Developer Guide

AWS Step Functions

AWS Step Functions: Developer Guide

Table of Contents

What is Step Functions?

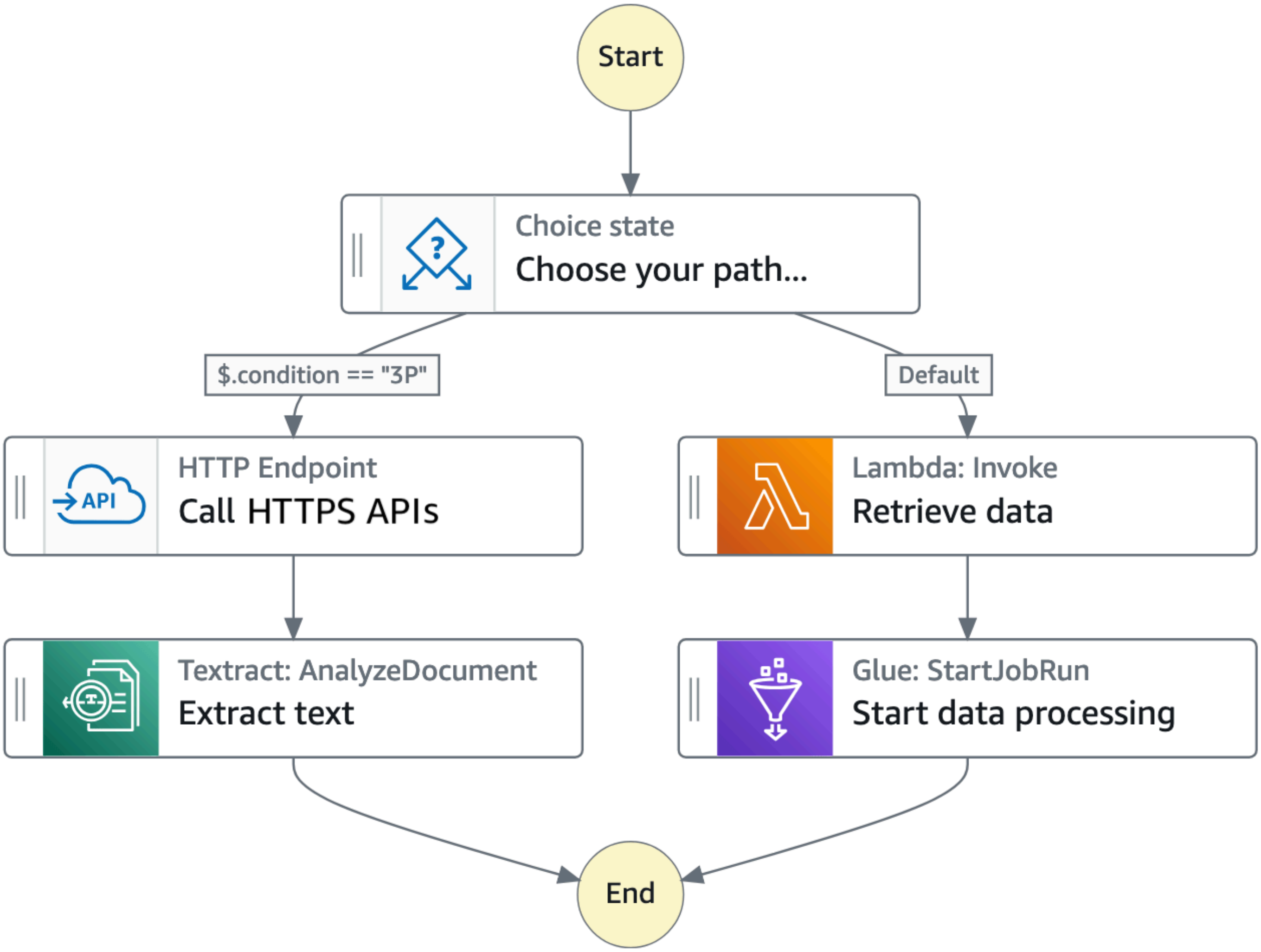
Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

With AWS Step Functions, you can create workflows, also called State machines, to build distributed applications, automate processes, orchestrate microservices, and create data and machine learning pipelines.

Step Functions is based on *state machines* and *tasks*. In Step Functions, state machines are called *workflows*, which are a series of event-driven steps. Each step in a workflow is called a *state*. For example, a Task state represents a unit of work that another AWS service performs, such as calling another AWS service or API. Instances of running workflows performing tasks are called *executions* in Step Functions.

The work in your state machine tasks can also be done using Activities which are workers that exist outside of Step Functions.



In the Step Functions' console, you can **visualize**, edit, and debug your application’s workflow. You can 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 have Step Functions control AWS services, such as AWS Glue, to create extract, transform, and load workflows. You also can create long-running, automated workflows for applications that require human interaction.

For a complete list of AWS Regions where Step Functions is available, see the AWS Region Table.

> **Learn how to use Step Functions**
>
> Start with the Getting started tutorial in this guide. For advanced topics and use cases, see the modules in The Step Functions Workshop.

Standard and Express workflows types

Step Functions has two workflow types:

- **Standard** workflows are ideal for long-running, auditable workflows, as they show execution history and visual debugging.

  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 are ideal for high-event-rate workloads, such as streaming data processing and IoT data ingestion.

  Express workflows have **at-least-once** workflow execution and can run for up to **five minutes**. This means that one or more steps in an Express Workflow can potentially run more than once, while each step in the workflow executes at least once.

Standard workflows	Express workflows
2,000 per second execution rate	100,000 per second execution rate
4,000 per second state transition rate	Nearly unlimited state transition rate
Priced by state transition	Priced by number and duration of executions
Show execution history and visual debugging	Show execution history and visual debugging based on **log level**
See execution history in Step Functions	Send execution history to CloudWatch

Standard workflows	Express workflows
Support integrations with all services. Support optimized integrati ons with some services.	Support integrations with all services.
Support *Request Response* pattern for all services Support *Run a Job* and/or *Wait for Callback* patterns in specific services (see following section for details)	Support *Request Response* pattern for all services

For more information on Step Functions pricing and choosing workflow type, see the following:

- AWS Step Functions pricing
- Choosing workflow type in Step Functions

Integrating with other services

Step Functions integrates with multiple AWS services. To call other AWS services, you can use two integration types:

- AWS SDK integrations provide a way to call any AWS service directly from your state machine, giving you access to thousands of API actions.
- Optimized integrations provide custom options for using those services in your state machines.

To combine Step Functions with other services, there are three **service integration patterns**:

- Request Response (default)

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

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

- Wait for a callback with a task token (.waitForTaskToken)

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

Standard Workflows and Express Workflows support the same **integrations** but not the same **integration patterns**.

- **Standard Workflows** support *Request Response* integrations. Certain services support *Run a Job (.sync)*, or *Wait for Callback (.waitForTaskToken)* , and both in some cases. See the following optimized integrations table for details.
- **Express Workflows** only support *Request Response* integrations.

To help decide between the two types, see Choosing workflow type in Step Functions.

AWS SDK integrations in Step Functions

Integrated service	Request Response	Run a Job - *.sync*	Wait for Callback - *.waitForTaskToken*
Over two hundred services	Standard & Express	*Not supported*	Standard

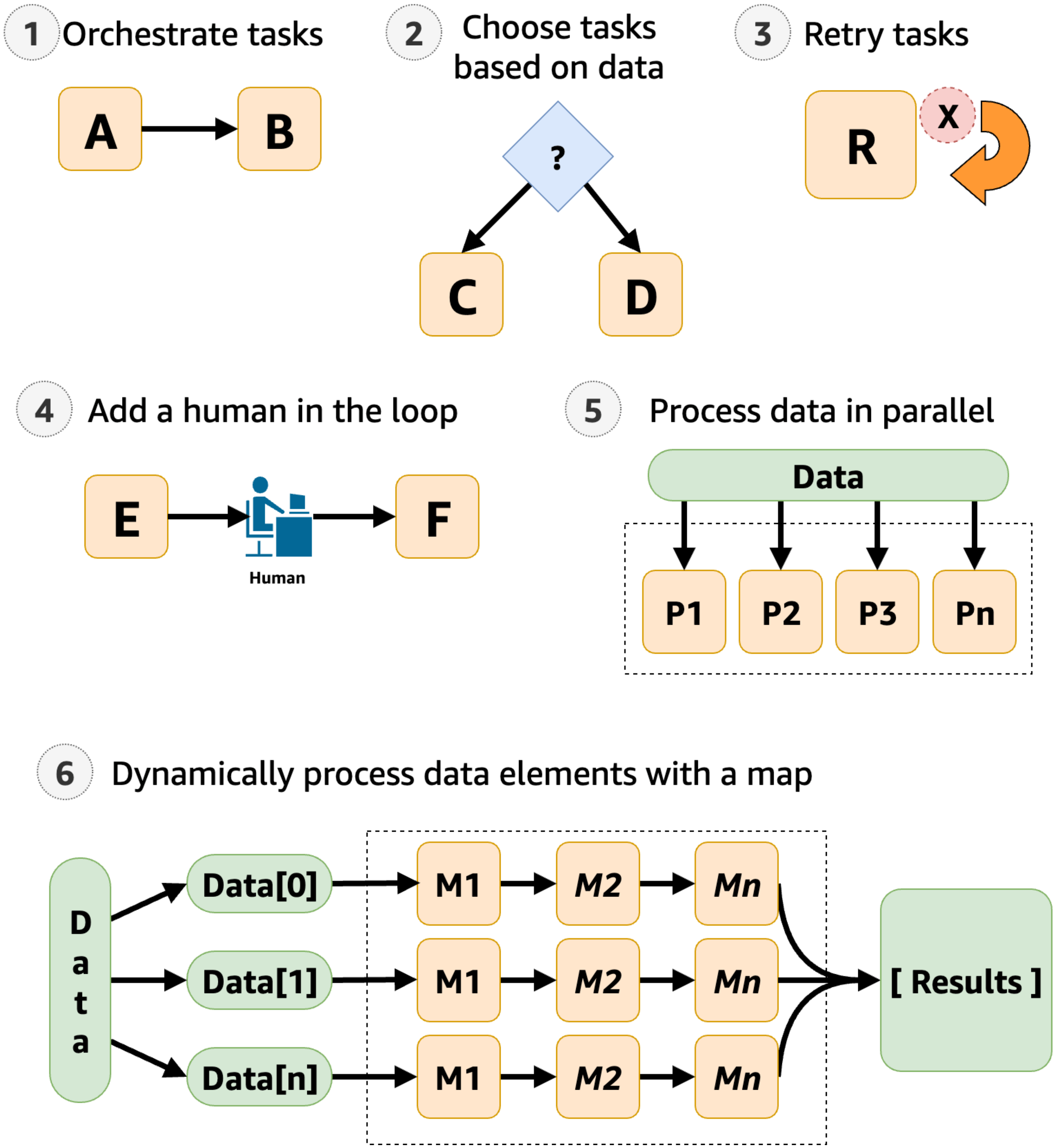
Optimized integrations in Step Functions

Integrated service	Request Response	Run a Job - *.sync*	Wait for Callback - *.waitForTaskToken*
Amazon API Gateway	Standard & Express	*Not supported*	Standard
Amazon Athena	Standard & Express	Standard	*Not supported*
AWS Batch	Standard & Express	Standard	*Not supported*
Amazon Bedrock	Standard & Express	Standard	Standard

Integrated service	**Request Response**	**Run a Job - *.sync***	**Wait for Callback - *.waitForTaskToken***
AWS CodeBuild	Standard & Express	Standard	*Not supported*
Amazon DynamoDB	Standard & Express	*Not supported*	*Not supported*
Amazon ECS/Fargate	Standard & Express	Standard	Standard
Amazon EKS	Standard & Express	Standard	Standard
Amazon EMR	Standard & Express	Standard	*Not supported*
Amazon EMR on EKS	Standard & Express	Standard	*Not supported*
Amazon EMR Serverless	Standard & Express	Standard	*Not supported*
Amazon EventBridge	Standard & Express	*Not supported*	Standard
AWS Glue	Standard & Express	Standard	*Not supported*
AWS Glue DataBrew	Standard & Express	Standard	*Not supported*
AWS Lambda	Standard & Express	*Not supported*	Standard
AWS Elemental MediaConvert	Standard & Express	Standard	*Not supported*
Amazon SageMaker AI	Standard & Express	Standard	*Not supported*
Amazon SNS	Standard & Express	*Not supported*	Standard
Amazon SQS	Standard & Express	*Not supported*	Standard
AWS Step Functions	Standard & Express	Standard	Standard

Example use cases for workflows

Step Functions manages your application's components and logic, so you can write less code and focus on building and updating your application quickly. The following image shows six use cases for Step Functions workflows.


1 Orchestrate tasks
A
B
2 Choose tasks based on data
?
C
D
3 Retry tasks
R
X
4 Add a human in the loop
E
Human
F
5 Process data in parallel
Data
P1
P2
P3
Pn
6 Dynamically process data elements with a map
Data
Data[0]
Data[1]
Data[n]
M1
M2
Mn
M1
M2
Mn
M1
M2
Mn
[Results]

1. **Orchestrate tasks** – You can create workflows that orchestrate a series of tasks, or *steps*, in a specific order. For example, *Task A* might be a Lambda function which provides inputs for another Lambda function in *Task B*. The last step in your workflow provides the final result.
2. **Choose tasks based on data** – Using a `Choice` state, you can have Step Functions make decisions based on the state's input. For example, imagine that a customer requests a credit limit increase. 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.
3. **Error handling** (`Retry` / `Catch`) – You can retry failed tasks, or catch failed tasks and automatically run alternative steps.

   For example, after a customer requests a username, perhaps the first call to your validation service fails, so your workflow may retry the request. When the second request is successful, the workflow can proceed.

   Or, perhaps the customer requested a username that is invalid or unavailable, a `Catch` statement could lead to a Step Functions workflow step that suggests an alternative username.

   For examples of `Retry` and `Catch`, see Handling errors in Step Functions workflows.
4. **Human in the loop** – Step Functions can include human approval steps in the workflow. For example, imagine a banking customer attempts to send funds to a friend. With a callback and a task token, you can have Step Functions wait until the customers friend confirms the transfer, and then Step Functions will continue the workflow to notify the banking customer that the transfer has completed.

   For an example, see Create a callback pattern example with Amazon SQS, Amazon SNS, and Lambda.
5. **Process data in parallel steps** – Using a `Parallel` state, Step Functions can process input data in parallel steps. For example, a customer might need to convert a video file into several display resolutions, so viewers can watch the video on multiple devices. Your workflow could send the original video file to several Lambda functions or use the optimized AWS Elemental MediaConvert integration to process a video into multiple display resolutions at the same time.
6. **Dynamically process data elements** – Using a `Map` state, Step Functions can run a set of workflow steps on each item in a dataset. The iterations run in parallel, which makes it possible to process a dataset quickly. For example, when your customer orders thirty items, your system needs to apply the same workflow to prepare each item for delivery. After all items have been

gathered and packaged for delivery, the next step might be to quickly send your customer a confirmation email with tracking information.

For an example **starter template**, see Process data with a Map.

Discover use cases for Step Functions workflows

With AWS Step Functions, you can build workflows that manage state over time, make decisions based on incoming data, and handle errors and exceptions.

Use case categories

- Data processing
- Machine learning
- Microservice orchestration
- IT and security automation

Data processing

As the volume of data grows from diverse sources, organizations need to process their data faster so they can quickly make 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.

With horizontal scaling and fault-tolerant workflows, Step Functions can operate millions of concurrent executions. You can process your data faster using parallel executions with `Parallel workflow state` state. Or, you can use the dynamic parallelism of the `Map workflow state` state to iterate over large data sets in a data stores, such as Amazon S3 buckets. Step Functions also provide the capability to retry failed executions, or choose a specific path to handle errors without managing complex error handling processes.

Step Functions directly integrates with other data processing services provided by AWS such as AWS Batch for batch processing, Amazon EMR for big data processing, AWS Glue for data preparation, Athena for data analysis, and AWS Lambda 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 provides a way for organizations to quickly analyze collected data to identify patterns and make decisions with minimal human intervention. Machine learning starts with an initial set of data, known as *training data*. Training data increases a machine learning model's prediction accuracy and acts as the foundation through which the model learns. After the trained model is considered accurate enough to meet business needs, you can deploy the model into production. The AWS Step Functions Data Science Project on Github is an open-source library that provides workflows to preprocess data, train, and then publish your models using Amazon SageMaker AI and Step Functions.

Preprocessing existing data sets is how an organization often creates training data. This preprocessing 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 an SageMaker AI notebook instance that runs in a Jupyter Notebook. After your data is ready, it can be uploaded to Amazon S3 for access. As machine learning models are trained, you can make adjustments to each model's parameters to improve accuracy.

Step Functions provides a way to orchestrate end-to-end machine learning workflows on SageMaker AI. These workflows can include data preprocessing, post-processing, feature engineering, data validation, and model evaluation. After 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

Step Functions gives you options to manage your microservice workflows.

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, so that independent teams can focus on a single business process, without needing to understand the entire application.

Microservices also provide individual components that suit your business needs, giving you flexibility without rewriting your entire workflow. Different teams can use the programming languages and frameworks of their choice to work with their microservice.

For long-running workflows you can use Standard Workflows with AWS Fargate integration to orchestrate applications running in containers. For short-duration, high-volume workflows that need an immediate response, Synchronous Express Workflows are ideal. One example are web-based or mobile applications, which 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 making the change immediately visible.
- Update an order during checkout, for example, adding, removing, or changing the quantity of an item; then immediately showing the updated cart to your 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

With Step Functions, you can create workflows that automatically scale and react to errors in your workflow. Your workflows can automatically retry failed tasks and use an exponential backoff to handle errors.

Error handling is essential in IT automation scenarios to manage complex and time-consuming operations, such as upgrading and patching software, deploying security updates to address vulnerabilities, selecting infrastructure, synchronizing data, and routing support tickets. By automating repetitive and time-consuming tasks, your organization can complete routine operations quickly and consistently at scale. Your focus can shift to strategic efforts such as feature development, complex support requests, and innovation while meeting your operational demands.

When human intervention is required for the workflow to proceed, for example approving a substantial credit increase, you can define branching logic in Step Functions, so that requests under a limit are automatically approved, and requests of the limit require human approval. When human approval is required, Step Functions can pause the workflow, wait for a human response, then continue the workflow after a response is received.

Some examples automation workflows include the following:

IT automation

- Auto-remediate incidents such as open SSH ports, low disk space, or public access granted to an Amazon S3 bucket.
- Automate the deployment of AWS CloudFormation StackSets.

Security automation

- Automate the response to a scenario where a user and user access key has been exposed.
- Auto-remediate security incident responses according to policy actions, such as restricting action to specific ARNs.
- Warn employees of phishing emails within seconds of receiving them.

Human Approval

- Automate machine learning model training, then get approval of the model by a data scientist before deploying the updated model.
- Automate customer feedback routing based on sentiment analysis so negative comments are quickly escalated for review.

Learn how to get started with Step Functions

With the Step Functions service, you can orchestrate complex application workflows. To get started, you'll use Workflow Studio to create and run a built-in **Hello World** workflow. You'll review the auto-generated Amazon States Language (ASL) definition in code. Finally, you'll drag-and-drop a service integration to do sentiment analysis.

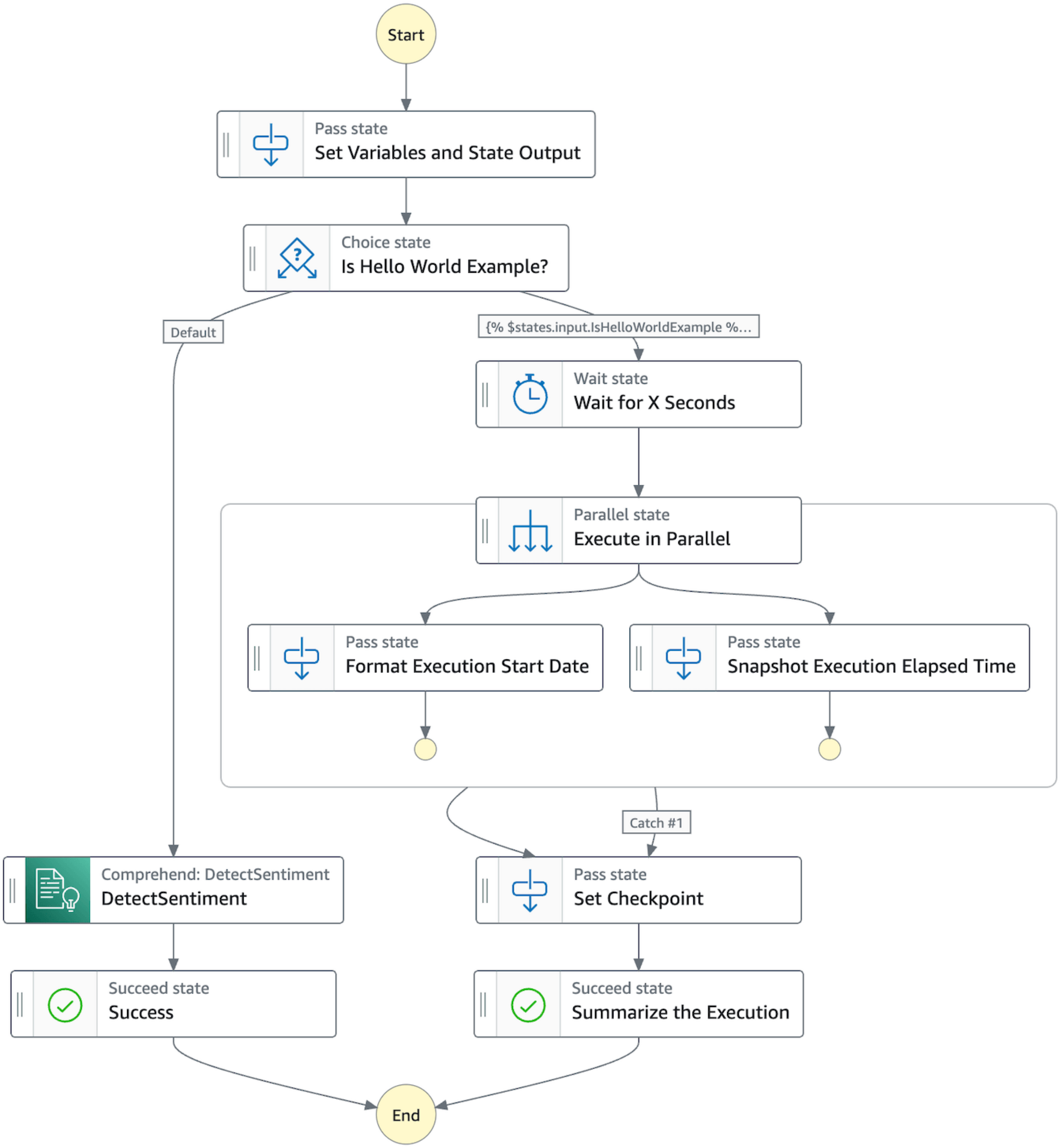
After you complete this tutorial, you'll know how to use Workflow Studio to create, configure, run, and update a workflow using both the **Design** and **Code** modes.

Estimated duration: **20-30 minutes**

What you will build

Your first state machine will start with *flow states*. Flow states are used to direct and control your workflow. After you learn how to run the workflow, you will add an **Action** to integrate the Amazon Comprehend service with a **Task state**.

The following diagram shows a visual of the complete state machine that you will build. When you first create the Hello World state machine, it will not need additional resources to run. The Step Functions console will create all the states and an IAM role in a single click. Later, when you add the service integration, you will need to create a role with a custom permission policy.

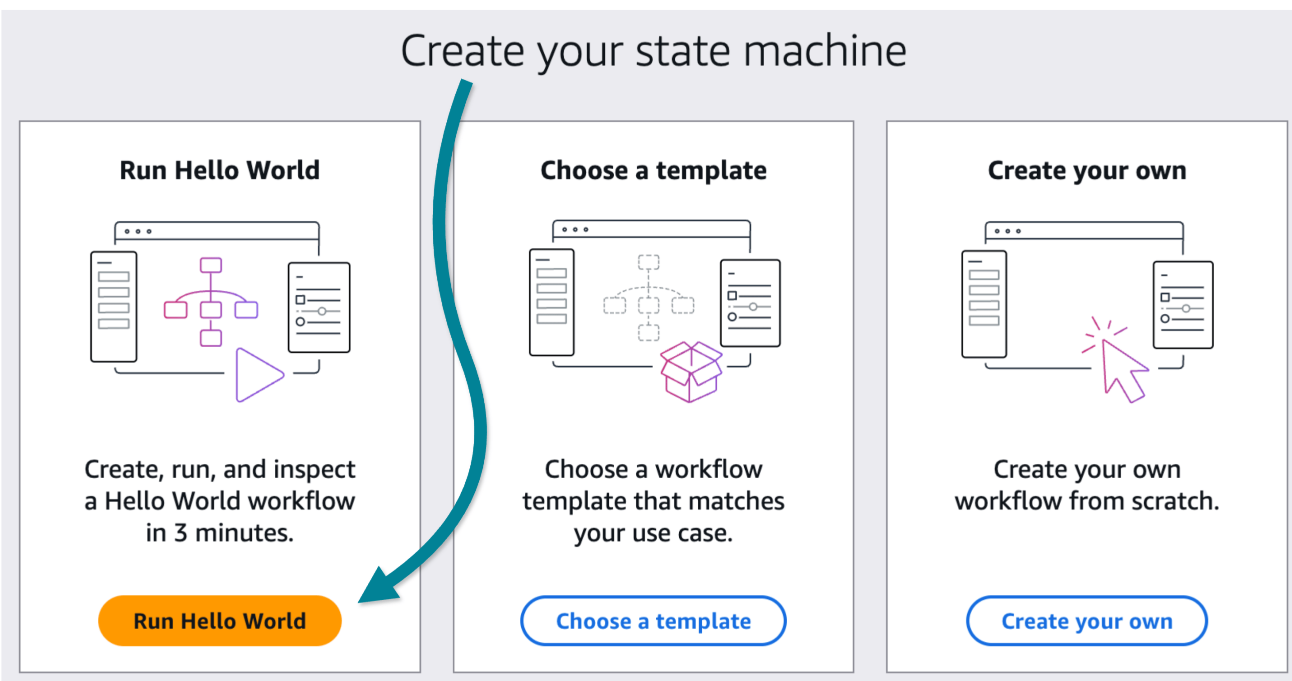

Start
Pass state
Set Variables and State Output
Choice state
Is Hello World Example?
Default
{% $states.input.IsHelloWorldExample %...
Wait state
Wait for X Seconds
Parallel state
Execute in Parallel
Pass state
Format Execution Start Date
Pass state
Snapshot Execution Elapsed Time
Catch #1
Comprehend: DetectSentiment
DetectSentiment
Pass state
Set Checkpoint
Succeed state
Success
Succeed state
Summarize the Execution
End

Step 1 - Create your state machine

In Step Functions, *workflows* are called **state machines**. We'll use both terms interchangeably. Your workflows will contain *states* that either **take action** or **control the flow** of your state machines.

1. Go to the **Step Functions console.**
2. In the Step Functions console, choose "**Step Functions**" from the upper left navigation, or the breadcrumbs, then choose **Get started**:

3. From the options, choose **Run Hello World**:

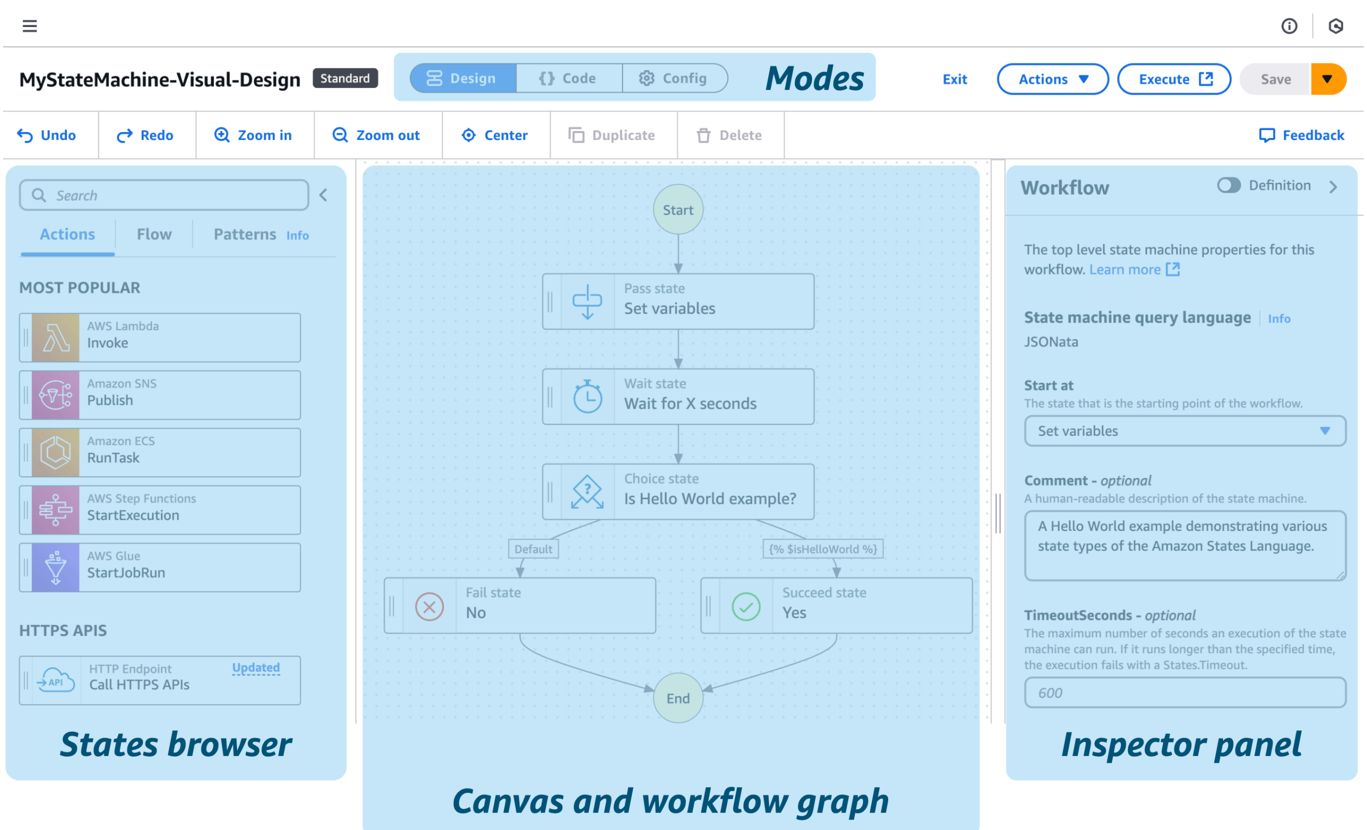


> **Tip**
>
> We recommend stepping through the short in-console walk through to become familiar with the UI.

Overview of Workflow Studio

With Workflow Studio for Step Functions, you can visually drag-and-drop states onto a canvas to build workflows.

You can add and edit states, configure steps, transform results, and set up error handling. The following screenshot shows four important areas of the interface that you will use to build your state machines.



Modes - Workflow Studio provides three modes of operation and defaults to the visual design mode.

- **Design** - a visual editing mode, where you can drag-and-drop states into your workflow.
- **Code** - a mode that focuses on the Amazon States Language code, also known as ASL code. You can edit ASL code directly and see changes reflected in the visual design.
- **Config** - configuration options including the name and type of the state machine (Standard or Express), assigned role when the workflow runs, logging, tracing, versioning, encryption, and tags.

States browser contains the following three tabs:

- **Actions** - a list of AWS APIs that you can drag-and-drop into your workflow. Each action represents a Task workflow state.
- **Flow** - flow states to control the order of steps in your workflow.
- **Patterns** - ready-to-use, reusable building blocks, such as iteratively processing data in an Amazon S3 bucket.

Canvas and workflow graph is where you drag-and-drop states on to your workflow graph, change the order of states, and select states to configure and test.

Inspector panel is where you view and edit the properties of any state selected on the canvas. You can turn on the *Definition* toggle to show the code for the currently selected state.

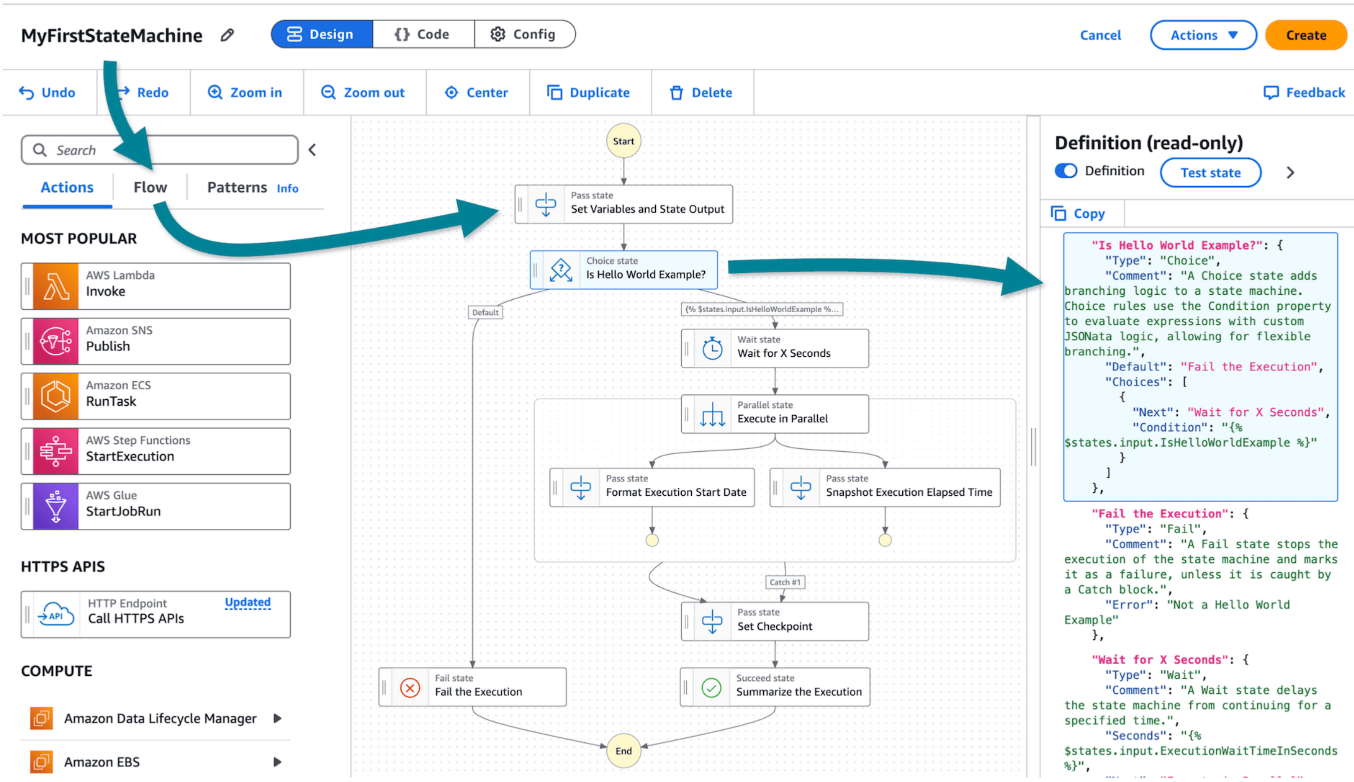
Overview of the state machine

The Hello World workflow starts with a **Pass state** which *passes* its input to its output, without performing work. Pass states can be used to generate static JSON output or transform JSON input before passing the data to the next state. Pass states are useful when constructing and debugging state machines.

