choose authoring method page, choose Run a sample project.
3. Choose Orchestrate Lambda functions.

   The state machine Definition and Visual Workflow are displayed.
4. Choose Next.

The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

- A state machine
- An Amazon SQS queue
- Five Lambda functions
- An Amazon SNS topic
- Related IAM roles

5. Choose Deploy Resources.

Note
It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.
Start a New Execution

After all the resources are provisioned and deployed, the **Start execution** dialog box is displayed.

1. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name 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.

2. Choose **Start execution**.

3. (Optional) After the execution is complete, choose individual states on the **Graph inspector**, and then choose the **Step input** and **Step output** tabs to view each state's input and output respectively.

Example State Machine Code

The state machine in this sample project integrates with AWS Lambda by passing parameters directly to those resources, uses an Amazon SQS queue to manage the request for human approval, and uses an Amazon SNS topic to return the results of the query.

Browse through this example state machine to see how Step Functions controls Lambda and Amazon SQS.

For more information about how AWS Step Functions can control other AWS services, see *Using AWS Step Functions with other services (p. 271)*.

```json
{
    "StartAt": "Check Stock Price",
    "States": {
        "Check Stock Price": {
            "Type": "Task",
            "Next": "Generate Buy/Sell recommendation"
        },
        "Generate Buy/Sell recommendation": {
            "Type": "Task",
            "ResultPath": "$.recommended_type",
            "Next": "Request Human Approval"
        },
        "Request Human Approval": {
            "Type": "Task",
            "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
            "Parameters": {
                "QueueUrl": "https://sqs.us-west-1.amazonaws.com/111122223333/StepFunctionsSample-HelloLambda4444-5555-6666-RequestHumanApprovalSqs-777788889999",
                "MessageBody": {
                    "Input.$": ",
                    "TaskToken.$": "$.Task.Token"
                }
            },
            "ResultPath": null,
            "Next": "Buy or Sell?"
        }
    }
}
```
IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
   "Statement": [
      {
         "Action": [
            "lambda:InvokeFunction"
         ],
         "Effect": "Allow"
      }
   ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
IAM Example

```json
{
    "Statement": [
        {
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Effect": "Allow"
        }
    ]
}
{
    "Statement": [
        {
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Effect": "Allow"
        }
    ]
}
{
    "Statement": [
        {
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Effect": "Allow"
        }
    ]
}
{
    "Statement": [
        {
            "Action": [
                "sqs:SendMessage*"
            ],
            "Effect": "Allow"
        }
    ]
}
{
    "Statement": [
        {
            "Action": [
                "sns:Publish"
            ]
        }
    ]
}
```
Start an Athena query

This sample project demonstrates how to use Step Functions and Amazon Athena to start an Athena query and send a notification with query results. This sample project creates the following:

- An Amazon Athena query
- An AWS Glue crawler
- An Amazon Simple Notification Service topic
- Related AWS Identity and Access Management (IAM) roles

In this project, Step Functions uses Lambda functions and an AWS Glue crawler to generate a set of example data. It then performs a query using the Athena service integration (p. 332) and returns the results using an SNS topic.

For more information about Athena and Step Functions service integrations, see the following:

- Using AWS Step Functions with other services (p. 271)
- Call Athena with Step Functions (p. 332)

**Note**

This sample project may incur charges. For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see Athena Pricing.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Start an Athena query.

The state machine Code and Visual Workflow are displayed.
3. Choose **Next**.

The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

- Two Lambda functions
- A state machine
- An SNS topic
- Related IAM roles

4. Choose **Deploy Resources**.

**Note**

It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.
Start a New Execution

1. Open the Step Functions console.
2. On the State machines page, choose the AthenaStateMachine state machine that was created by the sample project, and then choose Start execution.
3. On the New execution page, enter an execution name (optional), and then choose Start Execution.
4. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

   **Note**
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
5. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.
6. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example State Machine Code

The state machine in this sample project integrates with Athena and AWS Lambda by passing parameters directly to those resources, and uses an SNS topic to return the results of the query.

Browse through this example state machine to see how Step Functions controls Lambda and Athena.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
  "StartAt": "Generate example log",
  "States": {
    "Generate example log": {
      "Type": "Task",
      "Next": "Run Glue crawler"
    },
    "Run Glue crawler": {
      "Type": "Task",
      "Next": "Start an Athena query"
    },
    "Start an Athena query": {
      "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
      "Parameters": {
        "QueryString": "SELECT * FROM "athena-sample-project-db-wJalrXUtEFM"."log" limit 1",
        "WorkGroup": "stepfunctions-athena-sample-project-workgroup-wJalrXUtEFM"
      },
      "Type": "Task",
      "Next": "Get query results"
    },
    "Get query results": {
      "Resource": "arn:aws:states:::athena:getQueryResults",
      "Parameters": {
```
For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:us-east-1:111122223333:function:StepFunctionsSample-AthenaLambdaForInvokingCrawler-AKIAI44QH8DHBEXAMPLE"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "sns:Publish"
      ],
      "Resource": [
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "athena:getQueryResults",
        "athena:startQueryExecution",
        "athena:stopQueryExecution",
        "athena:getQueryExecution",
        "athena:getDataCatalog"
      ],
      "Resource": [
      ]
    }
  ]
}```
Execute multiple queries (Amazon Athena, Amazon SNS)

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

This sample project demonstrates how to run Athena queries in succession and then in parallel, handle errors and then send an Amazon SNS notification based on whether the queries succeed or fail.
Deploying this sample project will create an AWS Step Functions state machine, Amazon Athena queries, and an Amazon SNS topic.

In this project, Step Functions uses a state machine to run Athena queries synchronously. After the query results are returned, enter parallel state with two Athena queries executing in parallel. It then waits for the job to succeed or fail, and it sends an Amazon SNS topic with a message about whether the job succeeded or failed.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects for Step Functions, and then choose Execute multiple queries.

The state machine Code and Visual Workflow are displayed.

3. Choose Next.

The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

- Amazon Athena queries
- An Amazon SNS topic

4. Choose Deploy Resources.

Note
It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.
Start a New Execution

1. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.

2. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.

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

3. Optionally, you can go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.

4. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.

Example State Machine Code

The state machine in this sample project integrates with Amazon Athena and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls Amazon Athena and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the **Resource** field, and by passing **Parameters** to the service API.

For more information about how AWS Step Functions can control other AWS services, see **Using AWS Step Functions with other services** (p. 271).

```json
{
   "Comment": "An example of using Athena to execute queries in sequence and parallel, with error handling and notifications.",
   "StartAt": "Generate Example Data",
   "States": {
      "Generate Example Data": {
        "Type": "Task",
        "Resource": "arn:aws:states:::lambda:invoke",
        "OutputPath": ".Payload",
        "Parameters": {
          "FunctionName": "<ATHENA_FUNCTION_NAME>"
        },
        "Next": "Load Data to Database"
      },
      "Load Data to Database": {
        "Type": "Task",
        "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
        "Parameters": {
          "QueryString": "<ATHENA_QUERYSTRING>",
          "WorkGroup": "<ATHENA_WORKGROUP>"
        },
        "Catch": [
        {
          "ErrorEquals": ["States.ALL"],
          "Next": "Send query results"
        }
      ],
      "Next": "Map"
   }
}
```
"Map": {
  "Type": "Parallel",
  "ResultSelector": {
    "Query1Result.$": "$[0].ResultSet.Rows",
    "Query2Result.$": "$[1].ResultSet.Rows"
  },
  "Catch": [
    { "ErrorEquals": [ "States.ALL" ], "Next": "Send query results" }
  ],
  "Branches": [
    {
      "StartAt": "Start Athena query 1",
      "States": {
        "Start Athena query 1": {
          "Type": "Task",
          "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
          "Parameters": {
            "QueryString": "<ATHENA_QUERYSTRING>",
            "WorkGroup": "<ATHENA_WORKGROUP>"
          },
          "Next": "Get Athena query 1 results"
        },
        "Get Athena query 1 results": {
          "Type": "Task",
          "Resource": "arn:aws:states:::athena:getQueryResults",
          "Parameters": {
            "QueryExecutionId.$": "$[0].QueryExecution.QueryExecutionId"
          },
          "End": true
        }
      }
    },
    {
      "StartAt": "Start Athena query 2",
      "States": {
        "Start Athena query 2": {
          "Type": "Task",
          "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
          "Parameters": {
            "QueryString": "<ATHENA_QUERYSTRING>",
            "WorkGroup": "<ATHENA_WORKGROUP>"
          },
          "Next": "Get Athena query 2 results"
        },
        "Get Athena query 2 results": {
          "Type": "Task",
          "Resource": "arn:aws:states:::athena:getQueryResults",
          "Parameters": {
            "QueryExecutionId.$": "$[1].QueryExecution.QueryExecutionId"
          },
          "End": true
        }
      }
    }
  ],
  "Next": "Send query results"
},
"Send query results": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sns:publish",
  "Parameters": {
    "TargetArn": "<SNS_TOPIC_ARN>",
    "Message": "Send query results"
  }
}
IAM Examples

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

AthenaStartQueryExecution

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:startQueryExecution",
                "athena:stopQueryExecution",
                "athena:getQueryExecution",
                "athena:getDataCatalog"
            ],
            "Resource": [
                "arn:aws:athena:us-east-1:946551482283:datascatalog/**
            ],
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetBucketLocation",
                "s3:GetObject",
                "s3:ListBucket",
                "s3:ListBucketMultipartUploads",
                "s3:ListMultipartUploadParts",
                "s3:AbortMultipartUpload",
                "s3:CreateBucket",
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::*"
            ],
        },
        {
            "Effect": "Allow",
            "Action": [
                "glue:CreateDatabase",
                "glue:GetDatabase",
                "glue:GetDatabases",
                "glue:UpdateDatabase",
                "glue:DeleteDatabase",
                "glue:CreateTable",
                "glue:GetTable",
                "glue:GetTables",
                "glue:GetTableVersions"...
            ],
            "Resource": [
            ],
        }
    ]
}
```
"glue:DeleteTable",
"glue:BatchDeleteTable",
"glue:BatchCreateTable",
"glue:CreatePartition",
"glue:UpdatePartition",
"glue:GetPartition",
"glue:GetPartitions",
"glue:GetPartition",
"glue:DeletePartition",
"glue:BatchDeletePartition"
],
"Resource": [
  "arn:aws:glue:us-east-1:946551482283:catalog",
  "arn:aws:glue:us-east-1:946551482283:database/*",
  "arn:aws:glue:us-east-1:946551482283:table/*",
  "arn:aws:glue:us-east-1:946551482283:userDefinedFunction/*"
]
},
{
  "Effect": "Allow",
  "Action": ["lakeformation:GetDataAccess"
  ],
  "Resource": [
    "*"
  ]
}

AthenaGetQueryResults

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["athena:getQueryResults"
        ],
      "Resource": [
        "arn:aws:athena:us-east-1:946551482283:workgroup/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject"
        ],
      "Resource": [
        "arn:aws:s3:::*"
      ]
    }
  ]
}

SNSPublish

{
  "Version": "2012-10-17",
}
Query large datasets (Amazon Athena, Amazon S3, AWS Glue, Amazon SNS)

This sample project demonstrates how to ingest a large data set in Amazon S3 and partition it through AWS Glue Crawlers, then execute Amazon Athena queries against that partition. Deploying this sample project creates an AWS Step Functions state machine, an Amazon S3 Bucket, an AWS Glue crawler, and an Amazon SNS topic.

In this project, Step Functions uses a state machine to invoke an AWS Glue crawler that partitions a large dataset in Amazon S3. Once the AWS Glue crawler returns a success message, the workflow executes Athena queries against that partition. Once query execution is successfully complete, an Amazon SNS notification is sent to an Amazon SNS topic.

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects for Step Functions, and then choose Query large datasets.

The state machine Code and Visual Workflow are displayed.

3. Choose Next.

The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

- An AWS Glue crawler
- An Amazon S3 Bucket
- Amazon Athena queries
- An Amazon SNS topic

4. Choose Deploy Resources.

**Note**
It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.

Start a New Execution

1. On the New execution page, enter an execution name (optional), and then choose Start Execution.
2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name 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. (Optional) You can go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.

4. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example State Machine Code

The state machine in this sample project integrates with Amazon S3, AWS Glue, Amazon Athena and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls Amazon S3, AWS Glue, Amazon Athena and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the Resource field, and by passing Parameters to the service API.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
   "Comment": "An example demonstrates how to ingest a large data set in Amazon S3 and partition it through aws Glue Crawlers, then execute Amazon Athena queries against that partition."
   "StartAt": "Start Crawler",
   "States": {
      "Start Crawler": {
         "Type": "Task",
         "Next": "Get Crawler status",
         "Parameters": {
            "Name": "<GLUE_CRAWLER_NAME>"
         },
      },
      "Get Crawler status": {
         "Type": "Task",
         "Parameters": {
            "Name": "<GLUE_CRAWLER_NAME>"
         },
         "Next": "Check Crawler status"
      },
      "Check Crawler status": {
         "Type": "Choice",
         "Choices": [
            { "Variable": "$.Crawler.State",
              "StringEquals": "RUNNING",
              "Next": "Wait"
            }
         ],
         "Default": "Start an Athena query"
      },
      "Wait": {
         "Type": "Wait",
         "Seconds": 30,
         ... (continued)
IAM Examples

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

AthenaGetQueryResults
IAM Examples

AthenaStartQueryExecution

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "athena:startQueryExecution",
        "athena:stopQueryExecution",
        "athena:getQueryExecution",
        "athena:getDataCatalog"
      ],
      "Resource": [
        "arn:aws:athena:us-east-1:946551482283:datacatalog/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:ListBucketMultipartUploads",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload",
        "s3:CreateBucket",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "glue:CreateDatabase",
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:UpdateDatabase",
        "glue:DeleteDatabase",
        "glue:CreateTable",
        "glue:UpdateTable",
        "glue:GetTable",
        "glue:GetTables",
        "glue:DeleteTable",
        "glue:BatchDeleteTable",
        "glue:BatchCreatePartition",
        "glue:CreatePartition",
        "glue:UpdatePartition",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:BatchGetPartition",
        "glue:DeletePartition",
        "glue:BatchDeletePartition"
      ],
      "Resource": [
        "arn:aws:glue:us-east-1:946551482283:catalog",
        "arn:aws:glue:us-east-1:946551482283:database/*",
        "arn:aws:glue:us-east-1:946551482283:database/athena-sample-project-workgroup-8v7bshiv70"
      ]
    }
  ]
}
```
Keep data up to date (Amazon Athena, Amazon S3, AWS Glue)

This sample project demonstrates how to query a target table to get current data with AWS Glue Catalog, then update it with new data from other sources using Amazon Athena. Deploying this sample project creates an AWS Step Functions state machine, an Amazon S3 Bucket, Amazon Athena queries, and an AWS Glue Data Catalog call.

In this project, Step Functions uses a state machine to call AWS Glue Catalog to verify if a target table exists in an Amazon S3 Bucket, if no table is found one will be created. Then an Athena query will be run to add rows to the target table from a different data source: first querying the target table to get the most recent date, then querying the source table for more recent data and inserting it in to the target table.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects for Step Functions, and then choose Keep data up to date. The state machine Code and Visual Workflow are displayed.

3. Choose Next.

The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

- An AWS Glue Data Catalog call
- An Amazon S3 Bucket
- Amazon Athena queries

4. Choose Deploy Resources.

Note
It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.

Start a New Execution

1. On the New execution page, enter an execution name (optional), and then choose Start Execution.
2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name 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. (Optional) You can go to the newly created state machine on the Step Functions Dashboard, and then choose **New execution**.

4. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.

**Example State Machine Code**

The state machine in this sample project integrates with Amazon S3, AWS Glue, and Amazon Athena by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls Amazon S3, AWS Glue, and Amazon Athena by connecting to the Amazon Resource Name (ARN) in the **Resource** field, and by passing **Parameters** to the service API.

For more information about how AWS Step Functions can control other AWS services, see **Using AWS Step Functions with other services** (p. 271).

```json
{
  "Comment": "An example demonstrates how to use Athena to query a target table to get current data, then update it with new data from other sources.",
  "StartAt": "Get Target Table",
  "States": {
    "Get Target Table": {
      "Type": "Task",
      "Parameters": {
        "DatabaseName": "<GLUE_DATABASE_NAME>",
        "Name": "target"
      },
      "Catch": [
        {
          "ErrorEquals": [
            "Glue.EntityNotFoundException"
          ],
          "Next": "Create Target Table"
        }
      ],
      "Resource": "arn:aws:states:::aws-sdk:glue:getTable",
      "Next": "Update Target Table"
    },
    "Create Target Table": {
      "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
      "Parameters": {
        "QueryString": "<ATHENA_QUERYSTRING>",
        "WorkGroup": "<ATHENA_WORKGROUP>"
      },
      "Type": "Task",
      "Next": "Update Target Table"
    },
    "Update Target Table": {
      "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
      "Parameters": {
        "QueryString": "<ATHENA_QUERYSTRING>",
        "WorkGroup": "<ATHENA_WORKGROUP>"
      },
      "Type": "Task",
      "End": true
    }
  }
}
IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

AthenaStartQueryExecution

```
"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "athena:startQueryExecution",
      "athena:stopQueryExecution",
      "athena:getQueryExecution",
      "athena:getDataCatalog"
    ],
    "Resource": [
      "arn:aws:athena:us-east-1:946551482283:datasets/*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "s3:GetBucketLocation",
      "s3:GetObject",
      "s3:ListBucket",
      "s3:ListBucketMultipartUploads",
      "s3:ListMultipartUploadParts",
      "s3:AbortMultipartUpload",
      "s3:CreateBucket",
      "s3:PutObject"
    ],
    "Resource": [
      "arn:aws:s3:::*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "glue:CreateDatabase",
      "glue:GetDatabase",
      "glue:GetDatabases",
      "glue:UpdateDatabase",
      "glue:DeleteDatabase",
      "glue:CreateTable",
      "glue:GetTable",
      "glue:GetTables",
      "glue:DeleteTable",
      "glue:BatchDeleteTable",
      "glue:BatchCreateTable",
      "glue:BatchCreatePartition",
      "glue:CreatePartition",
      "glue:GetPartition",
      "glue:GetPartitions"
    ],
    "Resource": [
      "arn:aws:glue:us-east-1:946551482283:partition/*",
      "arn:aws:glue:us-east-1:946551482283:table/*",
      "arn:aws:glue:us-east-1:946551482283:partition/*"
    ]
  }
]
```
Manage an Amazon EKS cluster

This sample project demonstrates how to use Step Functions and Amazon Elastic Kubernetes Service to create an Amazon EKS cluster with a node group, run a job on Amazon EKS, then examine the output. When finished, it removes the node groups and Amazon EKS cluster. This sample project creates the following:

- An Amazon Elastic Kubernetes Service cluster
- An SNS topic
- Related AWS Identity and Access Management (IAM) roles

For more information about Step Functions and Step Functions service integrations, see the following:

- Using AWS Step Functions with other services (p. 271)
- Call Amazon EKS with Step Functions (p. 334)

**Note**

This sample project may incur charges. For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see Amazon EKS Pricing.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Run a sample project, and then choose Manage an Amazon EKS cluster.

The state machine Code and Visual Workflow are displayed.
3. Choose **Next**.

   The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

   - A state machine
   - An Amazon EKS cluster
   - An SNS topic
   - Related IAM roles

4. Choose **Deploy Resources**.

   **Note**
   
   It can take up to 25 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

**Start a New Execution**

1. Open the **Step Functions console**.
2. On the State machines page, choose the \texttt{EKSClusterManagementStateMachine} state machine that was created by the sample project, and then choose Start execution.

3. On the New execution page, enter an execution name (optional), and then choose Start Execution.

4. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

\textbf{Note}

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

5. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.

6. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

\section*{Example State Machine Code}

The state machine in this sample project integrates with Amazon EKS by creating an Amazon EKS cluster and node group, and uses an SNS topic to return results.

Browse through this example state machine to see how Step Functions manages Amazon EKS clusters and node groups.

For more information about how \textsc{AWS} Step Functions can control other \textsc{AWS} services, see Using \textsc{AWS} Step Functions with other services (p. 271).

\begin{verbatim}
{
"Comment": "An example of the Amazon States Language for running Amazon EKS Cluster",
"StartAt": "Create an EKS cluster",
"States": {
  "Create an EKS cluster": {
    "Type": "Task",
    "Resource": "arn:aws:states:::eks:createCluster.sync",
    "Parameters": {
      "Name": "ExampleCluster",
      "ResourcesVpcConfig": {
        "SubnetIds": ["subnet-0aacf887d9f00e6a7",
                      "subnet-0e5fc41e7507194ab"]
      },
      "RoleArn": "arn:aws:iam::111122223333:role/StepFunctionsSample-EKSClusterManagement-EKSServiceRole-ANPAJ2UCCR6DPCEXAMPLE"
    },
    "Retry": [{
      "ErrorEquals": ["States.ALL"],
      "IntervalSeconds": 30,
      "MaxAttempts": 2,
      "BackoffRate": 2
    }],
    "ResultPath": ".eks",
    "Next": "Create a node group"
  },
  "Create a node group": {
    "Type": "Task",
    "Resource": "arn:aws:states:::eks:createNodegroup.sync",
    "Parameters": {
      "ClusterName": "ExampleCluster",
\end{verbatim}
"NodegroupName": "ExampleNodegroup",
"NodeRole": "arn:aws:iam::111122223333:role/StepFunctionsSample-EKSClusterMan-
NodeInstanceRole-ANPAJ2UCCR6DPCEEXAMPLE",
"Subnets": [
  "subnet-0aacf887d9f00e6a7",
  "subnet-0e5fc41e7507194ab"
],
"Retry": [{
  "ErrorEquals": [ "States.ALL" ],
  "IntervalSeconds": 30,
  "MaxAttempts": 2,
  "BackoffRate": 2
}],
"ResultPath": ".nodegroup",
"Next": "Run a job on EKS"
},
"Run a job on EKS": {
  "Type": "Task",
  "Resource": "arn:aws:states:::eks:runJob.sync",
  "Parameters": {
    "ClusterName": "ExampleCluster",
    "Endpoint.$": ".eks.Cluster.Endpoint",
    "LogOptions": {
      "RetrieveLogs": true
    },
    "Job": {
      "apiVersion": "batch/v1",
      "kind": "Job",
      "metadata": {
        "name": "example-job"
      },
      "spec": {
        "backoffLimit": 0,
        "template": {
          "metadata": {
            "name": "example-job"
          },
          "spec": {
            "containers": [
            {
              "name": "pi-20",
              "image": "perl",
              "command": ["perl"],
              "args": ["-Mbignum=bpi","-wle",
                        "print '{ ' . '"pi": '. bpi(20) . ' }';"
                        ]
            }
            ],
            "restartPolicy": "Never"
          }
        }
      }
    },
    "ResultSelector": {
      "status.$": ".status",
      "logs.$": ".logs.pi"
    },
    "ResultPath": ".RunJobResult",
    "Next": "Examine output"
  }
}
"Examine output": {
  "Type": "Choice",
  "Choices": [
    {
      "And": [
        {
          "Variable": "$.RunJobResult.logs[0]",
          "NumericGreaterThan": 3.141
        },
        {
          "Variable": "$.RunJobResult.logs[0]",
          "NumericLessThan": 3.142
        }
      ],
      "Next": "Send expected result"
    },
    {
      "Default": "Send unexpected result"
    }
  ],
  "Send expected result": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish",
    "Parameters": {
      "Message": {
        "Input.$": "States.Format('Saw expected value for pi: {}',
        $.RunJobResult.logs[0])"
      },
      "ResultPath": "$.SNSResult",
      "Next": "Delete job"
    }
  },
  "Send unexpected result": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sns:publish",
    "Parameters": {
      "Message": {
        "Input.$": "States.Format('Saw unexpected value for pi: {}',
        $.RunJobResult.logs[0])"
      },
      "ResultPath": "$.SNSResult",
      "Next": "Delete job"
    }
  },
  "Delete job": {
    "Type": "Task",
    "Resource": "arn:aws:states:::eks:call",
    "Parameters": {
      "ClusterName": "ExampleCluster",
      "Endpoint.$": ".eks.Cluster.Endpoint",
      "Method": "DELETE",
      "Path": "/apis/batch/v1/namespaces/default/jobs/example-job"
    },
    "ResultSelector": {
      "status.$": ".ResponseBody.status"
    },
    "ResultPath": "$.DeleteJobResult",
    "Next": "Delete node group"
  },
  "Delete node group": {
    "Type": "Task",
    "Resource": "arn:aws:states:::eks:deleteNodegroup.sync",
    "Parameters": {
      "ClusterName": "ExampleCluster",
      "Namespace": "default",
      "JobName": "example-job"
    },
    "ResultSelector": {
      "status.$": ".ResponseBody.status"
    },
    "ResultPath": "$.DeleteNodeGroupResult",
    "Next": "Delete job"
  }
}
For information about how to configure IAM when using Step Functions with other AWS services, see IAM P...
Make a call to API Gateway

This sample project demonstrates how to use Step Functions to make a call to API Gateway and checks whether the call succeeded. This sample project creates the following:

- An Amazon API Gateway REST API that is called by the state machine.
- Related AWS Identity and Access Management (IAM) roles.

For more information about API Gateway and Step Functions service integrations, see the following:

- Using AWS Step Functions with other services (p. 271)
- Call API Gateway with Step Functions (p. 343)

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create state machine.
2. Choose Run a sample project, and then choose Make a call to API Gateway.
   The state machine Code and Visual Workflow are displayed.
3. Choose **Next**.

The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

- A state machine
- A call to the API Gateway REST API
- Related IAM roles

4. Choose **Deploy Resources**.

**Note**

It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.
Start a New Execution

1. Open the Step Functions console.
2. On the State machines page, choose the ApiGatewayWorkflowStateMachine state machine that was created by the sample project, and then choose Start execution.
3. On the New execution page, enter an execution name (optional), and then choose Start Execution.
4. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

5. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.
6. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example State Machine Code

The state machine in this sample project integrates with API Gateway by calling the API Gateway REST API and passing any necessary parameters.

Browse through this example state machine to see how Step Functions interacts with API Gateway.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
  "Comment": "Calling APIGW REST Endpoint",
  "StartAt": "Add Pet to Store",
  "States": {
    "Add Pet to Store": {
      "Type": "Task",
      "Resource": "arn:aws:states:::apigateway:invoke",
      "Parameters": {
        "ApiEndpoint": "<POST_PETS_API_ENDPOINT>",
        "Method": "POST",
        "Stage": "default",
        "Path": "pets",
        "RequestBody.$": "$\cdot NewPet",
        "AuthType": "IAM_ROLE"
      },
      "ResultSelector": {
        "ResponseBody.$": "$\cdot ResponseBody"
      },
      "Next": "Pet was Added Successfully?"
    },
    "Pet was Added Successfully?": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$\cdot ResponseBody.errors",
          "IsPresent": true,
          "Next": "Failure"
        }
      ]
    }
  }
}
```
IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-west-1:111122223333:c8hqe4kd5/default/POST/pets"
      ],
      "Effect": "Allow"
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

Call a microservice running on Fargate using API Gateway integration

This sample project demonstrates how to use Step Functions to make a call to API Gateway in order to interact with a service on AWS Fargate, and also to check whether the call succeeded. This sample project creates the following:
• An Amazon API Gateway HTTP API that is called by the state machine.
• An Amazon API Gateway Amazon VPC Link.
• An Amazon Virtual Private Cloud.
• An Application Load Balancer.
• A Fargate cluster.
• An Amazon SNS topic
• Related AWS Identity and Access Management (IAM) roles
• Several additional services that are required to enable these resources to work together.

For more information about API Gateway and Step Functions service integrations, see the following:
• Using AWS Step Functions with other services (p. 271)
• Call API Gateway with Step Functions (p. 343)

**Note**
This sample project may incur charges.
For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see Pricing.

**Create the State Machine and Provision Resources**

1. Open the Step Functions console and choose **Create a state machine**.
2. Choose **Sample Projects**, and then choose **Call a microservice with API Gateway**.

   The state machine **Code** and **Visual Workflow** are displayed.
3. Choose **Next**.

The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

- An API Gateway HTTP API
- An API Gateway VpcLink
- An Application Load Balancer
- A Fargate cluster
- A state machine
- An Amazon SNS topic
- Related IAM roles
- Several additional services that are required to enable these resources to work together.
4. Choose **Deploy Resources**.
   
   **Note**
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.

**Start a New Execution**

1. Open the **Step Functions** console.
2. On the **State machines** page, choose the **ApiGatewayECSStateMachine** state machine that was created by the sample project, and then choose **Start execution**.
3. On the **New execution** page, enter an execution name (optional), and then choose **Start Execution**.
4. (Optional) To identify your execution, you can specify a name for it in the **Name** box. By default, Step Functions generates a unique execution name automatically.
   
   **Note**
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.
5. (Optional) Go to the newly created state machine on the Step Functions **Dashboard**, and then choose **New execution**.
6. When an execution is complete, you can select states on the **Visual workflow** and browse the **Input** and **Output** under **Step details**.

**Example State Machine Code**

The state machine in this sample project integrates with API Gateway by calling an API Gateway HTTP API that is connected to a service on Fargate. This is hosted on a private subnet, and accessed through a private application load balancer.

Browse through this example state machine to see how Step Functions interacts with API Gateway and returns results.

For more information about how AWS Step Functions can control other AWS services, see **Using AWS Step Functions with other services** (p. 271).

```json
{
  "Comment": "Calling APIGW HTTP Endpoint",
  "StartAt": "Call API",
  "States": {
    "Call API": {
      "Type": "Task",
      "Resource": "arn:<PARTITION>:states::apigateway:invoke",
      "Parameters": {
        "ApiEndpoint": "<API_ENDPOINT>",
        "Method": "GET",
        "AuthType": "IAM_ROLE"
      },
      "Next": "Call Successful?"
    },
    "Call Successful?": {
      "Type": "Choice",
      "Choices": [
        { "Variable": "$.StatusCode",
```
"NumericEquals": 200,
  "Next": "Notify Success"
},
  "Default": "Notify Failure"
},
  "Notify Success": {
    "Type": "Task",
    "Resource": "arn:<PARTITION>:states::sns:publish",
    "Parameters": {
      "Message": "Call was successful",
      "TopicArn": "<SNS_TOPIC_ARN>"
    },
    "End": true
  },
  "Notify Failure": {
    "Type": "Task",
    "Resource": "arn:<PARTITION>:states::sns:publish",
    "Parameters": {
      "Message": "Call was not successful",
      "TopicArn": "<SNS_TOPIC_ARN>"
    },
    "End": true
  }
}

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

### IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "sns:Publish"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
      ],
      "Effect": "Allow"
    }
  ]
}
```
Send a custom event to EventBridge

This sample project demonstrates how to use Step Functions to send a custom event to an event bus that matches a rule with multiple targets (Amazon EventBridge, AWS Lambda, Amazon Simple Notification Service, Amazon Simple Queue Service). This sample project creates the following:

- Amazon EventBridge
- An Amazon SNS topic
- An Amazon SQS queue
- A Lambda function
- Related AWS Identity and Access Management (IAM) roles

For more information about Step Functions and Step Functions service integrations, see the following:

- Using AWS Step Functions with other services (p. 271)
• Call EventBridge with Step Functions (p. 349)

Note
This sample project may incur charges. For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see EventBridge Pricing.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Run a sample project, and then choose Send a custom event to EventBridge.

The state machine Code and Visual Workflow are displayed.

3. Choose Next.

The Deploy resources page is displayed, listing the resources that will be created. For this sample project, the resources include:

• An AWS Step Functions state machine
• An EventBridge event bus
• An EventBridge rule
• An Amazon SNS topic
• An Amazon SQS queue
• A Lambda Function
• Related IAM roles

4. Choose Deploy Resources.

Note
It can take up to 25 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.
Start a New Execution

1. Open the Step Functions console.
2. On the State machines page, choose the EventBridgeStateMachine state machine that was created by the sample project, and then choose Start execution.
3. On the New execution page, enter an execution name (optional), and then choose Start Execution.
4. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

5. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.
6. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example State Machine Code

The state machine in this sample project integrates with EventBridge by sending a custom event to an EventBridge event bus. The event sent to the event bus matches an EventBridge rule that triggers a Lambda function that sends messages to an Amazon SNS topic and an Amazon SQS queue.

Browse through this example state machine to see how Step Functions manages EventBridge.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
   "Comment": "An example of the Amazon States Language for sending a custom event to Amazon EventBridge",
   "StartAt": "Send a custom event",
   "States": {
      "Send a custom event": {
         "Resource": "arn:<PARTITION>:states:::events:putEvents",
         "Type": "Task",
         "Parameters": {
            "Entries": [{
               "Detail": {
                  "Message": "Hello from Step Functions!"
               },
               "EventType": "MessageFromStepFunctions",
               "EventBusName": "<EVENT_BUS_NAME>",
               "Source": "my.statemachine"
            }
         ]
      },
      "End": true
   }
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": [
            "events:PutEvents"
         ],
         "Resource": [
            "arn:aws:events:us-east-1:1234567890:event-bus/stepfunctions-sampleproject-eventbus"
         ],
         "Effect": "Allow"
      }
   ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

Invoke Synchronous Express Workflows

This sample project demonstrates how to invoke Synchronous Express Workflows through Amazon API Gateway to manage an employee database. This sample project creates the following:

- Three state machines.
- An Amazon API Gateway HTTPS API that is called by a state machine.
- An Amazon DynamoDB table.
- Related AWS Identity and Access Management (IAM) roles

In this project, Step Functions uses API Gateway endpoints to start Step Functions Synchronous Express Workflows. These then use DynamoDB to search for, add, and remove employees in an employee database.

For more information about Step Functions Synchronous Express Workflows, see Synchronous and Asynchronous Express Workflows (p. 20).

Note
This sample project may incur charges.
For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see Step Functions Pricing.

Create the State Machine and Provision Resources

1. Open the Step Functions console and choose Create a state machine.
2. Choose Sample Projects, and then choose Invoke Synchronous Express Workflows through API Gateway.
The state machine **Code** and **Visual Workflow** are displayed.

3. Choose **Next**.

The **Deploy resources** page is displayed, listing the resources that will be created. For this sample project, the resources include:

- Three state machines
- A DynamoDB table
- An API Gateway HTTPS API
- Related IAM roles

4. Choose **Deploy Resources**.

**Note**

It can take up to 10 minutes for these resources and related IAM permissions to be created. While the **Deploy resources** page is displayed, you can open the **Stack ID** link to see which resources are being provisioned.
Start a New Execution

1. Open the Step Functions console.
2. On the State machines page, choose the state machine that was created by the sample project, and then choose Start execution.
3. On the New execution page, enter an execution name (optional), and then choose Start Execution.
4. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

5. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.
6. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example State Machine Code

The state machine in this sample project integrates with API Gateway and DynamoDB by using API Gateway to invoke a Synchronous Express Workflow, which then updates or reads from the employee database using DynamoDB.

Browse through this example state machine to see how Step Functions reads from DynamoDB to retrieve employee information.

To understand more about how to invoke Step Functions using API Gateway, see the following.

- Call API Gateway with Step Functions (p. 343)
- How to invoke a private Gateway in the API Gateway Developer Guide.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
  "Comment": "This state machine returns an employee entry from DynamoDB",
  "StartAt": "Read From DynamoDB",
  "States": {
    "Read From DynamoDB": {
      "Type": "Task",
      "Resource": "arn:aws:states:::dynamodb:getItem",
      "Parameters": {
        "TableName": "StepFunctionsSample-SynchronousExpressWorkflowAKIAIOSFODNN7EXAMPLE-DynamoDBTable-ANPAJ2UCCR6DFCEEXAMPLE",
        "Key": {
          "EmployeeId": {"S.$": ".employee"}
        }
      },
      "Retry": [
        {"ErrorEquals": ["DynamoDB.AmazonDynamoDBException"]}
      ],
      "IntervalSeconds": 3,
```
For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

IAM Examples

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogDelivery",
                "logs:GetLogDelivery",
                "logs:UpdateLogDelivery",
                "logs:DeleteLogDelivery",
                "logs:ListLogDeliveries",
                "logs:PutResourcePolicy",
                "logs:DescribeResourcePolicies",
                "logs:DescribeLogGroups"
            ],
            "Resource": "*"
        }
    ]
}
```
Run ETL/ELT workflows using Amazon Redshift (Lambda, Amazon Redshift Data API)

This sample project demonstrates how to use Step Functions and the Amazon Redshift Data API to run an ETL/ELT workflow that loads data into the Amazon Redshift data warehouse. When you deploy this sample project, you create the following:

- An Amazon Redshift cluster
- A state machine
- Two lambda functions
- An Amazon Redshift schema
- Five Amazon Redshift tables
- Related AWS Identity and Access Management (IAM) roles.

In this project, Step Functions uses an AWS Lambda function and the Amazon Redshift Data API to create the required database objects and to generate a set of example data, then executes two jobs in parallel that perform loading dimension tables, followed by a fact table. Once both dimension load jobs end successfully, Step Functions executes the load job for the fact table, runs the validation job, then pauses the Amazon Redshift cluster.

**Note**
You can modify the ETL logic to receive data from other sources such as Amazon S3, which can use the COPY command to copy data from Amazon S3 to an Amazon Redshift table.

For more information about Amazon Redshift and Step Functions service integrations, see the following:

- Using AWS Step Functions with other services (p. 271)
- Using the Amazon Redshift Data API
- Amazon Redshift Data API service
- Creating a Step Functions State Machine That Uses Lambda (p. 137)
- IAM policies for:
  - AWS Lambda (p. 533)
• Authorizing access to the Amazon Redshift Data API

**Note**
This sample project may incur charges. For new AWS users, a free usage tier is available. On this tier, services are free below a certain level of usage. For more information about AWS costs and the Free Tier, see AWS Step Functions pricing.

**Create the State Machine and Provision Resources**

1. Open the Step Functions console and choose *Create a state machine*.
2. Choose *Sample Projects*, and then choose *ETL job in Amazon Redshift*.
   
The state machine *Code* and *Visual Workflow* display. The following figure shows the beginning of the training model workflow. For a diagram of the full training model workflow, see the full Run ETL/ELT Workflows diagram.

3. Choose *Next*.

   The *Deploy resources* page displays, listing the resources that will be created. For this sample project, the resources include:

   • An ETL state machine
   • A RedshiftOperations Lambda function
   • A RedshiftDataAPI Lambda function
   • A Amazon Redshift cluster
   • Related IAM roles

4. Choose *Deploy Resources*.
Start a New Execution

1. Open the Step Functions console.
2. On the State machines page, choose the ETL job in Amazon Redshift state machine that was created by the sample project, and then choose Start execution.
3. On the New execution page, enter an execution name (optional), and then choose Start Execution.
4. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.
   
   **Note**
   Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters.

5. (Optional) Go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.
6. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example State Machine Code

The state machine in this sample project integrates with AWS Lambda by passing the ETL logic as the InputPath directly to those resources and being executed asynchronously using Amazon Redshift Data API.