The next state, a **Choice state**, uses the data in `IsHelloWorldExample` to choose the next branch of the workflow. If the first rule matches, the workflow pauses in a **Wait state**, then runs two tasks in a **Parallel state**, before moving on to a checkpoint and the successful end of the workflow. When there is no match, the workflow defaults to the **Fail state** before stopping the state machine.

Wait states can be useful when you want to delay before performing more work. Perhaps your workflow will wait 30 seconds after an order entry, so your customer has time to notice and fix an incorrect shipping address.

Parallel states can run multiple processes on your data. Perhaps the workflow will print an order ticket, update inventory, and increase a daily sales report simultaneously.



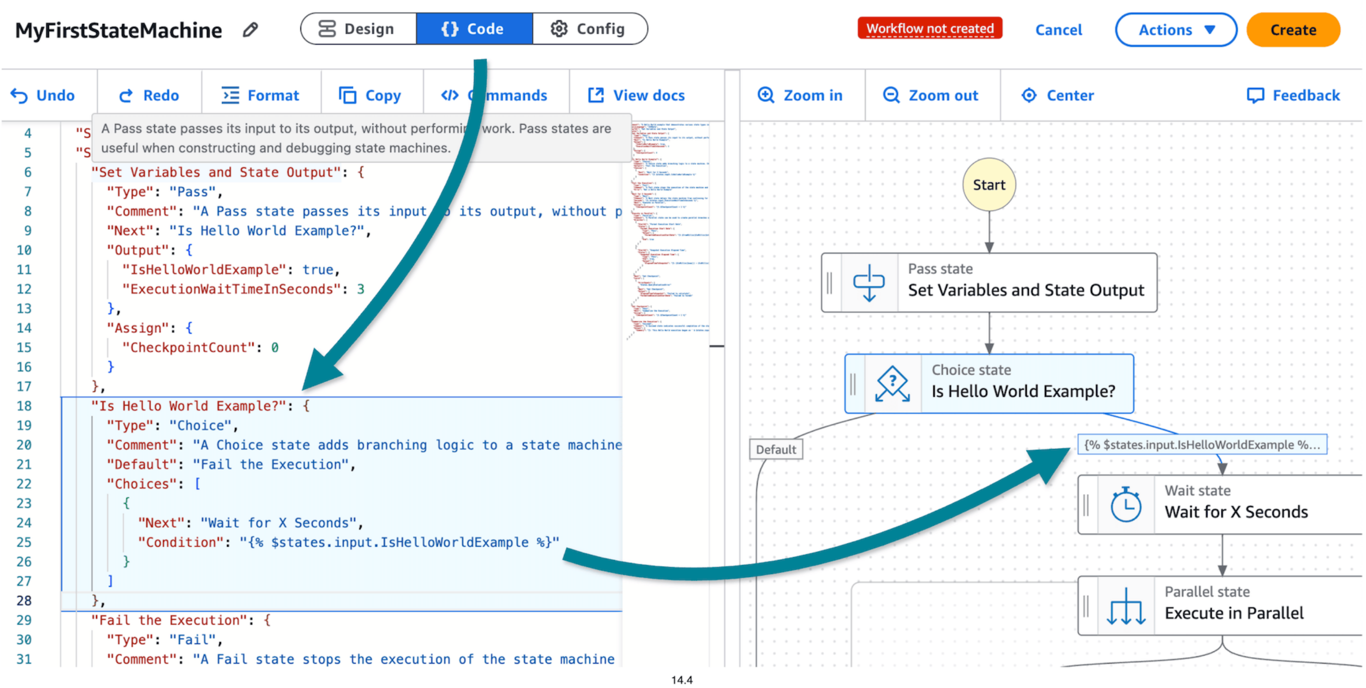
View the workflow code (ASL)

Your first state machine is in fact quite detailed, so explore further by reviewing the code.

State machines are defined using [Amazon States Language (ASL)](), an open source specification that describes a JSON-based language to describe state machines declaratively.

To view the entire state machine definition

1. Choose the **{ } Code** button to view the ASL code.
2. View the code on the left and compare with the state machine graph on the right.
3. Select some states on the canvas to review. For example, pick the **Choice state**.



Did you notice how the state's definition is highlighted in the code view?

To view code in the Inspector

1. Switch back to **Design** mode.
2. Expand the **Inspector panel** on the right.
3. Select the **Choice state** from the workflow graph on the Canvas.
4. In the **Inspector** panel, choose the **Definition** toggle.

Try choosing other states. See how the ASL code for each state you select is scrolled into view and highlighted?

(Actually) Create the state machine

> **Warning: name your state machine now!**
>
> You **cannot rename** a state machine after you create it. Choose a name **before** you save your state machine.

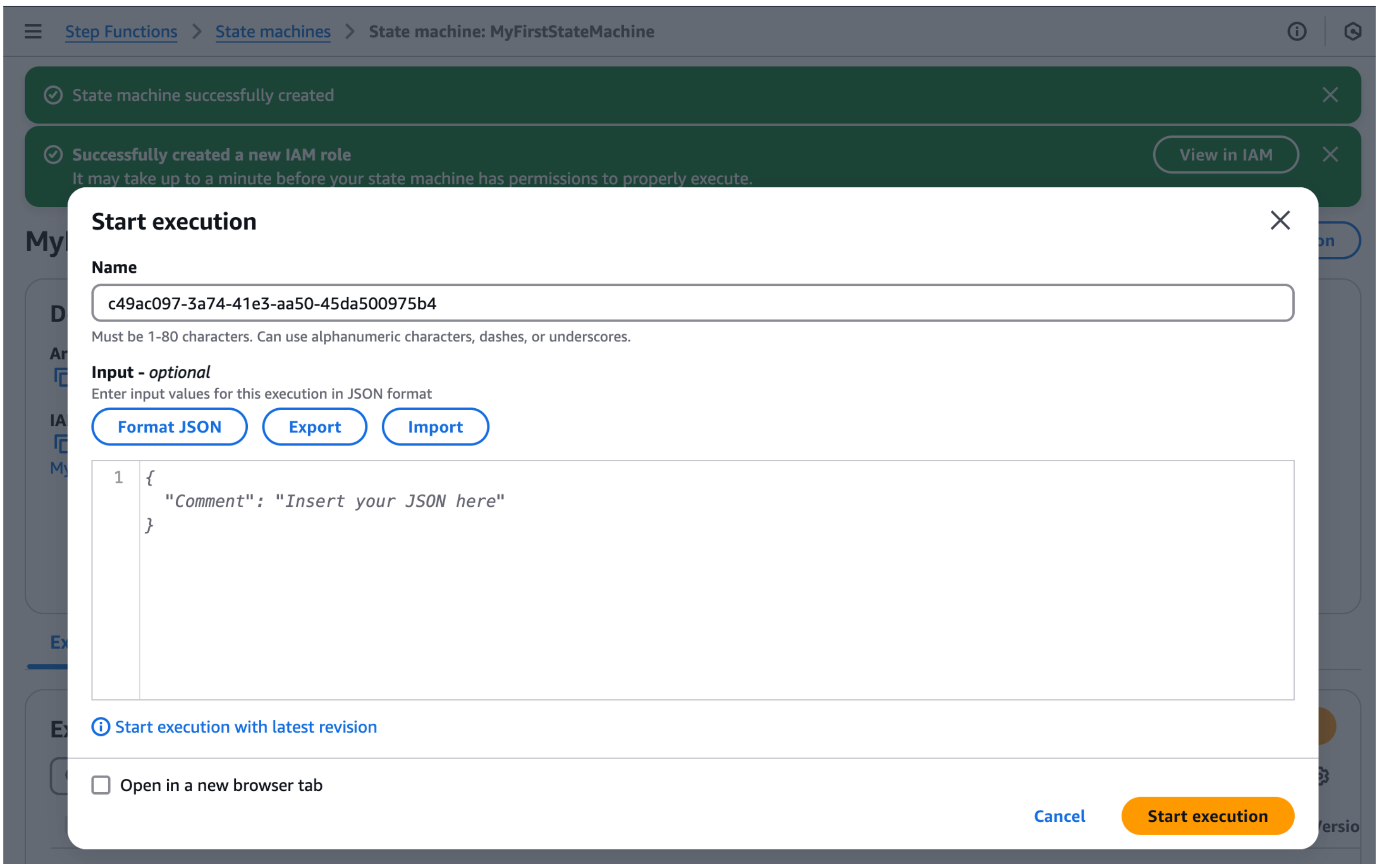
Until now, you've been working on a draft of your state machine. No resources have been created yet.

To rename and create your state machine

1. Choose **Config mode**.
2. For state machine name, enter `MyFirstStateMachine`
3. For permissions, accept the default to *Create a new role*.
4. Choose the **Create** button to **actually** create your state machine.

You should see notifications that your state machine and a new IAM role have been created.

You will be automatically presented with the option to start the state machine. You'll do that in the next step!



> **Workflow creation achieved!**
>
> Step Functions created your workflow and IAM role. Now, you are ready to start your state machine.

Step 2 - Start your state machine

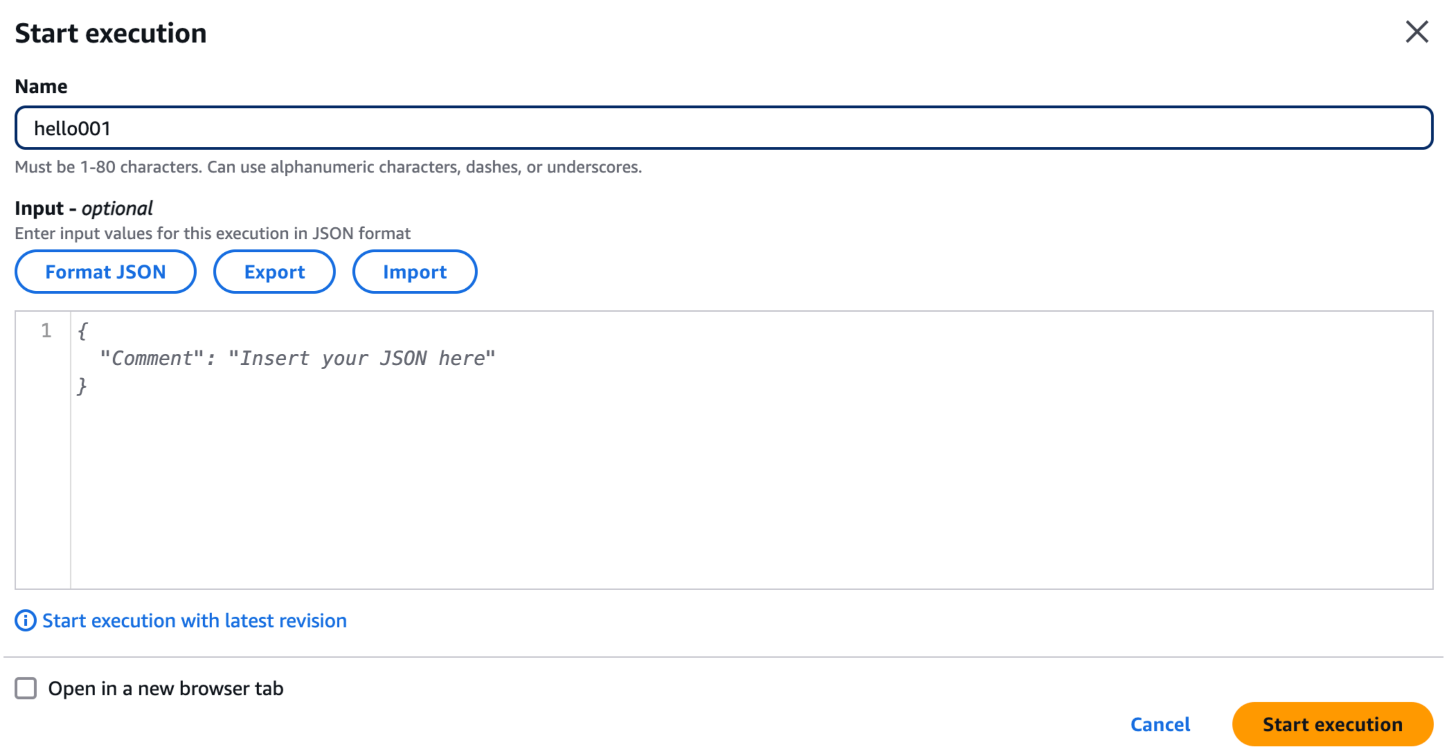
After your state machine has been created, you can start your workflow running.

Workflows optionally take **Input** that can be used in the state, sent to integrated services, and passed to the next state.

The **Hello World** state machine is self-contained and does not need input.

To start the state machine

1. Enter `hello001` for the name of the execution.
2. Leave the input field *empty*.
3. Choose the **Start execution** button.



Review the execution details

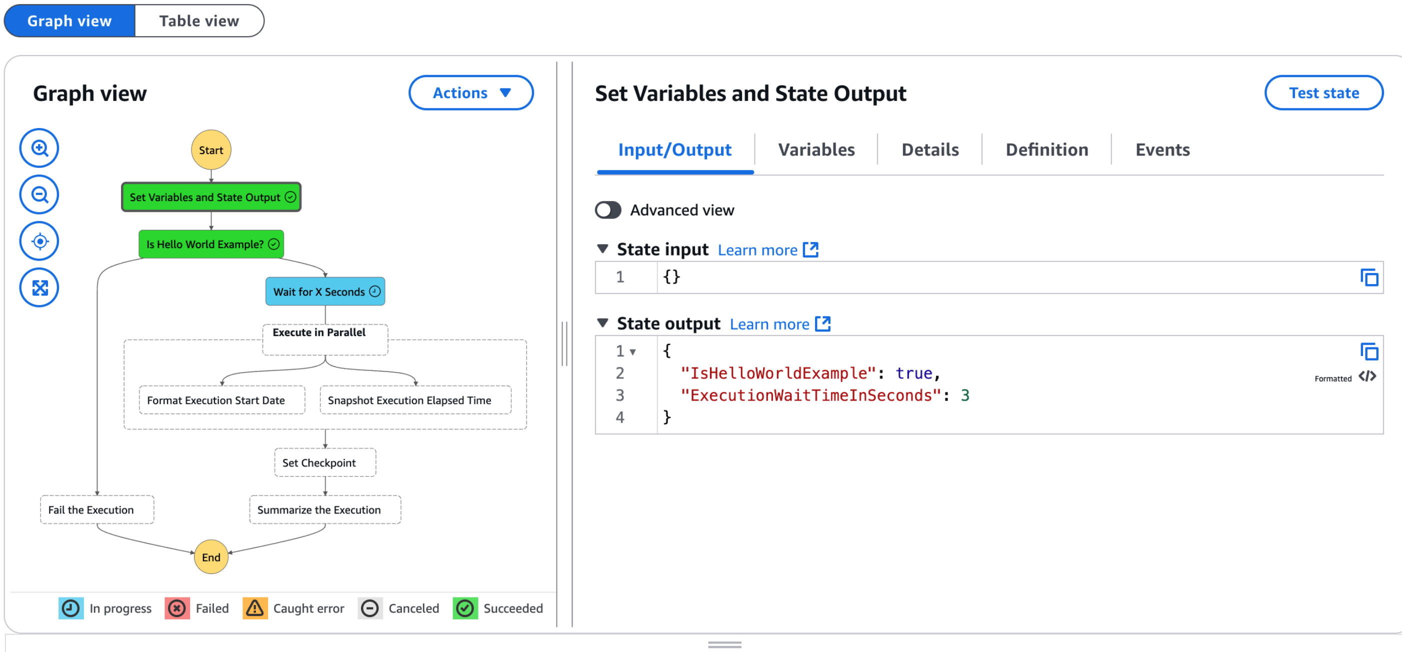
Immediately after starting, you should see the first two states have **succeeded**.

After a short wait, the rest of the state transitions will run to complete the workflow.

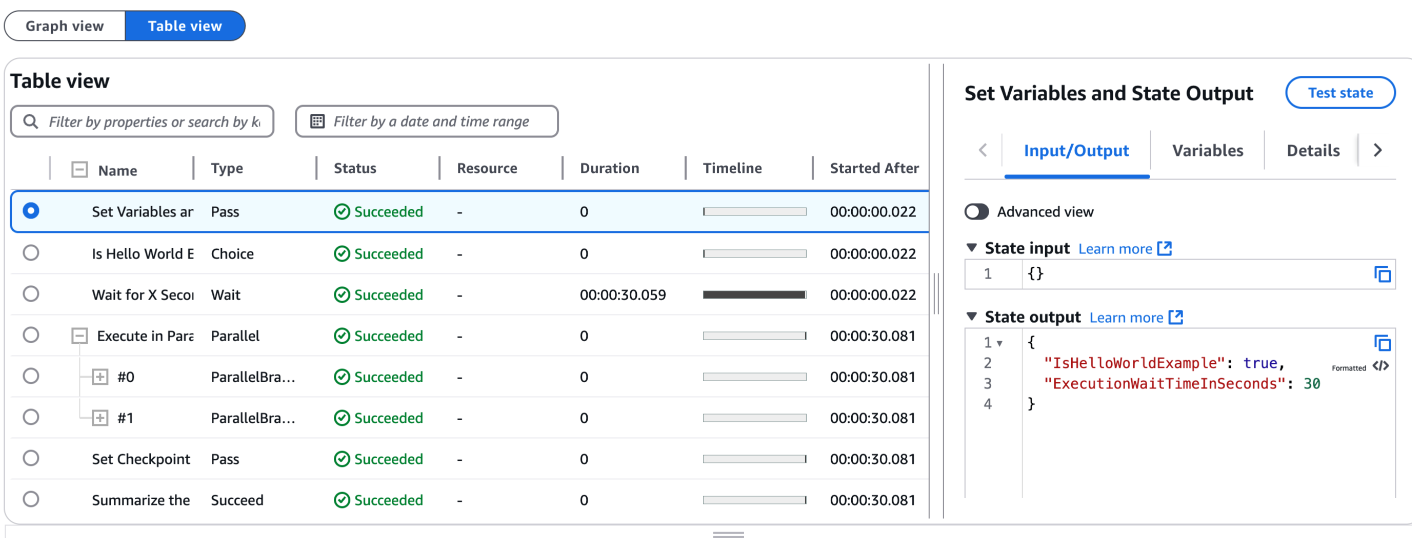
Are you wondering how the **Choice state** (*Is Hello World Example?*) decided to branch to the **Wait for X Seconds** state?

1. Hint: the first step in the state machine contains the data needed for the branch decision
2. In the **Graph View**, you can monitor progress during execution and explore details for each state.
3. Select the first **Pass state** (named *Set Variables and State Output*), then review the **Input/Output** tab.

You should see that **State input** is blank, but **State output** contains JSON that sets the value of `IsHelloWorldExample` to `true`.



Switch from the **Graph view** to the **Table view** to see a list of states by name, type, and status.

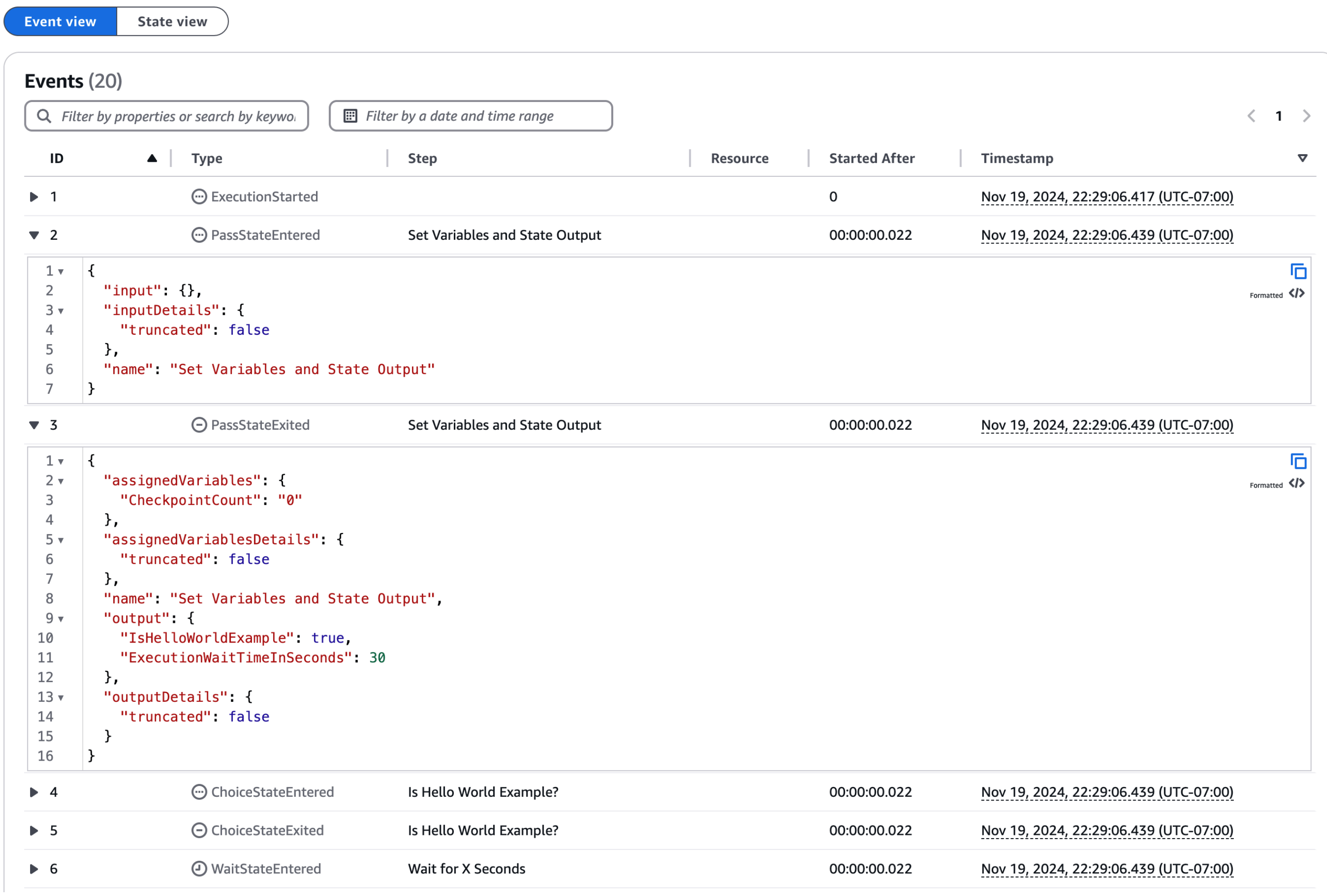


> **Tip**
>
> Take note of the **Duration** and **Timeline** fields in the previous screenshot. At a glance, you can see which states take more time than others.

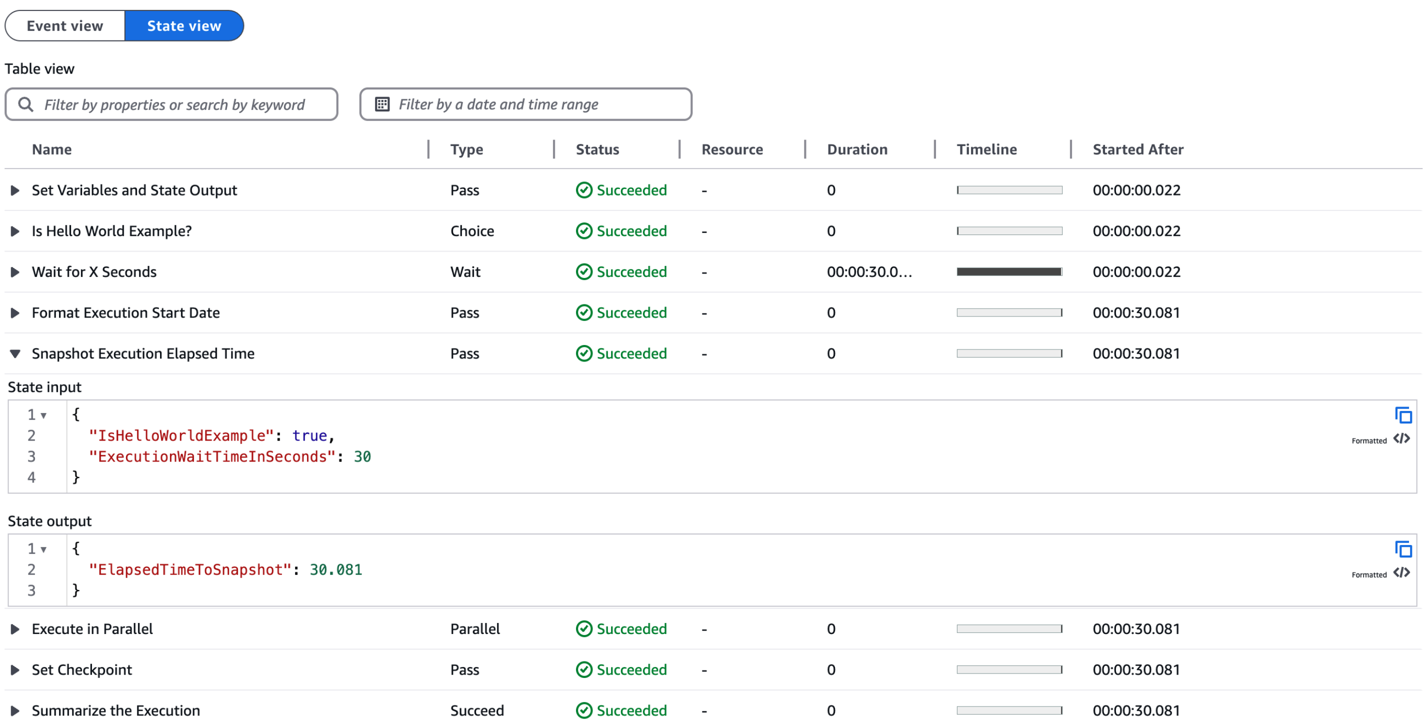
There are two more views to explore on this Executions Details page: **Event view** and **State view**.

The **Event view** is a detailed granular view of the flow from state to state.

Expand the first **PassStateEntered** and **PassStateExited** events in the **Event View** table to see how the state takes no input, assigns a variable called `CheckpointCount` the value of zero, and produces the output you saw previously.



Lastly, you have the **State view** which is similar to the **Table view**. In the **State view** table, you can selectively expand **states** to see just the Inputs and Outputs for each state:

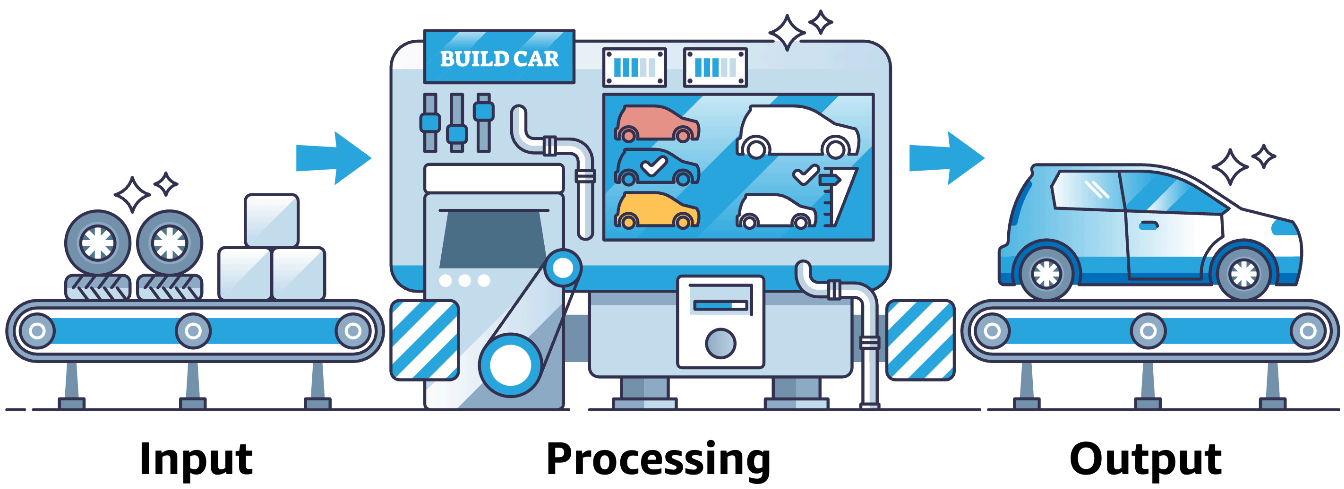


> **Congratulations! You've run your first Step Functions state machine!**
>
> Using a Pass state to add **static data** into a workflow is a common pattern, especially for troubleshooting.
> In the next step, you'll update the workflow so you can *dynamically* set your state machine input.

Step 3 - Process external input

Setting the value of `IsHelloWorldExample` to a constant value inside the workflow is not realistic. You should expect your state machine to respond to varying input data.

In this step, we'll show you how external JSON data can be used as input to your workflow:



Remove the hard-coded input

First, replace the hard-coded value in the **Output** of the first Pass state.

1. Edit your Hello World state machine by selecting the **Edit state machine** button located at the top right of the page.
2. Select the first **Pass state** after **Start** (named *Set Variables and State Output*), then select the **Output** tab.
3. Replace the **Output** with following JSON:

```
{
  "IsHelloWorldExample": "{% $states.input.hello_world %}",
  "ExecutionWaitTimeInSeconds":  "{% $states.input.wait %}"
}
```

4. Save the state machine.

The updated state output will pull input data from the reserved **$states** variable using a JSONata expression. Those values will be passed to the next state as output to become the input for the next state.

Run the updated workflow, with input data

Next, run the workflow and provide external input data as JSON.

1. Choose the **Execute** button to run the workflow.
2. For the **Name**, use the randomly generated ID.
3. Use the following JSON for the input field:

```
{
    "wait" : 20,
    "hello_world": true
}
```

4. Choose the **Start execution** button.

Your state machine execution should wait a lot longer (20 seconds), but eventually it should succeed using the input you provided.

In the Graph view, review the **Input/Output** for the first Pass State. Notice how the input you provided was converted into outputs. Also, take a look at the **Execution input and output** at the top of the execution details page. Both locations show the input that you used to start the execution.

> **Tip**
>
> What do you expect if you run a new execution with *hello_world* set to **false**? Try it!

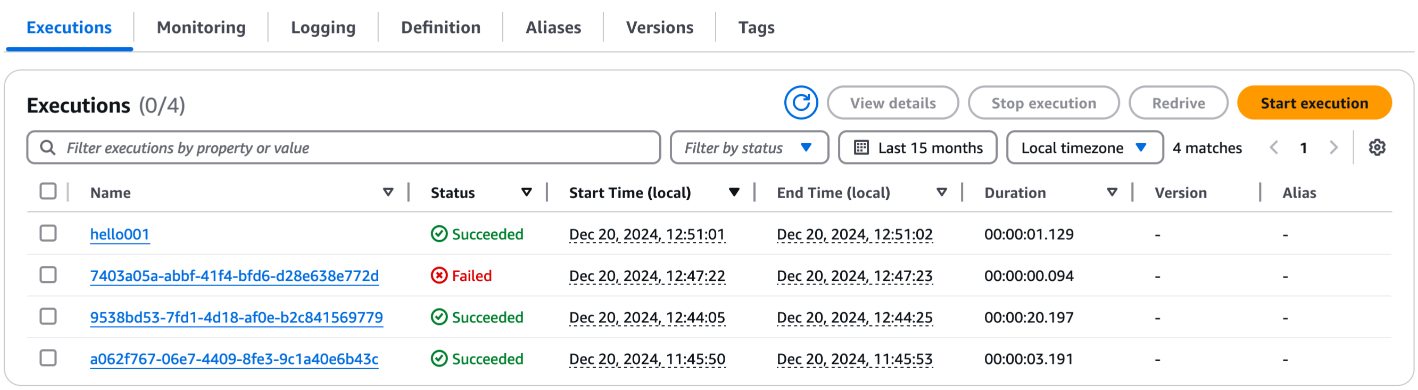
Review workflow executions

Now that you've run your workflow a few times, review the execution details to review runs of your workflow.

To review execution details

1. Choose **State machines** from the navigation breadcrumbs or left-hand menu.
2. Choose your state machine.

In the **Executions** tab, you should see a list of executions, similar to the following screenshot:



One final note: workflow execution names must be unique and **cannot** be reused. Although we suggested a short name (`hello001`) in this tutorial, we recommend using a naming convention that will always be unique for your production workloads.

> **Tip**
>
> **Congratulations!** You've modified your workflow to process *external input* that can vary every time you run your workflow.

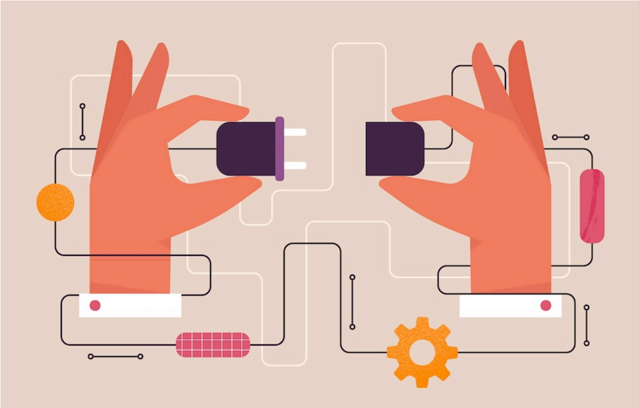
Step 4 - Integrate a service

Step Functions state machines can call over 220 AWS services using [AWS SDK integrations](). AWS services provide over 10,000 potential API actions for your state machines.

In this step, you will integrate an Amazon Comprehend task for **sentiment analysis** to process your state machine input.

Service integrations use one of three *service integration patterns*:

1. **Request a Response** (default) - wait for HTTP response, then *immediately* proceed to the next state.
2. **Run a Job** (*.sync*) - wait for a job to complete before moving to the next step.
3. **Wait for Callback** (*.waitForTaskToken*) - pause a workflow until a task token is returned by an external process.



For your first integration, you will use the **Request Response** (default) integration pattern.

How do integrations work?

A *Task state* represents a single unit of work performed by a state machine. All work in your state machine is done by tasks.

A task typically performs work by passing input to the API actions of other services which then perform their own work. You can specify how a Task performs, using a number of fields including: `Credentials`, `Retry`, `Catch`, `TimeoutSeconds`, and more. You can learn more about Tasks in the section called "Task".

To use AWS SDK integrations, you specify the **service name** and **API** to call. Some integrations also require parameters.

You can use Amazon States Language to specify an AWS API action in the **Resource** field of a task state. You may optionally add a service integration type to the service name.

To specify an API action, you will use the following resource name template:

```
arn:aws:states:::aws-sdk:serviceName:apiAction.[serviceIntegrationPattern]
```

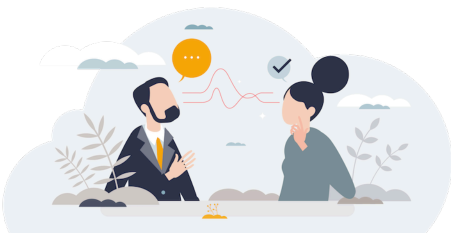
Parameter name case

Note that API actions will be *camelCase* (lowercase initial), but *ParameterNames* will be Pascal case (Uppercase initial).

Examples of resource names

- `arn:aws:states:::aws-sdk:ec2:describeInstances` will return the results from calling the Amazon EC2 describeInstances API.
- `arn:aws:states:::aws-sdk:s3:listBuckets` will return the results from calling the Amazon S3 listBuckets API.
- `arn:aws:states:::aws-sdk:sfn:startExecution` will start a nested Step Functions state machine execution and return the results of that workflow.