Browse through this example state machine to see how Step Functions controls AWS Lambda and the Amazon Redshift Data API.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
"Comment": "A simple ETL workflow for loading dimension and fact tables",
"StartAt": "InitializeCheckCluster",
"States": {
  "InitializeCheckCluster": {
    "Type": "Pass",
    "Next": "GetStateOfCluster",
    "Result": {
      "input": {
        "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
        "operation": "status"
      }
      },
  "GetStateOfCluster": {
    "Type": "Task",
    "TimeoutSeconds": 180,
    "TimeoutSeconds": 180,
```
"HeartbeatSeconds": 60,
"Next": "IsClusterAvailable",
"InputPath": ",
"ResultPath": ",
"IsClusterAvailable": {
"Type": "Choice",
"Choices": [
  {
    "Variable": ",
    "StringEquals": "available",
    "Next": "InitializeBuildDB"
  },
  {
    "Variable": ",
    "StringEquals": "paused",
    "Next": "InitializeResumeCluster"
  },
  {
    "Variable": ",
    "StringEquals": "unavailable",
    "Next": "ClusterUnavailable"
  },
  {
    "Variable": ",
    "StringEquals": "resuming",
    "Next": "ClusterWait"
  }
],
"ClusterWait": {
"Type": "Wait",
"Seconds": 720,
"Next": "InitializeCheckCluster"
},
"InitializeResumeCluster": {
"Type": "Pass",
"Next": "ResumeCluster",
"Result": {
  "input": {
    "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
    "operation": "resume"
  }
},
"ResumeCluster": {
"Type": "Task",
"TimeoutSeconds": 180,
"HeartbeatSeconds": 60,
"Next": "ClusterWait",
"InputPath": ",
"ResultPath": ",
"InitializeBuildDB": {
"Type": "Pass",
"Next": "BuildDB",
"Result": {
  "input": {
    "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
    "redshift_database": "dev",
    "redshift_user": "awsuser",
    "redshift_schema": "tpcds",
    "action": "build_database",
    "sql_statement": []
  }
}
"create schema if not exists {0} authorization {1};",
"create table if not exists {0}.customer",
"(c_customer_sk        int4     not null encode az64",
",c_customer_id         char(16) not null encode zstd",
",c_current_addr_sk     int4     encode az64",
",c_first_name          char(20)  encode zstd",
",c_last_name           char(30)  encode zstd",
") primary key (c_customer_sk);",
") distkey(c_customer_sk);",

"create table if not exists {0}.customer_address",
"(ca_address_sk        int4     not null encode az64",
",ca_address_id         char(16) not null encode zstd",
",ca_state              char(2)    encode zstd",
",ca_zip                 char(10)   encode zstd",
") primary key (ca_address_sk);",
") distkey(ca_address_sk);",

"create table if not exists {0}.date_dim",
"(d_date_sk           integer  not null encode az64",
",d_date_id            char(16) not null encode zstd",
",d_date               date              encode az64",
",d_day_name           char(9)     encode zstd",
") primary key (d_date_sk);",
") diststyle all;",

"create table if not exists {0}.item",
"(i_item_sk        int4     not null encode az64",
",i_item_id          char(16) not null encode zstd",
",i_rec_start_date   date              encode az64",
",i_rec_end_date     date              encode az64",
",i_current_price    numeric(7,2)      encode az64",
") primary key (i_item_sk);",
") distkey(i_item_sk) sortkey(i_category);",

"create table if not exists {0}.store_sales",
"(ss_sold_date_sk       int4",
",ss_item_sk            int4     not null encode az64",
",ss_customer_sk        int4     not null encode az64",
",ss_addr_sk            int4     encode az64",
",ss_store_sk           int4     encode az64",
") primary key (ss_item_sk, ss_ticket_number);",
") distkey(ss_item_sk) sortkey(ss_sold_date_sk);"
"Next": "CheckBuildDBStatus",
"TimeoutSeconds": 180,
"HeartbeatSeconds": 60,
"InputPath": "@",
"ResultPath": "$.status"
},
"CheckBuildDBStatus": {
  "Type": "Choice",
  "Choices": [
    {
      "Variable": "$.status",
      "StringEquals": "FAILED",
      "Next": "FailBuildDB"
    },
    {
      "Variable": "$.status",
      "StringEquals": "FINISHED",
      "Next": "InitializeBaselineData"
    }
  ],
  "Default": "BuildDBWait"
},
"BuildDBWait": {
  "Type": "Wait",
  "Seconds": 15,
  "Next": "GetBuildDBStatus"
},
"FailBuildDB": {
  "Type": "Fail",
  "Cause": "Database Build Failed",
  "Error": "Error"
},
"InitializeBaselineData": {
  "Type": "Pass",
  "Next": "LoadBaselineData",
  "Result": {
    "input": {
      "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44Q8DHBLEXAMPLE",
      "redshift_database": "dev",
      "redshift_user": "awsuser",
      "redshift_schema": "tpcds",
      "action": "load_baseline_data",
      "sql_statement": [
        "begin transaction;",
        "truncate table {0}.customer;",
        "insert into {0}.customer (c_customer_sk,c_customer_id,c_current_addr_sk,c_first_name,c_last_name)",
        "values",
        "("7550,'AAAAAAAAOHNBAAAA',9264662,'Michelle','Deaton'),",
        "("37079,'AAAAAAAAHHNMAAAAA',13971208,'Michael','Simms'),",
        "("40626,'AAAAAAAAACLOJAAA',1959255,'Susan','Ryder'),",
        "("2142876,'AAAAAAAAAMJCLACAA',7644556,'Justin','Brown');",
        "analyze {0}.customer;",
        "--",
        "truncate table {0}.customer_address;",
        "insert into {0}.customer_address (ca_address_sk,ca_address_id,ca_state,ca_zip,ca_country)",
        "values",
        "("13971208,'AAAAAAAAAPCFNAA','NE','63451','United States'),",
        "("7644556,'AAAAAAAAAMIFKEHAA','SD','58883','United States'),",
        "("9264662,'AAAAAAAAAGBOFNIAA','CA','99310','United States');",
        "analyze {0}.customer_address;",
        "--",
        "truncate table {0}.item;"
    }
  }
}
"insert into {0}.item
(i_item_sk,i_item_id,i_rec_start_date,i_rec_end_date,i_current_price,i_category,i_product_name)
values
(3417,'AAAAAAAIFNAAAAA','1997-10-27',NULL,14.29,'Electronics','ationoughtesepri '),
(9615,'AAAAAAA0IFCAAAAA','1997-10-27',NULL,9.68,'Home','antioughtcallyn st'),
(3693,'AAAAAAAAMGOAAAAA','2001-03-12',NULL,2.10,'Men','prin stcallypr'),
(3630,'AAAAAAAAMCOAAAAA','2001-10-27',NULL,2.95,'Electronics','barpricallypri'),
(16506,'AAAAAAAIAHAEAAAAA','2001-10-27',NULL,3.85,'Home','callybaranticallyought'),
(7866,'AAAAAAILOBAAAAA','2001-10-27',NULL,12.60,'Jewelry','callycallyeingation');",
"analyze {0}.item;",
"truncate table {0}.date_dim;
insert into {0}.date_dim (d_date_sk,d_date_id,d_date,d_day_name)
values
(2450521,'AAAAAAAAJFEGFCAA','1997-03-13','Thursday'),
(2450749,'AAAAAAAANDFGFCAA','1997-10-27','Monday'),
(2451251,'AAAAAAAADDHGFCAA','1997-10-27','Monday'),
(2451252,'AAAAAAAADDHGFCAA','1999-03-13','Saturday'),
(2451981,'AAAAAAAANAKGFCAA','2001-03-12','Monday'),
(2451982,'AAAAAAAANAKGFCAA','2001-03-13','Tuesday'),
(2452210,'AAAAAAAACPKGFCAA','2001-10-27','Saturday'),
(2452641,'AAAAAAABRMGFCAA','2003-01-01','Wednesday'),
(2452642,'AAAAAAACRMGFCAA','2003-01-02','Thursday');",
"analyze {0}.date_dim;",
"-- commit and End transaction",
"commit;
end transaction;"
}
"Variable": ".status",
"StringEquals": "FINISHED",
"Next": "ParallelizeDimensionLoadJob"
]
],
"Default": "BaselineDataWait"
],
"BaselineDataWait": {
"Type": "Wait",
"Seconds": 20,
"Next": "GetBaselineData"
],
"FailLoadBaselineData": {
"Type": "Fail",
"Cause": "Load Baseline Data Failed",
"Error": "Error"
],
"ParallelizeDimensionLoadJob": {
"Type": "Parallel",
"Next": "InitializeSalesFactLoadJob",
"ResultPath": ".status",
"Branches": [
"StartAt": "InitializeCustomerAddressDimensionLoadJob",
"States": {
"InitializeCustomerAddressDimensionLoadJob": {
"Type": "Pass",
"Next": "ExecuteCustomerAddressDimensionLoadJob",
"Result": {
"input": {
"redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
"redshift_database": "dev",
"redshift_user": "awsuser",
"redshift_schema": "tpcds",
"action": "load_customer_address",
"sql_statement": [
"begin transaction;
/* Create a staging table to hold the input data. Staging table is created with BACKUP NO option for faster inserts and also data temporary */;
"drop table if exists {0}.stg_customer_address;
"create table if not exists {0}.stg_customer_address,
"(ca_address_id varchar(16) encode zstd",
",ca_state varchar(2) encode zstd",
",ca_zip varchar(10) encode zstd",
",ca_country varchar(20) encode zstd",
")",
"backup no",
"diststyle even;
/* Ingest data from source */",
"insert into {0}.stg_customer_address
(ca_address_id,ca_state,ca_zip,ca_country)
values
('AAAAAAAACFBBAAAA','NE','','United States'),
('AAAAAAAAGAEFAAAA','NE','61749','United States'),
('AAAAAAAAAOKIIIIAAAA','OK','','United States'),
/* Perform UPDATE for existing data with refreshed attribute values */
","update {0}.customer_address",
"set ca_state = stg_customer_address.ca_state,",
"ca_zip = stg_customer_address.ca_zip",
"ca_country = stg_customer_address.ca_country",
"from {0}.stg_customer_address",
"where customer_address.ca_address_id =
stg_customer_address.ca_address_id;
",
"/* Perform insert for new rows */
"*/
"]
}
"insert into {0}.customer_address",
"(ca_address_sk",
",ca_address_id",
",ca_state",
",ca_zip",
",ca_country",
")",
"with max_customer_address_sk as",
"(select max(ca_address_sk) max_ca_address_sk",
"from {0}.customer_address)",
"select row_number() over (order by stg_customer_address.ca_address_id)
+ max_customer_address_sk.max_ca_address_sk as ca_address_sk",
"stg_customer_address.ca_address_id",
"stg_customer_address.ca_state",
"stg_customer_address.ca_zip",
"stg_customer_address.ca_country",
"from {0}.stg_customer_address,",
"max_customer_address_sk",
"where stg_customer_address.ca_address_id not in (select
customer_address.ca_address_id from {0}.customer_address);
",
"/* Commit and End transaction */",
"commit;",
"end transaction;"}
"Next": "GetCustomerAddressDimensionLoadStatus"
},
"CompleteCustomerAddressDimensionLoad": {
"Type": "Task",
"TimeoutSeconds": 180,
"HeartbeatSeconds": 60,
"End": true
},
"FailCustomerAddressDimensionLoad": {
"Type": "Fail",
"Cause": "ETL Workflow Failed",
"Error": "Error"
}
},
{
"StartAt": "InitializeItemDimensionLoadJob",
"States": {
"InitializeItemDimensionLoadJob": {
"Type": "Pass",
"Next": "ExecuteItemDimensionLoadJob",
"Result": {
"input": {
"redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEEXAMPLE",
"redshift_database": "dev",
"redshift_user": "awsuser",
"redshift_schema": "tpcds",
"action": "load_item",
"sql_statement": [
"begin transaction;
/* Create a staging table to hold the input data. Staging table is created with BACKUP NO option for faster inserts and also data temporary */",
"drop table if exists {0}.stg_item;",
"create table if not exists {0}.stg_item",
"(i_item_id varchar(16) encode zstd",
","i_rec_start_date date encode zstd",
","i_rec_end_date date encode zstd",
","i_current_price numeric(7,2) encode zstd",
","
"i_category varchar(50) encode zstd",
","
"i_product_name varchar(50) encode zstd",
")",
"insert into {0}.stg_item",
"("i_item_id,i_rec_start_date,i_rec_end_date,i_current_price,i_category,i_product_name")",
"values",
"("AAAAAABJBAAAA",'2000-10-27',NULL,4.10,'Books','ationoughtesecally'),",
"("AAAAAIAOPKBAAAAA",'2001-10-27',NULL,4.22,'Books','ableoughtnstcally'),",
"("AAAAAAAHGPAAAAA",'1997-10-27',NULL,29.30,'Books','pricesenstpri'),",
"("AAAAAAAAPKBAAAAA",'2001-10-27',NULL,1.93,'Books','eseoughtoughtpri'),",
"("AAAAAAAAGPGBAAAA",'2001-10-27',NULL,9.96,'Books','arereingangi'),",
"("AAAAAAAANESEBAAAAA",'1997-10-27',NULL,2.25,'Music','nseoughtanti'),",
"("AAAAAAACLAAAAAA",'2001-10-27',NULL,1.71,'Home','bareingought'),",
"("AAAAAAAOBBDDAAAA",'2001-10-27',NULL,5.55,'Books','callyationantiumableought');")",
"("AAAAAAABJDBAAAA",'2000-10-27',NULL,4.10,'Books','ationoughtesecally')",
"("AAAAAAAOGBAAAA",'2001-10-27',NULL,4.22,'Books','ableoughtnstcally'),",
"("AAAAAAAHGBPAAAA",'1997-10-27',NULL,29.30,'Books','pricesenstpri'),",
"("AAAAAAAAPGBAAAA",'2001-10-27',NULL,1.93,'Books','eseoughtoughtpri'),",
"("AAAAAAAAGPBAAAA",'2001-10-27',NULL,9.96,'Books','arereingangi'),",
"("AAAAAAAANESEBAAAA",'1997-10-27',NULL,2.25,'Music','nseoughtanti'),",
"("AAAAAAACLAAAA",'2001-10-27',NULL,1.71,'Home','bareingought'),",
"("AAAAAAAOBBDDAAAA",'2001-10-27',NULL,5.55,'Books','callyationantiumableought');")"}
"*/
#####################################################################################################
** Type 2 is maintained for i_current_price column.
** Update all attributes for the item when the price is not changed,
** Sunset existing active item record with current i_rec_end_date and
insert a new record when the price does not match,
#####################################################################################################*/
update {0}.item,
   set i_category = stg_item.i_category,
       i_product_name = stg_item.i_product_name,
   from {0}.stg_item,
   where item.i_item_id = stg_item.i_item_id,
   and item.i_rec_end_date is null,
   and item.i_current_price = stg_item.i_current_price;
insert into {0}.item,
   (i_item_sk,
    i_item_id,
    i_rec_start_date,
    i_rec_end_date,
    i_current_price,
    i_category,
    i_product_name,
    )
  with max_item_sk as",
   (select max(i_item_sk) max_item_sk,
    from {0}.item,
    select row_number() over (order by stg_item.i_item_id) + max_item_sk
as i_item_sk,
    stg_item.i_item_id,
    trunc(sysdate) as i_rec_start_date,
    null as i_rec_end_date,
    stg_item.i_current_price,
    stg_item.i_category,
    stg_item.i_product_name,
   from {0}.stg_item, {0}.item, max_item_sk,
   where item.i_item_id = stg_item.i_item_id,
   and item.i_rec_end_date is null,
   and item.i_current_price <> stg_item.i_current_price;
   /* Sunset penultimate records that were inserted as type 2 */
   update {0}.item,
   set i_rec_end_date = trunc(sysdate),
   from {0}.stg_item,
   where item.i_item_id = stg_item.i_item_id,
   and item.i_rec_end_date is null,
   and item.i_current_price <> stg_item.i_current_price;
   /* Commit and End transaction */
   commit;
   end transaction;
}
"TimeoutSeconds": 180,
"HeartbeatSeconds": 60,
"InputPath": ":",
"ResultPath": "$.status"
},
"CheckItemDimensionLoadStatus": {
"Type": "Choice",
"Choices": [
  {
    "Variable": "$.status",
    "StringEquals": "FAILED",
    "Next": "FailItemDimensionLoad"
  },
  {
    "Variable": "$.status",
    "StringEquals": "FINISHED",
    "Next": "CompleteItemDimensionLoad"
  }
],
"Default": "ItemWait"
},
"ItemWait": {
"Type": "Wait",
"Seconds": 5,
"Next": "GetItemDimensionLoadStatus"
},
"CompleteItemDimensionLoad": {
"Type": "Task",
"TimeoutSeconds": 180,
"HeartbeatSeconds": 60,
"End": true
},
"FailItemDimensionLoad": {
"Type": "Fail",
"Cause": "ETL Workflow Failed",
"Error": "Error"
}
}
],
"InitializeSalesFactLoadJob": {
"Type": "Pass",
"Next": "ExecuteSalesFactLoadJob",
"Result": {
"input": {
  "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
  "redshift_database": "dev",
  "redshift_user": "awsuser",
  "redshift_schema": "tpcds",
  "snapshot_date": "2003-01-02",
  "action": "load_sales_fact",
  "sql_statement": [
    "begin transaction;",
    "/* Create a stg_store_sales staging table */",
    "drop table if exists {0}.stg_store_sales;",
    "create table {0}.stg_store_sales",
    "(sold_date             date encode zstd",
    ",i_item_id             varchar(16) encode zstd",
    ",c_customer_id         varchar(16) encode zstd",
    ",ca_address_id         varchar(16) encode zstd",
    ",ss_ticket_number      integer encode zstd",
    ",ss_sales_channel_id   varchar(16) encode zstd",
    ",s_store_id            varchar(16) encode zstd",
    ",s_store_city          varchar(16) encode zstd",
    ",s_store_state         varchar(2) encode zstd",
    ",s_store_zip           varchar(9) encode zstd",
    ",s_data_provider      varchar(16) encode zstd",
    ",c_customer_key   integer encode zstd",
    ",i_item_key           integer encode zstd",
    ",ss_ticket_number      integer encode zstd",
    ");"
  ]
}
},
"ExecuteSalesFactLoadJob": {
"Type": "Task",
"TimeoutSeconds": 180,
"HeartbeatSeconds": 60,
"InputPath": "$.input",
"ResultPath": "$.result"
}
}
"ss_quantity integer encode zstd",
"ss_net_paid numeric(7,2) encode zstd",
"ss_net_profit numeric(7,2) encode zstd",
")",
"backup no",
"diststyle even;",
"/* Ingest data from source */",
"insert into {0}.stg_store_sales",
"("sold_date,i_item_id,c_customer_id,ca_address_id,ss_ticket_number,ss_quantity,ss_net_paid,ss_net_profit",
"values",
"('2003-01-02','AAAAAAAAIFNAAAAA','AAAAAAAAOHNBAAAA','AAAAAAAAGBOFN1IAA',1403191,13,5046.37,150.97),",
"('2003-01-02','AAAAAAAAAIFNAAAAA','AAAAAAAAOHNBAAAA','AAAAAAAAGBOFN1IAA',1403191,13,2103.72,-124.08),",
"('2003-01-02','AAAAAAAAAILBAAAA','AAAAAAAAAHHN1JAAAA','AAAAAAAAGBOFN1IAA',1403191,13,959.10,-1304.70),",
"('2003-01-02','AAAAAAAAAILBAAAA','AAAAAAAAAHHN1JAAAA','AAAAAAAAGBOFN1IAA',1403191,13,962.65,475.80),",
"('2003-01-02','AAAAAAAAAMCOAAAAA','AAAAAAAAAHHN1JAAAA','AAAAAAAAGBOFN1IAA',1201746,17,111.60,-241.65),",
"('2003-01-02','AAAAAAAAAMCOAAAAA','AAAAAAAAAHHN1JAAAA','AAAAAAAAGBOFN1IAA',1201746,17,4013.02,-1111.48),",
"('2003-01-02','AAAAAAAAAMCOAAAAA','AAAAAAAAAHHN1JAAAA','AAAAAAAAGBOFN1IAA',193971,18,1876.89,-556.35),",
"/* Delete any rows from target store_sales for the input date for idempotency */",
"delete from {0}.store_sales where ss_sold_date_sk in (select d_date_sk from {0}.date_dim where d_date='{1}');",
"/* Insert data from staging table to the target table */",
"insert into {0}.store_sales",
"(ss_sold_date_sk",
"      ,ss_item_sk",
"      ,ss_customer_sk",
"      ,ss_addr_sk",
"      ,ss_ticket_number",
"      ,ss_quantity",
"      ,ss_net_paid",
"      ,ss_net_profit",
")",
"select date_dim.d_date_sk ss_sold_date_sk",
"      ,item.i_item_sk ss_item_sk",
"      ,customer.c_customer_sk ss_customer_sk",
"      ,customer_address.ca_address_sk ss_addr_sk",
"      ,ss_ticket_number",
"      ,ss_quantity",
"      ,ss_net_paid",
"      ,ss_net_profit",
" from {0}.stg_store_sales as store_sales",
" inner join {0}.date_dim on store_sales.sold_date = date_dim.d_date",
" left join {0}.item on store_sales.i_item_id = item.i_item_id and item.i_rec_end_date is null",
" left join {0}.customer on store_sales.c_customer_id =
item.c_customer_id",
" left join {0}.customer_address on store_sales.ca_address_id =
customer_address.ca_address_id;",
"/* Drop staging table */",
"drop table if exists {0}.stg_store_sales;",
"/* Commit and End transaction */",
"commit;",
"end transaction;" ]
}
"ExecuteSalesFactLoadJob": {
  "Type": "Task",
  "TimeoutSeconds": 180,
  "HeartbeatSeconds": 60,
  "Next": "GetSalesFactLoadStatus",
  "InputPath": ",",
  "ResultPath": ","
},
"GetSalesFactLoadStatus": {
  "Type": "Task",
  "Next": "CheckSalesFactLoadStatus",
  "TimeoutSeconds": 180,
  "HeartbeatSeconds": 60,
  "InputPath": ",",
  "ResultPath": ",.status"
},
"CheckSalesFactLoadStatus": {
  "Type": "Choice",
  "Choices": [
    {
      "Variable": ",.status",
      "StringEquals": "FAILED",
      "Next": "FailSalesFactLoad"
    },
    {
      "Variable": ",.status",
      "StringEquals": "FINISHED",
      "Next": "SalesETLPipelineComplete"
    }
  ],
  "Default": "SalesWait"
},
"SalesWait": {
  "Type": "Wait",
  "Seconds": 5,
  "Next": "GetSalesFactLoadStatus"
},
"FailSalesFactLoad": {
  "Type": "Fail",
  "Cause": "ETL Workflow Failed",
  "Error": "Error"
},
"ClusterUnavailable": {
  "Type": "Fail",
  "Cause": "Redshift cluster is not available",
  "Error": "Error"
},
"SalesETLPipelineComplete": {
  "Type": "Pass",
  "Next": "ValidateSalesMetric",
  "Result": {
    "input": {
      "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
      "redshift_database": "dev",
      "redshift_user": "awsuser",
      "redshift_schema": "tpcds",
      "snapshot_date": "2003-01-02",
      "action": "validate_sales_metric",
      "sql_statement": [
        "select 1/count(1) from (select d_date_sk from tpcds.date_dim where d_date='\{1\}')"
      ]
    }
  }
}
"ValidateSalesMetric": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftDataApi-
  AIDACKCEVSQ6C2EXAMPLE",
  "TimeoutSeconds": 180,
  "HeartbeatSeconds": 60,
  "Next": "GetValidateSalesMetricStatus",
  "InputPath": "$",
  "ResultPath": "$"
},
"GetValidateSalesMetricStatus": {
  "Type": "Task",
  "Next": "CheckValidateSalesMetricStatus",
  "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftDataApi-
  AIDACKCEVSQ6C2EXAMPLE",
  "TimeoutSeconds": 180,
  "HeartbeatSeconds": 60,
  "InputPath": "$",
  "ResultPath": "$.status"
},
"CheckValidateSalesMetricStatus": {
  "Type": "Choice",
  "Choices": [
    {
      "Variable": "$.status",
      "StringEquals": "FAILED",
      "Next": "FailSalesMetricValidation"
    },
    {
      "Variable": "$.status",
      "StringEquals": "FINISHED",
      "Next": "DataValidationComplete"
    }
  ],
  "Default": "SalesValidationWait"
},
"SalesValidationWait": {
  "Type": "Wait",
  "Seconds": 5,
  "Next": "GetValidateSalesMetricStatus"
},
"FailSalesMetricValidation": {
  "Type": "Fail",
  "Cause": "Data Validation Failed",
  "Error": "Error"
},
"DataValidationComplete": {
  "Type": "Pass",
  "Next": "InitializePauseCluster"
},
"InitializePauseCluster": {
  "Type": "Pass",
  "Next": "PauseCluster",
  "Result": {
    "input": {
      "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHBEXAMPLE",
      "operation": "pause"
    }
  }
},
"PauseCluster": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:111122223333:function:CFN36-RedshiftDataApi-
  AIDACKCEVSQ6C2EXAMPLE",
  "TimeoutSeconds": 180,
  "HeartbeatSeconds": 60,
  "InputPath": "$",
  "ResultPath": "$"}
"TimeoutSeconds": 180,
"HeartbeatSeconds": 60,
"Next": "PauseClusterWait",
"InputPath": 
"ResultPath": 
"Catch": [
  
  "ErrorEquals": 
  "States.ALL"
  
  "Next": "ClusterPausedComplete"
  
],
"InitializeCheckPauseCluster": {
  "Type": "Pass",
  "Next": "GetStateOfPausedCluster",
  "Result": {
    "input": {
      "redshift_cluster_id": "cfn36-redshiftcluster-AKIAI44QH8DHEXAMPLE",
      "operation": "status"
    }
  }
  
},
"GetStateOfPausedCluster": {
  "Type": "Task",
  "TimeoutSeconds": 180,
  "HeartbeatSeconds": 60,
  "Next": "IsClusterPaused",
  "InputPath": 
  "ResultPath": 
},
"IsClusterPaused": {
  "Type": "Choice",
  "Choices": [
    
    "Variable": "$\cdot.clusterStatus",
    "StringEquals": "available",
    "Next": "InitializePauseCluster"
    
  ],
  "Variable": "$\cdot.clusterStatus",
  "StringEquals": "paused",
  "Next": "ClusterPausedComplete"
  
  ],
  "Variable": "$\cdot.clusterStatus",
  "StringEquals": "unavailable",
  "Next": "ClusterUnavailable"
  
  ],
  "Variable": "$\cdot.clusterStatus",
  "StringEquals": "resuming",
  "Next": "PauseClusterWait"
  
  ],
  "PauseClusterWait": {
    "Type": "Wait",
    "Seconds": 720,
    "Next": "InitializeCheckPauseCluster"
  }
  
}
"ClusterPausedComplete": {
  "Type": "Pass",
  "End": true
}
}

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

**IAM Example**

These example AWS Identity and Access Management (IAM) policies generated by the sample project include the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": ["lambda:InvokeFunction"],
      "Resource": [
      ],
      "Effect": "Allow"
    }
  ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

**Use Step Functions and AWS Batch with error handling**

This sample project demonstrates how to use Step Functions to run a batch job with error-handling, with various state types, using AWS Batch and Amazon SNS. Deploying this sample project will create an AWS Step Functions state machine, an AWS Batch job, and an Amazon SNS topic.

In this project, Step Functions uses a state machine to call the AWS Batch job synchronously. It then waits for the job to succeed or fail, retries and catches errors when a job fails, then sends an Amazon SNS topic with a message about whether the job succeeded or failed.

**Create the State Machine and Provision Resources**

1. Open the Error-handling using AWS Batch and Amazon SNS sample project. The state machine Code and Visual Workflow are displayed.
2. Choose Next.

The Deploy and run page is displayed, listing the resources that will be created. For this sample project, the resources include:

- An AWS Batch job
- An Amazon SNS topic

3. Choose Deploy Resources.

   **Note**
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.

**Start a New Execution**

1. On the New execution page, enter an execution name (optional), and then choose Start Execution.

2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

3. Optionally, you can go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.

4. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

**Example State Machine Code**

The state machine in this sample project integrates with AWS Batch and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls AWS Batch and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the Resource field, and by passing Parameters to the service API.
For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
    "Comment": "An example of the Amazon States Language for notification on an AWS Batch job completion",
    "StartAt": "Submit Batch Job",
    "TimeoutSeconds": 3600,
    "States": {
        "Submit Batch Job": {
            "Type": "Task",
            "Resource": "arn:aws:states:::batch:submitJob.sync",
            "Parameters": {
                "JobName": "BatchJobNotification",
                "JobDefinition": "arn:aws:batch:us-west-2:123456789012:job-definition/BatchJobDefinition-123456789abcdef1"
            },
            "Next": "Notify Success",
            "Retry": [
                { "ErrorEquals": [ "States.ALL" ], "IntervalSeconds": 30, "MaxAttempts": 2, "BackoffRate": 1.5 }
            ],
            "Catch": [
                { "ErrorEquals": [ "States.ALL" ], "Next": "Notify Failure" }
            ]
        },
        "Notify Success": {
            "Type": "Task",
            "Resource": "arn:aws:states:::sns:publish",
            "Parameters": {
                "Message": "Batch job submitted through Step Functions succeeded",
            },
            "End": true
        },
        "Notify Failure": {
            "Type": "Task",
            "Resource": "arn:aws:states:::sns:publish",
            "Parameters": {
                "Message": "Batch job submitted through Step Functions failed",
            },
            "End": true
        }
    }
}
```
IAM Example

This example AWS Identity and Access Management (IAM) policy generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

Example BatchJobNotificationAccessPolicy

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": ["sns:Publish"],
            "Effect": "Allow"
        },
        {
            "Action": ["batch:SubmitJob", "batch:DescribeJobs", "batch:TerminateJob"],
            "Resource": "*",
            "Effect": "Allow"
        },
        {
            "Action": ["events:PutTargets", "events:PutRule", "events:DescribeRule"],
            "Effect": "Allow"
        }
    ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

Fan out an AWS Batch job

This sample project demonstrates how to use Step Functions's Map state to fan out AWS Batch jobs. Deploying this sample project will create an AWS Step Functions state machine, a Lambda function and an AWS Batch job queue.

In this project, Step Functions uses a state machine to invoke a Lambda function to do simple pre-processing, then invokes multiple AWS Batch jobs in parallel using the map state.
Create the State Machine and Provision Resources

1. Open the Fan out an AWS Batch job sample project. The state machine Code and Visual Workflow are displayed.

![State Machine Diagram]

2. Choose Next.

   The Deploy and run page is displayed, listing the resources that will be created. For this sample project, the resources include:
   - An AWS Batch job queue
   - A Lambda function

3. Choose Deploy and run.

   **Note**
   It can take up to 10 minutes for these resources and related IAM permissions to be created. While the Deploy resources page is displayed, you can open the Stack ID link to see which resources are being provisioned.

Start a New Execution

1. On the New execution page, enter an execution name (optional), and then choose Start Execution.
2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default, Step Functions generates a unique execution name automatically.

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

3. Optionally, you can go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.
4. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example State Machine Code

The state machine in this sample project integrates with AWS Batch and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls AWS Batch and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the Resource field, and by passing Parameters to the service API.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
    "Comment": "An example of the Amazon States Language for fanning out AWS Batch job",
    "StartAt": "Generate batch job input",
    "TimeoutSeconds": 3600,
    "States": {
        "Generate batch job input": {
            "Type": "Task",
            "Resource": "arn:aws:states:::lambda:invoke",
            "OutputPath": "$Payload",
            "Parameters": {
                "FunctionName": "<GENERATE_BATCH_JOB_INPUT_LAMBDA_FUNCTION_NAME>",
            },
            "Next": "Fan out batch jobs"
        },
        "Fan out batch jobs": {
            "Comment": "Start multiple executions of batch job depending on pre-processed data",
            "Type": "Map",
            "End": true,
            "ItemsPath": "$",
            "Parameters": {
                "BatchNumber.$": "$$.Map.Item.Value"
            },
            "Iterator": {
                "StartAt": "Submit Batch Job",
                "States": {
                    "Submit Batch Job": {
                        "Type": "Task",
                        "Resource": "arn:aws:states:::batch:submitJob.sync",
                        "Parameters": {
                            "JobName": "BatchJobFanOut",
                            "JobQueue": "<BATCH_QUEUE_ARN>",
                            "JobDefinition": "<BATCH_JOB_DEFINITION_ARN>"
                        },
                        "End": true
                    }
                }
            }
        }
    }
}
```
IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

**Example BatchJobFanOutAccessPolicy**

```json
{
   "Version": "2012-10-17",
   "Statement": [
       {
           "Action": [
               "batch:SubmitJob",
               "batch:DescribeJobs",
               "batch:TerminateJob"
           ],
           "Resource": "*",
           "Effect": "Allow"
       },
       {
           "Action": [
               "events:PutTargets",
               "events:PutRule",
               "events:DescribeRule"
           ],
           "Resource": [
               "arn:aws:events:us-west-2:123456789012:rule/StepFunctionsGetEventsForBatchJobsRule"
           ],
           "Effect": "Allow"
       }
   ]
}
```

**Example InvokeGenerateBatchJobMapLambdaPolicy**

```json
{
   "Statement": [
       {
           "Action": [
               "lambda:InvokeFunction"
           ],
           "Effect": "Allow"
       }
   ]
}
```

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).

**AWS Batch with Lambda**

This sample project demonstrates how to use Step Functions to pre-process data with AWS Lambda functions and then orchestrate AWS Batch jobs. Deploying this sample project will create an AWS Step Functions state machine, a Lambda function, and an AWS Batch job.
In this project, Step Functions uses a state machine to invoke a Lambda function to do simple pre-
processing before an AWS Batch job is submitted. Multiple jobs may be invoked depending on the result/
success of the previous one.

Create the State Machine and Provision Resources

1. Open the AWS Batch with Lambda sample project.

   The state machine Code and Visual Workflow are displayed.

2. Choose Next.

   The Deploy and run page is displayed, listing the resources that will be created. For this sample
   project, the resources include:

   • An AWS Batch job
   • A Lambda function

3. Choose Deploy Resources.

   Note
   It can take up to 10 minutes for these resources and related IAM permissions to be created.
   While the Deploy resources page is displayed, you can open the Stack ID link to see which
   resources are being provisioned.

Start a New Execution

1. On the New execution page, enter an execution name (optional), and then choose Start Execution.
2. (Optional) To identify your execution, you can specify a name for it in the Name box. By default,
   Step Functions generates a unique execution name automatically.

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

3. Optionally, you can go to the newly created state machine on the Step Functions Dashboard, and then choose New execution.

4. When an execution is complete, you can select states on the Visual workflow and browse the Input and Output under Step details.

Example State Machine Code

The state machine in this sample project integrates with AWS Batch and Amazon SNS by passing parameters directly to those resources.

Browse through this example state machine to see how Step Functions controls AWS Batch and Amazon SNS by connecting to the Amazon Resource Name (ARN) in the Resource field, and by passing Parameters to the service API.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

```json
{
    "Comment": "An example of the Amazon States Language for using batch job with preprocessing lambda",
    "StartAt": "Generate batch job input",
    "TimeoutSeconds": 3600,
    "States": {
        "Generate batch job input": {
            "Type": "Task",
            "Resource": "arn:aws:states:::lambda:invoke",
            "OutputPath": "$.batch_input",
            "Parameters": {
                "FunctionName": "<GENERATE_BATCH_JOB_INPUT_LAMBDA_FUNCTION_NAME>"
            },
            "Next": "Submit Batch Job"
        },
        "Submit Batch Job": {
            "Type": "Task",
            "Resource": "arn:aws:states:::batch:submitJob.sync",
            "Parameters": {
                "JobName": "BatchJobFanOut",
                "JobQueue": "<BATCH_QUEUE_ARN>",
                "JobDefinition": "<BATCH_JOB_DEFINITION_ARN>",
                "Parameters.$": "$.batch_input"
            },
            "End": true
        }
    }
}
```

IAM Example

These example AWS Identity and Access Management (IAM) policies generated by the sample project includes the least privilege necessary to execute the state machine and related resources. We recommend that you include only those permissions that are necessary in your IAM policies.

Example BatchJobWithLambdaAccessPolicy

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::s3-access",
            "Condition": {
                "ArnLike": {
                    "AWS::AccountId": "<ACCOUNT_ID>"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::s3-access/*",
            "Condition": {
                "ArnLike": {
                    "AWS::AccountId": "<ACCOUNT_ID>"
                }
            }
        }
    ]
}
```
"Statement": [
{
"Action": ["sns:Publish"],
"Effect": "Allow"},
{
"Action": ["batch:SubmitJob",
"batch:DescribeJobs",
"batch:TerminateJob"],
"Resource": "*",
"Effect": "Allow"},
{
"Action": ["events:PutTargets",
"events:PutRule",
"events:DescribeRule"],
"Effect": "Allow"
}
]

Example InvokeGenerateBatchJobMapLambdaPolicy

{
"Statement": [
{
"Action": ["lambda:InvokeFunction"],
"Effect": "Allow"
}
]

For information about how to configure IAM when using Step Functions with other AWS services, see IAM Policies for integrated services (p. 531).
Quotas

AWS Step Functions places quotas on the sizes of certain state machine parameters, such as the number of API actions during a certain time period or the number of state machines that you can define. Although these quotas are designed to prevent a misconfigured state machine from consuming all of the resources of the system, many aren't hard quotas. Use the Support Center page in the AWS Management Console to request a quota increase for resources provided by AWS Step Functions on a per-Region basis. For more information, see AWS service quotas in the AWS General Reference.

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

Topics
• General quotas (p. 485)
• Quotas related to accounts (p. 486)
• Quotas related to state throttling (p. 486)
• Quotas related to API action throttling (p. 487)
• Quotas related to state machine executions (p. 487)
• Quotas related to task executions (p. 488)
• Other quotas (p. 488)
• Restrictions related to tagging (p. 489)

General quotas

<table>
<thead>
<tr>
<th>Quota</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Names in Step Functions</td>
<td>State machine, execution, and activity names must be 1–80 characters in length, must be unique for your account and AWS Region, and must not contain any of the following:</td>
</tr>
<tr>
<td></td>
<td>• Whitespace</td>
</tr>
<tr>
<td></td>
<td>• Wildcard characters (? * )</td>
</tr>
<tr>
<td></td>
<td>• Bracket characters (&lt; &gt; { } [ ] )</td>
</tr>
<tr>
<td></td>
<td>• Special characters ( : ; , \</td>
</tr>
<tr>
<td></td>
<td>• Control characters (\u0000 - \u001f or \u007f - \u009f)</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
Quotas related to accounts

<table>
<thead>
<tr>
<th>Resource</th>
<th>Default quota</th>
<th>Can be increased to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of registered state machines</td>
<td>10,000</td>
<td>22,000 to 23,000</td>
</tr>
<tr>
<td>Maximum number of registered activities</td>
<td>10,000</td>
<td>Tens of thousands</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>
<td>Hard quota</td>
</tr>
<tr>
<td>Maximum open executions per account</td>
<td>1,000,000 executions per AWS Region. Exceeding this will cause an ExecutionLimitExceeded error. Does not apply to Express Workflows.</td>
<td>Millions</td>
</tr>
<tr>
<td>Synchronous Express Workflows concurrent executions</td>
<td>1,000-3,700 (varies per region)</td>
<td>Tens of thousands. Synchronous Express Workflows do not contribute to the existing account capacity limits. For more information, see the section called “Synchronous and Asynchronous Express Workflows” (p. 20).</td>
</tr>
</tbody>
</table>

Quotas related to state throttling

Step Functions state transitions are throttled using a token bucket scheme to maintain service bandwidth. Standard Workflows and Express Workflows have different state transition throttling. Standard Workflows quotas are soft quotas and can be increased.

**Note**

Throttling on the StateTransition service metric is reported as ExecutionThrottled in Amazon CloudWatch. For more information, see the ExecutionThrottled CloudWatch metric (p. 492).

<table>
<thead>
<tr>
<th>Service metric</th>
<th>Standard Bucket size</th>
<th>Refill rate per second</th>
<th>Express Bucket size</th>
<th>Refill rate per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateTransition</td>
<td>5,000</td>
<td>1,500</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>— In US East (N. Virginia), US West (Oregon), and Europe (Ireland)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StateTransition</td>
<td>800</td>
<td>500</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>— All other regions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Quotas related to API action throttling

Some Step Functions API actions are throttled using a token bucket scheme to maintain service bandwidth. These quotas are soft quotas and can be increased.

**Note**
Throttling quotas are per account, per AWS Region. AWS Step Functions may increase both the bucket size and refill rate at any time. Synchronous Express execution API calls do not contribute to the existing account capacity limits. While Step Functions will provide capacity on demand and will automatically scale with sustained workload, surges in workload may be throttled until capacity is available. Do not rely on these throttling rates to limit your costs.

<table>
<thead>
<tr>
<th>API name</th>
<th>Standard</th>
<th>Express</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bucket size</td>
<td>Refill rate per second</td>
</tr>
<tr>
<td>StartExecution</td>
<td>1,300</td>
<td>300</td>
</tr>
<tr>
<td>— In US East (N. Virginia), US West (Oregon), and Europe (Ireland)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StartExecution</td>
<td>800</td>
<td>150</td>
</tr>
<tr>
<td>— All other regions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Quotas related to state machine executions

The following table describes quotas related to state machine executions. State machine execution quotas are hard quotas that can't be changed.

<table>
<thead>
<tr>
<th>Quota</th>
<th>Standard</th>
<th>Express</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum execution time</td>
<td>1 year. If an execution runs for more than the 1-year maximum, it will fail with a States.Timeout error and emit a ExecutionsTimedOut CloudWatch metric.</td>
<td>5 minutes. If an execution runs for more than the 5-minute maximum, it will fail with a States.Timeout error and emit a ExecutionsTimedOut CloudWatch metric.</td>
</tr>
<tr>
<td>Maximum execution history size</td>
<td>25,000 events in a single state machine execution history. If the execution history reaches this quota, the execution will fail. To avoid this, see Avoid reaching the history quota (p. 268).</td>
<td>Unlimited.</td>
</tr>
<tr>
<td>Maximum execution idle time</td>
<td>1 year (constrained by the maximum execution time)</td>
<td>5 minutes (constrained by the maximum execution time)</td>
</tr>
<tr>
<td>Execution history retention time</td>
<td>90 days. After this time, you can no longer retrieve or view the execution history. There is no further quota for the number</td>
<td>To see execution history, Amazon CloudWatch Logs logging must be configured. For</td>
</tr>
</tbody>
</table>
## Quotas related to task executions

The following table describes quotas related to task executions, and are hard quotas that can't be changed.

<table>
<thead>
<tr>
<th>Quota</th>
<th>Standard</th>
<th>Express</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum task execution time</td>
<td>1 year (constrained by the maximum execution time)</td>
<td>5 minutes (constrained by the maximum execution time)</td>
</tr>
<tr>
<td>Maximum time Step Functions keeps a task in the queue</td>
<td>1 year (constrained by the maximum execution time)</td>
<td>5 minutes (constrained by the maximum execution time)</td>
</tr>
<tr>
<td>Maximum activity pollers per Amazon Resource Name (ARN)</td>
<td>1,000 pollers calling GetActivityTask per ARN. Exceeding this quota results in this error: &quot;The maximum number of workers concurrently polling for activity tasks has been reached.&quot;</td>
<td>Does not apply to Express Workflows.</td>
</tr>
<tr>
<td>Maximum input or output size for a task, state, or execution</td>
<td>262,144 bytes of data as a UTF-8 encoded string. This quota affects tasks (activity, Lambda function, or integrated service), state or execution output, and input data when scheduling a task, entering a state, or starting an execution.</td>
<td>262,144 bytes of data as a UTF-8 encoded string. This quota affects tasks (activity, Lambda function, or integrated service), state or execution output, and input data when scheduling a task, entering a state, or starting an execution.</td>
</tr>
</tbody>
</table>

## Other quotas

These quotas are soft quotas and can be increased.

<table>
<thead>
<tr>
<th>API name</th>
<th>In US East (N. Virginia), US West (Oregon), and Europe (Ireland)</th>
<th>All other regions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bucket size</td>
<td>Refill rate per second</td>
</tr>
<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>
</tbody>
</table>
Restrictions related to tagging

Be aware of these restrictions when tagging Step Functions resources.

**Note**
Tagging restrictions cannot be increased like other quotas.

<table>
<thead>
<tr>
<th>Restriction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of tags per resource</td>
<td>50</td>
</tr>
<tr>
<td>Maximum key length</td>
<td>128 Unicode characters in UTF-8</td>
</tr>
<tr>
<td>Maximum value length</td>
<td>256 Unicode characters in UTF-8</td>
</tr>
<tr>
<td>Prefix restriction</td>
<td>Do not use the <code>aws:</code> prefix in your tag names or values because it is reserved for AWS use. You can't edit or delete tag names or values with this prefix. Tags with this prefix do not count against your tags per resource quota.</td>
</tr>
<tr>
<td>Restriction</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>Character restrictions</td>
<td>Tags may only contain Unicode letters, digits, whitespace, or these symbols: <code>_ . : / = + - @</code></td>
</tr>
</tbody>
</table>
Logging and monitoring in AWS Step Functions

Logging and monitoring are important for maintaining the reliability, availability, and performance of Step Functions and your AWS solutions. There are several tools available to use with Step Functions:

Topics

- Monitoring Step Functions Using CloudWatch (p. 491)
- EventBridge (CloudWatch Events) for Step Functions execution status changes (p. 499)
- Logging Step Functions Using AWS CloudTrail (p. 504)
- Logging using CloudWatch Logs (p. 508)
- AWS X-Ray and Step Functions (p. 512)

Monitoring Step Functions Using CloudWatch

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

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

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

For example, with Step Functions, you can monitor how many activities or AWS 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
Metrics That Report a Time Interval

Some of the Step Functions CloudWatch metrics are *time intervals*, always measured in milliseconds. These metrics generally correspond to stages of your execution for which you can set state machine, activity, and Lambda function timeouts, with descriptive names.

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

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

Metrics That Report a Count

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

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

Execution Metrics

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

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecutionTime</td>
<td>The interval, in milliseconds, between the time the execution starts and the time it closes.</td>
</tr>
<tr>
<td>ExecutionThrottled</td>
<td>The number of StateEntered events and retries that have been throttled. This is related to StateTransition throttling. For more</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>
<tr>
<td>LambdaFunctionScheduleTime</td>
<td>The interval, in milliseconds, for which the Lambda function stays in the schedule state.</td>
</tr>
<tr>
<td>LambdaFunctionTime</td>
<td>The interval, in milliseconds, between the time the Lambda function is scheduled and the time it closes.</td>
</tr>
<tr>
<td>LambdaFunctionsFailed</td>
<td>The number of failed Lambda functions.</td>
</tr>
<tr>
<td>LambdaFunctionsScheduled</td>
<td>The number of scheduled Lambda functions.</td>
</tr>
<tr>
<td>LambdaFunctionsStarted</td>
<td>The number of started Lambda functions.</td>
</tr>
<tr>
<td>LambdaFunctionsSucceeded</td>
<td>The number of successfully completed Lambda functions.</td>
</tr>
<tr>
<td>LambdaFunctionsTimedOut</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>LambdaFunctionArn</td>
<td>The ARN of the Lambda function.</td>
</tr>
</tbody>
</table>

**Note**

Lambda Function Metrics are emitted for Task states that specify the Lambda function ARN in the Resource field. Task states that use "Resource": "arn:aws:states:::lambda:invoke" emit Service Integration Metrics instead. For more information, see Invoke Lambda with Step Functions (p. 301).

Service Integration Metrics

The AWS/States namespace includes the following metrics for Step Functions service integrations. For more information, see Using AWS Step Functions with other services (p. 271).

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceIntegrationRunTime</td>
<td>The interval, in milliseconds, between the time the Service Task starts and the time it closes.</td>
</tr>
<tr>
<td>ServiceIntegrationScheduleTime</td>
<td>The interval, in milliseconds, for which the Service Task stays in the schedule state.</td>
</tr>
<tr>
<td>ServiceIntegrationTime</td>
<td>The interval, in milliseconds, between the time the Service Task is scheduled and the time it closes.</td>
</tr>
<tr>
<td>ServiceIntegrationsFailed</td>
<td>The number of failed Service Tasks.</td>
</tr>
</tbody>
</table>
Metric | Description
--- | ---
ServiceIntegrationsScheduled | The number of scheduled Service Tasks.
ServiceIntegrationsStarted | The number of started Service Tasks.
ServiceIntegrationsSucceeded | The number of successfully completed Service Tasks.
ServiceIntegrationsTimedOut | The number of Service Tasks that time out on close.

Dimension for Step Functions Service Integration Metrics

| Dimension | Description |
--- | ---|
ServiceIntegrationResourceArn | The resource ARN of the integrated service.

Service Metrics

The AWS/States namespace includes the following metrics for the Step Functions service.

| Metric | Description |
--- | ---|
ThrottledEvents | The count of requests that have been throttled.
ProvisionedBucketSize | The count of available requests per second.
ProvisionedRefillRate | The count of requests per second that are allowed into the bucket.
ConsumedCapacity | The count of requests per second.

Dimension for Step Functions Service Metrics

| Dimension | Description |
--- | ---|
StateTransition | Filters data to show State Transitions metrics.

API Metrics

The AWS/States namespace includes the following metrics for the Step Functions API.

| Metric | Description |
--- | ---|
ThrottledEvents | The count of requests that have been throttled.
ProvisionedBucketSize | The count of available requests per second.
ProvisionedRefillRate | The count of requests per second that are allowed into the bucket.
ConsumedCapacity | The count of requests per second.
Dimension for Step Functions API Metrics

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APIName</td>
<td>Filters data to an API of the specified API name.</td>
</tr>
</tbody>
</table>

Viewing Metrics for Step Functions

1. Sign in to the AWS Management Console and open the CloudWatch console.
2. Choose Metrics, and on the All Metrics tab, choose States.

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

- Execution Metrics
- Activity Function Metrics
- Lambda Function Metrics
- Service Integration 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 dropdown list to display values as lines, stacked areas, or numbers (values).
• To view the details about a graph, hover over the metric color code that appears below the graph.

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

#### To set an alarm on a metric

1. Sign in to the AWS Management Console and open the CloudWatch console.
2. Choose **Metrics**, and on the **All Metrics** tab, choose **States**.

If you ran any executions recently, you will see up to four types of metrics:
- Execution Metrics
- Activity Function Metrics
- Lambda Function Metrics
- Service Integration 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** page 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/AmazonS3/latest/userguide/creating-cloudwatch-alarms.html) in the *Amazon CloudWatch User Guide*.

**EventBridge (CloudWatch Events) for Step Functions execution status changes**

Amazon EventBridge is an AWS service that enables you to respond to state changes in an AWS resource. You can use AWS Step Functions Standard Workflows with EventBridge, while Express Workflows do not emit events to EventBridge. There are two ways to use Step Functions Standard Workflows with EventBridge.

You can configure Step Functions to emit EventBridge events when an execution status changes. This enables you to monitor your workflows without having to constantly poll using the `DescribeExecution` API. Based on changes in state machine executions you can use an EventBridge target to start new state machine executions, call AWS Lambda functions, publish messages to Amazon Simple Notification Service (Amazon SNS) topics, and more.

You can also configure a Step Functions state machine as a target in EventBridge. This enables you to trigger an execution of a Step Functions workflow in response to an event from another AWS service.

For more information, see the [Amazon EventBridge User Guide](https://docs.aws.amazon.com/AmazonEventBridge/latest/userguide/).


EventBridge payloads

An EventBridge event can contain an input property in its definition. For some events, an EventBridge event can also contain an output property in its definition.

- If the combined escaped input and escaped output sent to EventBridge exceeds 248KB, then the input will be excluded. Similarly, if the escaped output exceeds 248KB, then the output will be excluded. This is a result of the EventBridge events quotas.
- You can determine whether a payload has been truncated with the inputDetails and outputDetails properties. For more information, see the CloudWatchEventsExecutionDataDetails Data Type.
- For Standard Workflows, you can see the full input and output by using DescribeExecution.
- DescribeExecution is not available for Express Workflows. If you want to see the full input/output, you can wrap your Express Workflow with a Standard Workflow. Another option is to use Amazon S3 ARNs. For information about using ARNs, see the section called “Use Amazon S3 ARNs instead of passing large payloads” (p. 267).

Topics

- Step Functions event examples (p. 500)
- Routing a Step Functions event to EventBridge in the EventBridge console (p. 503)

Step Functions event examples

The following are examples of Step Functions sending events to EventBridge:

Topics

- Execution started (p. 500)
- Execution succeeded (p. 501)
- Execution failed (p. 501)
- Execution timed out (p. 502)
- Execution aborted (p. 502)

In each case, the detail section in the event data provides the same information as the DescribeExecution API. The status field indicates the status of the execution at the time the event was sent, one of RUNNING, SUCCEEDED, FAILED, TIMED_OUT, or ABORTED depending on the event emitted.

Execution started

```
{
  "version": "0",
  "id": "315c1398-40ff-a850-213b-158f73e60175",
  "detail-type": "Step Functions Execution Status Change",
  "source": "aws.states",
  "account": "012345678912",
  "time": "2019-02-26T19:42:21Z",
  "region": "us-east-1",
  "resources": [
  ],
  "detail": {
```
Execution succeeded

```
{
  "version": "0",
  "id": "315c1398-40ff-a850-213b-158f73e60175",
  "detail-type": "Step Functions Execution Status Change",
  "source": "aws.states",
  "account": "012345678912",
  "time": "2019-02-26T19:42:21Z",
  "region": "us-east-1",
  "resources": [
  ],
  "detail": {
    "name": "execution-name",
    "status": "SUCCEEDED",
    "startDate": 1547148840101,
    "stopDate": 1547148840122,
    "input": "{}
  },
  "output": "\"Hello World!\"
}
```

Execution failed

```
{
  "version": "0",
  "id": "315c1398-40ff-a850-213b-158f73e60175",
  "detail-type": "Step Functions Execution Status Change",
  "source": "aws.states",
  "account": "012345678912",
  "time": "2019-02-26T19:42:21Z",
  "region": "us-east-1",
  "resources": [
  ]
}
```
Execution timed out

```json
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "012345678912",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-1",
    "resources": [
    ],
    "detail": {
        "name": "execution-name",
        "status": "TIMEOUT",
        "startDate": 1551224926156,
        "stopDate": 1551224927157,
        "input": "{}",
        "inputDetails": {
            "included": true
        },
        "output": null,
        "outputDetails": null
    }
}
```

Execution aborted

```json
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "012345678912",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-1",
    "resources": [
    ],
    "detail": {
        "name": "execution-name",
        "status": "FAILED",
        "startDate": 1551225146847,
        "stopDate": 1551225151881,
        "input": "{}",
        "inputDetails": {
            "included": true
        },
        "output": null,
        "outputDetails": null
    }
}
```
Routing a Step Functions event to EventBridge in the EventBridge console

Use the following instructions to learn how to trigger the execution of a Step Functions state machine whenever a specific Step Functions state machine completes running successfully. You use the Amazon EventBridge console to specify the state machine whose execution you want to trigger.

1. On the Details page of a state machine, choose Actions, and then choose Create EventBridge (CloudWatch Events) rule.

   ![Create EventBridge (CloudWatch Events) rule](image)

   Alternatively, open the EventBridge console at `https://console.aws.amazon.com/events/`. In the navigation pane, choose Rules under Events.

2. Choose Create rule.

3. Enter a Name for your rule (for example, `StepFunctionsEventRule`) and optionally enter a Description for the rule.

4. For Event bus and Rule type, keep the default selections.

5. Choose Next.

6. Under Event Source, keep the default selection of AWS events or EventBridge partner events.

7. Under Event pattern, do the following:

   a. In the Event source dropdown list, keep the default selection of AWS services.

   b. From the AWS service dropdown list, choose Step Functions.

   c. From the Event Type dropdown list, choose Step Functions Execution Status Change.
d. (Optional) Configure a specific status, state machine Amazon Resource Name (ARN), or execution ARN. For this procedure, choose **Specific status(es)**, and then choose **SUCCEEDED** from the dropdown list.

8. Choose **Next**.

9. On the **Select target(s)** page, keep the default selection of **AWS service**.

10. From the **Select a target** dropdown list, choose an AWS service. For example, you could launch a Lambda function, or run a Step Functions state machine. For this procedure, choose **Step Functions state machine**.

11. From the **State machine** dropdown list, choose a state machine.

12. Under **Execution role**, keep the default selection of **Create a new role for this specific resource**.

13. Choose **Next**, and then choose **Next** again.

14. On the **Review and create** page, review the details of the rule and choose **Create rule**.

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

---

### Logging Step Functions Using AWS CloudTrail

Step Functions is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Step Functions. CloudTrail captures all API calls for Step Functions as events, including calls from the Step Functions console and from code calls to the Step Functions APIs.

If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon Simple Storage Service (Amazon S3) bucket, including events for Step Functions. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**.

Using the information collected by CloudTrail, you can determine the request that was made to Step Functions, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the [AWS CloudTrail User Guide](https://docs.aws.amazon.com/AmazonCloudTrail/latest/userguide/).

### Step Functions Information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Step Functions, that activity is recorded in a CloudTrail event with other AWS service events in **Event history**.

You can view, search, and download recent events in your AWS account. For more information, see [Viewing Events with CloudTrail Event History](https://docs.aws.amazon.com/AmazonCloudTrail/latest/userguide/).

For an ongoing record of events in your AWS account, including events for Step Functions, create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act on the event data collected in CloudTrail logs.

For more information, see the following:

- [Overview for Creating a Trail](https://docs.aws.amazon.com/AmazonCloudTrail/latest/userguide/)
- [CloudTrail Supported Services and Integrations](https://docs.aws.amazon.com/AmazonCloudTrail/latest/userguide/)
- [Configuring Amazon SNS Notifications for CloudTrail](https://docs.aws.amazon.com/AmazonCloudTrail/latest/userguide/)
- [Receiving CloudTrail Log Files from Multiple Regions](https://docs.aws.amazon.com/AmazonCloudTrail/latest/userguide/)
- [Receiving CloudTrail Log Files from Multiple Accounts](https://docs.aws.amazon.com/AmazonCloudTrail/latest/userguide/)

---

504
Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

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

For more information, see the CloudTrail userIdentity Element.

**Example: Step Functions Log File Entries**

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files are not an ordered stack trace of the public API calls, so they don't appear in any specific order.

**CreateActivity**

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

```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": {
    "name": "OtherActivityPrefix.2016-10-27-18-16-56.894c791e-2ced-4cf4-8523-376469410c25"
  },
  "responseElements": {
    "creationDate": "Oct 28, 2016 1:17:56 AM"
  },
  "requestID": "37c67602-9cac-11e6-aed5-5b57d226e9ef",
  "eventID": "dc3becef-d06d-49bf-bc93-9b76b5f00774",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

**CreateStateMachine**

The following example shows a CloudTrail log entry that demonstrates the `CreateStateMachine` action.

```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": "CreateStateMachine",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "10.61.88.189",
  "userAgent": "Coral/Netty",
  "requestParameters": {
    "name": "OtherStateMachineName.2016-10-27-18-16-56.894c791e-2ced-4cf4-8523-376469410c25"
  },
  "responseElements": {
    "creationDate": "Oct 28, 2016 1:17:56 AM"
  },
  "requestID": "37c67602-9cac-11e6-aed5-5b57d226e9ef",
  "eventID": "dc3becef-d06d-49bf-bc93-9b76b5f00774",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```
DeleteActivity

The following example shows a CloudTrail log entry that demonstrates the `DeleteActivity` action.

```json
{
  "eventVersion": "1.04",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDAJYDLDBVB14EXAMPLE",
    "arn": "arn:aws:iam::123456789012:user/test-user",
    "accountId": "123456789012",
    "accessKeyId": "AKIAJL5C75K4ZEXAMPLE",
    "userName": "test-user"
  },
  "eventTime": "2016-10-28T01:18:27Z",
  "eventSource": "states.amazonaws.com",
  "eventName": "DeleteActivity",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "10.61.88.189",
  "userAgent": "Coral/Netty",
  "requestParameters": {
  },
  "responseElements": null,
  "requestID": "490374ea-9cac-11e6-aed5-5b57d226e9ef",
  "eventID": "e5eb9a3d-13bc-4fa1-9531-232d1914d263",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```
DeleteStateMachine

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

```
```

StartExecution

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

```
```

507
StopExecution

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

```
{
  "eventVersion": "1.04",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDAJYDLDBVB14EXAMPLE",
    "arn": "arn:aws:iam::123456789012:user/test-user",
    "accountId": "123456789012",
    "accessKeyId": "AKIAJL5C75K4ZEXAMPLE",
    "userName": "test-user"
  },
  "eventTime": "2016-10-28T01:18:20Z",
  "eventSource": "states.amazonaws.com",
  "eventName": "StopExecution",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "10.61.88.189",
  "userAgent": "Coral/Netty",
  "requestParameters": {
  },
  "responseElements": {
    "stopDate": "Oct 28, 2016 1:18:20 AM"
  },
  "requestID": "4567625b-9cac-11e6-aed5-5b57d226e9ef",
  "eventID": "e658c743-c537-459a-aea7-daf83c18c53",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Logging using CloudWatch Logs

Standard Workflows record execution history in AWS Step Functions, although you can optionally configure logging to Amazon CloudWatch Logs.

Unlike Standard Workflows, Express Workflows don't record execution history in AWS Step Functions. To see execution history and results for an Express Workflow, you must configure logging to Amazon CloudWatch Logs. Publishing logs doesn't block or slow down executions.

**Note**

When you configure logging, CloudWatch Logs charges will apply.

**Configure logging**

When you create a Standard Workflow using the Step Functions console, it will not be configured to enable logging to CloudWatch Logs. An Express Workflow created using the Step Functions console will by default be configured to enable logging to CloudWatch Logs.
For Express workflows, Step Functions can create a role with the necessary AWS Identity and Access Management (IAM) policy for CloudWatch Logs. If you create a Standard Workflow, or an Express Workflow using the API, CLI, or AWS CloudFormation, Step Functions will not enable logging by default, and you will need ensure your role has the necessary permissions.

For each execution started from the console, Step Functions provides a link to CloudWatch Logs, configured with the correct filter to fetch log events specific for that execution.

To configure logging, you can pass the `LoggingConfiguration` parameter when using `CreateStateMachine` or `UpdateStateMachine`. You can further analyze your data in CloudWatch Logs by using CloudWatch Logs Insights. For more information see Analyzing Log Data with CloudWatch Logs Insights.

## CloudWatch Logs payloads

Execution history events may contain either input or output properties in their definitions. If escaped input or escaped output sent to CloudWatch Logs exceeds 248KB, it will be truncated as a result of CloudWatch Logs quotas.

- You can determine whether a payload has been truncated by reviewing the `inputDetails` and `outputDetails` properties. For more information, see the `HistoryEventExecutionDataDetails` Data Type.
- For Standard Workflows, you can see the full execution history by using `GetExecutionHistory`.
- `GetExecutionHistory` is not available for Express Workflows. If you want to see the full input and output, you can use Amazon S3 ARNs. For more information, see the section called "Use Amazon S3 ARNs instead of passing large payloads" (p. 267).

## IAM Policies for logging to CloudWatch Logs

You will also need to configure your state machine's execution IAM role to have the proper permission to log to CloudWatch Logs as shown in the following example.