When Step Functions calls another service using the `Task` state, the default pattern is [Request Response](). With the **Request Response** integration pattern, Step Functions calls a service, receives a response, and ***immediately*** proceeds to the next state.



Step 4.1 - Add sentiment analysis state

1. **Edit** your **MyFirstStateMachine** state machine.
2. From the **Actions** panel in the **States browser**, search for `DetectSentiment`.
3. Drag & drop **Comprehend DetectSentiment** onto the **Default** branch of the **Choice state**.
4. Select and delete the **Fail** state.
5. From the **Flow tab** in the **States browser**, drag the **Success** state after **DetectSentiment**.

Step 4.2 - Configure the sentiment analysis state

1. Select the **Comprehend** step to configure it in the Inspector panel.
2. Select the **Arguments & Output** tab, then replace the **Arguments** with the following JSON:

```
{
   "LanguageCode": "en",
   "Text": "{% %}"
}
```

3. Place your cursor **between** the percent signs: `{% %}` and type: $

4. Use **auto-complete** in the editor to choose `states`,

   then type `.` and choose `context`,

   then type `.` and choose `Execution`,

   then type `.` and choose `Input`,

   finally, type `.feedback_comment` to retrieve initial input from the **Context Object**.

After choosing those auto-complete options, you should have the following JSON for your states **Arguments**:

```
{
    "LanguageCode": "en",
    "Text": "{% $states.context.Execution.Input.feedback_comment %}"
}
```

> **Using editor auto-complete**
>
> With editor auto-complete, you can explore your options.
> Auto-complete will list your variables, the reserved **$states** variable which contains the context object, and available functions with their definitions!

Step 4.3 - Configure an identity policy

Before you can run the workflow, you need to create a **role** and **policy** to allow the state machine to perform API calls to the external service.

To create an IAM role for Step Functions

1. Go to the IAM console in a new tab and select **Roles**.
2. Choose **Create a new role**.
3. For **Trusted entity type** choose *`AWS Service`*.
4. For **Use case** choose *`Step Functions`*.
5. For **Add permissions** choose **Next** to accept the default policy. You will add a policy for Comprehend after creating the role.

6. For **Name**, enter *HelloWorldWorkflowRole*.
7. Choose **Create role**.

To add a policy to the HelloWorldWorkflowRole for Amazon Comprehend

1. Select and edit the **HelloWorldWorkflowRole** role.
2. Choose **Add permission** then **Create inline policy**.
3. Select **Comprehend** for the service.
4. In **Read** choose **DetectSentiment**, then **Next**
5. For **Policy name** enter *DetectSentimentPolicy*, then **Create policy**. You should have created a policy as JSON, similar to the following: TIM

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "comprehend:DetectSentiment"
            ],
            "Resource": [
                "*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

To attach the IAM role to the Step Functions state machine

1. Return to editing your state machine and select the **Config** tab.
2. From the **Execution role** dropdown, choose *HelloWorldWorkflowRole*.
3. Save your state machine.

Step 4.4 - Run your state machine

Start executing your state machine with the following JSON for input:

```
{
    "hello_world": false,
    "wait": 42,
    "feedback_comment" : "This getting started with Step Functions workshop is a
 challenge!"
}
```

Troubleshooting a permissions error...

Without the correct policy, you will receive a **permissions error**, similar to the following:

```
User: arn:aws:sts::account-id:assumed-role/StepFunctions-MyStateMachine-role is not
 authorized
to perform: comprehend:DetectSentiment because no identity-based policy allows the
 comprehend:DetectSentiment
action (Service: Comprehend, Status Code: 400, Request ID: a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111)
```

The previous error message is telling you that your state machine is not authorized to use the external service. Go back a step and make sure you have configured an identity policy.

Practice what you've learned!

Before you dive into more complex workflows, practice what you've learned with the following tasks:

- Review the **DetectSentiment** step. Take a look at the input/output in the various views to see the results of sentiment detection.
- Find the **duration** of the DetectSentiment state in the table view.
- Change the comment in the **JSON input**, then re-run your state machine.

To learn more about sentiment analysis results, see [Amazon Comprehend - Sentiment](#).

One way to think about Request Response integration is the response generally represents only an *acknowledgement* of the request. However, in some integrations, such as sentiment analysis, the acknowledgement actually represents *completion* of the task.

The key learning is the `Task` state does **not wait** for the underlying job in Request Response integrations. To wait for a response, you'll need to explore the *Run a Job (.sync)* service integration pattern.

> **Congratulations!**
>
> You created your first state machine and integrated a sentiment analysis task using the **Request Response** pattern.

> **We value your feedback!**
>
> If you found this getting started tutorial helpful, or you have suggestions to improve the tutorial, let us know by using the feedback options on this page.

Clean up resources

Take the following steps to clean up the resources you created:

1. Navigate to the Step Functions page in the AWS Console.
2. Select **State machines** from the navigation pane on the left.
3. Choose the **MyFirstStateMachine**
4. To delete the IAM roles

   1 - Follow the link for the **IAM role** to go to the IAM role page in a new tab. Delete the custom related role.

   2 - In IAM Roles, search for the auto-generated role containing *`MyFirstStateMachine`*. Delete the auto-generated role.
5. Return to your Step Functions console tab and select the **Actions** drop down, then select **Delete** to delete the state machine.

Your state machine and related role should now be deleted successfully.

Learn about state machines in Step Functions

Step Functions is based on *state machines*, which are also called *workflows*. Workflows are comprised of a series of event-driven steps.

You define a workflow using Amazon States Language, also known as ASL. You can optionally use Workflow Studio, a visual workflow designer, to build and edit your workflows.

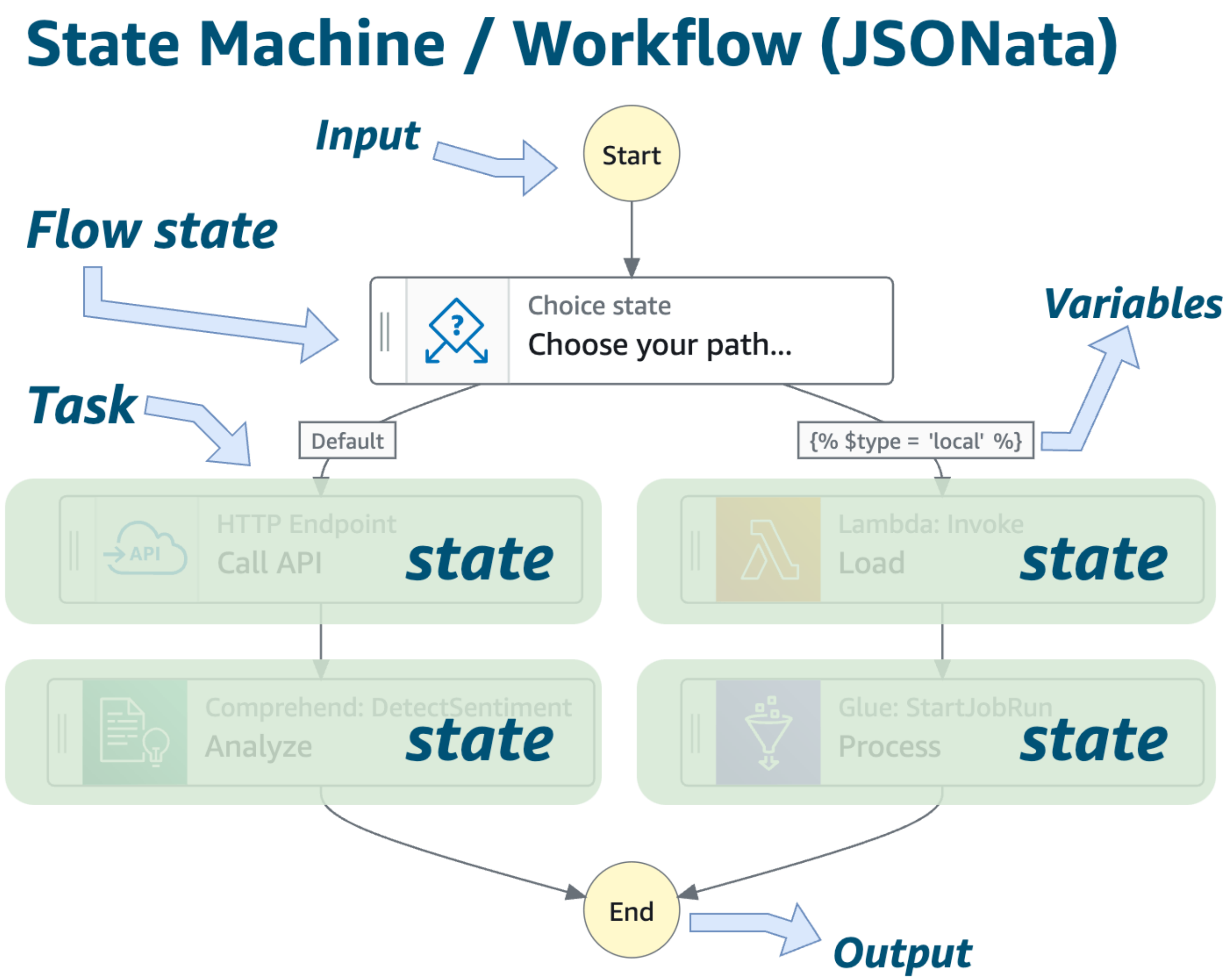
Each step in a workflow is called a *state*. There are two types of states: Flow states and Task states:

Flow states

Flow states control the flow of execution of the steps. For example, **Choice states** provide conditional logic; **Wait states** pause workflow execution; **Map states** run child workflows for each item in a dataset; and **Parallel states** create separate branches in your workflows.

Task states

Task states represent a unit of work that another AWS service performs, such as calling another AWS service or API. Tasks states are also known as **Actions**. You can choose hundreds of actions to perform work in AWS and external services. (Note: You can also use workers that run outside of Step Functions to perform tasks. For more info, see Activities.)



Executions and handling errors

When you run your workflows, Step Functions creates a workflow instance called an *execution*. You can monitor the status of your workflow executions. If an execution experiences an error, the workflow might catch the error. Depending on your use case, you might redrive the execution later to resume the workflow.

Passing data

You can optionally provide **input data** in the form of JSON text to your workflows. Each **step** can pass data to subsequent steps using variables and state output. Data stored in variables can be used by later steps. State output becomes the input for the very next step. To learn more about passing data, see the section called "Passing data with variables".

At the end of workflows, your state machine can optionally produce output, also in the form of JSON.

Transforming data

States and state machines can transform data using a **query language**. The recommended query language is **JSONata**; however, state machines created prior to re:Invent 2024 use **JSONPath**. For backward compatibility, your state machines or individual states must opt-in to using JSONata for their query language.

You can recognize JSONata state machines and individual states by the `QueryLanguage` field set to "JSONata". State machines and states that use JSONPath, lack the `QueryLanguage` field.

States that use JSONPath will have state fields such as InputPath, Parameters, ResultSelector, ResultPath, and OutputPath. In JSONPath state machine definitions, you will also see field names that end in `.$` and values prefixed with `$.` and `$$.`, both of which represent paths. In the paths, you might see various intrinsic functions, such as `States.MathAdd`. Intrinsic functions are **only** used in JSONPath.

JSONata states use **Arguments** and **Output** fields. In these optional fields, you might see JSONata expressions that look like the following: `"{% $type = 'local' %}"`. With JSONata, you can use expressions, operators, and functions. To learn more, see the section called "Transforming data with JSONata".

> **Note**
>
> You can use only one query language per state. You cannot mix JSONPath and JSONata within a single step.

Key concepts

The following provides an overview of the key Step Functions terms for context.

Term	Description
Workflow	A sequence of steps that often reflect a business process.
States	Individual steps in your state machine that can make decisions based on their input, perform actions from those inputs, and pass output to other states. For more information, see Discovering workflow states to use in Step Functions.
Workflow Studio	A visual workflow designer that helps you to prototype and build workflows faster. For more information, see Developing workflows in Step Functions Workflow Studio.
State machine	A workflow defined using JSON text representing the individual states or steps in the workflow along with fields, such as `StartAt`, `TimeoutSeconds` , and `Version`. For more information, see State machine structure in Amazon States Language for Step Functions workflows.
Amazon States Language	A JSON-based, structured language used to define your state machines. With ASL, you define a collection of states that can do work (`Task` state), determine which states to transition to next (`Choice` state), and stop an execution with an error (`Fail` state). For more information, see Using Amazon States Language to define Step Functions workflows.
Input and output configuration	States in a workflow receive JSON data as input and usually pass JSON data as output to the next state. Step Functions provides filters to control the data flow between states. For more information, see Processing input and output in Step Functions.
Service integration	You can call AWS service API actions from your workflow. For more information, see Integrating services with Step Functions.

Term	Description
Service integration type	• AWS SDK integrations – Standard way to call any of over two hundred AWS services and over nine thousand API actions directly from your state machine. • Optimized integrations – Custom integrations that streamline calling and exchanging data with certain services. For example, **Lambda Invoke** will automatically convert the `Payload` field of the response from an escaped JSON string into a JSON object.
Service integration pattern	When calling an AWS service, you use one of the following service integration patterns: • Request a response (default) – Call a service and move to the next state immediately after receiving an HTTP response. • Run a job (.sync) – Call a service and have Step Functions wait for a job to complete. • Wait for a callback with a task token (.waitForTaskToken) – Call a service with a task token and have Step Functions wait until the task token returns with a callback.
Execution	State machine executions are instances where you run your workflow to perform tasks. For more information, see Starting state machine executions in Step Functions.

State Machine Data

State machine data takes the following forms:

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

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

Topics

Data Format

State machine data is represented by JSON text. You can provide values to a state machine 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
>   - `null`
> - The output of a state becomes the input for the next state. However, you can restrict states to work on a subset of the input data by using Input and Output Processing.

State Machine Input/Output

You can give your initial input data to an AWS Step Functions state machine in one of two ways. You can pass the data to a `StartExecution` action when you start an execution. You can also pass the data to the state machine from 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, such as the `DescribeExecution` action. You can view execution results on the Step Functions console.

For Express Workflows, if you enabled logging, you can retrieve results from CloudWatch Logs, or view and debug the executions in the Step Functions console. For more information, see Using CloudWatch Logs to log execution history in Step Functions and Viewing execution details in the Step Functions console.

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

State Input/Output

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

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

1. Define the AWS Lambda function.

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

2. Define the state machine.

```
{
  "Comment": "An example that adds two numbers together.",
  "StartAt": "Add",
  "Version": "1.0",
  "TimeoutSeconds": 10,
  "States":
    {
        "Add": {
          "Type": "Task",
          "Resource": "arn:aws:lambda:region:123456789012:function:Add",
          "End": true
```

```
        }
    }
}
```

3. Start an execution with the following JSON text.

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

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

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

   The state returns the following value in its output.

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

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

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

For more information, see Processing input and output in Step Functions.

Invoke AWS Step Functions from other services

You can configure several other services to invoke state machines. Based on the state machine's workflow type, you can invoke state machines asynchronously or synchronously. To invoke state machines synchronously, use the `StartSyncExecution` API call or Amazon API Gateway integration with Express Workflows. With asynchronous invocation, Step Functions pauses the workflow execution until a task token is returned. However, waiting for a task token does make the workflow synchronous.

Services that you can configure 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:

- Step Functions service quotas

Transitions in state machines

When you start a new execution of your state machine, the system begins with the state referenced in the top-level `StartAt` field. This field, given as a string, must exactly match, including case, the name of a state in the workflow.

After a state runs, 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. This string is case-sensitive and must match the name of a state specified in the state machine description exactly

For example, the following state includes a transition to `NextState`.

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

Most states permit only a single transition rule with the `Next` field. However, certain flow-control states, such as a `Choice` state, allow you to specify multiple transition rules, each with its own `Next` field. The Amazon States Language 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 either reaches a terminal state (a state with `"Type": Succeed`, `"Type": Fail`, or `"End": true`), or a runtime error occurs.

When you redrive an execution, it's considered as a state transition. In addition, all states that are rerun in a redrive are also considered as state transitions.

The following rules apply to states within a state machine:

- States can occur in any order within the enclosing block. However, 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. The `start` state is defined 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 logic branches — you may have more than one `end` state.
- If your state machine consists of only one state, it can be both the start and end state.

Transitions in Distributed Map state

When you use the `Map` state in Distributed mode, you'll be charged one state transition for each child workflow execution that the *Distributed Map state* starts. When you use the `Map` state in Inline mode, you aren't charged a state transition for each iteration of the *Inline Map state*.

You can optimize cost by using the `Map` state in Distributed mode and include a nested workflow in the `Map` state definition. The *Distributed Map state* also adds more value when you start child workflow executions of type **Express**. Step Functions stores the response and status of the Express child workflow executions, which reduces the need to store execution data in CloudWatch Logs. You can also get access to flow controls available with a *Distributed Map state*, such as defining error thresholds or batching a group of items. For information about Step Functions pricing, see [AWS Step Functions pricing](#).

Read Consistency in Step Functions

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

For more information, see the following:

- [`UpdateStateMachine`](#) in the *AWS Step Functions API Reference*

Learn about Activities in Step Functions

With Step Functions activities, you can set up a task in your state machine where the actual work is performed by a *worker* running outside of Step Functions. For example you could have a worker program running on Amazon Elastic Compute Cloud (Amazon EC2), Amazon Elastic Container Service (Amazon ECS), or even mobile devices.

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 Avoiding latency when polling for activity tasks.

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` 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.

Example: Activity Worker in Ruby

The following example activity worker 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 in Step Functions. When an activity task is retrieved, it is passed through a bounded blocking queue for the activity thread to pick up.

- For more information, 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 code is the main entry point for this example Ruby activity worker.

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

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

# Start method block contains your custom implementation to process the input
activity.start do |input|
  { result: :SUCCESS, echo: input['value'] }
end
```

You must specify your activity ARN and region. The code includes defaults that you can set, such as number of threads and heartbeat delay.

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

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

Choosing workflow type in Step Functions

When you create a state machine, you must choose a **Type** of either *Standard* (default) or *Express*, referred to commonly as a standard workflow or an express workflow.

You define both state machine types using the Using Amazon States Language to define Step Functions workflows.

Both standard and express workflows can start in response to events, such as HTTP requests from Amazon API Gateway, IoT rules, and over 140 other event sources in Amazon EventBridge.

> **Workflow type is immutable**
>
> The workflow type can **not** be updated after you create a state machine.

Standard Workflows are ideal for long-running (up to one year), durable, and auditable workflows. You can retrieve the full execution history using the Step Functions API for up to 90 days after your execution completes.

Standard Workflows follow an *exactly-once* model, where your tasks and states are never run more than once, unless you have specified `Retry` behavior in ASL. The exactly-once model makes Standard Workflows suited to orchestrating **non-idempotent** actions, such as starting an Amazon EMR cluster or processing payments.

Standard Workflow 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 use an *at-least-once* model, so an execution could potentially run more than once. The at-least-once model makes Express Workflows better suited for orchestrating **idempotent** actions, such as transforming input data to store in Amazon DynamoDB using a PUT action.

Express Workflow executions are billed by number of executions, total duration of execution, and memory consumed during execution.

> **Tip**
>
> To deploy an example Express workflow, see Processing data in parallel in *The AWS Step Functions Workshop*.

Comparison of Standard and Express workflow types

Type / Category	Standard Workflows	Express Workflows: Synchronous and Asynchronous
Maximum duration	One year	Five minutes
Supported execution start rate	For information about quotas related to supported execution start rate, see Quotas related to API action throttling.	For information about quotas related to supported execution start rate, see Quotas related to API action throttling.
Supported state transition rate	For information about quotas related to supported state transition rate, see Quotas related to state throttling.	No limit
Pricing	Priced by number of state transitions. A *state transition* is counted each time a step in your execution is completed.	Priced by the number of executions you run, their duration, and memory consumption.
Execution history	Executions can be listed and described with Step Functions APIs. Executions can be visually debugged through the console. They can also be inspected in CloudWatch Logs by enabling logging on your state machine.	Unlimited execution history, that is, as many execution history entries are maintained as you can generate within a 5-minute period. Executions can be inspected in CloudWatch Logs or the

Type / Category	Standard Workflows	Express Workflows: Synchronous and Asynchronous
	For more information about debugging Standard Workflow executions in the console, see Standard and Express console experience differences and Viewing workflow runs.	Step Functions console by enabling logging on your state machine. For more information about debugging Express Workflow executions in the console, see Standard and Express console experience differences and Viewing workflow runs.
Execution semantics	*Exactly-once* workflow execution.	*Asynchronous Express Workflows*: *At-least-once* workflow execution. *Synchronous Express Workflows*: *At-most-once* workflow execution.
Service integrations	Supports all service integrations and patterns.	Supports all service integrations. **Note** Express Workflows do not support Job-run (`.sync`) or Callback (`.waitForTaskToken`) service integration patterns.
Distributed Map	Supported	Not supported
Activities	Supported	Not supported

Optimize workflow type

For a comparison and an example cost impact analysis, see Choosing the workflow type in the Large-scale data processing with Step Functions workshop.

Synchronous and Asynchronous Express Workflows in Step Functions

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 don't wait for the workflow to complete. To get the result, you must poll the service's CloudWatch Logs. You can use Asynchronous Express Workflows when you don't require immediate response output, such as messaging services or data processing that other services don't depend on. You can start Asynchronous Express Workflows 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, and then return the result. Synchronous Express Workflows can be used to orchestrate microservices. With Synchronous Express Workflows, you can develop applications without the need to develop additional code to handle errors, retries, or run parallel tasks. You can run Synchronous Express Workflows 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 expires after 60 seconds. To run the Express Workflows synchronously for a duration of up to five minutes, make the `StartSyncExecution` request using the AWS SDK or AWS Command Line Interface (AWS CLI) instead of the Step Functions console.

Synchronous Express execution API calls don't contribute to existing account capacity limits. Step Functions provides capacity on demand and automatically scales with sustained workload. Surges in workload may be throttled until capacity is available.

Execution guarantees in Step Functions workflows

Standard Workflows	Asynchronous Express Workflows	Synchronous Express Workflows
Exactly-once workflow execution	*At-least-once* workflow execution	*At-most-once* workflow execution
Execution state internally persists between state transitions.	Execution state doesn't persist between state transitions.	Execution state doesn't persist between state transitions.
Automatically returns an idempotent response on starting an execution with the same name as a currently-running workflow. The new workflow doesn't start and an exception is thrown once the currently-running workflow is complete.	Idempotency is not automatically managed. Starting multiple workflows with the same name results in concurrent executions. Can result in loss of internal workflow state if state machine logic is not idempotent.	Idempotency is not automatically managed. Step Functions waits once an execution starts and returns the state machine's result on completion. Workflows don't restart if an exception occurs.
Execution history data removed after 90 days. Workflow names can be reused after removal of out-of-date execution data. To meet compliance, organizational, or regulatory requirements, you can reduce the execution history	Execution history is not captured by Step Functions. Logging must be enabled through Amazon CloudWatch Logs.	Execution history is not captured by Step Functions. Logging must be enabled through Amazon CloudWatch Logs.

Standard Workflows	Asynchronous Express Workflows	Synchronous Express Workflows
retention period to 30 days by sending a quota request. To do this, use the AWS Support Center Console and create a new case.		

Using Amazon States Language to define Step Functions workflows

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

For 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.

Note

If you define your state machines outside the Step Functions' console, such as in an editor of your choice, you must save your state machine definitions with the extension *.asl.json*.

Example Amazon States Language Specification (JSONata)

```
{
  "Comment": "An example of the Amazon States Language using a choice state.",
  "QueryLanguage": "JSONata",
  "StartAt": "FirstState",
  "States": {
    "FirstState": {
      "Type": "Task",
      "Assign": {
        "foo" : "{% $states.input.foo_input %}"
        },
      "Resource": "arn:aws:lambda:region:123456789012:function:FUNCTION_NAME",
      "Next": "ChoiceState"
    },
    "ChoiceState": {
      "Type": "Choice",
      "Default": "DefaultState",
      "Choices": [
        {
```

```
          "Next": "FirstMatchState",
          "Condition": "{% $foo = 1 %}"
        },
        {
          "Next": "SecondMatchState",
          "Condition": "{% $foo = 2 %}"
        }
      ]
    },
    "FirstMatchState": {
      "Type" : "Task",
      "Resource": "arn:aws:lambda:region:123456789012:function:OnFirstMatch",
      "Next": "NextState"
    },

    "SecondMatchState": {
      "Type" : "Task",
      "Resource": "arn:aws:lambda:region:123456789012:function:OnSecondMatch",
      "Next": "NextState"
    },

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

    "NextState": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:region:123456789012:function:FUNCTION_NAME",
      "End": true
    }
  }
}
```

State machine structure in Amazon States Language for Step Functions workflows

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

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.

`QueryLanguage` (Optional; when omitted, defaults to `JSONPath`)

- The name of the query language used by the state machine. Allowed values are `JSONPath` and `JSONata`.
- If not provided for the state machine, the default value for each state is JSONPath.
- When the top-level state machine query language is `JSONPath`, individual states can override the query language by setting QueryLanguage to `JSONata`. Given this approach, you can incrementally convert a state machine from JSONPath to JSONata one state at a time.
- **Note:** You cannot revert a top-level JSONata-based state machine to a mix of JSONata and JSONPath states.

`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.

`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.

```
{
    "State1" : {
    },

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

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

The following is an example.

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

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

The following rules apply to states within a state machine:

- States can occur in any order within the enclosing block, but the order in which they're listed doesn't affect the order in which they're run. The contents of the states determines this order.
- Within a state machine, there can be only one state that's designated as the `start` state, designated by the value of the `StartAt` field in the top-level structure. This state is the one that is executed first when the execution starts.

- Any state for which the End field is `true` is considered an end (or `terminal`) state. Depending on your state machine logic—for example, if your state machine has multiple branches of execution—you might have more than one end state.
- If your state machine consists of only one state, it can be both the `start` state and the end state.

Common state fields in workflows

The following fields are common to all state elements.

Type (Required)

The state's type.

`QueryLanguage` (Optional; when omitted, defaults to `JSONPath`)

- The name of the query language used by the state. Allowed values are `JSONPath` and `JSONata`.
- When the top-level state machine query language is `JSONPath`, individual states can override the query language by setting QueryLanguage to `JSONata`. Given this approach, you can incrementally convert a state machine from JSONPath to JSONata one state at a time.

`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.

If the current state is the last state in your workflow, or a terminal state, such as Succeed workflow state or Fail workflow state, you don't need to specify the `Next` field.

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`, or terminal states, such as Succeed workflow state and Fail workflow state, don't support or use the End field.

`Comment` (Optional)

Holds a human-readable description of the state.

`Assign` (Optional)

Used to store variables. The `Assign` field accepts a JSON object with key/value pairs that define variable names and their assigned values. Any string value, including those inside objects or arrays, will be evaluated as JSONata when surrounded by `{% %}` characters

For more information, see the section called "Passing data with variables".

`Output` (Optional, JSONata only)

Used to specify and transform output from the state. When specified, the value overrides the state output default.

The output field accepts any JSON value (object, array, string, number, boolean, null). Any string value, including those inside objects or arrays, will be evaluated as JSONata if surrounded by {% %} characters.

Output also accepts a JSONata expression directly, for example: "Output": "{% jsonata expression %}"

For more information, see Input and Output Processing.

`InputPath` (Optional, JSONPath only)

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

`OutputPath` (Optional, JSONPath only)

A path 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 output. For more information, see Input and Output Processing.

Intrinsic functions for JSONPath states in Step Functions

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

> **Warning**
>
> Intrinsic functions are only available to states that use the **JSONPath** query language. For JSONata, see the section called “Transforming data with JSONata”.

The Amazon States Language provides several intrinsic functions, also known as *intrinsics*, for use in fields that accept JSONPath. With intrinsics, you can perform basic data processing operations without using a `Task` state.

Intrinsics look similar to functions in programming languages. They can be used to help payload builders process the data going to and from the `Resource` field of a `Task` state that uses the JSONPath query language.

In Amazon States Language, intrinsic functions are grouped into the following categories, based on the type of data processing task that you want to perform:

- Intrinsics for arrays
- Intrinsics for data encoding and decoding
- Intrinsic for hash calculation
- Intrinsics for JSON data manipulation
- Intrinsics for Math operations
- Intrinsic for String operation
- Intrinsic for unique identifier generation
- Intrinsic for generic operation

To use intrinsic functions, you must specify `.$` in the key value in your state machine definitions, as shown in the following example:

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

You can nest up to 10 intrinsic functions within a field in your workflows. The following example shows a field named *myArn* that includes nine nested intrinsic functions:

```
"myArn.$": "States.Format('{}.{}.{}',
 States.ArrayGetItem(States.StringSplit(States.ArrayGetItem(States.StringSplit($.ImageRecipe.Ar
 '/'), 2), '.'), 0),
```

```
States.ArrayGetItem(States.StringSplit(States.ArrayGetItem(States.StringSplit($.ImageRecipe.Ar
'/'), 2), '.'), 1))"
```

> **QueryLanguage required for intrinsic functions**
>
> To use intrinsic functions, the state machine must use the **JSONPath query language**. States that use JSONata cannot use intrinsic functions; however, JSONata and Step Functions provide equivalent options.

Fields that support intrinsic functions

The following states support intrinsic functions in the following fields:

- **Pass state** : Parameters
- **Task state** : Parameters, ResultSelector, Credentials
- **Parallel state**: Parameters, ResultSelector
- **Map state**: Parameters, ResultSelector

Intrinsics for arrays

Use the following intrinsics for performing array manipulations.

`States.Array`

The `States.Array` intrinsic function 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:

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

You could use

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

Which would return the following result:

```
“BuildId”: [123456]
```

`States.ArrayPartition`

Use the `States.ArrayPartition` intrinsic function to partition a large array. You can also use this intrinsic to slice the data and then send the payload in smaller chunks.

This intrinsic function takes two arguments. The first argument is an array, while the second argument defines the chunk size. The interpreter chunks the input array into multiple arrays of the size specified by chunk size. The length of the last array chunk may be less than the length of the previous array chunks if the number of remaining items in the array is smaller than the chunk size.

Input validation

- You must specify an array as the input value for the function's first argument.
- You must specify a non-zero, positive integer for the second argument representing the chunk size value.

  If you specify a non-integer value for the second argument, Step Functions will round it off to the nearest integer.
- The input array can't exceed Step Functions' payload size limit of 256 KiB.

For example, given the following input array:

```
{"inputArray": [1,2,3,4,5,6,7,8,9] }
```

You could use the `States.ArrayPartition` function to divide the array into chunks of four values:

```
"inputArray.$": "States.ArrayPartition($.inputArray,4)"
```

Which would return the following array chunks:

```
{"inputArray": [ [1,2,3,4], [5,6,7,8], [9]] }
```

In the previous example, the `States.ArrayPartition` function outputs three arrays. The first two arrays each contain four values, as defined by the chunk size. A third array contains the remaining value and is smaller than the defined chunk size.

States.ArrayContains

Use the `States.ArrayContains` intrinsic function to determine if a specific value is present in an array. For example, you can use this function to detect if there was an error in a `Map` state iteration.

This intrinsic function takes two arguments. The first argument is an array, while the second argument is the value to be searched for within the array.

Input validation

- You must specify an array as the input value for function's first argument.
- You must specify a valid JSON object as the second argument.
- The input array can't exceed Step Functions' payload size limit of 256 KiB.

For example, given the following input array:

```
{
   "inputArray": [1,2,3,4,5,6,7,8,9],
   "lookingFor": 5
}
```

You could use the `States.ArrayContains` function to find the `lookingFor` value within the `inputArray`:

```
"contains.$": "States.ArrayContains($.inputArray, $.lookingFor)"
```

Because the value stored in `lookingFor` is included in the `inputArray`, `States.ArrayContains` returns the following result:

```
{"contains": true }
```

States.ArrayRange

Use the `States.ArrayRange` intrinsic function to create a new array containing a specific range of elements. The new array can contain up to 1000 elements.

This function takes three arguments. The first argument is the first element of the new array, the second argument is the final element of the new array, and the third argument is the increment value between the elements in the new array.

Input validation

- You must specify integer values for all of the arguments.

  If you specify a non-integer value for any of the arguments, Step Functions will round it off to the nearest integer.
- You must specify a non-zero value for the third argument.
- The newly generated array can't contain more than 1000 items.

For example, the following use of the `States.ArrayRange` function will create an array with a first value of 1, a final value of 9, and values in between the first and final values increase by two for each item:

```
"array.$": "States.ArrayRange(1, 9, 2)"
```

Which would return the following array:

```
{"array": [1,3,5,7,9] }
```

`States.ArrayGetItem`

This intrinsic function returns a specified index's value. This function takes two arguments. The first argument is an array of values and the second argument is the array index of the value to return.

For example, use the following `inputArray` and `index` values:

```
{
   "inputArray": [1,2,3,4,5,6,7,8,9],
   "index": 5
}
```

From these values, you can use the `States.ArrayGetItem` function to return the value in the `index` position 5 within the array:

```
"item.$": "States.ArrayGetItem($.inputArray, $.index)"
```

In this example, `States.ArrayGetItem` would return the following result:

```
{ "item": 6 }
```

`States.ArrayLength`

The `States.ArrayLength` intrinsic function returns the length of an array. It has one argument, the array to return the length of.

For example, given the following input array:

```
{
   "inputArray": [1,2,3,4,5,6,7,8,9]
}
```

You can use `States.ArrayLength` to return the length of `inputArray`:

```
"length.$": "States.ArrayLength($.inputArray)"
```

In this example, `States.ArrayLength` would return the following JSON object that represents the array length:

```
{ "length": 9 }
```

`States.ArrayUnique`

The `States.ArrayUnique` intrinsic function removes duplicate values from an array and returns an array containing only unique elements. This function takes an array, which can be unsorted, as its sole argument.

For example, the following `inputArray` contains a series of duplicate values:

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

You could use the `States.ArrayUnique` function as and specify the array you want to remove duplicate values from:

```
"array.$": "States.ArrayUnique($.inputArray)"
```

The `States.ArrayUnique` function would return the following array containing only unique elements, removing all duplicate values:

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

Intrinsics for data encoding and decoding

Use the following intrinsic functions to encode or decode data based on the Base64 encoding scheme.

`States.Base64Encode`

Use the `States.Base64Encode` intrinsic function to encode data based on MIME Base64 encoding scheme. You can use this function to pass data to other AWS services without using an AWS Lambda function.

This function takes a data string of up to 10,000 characters to encode as its only argument.

For example, consider the following `input` string:

```
{"input": "Data to encode" }
```

You can use the `States.Base64Encode` function to encode the `input` string as a MIME Base64 string:

```
"base64.$": "States.Base64Encode($.input)"
```

The `States.Base64Encode` function returns the following encoded data in response:

```
{"base64": "RGF0YSB0byBlbmNvZGU=" }
```

`States.Base64Decode`

Use the `States.Base64Decode` intrinsic function to decode data based on MIME Base64 decoding scheme. You can use this function to pass data to other AWS services without using a Lambda function.

This function takes a Base64 encoded data string of up to 10,000 characters to decode as its only argument.

For example, given the following input:

```
{"base64": "RGF0YSB0byBlbmNvZGU=" }
```

You can use the `States.Base64Decode` function to decode the base64 string to a human-readable string:

```
"data.$": "States.Base64Decode($.base64)"
```

The `States.Base64Decode function` would return the following decoded data in response:

```
{"data": "Decoded data" }
```

Intrinsic for hash calculation

`States.Hash`

Use the `States.Hash` intrinsic function to calculate the hash value of a given input. You can use this function to pass data to other AWS services without using a Lambda function.

This function takes two arguments. The first argument is the data you want to calculate the hash value of. The second argument is the hashing algorithm to use to perform the hash calculation. The data you provide must be an object string containing 10,000 characters or less.

The hashing algorithm you specify can be any of the following algorithms:

- `MD5`
- `SHA-1`
- `SHA-256`
- `SHA-384`
- `SHA-512`

For example, you can use this function to calculate the hash value of the `Data` string using the specified `Algorithm`:

```
{
   "Data": "input data",
   "Algorithm": "SHA-1"
}
```

You can use the `States.Hash` function to calculate the hash value:

```
"output.$": "States.Hash($.Data, $.Algorithm)"
```

The `States.Hash` function returns the following hash value in response:

```
{"output": "aaff4a450a104cd177d28d18d7485e8cae074b7" }
```

Intrinsics for JSON data manipulation

Use these functions to perform basic data processing operations on JSON objects.

`States.JsonMerge`

Use the `States.JsonMerge` intrinsic function to merge two JSON objects into a single object. This function takes three arguments. The first two arguments are the JSON objects that you want to merge. The third argument is a boolean value of `false`. This boolean value determines if the deep merging mode is enabled.

Currently, Step Functions only supports the shallow merging mode; therefore, you must specify the boolean value as `false`. In the shallow mode, if the same key exists in both JSON objects, the latter object's key overrides the same key in the first object. Additionally, objects nested within a JSON object are not merged when you use shallow merging.

For example, you can use the `States.JsonMerge` function to merge the following JSON objects that share the key `a`.

```
{
   "json1": { "a": {"a1": 1, "a2": 2}, "b": 2 },
   "json2": { "a": {"a3": 1, "a4": 2}, "c": 3 }
}
```

You can specify the json1 and json2 objects as inputs in the `States.JsonMerge` function to merge them together:

```
"output.$": "States.JsonMerge($.json1, $.json2, false)"
```

The `States.JsonMerge` returns the following merged JSON object as result. In the merged JSON object `output`, the `json2` object's key `a` replaces the `json1` object's key `a`. Also, the nested object in `json1` object's key a is discarded because shallow mode doesn't support merging nested objects.

```
{
   "output": {
      "a": {"a3": 1, "a4": 2},
      "b": 2,
      "c": 3
   }
}
```

`States.StringToJson`

The `States.StringToJson` function takes a reference path to an escaped JSON string as its only argument.

The interpreter applies a JSON parser and returns the input's parsed JSON form. For example, you can use this function to escape the following input string:

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

Use the `States.StringToJson` function and specify the `escapedJsonString` as the input argument:

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

The `States.StringToJson` function returns the following result:

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

`States.JsonToString`

The `States.JsonToString` function takes only one argument, which is the path that contains the JSON data to return as an unescaped string. The interpreter returns a string that contains JSON text representing the data specified by the Path. For example, you can provide the following JSON Path containing an escaped value:

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

Provide the `States.JsonToString` function with the data contained within `unescapedJson`:

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

The `States.JsonToString` function returns the following response:

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

Intrinsics for Math operations

Use these functions to perform Math operations.

`States.MathRandom`

Use the `States.MathRandom` intrinsic function to return a random number between the specified start number (inclusive) and end number (exclusive).

You can use this function to distribute a specific task between two or more resources.

This function takes three arguments. The first argument is the start number, the second argument is the end number, and the last argument controls the optional seed value, Note that if you use this function with the same seed value, it will return identical numbers.

> **Important**
>
> Because the `States.MathRandom` function does not return cryptographically secure random numbers, we recommend that you don't use it for security sensitive applications.

Input validation

- You must specify integer values for the start number and end number arguments.

  If you specify a non-integer value for the start number or end number argument, Step Functions will round it off to the nearest integer.

For example, to generate a random number between one and 999, you can use the following input values:

```
{
   "start": 1,
   "end": 999
}
```

To generate the random number, provide the `start` and `end` values to the `States.MathRandom` function:

```
"random.$": "States.MathRandom($.start, $.end)"
```

The `States.MathRandom` function returns the following random number as a response:

```
{"random": 456 }
```

`States.MathAdd`

Use the `States.MathAdd` intrinsic function to return the sum of two numbers. For example, you can use this function to increment values inside a loop without invoking a Lambda function.

Input validation

- You must specify integer values for all the arguments.

  If you specify a non-integer value for one or both the arguments, Step Functions will round it off to the nearest integer.
- You must specify integer values in the range of -2147483648 and 2147483647.

For example, you can use the following values to subtract one from 111:

```
{
```

```
    "value1": 111,
    "step": -1
}
```

Then, use the `States.MathAdd` function defining `value1` as the starting value, and `step` as the value to increment `value1` by:

```
"value1.$": "States.MathAdd($.value1, $.step)"
```

The `States.MathAdd` function would return the following number in response:

```
{"value1": 110 }
```

Intrinsic for String operation

`States.StringSplit`

Use the `States.StringSplit` intrinsic function to split a string into an array of values. This function takes two arguments. The first argument is a string and the second argument is the delimiting character that the function will use to divide the string.

Example - Split an input string using a single delimiting character

For this example, use `States.StringSplit` to divide the following `inputString`, which contains a series of comma separated values:

```
{
    "inputString": "1,2,3,4,5",
    "splitter": ","
}
```

Use the `States.StringSplit` function and define `inputString` as the first argument, and the delimiting character `splitter` as the second argument:

```
"array.$": "States.StringSplit($.inputString, $.splitter)"
```

The `States.StringSplit` function returns the following string array as result:

```
{"array":  ["1","2","3","4","5"] }
```

Example - Split an input string using multiple delimiting characters

For this example, use `States.StringSplit` to divide the following `inputString`, which contains multiple delimiting characters:

```
{
  "inputString": "This.is+a,test=string",
  "splitter": ".+,="
}
```

Use the `States.StringSplit` function as follows:

```
{
  "myStringArray.$": "States.StringSplit($.inputString, $.splitter)"
}
```

The `States.StringSplit` function returns the following string array as result:

```
{"myStringArray": [
  "This",
  "is",
  "a",
  "test",
  "string"
]}
```

Intrinsic for unique identifier generation

`States.UUID`

Use the `States.UUID` intrinsic function to return a version 4 universally unique identifier (v4 UUID) generated using random numbers. For example, you can use this function to call other AWS services or resources that need a UUID parameter or insert items in a DynamoDB table.

The `States.UUID` function is called with no arguments specified:

```
"uuid.$": "States.UUID()"
```

The function returns a randomly generated UUID, as in the following example:

```
{"uuid": "ca4c1140-dcc1-40cd-ad05-7b4aa23df4a8" }
```

Intrinsic for generic operation

`States.Format`

Use the `States.Format` intrinsic function to construct a string from both literal and interpolated values. This function 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 intrinsic function invocation as there are occurrences of `{}`. The interpreter returns the string defined in the first argument with each `{}` replaced by the value of the positionally-corresponding argument in the Intrinsic invocation.

For example, you can use the following inputs of an individual's `name`, and a `template` sentence to have their name inserted into:

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

Use the `States.Format` function and specify the `template` string and the string to insert in place of the `{}` characters:

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

or

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

With either of the previous inputs, the `States.Format` function returns the completed string in response:

```
Hello, my name is Arnav.
```

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, you must escape it with a backslash. The following escaped character sequences are used with intrinsic functions:

- 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 `\\`.

Note

If an open escape backslash \ is found in the intrinsic invocation string, the interpreter will return a runtime error.

You must use square bracket notation for a **Path** passed as an argument to an Intrinsic Function if the field name contains any character that is not included in the `member-name-shorthand` definition of the JsonPath ABNF rule. If your **Path** contains non-alphanumeric characters, besides **_**, you must use square bracket notation. For example, `$.abc.['def ghi']`.

Discovering workflow states to use in Step Functions

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.

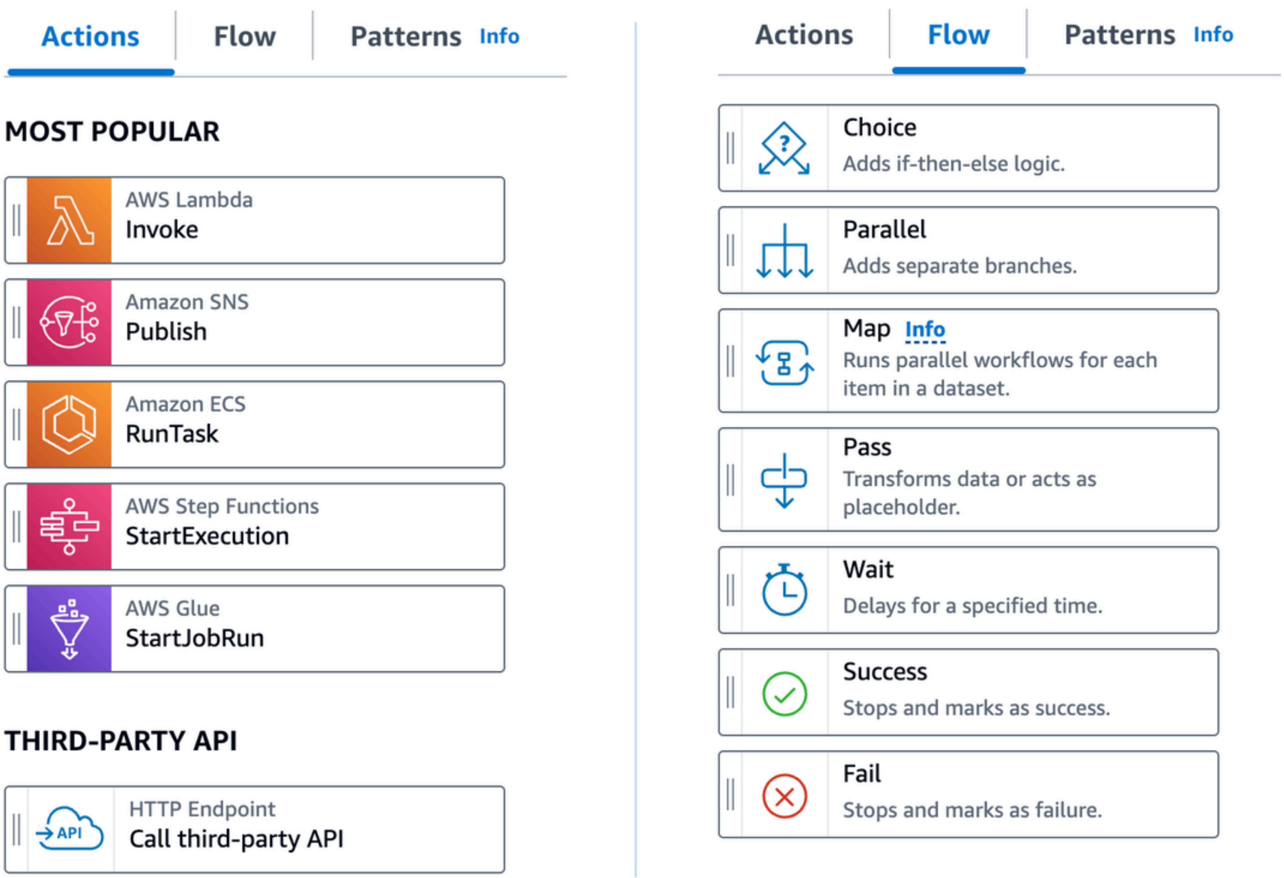
States take input from the invocation or a previous state. States can filter the input and then manipulate the output that is sent to the next state.

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

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

Individual states can make decisions based on their input, perform actions from those inputs, and pass output to other states. In AWS Step Functions, you define your workflows in the Amazon States Language (ASL). The Step Functions console provides a graphical representation of your state machine to help visualize your application's logic.

The following screenshot shows some of the most popular **Actions** and the seven **Flow** states from Workflow Studio:



States share many common features:

- A `Type` field indicating what type of state it is.
- An optional `Comment` field to hold a human-readable comment about, or description of, the state.
- Each state (except `Succeed` or `Fail` states) requires a `Next` field that specifies the next state in the workflow. `Choice` states can actually have more than one `Next` within each Choice Rule. Alternatively, a state can become a terminal state by setting the `End` field to true.

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

To access log information for workflows

- After you have created and run 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.

- After you have created and Express Workflow executions and if logging is enabled, you can see execution history in the Step Functions console or Amazon CloudWatch Logs.

For information about viewing and debugging executions, see Viewing workflow runs and the section called “Logging in CloudWatch Logs”.

Reference list of workflow states

States are separated in Workflow Studio into **Actions**, also known as **Task states**, and seven **Flow states**. Using **Task states**, or actions in Workflow Studio, you can call third party services, invoke functions, and use hundreds of AWS service endpoints. With **Flow states**, you can direct and control your workflow. All states take input from the previous state, and many provide input filtering, and filtering/transformation for output that is passed to the next state in your workflow.

- Task workflow state: Add a single unit of work to be performed by your state machine.
- Choice workflow state: Add a choice between branches of execution to your workflow.
- Parallel workflow state: Add parallel branches of execution to your workflow.
- Map workflow state: 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 workflow state: Pass state input through to the output. Optionally, filter, transform, and add fixed data into the output.
- Wait workflow state: Pause your workflow for a certain amount of time or until a specified time or date.
- Succeed workflow state: Stops your workflow with a success.
- Fail workflow state: Stops your workflow with a failure.

Task workflow state

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

A `Task` state (`"Type": "Task"`) represents a single unit of work performed by a state machine. A task performs work by using an activity or an AWS Lambda function, by integrating with other supported AWS services, or by invoking a HTTPS API, such as Stripe.

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, Lambda function, or the HTTPS API endpoint.

Invoke a function with JSONata Arguments

The following Task state definition (JSONata) invokes a Lambda function named *`priceWatcher`*.

Note the use of JSONata expressions to query input data to use in Arguments and the task result in the assign field.

```
"Get Current Price": {
  "Type": "Task",
  "QueryLanguage" : "JSONata",
  "Resource": "arn:aws:states:::lambda:invoke",
  "Next": "Check Price",
  "Arguments": {
    "Payload": {
    "product": "{% $states.context.Execution.Input.product %}"
    },
    "FunctionName": "arn:aws:lambda:<region>:account-id:function:priceWatcher:$LATEST"
  },
  "Assign": {
    "currentPrice": "{% $states.result.Payload.current_price %}"
  }
}
```

Invoke a function with JSONPath Parameters

The following Task state definition (JSONPath) invokes a Lambda function named *`HelloFunction`*.

```
"Lambda Invoke": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "Parameters": {
    "Payload.$": "$",
    "FunctionName": "arn:aws:lambda:region:account-id:function:HelloFunction:$LATEST"
```

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

Task types

Step Functions supports the following task types that you can specify in a Task state definition:

- Activity
- Lambda functions
- A supported AWS service
- An HTTP Task

You specify a task type by providing its ARN in the `Resource` field of a Task state definition. The following example shows the syntax of the `Resource` field. All Task types except the one that invokes an HTTPS API, use the following syntax. For information about syntax of the HTTP Task, see Call HTTPS APIs in Step Functions workflows.

In your Task state definition, replace the italicized text in the following syntax with the AWS resource-specific information.

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

The following list explains the individual components in this syntax:

- `partition` is the AWS Step Functions partition to use, most commonly `aws`.
- `service` indicates the AWS service used to execute the task, and can be one of the following values:
  - `states` for an activity.
  - `lambda` for a Lambda function. If you integrate with other AWS services, for example, Amazon SNS or Amazon DynamoDB, use `sns` or `dynamodb`.
- `region` is the AWS Region code in which the Step Functions activity or state machine type, Lambda function, or any other AWS resource has been created.
- `account` is the AWS account ID in which you've defined the resource.
- `task_type` is the type of task to run. It can be one of the following values:
  - `activity` – An activity.

  - `function` – A Lambda function.
  - *`servicename`* – The name of a supported connected service (see Integrating services with Step Functions).
- `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 or regions. For example, `aws-cn` can't invoke tasks in the `aws` partition, and the other way around.

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
```

> **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.

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.

The form of your Lambda function `Resource` field varies based on the type of integration.

For a standard AWS SDK integration with a Lambda function, the `Resource` field will contain the following value:

```
"arn:aws:states:::aws-sdk:lambda:invoke"
```

We **recommend** using the optimized integration for your Lambda functions, using the following value for the `Resource` field:

```
"arn:aws:states:::lambda:invoke"
```

The following `Task` state definition shows an example of an optimized integration with a Lambda function named *`HelloWorld`* using JSONata.

```
"Optimized call to Lambda function (JSONata)": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Output": "{% $states.result.Payload %}",
      "Arguments": {
        "FunctionName": "arn:aws:lambda:region:account-id:function:HelloWorld:$LATEST",
        "Payload": {
          "key": "{% $states.input.myKey %}"
        }
      },
  "Next": "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-id: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 Integrating services.

For example:

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

Task state fields

In addition to the common state fields, `Task` states have the following fields.

`Resource` (Required)

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

`Arguments` (Optional, JSONata only)

Used to pass information to the API actions of connected resources. Values can include JSONata expressions. For more information, see Transforming data with JSONata in Step Functions.

`Output` (Optional, JSONata only)

Used to specify and transform output from the state. When specified, the value overrides the state output default.

The output field accepts any JSON value (object, array, string, number, boolean, null). Any string value, including those inside objects or arrays, will be evaluated as JSONata if surrounded by {% %} characters.

Output also accepts a JSONata expression directly, for example: "Output": "{% jsonata expression %}"

For more information, see Input and Output Processing.

`Parameters` (Optional, JSONPath only)

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 Passing parameters to a service API in Step Functions.

`Credentials` (Optional)

Specifies a target role the state machine's execution role must assume before invoking the specified `Resource`. Alternatively, you can also specify a JSONPath value or an intrinsic function that resolves to an IAM role ARN at runtime based on the execution input. If you specify a JSONPath value, you must prefix it with the `$.` notation.

For examples of using this field in the `Task` state, see Task state's Credentials field examples. For an example of using this field to access a cross-account AWS resource from your state machine, see Accessing cross-account AWS resources in Step Functions.

> **Note**
>
> This field is supported by the Task types that use Lambda functions and a supported AWS service.

`ResultPath` (Optional, JSONPath only)

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.

`ResultSelector` (Optional, JSONPath only)

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

`Retry` (Optional)

An array of objects, called Retriers, that define a retry policy if the state encounters runtime errors. For more information, see State machine examples using Retry and using Catch.

`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.

`TimeoutSeconds` (Optional)

Specifies the maximum time an activity or a task can run before it times out with the States.Timeout error and fails. The timeout value must be positive, non-zero integer. The default value is 99999999.

The timeout count begins when the start event is executed, such as when `TaskStarted`, `ActivityStarted`, or `LambdaFunctionStarted` events are logged in the execution event history. For Activities, the count begins when `GetActivityTask` receives a token and `ActivityStarted` is logged in the execution event history.

When a task starts, Step Functions waits for a success or failure response from the task or activity worker within the specified `TimeoutSeconds` duration. If the task or activity worker fails to respond within this time, Step Functions marks the workflow execution as failed.

> **Note**
>
> HTTP task timeout has a maximum of 60 seconds, even if `TimeoutSeconds` exceeds that limit. See the section called “Quotas related to HTTP Task”

`TimeoutSecondsPath` (Optional, JSONPath only)

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.

> **Note**
>
> A `Task` state cannot include both `TimeoutSeconds` and `TimeoutSecondsPath`. HTTP task timeout has a maximum of 60 seconds, even if the `TimeoutSecondsPath` value exceeds that limit.

`HeartbeatSeconds` (Optional)

Determines the frequency of heartbeat signals an activity worker sends during the execution of a task. Heartbeats indicate that a task is still running and it needs more time to complete. Heartbeats prevent an activity or task from timing out within the `TimeoutSeconds` duration.

`HeartbeatSeconds` must be a positive, non-zero integer value less than the `TimeoutSeconds` field value. The default value is 99999999. If more time than the specified seconds elapses between heartbeats from the task, the Task state fails with a States.Timeout error.

For Activities, the count begins when `GetActivityTask` receives a token and `ActivityStarted` is logged in the execution event history.

`HeartbeatSecondsPath` (Optional, JSONPath only)

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 `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 definition examples

The following examples show how you can specify the Task state definition based on your requirement.

- Specifying Task state timeouts and heartbeat intervals
  - Static timeout and heartbeat notification example
  - Dynamic task timeout and heartbeat notification example
- Using Credentials field
  - Specifying hard-coded IAM role ARN

- [Specifying JSONPath as IAM role ARN]
- [Specifying an intrinsic function as IAM role ARN]

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",
  "Resource": "arn:aws:states:region:123456789012:activity:HelloWorld",
  "TimeoutSeconds": 300,
  "HeartbeatSeconds": 60,
  "Next": "NextState"
}
```

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, the task is marked as `failed`.

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

Task state's Credentials field examples

Specifying hard-coded IAM role ARN

The following example specifies a target IAM role that a state machine's execution role must assume to access a cross-account Lambda function named Echo. In this example, the target role ARN is specified as a hard-coded value.

```
{
  "StartAt": "Cross-account call",
  "States": {
    "Cross-account call": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Credentials": {
        "RoleArn": "arn:aws:iam::111122223333:role/LambdaRole"
      },
      "Parameters": {
        "FunctionName": "arn:aws:lambda:us-east-2:111122223333:function:Echo"
      },
      "End": true
    }
  }
}
```

Specifying JSONPath as IAM role ARN

The following example specifies a JSONPath value, which will resolve to an IAM role ARN at runtime.

```
{
  "StartAt": "Lambda",
  "States": {
    "Lambda": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Credentials": {
        "RoleArn.$": "$.roleArn"
      },
      ...
    }
  }
```

```
}
```

Specifying an intrinsic function as IAM role ARN

The following example uses the `States.Format` intrinsic function, which resolves to an IAM role ARN at runtime.

```
{
  "StartAt": "Lambda",
  "States": {
    "Lambda": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Credentials": {
        "RoleArn.$": "States.Format('arn:aws:iam::{}:role/ROLENAME', $.accountId)"
      },
      ...
    }
  }
}
```

Choice workflow state

> **Managing state and transforming data**
>
> Learn about Passing data between states with variables and Transforming data with JSONata.

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

In addition to most of the common state fields, `Choice` states contains the following additional fields.

`Choices` (Required)

An array of Choice Rules that determines which state the state machine transitions to next.

When a `Choice` state is run, it evaluates each Choice Rule to true or false. Based on the result of this evaluation, Step Functions transitions to the next state in the workflow.

You must define at least one rule in the `Choice` state.

`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 Rules (JSONata)

A `Choice` state must have a `Choices` field whose value is a non-empty array of Choice Rules, which contain the following fields when using JSONata:

- **`Condition` field** – a JSONata expression that evaluates to true/false.
- **`Next` field** – a value that must match a state name in the state machine.

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

```
{
  "Condition": "{% $foo = 1 %}",
  "Next": "NumericMatchState"
}
```

The following example checks whether the `type`variable is equal to `local`.

```
{
  "Condition": "{% $type = 'local' %}",
  "Next": "StringMatchState"
}
```

The following example checks whether the string is greater than `MyStringABC`.

```
{
  "Condition": "{% $foo > 'MyStringABC' %}",
  "Next": "StringGreaterMatchState"
```

```
}
```

The following example checks whether the string is not null.

```
{
 "Condition" : "{% $possiblyNullValue != null and $possiblyNullValue = 42 %}",
 "Next": "NotNullAnd42"
}
```

Choice Rules (JSONPath)

A `Choice` state must have a `Choices` field whose value is a non-empty array of Choice Rules, which contain the following fields when using JSONPath:

- 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. The values of `Variable` and Path fields in a comparison must be valid Reference Paths.
- A **`Next` field** – The value of this field must match a state name in the state machine.

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

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

The following example checks whether the string is equal to `MyString`.

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

The following example checks whether the string is greater than `MyStringABC`.

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

The following example checks whether the string is null.

```
{
 "Variable": "$.possiblyNullValue",
 "IsNull": true
}
```

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` and `Or` operators must be non-empty arrays of Choice Rules that must not themselves contain `Next` fields. Likewise, the value of a `Not` operator must be a single Choice Rule that must not contain `Next` fields.

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

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.

Parallel workflow state

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

The `Parallel` state (`"Type": "Parallel"`) can be used to add separate branches of execution in your state machine.

In addition to the common state fields, `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.

`Parameters` (Optional, JSONPath only)

Used to pass information to the state machines defined in the `Branches` array.

`Arguments` (Optional, JSONata only)

Used to pass information to the API actions of connected resources. Values can include JSONata expressions. For more information, see Transforming data with JSONata in Step Functions.

`Output` (Optional, JSONata only)

Used to specify and transform output from the state. When specified, the value overrides the state output default.

The output field accepts any JSON value (object, array, string, number, boolean, null). Any string value, including those inside objects or arrays, will be evaluated as JSONata if surrounded by {% %} characters.

Output also accepts a JSONata expression directly, for example: "Output": "{% jsonata expression %}"

For more information, see the section called “Transforming data with JSONata”.

`Assign` (Optional)

Used to store variables. The `Assign` field accepts a JSON object with key/value pairs that define variable names and their assigned values. Any string value, including those inside objects or arrays, will be evaluated as JSONata when surrounded by `{% %}` characters

For more information, see the section called “Passing data with variables”.

`ResultPath` (Optional, JSONPath only)

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.

`ResultSelector` (Optional, JSONPath only)

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

`Retry` (Optional)

An array of objects, called Retriers, that define a retry policy in case the state encounters runtime errors. For more information, see State machine examples using Retry and using Catch.

`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.

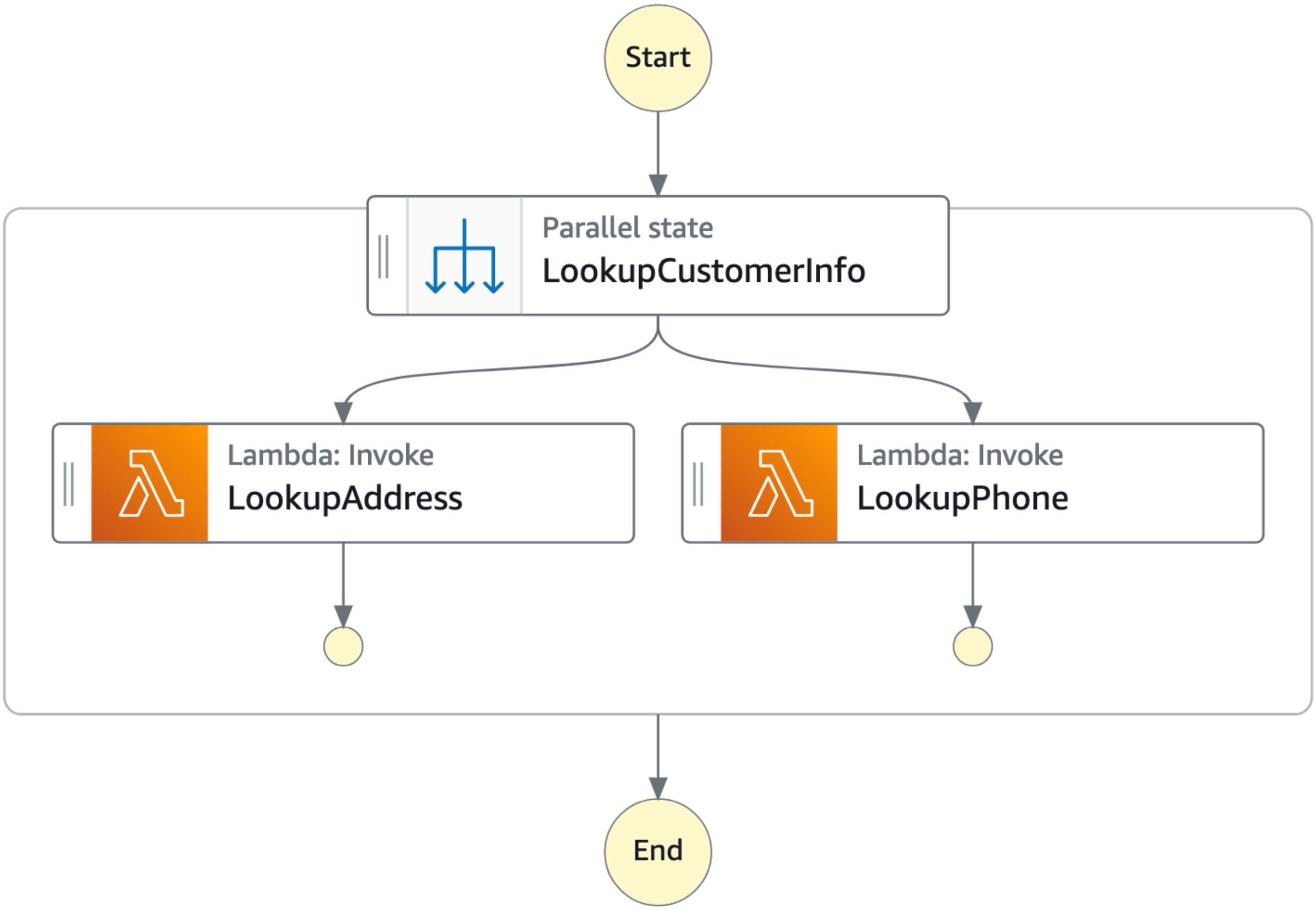
A `Parallel` state causes AWS Step Functions to execute each branch, starting with the state named in that branch's `StartAt` field, as concurrently as possible, and wait until all branches terminate (reach a terminal state) before processing the `Parallel` state's `Next` field.

Parallel State Example

```
{
```

```
  "Comment": "Parallel Example.",
  "StartAt": "LookupCustomerInfo",
  "States": {
    "LookupCustomerInfo": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
         "StartAt": "LookupAddress",
         "States": {
           "LookupAddress": {
             "Type": "Task",
             "Resource": "arn:aws:lambda:region:account-id:function:AddressFinder",
             "End": true
           }
         }
       },
       {
         "StartAt": "LookupPhone",
         "States": {
           "LookupPhone": {
             "Type": "Task",
             "Resource": "arn:aws:lambda:region:account-id:function:PhoneFinder",
             "End": true
           }
         }
       }
      ]
    }
  }
}
```

In this example, the `LookupAddress` and `LookupPhone` branches are executed in parallel. Here is how the visual workflow looks in the Step Functions console.



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

Parallel State Input and Output Processing

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

```
{
  "Comment": "Parallel Example.",
  "StartAt": "FunWithMath",
```

```
  "States": {
    "FunWithMath": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "Add",
          "States": {
            "Add": {
              "Type": "Task",
              "Resource": "arn:aws:states:region:123456789012:activity:Add",
              "End": true
            }
          }
        },
        {
          "StartAt": "Subtract",
          "States": {
            "Subtract": {
              "Type": "Task",
              "Resource": "arn:aws:states:region:123456789012:activity:Subtract",
              "End": true
            }
          }
        }
      ]
    }
  }
}
```

If the `FunWithMath` state was given the array `[3, 2]` as input, then both the `Add` and `Subtract` states receive that array as input. The output of 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 ]
```

> **Tip**
>
> If the Parallel or Map state you use in your state machines returns an array of arrays, you can transform them into a flat array with the ResultSelector field. For more information, see Flattening an array of arrays.

Error Handling

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

> **Note**
>
> When a parallel state fails, invoked Lambda functions continue to run and activity workers processing a task token are not stopped.
>
> - To stop long-running activities, use heartbeats to detect if its branch has been stopped by Step Functions, and stop workers that are processing tasks. Calling `SendTaskHeartbeat`, `SendTaskSuccess`, or `SendTaskFailure` will throw an error if the state has failed. See Heartbeat Errors.
> - Running Lambda functions cannot be stopped. If you have implemented a fallback, use a `Wait` state so that cleanup work happens after the Lambda function has finished.

Map workflow state

Use the `Map` state to run a set of workflow steps for each item in a dataset. The `Map` state's iterations run in parallel, which makes it possible to process a dataset quickly. `Map` states can use a variety of input types, including a JSON array, a list of Amazon S3 objects, or a CSV file.

Step Functions provides two types of processing modes for using the `Map` state in your workflows: *Inline* mode and *Distributed* mode.

> **Tip**
>
> To deploy an example of a workflow that uses a `Map` state, see Processing arrays of data with Choice and Map in *The AWS Step Functions Workshop*.

Map state processing modes

Step Functions provides the following processing modes for the `Map` state depending on how you want to process the items in a dataset.

- **Inline** – Limited-concurrency mode. In this mode, each iteration of the `Map` state runs in the context of the workflow that contains the `Map` state. Step Functions adds the execution history of these iterations to the parent workflow's execution history. By default, `Map` states run in Inline mode.

  In this mode, the `Map` state accepts only a JSON array as input. Also, this mode supports up to 40 concurrent iterations.

  For more information, see Using Map state in Inline mode in Step Functions workflows.
- **Distributed** – High-concurrency mode. In this mode, the `Map` state runs each iteration as a child workflow execution, which enables high concurrency of up to 10,000 parallel child workflow executions. Each child workflow execution has its own, separate execution history from that of the parent workflow.

  In this mode, the `Map` state can accept either a JSON array or an Amazon S3 data source, such as a CSV file, as its input.

  For more information, see Distributed mode.

The mode you should use depends on how you want to process the items in a dataset. Use the `Map` state in Inline mode if your workflow's execution history won't exceed 25,000 entries, or if you don't require more than 40 concurrent iterations.

Use the `Map` state in Distributed mode when you need to orchestrate large-scale parallel workloads that meet any combination of the following conditions:

- The size of your dataset exceeds 256 KiB.

- The workflow's execution event history would exceed 25,000 entries.
- You need a concurrency of more than 40 concurrent iterations.

Inline mode and Distributed mode differences

The following table highlights the differences between the Inline and Distributed modes.

Inline mode	Distributed mode
Supported data sources	
Accepts a JSON array passed from a previous step in the workflow as input.	Accepts the following data sources as input: • JSON array passed from a previous step in the workflow • JSON file in an Amazon S3 bucket that contains an array • CSV file in an Amazon S3 bucket • Amazon S3 object list • Amazon S3 inventory
Map iterations	
In this mode, each iteration of the `Map` state runs in the context of the workflow that contains the `Map` state. Step Functions adds the execution history of these iterations to the parent workflow's execution history.	In this mode, the `Map` state runs each iteration as a child workflow execution, which enables high concurrency of up to 10,000 parallel child workflow executions. Each child workflow execution has its own, separate execution history from that of the parent workflow.
Maximum concurrency for parallel iterations	
Lets you run up to 40 iterations as concurren tly as possible.	Lets you run up to 10,000 parallel child workflow executions to process millions of data items at one time.
Input payload and event history sizes	

Inline mode	Distributed mode
Enforces a limit of 256 KiB on the input payload size and 25,000 entries in the execution event history.	Lets you overcome the payload size limitation because the `Map` state can read input directly from Amazon S3 data sources. In this mode, you can also overcome execution history limitations because the child workflow executions started by the `Map` state maintain their own, separate execution histories from the parent workflow's execution history.
Monitoring and observability	
You can review the workflow's execution history from the console or by invoking the `GetExecutionHistory` API action. You can also view the execution history through CloudWatch and X-Ray.	When you run a `Map` state in Distributed mode, Step Functions creates a Map Run resource. A Map Run refers to a set of child workflow executions that a *Distributed Map state* starts. You can view a Map Run in the Step Functions console. You can also invoke the `DescribeMapRun` API action. A Map Run also emits metrics to CloudWatch. For more information, see Viewing a Distributed Map Run execution in Step Functions.

Using Map state in Inline mode in Step Functions workflows

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

By default, `Map` states runs in **Inline** mode. In Inline mode, the Map state accepts only a JSON array as input. It receives this array from a previous step in the workflow. In this mode, each iteration of the `Map` state runs in the context of the workflow that contains the `Map` state. Step Functions adds the execution history of these iterations to the parent workflow's execution history.