### IAM policy example

The following is an example policy you can use to configure your permissions.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogDelivery",
                "logs:GetLogDelivery",
                "logs:UpdateLogDelivery",
                "logs:DeleteLogDelivery",
                "logs:ListLogDeliveries",
                "logs:PutLogEvents",
                "logs:PutResourcePolicy",
                "logs:DescribeResourcePolicies",
                "logs:DescribeLogGroups"
            ],
            "Resource": "*
        }
    ]
}
```

If you're unable to access the CloudWatch Logs, make sure you've done the following:
Log levels

You can choose from OFF, ALL, ERROR, or FATAL. No event types log when set to OFF and all event types do when set to ALL. For ERROR and FATAL, see the following table.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>ALL</th>
<th>ERROR</th>
<th>FATAL</th>
<th>OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChoiceStateEntered</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChoiceStateExited</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExecutionAborted</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ExecutionFailed</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ExecutionStarted</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExecutionSucceeded</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExecutionTimedOut</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FailStateEntered</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LambdaFunctionFailed</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LambdaFunctionScheduled</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LambdaFunctionScheduleFailed</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LambdaFunctionStartFailed</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LambdaFunctionSucceeded</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LambdaFunctionTimedOut</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>MapIterationAborted</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>MapIterationFailed</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>MapIterationStarted</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>MapIterationSucceeded</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
### Log levels

<table>
<thead>
<tr>
<th>Event Type</th>
<th>ALL</th>
<th>ERROR</th>
<th>FATAL</th>
<th>OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapStateAborted</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MapStateEntered</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MapStateExited</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MapStateFailed</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MapStateStarted</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MapStateSucceeded</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelStateAborted</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelStateEntered</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelStateExited</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelStateFailed</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelStateStarted</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelStateSucceeded</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PassStateEntered</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PassStateExited</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SucceedStateEntered</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SucceedStateExited</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskFailed</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TaskScheduled</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskStarted</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskStartFailed</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TaskStateAborted</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TaskStateEntered</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskStateExited</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskSubmitFailed</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TaskSubmitted</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskSucceeded</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskTimedOut</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>WaitStateAborted</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>WaitStateEntered</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WaitStateExited</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
AWS X-Ray and Step Functions

You can use AWS X-Ray to visualize the components of your state machine, identify performance bottlenecks, and troubleshoot requests that resulted in an error. Your state machine sends trace data to X-Ray, and X-Ray processes the data to generate a service map and searchable trace summaries.

With X-Ray enabled for your state machine, you can trace requests as they are executed in Step Functions, in all AWS Regions where X-Ray is available. This gives you a detailed overview of an entire Step Functions request. Step Functions will send traces to X-Ray for state machine executions, even when a trace ID is not passed by an upstream service. You can use an X-Ray service map to view the latency of a request, including any AWS services that are integrated with X-Ray. You can also configure sampling rules to tell X-Ray which requests to record, and at what sampling rates, according to criteria that you specify.

When X-Ray is not enabled for your state machine, and an upstream service does not pass a trace ID, Step Functions will not send traces to X-Ray for state machine executions. However, if a trace ID is passed by an upstream service, Step Functions will then send traces to X-Ray for state machine executions.

You can use AWS X-Ray with Step Functions in regions where both are supported. See the Step Functions and X-Ray endpoints and quotas pages for information on region support for X-Ray and Step Functions.

**X-Ray and Step Functions Combined Quotas**

You can add data to a trace for up to seven days, and query trace data going back thirty days, the length of time that X-Ray stores trace data. Your traces will be subject to X-Ray quotas. In addition to other quotas, X-Ray provides a minimum guaranteed trace size of 100KB for Step Functions state machines. If more than 100KB of trace data is provided to X-Ray, this may result in a frozen trace. See the service quotas section of the X-Ray endpoints and quotas page for more information on other quotas for X-Ray.

**Topics**

- Setup and configuration (p. 512)
- Concepts (p. 514)
- Step Functions service integrations and X-Ray (p. 515)
- Viewing the X-Ray console (p. 516)
- Viewing X-Ray tracing information for Step Functions (p. 516)
- Traces (p. 516)
- Service map (p. 517)
- Segments and subsegments (p. 518)
- Analytics (p. 520)
- Configuration (p. 521)
- What if there is no data in the trace map or service map? (p. 522)

**Setup and configuration**

**Enable X-Ray tracing when creating a state machine**

You can enable X-Ray tracing when creating a new state machine by selecting Enable X-Ray tracing on the Specify details page.

1. Open the Step Functions console and choose Create state machine.
2. On the Choose authoring method page, choose an appropriate option to create your state machine. If you choose Run a sample project, you cannot enable X-Ray tracing during the state machine
creation, and you will need to enable X-Ray tracing after your state machine has been created. For more information about enabling X-Ray in an existing state machine, see Enable X-Ray in an existing state machine (p. 513).

Choose Next.

3. On the Specify details page, configure your state machine.


Your Step Functions state machine will now send traces to X-Ray for state machine executions.

Note
If you choose to use an existing IAM role, you should ensure that X-Ray writes are allowed.

Enable X-Ray in an existing state machine

To enable X-Ray in an existing state machine:

1. In the Step Functions console, select the state machine for which you want to enable tracing.
2. Choose Edit.
3. Choose Enable X-Ray tracing.

You will see a notification telling you that you that you may need to make additional changes.

Note
When you enable X-Ray for an existing state machine, you must ensure that you have an IAM policy that grants sufficient permissions for X-Ray to perform traces. You can either add one manually, or generate one. For more information, see the IAM policy section for X-Ray (p. 564).

4. (Optional) Auto-generate a new role for your state machine to include X-Ray permissions.
5. Choose Save.

Configure X-Ray tracing for Step Functions

When you first run a state machine with X-Ray tracing enabled, it will use the default configuration values for X-Ray tracing. AWS X-Ray does not collect data for every request that is sent to an application. Instead, it collects data for a statistically significant number of requests. The default is to record the first request each second, and five percent of any additional requests. One request per second is the reservoir.
This ensures that at least one trace is recorded each second as long as the service is serving requests. Five percent is the rate at which additional requests beyond the reservoir size are sampled.

To avoid incurring service charges when you are getting started, the default sampling rate is conservative. You can configure X-Ray to modify the default sampling rule and configure additional rules that apply sampling based on properties of the service or request.

For example, you might want to disable sampling and trace all requests for calls that modify state or handle user accounts or transactions. For high-volume read-only calls, like background polling, health checks, or connection maintenance, you can sample at a low rate and still get enough data to observe issues that occur.

To configure a sampling rule for your state machine:

1. Go to the X-Ray console.
2. Choose Sampling.
3. To create a rule, choose Create sampling rule.
   - To edit a rule, choose a rule's name.
   - To delete a rule, choose a rule and use the Actions menu to delete it.

Some parts of existing sampling rules, such as the name and priority, cannot be changed. Instead, add or clone an existing rule, make the changes you want, then use the new rule.

For detailed information on X-Ray sampling rules and how to configure the various parameters, see Configuring sampling rules in the X-Ray console.

**Concepts**

**The X-Ray console**

The AWS X-Ray console enables you to view service maps and traces for requests that your applications serve. You can access the console to view detailed information collected by X-Ray when it's enabled for your state machine.

See Viewing the X-Ray console (p. 516) for information on how to access the X-Ray console for your state machine executions.

For detailed information about the X-Ray console, see the X-Ray console documentation.

**Segments, subsegments, and traces**

A segment records information about a request to your state machine. It contains information such as the work that your state machine performs, and may also contain subsegments with information about downstream calls.

A trace collects all the segments generated by a single request.

**Sampling**

To ensure efficient tracing and provide a representative sample of the requests that your application serves, X-Ray applies a sampling algorithm to determine which requests get traced. This can be changed by editing the sampling rules.
Metrics

For your state machine, X-Ray will meter invocation time, state transition time, the overall execution time of Step Functions, and variances in this execution time. This information can be accessed through the X-Ray console.

Analytics

The AWS X-Ray Analytics console is an interactive tool for interpreting trace data. You can refine the active dataset with increasingly granular filters by clicking the graphs and the panels of metrics and fields that are associated with the current trace set. This lets you analyze how your state machine is performing, and quickly locate and identify performance issues.

For detailed information about X-Ray analytics, see Interacting with the AWS X-Ray Analytics console

Step Functions service integrations and X-Ray

Some of the AWS services that integrate with Step Functions provide integration with AWS X-Ray by adding a tracing header to requests, running the X-Ray daemon, or making sampling decisions and uploading trace data to X-Ray. Others must be instrumented using the AWS X-Ray SDK. A few do not yet support X-Ray integration. X-Ray integration is necessary to provide complete trace data when using a service integration with Step Functions.

Native X-Ray support

Service integrations with native X-Ray support include:

- Amazon Simple Notification Service
- Amazon Simple Queue Service
- AWS Lambda
- AWS Step Functions

Instrumentation required

Service integrations that require X-Ray instrumentation:

- Amazon Elastic Container Service
- AWS Batch
- AWS Fargate

Client-side trace only

Other service integrations do not support X-Ray traces. However, client side traces can still be collected:

- Amazon DynamoDB
- Amazon EMR
- Amazon SageMaker
- AWS CodeBuild
- AWS Glue
**Viewing the X-Ray console**

X-Ray receives data from services as segments. X-Ray groups segments that have a common request into traces. X-Ray processes the traces to generate a service graph that provides a visual representation of your application.

After you start your state machine's execution, you can view its X-Ray traces by choosing the **X-Ray trace map** link in the **Execution details** section.

<table>
<thead>
<tr>
<th>Execution details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Status</strong></td>
</tr>
<tr>
<td>Succeeded</td>
</tr>
<tr>
<td><strong>Execution ARN</strong></td>
</tr>
<tr>
<td>arn:aws:states:sa-east-1:execution.WaitForCallbackStateMachine</td>
</tr>
</tbody>
</table>

After you have enabled X-Ray for your state machine, you can view tracing information for its executions in the X-Ray console.

**Viewing X-Ray tracing information for Step Functions**

The following steps illustrate what kind of information you can see in the console after you enable X-Ray and run an execution. X-Ray traces for the Callback Pattern Example (Amazon SQS, Amazon SNS, Lambda) (p. 369) sample project are shown.

**Traces**

After the an execution has finished, you can navigate to the X-Ray console, where you will see the X-Ray **Traces** page. This displays an overview of the service map as well as trace and segment information for your state machine.
Service map

The service map in the X-Ray console helps you to identify services where errors are occurring, where there are connections with high latency, or see traces for requests that were unsuccessful.

On the trace map, you can choose a service node to view requests for that node, or an edge between two nodes to view requests that traveled that connection. Here, the `WaitForCallback` node has been selected, and you can view additional information about its execution and response status.
You can see how the X-Ray service map correlates to the state machine. There is a service map node for each service integration that is called by Step Functions, provided it supports X-Ray.

**Segments and subsegments**

A **trace** is a collection of segments generated by a single request. Each segment provides the resource's name, details about the request, and details about the work done. On the Traces page, you can see the segments and, if expanded, its corresponding subsegments. You can choose a segment or subsegment to view detailed information about it.

Choose each of the tabs to see how information for segments and subsegments is displayed.

**Overview of Segments**

An overview of segments and subsegments for this state machine. There is a different segment for each node on the service map.
Choosing a segment provides the resource's name, details about the request, and details about the work done.

A segment can break down the data about the work done into subsegments. Choosing a subsegment lets you view more granular timing information and details. A subsegment can contain additional details about a call to an AWS service, an external HTTP API, or an SQL database.
Analytics

The AWS X-Ray Analytics console is an interactive tool for interpreting trace data. You can use this to more easily understand how your state machine is performing. The console enables you to explore, analyze, and visualize traces through interactive response time and time-series graphs. This can help you quickly locate performance and latency issues.

You can refine the active dataset with increasingly granular filters by clicking the graphs and the panels of metrics and fields that are associated with the current trace set.
Configuration

You can configure sampling and encryption options from the X-Ray console.

Sampling

Choose **Sampling** to view details about the sampling rate and configuration. You can change the sampling rules to control the amount of data that you record, and modify sampling behavior to suit your specific requirements.
Encryption

Choose Encryption to modify the encryption settings. You can use the default setting, where X-Ray encrypts traces and data at rest, or, if needed, you can choose a customer master key. Standard AWS KMS charges apply in the latter case.

What if there is no data in the trace map or service map?

If you have enabled X-Ray, but can't see any data in the X-Ray console, check that:

- Your IAM roles are set up correctly to allow writing to X-Ray.
- Sampling rules allow sampling of data.
- Since there can be a short delay before newly created or modified IAM roles are applied, check the trace or service maps again after a few minutes.
- If you see Data Not Found in the X-Ray Traces panel, check your IAM account settings and ensure that AWS Security Token Service is enabled for the intended region. For more information, see the IAM user guide.
Security in AWS Step Functions

This section provides information about AWS Step Functions security and authentication.

Topics
- Data protection in AWS Step Functions (p. 523)
- Identity and Access Management in AWS Step Functions (p. 524)
- Logging and Monitoring (p. 577)
- Compliance Validation for AWS Step Functions (p. 577)
- Resilience in AWS Step Functions (p. 578)
- Infrastructure Security in AWS Step Functions (p. 578)
- Configuration and Vulnerability Analysis in AWS Step Functions (p. 579)

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.

Data protection in AWS Step Functions

The AWS shared responsibility model applies to data protection in AWS Step Functions. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the AWS Security Blog.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM). That way each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We recommend TLS 1.2 or later.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

We strongly recommend that you never put confidential or sensitive information, such as your customers’ email addresses, into tags or free-form fields such as a Name field. This includes when you
work with Step Functions or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Encryption in AWS Step Functions

Encryption at Rest

Step Functions always encrypts your data at rest. Data in AWS Step Functions is encrypted at rest using transparent server-side encryption. This helps reduce the operational burden and complexity involved in protecting sensitive data. With encryption at rest, you can build security-sensitive applications that meet encryption compliance and regulatory requirements.

Encryption in transit

Step Functions encrypts data in transit between the service and other integrated AWS services (see Using AWS Step Functions with other services (p. 271)). All data that passes between Step Functions and integrated services is encrypted using Transport Layer Security (TLS).

Identity and Access Management in AWS Step Functions

Access to AWS Step Functions requires credentials that AWS can use to authenticate your requests. Those credentials must have permissions to access AWS resources, such as retrieving event data from other AWS resources. The following sections provide details on how you can use AWS Identity and Access Management (IAM) and Step Functions to help secure your resources by controlling who can access them:

- Authentication (p. 524)
- Access Control (p. 525)

Authentication

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

- **AWS account root user** – When you sign up for AWS, you provide an email address and password that is associated with your account. These are your root credentials, and they provide complete access to all of your AWS resources.

  **Important**
  
  For security reasons, we recommend that you use the root credentials only to create an administrator, which is an IAM user with full permissions to your account. Then you can use this administrator to create other IAM users and roles with limited permissions. For more information, see IAM Best Practices and Creating an Admin User and Group in the IAM User Guide.

- **IAM user** – An IAM user is an identity within your account that has specific custom permissions (for example, permissions to send event data to a target in Step Functions). You can use an IAM user name and password to sign in to secure AWS webpages such as 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 (AWS CLI). The SDK and AWS CLI tools use the access keys to cryptographically sign your request. If you don’t use the 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 Amazon Web Services General Reference.

- **IAM role** – An IAM role is another IAM identity that you can create in your account that has specific permissions. It's similar to an IAM user, but it isn't 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 preexisting identities from AWS Directory Service, your enterprise user directory, or a web identity provider (IdP). 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.
  - **Cross-account access** – You can use an IAM role in your account to grant another account permissions to access your account's resources. For an example, see Tutorial: Delegate Access Across AWS Accounts Using IAM Roles in the IAM User Guide.
  - **AWS service access** – You can use an IAM role in your account to grant to an AWS service the permissions needed 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 stored in the 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** – Instead of storing access keys in the EC2 instance for use by applications running on the instance and making AWS API requests, you can use an IAM role to manage temporary credentials for these applications. To assign an AWS role to an EC2 instance and make it available to all of its applications, you can create an instance profile that is attached to the instance. An instance profile contains the role and enables programs running on the EC2 instance to get temporary credentials. For more information, see Using Roles for Applications on Amazon EC2 in the IAM User Guide.

Access Control

You can have valid credentials to authenticate your requests, but unless you have permissions you cannot create or access Step Functions resources. For example, you must have permissions to invoke AWS Lambda, Amazon Simple Notification Service (Amazon SNS), and Amazon Simple Queue Service (Amazon SQS) targets associated with your Step Functions rules.

The following sections describe how to manage permissions for Step Functions.

- **How AWS Step Functions Works with IAM** (p. 526)
How AWS Step Functions Works with IAM

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 for Step Functions (p. 137) in this guide enable you to take advantage of automatically generated IAM roles that are valid for the AWS Region in which you create the state machine. However, you can create your own IAM role for a state machine.

When creating an IAM policy for your state machines to use, the policy should include the permissions that you would like the state machines to assume. You can use an existing AWS managed policy that as an example or you can create a custom policy from scratch that meets your specific needs. For more information, see Creating IAM policies in the IAM User Guide.

To create your own IAM role for a state machine, follow the steps in this section.

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

Create a role for Step Functions

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

The IAM role appears in the list of roles.

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

Prevent cross-service confused deputy issue

A confused deputy is an entity (a service or an account) that is coerced by a different entity to perform an action. In AWS, cross-service impersonation can occur when one service (the calling service) calls another service (the called service). The calling service can be manipulated to use its permissions to act on another customer’s resources in a way it should not otherwise have permission to access. This type of impersonation can happen cross-account and cross-service.

To prevent confused deputies, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account. This section focuses on cross-service confused deputy prevention specific to AWS Step Functions; however, you can learn more about this topic in the confused deputy problem section of the IAM User Guide.

To limit the permissions IAM gives to Step Functions to access your resources, we recommend using the aws:SourceArn and aws:SourceAccount global condition context keys in your resource policies. Note that if you use both of these global condition context keys, and the aws:SourceArn value contains the AWS account ID, the aws:SourceAccount value and the AWS account in the aws:SourceArn value must use the same AWS account ID when used in the same policy statement.
The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don’t know the full ARN, or if you’re specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcards (**) for the unknown portions of the ARN. For example, 

```
arn:aws:states:*:111122223333:**
```

Here’s an example of a trusted policy that shows how you can use `aws:SourceArn` and `aws:SourceAccount` with Step Functions to prevent the confused deputy issue.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect":"Allow",
      "Principal":{"
        "Service": ["
          "states.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {"aws:SourceAccount": "111122223333"}
      }
    }
  ]
}
```

### Attach an Inline Policy

Step Functions can control other services directly in a Task state. Attach inline policies to allow Step Functions to access the API actions of the services you need to control.

1. Open the IAM console, choose Roles, search for your Step Functions role, and select that role.
2. Select Add inline policy.
3. Use the Visual editor or the JSON tab to create policies for your role.

For more information about how AWS Step Functions can control other AWS services, see Using AWS Step Functions with other services (p. 271).

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

The default managed policies in IAM, such as ReadOnly, don’t fully cover all types of AWS Step Functions permissions. This section describes these different types of permissions and provides some example configurations.

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. 528)
Apply to components of the API that don't act on a specific resource.

State Machine-Level Permissions (p. 528)
Apply to all API components that act on a specific state machine.

Execution-Level Permissions (p. 529)
Apply to all API components that act on a specific execution.

Activity-Level Permissions (p. 529)
Apply to all API components that act on a specific activity or on a particular instance of an activity.

**Service-Level Permissions**

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

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

**State Machine-Level Permissions**

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

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeStateMachine",
        "states:StartExecution",
        "states:DeleteStateMachine",
        "states:DeleteActivity",
        "states:StopExecution",
        "states:PutActivityTaskFailed",
        "states:PutTaskFailed",
        "states:PutTaskFailedWithsuppressDeadLetterTaskResult",
        "states:PutTaskFailedWithFailureReason",
        "states:PutTaskSucceeded",
        "states:PutTaskSucceededWithFailureReason",
        "states:PutTaskSucceededWithTaskRunId",
        "states:PutTaskSucceededWithTaskRunIdWithFailureReason",
        "states:PutTaskSucceededWithTaskRunIdWithTaskRunId",
        "states:PutTaskSucceededWithTaskRunIdWithTaskRunIdWithFailureReason",
        "states:PutTaskSucceededWithTaskRunIdWithTaskRunIdWithTaskRunIdWithFailureReason",
        "states:PutTaskSucceededWithTaskRunIdWithTaskRunIdWithTaskRunIdWithTaskRunIdWithFailureReason"
      ],
      "Resource": [
        "arn:aws:states:*:*:*",
        "arn:aws:states:*:*:*:task-run-id-1",
        "arn:aws:states:*:*:*:task-run-id-2",
        "arn:aws:states:*:*:*:task-run-id-3"
      ]
    }
  ]
}
```
Execution-Level Permissions

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

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeExecution",
        "states:DescribeStateMachineForExecution",
        "states:GetExecutionHistory",
        "states:StopExecution"
      ],
      "Resource": [
        "arn:aws:states:*:*:execution:*:ExecutionPrefix*"
      ]
    }
  ]
}
```

Activity-Level Permissions

This permission level applies to all the API actions that act on a specific activity or on a particular instance of it. These API operations require the ARN of the activity or the token of the instance as part of the request, such as `DeleteActivity`, `DescribeActivity`, `GetActivityTask`, `SendTaskSuccess`, `SendTaskFailure`, and `SendTaskHeartbeat`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeActivity",
        "states:DeleteActivity",
        "states:GetActivityTask",
        "states:SendTaskSuccess",
        "states:SendTaskFailure",
        "states:SendTaskHeartbeat"
      ],
      "Resource": [
        "arn:aws:states:*:*:activity:ActivityPrefix*"
      ]
    }
  ]
}
```
Amazon VPC Endpoints for Step Functions

If you use Amazon Virtual Private Cloud (Amazon VPC) to host your AWS resources, you can establish a connection between your Amazon VPC and AWS Step Functions workflows. You can use this connection with your Step Functions workflows without crossing the public internet. Amazon VPC endpoints are supported by Standard Workflows, Express Workflows, and Synchronous Express Workflows.

Amazon VPC lets you launch AWS resources in a custom virtual network. You can use a VPC to control your network settings, such as the IP address range, subnets, route tables, and network gateways. For more information about VPCs, see the Amazon VPC User Guide.

To connect your Amazon VPC to Step Functions, you must first define an interface VPC endpoint, which lets you connect your VPC to other AWS services. The endpoint provides reliable, scalable connectivity, without requiring an internet gateway, network address translation (NAT) instance, or VPN connection. For more information, see Interface VPC Endpoints (AWS PrivateLink) in the Amazon VPC User Guide.

Creating the Endpoint

You can create an AWS Step Functions endpoint in your VPC using the AWS Management Console, the AWS Command Line Interface (AWS CLI), an AWS SDK, the AWS Step Functions API, or AWS CloudFormation.

For information about creating and configuring an endpoint using the Amazon VPC console or the AWS CLI, see Creating an Interface Endpoint in the Amazon VPC User Guide.

**Note**
When you create an endpoint, specify Step Functions as the service that you want your VPC to connect to. In the Amazon VPC console, service names vary based on the AWS Region. For example, if you choose US East (N. Virginia), the service name for Standard Workflows and Express Workflows is com.amazonaws.us-east-1.states, and the service name for Synchronous Express Workflows is com.amazonaws.us-east-1.sync-states.

**Note**
It's possible to use VPC Endpoints without overriding the endpoint in the SDK through Private DNS. However, if you want to override the endpoint in the SDK for Synchronous Express Workflows, you need to set DisableHostPrefixInjection configuration to true. Example (Java SDK V2):

```java
SfnClient.builder()
    .endpointOverride(URI.create("https://vpce-{vpceId}.sync-states.us-east-1.vpce.amazonaws.com"))
    .overrideConfiguration(ClientOverrideConfiguration.builder()
        .advancedOptions(ImmutableMap.of(SdkAdvancedClientOption.DISABLE_HOST_PREFIX_INJECTION, true))
        .build())
    .build();
```

For information about creating and configuring an endpoint using AWS CloudFormation, see the AWS::EC2::VPCEndpoint resource in the AWS CloudFormation User Guide.

Amazon VPC Endpoint Policies

To control connectivity access to Step Functions you can attach an AWS Identity and Access Management (IAM) endpoint policy while creating an Amazon VPC endpoint. You can create complex IAM rules by attaching multiple endpoint policies. For more information, see:

- Amazon Virtual Private Cloud Endpoint Policies for Step Functions (p. 531)
- Creating Granular IAM Permissions for Non-Admin Users (p. 527)
Amazon Virtual Private Cloud Endpoint Policies for Step Functions

You can create an Amazon VPC endpoint policy for Step Functions in which you specify the following:

- The principal that can perform actions.
- The actions that can be performed.
- The resources on which the actions can be performed.

The following example shows an Amazon VPC endpoint policy that allows one IAM user to create state machines, and denies all IAM users permission to delete state machines. The example policy also grants all IAM users execution permission.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "*Execution",
      "Resource": "*",
      "Effect": "Allow",
      "Principal": "*
    },
    {
      "Action": "states:CreateStateMachine",
      "Resource": "*",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:user/MyUser"
      }
    },
    {
      "Action": "states:DeleteStateMachine",
      "Resource": "*",
      "Effect": "Deny",
      "Principal": "*"
    }
  ]
}
```

For more information about creating endpoint policies, see the following:

- Creating Granular IAM Permissions for Non-Admin Users (p. 527)
- Controlling Access to Services with VPC Endpoints

IAM Policies for integrated services

When you create a state machine in the AWS Step Functions console, Step Functions produces an AWS Identity and Access Management (IAM) policy based on the resources used in your state machine definition as follows:

- If your state machine uses one of the Optimized integrations, it will create a policy with the necessary permissions and roles for your state machine.
- If your state machine uses one of the AWS SDK integrations, an IAM role with partial permissions will be created. Afterwards, you can use the IAM console to add any missing role policies.
The following examples show how Step Functions generates an IAM policy based on your state machine definition. Items in the example code such as `[[resourceName]]` are replaced with the static resources listed in your state machine definition. If you have multiple static resources, there will be an entry for each in the IAM role.

Dynamic vs. Static Resources

Static resources are defined directly in the task state of your state machine. When you include the information about the API actions you call directly in your task states, Step Functions creates an IAM role for only those resources.

Dynamic resources are those that are passed in to your state input, and accessed using a Path (see Paths (p. 57)). If you are passing dynamic resources to your task, Step Functions will create a more privileged policy that specifies: "Resource": ".*".

Additional permissions for tasks using the Run a Job pattern

For tasks that use the Run a Job (p. 292) pattern (those ending in .sync), additional permissions are needed to monitor and receive a response from the API actions of connected services. The related policies include more permissions than for tasks that use the Request Response or Wait for Callback patterns. See Service Integration Patterns (p. 291) for information about synchronous tasks.

Step Functions uses two methods to monitor a job's status when a job is run on a connected service, polling and events.

Polling requires permission for Describe or Get API actions, such as `ecs:DescribeTasks` or `glue:GetJobRun`. If these permissions are missing from your role, then Step Functions may be unable to determine the status of your job. This is because some Run a Job (.sync) service integrations do not support EventBridge events, and some services only send events on a best-effort basis.

Events sent from AWS services to Amazon EventBridge are directed to Step Functions using a managed rule, and require permissions for `events:PutTargets`, `events:PutRule`, and `events:DescribeRule`. If these permissions are missing from your role, there may be a delay before Step Functions becomes aware of the completion of your job. For more information about EventBridge events, see Events from AWS services.

Note

For Run a Job (sync) tasks that support both polling and events, your task may still complete properly using events. This can occur even if your role lacks the required permissions for polling. In this case, you may not immediately notice that the polling permissions are incorrect or missing. In the rare instance that the event fails to be delivered to or processed by Step Functions, your execution could become stuck. To verify that your polling permissions are configured correctly, you can run an execution in an environment without EventBridge events in the following ways:

- Delete the managed rule from EventBridge, which is responsible for forwarding events to Step Functions. This managed rule is shared by all state machines in your account, so you should perform this action only in a test or development account to avoid any unintentional impact on other state machines. You can identify the specific managed rule to delete by inspecting the `Resource` field used for `events:PutRule` in the policy template for the target service. The managed rule will be recreated the next time you create or update a state machine that uses that service integration. For more information on deleting EventBridge rules, see Disabling or deleting a rule.
- Test with Step Functions Local, which does not support the use of events to complete Run a Job (sync) tasks. To use Step Functions Local, assume the IAM role used by your state machine. You may need to edit the Trust Relationship. Set the `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN` environment variables to the assumed role's values, then launch Step Functions Local using `java -jar`
StepFunctionsLocal.jar. Last, use the AWS CLI with the --endpoint-url parameter to create a state machine, start an execution, and get the execution history. For more information, see Testing Step Functions State Machines Locally (p. 247).

If a task that uses the Run a Job (.sync) pattern is stopped, Step Functions will make a best-effort attempt to cancel the task. This requires permission to Cancel, Stop, Terminate, or Delete API actions, such as batch:TerminateJob or eks:DeleteCluster. If these permissions are missing from your role, Step Functions will be unable to cancel your task and you may accrue additional charges while it continues to run. For more information on stopping tasks, see Run a Job (p. 292).

Policy templates used to create IAM roles

The following topics include the policy templates used when you choose to have Step Functions create a new role for you.

Note
Review these templates to understand how Step Functions creates your IAM policies, and as an example of how to manually create IAM policies for Step Functions when working with other AWS services. For more information about Step Functions service integrations, see Using AWS Step Functions with other services (p. 271).

Topics
- AWS Lambda (p. 533)
- AWS Batch (p. 534)
- Amazon DynamoDB (p. 535)
- Amazon ECS/AWS Fargate (p. 536)
- Amazon Simple Notification Service (p. 538)
- Amazon Simple Queue Service (p. 538)
- AWS Glue (p. 539)
- Amazon SageMaker (p. 540)
- Amazon EMR (p. 546)
- Amazon EMR on EKS (p. 551)
- AWS CodeBuild (p. 555)
- X-Ray (p. 564)
- Amazon Athena (p. 564)
- Amazon EKS (p. 570)
- Amazon API Gateway (p. 573)
- AWS Glue DataBrew (p. 573)
- Amazon EventBridge (p. 574)
- AWS Step Functions (p. 575)
- Activities or No Tasks (p. 576)

AWS Lambda

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)
AWS Step Functions generates an IAM policy based on your state machine definition. For a state machine with two AWS Lambda task states that call `function1` and `function2`, a policy with `lambda:Invoke` permissions for the two functions must be used.

This is shown in the following example.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:[]:[]:function:[]",
        "arn:aws:lambda:[]:[]:function:[]"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "batch:SubmitJob",
        "batch:DescribeJobs",
        "batch:TerminateJob"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:[]:[]:rule/StepFunctionsGetEventsForBatchJobsRule"
      ]
    }
  ]
}
```

**AWS Batch**

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

Because AWS Batch provides partial support for resource-level access control, you must use "Resource": "*".

**Run a Job (.sync)**

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "batch:SubmitJob",
        "batch:DescribeJobs",
        "batch:TerminateJob"
      ],
      "Resource": "*
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:[]:[]:rule/StepFunctionsGetEventsForBatchJobsRule"
      ]
    }
  ]
}
```
IAM Policies for integrated services

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

Static resources

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb:DeleteItem"
      ],
      "Resource": ["arn:aws:dynamodb:::table/"[tableName]"
    }
  ]
}
```

Dynamic resources

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb:DeleteItem"
      ],
      "Resource": ["arn:aws:dynamodb:region::account:table/tableName"
    }
  ]
}
```
Amazon ECS/AWS Fargate

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

Because the value for TaskId is not known until the task is submitted, Step Functions creates a more privileged "Resource": 

Note
You can only stop Amazon Elastic Container Service (Amazon ECS) tasks that were started by Step Functions, despite the "*" IAM policy.

Run a Job (.sync)

Static resources

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecs:RunTask"
      ],
      "Resource": ["arn:aws:ecs:[region]:[[accountId]]:task-definition/[[taskDefinition]]"
    },
    {
      "Effect": "Allow",
      "Action": [
        "ecs:StopTask",
        "ecs:DescribeTasks"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": ["arn:aws:events:[region]:[[accountId]]:rule/StepFunctionsGetEventsForECSTaskRule"
    }
  ]
}
```
Dynamic resources

```json
{
   "Version": "2012-10-17",
   "Statement": [
       {
           "Effect": "Allow",
           "Action": [
               "ecs:RunTask",
               "ecs:StopTask",
               "ecs:DescribeTasks"
           ],
           "Resource": "*"
       },
       {
           "Effect": "Allow",
           "Action": [
               "events:PutTargets",
               "events:PutRule",
               "events:DescribeRule"
           ],
           "Resource": [
               "arn:aws:events:[region]:[accountId]:rule/StepFunctionsGetEventsForECSTaskRule"
           ]
       }
   ]
}
```

Request Response and Callback (.waitForTaskToken)

Static resources

```json
{
   "Version": "2012-10-17",
   "Statement": [
       {
           "Effect": "Allow",
           "Action": [
               "ecs:RunTask"
           ],
           "Resource": [
               "arn:aws:ecs:[region]:[accountId]:task-definition/[[taskDefinition]]"
           ]
       }
   ]
}
```

Dynamic resources

```json
{
   "Version": "2012-10-17",
   "Statement": [
       {
           "Effect": "Allow",
           "Action": [
               "ecs:RunTask"
           ],
           "Resource": "*"
       }
   ]
}
```
Amazon Simple Notification Service

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

Static resources

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "sns:Publish"
         ],
         "Resource": [
            "arn:aws:sns:{{region}}:{{accountId}}:{{topicName}}"
         ]
      }
   ]
}

Resources based on a Path, or publishing to TargetArn or PhoneNumber

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": ["sns:Publish"],
         "Resource": "*"
      }
   ]
}

Amazon Simple Queue Service

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

Static resources

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
      }
   ]
}
```
IAM Policies for integrated services

Dynamic resources

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [ "sqs:SendMessage" ],
            "Resource": [ "arn:aws:sqs:[]:[]:[]" ]
        }
    ]
}
```

AWS Glue

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

AWS Glue does not have resource-based control.

Run a Job (.sync)

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [ "glue:StartJobRun", "glue:GetJobRun", "glue:GetJobRuns", "glue:BatchStopJobRun" ],
            "Resource": "*"
        }
    ]
}
```

Request Response and Callback (.waitForTaskToken)

```json
{}
```
Amazon SageMaker

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

**Note**

For these examples, `[[roleArn]]` refers to the Amazon Resource Name (ARN) of the IAM role that SageMaker uses to access model artifacts and docker images for deployment on ML compute instances, or for batch transform jobs. For more information, see Amazon SageMaker Roles.

**Topics**

- CreateTrainingJob (p. 540)
- CreateTransformJob (p. 543)

CreateTrainingJob

**Static resources**

Run a Job (sync)

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "sagemaker:CreateTrainingJob",
            "sagemaker:DescribeTrainingJob",
            "sagemaker:StopTrainingJob"
         ],
         "Resource": [
            "arn:aws:sagemaker:{{region}}:{{accountId}}:training-job/[[trainingJobName]]*
         ]
      },
      {
         "Effect": "Allow",
         "Action": ["sagemaker:ListTags"],
         "Resource": ["*
```
IAM Policies for integrated services

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:{{region}}:{{accountId}}:training-job/{{trainingJobName}}*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "{{roleArn}}"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}

Request Response and Callback (.waitForTaskToken)

{ "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:{{region}}:{{accountId}}:training-job/{{trainingJobName}}*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "{{roleArn}}"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}
Dynamic resources

.sync or .waitForTaskToken

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["sagemaker:CreateTrainingJob", "sagemaker:DescribeTrainingJob", "sagemaker:StopTrainingJob"],
            "Resource": ["arn:aws:sagemaker:{{region}}:{{accountId}}:training-job/*"]
        },
        {
            "Effect": "Allow",
            "Action": ["sagemaker:ListTags"],
            "Resource": ["*"]
        },
        {
            "Effect": "Allow",
            "Action": ["iam:PassRole"],
            "Resource": ["{{roleArn}}"]
        },
        {
            "Effect": "Allow",
            "Action": ["events:PutTargets", "events:PutRule", "events:DescribeRule"],
            "Resource": ["arn:aws:events:{{region}}:{{accountId}}:rule/StepFunctionsGetEventsForSageMakerTrainingJobsRule"]
        }
    ]
}
```
Request Response and Callback (.waitForTaskToken)

```
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Effect": "Allow",
        "Action": [ "sagemaker:CreateTrainingJob"
                    ],
        "Resource": [ "arn:aws:sagemaker:[region]:[accountId]:training-job/*" ]
    },
    {
        "Effect": "Allow",
        "Action": [ "sagemaker:ListTags"
                    ],
        "Resource": [ "*" ]
    },
    {
        "Effect": "Allow",
        "Action": [ "iam:PassRole"
                    ],
        "Resource": [ "[roleArn]" ]
    },
    "Condition": {
        "StringEquals": {
            "iam:PassedToService": "sagemaker.amazonaws.com"
        }
    }
    }
}   
```

CreateTransformJob

**Note**
AWS Step Functions will not automatically create a policy for `CreateTransformJob` when you create a state machine that integrates with SageMaker. You must attach an inline policy to the created role based on one of the following IAM examples.

**Static resources**

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Effect": "Allow",
        "Action": [ "sagemaker:CreateTransformJob",
                    "sagemaker:DescribeTransformJob",
                    "sagemaker:StopTransformJob"
                    ],
        "Resource": [ 
```
"arn:aws:sagemaker:[]:[]:transform-job/[]"
},
{
  "Effect": "Allow",
  "Action": [
    "sagemaker:ListTags"
  ],
  "Resource": ["*"]
},
{
  "Effect": "Allow",
  "Action": [
    "iam:PassRole"
  ],
  "Resource": ["[]"]
},
"Condition": {
  "StringEquals": {
    "iam:PassedToService": "sagemaker.amazonaws.com"
  }
}
},
{
  "Effect": "Allow",
  "Action": [
    "events:PutTargets",
    "events:PutRule",
    "events:DescribeRule"
  ],
  "Resource": [
    "arn:aws:events:[]:[]:rule/StepFunctionsGetEventsForSageMakerTransformJobsRule"
  ]
]
}

Request Response and Callback (.waitForTaskToken)

{"Version": "2012-10-17",
 "Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "sagemaker:CreateTransformJob"
    ],
    "Resource": [
      "arn:aws:sagemaker:[]:[]:transform-job/[]"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "sagemaker:ListTags"
    ],
    "Resource": ["*"]
  }
]}
IAM Policies for integrated services

Dynamic resources

Run a Job (.sync)

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "sagemaker:CreateTransformJob",
            "sagemaker:DescribeTransformJob",
            "sagemaker:StopTransformJob"
         ],
         "Resource": [
            "arn:aws:sagemaker:[]:transform-job/*"
         ],
         "Condition": {
            "StringEquals": {
               "iam:PassedToService": "sagemaker.amazonaws.com"
            }
         }
      },
      {
         "Effect": "Allow",
         "Action": [
            "sagemaker:ListTags"
         ],
         "Resource": [
            "*"
         ],
         "Condition": {
            "StringEquals": {
               "iam:PassedToService": "sagemaker.amazonaws.com"
            }
         }
      },
      {
         "Effect": "Allow",
         "Action": [
            "iam:PassRole"
         ],
         "Resource": [
            "[roleArn]"
         ],
         "Condition": {
            "StringEquals": {
               "iam:PassedToService": "sagemaker.amazonaws.com"
            }
         }
      }
   ]
}
```
"Action": [
    "events:PutTargets",
    "events:PutRule",
    "events:DescribeRule"
],
"Resource": [
    "arn:aws:events:{{region}}:{{accountId}}:rule/StepFunctionsGetEventsForSageMakerTransformJobsRule"
]
}

Request Response and Callback (.waitForTaskToken)

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sagemaker:CreateTransformJob"
            ],
            "Resource": [
                "arn:aws:sagemaker:{{region}}:{{accountId}}:transform-job/*"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "sagemaker.amazonaws.com"
                }
            }
        }
    ]
}

Amazon EMR

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)
addStep

Static resources

{  
  "Version": "2012-10-17",
  "Statement": [
    
    
    
    
    ],
},

{  
  "Effect": "Allow",
  "Action": [
    "events:PutTargets",
    "events:PutRule",
    "events:DescribeRule"
  ],
  "Resource": [
    "arn:aws:events:[[region]]:[[accountId]]:rule/StepFunctionsGetEventForEMRAddJobFlowStepsRule"
  ]
}

Dynamic resources

{  
  "Version": "2012-10-17",
  "Statement": [
    
    
    
    ],
},

{  
  "Effect": "Allow",
  "Action": [
    "events:PutTargets",
    "events:PutRule",
    "events:DescribeRule"
  ],
  "Resource": [
    "arn:aws:events:[[region]]:[[accountId]]:rule/StepFunctionsGetEventForEMRAddJobFlowStepsRule"
  ]
}
cancelStep

Static resources

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "elasticmapreduce:CancelSteps",
         "Resource": ["arn:aws:elasticmapreduce:[]:[]:cluster/[]"]
      }
   ]
}
```

Dynamic resources

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "elasticmapreduce:CancelSteps",
         "Resource": "arn:aws:elasticmapreduce::*:*:cluster/*"
      }
   ]
}
```

createCluster

Static resources

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": ["elasticmapreduce:RunJobFlow",
                     "elasticmapreduce:DescribeCluster",
                     "elasticmapreduce:TerminateJobFlows"],
         "Resource": "*"
      },
      {
         "Effect": "Allow",
         "Action": "iam:PassRole",
         "Resource": ["arn:aws:iam:::role/[]"]
      },
      {
         "Effect": "Allow",
         "Action": ["events:PutTargets",
                     "events:PutRule",
                     "events:DescribeRule"],
         "Resource": ["arn:aws:events:[]:[]:rule/[]"]
      }
   ]
}
```
**setClusterTerminationProtection**

**Static resources**

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "elasticmapreduce:SetTerminationProtection",
         "Resource": 
            "arn:aws:elasticmapreduce:[$.region]:[$accountId]:cluster/$clusterId"
      }
   ]
}
```

**Dynamic resources**

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "elasticmapreduce:SetTerminationProtection",
         "Resource": "arn:aws:elasticmapreduce::*:*:cluster/*"
      }
   ]
}
```

**modifyInstanceFleetByName**

**Static resources**

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [ "elasticmapreduce:ModifyInstanceFleet", "elasticmapreduce:ListInstanceFleets" ],
         "Resource": 
            "arn:aws:elasticmapreduce:[$region]:[$accountId]:cluster/$clusterId"
      }
   ]
}
```

**Dynamic resources**
IAM Policies for integrated services

modifyInstanceGroupByName

Static resources

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elasticmapreduce:ModifyInstanceGroups",
                "elasticmapreduce:ListInstanceGroups"
            ],
            "Resource": [
                "arn:aws:elasticmapreduce:[]region[]:[]accountId[]:cluster/[]clusterId"
            ]
        }
    ]
}
```

Dynamic resources

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elasticmapreduce:ModifyInstanceGroups",
                "elasticmapreduce:ListInstanceGroups"
            ],
            "Resource": "*
        }
    ]
}
```

terminateCluster

Static resources

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["elasticmapreduce:ModifyInstanceFleet", "elasticmapreduce:ListInstanceFleets"
            ],
            "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
        }
    ]
}
```
"elasticmapreduce:TerminateJobFlows",
"elasticmapreduce:DescribeCluster"
],
"Resource": [
  "arn:aws:elasticmapreduce:{{region}}:{{accountId}}:cluster/{{clusterId}}"
],
"Effect": "Allow",
"Action": [
  "events:PutTargets",
  "events:PutRule",
  "events:DescribeRule"
],
"Resource": [
  "arn:aws:events:{{region}}:{{accountId}}:rule/StepFunctionsGetEventForEMRTerminateJobFlowsRule"
]
}
]
}

**Dynamic resources**

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:TerminateJobFlows",
        "elasticmapreduce:DescribeCluster"
      ],
      "Resource": "arn:aws:elasticmapreduce::*:*:cluster/*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:{{region}}:{{accountId}}:rule/StepFunctionsGetEventForEMRTerminateJobFlowsRule"
      ]
    }
  ]
}
```

**Amazon EMR on EKS**

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

**CreateVirtualCluster**

**Resources**
IAM Policies for integrated services

DeleteVirtualCluster

Static resources

Run a Job (sync)

Request Response
Dynamic resources

Run a Job (sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-containers:DeleteVirtualCluster",
        "emr-containers:DescribeVirtualCluster"
      ],
      "Resource": [
        "arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/*"
      ]
    }
  ]
}
```

Request Response

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["emr-containers:DeleteVirtualCluster"],
      "Resource": ["arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/*"]
    }
  ]
}
```

StartJobRun

Static resources

Run a Job (sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "emr-containers:StartJobRun",
      "Resource": ["arn:aws:emr-containers:{{region}}:{{accountId}}:/virtualclusters/[[virtualClusterId]]"],
      "Condition": {
        "StringEquals": {
          "emr-containers:ExecutionRoleArn": ["[[executionRoleArn]]"]
        }
      }
    }
  ]
}
```
IAM Policies for integrated services

Request Response

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "emr-containers:StartJobRun",
      "Resource": [
        "arn:aws:emr-containers:{region}:{accountId}:virtualclusters/{{virtualClusterId}}
      ],
      "Condition": {
        "StringEquals": {
          "emr-containers:ExecutionRoleArn": [
            "{{executionRoleArn}}"
          ]
        }
      }
    }
  ]
}
```

Dynamic resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "emr-containers:StartJobRun",
      "Resource": [
        "arn:aws:emr-containers:{region}:{accountId}:virtualclusters/*
      ],
      "Condition": {
        "StringEquals": {
          "emr-containers:ExecutionRoleArn": [
            "{{executionRoleArn}}"
          ]
        }
      }
    }
  ]
}
```
AWS CodeBuild

These example templates show how AWS Step Functions generates IAM policies based on the resources
in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

Resources:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": ["sns:Publish"],
      "Effect": "Allow"
    },
    {
      "Action": [
```

{  "Version": "2012-10-17",
   "Statement": [
     {
       "Action": ["emr-containers:DescribeJobRun",
                   "emr-containers:CancelJobRun"
                  ],
       "Resource": [
         "arn:aws:emr-containers:{{region}}:{{accountId}}:virtualclusters/*"
       ]
     },
     {
       "Action": ["emr-containers:StartJobRun",
                   "emr-containers:StopJobRun"
                  ],
       "Resource": [
         "arn:aws:emr-containers:{{region}}:{{accountId}}:virtualclusters/*"
       ],
       "Condition": {
         "StringEquals": {
           "emr-containers:ExecutionRoleArn": ["[[executionRoleArn]]"
         }
       }
     }
   ]
}
"codebuild:StartBuild",
"codebuild:StopBuild",
"codebuild:BatchGetBuilds",
"codebuild:BatchGetReports"
],
"Resource": "*",
"Effect": "Allow"
},
{
"Action": [
"events:PutTargets",
"events:PutRule",
"events:DescribeRule"
],
"Resource": [
"arn:aws:events:sa-east-1:123456789012:rule/StepFunctionsGetEventForCodeBuildStartBuildRule"
],
"Effect": "Allow"
}
]
}

**StartBuild**

**Static resources**

Run a Job (.sync)

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuild",
        "codebuild:StopBuild",
        "codebuild:BatchGetBuilds"
      ],
      "Resource": [
        "arn:aws:codebuild:::[region]:[[accountId]]:project/[[projectName]]"
      ],
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:::[region]:[[accountId]]:rule/StepFunctionsGetEventForCodeBuildStartBuildRule"
      ]
    }
  ]
}
```

**Request Response**

```json
{
  "Version": "2012-10-17",
  "Id": "4ce3f85b-4a91-4402-bf5c-736b37df47e9",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuild",
        "codebuild:StopBuild",
        "codebuild:BatchGetBuilds",
        "codebuild:BatchGetReports"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:::rule/StepFunctionsGetEventForCodeBuildStartBuildRule"
      ],
      "Effect": "Allow"
    }
  ]
}
```
"Statement": [  
  {  
    "Effect": "Allow",  
    "Action": [  
      "codebuild:StartBuild"  
    ],  
    "Resource": [  
      "arn:aws:codebuild:[region]:[account]:project/[[projectName]]"  
    ]  
  }  
]}

Dynamic resources

Run a Job (sync)

{

  "Version": "2012-10-17",
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "codebuild:StartBuild",
        "codebuild:StopBuild",
        "codebuild:BatchGetBuilds"
      ],  
      "Resource": [  
        "arn:aws:codebuild:[region]:*:project/*"
      ]  
    },
    {  
      "Effect": "Allow",  
      "Action": [  
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],  
      "Resource": [  
        "arn:aws:events:[region]:[account]:rule/StepFunctionsGetEventForCodeBuildStartBuildRule"
      ]  
    }
  ]
}

Request Response

{

  "Version": "2012-10-17",
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "codebuild:StartBuild"
      ],  
      "Resource": [  
        "arn:aws:codebuild:[region]:*:project/*"
      ]  
    }
  ]
}
StopBuild

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["codebuild:StopBuild"],
      "Resource": ["arn:aws:codebuild:[region]:[accountId]:project/[projectName]"
    }
  ]
}
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["codebuild:StopBuild"],
      "Resource": ["arn:aws:codebuild:[region]*:project/*"
    }
  ]
}
```

BatchDeleteBuilds

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["codebuild:BatchDeleteBuilds"],
      "Resource": ["arn:aws:codebuild:[region]:[accountId]:project/[projectName]"
    }
  ]
}
```

Dynamic resources

```
IAM Policies for integrated services

**BatchGetReports**

**Static resources**

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": ["codebuild:BatchGetReports"],
         "Resource": ["arn:aws:codebuild:[[region]]::report-group/[[reportName]]"
      }
   ]
}
```

**Dynamic resources**

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": ["codebuild:BatchGetReports"],
         "Resource": ["arn:aws:codebuild:[[region]]::report-group/"
      }
   ]
}
```

**StartBuildBatch**

**Static resources**

Run a Job (.sync)

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": ["codebuild:StartBuildBatch"],
         "Resource": ["arn:aws:codebuild:[[region]]::project/"
      }
   ]
}
```
"Effect": "Allow",
"Action": [
  "codebuild:StartBuildBatch",
  "codebuild:StopBuildBatch",
  "codebuild:BatchGetBuildBatches"
],
"Resource": [
  "arn:aws:codebuild:[region]:[accountId]:project/[projectName]"
],
{
  "Effect": "Allow",
  "Action": [
    "events:PutTargets",
    "events:PutRule",
    "events:DescribeRule"
  ],
  "Resource": [
    "arn:aws:events:[region]:[accountId]:rule/StepFunctionsGetEventForCodeBuildStartBuildBatchRule"
  ]
}
}

Request Response

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuildBatch"
      ],
      "Resource": [
        "arn:aws:codebuild:[region]:[accountId]:project/[projectName]"
      ]
    }
  ]
}

Dynamic resources
Run a Job (sync)

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuildBatch",
        "codebuild:StopBuildBatch",
        "codebuild:BatchGetBuildBatches"
      ],
      "Resource": [
        "arn:aws:codebuild:[region]:[accountId]:project/*"
      ]
    }
  ]
}
IAM Policies for integrated services

```json
{
   "Effect": "Allow",
   "Action": [
      "events:PutTargets",
      "events:PutRule",
      "events:DescribeRule"
   ],
   "Resource": [
      "arn:aws:events:[$region]::$account Id]:rule/StepFunctionsGetEventForCodeBuildStartBuildBatchRule"
   ]
}
```

Request Response

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "codebuild:StartBuildBatch"
         ],
         "Resource": [
            "arn:aws:codebuild:[$region]::$account Id]:project/*"
         ]
      }
   ]
}
```

StopBuildBatch

**Static resources**

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "codebuild:StopBuildBatch"
         ],
         "Resource": [
            "arn:aws:codebuild:[$region]::$account Id]:project/[$project Name]"
         ]
      }
   ]
}
```

**Dynamic resources**

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "codebuild:StopBuildBatch"
         ],
         "Resource": [
            "arn:aws:codebuild:[$region]::$account Id]:project/[$project Name]"
         ]
      }
   ]
}
```
IAM Policies for integrated services

```json

{  
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:RetryBuildBatch",
        "codebuild:StopBuildBatch",
        "codebuild:BatchGetBuildBatches"
      ],
      "Resource": [
        "arn:aws:codebuild:[region]:[accountId]:project/*"
      ]
    }
  ]
}

RetryBuildBatch

Static resources

Run a Job (sync)

```json

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:RetryBuildBatch"
      ],
      "Resource": [
        "arn:aws:codebuild:[region]:[accountId]:project/[[projectName]]"
      ]
    }
  ]
}

Request Response

```json

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:RetryBuildBatch"
      ],
      "Resource": [
        "arn:aws:codebuild:[region]:[accountId]:project/[[projectName]]"
      ]
    }
  ]
}

Dynamic resources

Run a Job (sync)

```json

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:RetryBuildBatch"
      ],
      "Resource": [
        "arn:aws:codebuild:[region]:[accountId]:project/[[projectName]]"
      ]
    }
  ]
}
IAM Policies for integrated services

"codebuild:StopBuildBatch",
"codebuild:BatchGetBuildBatches"
],
"Resource": [
  "arn:aws:codebuild:[region]:[accountId]:project/*"
]
}

Request Response

{  
  "Version": "2012-10-17",
  "Statement": [
    
      {  
          "Effect": "Allow",
          "Action": [
            "codebuild:RetryBuildBatch"
          ],
          "Resource": [
            "arn:aws:codebuild:[region]:[accountId]:project/*"
          ]
      }  
  ]
}

DeleteBuildBatch

Static resources

{  
  "Version": "2012-10-17",
  "Statement": [
    
      {  
          "Effect": "Allow",
          "Action": [
            "codebuild:DeleteBuildBatch"
          ],
          "Resource": [
            "arn:aws:codebuild:[region]:[accountId]:project/[projectName]"
          ]
      }  
  ]
}

Dynamic resources

{  
  "Version": "2012-10-17",
  "Statement": [
    
      {  
          "Effect": "Allow",
          "Action": [
            "codebuild:DeleteBuildBatch"
          ],
          "Resource": [
            "arn:aws:codebuild:[region]:[accountId]:project/*"
          ]
      }  
  ]
}
X-Ray

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

To enable X-Ray tracing, you will need an IAM policy with suitable permissions to allow tracing. If your state machine uses other integrated services, you may need additional IAM policies. See the IAM policies for your specific service integrations.

When you create a state machine with X-Ray tracing enabled, an IAM policy is automatically created.

**Note**
If you enable X-Ray tracing for an existing state machine you must ensure that you add a policy with sufficient permissions to enable X-Ray traces.

```json
{
   "Version": "2012-10-17",
   "Statement": [
     {
       "Effect": "Allow",
       "Action": [
         "xray:PutTraceSegments",
         "xray:PutTelemetryRecords",
         "xray:GetSamplingRules",
         "xray:GetSamplingTargets"
       ],
       "Resource": [
         "*
       ]
     }
   ]
}
```

For more information about using X-Ray with Step Functions, see AWS X-Ray and Step Functions (p. 512).