In this mode, the `Map` state supports up to 40 concurrent iterations.

A `Map` state set to **Inline** is known as an *Inline Map state*. Use the `Map` state in Inline mode if your workflow's execution history won't exceed 25,000 entries, or if you don't require more than 40 concurrent iterations.

For an introduction to using the *Inline Map state*, see the tutorial Repeat actions with Inline Map.

Contents

Key concepts in this topic

Inline mode

A limited-concurrency mode of the `Map` state. In this mode, each iteration of the `Map` state runs in the context of the workflow that contains the `Map` state. Step Functions adds the execution history of these iterations to the parent workflow's execution history. `Map` states run in the Inline mode by default.

This mode accepts only a JSON array as input and supports up to 40 concurrent iterations.

Inline Map state

A `Map` state set to the **Inline** mode.

Map workflow

The set of steps that the `Map` state runs for each iteration.

Map state iteration

A repetition of the workflow defined inside of the `Map` state.

Inline Map state fields

To use the *Inline Map state* in your workflows, specify one or more of these fields. You specify these fields in addition to the common state fields.

Type (Required)

Sets the type of state, such as `Map`.

`ItemProcessor` (Required)

Contains the following JSON objects that specify the `Map` state processing mode and definition.

The definition contains the set of steps to repeat for processing each array item.

- `ProcessorConfig` – An optional JSON object that specifies the processing mode for the `Map` state. This object contains the `Mode` sub-field. This field defaults to `INLINE`, which uses the `Map` state in Inline mode.

  In this mode, the failure of any iteration causes the `Map` state to fail. All iterations stop when the `Map` state fails.
- `StartAt` – Specifies a string that indicates the first state in a workflow. This string is case-sensitive and must match the name of one of the state objects. This state runs first for each item in the dataset. Any execution input that you provide to the `Map` state passes to the `StartAt` state first.
- `States` – A JSON object containing a comma-delimited set of states. In this object, you define the Map workflow.

> **Note**
>
> - States within the `ItemProcessor` field can only transition to each other. No state outside the `ItemProcessor` field can transition to a state within it.
> - The `ItemProcessor` field replaces the now deprecated `Iterator` field. Although you can continue to include `Map` states that use the `Iterator` field, we highly recommend that you replace this field with `ItemProcessor`.
>
>   Step Functions Local doesn't currently support the `ItemProcessor` field. We recommend that you use the `Iterator` field with Step Functions Local.

`Items` (Optional, JSONata only)

A JSON array or a JSONata expression that must evaluate to an array.

`ItemsPath` (Optional, JSONPath only)

Specifies a reference path using the JsonPath syntax. This path selects the JSON node that contains the array of items inside the state input. For more information, see ItemsPath (Map, JSONPath only).

`ItemSelector` (Optional)

Overrides the values of the input array items before they're passed on to each `Map` state iteration.

In this field, you specify a valid JSON that contains a collection of key-value pairs. These pairs can contain any of the following:

- Static values you define in your state machine definition.
- Values selected from the state input using a path.
- Values accessed from the context object.

For more information, see ItemSelector (Map).

The `ItemSelector` field replaces the now deprecated `Parameters` field. Although you can continue to include `Map` states that use the `Parameters` field, we highly recommend that you replace this field with `ItemSelector`.

`MaxConcurrency` (Optional)

Specifies an integer value that provides the upper bound on the number of `Map` state iterations that can run in parallel. For example, a `MaxConcurrency` value of 10 limits the `Map` state to 10 concurrent iterations running at one time.

In JSONata states, you can specify a JSONata expression that evaluates to an integer.

> **Note**
>
> Concurrent iterations may be limited. When this occurs, some iterations won't begin until previous iterations are complete. The likelihood of this occurring increases when your input array has more than 40 items.
> To achieve a higher concurrency, consider Distributed mode.

The default value is `0`, which places no limit on concurrency. Step Functions invokes iterations as concurrently as possible.

A `MaxConcurrency` value of `1` invokes the `ItemProcessor` once for each array element. Items in the array are processed in the order of their appearance in the input. Step Functions doesn't start a new iteration until it completes the previous iteration.

`MaxConcurrencyPath` (Optional, JSONPath only)

If you want to provide a maximum concurrency value dynamically from the state input using a reference path, use `MaxConcurrencyPath`. When resolved, the reference path must select a field whose value is a non-negative integer.

> **Note**
>
> A `Map` state cannot include both `MaxConcurrency` and `MaxConcurrencyPath`.

`ResultPath` (Optional, JSONPath only)

Specifies where in the input to store the output of the `Map` state's iterations. The Map state then filters the input as specified by the `OutputPath` field, if specified. Then, it uses the filtered input as the state's output. For more information, see Input and Output Processing.

`ResultSelector` (Optional, JSONPath only)

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

> **Tip**
>
> If the Parallel or Map state you use in your state machines returns an array of arrays, you can transform them into a flat array with the ResultSelector field. For more information, see Flattening an array of arrays.

`Retry` (Optional)

An array of objects, called Retriers, that define a retry policy. States use a retry policy when they encounter runtime errors. For more information, see State machine examples using Retry and using Catch.

> **Note**
>
> If you define Retriers for the *Inline Map state*, the retry policy applies to all `Map` state iterations, instead of only failed iterations. For example, your `Map` state contains two successful iterations and one failed iteration. If you have defined the `Retry` field for the `Map` state, the retry policy applies to all three `Map` state iterations instead of only the failed iteration.

`Catch` (Optional)

An array of objects, called Catchers, that define a fallback state. States run a catcher if they encounter runtime errors and either don't have a retry policy, or their retry policy is exhausted. For more information, see Fallback States.

`Output` (Optional, JSONata only)

Used to specify and transform output from the state. When specified, the value overrides the state output default.

The output field accepts any JSON value (object, array, string, number, boolean, null). Any string value, including those inside objects or arrays, will be evaluated as JSONata if surrounded by {% %} characters.

Output also accepts a JSONata expression directly, for example: "Output": "{% jsonata expression %}"

For more information, see the section called “Transforming data with JSONata”.

`Assign` (Optional)

Used to store variables. The `Assign` field accepts a JSON object with key/value pairs that define variable names and their assigned values. Any string value, including those inside objects or arrays, will be evaluated as JSONata when surrounded by `{% %}` characters

For more information, see the section called “Passing data with variables”.

Deprecated fields

> **Note**
>
> Although you can continue to include `Map` states that use the following fields, we highly recommend that you replace `Iterator` with [`ItemProcessor`] and `Parameters` with [`ItemSelector`].

`Iterator`

Specifies a JSON object that defines a set of steps that process each element of the array.

`Parameters`

Specifies a collection of key-value pairs, where the values can contain any of the following:

- Static values that you define in your state machine definition.
- Values selected from the input using a path.

Inline Map state example (JSONPath)

Consider the following input data for a `Map` state running in **Inline** mode.

```
{
  "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 invokes an AWS Lambda function named `ship-val` once for each item of the array in the `shipped` field.

```
"Validate All": {
```

```
    "Type": "Map",
    "InputPath": "$.detail",
    "ItemProcessor": {
        "ProcessorConfig": {
            "Mode": "INLINE"
        },
        "StartAt": "Validate",
        "States": {
            "Validate": {
                "Type": "Task",
                "Resource": "arn:aws:states:::lambda:invoke",
                "OutputPath": "$.Payload",
                "Parameters": {
                    "FunctionName": "arn:aws:lambda:us-east-2:account-id:function:ship-
val:$LATEST"
                },
                "End": true
            }
        }
    },
    "End": true,
    "ResultPath": "$.detail.shipped",
    "ItemsPath": "$.shipped"
}
```

Each iteration of the `Map` state sends an item in the array, selected with the [`ItemsPath`] field, as input to the `ship-val` Lambda function. The following values are an example of input the `Map` state sends to an invocation of the Lambda function:

```
{
  "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, this array contains the output of the `ship-val` Lambda function.

Inline Map state example with `ItemSelector`

Suppose that the `ship-val` Lambda function in the previous example also needs information about the shipment's courier. This information is in addition to 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 ItemSelector field in the following example:

```
"Validate-All": {
  "Type": "Map",
  "InputPath": "$.detail",
  "ItemsPath": "$.shipped",
  "MaxConcurrency": 0,
  "ResultPath": "$.detail.shipped",
  "ItemSelector": {
    "parcel.$": "$$.Map.Item.Value",
    "courier.$": "$.delivery-partner"
  },
  "ItemProcessor": {
    "StartAt": "Validate",
    "States": {
      "Validate": {
        "Type": "Task",
 "Resource": "arn:aws:lambda:region:account-id:function:ship-val",
        "End": true
      }
    }
  },
  "End": true
}
```

The ItemSelector block replaces the input to the iterations with a JSON node. This node contains both the current item data from the [Context object](#) and the courier information from the Map state input's delivery-partner field. The following is an example of input to a single iteration. The Map state passes this input to an invocation of the ship-val Lambda function.

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

In the previous *Inline Map state* example, the `ResultPath` field produces output in the same format as the input. However, it overwrites the `detail.shipped` field with an array in which each element is the output of each iteration's `ship-val` Lambda invocation.

For more information about using the *Inline Map state* state and its fields, see the following.

- Repeat actions with Inline Map
- Processing input and output in Step Functions
- ItemsPath (Map, JSONPath only)
- Context object data for Map states

Inline Map state input and output processing

For a given `Map` state, `InputPath` selects a subset of the state's input.

The input of a `Map` state must include a JSON array. The `Map` state runs the `ItemProcessor` section once for each item in the array. If you specify the `ItemsPath` field, the `Map` state selects where in the input to find the array to iterate over. If not specified, the value of `ItemsPath` is `$`, and the `ItemProcessor` section expects that the array is the only input. If you specify the `ItemsPath` field, its value must be a Reference Path. The `Map` state applies this path to the effective input after it applies the `InputPath`. The `ItemsPath` 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. You can override this value with the `ItemSelector (Map)` 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 about Inline Map state inputs and outputs, see the following:

- Repeat actions with Inline Map
- Inline Map state example with `ItemSelector`
- Processing input and output in Step Functions
- Context object data for Map states
- Process data from a queue with a Map state in Step Functions

Using Map state in Distributed mode for large-scale parallel workloads in Step Functions

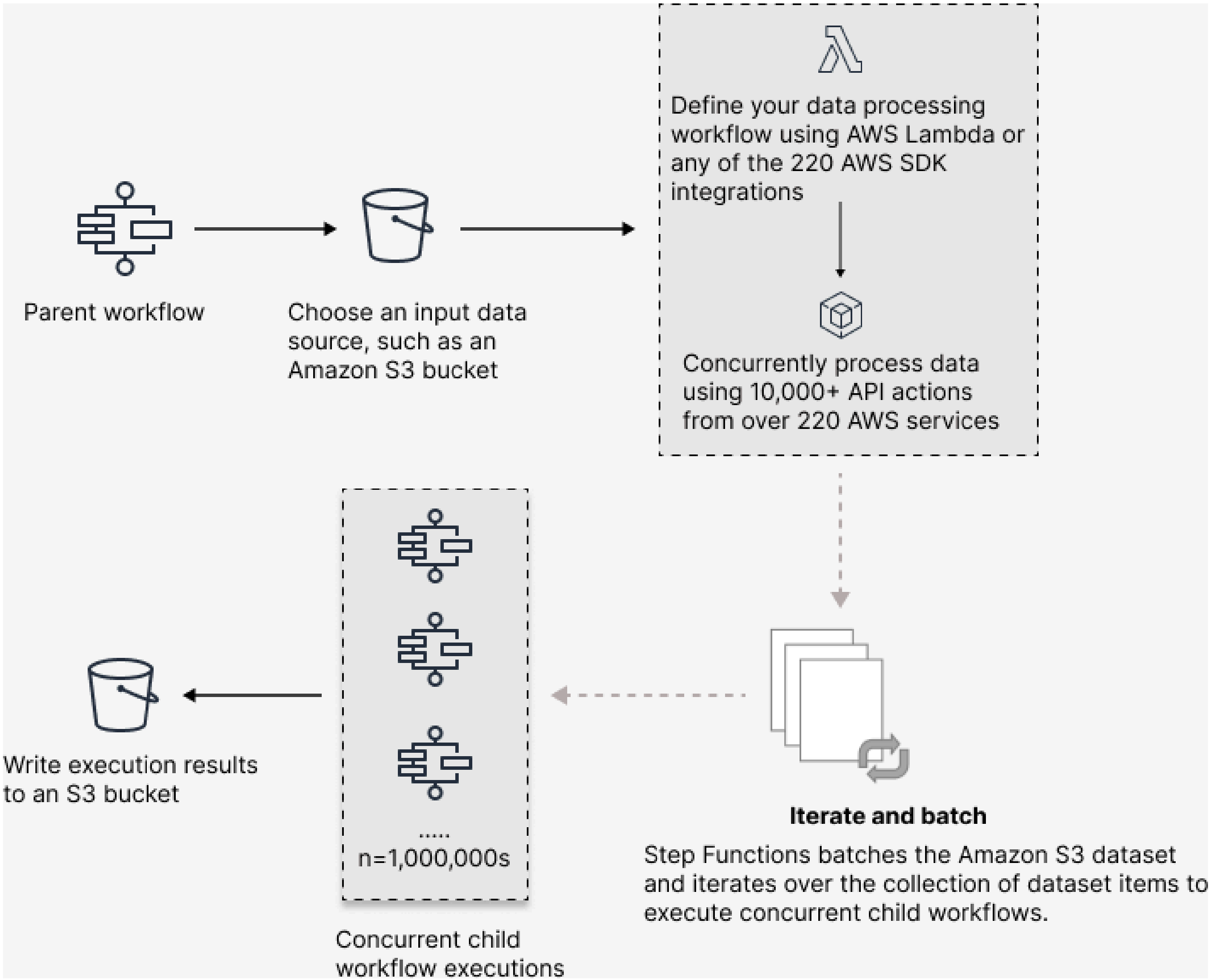
Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

With Step Functions, you can orchestrate large-scale parallel workloads to perform tasks, such as on-demand processing of semi-structured data. These parallel workloads let you concurrently process large-scale data sources stored in Amazon S3. For example, you might process a single JSON or CSV file that contains large amounts of data. Or you might process a large set of Amazon S3 objects.

To set up a large-scale parallel workload in your workflows, include a `Map` state in Distributed mode. The *Map state* processes items in a dataset concurrently. A `Map` state set to **Distributed** is known as a *Distributed Map state*. In *Distributed mode*, the `Map` state allows high-concurrency processing. In Distributed mode, the `Map` state processes the items in the dataset in iterations called *child workflow executions*. You can specify the number of child workflow executions that can run in parallel. Each child workflow execution has its own, separate execution history from that of the parent workflow. If you don't specify, Step Functions runs 10,000 parallel child workflow executions in parallel.

The following illustration explains how you can set up large-scale parallel workloads in your workflows.



Learn in a workshop

Learn how serverless technologies such as Step Functions and Lambda can simplify management and scaling, offload undifferentiated tasks, and address the challenges of large-scale distributed data processing. Along the way, you will work with distributed map for high concurrency processing. The workshop also presents best practices for optimizing your workflows, and practical use cases for claims processing, vulnerability scanning, and Monte Carlo simulation.

Workshop: [Large-scale Data Processing with Step Functions]

In this topic

Key terms

Distributed mode

A processing mode of the Map state. In this mode, each iteration of the `Map` state runs as a child workflow execution that enables high concurrency. Each child workflow execution has its own execution history, which is separate from the parent workflow's execution history. This mode supports reading input from large-scale Amazon S3 data sources.

Distributed Map state

A Map state set to **Distributed** processing mode.

Map workflow

A set of steps that a `Map` state runs.

Parent workflow

A workflow that contains one or more Distributed Map states.

Child workflow execution

An iteration of the *Distributed Map state*. A child workflow execution has its own execution history, which is separate from the parent workflow's execution history.

Map Run

When you run a `Map` state in Distributed mode, Step Functions creates a Map Run resource. A Map Run refers to a set of child workflow executions that a *Distributed Map state* starts, and the runtime settings that control these executions. Step Functions assigns an Amazon Resource Name (ARN) to your Map Run. You can examine a Map Run in the Step Functions console. You can also invoke the `DescribeMapRun` API action.

Map Runs do **not** emit metrics to CloudWatch. However, child workflow executions of a Map Run **do** emit metrics to CloudWatch. These metrics will have a labelled State Machine ARN with the following format:

```
arn:partition:states:region:account:stateMachine:stateMachineName/MapRunLabel
or UUID
```

For more information, see Viewing Map Runs.

Distributed Map state definition example (JSONPath)

Use the `Map` state in Distributed mode when you need to orchestrate large-scale parallel workloads that meet any combination of the following conditions:

- The size of your dataset exceeds 256 KiB.
- The workflow's execution event history would exceed 25,000 entries.
- You need a concurrency of more than 40 concurrent iterations.

The following *Distributed Map state* definition example specifies the dataset as a CSV file stored in an Amazon S3 bucket. It also specifies a Lambda function that processes the data in each row of the CSV file. Because this example uses a CSV file, it also specifies the location of the CSV column headers. To view the complete state machine definition of this example, see the tutorial Copying large-scale CSV data using Distributed Map.

```
{
  "Map": {
    "Type": "Map",
    "ItemReader": {
      "ReaderConfig": {
        "InputType": "CSV",
        "CSVHeaderLocation": "FIRST_ROW"
      },
      "Resource": "arn:aws:states:::s3:getObject",
      "Parameters": {
        "Bucket": "amzn-s3-demo-bucket",
        "Key": "csv-dataset/ratings.csv"
      }
    },
    "ItemProcessor": {
```

```
      "ProcessorConfig": {
        "Mode": "DISTRIBUTED",
        "ExecutionType": "EXPRESS"
      },
      "StartAt": "LambdaTask",
      "States": {
        "LambdaTask": {
          "Type": "Task",
          "Resource": "arn:aws:states:::lambda:invoke",
          "OutputPath": "$.Payload",
          "Parameters": {
            "Payload.$": "$",
            "FunctionName": "arn:aws:lambda:us-east-2:account-
id:function:processCSVData"
          },
          "End": true
        }
      }
    },
    "Label": "Map",
    "End": true,
    "ResultWriter": {
      "Resource": "arn:aws:states:::s3:putObject",
      "Parameters": {
        "Bucket": "amzn-s3-demo-destination-bucket",
        "Prefix": "csvProcessJobs"
      }
    }
  }
}
```

Permissions to run Distributed Map

When you include a *Distributed Map state* in your workflows, Step Functions needs appropriate permissions to allow the state machine role to invoke the `StartExecution` API action for the *Distributed Map state*.

The following IAM policy example grants the least privileges required to your state machine role for running the *Distributed Map state*.

Note

Make sure that you replace *stateMachineName* with the name of the state machine in which you're using the *Distributed Map state*. For example, `arn:aws:states:region:account-id:stateMachine:mystateMachine`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": [
        "arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachineName"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeExecution"
      ],
      "Resource": "arn:aws:states:us-
east-1:123456789012:execution:myStateMachineName:*"
    }
  ]
}
```

In addition, you need to make sure that you have the least privileges necessary to access the AWS resources used in the *Distributed Map state*, such as Amazon S3 buckets. For information, see IAM policies for using Distributed Map states.

Distributed Map state fields

To use the *Distributed Map state* in your workflows, specify one or more of these fields. You specify these fields in addition to the common state fields.

Type (Required)

Sets the type of state, such as `Map`.

`ItemProcessor` (Required)

Contains the following JSON objects that specify the `Map` state processing mode and definition.

- `ProcessorConfig` – JSON object that specifies the mode for processing items, with the following sub-fields:
  - `Mode` – Set to **`DISTRIBUTED`** to use the `Map` state in Distributed mode.

    > **Warning**
    >
    > Distributed mode is supported in Standard workflows but not supported in Express workflows.

  - `ExecutionType` – Specifies the execution type for the Map workflow as either **STANDARD** or **EXPRESS**. You must provide this field if you specified `DISTRIBUTED` for the `Mode` sub-field. For more information about workflow types, see Choosing workflow type in Step Functions.
- `StartAt` – Specifies a string that indicates the first state in a workflow. This string is case-sensitive and must match the name of one of the state objects. This state runs first for each item in the dataset. Any execution input that you provide to the `Map` state passes to the `StartAt` state first.
- `States` – A JSON object containing a comma-delimited set of states. In this object, you define the Map workflow.

`ItemReader`

Specifies a dataset and its location. The `Map` state receives its input data from the specified dataset.

In Distributed mode, you can use either a JSON payload passed from a previous state or a large-scale Amazon S3 data source as the dataset. For more information, see ItemReader (Map).

`Items` (Optional, JSONata only)

A JSON array or a JSONata expression that must evaluate to an array.

`ItemsPath` (Optional, JSONPath only)

Specifies a reference path using the JsonPath syntax to select the JSON node that contains an array of items inside the state input.

In Distributed mode, you specify this field only when you use a JSON array from a previous step as your state input. For more information, see ItemsPath (Map, JSONPath only).

`ItemSelector` (Optional, JSONPath only)

Overrides the values of individual dataset items before they're passed on to each `Map` state iteration.

In this field, you specify a valid JSON input that contains a collection of key-value pairs. These pairs can either be static values that you define in your state machine definition, values selected from the state input using a path, or values accessed from the context object. For more information, see ItemSelector (Map).

`ItemBatcher` (Optional)

Specifies to process the dataset items in batches. Each child workflow execution then receives a batch of these items as input. For more information, see ItemBatcher (Map).

`MaxConcurrency` (Optional)

Specifies the number of child workflow executions that can run in parallel. The interpreter only allows up to the specified number of parallel child workflow executions. If you don't specify a concurrency value or set it to zero, Step Functions doesn't limit concurrency and runs 10,000 parallel child workflow executions. In JSONata states, you can specify a JSONata expression that evaluates to an integer.

> **Note**
>
> While you can specify a higher concurrency limit for parallel child workflow executions, we recommend that you don't exceed the capacity of a downstream AWS service, such as AWS Lambda.

`MaxConcurrencyPath` (Optional, JSONPath only)

If you want to provide a maximum concurrency value dynamically from the state input using a reference path, use `MaxConcurrencyPath`. When resolved, the reference path must select a field whose value is a non-negative integer.

> **Note**
>
> A `Map` state cannot include both `MaxConcurrency` and `MaxConcurrencyPath`.

`ToleratedFailurePercentage` (Optional)

Defines the percentage of failed items to tolerate in a Map Run. The Map Run automatically fails if it exceeds this percentage. Step Functions calculates the percentage of failed items as the result of the total number of failed or timed out items divided by the total number of items. You must specify a value between zero and 100. For more information, see Setting failure thresholds for Distributed Map states in Step Functions.

In JSONata states, you can specify a JSONata expression that evaluates to an integer.

`ToleratedFailurePercentagePath` (Optional, JSONPath only)

If you want to provide a tolerated failure percentage value dynamically from the state input using a reference path, use `ToleratedFailurePercentagePath`. When resolved, the reference path must select a field whose value is between zero and 100.

`ToleratedFailureCount` (Optional)

Defines the number of failed items to tolerate in a Map Run. The Map Run automatically fails if it exceeds this number. For more information, see Setting failure thresholds for Distributed Map states in Step Functions.

In JSONata states, you can specify a JSONata expression that evaluates to an integer.

`ToleratedFailureCountPath` (Optional, JSONPath only)

If you want to provide a tolerated failure count value dynamically from the state input using a reference path, use `ToleratedFailureCountPath`. When resolved, the reference path must select a field whose value is a non-negative integer.

`Label` (Optional)

A string that uniquely identifies a `Map` state. For each Map Run, Step Functions adds the label to the Map Run ARN. The following is an example of a Map Run ARN with a custom label named `demoLabel`:

```
arn:aws:states:region:account-id:mapRun:demoWorkflow/
demoLabel:3c39a231-69bb-3d89-8607-9e124eddbb0b
```

If you don't specify a label, Step Functions automatically generates a unique label.

> **Note**
>
> Labels can't exceed 40 characters in length, must be unique within a state machine definition, and can't contain any of the following characters:
>
> - Whitespace
> - Wildcard characters (`? *`)
> - Bracket characters (`< > { } [ ]`)
> - Special characters (`: ; , \ | ^ ~ $ # % & ` "`)
> - Control characters (`\\u0000` - `\\u001f` or `\\u007f` - `\\u009f`).
>
> Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

`ResultWriter` (Optional)

Specifies the Amazon S3 location where Step Functions writes all child workflow execution results.

Step Functions consolidates all child workflow execution data, such as execution input and output, ARN, and execution status. It then exports executions with the same status to their respective files in the specified Amazon S3 location. For more information, see ResultWriter (Map).

If you don't export the `Map` state results, it returns an array of all the child workflow execution results. For example:

```
[1, 2, 3, 4, 5]
```

`ResultPath` (Optional, JSONPath only)

Specifies where in the input to place the output of the iterations. The input is then filtered as specified by the `OutputPath` field if present, before it is passed as the state's output. For more information, see Input and Output Processing.

`ResultSelector` (Optional)

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

> **Tip**
>
> If the Parallel or Map state you use in your state machines returns an array of arrays, you can transform them into a flat array with the ResultSelector field. For more information, see Flattening an array of arrays.

`Retry` (Optional)

An array of objects, called Retriers, that define a retry policy. An execution uses the retry policy if the state encounters runtime errors. For more information, see State machine examples using Retry and using Catch.

> **Note**
>
> If you define Retriers for the *Distributed Map state*, the retry policy applies to all of the child workflow executions the `Map` state started. For example, imagine your `Map` state started three child workflow executions, out of which one fails. When the failure occurs, the execution uses the `Retry` field, if defined, for the `Map` state. The retry policy applies to all the child workflow executions and not just the failed execution. If one or more child workflow executions fails, the Map Run fails.
> When you retry a `Map` state, it creates a new Map Run.

`Catch` (Optional)

An array of objects, called Catchers, that define a fallback state. Step Functions uses the Catchers defined in `Catch` if the state encounters runtime errors. When an error occurs, the execution first uses any retriers defined in `Retry`. If the retry policy isn't defined or is exhausted, the execution uses its Catchers, if defined. For more information, see Fallback States.

`Output` (Optional, JSONata only)

Used to specify and transform output from the state. When specified, the value overrides the state output default.

The output field accepts any JSON value (object, array, string, number, boolean, null). Any string value, including those inside objects or arrays, will be evaluated as JSONata if surrounded by {% %} characters.

Output also accepts a JSONata expression directly, for example: "Output": "{% jsonata expression %}"

For more information, see the section called "Transforming data with JSONata".

`Assign` (Optional)

Used to store variables. The `Assign` field accepts a JSON object with key/value pairs that define variable names and their assigned values. Any string value, including those inside objects or arrays, will be evaluated as JSONata when surrounded by `{% %}` characters

For more information, see the section called "Passing data with variables".

Setting failure thresholds for Distributed Map states in Step Functions

When you orchestrate large-scale parallel workloads, you can also define a tolerated failure threshold. This value lets you specify the maximum number of, or percentage of, failed items as a failure threshold for a Map Run. Depending on which value you specify, your Map Run fails automatically if it exceeds the threshold. If you specify both values, the workflow fails when it exceeds either value.

Specifying a threshold helps you fail a specific number of items before the entire Map Run fails. Step Functions returns a `States.ExceedToleratedFailureThreshold` error when the Map Run fails because the specified threshold is exceeded.

> **Note**
>
> Step Functions may continue to run child workflows in a Map Run even after the tolerated failure threshold is exceeded, but before the Map Run fails.

To specify the threshold value in Workflow Studio, select **Set a tolerated failure threshold** in **Additional configuration** under the **Runtime settings** field.

Tolerated failure percentage

Defines the percentage of failed items to tolerate. Your Map Run fails if this value is exceeded. Step Functions calculates the percentage of failed items as the result of the total number of failed or timed out items divided by the total number of items. You must specify a value between zero and 100. The default percentage value is zero, which means that the workflow fails if any one of its child workflow executions fails or times out. If you specify the percentage as 100, the workflow won't fail even if all child workflow executions fail.

Alternatively, you can specify the percentage as a reference path to an existing key-value pair in your *Distributed Map state* input. This path must resolve to a positive integer between 0 and 100 at runtime. You specify the reference path in the `ToleratedFailurePercentagePath` sub-field.

For example, given the following input:

```
{
  "percentage": 15
}
```

You can specify the percentage using a reference path to that input as follows:

```
{
  ...
  "Map": {
    "Type": "Map",
    ...
    "ToleratedFailurePercentagePath": "$.percentage"
    ...
  }
}
```

Important

You can specify either `ToleratedFailurePercentage` or `ToleratedFailurePercentagePath`, but not both in your *Distributed Map state* definition.

Tolerated failure count

Defines the number of failed items to tolerate. Your Map Run fails if this value is exceeded.

Alternatively, you can specify the count as a reference path to an existing key-value pair in your *Distributed Map state* input. This path must resolve to a positive integer at runtime. You specify the reference path in the `ToleratedFailureCountPath` sub-field.

For example, given the following input:

```
{
  "count": 10
}
```

You can specify the number using a reference path to that input as follows:

```
{
  ...
  "Map": {
    "Type": "Map",
    ...
    "ToleratedFailureCountPath": "$.count"
    ...
  }
}
```

> **Important**
>
> You can specify either `ToleratedFailureCount` or `ToleratedFailureCountPath`, but not both in your *Distributed Map state* definition.

Learn more about distributed maps

To continue learning more about *Distributed Map state*, see the following resources:

- **Input and output processing**

  To configure the input that a *Distributed Map state* receives and the output that it generates, Step Functions provides the following fields:

  - ItemReader (Map)

    - ItemsPath (Map, JSONPath only)
    - ItemSelector (Map)
    - ItemBatcher (Map)
    - ResultWriter (Map)
    - How Step Functions parses input CSV files

  In addition to these fields, Step Functions also provides you the ability to define a tolerated failure threshold for Distributed Map. This value lets you specify the maximum number of, or percentage of, failed items as a failure threshold for a Map Run. For more information about configuring the tolerated failure threshold, see Setting failure thresholds for Distributed Map states in Step Functions.

- **Using Distributed Map state**

  Refer the following tutorials and sample projects to get started with using *Distributed Map state*.

    - Copy large-scale CSV using Distributed Map
    - Processing batch data with a Lambda function in Step Functions
    - Processing individual items with a Lambda function in Step Functions
    - Sample project: Process a CSV file with Distributed Map
    - Sample project: Process data in an Amazon S3 bucket with Distributed Map

- **Examine Distributed Map state execution**

  The Step Functions console provides a *Map Run Details* page, which displays all the information related to a *Distributed Map state* execution. For information about how to examine the information displayed on this page, see Viewing Map Runs.

Pass workflow state

> **Managing state and transforming data**
>
> Learn about Passing data between states with variables and Transforming data with JSONata.

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

You can also use a `Pass` state to transform JSON state input using filters, and then pass the transformed data to the next state in your workflows. For information about input transformation, see Manipulate parameters in Step Functions workflows.

In addition to the common state fields, `Pass` states allow the following fields.

`Assign` (Optional, JSONata only)

A collection of key-value pairs to assign data to variables. For more information, see the section called "Passing data with variables".

`Output` (Optional, JSONata only)

Used to specify and transform output from the state. When specified, the value overrides the state output default.

The output field accepts any JSON value (object, array, string, number, boolean, null). Any string value, including those inside objects or arrays, will be evaluated as JSONata if surrounded by {% %} characters.

Output also accepts a JSONata expression directly, for example: "Output": "{% jsonata expression %}"

For more information, see the section called "Transforming data with JSONata".

`Result` (Optional, JSONPath only)

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, JSONPath only)

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 *Processing input and output*.

`Parameters` (Optional, JSONPath only)

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 the section called "Manipulate parameters with paths".

Pass State Example (JSONPath)

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",
  "End": true
}
```

Suppose the input to this state is the following.

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

Then the output would be this.

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

Wait workflow state

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

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

In addition to the common state fields, `Wait` states have one of the following fields.

`Seconds`

A time, in seconds, to wait before beginning the state specified in the `Next` field. You must specify time as an integer value from 0 to 99999999. In JSONata states, you can alternatively specify a JSONata expression which must evaluate to an integer in the stated range.

`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, `2024-08-18T17:33:00Z`.

In JSONata states, you can specify a JSONata expression which results in a string that conforms to the previous requirements.

> **Note**
>
> Currently, if you specify the wait time as a timestamp, Step Functions considers the time value up to seconds and truncates milliseconds.

`SecondsPath` (Optional, JSONPath only)

A path in the states input data to an integer value that specifies the time to wait, in seconds, before proceeding to the next state.

`TimestampPath` (Optional, JSONPath only)

A path in the states input data to an absolute date and time (timestamp) to wait before proceeding to the next state.

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, 2024, at 1:59 AM UTC.

```
"wait_until" : {
  "Type": "Wait",
  "Timestamp": "2024-03-14T01:59:00Z",
  "Next": "NextState"
}
```

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

```
{
  "expirydate": "2024-03-14T01:59:00Z"
}
```

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

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

Succeed workflow state

A `Succeed` state (`"Type": "Succeed"`) either terminates a state machine successfully, ends a branch of a the section called "Parallel", or ends an iteration of a the section called "Map". The `Succeed` state is a useful target for `Choice` state branches that don't do anything except terminate the state machine.

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

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

`Output` (Optional, JSONata only)

In addition to the common state fields, `Succeed` states that use JSONata can include an Output field to specify and transform output from the state. When specified, the `Output` value overrides the state output default.

The output field accepts any JSON value (object, array, string, number, boolean, null). Any string value, including those inside objects or arrays, will be evaluated as JSONata if surrounded by `{% %}` characters.

Output also accepts a JSONata expression directly, for example:

```
"Output" : "{% jsonata expression %}"
```

For more information on JSONata, see the section called "Transforming data with JSONata".

Fail workflow state

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

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. In addition, the `Fail` state allows the following fields.

Cause (Optional)

A custom string that describes the cause of the error. You can specify this field for operational or diagnostic purposes.

In JSONata states, you can also specify a JSONata expression.

CausePath (Optional, JSONPath only)

If you want to provide a detailed description about the cause of the error dynamically from the state input using a reference path, use `CausePath`. When resolved, the reference path must select a field that contains a string value.

You can also specify `CausePath` using an intrinsic function that returns a string. These intrinsics are: States.Format, States.JsonToString, States.ArrayGetItem, States.Base64Encode, States.Base64Decode, States.Hash, and States.UUID.

Important

- You can specify either `Cause` or `CausePath`, but not both in your Fail state definition.
- As an information security best practice, we recommend that you remove any sensitive information or internal system details from the cause description.

Error (Optional)

An error name that you can provide to perform error handling using Retry or Catch fields. You can also provide an error name for operational or diagnostic purposes.

In JSONata states, you can also specify a JSONata expression.

ErrorPath (Optional, JSONPath only)

If you want to provide a name for the error dynamically from the state input using a reference path, use `ErrorPath`. When resolved, the reference path must select a field that contains a string value.

You can also specify `ErrorPath` using an intrinsic function that returns a string. These intrinsics are: States.Format, States.JsonToString, States.ArrayGetItem, States.Base64Encode, States.Base64Decode, States.Hash, and States.UUID.

Important

- You can specify either `Error` or `ErrorPath`, but not both in your Fail state definition.
- As an information security best practice, we recommend that you remove any sensitive information or internal system details from the error name.

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

Fail state definition examples

The following Fail state definition example specifies static `Error` and `Cause` field values.