Amazon Athena

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

**StartQueryExecution**

*Static resources*

Run a Job (.sync)

```json
{}
```
"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "athena:startQueryExecution",
      "athena:stopQueryExecution",
      "athena:getQueryExecution",
      "athena:getDataCatalog"
    ],
    "Resource": [
      "arn:aws:athena:{{region}}:{{accountId}}:workgroup/[[workGroup]]",
      "arn:aws:athena:{{region}}:{{accountId}}:datacatalog/*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "s3:GetBucketLocation",
      "s3:GetObject",
      "s3:ListBucket",
      "s3:ListBucketMultipartUploads",
      "s3:ListMultipartUploadParts",
      "s3:AbortMultipartUpload",
      "s3:CreateBucket",
      "s3:PutObject"
    ],
    "Resource": [
      "arn:aws:s3:::*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [
      "glue:CreateDatabase",
      "glue:GetDatabase",
      "glue:GetDatabases",
      "glue:UpdateDatabase",
      "glue:DeleteDatabase",
      "glue:CreateTable",
      "glue:UpdateTable",
      "glue:GetTable",
      "glue:GetTables",
      "glue:DeleteTable",
      "glue:BatchDeleteTable",
      "glue:BatchCreatePartition",
      "glue:CreatePartition",
      "glue:UpdatePartition",
      "glue:GetPartition",
      "glue:GetPartitions",
      "glue:BatchGetPartition",
      "glue:DeletePartition",
      "glue:BatchDeletePartition"
    ],
    "Resource": [
      "arn:aws:glue:{{region}}:{{accountId}}:catalog",
      "arn:aws:glue:{{region}}:{{accountId}}:database/*",
      "arn:aws:glue:{{region}}:{{accountId}}:table/**",
      "arn:aws:glue:{{region}}:{{accountId}}:userDefinedFunction/*"
    ]
  }
],

{ "Effect": "Allow",
  "Action": ["lakeformation:GetDataAccess"]}
"Resource": [  
  "*"
]
}
]  
}

Request Response

{
  "Version": "2012-10-17",
  "Statement": [
  {
    "Effect": "Allow",
    "Action": [  
      "athena:startQueryExecution",
      "athena:getDataCatalog"
    ],
    "Resource": [  
      "arn:aws:athena:{{region}}:{{accountId}}:workgroup/[[workGroup]]",
      "arn:aws:athena:{{region}}:{{accountId}}:datacatalog/*"  
    ]
  },
  {
    "Effect": "Allow",
    "Action": [  
      "s3:GetBucketLocation",
      "s3:GetObject",
      "s3:ListBucket",
      "s3:ListBucketMultipartUploads",
      "s3:ListMultipartUploadParts",
      "s3:AbortMultipartUpload",
      "s3:CreateBucket",
      "s3:PutObject"
    ],
    "Resource": [  
      "arn:aws:s3:::*"
    ]
  },
  {
    "Effect": "Allow",
    "Action": [  
      "glue:CreateDatabase",
      "glue:GetDatabase",
      "glue:GetDatabases",
      "glue:UpdateDatabase",
      "glue:DeleteDatabase",
      "glue:CreateTable",
      "glue:UpdateTable",
      "glue:GetTable",
      "glue:GetTables",
      "glue:DeleteTable",
      "glue:BatchDeleteTable",
      "glue:BatchCreatePartition",
      "glue:CreatePartition",
      "glue:UpdatePartition",
      "glue:GetPartition",
      "glue:GetPartitions",
      "glue:BatchGetPartition",
      "glue:DeletePartition",
      "glue:BatchDeletePartition"
    ],
    "Resource": [  
      "arn:aws:glue:{{region}}:{{accountId}}:catalog",
      "arn:aws:glue:{{region}}:{{accountId}}:database/*",
      "arn:aws:glue:{{region}}:{{accountId}}:table/*",
      "arn:aws:glue:{{region}}:{{accountId}}:partition/*",
      "arn:aws:glue:{{region}}:{{accountId}}:partition-set/*",
      "arn:aws:glue:{{region}}:{{accountId}}:database-tag:*",
      "arn:aws:glue:{{region}}:{{accountId}}:table-tag:*",
      "arn:aws:glue:{{region}}:{{accountId}}:partition-tag:*",
      "arn:aws:glue:{{region}}:{{accountId}}:partition-set-tag:*",
      "arn:aws:glue:{{region}}:{{accountId}}:column-tag:*",
      "arn:aws:glue:{{region}}:{{accountId}}:table-type:*",
      "arn:aws:glue:{{region}}:{{accountId}}:catalog-tag:*",
      "arn:aws:glue:{{region}}:{{accountId}}:table-instance-tag:*",
      "arn:aws:glue:{{region}}:{{accountId}}:table-instance:*",
      "arn:aws:glue:{{region}}:{{accountId}}:target-queue:*"
    ]
  }
]
"arn:aws:glue:{{region}}:{{accountId}}:table/**",
"arn:aws:glue:{{region}}:{{accountId}}:userDefinedFunction/**"
],
{
"Effect": "Allow",
"Action": [  
"lakeformation:GetDataAccess"
],
"Resource": [  
"**"  
]
}
}],
{
"Effect": "Allow",
"Action": [  
"athena:startQueryExecution",
"athena:stopQueryExecution",
"athena:getQueryExecution",
"athena:getDataCatalog"
],
"Resource": [  
"arn:aws:athena:{{region}}:{{accountId}}:workgroup/**",
"arn:aws:athena:{{region}}:{{accountId}}:datacatalog/**"
]
},
{
"Effect": "Allow",
"Action": [  
"s3:GetBucketLocation",
"s3:GetObject",
"s3:ListBucket",
"s3:ListBucketMultipartUploads",
"s3:ListMultipartUploadParts",
"s3:AbortMultipartUpload",
"s3:CreateBucket",
"s3:PutObject"
],
"Resource": [  
"arn:aws:s3:::*"
]
},
{
"Effect": "Allow",
"Action": [  
"glue:CreateDatabase",
"glue:GetDatabase",
"glue:GetDatabases",
"glue:UpdateDatabase",
"glue:DeleteDatabase",
"glue:CreateTable",
"glue:CreateTable",
"glue:UpdateTable",
"glue:GetTable",
"glue:GetCatalog",
"glue:GetTableVersion",
"glue:GetTableVersion`
]}

**Dynamic resources**

**Run a Job (sync)**

```json
{
    "Version": "2012-10-17",
    "Statement": [  
        {
            "Effect": "Allow",
            "Action": [  
                "athena:startQueryExecution",
                "athena:stopQueryExecution",
                "athena:getQueryExecution",
                "athena:getDataCatalog"
            ],
            "Resource": [  
                "arn:aws:athena:{{region}}:{{accountId}}:workgroup/**",
                "arn:aws:athena:{{region}}:{{accountId}}:datacatalog/**"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [  
                "s3:GetBucketLocation",
                "s3:GetObject",
                "s3:ListBucket",
                "s3:ListBucketMultipartUploads",
                "s3:ListMultipartUploadParts",
                "s3:AbortMultipartUpload",
                "s3:CreateBucket",
                "s3:PutObject"
            ],
            "Resource": [  
                "arn:aws:s3:::*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [  
                "glue:CreateDatabase",
                "glue:GetDatabase",
                "glue:GetDatabases",
                "glue:UpdateDatabase",
                "glue:DeleteDatabase",
                "glue:CreateTable",
                "glue:CreateTable",
                "glue:UpdateTable",
                "glue:GetTable",
                "glue:GetCatalog",
                "glue:GetTableVersion",
                "glue:GetTableVersion`
```
IAM Policies for integrated services

"glue:GetTables",
"glue:DeleteTable",
"glue:BatchDeleteTable",
"glue:BatchCreatePartition",
"glue:CreatePartition",
"glue:UpdatePartition",
"glue:GetPartition",
"glue:GetPartitions",
"glue:BatchGetPartition",
"glue:DeletePartition",
"glue:BatchDeletePartition"
]
},
"Resource": [
"arn:aws:glue:{{region}}:{{accountId}}:catalog",
"arn:aws:glue:{{region}}:{{accountId}}:database/**",
"arn:aws:glue:{{region}}:{{accountId}}:table/**",
"arn:aws:glue:{{region}}:{{accountId}}:userDefinedFunction/**"
],
"Effect": "Allow",
"Action": [
"lakeformation:GetDataAccess"
],
"Resource": [
"*"
]
}
)

Request Response

{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": [
"athena:startQueryExecution",
"athena:getDataCatalog"
],
"Resource": [
"arn:aws:athena:{{region}}:{{accountId}}:workgroup/**",
"arn:aws:athena:{{region}}:{{accountId}}:datatag/catalog/*"
]
},
{
"Effect": "Allow",
"Action": [
"s3:GetBucketLocation",
"s3:GetObject",
"s3:ListBucket",
"s3:ListBucketMultipartUploads",
"s3:ListMultipartUploadParts",
"s3:AbortMultipartUpload",
"s3:CreateBucket",
"s3:PutObject"
],
"Resource": [
"arn:aws:s3:::*
]
},
{
"Effect": "Allow",
"Action": 
]
"Action": [
    "glue:CreateDatabase",
    "glue:GetDatabase",
    "glue:GetDatabases",
    "glue:UpdateDatabase",
    "glue:DeleteDatabase",
    "glue:CreateTable",
    "glue:UpdateTable",
    "glue:GetTable",
    "glue:GetTables",
    "glue:DeleteTable",
    "glue:BatchDeleteTable",
    "glue:BatchCreatePartition",
    "glue:CreatePartition",
    "glue:UpdatePartition",
    "glue:GetPartition",
    "glue:GetPartitions",
    "glue:BatchGetPartition",
    "glue:DeletePartition",
    "glue:BatchDeletePartition"
],
"Resource": [
    "arn:aws:glue:{{region}}:{{accountId}}:catalog",
    "arn:aws:glue:{{region}}:{{accountId}}:database/*",
    "arn:aws:glue:{{region}}:{{accountId}}:table/*",
    "arn:aws:glue:{{region}}:{{accountId}}:userDefinedFunction/*"
]
},
{
    "Effect": "Allow",
    "Action": [
        "lakeformation:GetDataAccess"
    ],
    "Resource": [
        "*"
    ]
}

StopQueryExecution

Resources

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:StopQueryExecution"
            ],
            "Resource": [
                "arn:aws:athena:{{region}}:{{accountId}}:workgroup/*"
            ]
        }
    ]
}

GetQueryExecution

Resources
IAM Policies for integrated services

GetQueryResults

Resources

Amazon EKS

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

CreateCluster

Resources

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "athena:getQueryExecution"
            ],
            "Resource": [
                "arn:aws:athena:{{region}}:{accountId}:workgroup/**"
            ]
        }
    ]
}
```
CreateNodeGroup

Resources

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["ec2:DescribeSubnets", "eks:CreateNodegroup"],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": ["eks:DescribeNodegroup", "eks:DeleteNodegroup"],
    },
    {
      "Effect": "Allow",
      "Action": ["iam:GetRole", "iam:ListAttachedRolePolicies"],
      "Resource": "arn:aws:iam::444455556666:role/**"
    }
  ]
}
```
IAM Policies for integrated services

```
"Effect": "Allow",
"Action": "iam:PassRole",
"Resource": [
  "arn:aws:iam::44445556666:role/StepFunctionsSample-EKSClusterManager-NodeInstanceRole-ANPAJ2UCR6DPCEXAMPLE"
],
"Condition": {
  "StringEquals": {
    "iam:PassedToService": "eks.amazonaws.com"
  }
}
```

DeleteCluster

```
Resources

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["eks:DeleteCluster", "eks:DescribeCluster"],
    }
  ]
}
```

DeleteNodegroup

```
Resources

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["eks:DeleteNodegroup", "eks:DescribeNodegroup"],
    }
  ]
}
```

For more information about using Amazon EKS with Step Functions, see Call Amazon EKS with Step Functions (p. 334).
Amazon API Gateway

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

Resources:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["execute-api:Invoke"],
      "Resource": ["arn:[:region]::[:account-id]:*"
    }
  ]
}
```

The following code example shows a resource policy for calling API Gateway.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "states.amazonaws.com"
      },
      "Action": "execute-api:Invoke",
      "Resource": "arn:aws:execute-api::<region>::<account-id>/<api-id>/<stage-name>/<HTTP-VERB>/<resource-path-specifier>",
      "Condition": {
        "StringEquals": {
          "aws:SourceArn": [<SourceStateMachineArn>]
        }
      }
    }
  ]
}
```

AWS Glue DataBrew

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)
Run a Job (sync)

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "databrew:startJobRun",
            "databrew:listJobRuns",
            "databrew:stopJobRun"
         ],
         "Resource": [
            "arn:aws:databrew:{{region}}:{{accountId}}:job/*"
         ]
      }
   ]
}
```

Request Response

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "databrew:startJobRun"
         ],
         "Resource": [
            "arn:aws:databrew:{{region}}:{{accountId}}:job/*"
         ]
      }
   ]
}
```

Amazon EventBridge

These example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see:

- IAM Policies for integrated services (p. 531)
- Service Integration Patterns (p. 291)

PutEvents

Static resources

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": [
            "events:PutEvents"
         ],
         "Resource": [
            "arn:aws:events:us-east-1:123456789012:event-bus/stepfunctions-sampleproject-eventbus"
         ]
      }
   ]
}
```
IAM Policies for integrated services

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["events:PutEvents"],
            "Resource": "arn:aws:events:::event-bus/*"
        }
    ]
}
```

For more information about using EventBridge with Step Functions, see Call EventBridge with Step Functions (p. 349).

AWS Step Functions

For a state machine that calls StartExecution for a single nested workflow execution, use an IAM policy that limits permissions to that state machine.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["states:StartExecution"],
            "Resource": ["arn:aws:states:[region]:[accountId]:stateMachine:[stateMachineName]"
        }
    ]
}
```

For more information, see the following:

- Working with other services (p. 271)
- Pass Parameters to a Service API (p. 296)
- AWS Step Functions (p. 351)

Synchronous

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```
"states:StartExecution"
],
"Resource": [
  "arn:aws:states:[region]:[accountId]:stateMachine:
  [[stateMachineName]]"
],
"Effect": "Allow",
"Action": [
  "states:DescribeExecution",
  "states:StopExecution"
],
"Resource": "*"
},
{
  "Effect": "Allow",
  "Action": [
    "events:PutTargets",
    "events:PutRule",
    "events:DescribeRule"
],
  "Resource": [
    "arn:aws:events:[region]:[accountId]:rule/StepFunctionsGetEventsForStepFunctionsExecutionRule"
  ]
}
}
}

Asynchronous

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": [
        "arn:aws:states:[region]:[accountId]:stateMachine:[stateMachineName]"
      ]
    }
  ]
}

For more information about nested workflow executions, see Start Workflow Executions from a Task State (p. 73).

Activities or No Tasks

For a state machine that has only Activity tasks, or no tasks at all, use an IAM policy that denies access to all actions and resources.

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "states:StartExecution",
        "states:DescribeExecution",
        "states:StopExecution",
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:states:[region]:[accountId]:stateMachine:[stateMachineName]"
      ]
    }
  ]
}
Tag-based Policies

Step Functions supports policies based on tags. For example, you could restrict access to all Step Functions resources that include a tag with the key `environment` and the value `production`.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "states:TagResource",
        "states:UntagResource",
        "states:DeleteActivity",
        "states:DeleteStateMachine",
        "states:StopExecution"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/environment": "production"
        }
      }
    }
  ]
}
```

This policy will Deny the ability to delete state machines or activities, stop executions, and add or delete new tags for all resources that have been tagged as `environment/production`.

For more information about tagging, see the following:

- Tagging in Step Functions (p. 97)
- Controlling Access Using IAM Tags

Logging and Monitoring

For information about logging and monitoring in AWS Step Functions, see the Logging and monitoring (p. 491) section.

Compliance Validation for AWS Step Functions

Third-party auditors assess the security and compliance of AWS Step Functions as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others.

For a list of AWS services in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. For general information, see AWS Compliance Programs.
You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using Step Functions is determined by the sensitivity of your data, your company’s compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- **Security and Compliance Quick Start Guides** – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- **Architecting for HIPAA Security and Compliance Whitepaper** – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.
- **AWS Compliance Resources** – This collection of workbooks and guides might apply to your industry and location.
- **Evaluating Resources with Rules** in the AWS Config Developer Guide – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- **AWS Security Hub** – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

### Resilience in AWS Step Functions

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, Step Functions offers several features to help support your data resiliency and backup needs.

### Infrastructure Security in AWS Step Functions

As a managed service, AWS Step Functions is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access Step Functions through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

You can call these API operations from any network location, but Step Functions does support resource-based access policies, which can include restrictions based on the source IP address. You can also use Step Functions policies to control access from specific Amazon Virtual Private Cloud (Amazon VPC) endpoints or specific VPCs. Effectively, this isolates network access to a given Step Functions resource from only the specific VPC within the AWS network.
Configuration and Vulnerability Analysis in AWS Step Functions

Configuration and IT controls are a shared responsibility between AWS and you, our customer. For more information, see the AWS shared responsibility model.
Troubleshooting

If you encounter difficulties when working with Step Functions, use the following troubleshooting resources.

Topics
- General troubleshooting (p. 580)
- Troubleshooting service integrations (p. 581)
- Troubleshooting activities (p. 582)
- Troubleshooting Express Workflows (p. 583)

General troubleshooting

I'm unable to create a state machine.

The IAM role associated with the state machine might not have sufficient permissions (p. ). Check the IAM role's permissions, including for AWS service integration tasks, X-Ray, and CloudWatch logging. Additional permissions are required for .sync task states.

I'm unable to use a JsonPath to reference the previous task's output.

For a JsonPath, a JSON key must end with $. This means a JsonPath can only be used in a key-value pair. If you want to use a JsonPath other places, such as an array, you can use intrinsic functions (p. ). For example, you could use something similar to the following:

Task A output:

```json
{
    "sample": "test"
}
```

Task B:

```json
{
    "JsonPathSample.$": ".sample"
}
```

Tip
Use the data flow simulator in the Step Functions console to test JSON path syntax, to better understand how data is manipulated within a state, and to see how data is passed between states.

There was a delay in state transitions.

For standard workflows, there is a limit on the number of state transitions. When you exceed the state transition limit, Step Functions delays state transitions until the bucket for the quota is filled. State transition limit throttling can be monitored by reviewing the StateTransition metric in the Dimension for Step Functions Service Metrics (p. 495) section of the CloudWatch Metrics page.
When I start new Standard Workflow executions, they fail with the ExecutionLimitExceeded error.

Step Functions has a limit of 1,000,000 open executions for each AWS account. If you exceed this limit, Step Functions throws an ExecutionLimitExceeded error. This limit does not apply to Express Workflows. You can use the following CloudWatch Metrics math to approximate the number of open executions: ExecutionsStarted - (ExecutionsSucceeded + ExecutionsTimedOut + ExecutionsFailed + ExecutionsAborted).

A failure on one branch in a parallel state causes the whole execution to fail.

This is an expected behavior. To avoid encountering failures when using a parallel state, configure Step Functions to catch errors (p. ) thrown from each branch.

Troubleshooting service integrations

My job is complete in the downstream service, but in Step Functions the task state remains "In progress" or its completion is delayed.

For .sync service integration patterns, Step Functions uses EventBridge rules, downstream APIs, or a combination of both to detect the downstream job status. For some services, Step Functions does not create EventBridge rules to monitor. For example, for the AWS Glue service integration, instead of using EventBridge rules, Step Functions makes a glue:GetJobRun call. Because of the frequency of API calls, there is a difference between the downstream task completion and the Step Functions task completion time. Step Functions requires IAM permissions to manage the EventBridge rules and to make calls to the downstream service. For more details about how insufficient permissions on your execution role can affect the completion of tasks, see Additional permissions for tasks using the Run a Job pattern (p. 532).

I want to return a JSON output from a nested state machine execution.

There are two Step Functions synchronous service integrations for Step Functions: startExecution.sync and startExecution.sync:2. Both wait for the nested state machine to complete, but they return different Output formats. You can use startExecution.sync:2 to return a JSON output under Output.

I can't invoke a Lambda function from another account.

In the Task state’s Resource field, use arn:aws:states:::lambda:invoke and pass the FunctionArn in parameters. The IAM role that is associated with the state machine must have the right permissions to invoke cross-account Lambda functions: lambda:invokeFunction.
I'm unable to see task tokens passed from `.waitForTaskToken` states.

In the Task state's `Parameters` field, you must pass a task token. For example, you could use something similar to the following code.

```json
{
    "StartAt":"taskToken",
    "States":{
        "taskToken":{
            "Type":"Task",
            "Resource":"arn:aws:states:::lambda:invoke.waitForTaskToken",
            "Parameters":{
                "FunctionName":"get-model-review-decision",
                "Payload":{
                    "token.$":"$.Task.Token"
                }
            },
            "End":true
        }
    }
}
```

**Note**

You can try to use `.waitForTaskToken` with any API action. However, some APIs don't have any suitable parameters.

**Troubleshooting activities**

**My state machine execution is stuck at an activity state.**

An activity task state doesn't start until you poll a task token by using the `GetActivityTask` API action. As a best practice, add a task level timeout in order to avoid a stuck execution. For more information, see [Use timeouts to avoid stuck executions](p. 267).

**My activity worker times out while waiting for a task token.**

Workers use the `GetActivityTask` API action to retrieve a task with the specified activity ARN that is scheduled for execution by a running state machine. `GetActivityTask` starts a long poll, so the service
holds the HTTP connection open and responds as soon as a task becomes available. The maximum time
the service hold the request before responding is 60 seconds. If no task is available within 60 seconds,
the poll returns a taskToken with a null string. To avoid this timeout, configure a client side socket with
a timeout of at least 65 seconds in the AWS SDK or in the client you are using to make the API call.

Troubleshooting Express Workflows

My application times out before receiving a response from a StartSyncExecution API call.

Configure a client side socket timeout in the AWS SDK or client you are using to make the API call.
To receive a response, the timeout must have a value higher than the duration of Express Workflow
executions.

I'm unable to see the execution history in order to troubleshoot Express Workflow failures.

Express Workflows do not have an execution history. Instead, you need to enable CloudWatch logging.
Then you can use CloudWatch Logs Insights queries to better understand your Express Workflow
executions.

To list executions based on duration:

```plaintext
fields ispresent(execution_arn) as exec_arn
| filter exec_arn
| filter type in ["ExecutionStarted", "ExecutionSucceeded", "ExecutionFailed",
"ExecutionAborted", "ExecutionTimedOut"]
| stats latest(type) as status,
tomillis(earliest(event_timestamp)) as UTC_starttime,
tomillis(latest(event_timestamp)) as UTC_endtime,
latest(event_timestamp) - earliest(event_timestamp) as duration_in_ms
| sort duration desc
```

To list failed and cancelled executions:

```plaintext
fields ispresent(execution_arn) as isRes | filter type in ["ExecutionFailed",
"ExecutionAborted", "ExecutionTimedOut"]
```
Related information

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 webpage 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 Support Information</td>
<td>The primary webpage for information about AWS 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*.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date changed</th>
</tr>
</thead>
</table>
| Update | **Step Functions console enhancements**  
The console now features a redesigned *Execution Details* page that includes the following enhancements:  
- Ability to identify the reason for a failed execution at a glance.  
- Two new modes of visualizations for your state machine – *Table view* and *Event view*. These views also provide you the ability to apply filters to only view the information of interest. In addition, you can sort the *Event view* contents based on the event timestamps.  
- Switch between the different iterations of *Map* state in the *Graph view* mode using a dropdown list or in the *Table view* mode's tree view for *Map* states.  
- View in-depth information about each state in the workflow, including the complete input and output data transfer path and retry attempts for *Task* or *Parallel* states.  
- Miscellaneous enhancements including the option to copy the state machine's execution Amazon Resource Name, view the count of total state machine transitions, and export the execution details in JSON format. | May 09, 2022 |
| Documentation-only updates | Added a new topic to explain the various types of information displayed in the *Execution Details* page. Also, added a tutorial to show how to examine this information. For more information, see:  
- Viewing and debugging executions on the Step Functions console (p. 75)  
- Tutorial: Examining state machine executions using the Step Functions console (p. 83) | |
| Update | Step Functions now provides a workaround to prevent the confused deputy security issue, which arises when an entity (a service or an account) is coerced by a different entity to perform an action. For more information, see:  
- Prevent cross-service confused deputy issue (p. 526) | May 02, 2022 |
| Update | Step Functions has expanded support for AWS SDK integrations by adding 21 more AWS services. For more information, see: Supported AWS SDK service integrations (p. 275).  
Documentation-only updates:  
- Added a list of all the exception prefixes present in the exceptions that are generated when you erroneously perform an AWS SDK service integration with Step Functions. For more information, see: Supported AWS SDK service integrations (p. 275). | April 19, 2022 |
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>New feature</td>
<td>• Added a list of all the unsupported API actions for supported AWS SDK integrations. For more information, see: Unsupported API actions for supported services (p. 290).</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>• Added a list of all the supported AWS SDK integrations that are now deprecated. For more information, see: Deprecated AWS SDK service integrations (p. 291).</td>
<td>January 28, 2022</td>
</tr>
<tr>
<td>New feature</td>
<td>Step Functions Local now supports AWS SDK integration and mocking of service integrations. For more information, see:</td>
<td>December 10, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>• Using Mocked Service Integrations (p. 255)</td>
<td>November 15, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions now supports creating an Amazon API Gateway REST API with synchronous express state machine as backend integration using the AWS Cloud Development Kit (CDK). For more information, see:</td>
<td>October 14, 2021</td>
</tr>
<tr>
<td>Update</td>
<td>Step Functions has added three new sample projects that demonstrate the integration of Step Functions and Amazon Athena's upgraded console. For more information, see:</td>
<td>November 22, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>• Creating an API Gateway REST API with Synchronous Express State Machine Using the AWS CDK (p. 234)</td>
<td>September 30, 2021</td>
</tr>
<tr>
<td>Update</td>
<td>Step Functions has added three new sample projects that demonstrate the integration of Step Functions and Amazon Athena's upgraded console. For more information, see:</td>
<td>September 30, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>Step Functions has added Amazon VPC endpoints support for Synchronous Express Workflows. For more information, see:</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>• Amazon VPC Endpoints for Step Functions (p. 530)</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>Update</td>
<td>AWS Step Functions has added three new sample projects that demonstrate how to use the Step Functions AWS Batch integration. For more information, see:</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>• Fan out an AWS Batch job (p. 478)</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>• AWS Batch with Lambda (p. 481)</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>• Use Step Functions and AWS Batch with error handling (p. 475)</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions has added AWS SDK integrations, letting you use the API actions for all of the more than two hundred AWS services. For more information, see:</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>• AWS SDK service integrations (p. 274)</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>• Gather Amazon S3 bucket info using AWS SDK service integrations (p. 201)</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions has added a visual workflow designer, the AWS Step Functions Workflow Studio. For more information, see:</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>• AWS Step Functions Workflow Studio (p. 100)</td>
<td>June 17, 2021</td>
</tr>
<tr>
<td>New feature</td>
<td>• Learn to use the AWS Step Functions Workflow Studio (p. 126)</td>
<td>June 17, 2021</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>AWS Step Functions has added four new APIs, <code>StartBuildBatch</code>, <code>StopBuildBatch</code>, <code>RetryBuildBatch</code> and <code>DeleteBuildBatch</code>, to the CodeBuild integration. For more information, see:</td>
<td>June 4, 2021</td>
</tr>
<tr>
<td></td>
<td>• [Call AWS CodeBuild with Step Functions](p. 329)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions now integrates with Amazon EventBridge. For more information, see:</td>
<td>May 14, 2021</td>
</tr>
<tr>
<td></td>
<td>• [Call EventBridge with Step Functions](p. 349)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• [IAM policies for Step Functions and Amazon EventBridge](p. 574)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A sample project that shows how to [Send a custom event to EventBridge](p. 452)</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>AWS Step Functions has added a new sample project that shows how to use Step Functions and the Amazon Redshift Data API to run an ETL/ELT workflow. For more information, see:</td>
<td>April 16, 2021</td>
</tr>
<tr>
<td></td>
<td>• [Run ETL/ELT workflows using Amazon Redshift](Lambda, Amazon Redshift Data API) (p. 459)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions has a new data flow simulator in the console. For more information, see:</td>
<td>April 8, 2021</td>
</tr>
<tr>
<td></td>
<td>• [Step Functions console](p. 208)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions now integrates with Amazon EMR on EKS. For more information, see:</td>
<td>March 29, 2021</td>
</tr>
<tr>
<td></td>
<td>• [Call Amazon EMR on EKS with AWS Step Functions](p. 327)</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>YAML support for state machine definitions has been added to AWS Toolkit for Visual Studio Code and AWS CloudFormation. For more information, see:</td>
<td>March 4, 2021</td>
</tr>
<tr>
<td></td>
<td>• [Definition format support](p. 211)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• [AWS Toolkit for Visual Studio Code](p. 5)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions now integrates with AWS Glue DataBrew. For more information, see:</td>
<td>January 6, 2021</td>
</tr>
<tr>
<td></td>
<td>• [Manage AWS Glue DataBrew Jobs with Step Functions](p. 348)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• [What is AWS Glue DataBrew?](in the DataBrew developer guide.)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions Synchronous Express Workflows are now available, giving you an easy way to orchestrate microservices. For more information, see:</td>
<td>November 24, 2020</td>
</tr>
<tr>
<td></td>
<td>• [Synchronous and Asynchronous Express Workflows](p. 20)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A sample project that shows how to [Invoke Synchronous Express Workflows](p. 455)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The [StartSyncExecution](API documentation.</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>AWS Step Functions now integrates with Amazon API Gateway. For more information, see:</td>
<td>November 17, 2020</td>
</tr>
<tr>
<td></td>
<td>• Call API Gateway with Step Functions (p. 343)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• IAM policies for Step Functions and Amazon API Gateway (p. 573)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A sample project that shows how to Make a call to API Gateway (p. 444)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions now integrates with Amazon Elastic Kubernetes Service. For more information, see:</td>
<td>November 16, 2020</td>
</tr>
<tr>
<td></td>
<td>• Call Amazon EKS with Step Functions (p. 334)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• IAM policies for Step Functions and Amazon EKS (p. 570)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A sample project that shows how to Manage an Amazon EKS cluster (p. 438)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions now integrates with Amazon Athena. For more information, see:</td>
<td>October 22, 2020</td>
</tr>
<tr>
<td></td>
<td>• Call Athena with Step Functions (p. 332)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• IAM policies for Step Functions and Amazon Athena (p. 564)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A sample project that shows how to Start an Athena query (p. 419)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions now supports tracing end-to-end workflows with AWS X-Ray, giving you full visibility across state machine executions and making it easier to analyze and debug your distributed applications. For more information, see:</td>
<td>September 14, 2020</td>
</tr>
<tr>
<td></td>
<td>• AWS X-Ray and Step Functions (p. 512)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• IAM policies for Step Functions and X-Ray (p. 564)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• AWS Step Functions API Reference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TracingConfiguration</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>AWS Step Functions now supports payload sizes up to 262,144 bytes of data as a UTF-8 encoded string. This lets you process larger payloads in both Standard and Express workflows.</td>
<td>September 3, 2020</td>
</tr>
<tr>
<td></td>
<td>Your existing state machines do not need to be changed in order to use the larger payloads. However, you will need to update to the latest versions of the Step Functions SDK and Local Runner to use the updated APIs. For more information, see:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>Quotas</em> (p. 485)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• the section called “Use Amazon S3 ARNs instead of passing large payloads” (p. 267)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>States.DataLimitExceeded</em> (p. 89)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• the section called “CloudWatch Logs payloads” (p. 509)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• the section called “EventBridge payloads” (p. 500)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• AWS Step Functions API Reference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>CloudWatchEventsExecutionDataDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>HistoryEventExecutionDataDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>GetExecutionHistory</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>ActivityScheduledEventDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>ActivitySucceededEventDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>CloudWatchEventsExecutionDataDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>ExecutionSucceededEventDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>LambdaFunctionScheduledEventDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>ExecutionSucceededEventDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>StateEnteredEventDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>StateExitedEventDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>TaskSubmittedEventDetails</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>TaskSucceededEventDetails</em></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>The Amazon States Language has been updated as follows:</td>
<td>August 13, 2020</td>
</tr>
<tr>
<td></td>
<td>• <strong>Choice Rules (p. 42)</strong> has added</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A null comparison operator, IsNull. IsNull tests against the JSON null value, and can be used to detect if the output of a previous state is null or not.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Four other new operators have been added, IsBoolean, IsNumeric, IsString and IsTimestamp.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A test for the existence or non-existence of a field using the IsPresent operator. IsPresent can be used to prevent States.Runtime errors when there is an attempt to access a non-existent key.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Wildcard pattern matching to support string comparison against patterns with one or more wildcards.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Comparison between two variables for supported comparison operators.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Timeout and heartbeat values in a Task state can now be provided dynamically from the state input instead of a fixed value using the TimeoutSecondsPath and HeartbeatSecondsPath fields. See the Task (p. 29) state for more information.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The new ResultSelector (p. 60) field provides a way to manipulate a state's result before ResultPath is applied. The ResultSelector field is an optional field in the Map (p. 49), Parallel (p. 46), and Task (p. 29) states.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Intrinsic functions (p. 25) have been added to allow basic operations without Task states. Intrinsic functions can be used within the Parameters and ResultSelector fields.</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>AWS Step Functions now supports the Amazon SageMaker CreateProcessingJob API call. For more information, see:</td>
<td>August 4, 2020</td>
</tr>
<tr>
<td></td>
<td>• Manage SageMaker with Step Functions (p. 313)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Preprocess data and train a machine learning model (p. 409), a sample project that demonstrates CreateProcessingJob.</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions is now supported by AWS Serverless Application Model, making it easier to integrate workflow orchestration into your serverless applications. For more information, see:</td>
<td>May 27, 2020</td>
</tr>
<tr>
<td></td>
<td>• AWS Step Functions and AWS SAM (p. 215)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• AWS::Serverless::StateMachine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• AWS SAM Policy Templates</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions has introduced a new synchronous invocation for nesting Step Functions executions. The new invocation, arn:aws:states:::states:startExecution.sync:2, returns a JSON object. The original invocation, arn:aws:states:::states:startExecution.sync, continues to be supported, and returns a JSON-escaped string. For more information, see:</td>
<td>May 19, 2020</td>
</tr>
<tr>
<td></td>
<td>• Manage AWS Step Functions Executions as an Integrated Service (p. 351)</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>AWS Step Functions now integrates with AWS CodeBuild. For more information, see:</td>
<td>May 5, 2020</td>
</tr>
<tr>
<td></td>
<td>• Using AWS Step Functions with other services (p. 271)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Call AWS CodeBuild with Step Functions (p. 329)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Optimized integrations for Step Functions (p. 299)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>Step Functions is now supported in AWS Toolkit for Visual Studio Code, making it easier to create and visualize state machine based workflows without leaving your code editor.</td>
<td>March 31, 2020</td>
</tr>
<tr>
<td>Update</td>
<td>You can now configure logging to Amazon CloudWatch Logs for Standard workflows. For more information, see:</td>
<td>February 25, 2020</td>
</tr>
<tr>
<td></td>
<td>• Logging using CloudWatch Logs (p. 508)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>AWS Step Functions can now be accessed without requiring a public IP address, directly from Amazon Virtual Private Cloud (VPC). For more information, see:</td>
<td>December 23, 2019</td>
</tr>
<tr>
<td></td>
<td>• Amazon VPC Endpoints for Step Functions (p. 530)</td>
<td></td>
</tr>
</tbody>
</table>
### New feature