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

The following Fail state definition example uses reference paths dynamically to resolve the `Error` and `Cause` field values.

```
"FailState": {
  "Type": "Fail",
  "CausePath": "$.Cause",
  "ErrorPath": "$.Error"
}
```

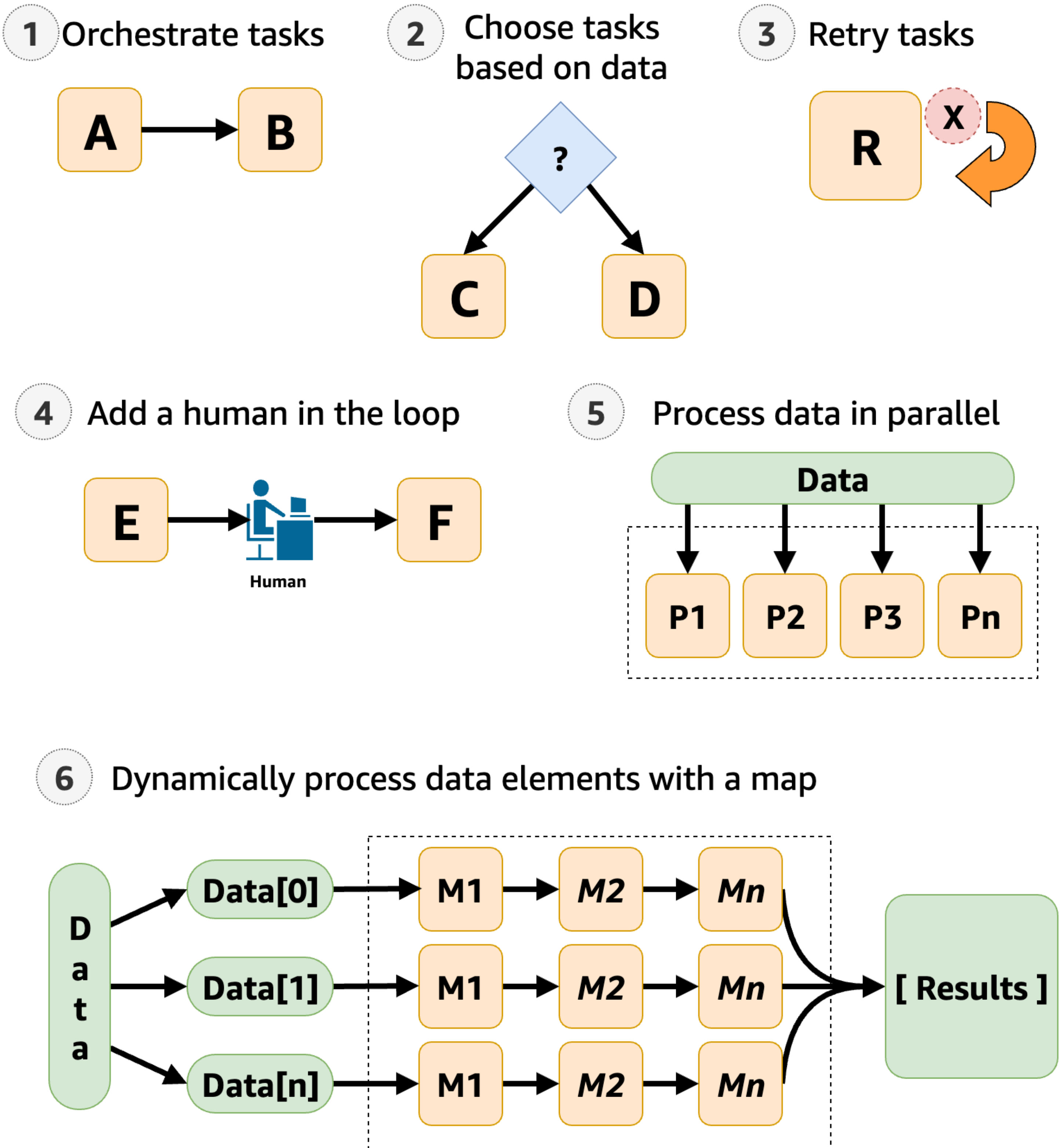
The following Fail state definition example uses the States.Format intrinsic function to specify the `Error` and `Cause` field values dynamically.

```
"FailState": {
  "Type": "Fail",
  "CausePath": "States.Format('This is a custom error message for {}, caused by {}.',
 $.Error, $.Cause)",
```

```
  "ErrorPath": "States.Format('{}', $.Error)"
}
```

Tutorials and workshops for learning Step Functions

Learn from this guide, workshops, and practical tutorials how to integrate and orchestrate services with Step Functions.



Tutorials for learning Step Functions

For a quick introduction, see: [Getting started tutorial](#).

For specific scenarios, see the following tutorials:

- the section called “Handle error conditions”
- the section called “Create a state machine using AWS SAM”
- the section called “Create a state machine with CloudFormation”
- the section called “Using CDK to create an Express workflow”
- the section called “Using CDK to create a Standard workflow”
- the section called “Examine executions”
- the section called “Create a state machine that uses Lambda”
- the section called “Wait for human approval”
- the section called “Repeat actions with Inline Map”
- the section called “Copy large-scale CSV using Distributed Map”
- the section called “Iterate a loop with Lambda”
- the section called “Process batch data with Lambda”
- the section called “Process individual items with Lambda”
- the section called “Start a workflow from EventBridge”
- the section called “Create an API using API Gateway”
- the section called “Create an Activity state machine”
- the section called “View X-Ray traces”
- the section called “Gather Amazon S3 bucket info”
- the section called “Continue long-running workflows using Step Functions API (recommended)”
- the section called “Using Lambda to continue a workflow”
- the section called “Access cross-account resources”

Learn with starter templates

To deploy and learn from ready-to-run state machines for a variety of use cases, see Starter templates.

Handling error conditions in a Step Functions state machine

In this tutorial, you create an AWS Step Functions state machine with a **Task** state that invokes an example Lambda function built to throw a custom error.

Tasks are one of the [Fallback states](), for which you can configure a `Catch` field. When errors are received by the integration, next steps are chosen by the catch field based on the error name.

Step 1: Create a Lambda function that throws an error

Use a Lambda function to simulate an error condition.

1. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Choose **Create function**.
3. Choose **Use a blueprint**, search for `Step Functions`, and choose **Throw a custom error**.
4. For **Function name**, enter `ThrowErrorFunction`.
5. For **Role**, choose **Create a new role with basic Lambda permissions**.
6. Choose **Create function**.

   The following code should be displayed in the **code** pane.

```
export const handler = async () => {
    function CustomError(message) {
        this.name = 'CustomError';
        this.message = message;
    }
    CustomError.prototype = new Error();

    throw new CustomError('This is a custom error!');
};
```

Step 2: Test your Lambda function

Before creating a state machine, verify your Lambda function throws your `CustomError` when invoked.

1. Choose the **Test** tab.
2. Choose **Create a new event** and keep the default **Event JSON**

3. Choose **Test** to invoke your function with your test event.
4. Expand **Executing function** to review the details of the thrown error.

You now have a Lambda function ready to throw custom error.

In the next step, you will set up a state machine to catch and retry on that error.

Step 3: Create your state machine machine

Use the Step Functions console to create a state machine that uses a Task workflow state with a `Catch` configuration. The state machine will invoke the Lambda function, which you've built to simulate throwing an error when the function is invoked. Step Functions retries the function using exponential backoff between retries.

1. Open the Step Functions console, choose **State machines** from the menu, then choose **Create state machine**.
2. Choose **Create from blank**, and for **State machine name**, enter *`CatchErrorStateMachine`*.
3. Accept the default type (Standard), then choose **Continue** to edit your state machine in Workflow Studio.
4. Choose **Code** to switch to the ASL editor, then replace the code with following state machine definition:

```
{
  "Comment": "Example state machine that can catch a custom error thrown by a
 function integration.",
  "StartAt": "CreateAccount",
  "States": {
    "CreateAccount": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Output": "{% $states.result.Payload %}",
      "Arguments": {
        "FunctionName": "arn:aws:lambda:region:account-
id:function:ThrowErrorFunction:$LATEST",
        "Payload": "{% $states.input %}"
      },
      "Catch": [
        {
          "ErrorEquals": [
            "CustomError"
```

```
          ],
          "Next": "CustomErrorFallback"
        },
        {
          "ErrorEquals": [
            "States.ALL"
          ],
          "Next": "CatchAllFallback"
        }
      ],
      "End": true,
      "Retry": [
        {
          "ErrorEquals": [
            "CustomError",
            "Lambda.ServiceException",
            "Lambda.AWSLambdaException",
            "Lambda.SdkClientException",
            "Lambda.TooManyRequestsException"
          ],
          "IntervalSeconds": 1,
          "MaxAttempts": 3,
          "BackoffRate": 2,
          "JitterStrategy": "FULL"
        }
      ]
    },
    "CustomErrorFallback": {
      "Type": "Pass",
      "End": true,
      "Output": {
        "Result": "Fallback from a custom error function."
      }
    },
    "CatchAllFallback": {
      "Type": "Pass",
      "End": true,
      "Output": {
        "Result": "Fallback from all other error codes."
      }
    }
  },
  "QueryLanguage": "JSONata"
```

```
}
```

Step 4: Configure your state machine

Before you run your state machine, you must first connect to the Lambda function that you previously created.

1. Switch back to **Design** mode and select the **Lambda : Invoke** task state named **CreateAccount**.
2. On the **Configuration** tab, look for **API Arguments**. For **Function name** choose the Lambda function that you created earlier.
3. Choose **Create**, review the roles, then choose **Confirm** to create your state machine.

Step 5: Run the state machine

After you create and configure your state machine, you can run it and examine the flow.

1. In the editor, choose **Execute**.

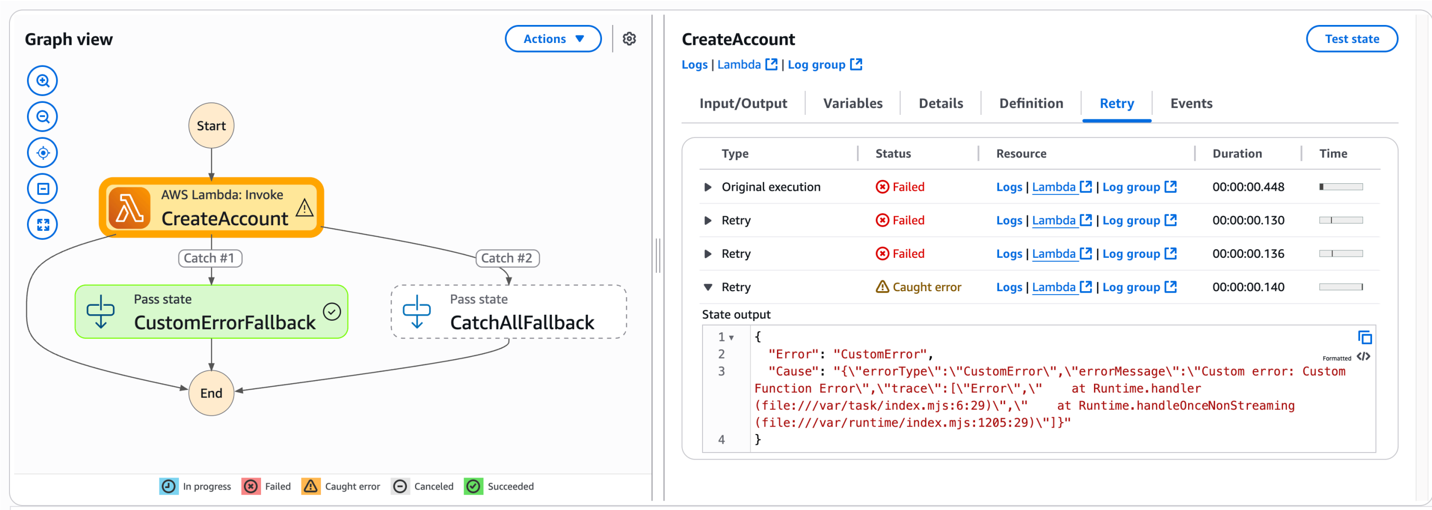
   Alternatively, from the **State machines** list, choose **Start execution**.
2. In the **Start execution** dialog box, accept the generated ID, and for **Input**, enter the following JSON:

   ```
   { "Cause" : "Custom Function Error" }
   ```
3. Choose **Start execution**.

The Step Functions console directs you to a page that's titled with your execution ID, known as the *Execution Details* page. You can review the execution results as the workflow progresses and after it completes.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

Your state machine invokes the Lambda function, which throws a `CustomError`. Choose the **CreateAccount** step in the **Graph view** to see the state output. Your state machine output should look similar to the following illustration:



Congratulations!

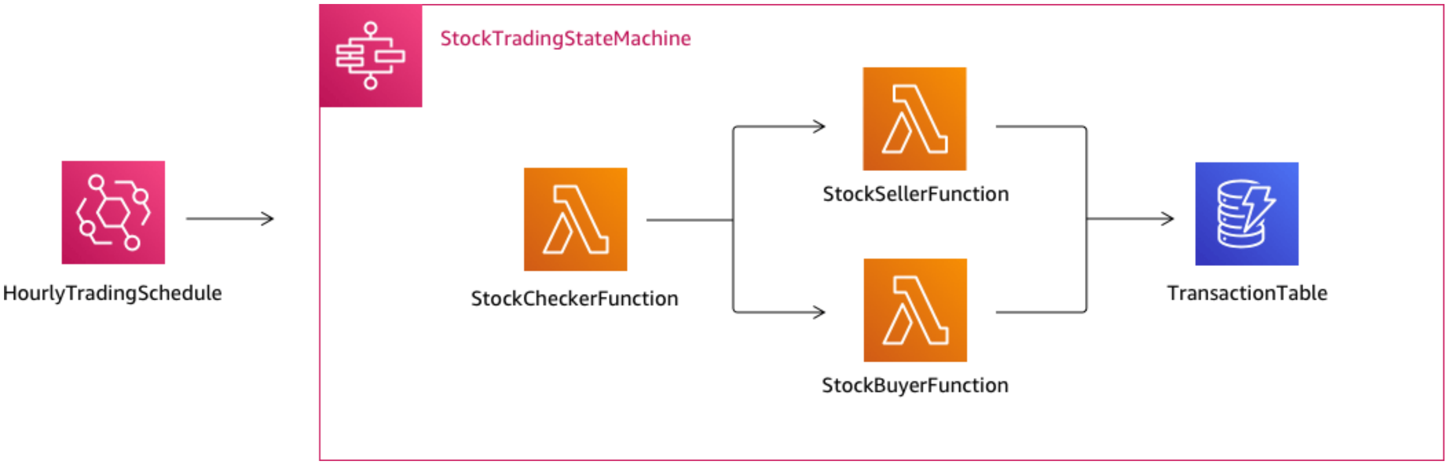
You now have a state machine that can catch and handle error conditions thrown by a Lambda function. You can use this pattern to implement robust error handling in your workflows.

> **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.

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 diagram shows the components of this application:



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 'Multi-step workflow'
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:** Multi-step workflow

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:

```
### 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 Using Amazon States Language to define Step Functions workflows.

You can see the following entry in the `template.yaml` file, which points to the state machine definition file:

```
    Properties:
      DefinitionUri: statemachine/stock_trader.asl.json
```

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 take 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:

1. Sign in to the AWS Management Console and open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
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:

```
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.

Examining state machine executions in Step Functions

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 on the **Table view** and apply suitable filters to view only the information of interest to you.

In this tutorial, you create a Standard type state machine, which 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 designed to throw an error if the price of the fruits is less than or equal to a threshold value.

> **Note**
>
> While the following procedure contains instructions for how to examine the details of a Standard workflow execution, you can also examine the details for Express workflow executions. For information about the differences between the execution details for Standard and Express workflow types, see Standard and Express console experience differences.

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 section. Make sure to name the Lambda function **`GetListOfFruits`**.
2. After you create your Lambda function, copy the function's Amazon Resource Name (ARN) displayed in the upper-right corner of the page. To copy the ARN, click the copy icon to copy the Lambda function's Amazon Resource Name. The following is an example ARN, where *`function-name`* is the name of the Lambda function (in this case, `GetListOfFruits`):

```
arn:aws:lambda:region:123456789012:function:function-name
```

3. Copy the following code for the Lambda function into the **Code source** area of the **GetListOfFruits** page.

```
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','Jos
```

```
     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;
     if (num <= errorChance) {
         throw(new Error('Error'));
     }

    return getRandomSubarray(fruits, 4);
};
```

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

5. Create two additional Lambda functions, named **`GetFruitPrice`** and **`CalculateAverage`** respectively, with the following steps:

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

```
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:

```
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]));
    }

    return average(event);
};
```

c. Make sure to copy the ARNs of these two Lambda functions, and then **Deploy** and **Test** them.

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. In this state machine, three `Map` states are defined. Each of these `Map` states contains a `Task` state that invokes one of your 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 number of retry attempts defined 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.

```
{
    "StartAt": "LoopOverStores",
    "States": {
        "LoopOverStores": {
            "Type": "Map",
            "Iterator": {
                "StartAt": "GetListOfFruits",
                "States": {
                    "GetListOfFruits": {
                        "Type": "Task",
                        "Resource": "arn:aws:states:::lambda:invoke",
                        "OutputPath": "$.Payload",
                        "Parameters": {
                            "FunctionName":
 "arn:aws:lambda:region:123456789012:function:GetListofFruits:$LATEST",
                            "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": "$.Payload",
                                      "Parameters": {
                                          "FunctionName":
  "arn:aws:lambda:region:123456789012:function:GetFruitPrice:$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": "$.Payload",
                        "Parameters": {
                            "FunctionName":
 "arn:aws:lambda:region:123456789012:function:Calculate-average:$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 this page and choose **Create state machine**.

5. Open the page titled with your state machine name. Perform steps 1 through 4 in the Step 4: Run the state machine section, but use the following data as the execution input:

```
{
    "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* 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* section. Notice that the step that caused the error, which is a `Task` state named **GetListofFruits**, is 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 either 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 third and fourth iterations, do the following:

1. Choose the `Map` state that you want to view the iteration data for.

2. From **Map iteration viewer**, choose the iteration that 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, Step Functions displays the parent and child `Map` state iterations as two separate dropdown lists representing the iteration data for the nested states.

3. (Optional) If one or more of your `Map` state iterations failed to execute or was 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 iteration number three. Similarly, choose the row named **#3** for iteration number four.

Table view – Configure the columns to display

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

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**
>
> Step Functions is based on *state machines* and *tasks*. In Step Functions, state machines are called *workflows*, which are a series of event-driven steps. Each step in a workflow is called a *state*. For example, a Task state represents a unit of work that another AWS service performs, such as calling another AWS service or API. Instances of running workflows performing tasks are called *executions* in Step Functions.
> For more information, see:
>
> - What is Step Functions?
> - Integrating services with Step Functions

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.

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. For **Function name**, enter `HelloFunction`.
4. Keep the default selections for all other options, and then choose **Create function**.
5. After your Lambda function is created, copy the function's Amazon Resource Name (ARN) displayed in the upper-right corner of the page. The following is an example ARN:

   ```
   arn:aws:lambda:region:123456789012:function:HelloFunction
   ```

6. Copy the following code for the Lambda function into the **Code source** section of the ***`HelloFunction`*** page.

   ```
   export const handler = async(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. The `callback` method returns the assembled greeting from your function.

7. Choose **Deploy**.

Step 2: Test the Lambda function

Test your Lambda function to see it in operation.

1. Choose **Test**.
2. For **Event name**, enter `HelloEvent`.
3. Replace the **Event JSON** data with the following.

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

   The `"who"` entry corresponds to the `event.who` field in your Lambda function, completing the greeting. You will input the same input data when you run your state machine.
4. Choose **Save** and then choose **Test**.
5. To review the test results, under **Execution result**, expand **Details**.

Step 3: Create a state machine

Use the Step Functions console to create a state machine that invokes the Lambda function that you created in Step 1.

1. Open the Step Functions console, choose **State machines** from the menu, then choose **Create state machine**.

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

2. Choose **Create from blank**.
3. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.
4. In the States browser on the left, make sure you've chosen the **Actions** tab. Then, drag and drop the **AWS Lambda Invoke** API into the empty state labelled **Drag first state here**.
5. In the Inspector panel on the right, configure the Lambda function:

a. In the **API Parameters** section, choose the Lambda function that you created earlier in the **Function name** dropdown list.

b. Keep the default selection in the **Payload** dropdown list.

6. (Optional) Choose **Definition** to view the state machine's Amazon States Language (ASL) definition, which is automatically generated based on your selections in the **Actions** tab and **Inspector** panel.

7. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

   For example, enter the name **`LambdaStateMachine`**.

   > **Note**
   >
   > Names of state machines, executions, and activity tasks must not exceed 80 characters in length. These names 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 accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

8. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

   For this tutorial, keep all the default selections in **State machine settings**.

9. Choose **Create**.

10. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

You can also choose **View role settings** to go back to **State machine configuration**.

> **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.

Step 4: Run the state machine

After you create your state machine, you can run it.

1. On the **State machines** page, choose **LambdaStateMachine**.
2. Choose **Start execution**.

   The **Start execution** dialog box is displayed.
3. (Optional) Enter a custom execution name to override the generated default.

   > **Non-ASCII names and logging**
   >
   > Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.
4. In the **Input** area, replace the example execution data with the following.

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

   `"who"` is the key name that your Lambda function uses to get the name of the person to greet.
5. Choose **Start Execution**.

   Your state machine's execution starts, and a new page showing your running execution is displayed.

6. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

   To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

> **Note**
>
> You can also pass payloads while invoking Lambda from a state machine. For more information and examples about invoking Lambda by passing payload in the `Parameters` field, see Invoke an AWS Lambda function with Step Functions.

Deploying a workflow that waits for human approval in Step Functions

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

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

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

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

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

Step 1: Create an AWS CloudFormation template

1. Copy the example code from the AWS CloudFormation Template Source Code 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**, and then choose **With new resources (standard)**.
3. On the **Create stack** page, do the following:
   a. In the **Prerequisite - Prepare template** section, make sure **Template is ready** is selected.
   b. In the **Specify template** section, choose **Upload a template file** and then choose **Choose file** to upload the `human-approval.yaml` file you created earlier that includes the template source code.
4. Choose **Next**.
5. On the **Specify stack details** page, do the following:
   a. For **Stack name**, enter a name for your stack.
   b. Under **Parameters**, enter a valid email address. You'll use this email address to subscribe to the Amazon SNS topic.
6. Choose **Next**, and then choose **Next** again.
7. 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.

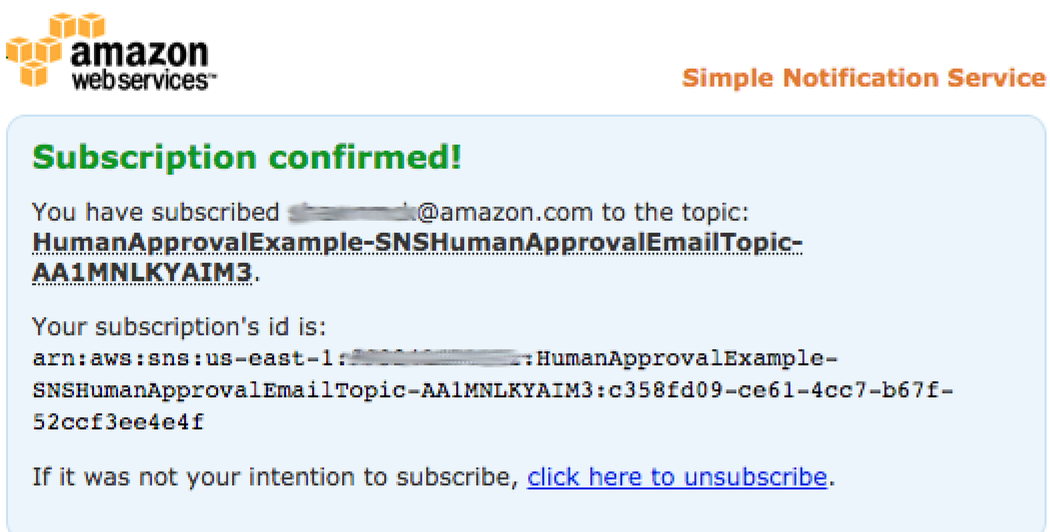
8. (Optional) To display the resources in your stack, select the stack and choose the **Resources** tab.

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.



Step 4: Run the state machine

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

The **Start execution** dialog box is displayed.

2. In the **Start execution** dialog box, do the following:

   a. (Optional) Enter a custom execution name to override the generated default.

      Non-ASCII names and logging

      Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

   b. In the **Input** box, enter the following JSON input to run your workflow.

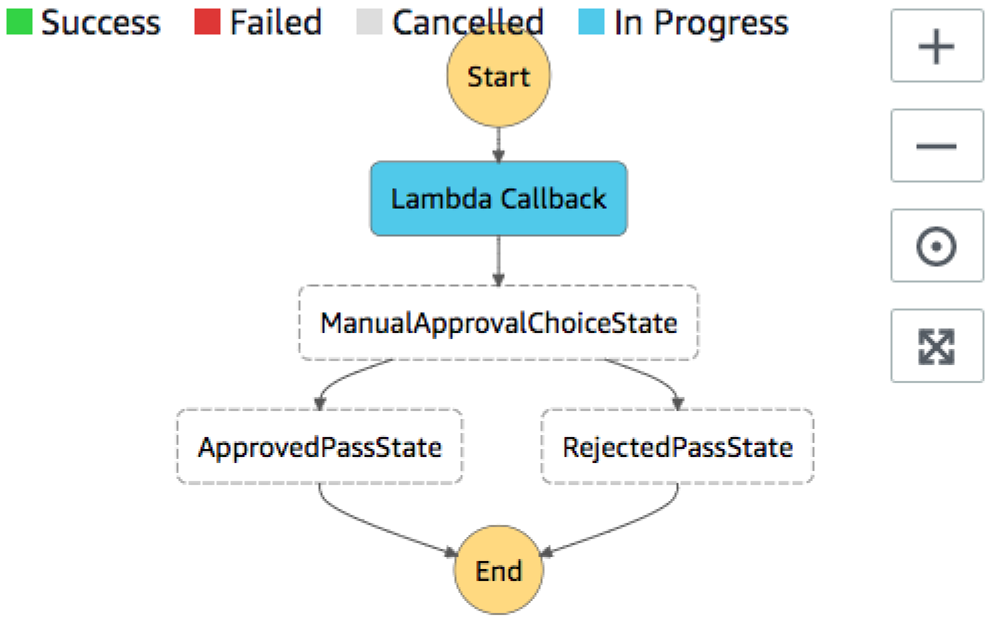
      ```
      {
          "Comment": "Testing the human approval tutorial."
      }
      ```

   c. Choose **Start execution**.

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

   d. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

      To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

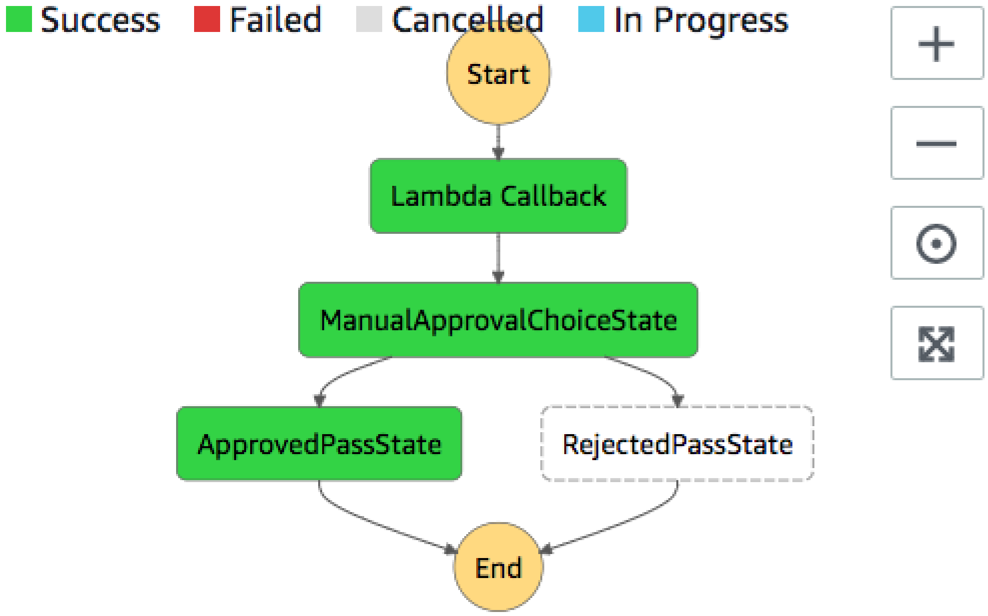


3. In the email account you used for the Amazon SNS topic earlier, open the message with the subject **Required approval from AWS Step Functions**.

   The message includes separate URLs for **Approve** and **Reject**.

4. Choose the **Approve** URL.

   The workflow continues based on your choice.



AWS CloudFormation Template Source Code

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

```
AWSTemplateFormatVersion: "2010-09-09"
Description: "AWS Step Functions Human based task example. It sends an email with an
 HTTP URL for approval."
Parameters:
  Email:
    Type: String
    AllowedPattern: "^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$"
    ConstraintDescription: Must be a valid email address.
Resources:
  # Begin API Gateway Resources
  ExecutionApi:
    Type: "AWS::ApiGateway::RestApi"
    Properties:
      Name: "Human approval endpoint"
      Description: "HTTP Endpoint backed by API Gateway and Lambda"
      FailOnWarnings: true

  ExecutionResource:
```

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

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

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

                #end
              },
              "query": {
                #foreach($queryParam in $input.params().querystring.keySet())
```

```
                "$queryParam":
"$util.escapeJavaScript($input.params().querystring.get($queryParam))"
#if($foreach.hasNext),#end

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

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

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

  ExecutionApiStage:
    DependsOn:
      - ApiGatewayAccount
```

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

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

  # Begin
  # Lambda that will be invoked by API Gateway
  LambdaApprovalFunction:
    Type: 'AWS::Lambda::Function'
    Properties:
      Code:
        ZipFile:
          Fn::Sub: |
            const { SFN: StepFunctions } = require("@aws-sdk/client-sfn");
            var redirectToStepFunctions = function(lambdaArn, statemachineName,
 executionName, callback) {
              const lambdaArnTokens = lambdaArn.split(":");
              const partition = lambdaArnTokens[1];
              const region = lambdaArnTokens[3];
              const accountId = lambdaArnTokens[4];

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

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

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

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

            const stepfunctions = new StepFunctions();

            var message = "";

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

            stepfunctions.sendTaskSuccess({
              output: JSON.stringify(message),
              taskToken: event.query.taskToken
            })
            .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: nodejs18.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:
${AWS::Partition}:states:::lambda:invoke.waitForTaskToken",
                      "Parameters": {
                        "FunctionName": "${LambdaHumanApprovalSendEmailFunction.Arn}",
                        "Payload": {
                          "ExecutionContext.$": "$$",
                          "APIGatewayEndpoint": "https://${ExecutionApi}.execute-api.
${AWS::Region}.amazonaws.com/states"
                        }
                      },
                      "Next": "ManualApprovalChoiceState"
                  },
                  "ManualApprovalChoiceState": {
                    "Type": "Choice",
                    "Choices": [
                      {
                        "Variable": "$.Status",
                        "StringEquals": "Approved! Task approved by ${Email}",
                        "Next": "ApprovedPassState"
                      },
                      {
                        "Variable": "$.Status",
                        "StringEquals": "Rejected! Task rejected by ${Email}",
                        "Next": "RejectedPassState"
                      }
                    ]
                  },
                  "ApprovedPassState": {
                    "Type": "Pass",
                    "End": true
```

```
                },
                "RejectedPassState": {
                  "Type": "Pass",
                  "End": true
                }
            }
        }

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

  LambdaHumanApprovalSendEmailFunction:
    Type: "AWS::Lambda::Function"
    Properties:
      Handler: "index.lambda_handler"
      Role: !GetAtt LambdaSendEmailExecutionRole.Arn
      Runtime: "nodejs18.x"
      Timeout: "25"
      Code:
        ZipFile:
          Fn::Sub: |
            console.log('Loading function');
            const { SNS } = require("@aws-sdk/client-sns");
            exports.lambda_handler = (event, context, callback) => {
                console.log('event= ' + JSON.stringify(event));
                console.log('context= ' + JSON.stringify(context));

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

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

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

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

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

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

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

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

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

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

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

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

  LambdaSendEmailExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: "sts:AssumeRole"
      Policies:
        - PolicyName: CloudWatchLogsPolicy
          PolicyDocument:
            Statement:
              - Effect: Allow
                Action:
                  - "logs:CreateLogGroup"
                  - "logs:CreateLogStream"
                  - "logs:PutLogEvents"
                Resource: !Sub "arn:${AWS::Partition}:logs:*:*:*"
        - PolicyName: SNSSendEmailPolicy
          PolicyDocument:
            Statement:
              - Effect: Allow
```

```
                Action:
                  - "SNS:Publish"
                Resource:
                  - !Sub "${SNSHumanApprovalEmailTopic}"

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

Using Inline Map state to repeat an action in Step Functions

This tutorial helps you get started with using the `Map` state in Inline mode. You use the *Inline Map state* in your workflows to repeatedly perform an action. For more information about Inline mode, see Map state in Inline mode.

In this tutorial, you use the *Inline Map state* to repeatedly generate version 4 universally unique identifiers (v4 UUID). You start by creating a workflow that contains two Pass workflow state states and an *Inline Map state* in the Workflow Studio. Then, you configure the input and output, including the input JSON array for the `Map` state. The `Map` state returns an output array that contains the v4 UUIDs generated for each item in the input array.

Step 1: Create the workflow prototype

In this step, you create the prototype for your workflow using Workflow Studio. Workflow Studio is a visual workflow designer available in the Step Functions console. You'll choose the required states from the **Flow** tab and use the drag and drop feature of Workflow Studio to create the workflow prototype.

1. Open the Step Functions console, choose **State machines** from the menu, then choose **Create state machine**.
2. Choose **Create from blank**.
3. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.

4. From the **Flow** tab, drag a **Pass** state and drop it to the empty state labelled **Drag first state here**.
5. Drag a **Map** state and drop it below the **Pass** state. Rename the **Map** state to **`Map demo`**.
6. Drag a second **Pass** state and drop it inside of the **Map demo** state.
7. Rename the second **Pass** state to **`Generate UUID`**.

Step 2: Configure input and output

In this step, you configure input and output for all the states in your workflow prototype. First, you inject some fixed data into the workflow using the first **Pass** state. This **Pass** state passes on this data as input to the **Map demo** state. Within this input, you specify the node that contains the input array the **Map demo** state should iterate over. Then you define the step that the **Map demo** state should repeat to generate the v4 UUIDs. Finally, you configure the output to return for each repetition.

1. Choose the first **Pass** state in your workflow prototype. In the **Output** tab, enter the following under **Result**:

```
{
  "foo": "bar",
  "colors": [
    "red",
    "green",
    "blue",
    "yellow",
    "white"
  ]
}
```

2. Choose the **Map demo** state and in the **Configuration** tab, do the following:

   a. Choose **Provide a path to items array**.

   b. Specify the following reference path to select the node that contains the input array:

   ```
   $.colors
   ```

3. Choose the **Generate UUID** state and in the **Input** tab, do the following:

   a. Choose **Transform input with Parameters**.

b. Enter the following JSON input to generate the v4 UUIDs for each of the input array items. You use the `States.UUID` intrinsic function to generate the UUIDs.