**Express Workflows** are a new workflow type, suitable for high-volume event processing workloads such as IoT data ingestion, streaming data processing and transformation, and mobile application backends.

For more information, review the following new and updated topics.

- Standard vs. Express Workflows (p. 19)
- Execution guarantees (p. 21)
- Getting started with AWS Step Functions (p. 10)
- Using AWS Step Functions with other services (p. 271)
- Optimized integrations for Step Functions (p. 299)
- Process High-Volume Messages from Amazon SQS (Express Workflows) (p. 395)
- Selective Checkpointing Example (Express Workflows) (p. 399)
- Quotas (p. 485)
- Quotas (p. 485)
- Logging using CloudWatch Logs (p. 508)
- AWS Step Functions API Reference
  - CreateStateMachine
  - UpdateStateMachine
  - DescribeStateMachine
  - DescribeStateMachineForExecution
  - StopExecution
  - DescribeExecution
  - GetExecutionHistory
  - ListExecutions
  - ListStateMachines
  - StartExecution
  - CloudWatchLogsLogGroup
  - LogDestination
  - LoggingConfiguration

**Date changed**: December 3, 2019

---

### New feature

AWS Step Functions now integrates with Amazon EMR. For more information, see:

- Using AWS Step Functions with other services (p. 271)
- Call Amazon EMR with Step Functions (p. 320)
- Optimized integrations for Step Functions (p. 299)

**Date changed**: November 19, 2019
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>AWS Step Functions has released the AWS Step Functions Data Science SDK. For more information, see the following.</td>
<td>November 7, 2019</td>
</tr>
<tr>
<td></td>
<td>• Project on Github</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SDK Documentation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The following Example Notebooks, which are available in the SageMaker console and the related GitHub project.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• hello_world_workflow.ipynb</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• machine_learning_workflow_abalone.ipynb</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• training_pipeline_pytorch_mnist.ipynb</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Step Functions now supports more API actions for Amazon SageMaker, and includes two new sample projects to demonstrate the functionality. For more information, see the following.</td>
<td>October 3, 2019</td>
</tr>
<tr>
<td></td>
<td>• Manage SageMaker with Step Functions (p. 313)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Using AWS Step Functions with other services (p. 271)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Train a Machine Learning Model (p. 384)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Tune a Machine Learning Model (p. 388)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>Step Functions includes a new Map state type. You can use a Map state to run a series of steps for each item in a JSON array in the input. For more information, see the following.</td>
<td>September 18, 2019</td>
</tr>
<tr>
<td></td>
<td>• Map State (p. 49)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Map State Example (p. 50)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Map State Tutorial (p. 144)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Map State Input and Output Processing (p. 52)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ItemsPath (p. 61)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Context Object Data for Map States (p. 71)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Map State Sample Project (p. 379)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>Step Functions supports starting new workflow executions by calling StartExecution as an integrated service API. See:</td>
<td>August 12, 2019</td>
</tr>
<tr>
<td></td>
<td>• Start Workflow Executions from a Task State (p. 73)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Manage AWS Step Functions Executions as an Integrated Service (p. 351)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Using AWS Step Functions with other services (p. 271)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• IAM Policies for Starting Step Functions Workflow Executions (p. 575)</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>
| New feature   | Step Functions includes the ability to pass a task token to integrated services, and pause the execution until that task token is returned with `SendTaskSuccess` or `SendTaskFailure`. See:  
  - Service Integration Patterns (p. 291)  
  - Wait for a Callback with the Task Token (p. 293)  
  - Callback Pattern Example (Amazon SQS, Amazon SNS, Lambda) (p. 369)  
  - Optimized integrations for Step Functions (p. 299)  
  - Deploying an Example Human Approval Project (p. 185)  
  - Service Integration Metrics (p. 494)  
  Step Functions now provides a way to access dynamic information about your current execution directly in the "Parameters" field of a state definition. See:  
  - Context Object (p. 70)  
  - Pass Context Object Nodes as Parameters (p. 298) | May 23, 2019 |
| New feature   | Step Functions supports CloudWatch Events for execution status changes, see:  
  - EventBridge (CloudWatch Events) for Step Functions execution status changes (p. 499)  
  - Amazon CloudWatch Events User Guide | May 8, 2019  |
| New feature   | Step Functions supports IAM permissions using tags. For more information, see:  
  - Tagging in Step Functions (p. 97)  
  - Tag-based Policies (p. 577) | March 5, 2019 |
<p>| New feature   | Step Functions Local is now available. You can run Step Functions on your local machine for testing and development. Step Functions Local is available for download as either a Java application, or as a Docker image. See Testing Step Functions State Machines Locally (p. 247). | February 4, 2019 |
| New feature   | AWS Step Functions is now available in the Beijing and Ningxia regions. See Supported regions (p. 7). | January 15, 2018 |
| New feature   | Step Functions supports resource tagging to help track your cost allocation. You can tag state machines on the Details page, or through API actions. See Tagging in Step Functions (p. 97). | January 7, 2019 |
| New feature   | AWS Step Functions is now available in the Europe (Paris), and South America (São Paulo) regions. See Supported regions (p. 7). | December 13, 2018 |
| New feature   | AWS Step Functions is now available the Europe (Stockholm) region. See Supported regions (p. 7) for a list of supported regions. | December 12, 2018 |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>New feature</td>
<td>You can now easily configure and generate a state definition for integrated services when editing your state definition. For more information, see:</td>
<td>December 10, 2018</td>
</tr>
</tbody>
</table>
|             |   • Code Snippets (p. 298)  
|             |     • Using Code Snippets (p. 181)                                                                                                                                                                            |                  |
| New feature | Step Functions now integrates with some AWS services. You can now directly call and pass parameters to the API of these integrated services from a task state in the Amazon States Language. For more information, see:                      | November 29, 2018|
|             |   • Using AWS Step Functions with other services (p. 271)  
|             |     • Pass Parameters to a Service API (p. 296)  
|             |     • Optimized integrations for Step Functions (p. 299)                                                                                                                                                      |                  |
| Update      | Improved the description of TimeoutSeconds and HeartbeatSeconds in the documentation for task states. See Task (p. 29).                                                                                      | October 24, 2018  |
| Update      | Improved the description for the Maximum execution history size limit and provided a link to the related best practices topic.                                                                             | October 17, 2018  |
|             |   • Quotas related to state machine executions (p. 487)  
<p>|             |     • Avoid reaching the history quota (p. 268)                                                                                                                                                             |                  |
| Update      | Added a new tutorial to the AWS Step Functions documentation: See Starting a State Machine Execution in Response to Amazon S3 Events (p. 149).                                                             | September 25, 2018|
| Update      | Removed the entry Maximum executions displayed in Step Functions console from the limits documentation. See Quotas (p. 485).                                                                                   | September 13, 2018|
| Update      | Added a best practices topic to the AWS Step Functions documentation on improving latency when polling for activity tasks. See Avoid latency when polling for activity tasks (p. 269).                               | August 30, 2018   |
| Update      | Improved the AWS Step Functions topic on activities and activity workers. See Activities (p. 33).                                                                                                              | August 29, 2018   |
| Update      | Improved the AWS Step Functions topic on CloudTrail integration. See Logging Step Functions Using AWS CloudTrail (p. 504).                                                                                     | August 7, 2018    |
| Update      | Added JSON examples to AWS CloudFormation tutorial. See Creating a Lambda state machine for Step Functions using AWS CloudFormation (p. 217).                                                             | June 23, 2018     |
| Update      | Added a new topic on handling Lambda service errors. See Handle Lambda service exceptions (p. 268).                                                                                                         | June 20, 2018     |
| New feature | AWS Step Functions is now available the Asia Pacific (Mumbai) region. See Supported regions (p. 7) for a list of supported regions.                                                                         | June 28, 2018     |
| New feature | AWS Step Functions is now available the AWS GovCloud (US-West) region. For information about using Step Functions in the AWS GovCloud (US-West) Region, see AWS GovCloud (US). | June 28, 2018     |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>Improved documentation on error handling for Parallel states. See Error Handling (p. 49).</td>
<td>June 20, 2018</td>
</tr>
</tbody>
</table>
| Update       | Improved documentation about Input and Output processing in Step Functions. Learn how to use InputPath, ResultPath, and OutputPath to control the flow of JSON through your workflows, states, and tasks. See:  
  - Input and Output Processing in Step Functions (p. 55)  
  - ResultPath (p. 62)                                                                                                                    | June 7, 2018   |
| Update       | Improved code examples for parallel states. See Parallel (p. 46).                                                                                                                                       | June 4, 2018   |
| New feature  | You can now monitor API and Service metrics in CloudWatch. See Monitoring Step Functions Using CloudWatch (p. 491).                                                                                      | May 25, 2018   |
| Update       | StartExecution, StopExecution, and StateTransition now have increased throttling limits in the following regions:                                                                                           | May 16, 2018   |
|              |   - US East (N. Virginia)  
|              |   - US West (Oregon)  
|              |   - Europe (Ireland)    
<p>|              | For more information see Quotas (p. 485).                                                                                                                                                    |                |
| New feature  | AWS Step Functions is now available the US West (N. California) and Asia Pacific (Seoul) regions. See Supported regions (p. 7) for a list of supported regions.                                               | May 5, 2018    |
| Update       | Updated procedures and images to match changes to the interface.                                                                                                                                         | April 25, 2018 |
| Update       | Added a new tutorial that shows how to start a new execution to continue your work. See Continuing as a New Execution (p. 172). This tutorial describes a design pattern that can help avoid some service limitations. See Avoid reaching the history quota (p. 268). | April 19, 2018 |
| Update       | Improved introduction to states documentation by adding conceptual information about state machines. See States (p. 21).                                                                                | March 9, 2018  |
| Update       | In addition to HTML, PDF, and Kindle, the AWS Step Functions Developer Guide is available on GitHub. To leave feedback, choose the GitHub icon in the upper right-hand corner.                             | March 2, 2018  |
| Update       | Added a topic describing other resources relating to Step Functions. See Related information (p. 584).                                                                                                  | February 20, 2018 |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date changed</th>
</tr>
</thead>
</table>
| New feature  | • When you create a new state machine, you must acknowledge that AWS Step Functions will create an IAM role which allows access to your Lambda functions.  
• Updated the following tutorials to reflect the minor changes in the state machine creation workflow:  
  • Getting started with AWS Step Functions (p. 10)  
  • Creating a Step Functions State Machine That Uses Lambda (p. 137)  
  • Creating an Activity State Machine Using Step Functions (p. 162)  
  • Handling Error Conditions Using a Step Functions State Machine (p. 140)  
  • Iterating a Loop Using Lambda (p. 167)                                                                                   | February 19, 2018      |
| Update       | Added a topic that describes an example activity worker written in Ruby. This implementation can be used to create a Ruby activity worker directly, or as a design pattern for creating an activity worker in another language.  
See Example Activity Worker in Ruby (p. 34).                                                                                 | February 6, 2018       |
| Update       | Added a new tutorial describing a design pattern that uses a Lambda function to iterate a count.                                                                                                          | January 31, 2018       |
|              | See Creating a Step Functions State Machine That Uses Lambda (p. 137).                                                                                                                                   |                        |
| Update       | Updated content on IAM permissions to include DescribeStateMachineForExecution and UpdateStateMachine APIs.                                                                                              | January 26, 2018       |
|              | See Creating Granular IAM Permissions for Non-Admin Users (p. 527).                                                                                                                                       |                        |
| Update       | Added newly available regions: Canada (Central), Asia Pacific (Singapore).                                                                                                                                  | January 25, 2018       |
|              | See Supported regions (p. 7).                                                                                                                                                                               |                        |
| Update       | Updated tutorials and procedures to reflect that IAM allows you to select Step Functions as a role.                                                                                                        | January 24, 2018       |
| Update       | Added a new Best Practices topic that suggests not passing large payloads between states.                                                                                                                  | January 23, 2018       |
|              | See Use Amazon S3 ARNs instead of passing large payloads (p. 267).                                                                                                                                       |                        |
| Update       | Corrected procedures to match updated interface for creating a state machine:  
  • Getting started with AWS Step Functions (p. 10)  
  • Creating a Step Functions State Machine That Uses Lambda (p. 137)  
  • Creating an Activity State Machine Using Step Functions (p. 162)  
  • Handling Error Conditions Using a Step Functions State Machine (p. 140)                                                                 | January 17, 2018       |
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Feature</td>
<td>You can use <em>Sample Projects</em> to quickly provision state machines and all related AWS resources. See <a href="#">Sample projects for Step Functions</a> (p. 354), Available sample projects include:</td>
<td>January 11, 2018</td>
</tr>
</tbody>
</table>
|                 | • Poll for Job Status (Lambda, AWS Batch) (p. 364)  
|                 | • Task Timer (Lambda, Amazon SNS) (p. 367)  
|                 | **Note**  
|                 | These sample projects and related documentation replace tutorials that described implementing the same functionality.                                                                                                                                                                         |              |
| Update          | Added a *Best Practices* section that includes information on avoiding stuck executions. See [Best practices for Step Functions](#) (p. 267).                                                                 | January 5, 2018 |
| Update          | Added a note on how retries can affect pricing:  
|                 | **Note**  
|                 | Retries are treated as state transitions. For information about how state transitions affect billing, see [Step Functions Pricing](#).                                                                                                                                               | December 8, 2017 |
| Update          | Added information related to resource names:  
|                 | **Note**  
|                 | Step Functions allows you to create state machine, execution, and activity names that contain non-ASCII characters. These non-ASCII names don't work with Amazon CloudWatch. To ensure that you can track CloudWatch metrics, choose a name that uses only ASCII characters. | December 6, 2017 |
| Update          | Improved security overview information and added a topic on granular IAM permissions. See [Security in AWS Step Functions](#) (p. 523) and [Creating Granular IAM Permissions for Non-Admin Users](#) (p. 527). | November 27, 2017 |
| New Feature     | You can update an existing state machine. See [Step 3: Update a state machine](#) (p. 12).                                                                                                                  | 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. 88)  
|                 | • Step 3: Create a State Machine with a Catch Field (p. 142)  
|                 | **Note**  
<p>|                 | Unhandled errors in Lambda are reported as <em>Lambda Unknown</em> in the error output. These include out-of-memory errors and function timeouts. You can match on <em>Lambda Unknown</em>, <em>States.ALL</em>, or <em>States.TaskFailed</em> to handle these errors. When Lambda hits the maximum number of invocations, the error is <em>Lambda.TooManyRequestsException</em>. For more information about Lambda Handled and Unhandled errors, see <a href="#">FunctionError</a> in the <a href="#">AWS Lambda Developer Guide</a>. | October 17, 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 IAM instructions and updated the screenshots in all tutorials (p. 137).</td>
<td>October 11, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>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:</td>
<td>October 6, 2017</td>
</tr>
<tr>
<td></td>
<td>• Creating a Step Functions State Machine That Uses Lambda (p. 137)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Job Status Poller</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Task Timer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handling Error Conditions Using a Step Functions State Machine (p. 140)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Corrected and clarified information about creating state machines in the following sections:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Getting started with AWS Step Functions (p. 10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating an Activity State Machine Using Step Functions (p. 162)</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Rewrote the IAM instructions in the following sections to reflect changes in the IAM console:</td>
<td>October 5, 2017</td>
</tr>
<tr>
<td></td>
<td>• How AWS Step Functions Works with IAM (p. 526)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Step Functions State Machine That Uses Lambda (p. 137)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Job Status Poller</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Task Timer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handling Error Conditions Using a Step Functions State Machine (p. 140)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Step Functions API Using API Gateway (p. 152)</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Rewrote the State Machine Data (p. 53) section.</td>
<td>September 28, 2017</td>
</tr>
<tr>
<td>New feature</td>
<td>The limits related to API action throttling (p. 487) are increased for all regions where Step Functions is available.</td>
<td>September 18, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Corrected and clarified information about starting new executions in all tutorials.</td>
<td>September 14, 2017</td>
</tr>
<tr>
<td></td>
<td>Corrected and clarified information in the Quotas related to accounts (p. 486) section.</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Rewrote the following tutorials to reflect changes in the Lambda console:</td>
<td>August 28, 2017</td>
</tr>
<tr>
<td></td>
<td>• Creating a Step Functions State Machine That Uses Lambda (p. 137)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handling Error Conditions Using a Step Functions State Machine (p. 140)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Job Status Poller</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>Step Functions is available in Europe (London).</td>
<td>August 23, 2017</td>
</tr>
<tr>
<td>New feature</td>
<td>The visual workflows of state machines let you zoom in, zoom out, and center the graph.</td>
<td>August 21, 2017</td>
</tr>
</tbody>
</table>
### Change

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date changed</th>
</tr>
</thead>
</table>
| 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.  
| Execution State | Input Type                     | **Open** | **Closed** |
|              | Identical                      | Success | ExecutionAlreadyExists |
|              | Different                      | ExecutionAlreadyExists | ExecutionAlreadyExists |
|              | For more information, see the `name` request parameter of the `StartExecution` API action in the *AWS Step Functions API Reference*. |             |             |
| Update       | Added information about an alternative way of passing the state machine ARN to the *Creating a Step Functions API Using API Gateway* (p. 152) tutorial. | August 17, 2017 |
| Update       | Added the new *Creating a Job Status Poller* tutorial.                      | August 10, 2017 |
| New feature  | • Step Functions emits the `ExecutionThrottled` CloudWatch metric. For more information, see *Monitoring Step Functions Using CloudWatch* (p. 491).  
• Added the Quotas related to state throttling (p. 486) section. | August 3, 2017 |
| Update       | Updated the instructions in the *Step 1: Create an IAM Role for API Gateway* (p. 153) section. | July 18, 2017 |
| Update       | Corrected and clarified information in the *Choice* (p. 40) section.       | June 23, 2017 |
| Update       | Added information about using resources under other AWS accounts to the following tutorials:  
• *Creating a Step Functions State Machine That Uses Lambda* (p. 137)  
• *Creating a Lambda state machine for Step Functions using AWS CloudFormation* (p. 217)  
• *Creating an Activity State Machine Using Step Functions* (p. 162)  
• *Handling Error Conditions Using a Step Functions State Machine* (p. 140) | June 22, 2017 |
| Update       | Corrected and clarified information in the following sections:  
• *Getting started with AWS Step Functions* (p. 10)  
• *Handling Error Conditions Using a Step Functions State Machine* (p. 140)  
• *States* (p. 21)  
• *Error handling in Step Functions* (p. 88) | June 21, 2017 |
<p>| Update       | Rewrote all tutorials to match the Step Functions console refresh.         | June 12, 2017 |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>New feature</td>
<td>Step Functions is available in Asia Pacific (Sydney).</td>
<td>June 8, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Restructured the Amazon States Language (p. 23) section.</td>
<td>June 7, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Corrected and clarified information in the Creating an Activity State Machine Using Step Functions (p. 162) section.</td>
<td>June 6, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Corrected the code examples in the Examples using Retry and using Catch (p. 93) section.</td>
<td>June 5, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Restructured this guide using AWS documentation standards.</td>
<td>May 31, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Corrected and clarified information in the Parallel (p. 46) section.</td>
<td>May 25, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Merged the Paths and Filters sections into the Input and Output Processing in Step Functions (p. 55) section.</td>
<td>May 24, 2017</td>
</tr>
<tr>
<td>Update</td>
<td>Corrected and clarified information in the Monitoring Step Functions Using CloudWatch (p. 491) 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 Using Step Functions (p. 162) 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 with AWS Step Functions (p. 10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Step Functions State Machine That Uses Lambda (p. 137)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating an Activity State Machine Using Step Functions (p. 162)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handling Error Conditions Using a Step Functions State Machine (p. 140)</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>Added information about Lambda templates to the Creating a Step Functions State Machine That Uses Lambda (p. 137) and Handling Error Conditions Using a Step Functions State Machine (p. 140) tutorials.</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 Quotas related to task executions (p. 488).</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></td>
<td>• Added the Periodically Start a State Machine Execution Using EventBridge (p. 148) 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></td>
<td>• Updated the Handling Error Conditions Using a Step Functions State Machine (p. 140) tutorial and the Error handling in Step Functions (p. 88) section.</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 is available in Europe (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 with AWS Step Functions (p. 10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating a Step Functions State Machine That Uses Lambda (p. 137)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Creating an Activity State Machine Using Step Functions (p. 162)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handling Error Conditions Using a Step Functions State Machine (p. 140)</td>
<td></td>
</tr>
<tr>
<td>New feature</td>
<td>• The <strong>State Machines</strong> page of the Step Functions console includes the <strong>Copy to New</strong> and <strong>Delete</strong> buttons.</td>
<td>February 23, 2017</td>
</tr>
<tr>
<td></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 <strong>Creating a Step Functions API Using API Gateway (p. 152)</strong> tutorial.</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 <strong>Creating a Lambda state machine for Step Functions using AWS CloudFormation (p. 217)</strong> 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>Update</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.