```
{
  "uuid.$": "States.UUID()"
}
```

4. For the **Generate UUID** state, choose the **Output** tab and do the following:

a. Choose **Filter output with OutputPath**.

b. Enter the following reference path to select the JSON node that contains the output array items:

```
$.uuid
```

Step 3: Review and save auto-generated definition

As you drag and drop states from the **Flow** panel onto the canvas, Workflow Studio automatically composes the Amazon States Language (ASL) definition of your workflow in real-time. You can edit this definition as required.

1. (Optional) Choose **Definition** on the Inspector panel panel to view the automatically-generated Amazon States Language definition of your workflow.

Tip

You can also view the ASL definition in the Code editor of Workflow Studio. In the code editor, you can also edit the ASL definition of your workflow.

The following example shows the automatically generated Amazon States Language definition for your workflow.

```
{
    "Comment": "Using Map state in Inline mode",
    "StartAt": "Pass",
    "States": {
      "Pass": {
```

```
        "Type": "Pass",
        "Next": "Map demo",
        "Result": {
          "foo": "bar",
          "colors": [
            "red",
            "green",
            "blue",
            "yellow",
            "white"
          ]
        }
      },
      "Map demo": {
        "Type": "Map",
        "ItemsPath": "$.colors",
        "ItemProcessor": {
          "ProcessorConfig": {
            "Mode": "INLINE"
          },
          "StartAt": "Generate UUID",
          "States": {
            "Generate UUID": {
              "Type": "Pass",
              "End": true,
              "Parameters": {
                "uuid.$": "States.UUID()"
              },
              "OutputPath": "$.uuid"
            }
          }
        },
        "End": true
      }
    }
  }
```

2. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

   For this tutorial, enter the name **`InlineMapDemo`**.

3. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

   For this tutorial, keep all the default selections in **State machine configuration**.

4. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

   You can also choose **View role settings** to go back to **State machine configuration**.

   > **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.

Step 4: Run the state machine

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

1. On the **InlineMapDemo** page, choose **Start execution**.
2. In the **Start execution** dialog box, do the following:

   1. (Optional) Enter a custom execution name to override the generated default.

      > **Non-ASCII names and logging**
      >
      > Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

   2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.
   3. Choose **Start execution**.
   4. The Step Functions console directs you to a page that's titled with your execution ID, known as the *Execution Details* page. You can review the execution results as the workflow progresses and after it completes.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

To view the execution input and output side-by-side, choose **Execution input and output**. Under **Output**, view the output array returned by the `Map` state. The following is an example of the output array:

```
[
  "a85cbc7b-4e65-4ac2-97af-80ed504adc1d",
  "b05bca11-d481-414e-aa9a-88285ec6590d",
  "f42d59f7-bd32-480f-b270-caddb518ce2a",
  "15f18616-517d-4b69-b7c3-bf22222d2efd",
  "690bcfee-6d58-408c-a6b4-1995ccafdbd2"
]
```

Copying large-scale CSV data using Distributed Map in Step Functions

This tutorial helps you start using the `Map` state in Distributed mode. A `Map` state set to **Distributed** is known as a *Distributed Map state*. You use the *Distributed Map state* in your workflows to iterate over large-scale Amazon S3 data sources. The `Map` state runs each iteration as a child workflow execution, which enables high concurrency. For more information about Distributed mode, see Map state in Distributed mode.

In this tutorial, you use the *Distributed Map state* to iterate over a CSV file in an Amazon S3 bucket. You then return its contents, along with the ARN of a child workflow execution, in another Amazon S3 bucket. You start by creating a workflow prototype in the Workflow Studio. Next, you set the Map state's processing mode to Distributed, specify the CSV file as the dataset, and provide its location to the `Map` state. You also specify the workflow type for the child workflow executions that the *Distributed Map state* starts as **Express**.

In addition to these settings, you also specify other configurations, such as the maximum number of concurrent child workflow executions and the location to export the `Map` result, for the example workflow used in this tutorial.

Prerequisites

- Upload a CSV file to an Amazon S3 bucket. You must define a header row within your CSV file. For information about size limits imposed on the CSV file and how to specify the header row, see CSV file in an Amazon S3 bucket.
- Create another Amazon S3 bucket and a folder within that bucket to export the `Map` state result to.

> **Important**
>
> Make sure that your Amazon S3 buckets are in the same AWS account and AWS Region as your state machine.
> Note that even though your state machine may be able to access files in buckets across different AWS accounts that are in the same AWS Region, Step Functions only supports state machines to list objects in S3 buckets that are in *both* the same AWS account and the same AWS Region as the state machine.

Step 1: Create the workflow prototype

In this step, you create the prototype for your workflow using Workflow Studio. Workflow Studio is a visual workflow designer available in the Step Functions console. You choose the required state and API action from the **Flow** and **Actions** tabs respectively. You'll use the drag and drop feature of Workflow Studio to create the workflow prototype.

1. Open the Step Functions console, choose **State machines** from the menu, then choose **Create state machine**.
2. Choose **Create from blank**.
3. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.
4. From the **Flow** tab, drag a **Map** state and drop it to the empty state labelled **Drag first state here**.
5. In the **Configuration** tab, for **State name**, enter **`Process data`**.
6. From the **Actions** tab, drag an **AWS Lambda Invoke** API action and drop it inside the **Process data** state.

7. Rename the **AWS Lambda Invoke** state to **`Process CSV data`**.

Step 2: Configure the required fields for Map state

In this step, you configure the following required fields of the *Distributed Map state*:

- ItemReader – Specifies the dataset and its location from which the `Map` state can read input.
- ItemProcessor – Specifies the following values:
  - `ProcessorConfig` – Set the `Mode` and `ExecutionType` to `DISTRIBUTED` and `EXPRESS` respectively. This sets the `Map` state's processing mode and the workflow type for child workflow executions that the *Distributed Map state* starts.
  - `StartAt` – The first state in the Map workflow.
  - `States` – Defines the Map workflow, which is a set of steps to repeat in each child workflow execution.
- ResultWriter – Specifies the Amazon S3 location where Step Functions writes the *Distributed Map state* results.

> **Important**
>
> Make sure that the Amazon S3 bucket you use to export the results of a Map Run is under the same AWS account and AWS Region as your state machine. Otherwise, your state machine execution will fail with the `States.ResultWriterFailed` error.

To configure the required fields:

1. Choose the **Process data** state and, in the **Configuration** tab, do the following:
   a. For **Processing mode**, choose **Distributed**.
   b. For **Item source**, choose **Amazon S3**, and then choose **CSV file in S3** from the **S3 item source** dropdown list.
   c. Do the following to specify the Amazon S3 location of your CSV file:
      i. For **S3 object**, select **Enter bucket and key** from the dropdown list.
      ii. For **Bucket**, enter the name of the Amazon S3 bucket, which contains the CSV file. For example, **`amzn-s3-demo-source-bucket`**.

iii. For **Key**, enter the name of the Amazon S3 object in which you saved the CSV file. You must also specify the name of the CSV file in this field. For example, **`csvDataset/ratings.csv`**.

d. For CSV files, you must also specify the location of the column header. To do this, choose **Additional configuration**, and then for **CSV header location** keep the default selection of **First row** if the first row of your CSV file is the header. Otherwise, choose **Given** to specify the header within the state machine definition. For more information, see `ReaderConfig`.

e. For **Child execution type**, choose **Express**.

2. In **Export location**, to export the Map Run results to a specific Amazon S3 location, choose **Export Map state's output to Amazon S3**.

3. Do the following:

   a. For **S3 bucket**, choose **Enter bucket name and prefix** from the dropdown list.

   b. For **Bucket**, enter the name of the Amazon S3 bucket where you want to export the results to. For example, **`mapOutputs`**.

   c. For **Prefix**, enter the folder name where you want to save the results to. For example, **`resultData`**.

Step 3: Configure additional options

In addition to the required settings for a *Distributed Map state*, you can also specify other options. These can include the maximum number of concurrent child workflow executions and the location to export the `Map` state result to.

1. Choose the **Process data** state. Then, in **Item source**, choose **Additional configuration**.

2. Do the following:

   a. Choose **Modify items with ItemSelector** to specify a custom JSON input for each child workflow execution.

   b. Enter the following JSON input:

```
{
  "index.$": "$$.Map.Item.Index",
  "value.$": "$$.Map.Item.Value"
}
```

For information about how to create a custom input, see [ItemSelector (Map)].

3. In **Runtime settings**, for **Concurrency limit**, specify the number of concurrent child workflow executions that the *Distributed Map state* can start. For example, enter **100**.
4. Open a new window or tab on your browser and complete the configuration of the Lambda function you'll use in this workflow, as explained in [Step 4: Configure the Lambda function].

Step 4: Configure the Lambda function

> **Important**
>
> Ensure that your Lambda function is under the same 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 **distributedMapLambda**.

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

   c. Keep all of the default selections and choose **Create function**.

   d. After you create your Lambda function, copy the function's Amazon Resource Name (ARN) displayed in the upper-right corner of the page. You'll need to provide this in your workflow prototype. The following is an example ARN:

   ```
   arn:aws:lambda:us-east-2:123456789012:function:distributedMapLambda
   ```

4. Copy the following code for the Lambda function and paste it into the **Code source** section of the **distributedMapLambda** page.

```
exports.handler = async function(event, context) {
    console.log("Received Input:\n", event);

    return {
        'statusCode' : 200,
        'inputReceived' : event //returns the input that it received
    }
```

```
};
```

5. Choose **Deploy**. Once your function deploys, choose **Test** to see the output of your Lambda function.

Step 5: Update the workflow prototype

In the Step Functions console, you'll update your workflow to add the Lambda function's ARN.

1. Return to the tab or window where you created the workflow prototype.
2. Choose the **Process CSV data** step, and in the **Configuration** tab, do the following:

    a. For **Integration type**, choose **Optimized**.

    b. For **Function name**, start to enter the name of your Lambda function. Choose the function from the dropdown list that appears, or choose **Enter function name** and provide the Lambda function ARN.

Step 6: Review the auto-generated Amazon States Language definition and save the workflow

As you drag and drop states from the **Action** and **Flow** tabs onto the canvas, Workflow Studio automatically composes the Amazon States Language definition of your workflow in real-time. You can edit this definition as required.

1. (Optional) Choose **Definition** on the Inspector panel panel and view the state machine definition.

    > **Tip**
    >
    > You can also view the ASL definition in the Code editor of Workflow Studio. In the code editor, you can also edit the ASL definition of your workflow.

    The following example code shows the automatically generated Amazon States Language definition for your workflow.

```
{
```

```
  "Comment": "Using Map state in Distributed mode",
  "StartAt": "Process data",
  "States": {
    "Process data": {
      "Type": "Map",
      "MaxConcurrency": 100,
      "ItemReader": {
        "ReaderConfig": {
          "InputType": "CSV",
          "CSVHeaderLocation": "FIRST_ROW"
        },
        "Resource": "arn:aws:states:::s3:getObject",
        "Parameters": {
          "Bucket": "amzn-s3-demo-source-bucket",
          "Key": "csvDataset/ratings.csv"
        }
      },
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "DISTRIBUTED",
          "ExecutionType": "EXPRESS"
        },
        "StartAt": "Process CSV data",
        "States": {
          "Process CSV data": {
            "Type": "Task",
            "Resource": "arn:aws:states:::lambda:invoke",
            "OutputPath": "$.Payload",
            "Parameters": {
              "Payload.$": "$",
              "FunctionName": "arn:aws:lambda:us-east-2:account-
id:function:distributedMapLambda"
            },
            "End": true
          }
        }
      },
      "Label": "Processdata",
      "End": true,
      "ResultWriter": {
        "Resource": "arn:aws:states:::s3:putObject",
        "Parameters": {
          "Bucket": "mapOutputs",
          "Prefix": "resultData"
```

```
          }
        },
        "ItemSelector": {
          "index.$": "$$.Map.Item.Index",
          "value.$": "$$.Map.Item.Value"
        }
      }
    }
  }
```

2. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

   For this tutorial, enter the name **`DistributedMapDemo`**.

3. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

   For this tutorial, keep all the default selections in **State machine configuration**.

4. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

   You can also choose **View role settings** to go back to **State machine configuration**.

   > **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.

Step 7: Run the state machine

An *execution* is an instance of your state machine where you run your workflow to perform tasks.

1. On the **DistributedMapDemo** page, choose **Start execution**.
2. In the **Start execution** dialog box, do the following:

   1. (Optional) Enter a custom execution name to override the generated default.

> **Non-ASCII names and logging**
>
> Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.
3. Choose **Start execution**.
4. The Step Functions console directs you to a page that's titled with your execution ID, known as the *Execution Details* page. You can review the execution results as the workflow progresses and after it completes.

   To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

For example, choose the `Map` state, and then choose **Map Run** to open the *Map Run Details* page. On this page, you can view all the execution details of the *Distributed Map state* and the child workflow executions that it started. For information about this page, see Viewing Map Runs.

Iterate a loop with a Lambda function in Step Functions

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 tutorial to ensure you are familiar with using Lambda and Step Functions together.

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**.

    c. In **Change default execution role**, choose **Create a new role with basic Lambda permissions**.

    d. Choose **Create function**.
4. Copy the following code for the Lambda function into the **Code source**.

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

  index = index + step

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

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

5. Choose **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 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 `true` for `continue` when the index is less than `count`. For this test, the index has already incremented to 5. The test will increment `index` to 6 and set `continue` to `true`.

4. Choose **Create**.
5. 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 and test again, the `index` increments to `10`, and `continue` will be `false`.

Step 3: Create a State Machine

Before you leave the Lambda console...

Copy the Lambda function ARN. Paste it into a note. You'll need it in the next step.

Next, you will create a state machine with the following states:

- `ConfigureCount` – Sets default values for `count`, `index`, and `step`.
- `Iterator` – Refers to the Lambda function you created earlier, passing in the values configured in `ConfigureCount`.
- `IsCountReached` – A choice state that continues the loop or proceeds to `Done` state, based on the value returned from your `Iterator` function.
- `ExampleWork` – A stub for work that needs to be done. In this example, the workflow has a `Pass` state, but in a real solution, you would likely use a `Task`.
- `Done` – End state of your workflow.

To create the state machine in the console:

1. Open the [Step Functions console](), and then choose **Create a state machine**.

> **Important**
>
> Your state machine must be in the same AWS account and Region as your Lambda function.

2. Select the **Blank** template.
3. In the **Code** pane, paste the following JSON which defines the state machine.

   For more information about the Amazon States Language, see State Machine Structure.

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

        "ConfigureCount": {
            "Type": "Pass",
            "Result": {
                "count": 10,
                "index": 0,
                "step": 1
            },
            "ResultPath": "$.iterator",
            "Next": "Iterator"
        },
        "Iterator": {
            "Type": "Task",
            "Resource": "arn:aws:lambda:region:123456789012:function:Iterate",
            "ResultPath": "$.iterator",
            "Next": "IsCountReached"
        },
        "IsCountReached": {
            "Type": "Choice",
            "Choices": [
                {
                    "Variable": "$.iterator.continue",
                    "BooleanEquals": true,
                    "Next": "ExampleWork"
                }
            ],
            "Default": "Done"
```

```
        },
        "ExampleWork": {
            "Comment": "Your application logic, to run a specific number of times",
            "Type": "Pass",
            "Result": {
              "success": true
            },
            "ResultPath": "$.result",
            "Next": "Iterator"
        },
        "Done": {
            "Type": "Pass",
            "End": true

        }
    }
}
```

4. Replace the `Iterator Resource` field with the ARN for your `Iterator` Lambda function that you created earlier.
5. Select **Config**, and enter a **Name** for your state machine, such as *`IterateCount`*.

> **Note**
>
> Names of state machines, executions, and activity tasks must not exceed 80 characters in length. These names 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 accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

6. For **Type**, accept default value of **Standard**. For **Permissions**, choose **Create new role**.
7. Choose **Create**, and then **Confirm** the role creations.

Step 4: Start a New Execution

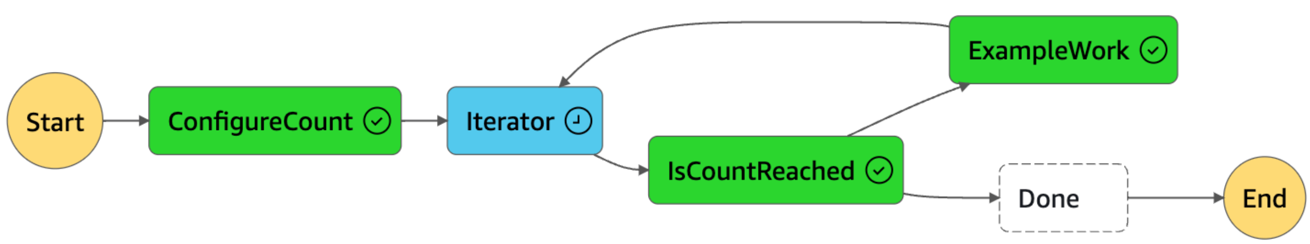
After you create your state machine, you can start an execution.

1. On the **IterateCount** page, choose **Start execution**.
2. (Optional) Enter a custom execution name to override the generated default.

> **Non-ASCII names and logging**
>
> Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

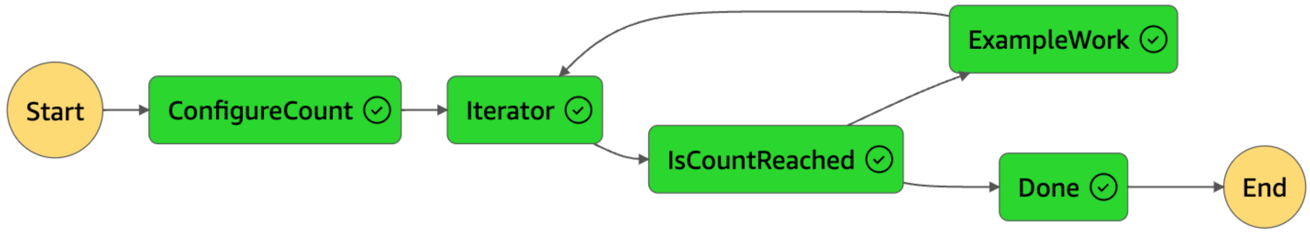
3. Choose **Start Execution**.

   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.

   When the count reaches the number specified in the `ConfigureCount` state in your state machine, the execution quits iterating and ends.



Processing batch data with a Lambda function in Step Functions

In this tutorial, you use the *Distributed Map state*'s ItemBatcher (Map) field to process an entire batch of items inside a Lambda function. Each batch contains a maximum of three items. The *Distributed Map state* starts four child workflow executions, where each execution processes three items, while one execution processes a single item. Each child workflow execution invokes a Lambda function that iterates over the individual items present in the batch.

You'll create a state machine that performs multiplication on an array of integers. Say that the integer array you provide as input is `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` and the multiplication factor is 7. Then, the resulting array formed after multiplying these integers with a factor of 7, will be `[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]`.

Step 1: Create the state machine

In this step, you create the workflow prototype of the state machine that passes an entire batch of data to the Lambda function you'll create in Step 2.

- Use the following definition to create a state machine using the Step Functions console. For information about creating a state machine, see Step 1: Create the workflow prototype in the Getting started with using Distributed Map state tutorial.

  In this state machine, you define a *Distributed Map state* that accepts an array of 10 integers as input and passes this array to a Lambda function in batches of 3. The Lambda function iterates over the individual items present in the batch and returns an output array named `multiplied`. The output array contains the result of the multiplication performed on the items passed in the input array.

> **Important**
>
> Make sure to replace the Amazon Resource Name (ARN) of the Lambda function in the following code with the ARN of the function you'll create in [Step 2](#).

```
{
  "StartAt": "Pass",
  "States": {
    "Pass": {
      "Type": "Pass",
      "Next": "Map",
      "Result": {
        "MyMultiplicationFactor": 7,
        "MyItems": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      }
    },
    "Map": {
      "Type": "Map",
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "DISTRIBUTED",
          "ExecutionType": "STANDARD"
        },
        "StartAt": "Lambda Invoke",
        "States": {
          "Lambda Invoke": {
            "Type": "Task",
            "Resource": "arn:aws:states:::lambda:invoke",
            "OutputPath": "$.Payload",
            "Parameters": {
              "Payload.$": "$",
              "FunctionName": "arn:aws:lambda:region:account-
id:function:functionName"
            },
            "Retry": [
              {
                "ErrorEquals": [
                  "Lambda.ServiceException",
                  "Lambda.AWSLambdaException",
                  "Lambda.SdkClientException",
```

```
                    "Lambda.TooManyRequestsException"
                  ],
                  "IntervalSeconds": 2,
                  "MaxAttempts": 6,
                  "BackoffRate": 2
                }
              ],
              "End": true
            }
          }
        },
        "End": true,
        "Label": "Map",
        "MaxConcurrency": 1000,
        "ItemBatcher": {
          "MaxItemsPerBatch": 3,
          "BatchInput": {
            "MyMultiplicationFactor.$": "$.MyMultiplicationFactor"
          }
        },
        "ItemsPath": "$.MyItems"
      }
    }
  }
```

Step 2: Create the Lambda function

In this step, you create the Lambda function that processes all the items passed in the batch.

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

To create the Lambda function

1. Use the Lambda console to create a **Python** Lambda function named **`ProcessEntireBatch`**. For information about creating a Lambda function, see Step 4: Configure the Lambda function in the Getting started with using Distributed Map state tutorial.
2. Copy the following code for the Lambda function and paste it into the **Code source** section of your Lambda function.

```
import json

def lambda_handler(event, context):
    multiplication_factor = event['BatchInput']['MyMultiplicationFactor']
    items = event['Items']

    results = [multiplication_factor * item for item in items]

    return {
        'statusCode': 200,
        'multiplied': results
    }
```

3. After you create your Lambda function, copy the function's ARN displayed in the upper-right corner of the page. The following is an example ARN, where *`function-name`* is the name of the Lambda function (in this case, `ProcessEntireBatch`):

   ```
   arn:aws:lambda:region:123456789012:function:function-name
   ```

   You'll need to provide the function ARN in the state machine you created in Step 1.

4. Choose **Deploy** to deploy the changes.

Step 3: Run the state machine

When you run the state machine, the *Distributed Map state* starts four child workflow executions, where each execution processes three items, while one execution processes a single item.

The following example shows the data passed to the `ProcessEntireBatch` function by one of the child workflow executions.

```
{
  "BatchInput": {
    "MyMultiplicationFactor": 7
  },
  "Items": [1, 2, 3]
}
```

Given this input, the following example shows the output array named `multiplied` that is returned by the Lambda function.

```
{
  "statusCode": 200,
  "multiplied": [7, 14, 21]
}
```

The state machine returns the following output that contains four arrays named `multiplied` for the four child workflow executions. These arrays contain the multiplication results of the individual input items.

```
[
  {
    "statusCode": 200,
    "multiplied": [7, 14, 21]
  },
  {
    "statusCode": 200,
    "multiplied": [28, 35, 42]
  },
  {
    "statusCode": 200,
    "multiplied": [49, 56, 63]
  },
  {
    "statusCode": 200,
    "multiplied": [70]
  }
]
```

To combine all the array items returned into a single output array, you can use the ResultSelector field. Define this field inside the *Distributed Map state* to find all the `multiplied` arrays, extract all the items inside these arrays, and then combine them into a single output array.

To use the `ResultSelector` field, update your state machine definition as shown in the following example.

```
{
  "StartAt": "Pass",
  "States": {
    ...
    ...
    "Map": {
```

```
      "Type": "Map",
      ...
      ...
      "ItemsPath": "$.MyItems",
      "ResultSelector": {
        "multiplied.$": "$..multiplied[*]"
      }
    }
  }
}
```

The updated state machine returns a consolidated output array as shown in the following example.

```
{
  "multiplied": [7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
}
```

Processing individual items with a Lambda function in Step Functions

In this tutorial, you use the *Distributed Map state*'s ItemBatcher (Map) field to iterate over individual items present in a batch using a Lambda function. The *Distributed Map state* starts four child workflow executions. Each of these child workflows runs an *Inline Map state*. For its each iteration, the *Inline Map state* invokes a Lambda function and passes a single item from the batch to the function. The Lambda function then processes the item and returns the result.

You'll create a state machine that performs multiplication on an array of integers. Say that the integer array you provide as input is `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` and the multiplication factor is 7. Then, the resulting array formed after multiplying these integers with a factor of 7, will be `[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]`.

Step 1: Create the state machine

In this step, you create the workflow prototype of the state machine that passes a single item from a batch of items to each invocation of the Lambda function you'll create in Step 2.

- Use the following definition to create a state machine using the Step Functions console. For information about creating a state machine, see Step 1: Create the workflow prototype in the Getting started with using Distributed Map state tutorial.

In this state machine, you define a *Distributed Map state* that accepts an array of 10 integers as input and passes these array items to the child workflow executions in batches. Each child workflow execution receives a batch of three items as input and runs an *Inline Map state*. Every iteration of the *Inline Map state* invokes a Lambda function and passes an item from the batch to the function. This function then multiplies the item with a factor of 7 and returns the result.

The output of each child workflow execution is a JSON array that contains the multiplication result for each of the items passed.

Important

Make sure to replace the Amazon Resource Name (ARN) of the Lambda function in the following code with the ARN of the function you'll create in Step 2.

```
{
  "StartAt": "Pass",
  "States": {
    "Pass": {
      "Type": "Pass",
      "Next": "Map",
      "Result": {
        "MyMultiplicationFactor": 7,
        "MyItems": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      }
    },
    "Map": {
      "Type": "Map",
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "DISTRIBUTED",
          "ExecutionType": "STANDARD"
        },
        "StartAt": "InnerMap",
        "States": {
          "InnerMap": {
            "Type": "Map",
            "ItemProcessor": {
              "ProcessorConfig": {
                "Mode": "INLINE"
```

```
              },
              "StartAt": "Lambda Invoke",
              "States": {
                "Lambda Invoke": {
                  "Type": "Task",
                  "Resource": "arn:aws:states:::lambda:invoke",
                  "OutputPath": "$.Payload",
                  "Parameters": {
                    "Payload.$": "$",
                    "FunctionName": "arn:aws:lambda:region:account-
id:function:functionName"
                  },
                  "Retry": [
                    {
                      "ErrorEquals": [
                        "Lambda.ServiceException",
                        "Lambda.AWSLambdaException",
                        "Lambda.SdkClientException",
                        "Lambda.TooManyRequestsException"
                      ],
                      "IntervalSeconds": 2,
                      "MaxAttempts": 6,
                      "BackoffRate": 2
                    }
                  ],
                  "End": true
                }
              }
            },
            "End": true,
            "ItemsPath": "$.Items",
            "ItemSelector": {
              "MyMultiplicationFactor.$": "$.BatchInput.MyMultiplicationFactor",
              "MyItem.$": "$$.Map.Item.Value"
            }
          }
        }
      },
      "End": true,
      "Label": "Map",
      "MaxConcurrency": 1000,
      "ItemsPath": "$.MyItems",
      "ItemBatcher": {
        "MaxItemsPerBatch": 3,
```

```
        "BatchInput": {
          "MyMultiplicationFactor.$": "$.MyMultiplicationFactor"
        }
      }
    }
  }
}
```

Step 2: Create the Lambda function

In this step, you create the Lambda function that processes each item passed from the batch.

Important

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

To create the Lambda function

1. Use the Lambda console to create a **Python** Lambda function named **`ProcessSingleItem`**. For information about creating a Lambda function, see Step 4: Configure the Lambda function in the Getting started with using Distributed Map state tutorial.
2. Copy the following code for the Lambda function and paste it into the **Code source** section of your Lambda function.

```
import json

def lambda_handler(event, context):

    multiplication_factor = event['MyMultiplicationFactor']
    item = event['MyItem']

    result = multiplication_factor * item

    return {
        'statusCode': 200,
        'multiplied': result
    }
```

3. After you create your Lambda function, copy the function's ARN displayed in the upper-right corner of the page. The following is an example ARN, where *function-name* is the name of the Lambda function (in this case, `ProcessSingleItem`):

   ```
   arn:aws:lambda:region:123456789012:function:function-name
   ```

   You'll need to provide the function ARN in the state machine you created in Step 1.

4. Choose **Deploy** to deploy the changes.

Step 3: Run the state machine

When you run the state machine, the *Distributed Map state* starts four child workflow executions, where each execution processes three items, while one execution processes a single item.

The following example shows the data passed to one of the `ProcessSingleItem` function invocations inside a child workflow execution.

```
{
  "MyMultiplicationFactor": 7,
  "MyItem": 1
}
```

Given this input, the following example shows the output that is returned by the Lambda function.

```
{
  "statusCode": 200,
  "multiplied": 7
}
```

The following example shows the output JSON array for one of the child workflow executions.

```
[
  {
    "statusCode": 200,
    "multiplied": 7
  },
  {
    "statusCode": 200,
    "multiplied": 14
```

```
  },
  {
    "statusCode": 200,
    "multiplied": 21
  }
]
```

The state machine returns the following output that contains four arrays for the four child workflow executions. These arrays contain the multiplication results of the individual input items.

Finally, the state machine output is an array named `multiplied` that combines all the multiplication results returned for the four child workflow executions.

```
[
  [
    {
      "statusCode": 200,
      "multiplied": 7
    },
    {
      "statusCode": 200,
      "multiplied": 14
    },
    {
      "statusCode": 200,
      "multiplied": 21
    }
  ],
  [
    {
      "statusCode": 200,
      "multiplied": 28
    },
    {
      "statusCode": 200,
      "multiplied": 35
    },
    {
      "statusCode": 200,
      "multiplied": 42
    }
  ],
  [
```

```
    {
      "statusCode": 200,
      "multiplied": 49
    },
    {
      "statusCode": 200,
      "multiplied": 56
    },
    {
      "statusCode": 200,
      "multiplied": 63
    }
  ],
  [
    {
      "statusCode": 200,
      "multiplied": 70
    }
  ]
]
```

To combine all the multiplication results returned by the child workflow executions into a single output array, you can use the ResultSelector field. Define this field inside the *Distributed Map state* to find all the results, extract the individual results, and then combine them into a single output array named `multiplied`.

To use the `ResultSelector` field, update your state machine definition as shown in the following example.

```
{
  "StartAt": "Pass",
  "States": {
    ...
    ...
    "Map": {
      "Type": "Map",
      ...
      ...
      "ItemBatcher": {
        "MaxItemsPerBatch": 3,
        "BatchInput": {
          "MyMultiplicationFactor.$": "$.MyMultiplicationFactor"
        }
```

```
      },
      "ItemsPath": "$.MyItems",
      "ResultSelector": {
        "multiplied.$": "$..multiplied"
      }
    }
  }
}
```

The updated state machine returns a consolidated output array as shown in the following example.

```
{
  "multiplied": [7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
}
```

Starting a Step Functions workflow in response to events

You can execute an AWS Step Functions state machine in response to an event routed by an Amazon EventBridge rule to Step Functions as a target.

The following tutorial shows you how to configure a state machine as a target of an Amazon EventBridge rule. Whenever files are added to an Amazon Simple Storage Service (Amazon S3) bucket, the EventBridge rule will start the state machine.

A practical example of this approach could be a state machine that runs Amazon Rekognition analysis on image files added to the bucket to categorize and assign keywords.

In this tutorial, you start the execution of a `Helloworld` state machine by uploading a file to an Amazon S3 bucket. Then you review the example input of that execution to identify the information that is included in input from the Amazon S3 event notification delivered to EventBridge.

Prerequisite: Create a State Machine

Before you can configure a state machine as an Amazon EventBridge target, you must create the state machine.

- To create a basic state machine, use the Creating state machine that uses a Lambda function 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 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 uploaded to this bucket, EventBridge triggers an execution of your 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. Keep all the default selections on the page, and 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.

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. Scroll down the page to view the **Event Notifications** section, and then choose **Edit** in the **Amazon EventBridge** subsection.
5. Under **Send notifications to Amazon EventBridge for all events in this bucket**, choose **On**.
6. Choose **Save changes**.

   > **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**.

   > **Tip**
   >
   > Alternatively, in the navigation pane on the EventBridge console, choose **Rules** under **Buses**, and then choose **Create rule**.

2. Enter a **Name** for your rule (for example, *`S3Step Functions`*) and optionally enter a **Description** for the rule.
3. For **Event bus** and **Rule type**, keep the default selections.
4. Choose **Next**. This opens the **Build event pattern** page.
5. Scroll down to the **Event pattern** section, and do the following:

   a. For **Event source**, keep the default selection of **AWS events or EventBridge partner events**.

   b. For **AWS service**, choose **Simple Storage Service (S3)**.

   c. For **Event type**, choose **Amazon S3 Event Notification**.

   d. Choose **Specific event(s)**, and then choose **Object Created**.

   e. Choose **Specific bucket(s) by name** and enter the bucket name you created in Step 1 (*`username`*`-sfn-tutorial`) to store your files.

   f. Choose **Next**. This opens the **Select target(s)** page.

To create the target

1. In **Target 1**, keep the default selection of **AWS service**.
2. In the **Select a target** dropdown list, select **Step Functions state machine**.
3. In the **State machine** list, select the state machine that you created earlier (for example, `Helloworld`).
4. Keep all the default selections on the page, and choose **Next**. This opens the **Configure tags** page.
5. Choose **Next** again. This opens the **Review and create** page.
6. 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.

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, choose the bucket you created to store files (*`username`*`-sfn-tutorial`), and then choose **Upload**.
2. Add a file, for example *`test.png`*, 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.

```
{
    "version": "0",
    "id": "6c540ad4-0671-9974-6511-756fbd7771c3",
    "detail-type": "Object Created",
    "source": "aws.s3",
    "account": "123456789012",
    "time": "2023-06-23T23:45:48Z",
    "region": "us-east-2",
    "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": "200.0.100.11",
      "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.

> **Note**
>
> Although Amazon API Gateway can start a Step Functions execution by calling `StartExecution`, you must call [DescribeExecution](https://docs.aws.amazon.com/step-functions/latest/apireference/API_DescribeExecution.html) to get the result.

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 trusted entity** page, do the following:

    a. For **Trusted entity type**, keep the default selection of **AWS service**.

    b. For **Use case**, choose **API Gateway** from the dropdown list.
3. Select **API Gateway**, and then choose **Next**.
4. On the **Add permissions** page, choose **Next**.
5. (Optional) On the **Name, review, and create** page, enter details, such as the role name. For example, enter **`APIGatewayToStepFunctions`**.
6. Choose **Create role**.

    The IAM role appears in the list of roles.
7. 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 **Add permissions**, and then choose **Attach policies**.
3. On the **Attach Policy** page, search for `AWSStepFunctionsFullAccess`, choose the policy, and then choose **Add permissions**.

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](), and then choose **Create API**.
2. On the **Choose an API type** page, in the **REST API** pane, choose **Build**.
3. On the **Create REST API** page, select **New API**, and then enter ***`StartExecutionAPI`*** for the **API name**.
4. Keep **API endpoint type** as **Regional**, and then choose **Create API**.

To create a resource

1. On the **Resources** page of ***`StartExecutionAPI`***, choose **Create resource**.
2. On the **Create resource** page, enter **`execution`** for **Resource name**, and then choose **Create resource**.

To create a POST method

1. Choose the **/execution** resource, and then choose **Create method**.
2. For **Method type**, choose `POST`.
3. For **Integration type**, choose **AWS service**.
4. For **AWS Region**, choose a Region from the list.
5. For **AWS service**, choose **Step Functions** from the list.
6. Keep **AWS subdomain** blank.
7. For **HTTP method**, choose **POST** from the list.

> **Note**
>
> All Step Functions API actions use the HTTP POST method.

8. For **Action type**, select **Use action name**.
9. For **Action name**, enter **`StartExecution`**.
10. For **Execution role**, enter the role ARN of the IAM role that you created earlier, as shown in the following example.

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

11. Keep the default options for **Credential cache** and **Default timeout**, and then 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

1. On the **/execution - POST - Method Execution** page, choose the **Test** tab. You might need to choose the right arrow button to show the tab.
2. On the **/execution - POST - Method Test** tab, copy the following request parameters into the **Request body** section using the ARN of an existing state machine (or create a new state machine that uses a Lambda function), and then choose **Test**.

```
{
   "input": "{}",
   "name": "MyExecution",
   "stateMachineArn": "arn:aws:states:region:123456789012:stateMachine:HelloWorld"
}
```

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

> **Note**
>
> 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 mapping template in the **Integration request** tab, as shown in the following example.
>
> ```
> {
>     "input": "$util.escapeJavaScript($input.json('$'))",
>     "stateMachineArn": "$util.escapeJavaScript($stageVariables.arn)"
> }
> ```
>
> With this approach, you can specify ARNs of different state machines based on your development stage (for example, `dev`, `test`, and `prod`). For more information about specifying stage variables in a mapping template, see [`$stageVariables`] in the *API Gateway Developer Guide*.

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

```
{
   "executionArn":
 "arn:aws:states:region:123456789012:execution:HelloWorld:MyExecution",
   "startDate": 1486768956.878
}
```

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

To deploy your API

1. On the **Resources** page of ***StartExecutionAPI***, choose **Deploy API**.
2. For **Stage**, select **New stage**.
3. For **Stage name**, enter **`alpha`**.

4. (Optional) For **Description**, enter a description.
5. Choose **Deploy**.

To test your deployment

1. On the **Stages** page of ***StartExecutionAPI***, expand **alpha**, **/**, **/execution**, **POST**, and then choose the **POST** method.
2. Under **Method overrides**, choose the copy icon to copy your API's invoke URL. The full URL should look like the following example.

```
https://a1b2c3d4e5.execute-api.region.amazonaws.com/alpha/execution
```

3. From the command line, run the `curl` command using the ARN of your state machine, and then invoke the URL of your deployment, as shown in the following example.

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

The execution ARN and its epoch date are returned, as shown in the following example.

```
{"executionArn":"arn:aws:states:region:123456789012:execution:HelloWorld:MyExecution","star
```

> **Note**
>
> If you get a "Missing Authentication Token" error, make sure that the invoke URL ends with **/execution**.

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 Learn about Activities in Step Functions in Learn about state machines in Step Functions.

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*.

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.

1. In the Step Functions console, in the navigation pane on the left, choose **Activities**.
2. Choose **Create activity**.
3. Enter a **Name** for the activity, for example, *get-greeting*, and then choose **Create activity**.
4. When your activity task is created, make a note of its ARN, as shown in the following example.

```
arn:aws:states:region:123456789012:activity:get-greeting
```

Step 2: Create a state machine

Create a state machine that determines when your activity is invoked and when your worker should perform its primary work, collect its results, and return them. To create the state machine, you'll use the Code editor of Workflow Studio.

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**.
3. Choose **Create from blank**.
4. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.

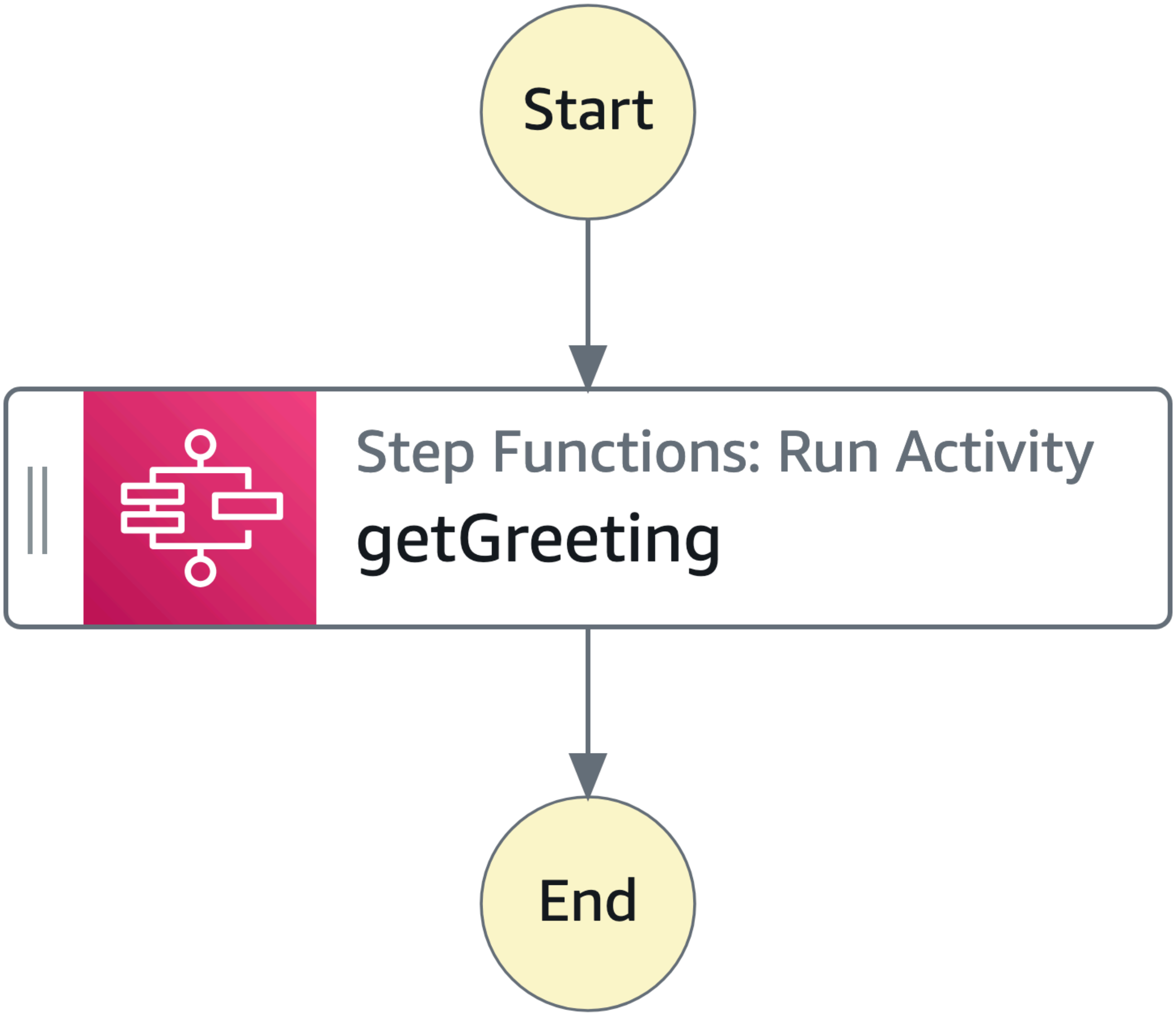
5. For this tutorial, you'll write the Amazon States Language (ASL) definition of your state machine in the code editor. To do this, choose **Code**.

6. Remove the existing boilerplate code and paste the following code. Remember to replace the example ARN in the `Resource` field with the ARN of the activity task that you created earlier in Step 1: Create an Activity.

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

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

7. On the Graph visualization, make sure the workflow graph for the ASL definition you added looks similar to the following graph.



8. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

   For this tutorial, enter the name **`ActivityStateMachine`**.

9. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

   For this tutorial, keep all the default selections in **State machine settings**.

If you've previously created an IAM role with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

10. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

    You can also choose **View role settings** to go back to **State machine configuration**.

> **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.

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. This example provides an implementation based on best practices, which you can use as a reference for your activity worker. The code implements a consumer-producer pattern with a configurable number of threads for pollers and activity workers.

To implement the worker

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

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


public class GreeterActivities {

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

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

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

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

            if (getActivityTaskResult.getTaskToken() != null) {
                try {
                    JsonNode json =
 Jackson.jsonNodeOf(getActivityTaskResult.getInput());
                    String greetingResult =
```

```
greeterActivities.getGreeting(json.get("who").textValue());
                    client.sendTaskSuccess(
                            new SendTaskSuccessRequest().withOutput(

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

Note

The `EnvironmentVariableCredentialsProvider` class in this example assumes that the `AWS_ACCESS_KEY_ID` (or `AWS_ACCESS_KEY`) and `AWS_SECRET_KEY` (or `AWS_SECRET_ACCESS_KEY`) environment variables are set. For more information about providing the required credentials to the factory, see [AWSCredentialsProvider](#) in the *AWS SDK for Java API Reference* and [Set Up AWS Credentials and Region for Development](#) in the *AWS SDK for Java Developer Guide*.
By default the AWS SDK will wait up to 50 seconds to receive data from the server for any operation. The `GetActivityTask` operation is a long-poll operation that will wait up to 60 seconds for the next available task. To prevent receiving a `SocketTimeoutException` 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 in [Step 1: Create an Activity](#).

Step 4: Run the state machine

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 **Start execution** dialog box is displayed.

2. In the **Start execution** dialog box, do the following:

   a. (Optional) Enter a custom execution name to override the generated default.

      Non-ASCII names and logging

      Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

   b. In the **Input** box, enter the following JSON input to run your workflow.

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

   c. Choose **Start execution**.

   d. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

      To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

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. On the Step Functions console, navigate to the *Execution Details* page.
6. When the execution completes, examine the results of your execution.
7. Stop the worker.

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 tutorial walks you though creating a state machine that calls a Lambda function. If you have completed that tutorial, skip to Step 2 and use the AWS Identity and Access Management (IAM) role that you previously created.

Step 1: Create an IAM role for Lambda

Both AWS Lambda and AWS Step Functions can run 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 18.x**.

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

d. For **Existing role**, select [the Lambda role that you created earlier](#).

> **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:region:123456789012:function:TestFunction1
```

4. Copy the following code for the Lambda function into the **Function code** section of the ***`TestFunction1`*** page.

```
function getRandomSeconds(max) {
    return Math.floor(Math.random() * Math.floor(max)) * 1000;
}
function sleep(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}
export const handler = async (event) => {
    if(getRandomSeconds(4) === 0) {
        throw new Error("Something went wrong!");
    }
    let wait_time = getRandomSeconds(5);
    await sleep(wait_time);
    return { 'response': true }
};
```

This code creates randomly timed failures, which will be used to generate example errors in your state machine that can be viewed and analyzed using X-Ray traces.

5. Choose **Save**.

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

In this step, you'll use the Step Functions console to create a state machine with three `Task` states. Each `Task` state will a reference one of your three Lambda functions.

1. Open the Step Functions console, choose **State machines** from the menu, then choose **Create state machine**.

   > **Important**
   >
   > Make sure that your state machine is under the same AWS account and Region as the Lambda functions you created earlier in Step 2 and Step 3.

2. Choose **Create from blank**.
3. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.
4. For this tutorial, you'll write the Amazon States Language (ASL) definition of your state machine in the Code editor. To do this, choose **Code**.
5. Remove the existing boilerplate code and paste the following code. In the Task state definition, remember to replace the example ARNs with the ARNs of the Lambda functions you created.

```
{
  "StartAt": "CallTestFunction1",
  "States": {
    "CallTestFunction1": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:region:123456789012:function:test-function1",
      "Catch": [
        {
          "ErrorEquals": [
            "States.TaskFailed"
          ],
          "Next": "AfterTaskFailed"
```

```
        }
      ],
      "Next": "CallTestFunction2"
    },
    "CallTestFunction2": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:region:123456789012:function:test-function2",
      "Catch": [
        {
          "ErrorEquals": [
            "States.TaskFailed"
          ],
          "Next": "AfterTaskFailed"
        }
      ],
      "Next": "CallTestFunction3"
    },
    "CallTestFunction3": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:region:123456789012:function:test-function3",
      "TimeoutSeconds": 5,
      "Catch": [
        {
          "ErrorEquals": [
            "States.Timeout"
          ],
          "Next": "AfterTimeout"
        },
        {
          "ErrorEquals": [
            "States.TaskFailed"
          ],
          "Next": "AfterTaskFailed"
        }
      ],
      "Next": "Succeed"
    },
    "Succeed": {
      "Type": "Succeed"
    },
    "AfterTimeout": {
      "Type": "Fail"
    },
    "AfterTaskFailed": {
```

```
      "Type": "Fail"
    }
  }
}
```

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.

6. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

   For this tutorial, enter the name **`TraceFunctions`**.

7. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

   For this tutorial, under **Additional configuration**, choose **Enable X-Ray tracing**. Keep all the other default selections in **State machine settings**.

   If you've previously created an IAM role with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

8. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

   You can also choose **View role settings** to go back to **State machine configuration**.

> **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.

Step 5: Run the state machine

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

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

The **New execution** page is displayed.

2. In the **Start execution** dialog box, do the following:

   a. (Optional) Enter a custom execution name to override the generated default.

      > **Non-ASCII names and logging**
      >
      > Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

   b. Choose **Start execution**.

   c. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

      To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

      Run several (at least three) executions.

3. 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.

4. 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`.

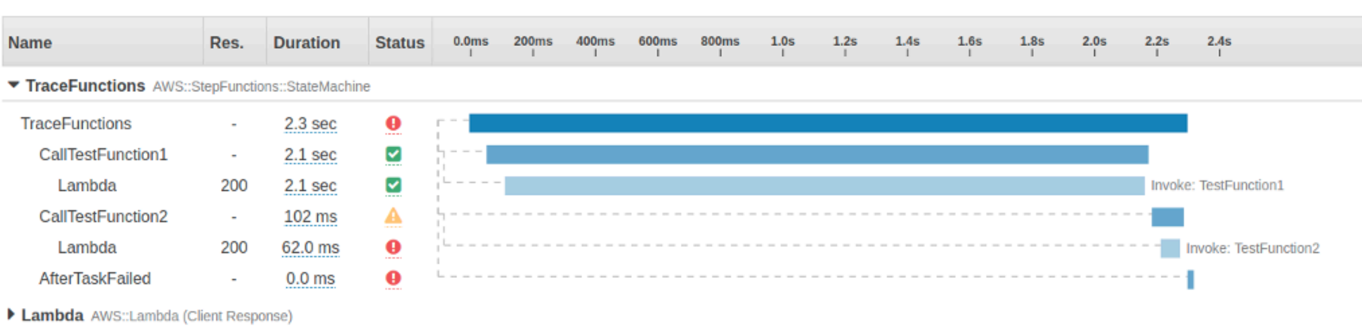
   The service map indicates the health of each node by coloring it based on the ratio of successful calls to errors and faults:

   - **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.

5. 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 [Trace Step Functions request data in AWS X-Ray](#)

Gather Amazon S3 bucket info using AWS SDK service integrations

This tutorial shows you how to perform an AWS SDK integration with Amazon Simple Storage Service. The state machine you create in this tutorial gathers information about your Amazon S3 buckets, then list your buckets along with version information for each bucket in the current region.

Step 1: Create the state machine

Using the Step Functions console, you'll create a state machine that includes a `Task` state to list all the Amazon S3 buckets in the current account and region. Then, you'll add another `Task` state that invokes the `HeadBucket` API to verify if the returned bucket is accessible in the current region. If the bucket isn't accessible, the `HeadBucket` API call returns the `S3.S3Exception` error. You'll include a `Catch` block to catch this exception and a `Pass` state as the fallback state.

1. Open the Step Functions console, choose **State machines** from the menu, then choose **Create state machine**.
2. Choose **Create from blank**.
3. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.
4. For this tutorial, you'll write the Amazon States Language (ASL) definition of your state machine in the Code editor. To do this, choose **Code**.
5. Remove the existing boilerplate code and paste the following state machine definition.

```
{
  "Comment": "A description of my state machine",
  "StartAt": "ListBuckets",
  "States": {
    "ListBuckets": {
      "Type": "Task",
      "Parameters": {},
      "Resource": "arn:aws:states:::aws-sdk:s3:listBuckets",
      "Next": "Map"
    },
    "Map": {
      "Type": "Map",
      "ItemsPath": "$.Buckets",
```

```
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "INLINE"
        },
        "StartAt": "HeadBucket",
        "States": {
          "HeadBucket": {
            "Type": "Task",
            "ResultPath": null,
            "Parameters": {
              "Bucket.$": "$.Name"
            },
            "Resource": "arn:aws:states:::aws-sdk:s3:headBucket",
            "Catch": [
              {
                "ErrorEquals": [
                  "S3.S3Exception"
                ],
                "ResultPath": null,
                "Next": "Pass"
              }
            ],
            "Next": "GetBucketVersioning"
          },
          "GetBucketVersioning": {
            "Type": "Task",
            "End": true,
            "Parameters": {
              "Bucket.$": "$.Name"
            },
            "ResultPath": "$.BucketVersioningInfo",
            "Resource": "arn:aws:states:::aws-sdk:s3:getBucketVersioning"
          },
          "Pass": {
            "Type": "Pass",
            "End": true,
            "Result": {
              "Status": "Unknown"
            },
            "ResultPath": "$.BucketVersioningInfo"
          }
        }
      },
      "End": true
```

```
      }
    }
  }
```

6. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

   For this tutorial, enter the name `Gather-S3-Bucket-Info-Standard`.

7. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

   Keep all the default selections in **State machine settings**.

   If you've previously created an IAM role with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

8. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

   You can also choose **View role settings** to go back to **State machine configuration**.

   > **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.

   In Step 2, you'll add the missing permissions to the state machine role.

Step 2: Add the necessary IAM role permissions

To gather information about the Amazon S3 buckets in your current region, you must provide your state machine the necessary permissions to access the Amazon S3 buckets.

1. On the state machine page, choose **IAM role ARN** to open the **Roles** page for the state machine role.
2. Choose **Add permissions** and then choose **Create inline policy.**

3. Choose the **JSON** tab, and then paste the following permissions into the JSON editor.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "s3:ListAllMyBuckets",
                "s3:ListBucket",
                "s3:GetBucketVersioning"
            ],
            "Resource": "*"
        }
    ]
}
```

4. Choose **Review policy**.
5. Under **Review policy**, for the policy **Name**, enter **`s3-bucket-permissions`**.
6. Choose **Create policy**.

Step 3: Run a Standard state machine execution

1. On the **Gather-S3-Bucket-Info-Standard** page, choose **Start execution**.
2. In the **Start execution** dialog box, do the following:

    a. (Optional) Enter a custom execution name to override the generated default.

    > **Non-ASCII names and logging**
    >
    > Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

    b. Choose **Start execution**.

c. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

   To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

Step 4: Run an Express state machine execution

1. Create an Express state machine using the state machine definition provided in Step 1. Make sure that you also include the necessary IAM role permissions as explained in Step 2.

   Tip

   To distinguish from the Standard machine you created earlier, name the Express state machine as **`Gather-S3-Bucket-Info-Express`**.

2. On the **Gather-S3-Bucket-Info-Standard** page, choose **Start execution**.
3. In the **Start execution** dialog box, do the following:

   a. (Optional) Enter a custom execution name to override the generated default.

      Non-ASCII names and logging

      Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

   b. Choose **Start execution**.

   c. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

Continue long-running workflows using Step Functions API (recommended)

AWS Step Functions is designed to run workflows with a finite duration and number of steps. Standard workflow executions have a maximum duration of one year and 25,000 events (see Step Functions service quotas).

For long-running executions, you can avoid reaching the hard quota by starting a new workflow execution from the `Task` state. You need to break your workflows up into smaller state machines which continue ongoing work in a new execution.

To start new workflow executions, you will call the `StartExecution` API action from your `Task` state and pass the necessary parameters.

Step Functions can start workflow executions by calling its own API as an integrated service. We recommend that you use this approach to avoid exceeding service quotas for long-running executions.

Step 1: Create a long-running state machine

Create a long-running state machine that you want to start from the `Task` state of a different state machine. For this tutorial, use the state machine that uses a Lambda function.

> **Note**
>
> Make sure to copy the name and Amazon Resource Name of this state machine in a text file for later use.

Step 2: Create a state machine to call the Step Functions API action

To start workflow executions from a Task state

1. Open the [Step Functions console](), choose **State machines** from the menu, then choose **Create state machine**.
2. Choose **Create from blank**.
3. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.
4. From the **Actions** tab, drag the **StartExecution** API action and drop it on the empty state labelled **Drag first state here**.
5. Choose the **StartExecution** state and do the following in the **Configuration** tab in [Design mode]():
   a. Rename the state to **`Start nested execution`**.
   b. For **Integration type**, choose **AWS SDK - new** from the dropdown list.
   c. In **API Parameters**, do the following:
      i. For `StateMachineArn`, replace the sample Amazon Resource Name with the ARN of your state machine. For example, enter the ARN of the [state machine that uses Lambda]().
      ii. For `Input` node, replace the existing placeholder text with the following value:

      ```
      "Comment": "Starting workflow execution using a Step Functions API action"
      ```

      iii. Make sure your inputs in **API Parameters** look similar to the following:

      ```
      {
        "StateMachineArn": "arn:aws:states:us-
      east-2:123456789012:stateMachine:LambdaStateMachine",
        "Input": {
          "Comment": "Starting workflow execution using a Step Functions API
       action",
          "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
        }
      ```
6. (Optional) Choose **Definition** on the [Inspector panel]() panel to view the automatically-generated [Amazon States Language]() (ASL) definition of your workflow.

> **Tip**
>
> You can also view the ASL definition in the Code editor of Workflow Studio. In the code editor, you can also edit the ASL definition of your workflow.

7. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

   For this tutorial, enter the name **`ParentStateMachine`**.

8. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

   For this tutorial, keep all the default selections in **State machine settings**.

   If you've previously created an IAM role with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

9. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

   You can also choose **View role settings** to go back to **State machine configuration**.

> **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.

Step 3: Update the IAM policy

To make sure your state machine has permissions to start the execution of the state machine that uses a Lambda function, you need to attach an inline policy to your state machine's IAM role. For more information, see Embedding Inline Policies in the *IAM User Guide*.

1. On the **ParentStateMachine** page, choose the **IAM role ARN** to navigate to the IAM **Roles** page for your state machine.

2. Assign an appropriate permission to the IAM role of the **ParentStateMachine** for it to be able to start execution of another state machine. To assign the permission, do the following:

   a. On the IAM **Roles** page, choose **Add permissions**, and then choose **Create inline policy**.

   b. On the **Create policy** page, choose the **JSON** tab.

   c. Replace the existing text with the following policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
      {
          "Effect": "Allow",
          "Action": [
              "states:StartExecution"
          ],
          "Resource": [
              "arn:aws:states:us-
east-1:123456789012:stateMachine:LambdaStateMachine"
          ]
      }
  ]
}
```

   d. Choose **Review policy**.

   e. Specify a name for the policy, and then choose **Create policy**.

Step 4: Run the state machine

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

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

   The **Start execution** dialog box is displayed.

2. In the **Start execution** dialog box, do the following:

   a. (Optional) Enter a custom execution name to override the generated default.

> **Non-ASCII names and logging**
>
> Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

b. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.

c. Choose **Start execution**.

d. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

   To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

3. Open the **LambdaStateMachine** page and notice a new execution triggered by the **ParentStateMachine**.

Using a Lambda function to continue a new execution in Step Functions

> **Tip**
>
> The following approach uses a Lambda function to start a new workflow execution. We **recommend** using a Step Functions Task state to start new workflow executions. See how in the following tutorial: **the section called "Continue long-running workflows using Step Functions API (recommended)"** .

You can create a state machine that uses a Lambda function to start a new execution before the current execution terminates. With this approach to continue ongoing work in a new execution, you can break large jobs into smaller workflows, or run a workflow indefinitely.

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

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

State	Purpose
`ConfigureCount`	A `Pass` state that configures the `count`, `index`, and `step` values that the `Iterator` Lambda function uses to step through iterations of work.
`Iterator`	A `Task` state that references the `Iterator` Lambda function.
`IsCountReached`	A `Choice` state that uses a Boolean value from the `Iterator` function to decide whether the state machine should continue the example work, or move to the `ShouldRestart` state.
`ExampleWork`	A `Pass` state that represents the `Task` state that would perform work in an actual implementation.
`ShouldRestart`	A `Choice` state that uses the `executionCount` value to decide whether it should end one execution and start another, or simply end.
`Restart`	A `Task` 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. The `Restart` state passes the decremented value of the count to the input of the new execution.

Prerequisites

Before you begin, go through the Creating a Step Functions state machine that uses Lambda tutorial to ensure that you're familiar with using Lambda and Step Functions together.

Step 1: Create a Lambda function to iterate a count

> **Note**
>
> If you have completed the Iterate a loop with a Lambda function in Step Functions tutorial, you can skip this step and use that Lambda function.

This section and the Iterate a loop with a Lambda function in Step Functions tutorial show how you can use a Lambda function to track a count, for example, 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. 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 16.x**.

   c. Keep all the default selections on the page, and then 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:region:123456789012:function:Iterator
      ```

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

```
exports.handler = function iterator (event, context, callback) {
```

```
  let index = event.iterator.index;
  let step = event.iterator.step;
  let count = event.iterator.count;

  index = 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 **Deploy** to deploy the code.

Test the Iterate Lambda function

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

To test your Lambda function

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

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

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

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

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

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

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

Step 2: Create a Restart Lambda function to start a new Step Functions execution

1. Open the [Lambda console](), and then choose **Create function**.
2. 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 `Restart`.
   b. For **Runtime**, choose **Node.js 16.x**.
4. Keep all the default selections on the page, and then 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:region:123456789012:function:Iterator
```

5. Copy the following code for the Lambda function into the **Code source** section of the ***Restart*** page in the Lambda console.

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

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

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

  let StateMachineArn = event.restart.StateMachineArn;
  event.restart.executionCount -= 1;
  event = JSON.stringify(event);

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

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

}
```

6. Choose **Deploy** to deploy the code.

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

The following excerpt shows the ShouldRestart[Choice](#) state. This state determines whether or not you should restart the execution.

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

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

Example Restart Task state

The following excerpt shows the Restart[Task](#) state. This state uses the Lambda function you created earlier to restart the execution, and to decrement the count to track the remaining number of executions to start.

```
"Restart": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:region:123456789012:function:Restart",
  "Next": "Done"
},
```

To create the state machine

1. Open the Step Functions console, choose **State machines** from the menu, then choose **Create state machine**.

   Important

   Make sure that your state machine is under the same AWS account and Region as the Lambda functions you created earlier in Step 1 and Step 2.

2. Choose **Create from blank**.
3. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.
4. For this tutorial, you'll write the [Amazon States Language](#) (ASL) definition of your state machine in the [Code editor](#). To do this, choose **Code**.
5. Remove the existing boilerplate code and paste the following code. Remember to replace the ARNs in this code with the ARNs of the Lambda functions you created.

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

```
            "Result": {
              "success": true
            },
            "ResultPath": "$.result",
            "Next": "Iterator"
        },
        "ShouldRestart": {
          "Type": "Choice",
          "Choices": [
            {
              "Variable": "$.restart.executionCount",
              "NumericGreaterThan": 0,
              "Next": "Restart"
            }
          ],
          "Default": "Done"
        },
        "Restart": {
          "Type": "Task",
          "Resource": "arn:aws:lambda:region:123456789012:function:Restart",
          "Next": "Done"
        },
        "Done": {
            "Type": "Pass",
            "End": true
        }
    }
}
```

6. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.

   For this tutorial, enter the name **`ContinueAsNew`**.

7. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

   For this tutorial, keep all the default selections in **State machine settings**.

   If you've previously created an IAM role with the correct permissions for your state machine and want to use it, in **Permissions**, select **Choose an existing role**, and then select a role from the list. Or select **Enter a role ARN** and then provide an ARN for that IAM role.

8. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

   You can also choose **View role settings** to go back to **State machine configuration**.

   > **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. Save the Amazon Resource Name (ARN) of this state machine in a text file. You'll need to provide the ARN while providing permission to the Lambda function to start a new Step Functions execution.

Step 4: Update the IAM Policy

To make sure 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*.

> **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.

```
{
 "Version": "2012-10-17",
 "Statement": [
     {
         "Sid": "VisualEditor0",
         "Effect": "Allow",
         "Action": [
             "states:StartExecution"
         ],
         "Resource": "arn:aws:states:us-east-2:123456789012:stateMachine:ContinueAsNew"
     }
 ]
```

```
}
```

Step 5: Run the state machine

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 **Start execution**.

   The **Start execution** dialog box is displayed.

2. In the **Start execution** dialog box, do the following:

   a. (Optional) Enter a custom execution name to override the generated default.

      Non-ASCII names and logging

      Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

   b. In the **Input** box, enter the following JSON input to run your workflow.

```
{
  "restart": {
    "StateMachineArn": "arn:aws:states:region:account-
id:stateMachine:ContinueAsNew",
    "executionCount": 4
  }
}
```

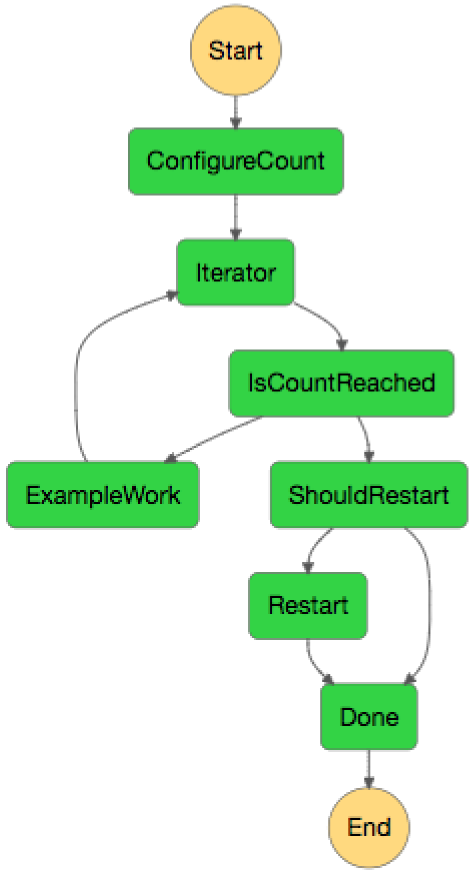
   c. Update the `StateMachineArn` field with the ARN for your `ContinueAsNew` state machine.

   d. Choose **Start execution**.

   e. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

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



As this execution completes, 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 started.

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.

Accessing cross-account AWS resources in Step Functions

With the cross-account access support in Step Functions, you can share resources configured in different AWS accounts. In this tutorial, we walk you through the process of accessing a cross-account Lambda function defined in an account called **Production**. This function is invoked from a state machine in an account called **Development**. In this tutorial, the **Development** account is referred to as the *source account* and the **Production** account is the *target account* containing the target IAM role.

To start, in your `Task` state's definition, you specify the target IAM role the state machine must assume before invoking the cross-account Lambda function. Then, modify the trust policy in the target IAM role to allow the source account to assume the target role temporarily. Also, to call the AWS resource, define the appropriate permissions in the target IAM role. Finally, update the source account's execution role to specify the required permission to assume the target role.

You can configure your state machine to assume an IAM role for accessing resources from multiple AWS accounts. However, a state machine can assume only one IAM role at a given time based on the `Task` state's definition.

> **Note**
>
> Currently, cross-Region AWS SDK integration and cross-Region AWS resource access aren't available in Step Functions.

Prerequisites

- This tutorial uses the example of a Lambda function for demonstrating how to set up cross-account access. You can use any other AWS resource, but make sure you've configured the resource in a different account.

> **Important**
>
> IAM roles and resource-based policies delegate access across accounts only within a single partition. For example, assume that you have an account in US West (N. California) in the standard `aws` partition. You also have an account in China (Beijing) in the `aws-cn` partition. You can't use an Amazon S3 resource-based policy in your account in China (Beijing) to allow access for users in your standard `aws` account.

- Make a note of the cross-account resource's Amazon Resource Name (ARN) in a text file. Later in this tutorial, you'll provide this ARN in your state machine's `Task` state definition. The following is an example of a Lambda function ARN:

```
arn:aws:lambda:us-east-2:account-id:function:functionName
```

- Make sure you've created the target IAM role that the state machine needs to assume.

Step 1: Update the Task state definition to specify the target role

In the `Task` state of your workflow, add a `Credentials` field containing the identity the state machine must assume before invoking the cross-account Lambda function.

The following procedure demonstrates how to access a cross-account Lambda function called `Echo`. You can call any AWS resource by following these steps.

1. Open the Step Functions console and choose **Create state machine**.
2. On the **Choose authoring method** page, choose **Design your workflow visually** and keep all the default selections.
3. To open Workflow Studio, choose **Next**.
4. On the **Actions** tab, drag and drop a `Task` state on the canvas. This invokes the cross-account Lambda function that's using this `Task` state.
5. On the **Configuration** tab, do the following:

a. Rename the state to **`Cross-account call`**.

b. For **Function name**, choose **Enter function name**, and then enter the Lambda function ARN in the box. For example, `arn:aws:lambda:us-east-2:111122223333:function:`*`Echo`*.

c. For **Provide IAM role ARN**, specify the target IAM role ARN. For example, `arn:aws:iam::111122223333:role/LambdaRole`.

> **Tip**
>
> Alternatively, you can also specify a [reference path](#) to an existing key-value pair in the state’s JSON input that contains the IAM role ARN. To do this, choose **Get IAM role ARN at runtime from state input**. For an example of specifying a value by using a reference path, see [Specifying JSONPath as IAM role ARN](#).

6. Choose **Next**.
7. On the **Review generated code** page, choose **Next**.
8. On the **Specify state machine settings** page, specify details for the new state machine, such as a name, permissions, and logging level.
9. Choose **Create state machine**.
10. Make a note of the state machine's IAM role ARN and the state machine ARN in a text file. You'll need to provide these ARNs in the target account's trust policy.

Your `Task` state definition should now look similar to the following definition.

```
{
  "StartAt": "Cross-account call",
  "States": {
    "Cross-account call": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Credentials": {
        "RoleArn": "arn:aws:iam::111122223333:role/LambdaRole"
      },
      "Parameters": {
        "FunctionName": "arn:aws:lambda:us-east-2:111122223333:function:Echo",
      },
      "End": true
```

```
    }
  }
}
```

Step 2: Update the target role's trust policy

The IAM role must exist in the target account and you must modify its trust policy to allow the source account to assume this role temporarily. Additionally, you can control who can assume the target IAM role.

After you create the trust relationship, a user from the source account can use the AWS Security Token Service (AWS STS) AssumeRole API operation. This operation provides temporary security credentials that enable access to AWS resources in a target account.

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. On the navigation pane of the console, choose **Roles** and then use the Search box to search for the target IAM role. For example, *LambdaRole*.
3. Choose the **Trust relationships** tab.
4. Choose **Edit trust policy** and paste the following trust policy. Make sure to replace the AWS account number and IAM role ARN. The `sts:ExternalId` field further controls who can assume the role. The state machine's name must include only characters that the AWS Security Token Service `AssumeRole` API supports. For more information, see AssumeRole in the *AWS Security Token Service API Reference*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Principal": {
        "AWS": "arn:aws:iam::account-id:role/ExecutionRole"  // The source
 account's state machine execution role ARN
      },
      "Condition": {  // Control which account and state machine can assume the
 target IAM role
        "StringEquals": {
          "sts:ExternalId": "arn:aws:states:region:account-
id:stateMachine:testCrossAccount"   //// ARN of the state machine that will assume
 the role.
```

```
          }
        }
      }
    ]
  }
```

5. Keep this window open and proceed to the next step for further actions.

Step 3: Add the required permission in the target role

Permissions in the IAM policies determine whether a specific request is allowed or denied. The target IAM role must have the correct permission to invoke the Lambda function.

1. Choose the **Permissions** tab.
2. Choose **Add permissions** and then choose **Create inline policy**.
3. Choose the **JSON** tab and replace the existing content with the following permission. Make sure to replace your Lambda function ARN.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:111122223333:function:Echo"  // The
 cross-account AWS resource being accessed
    }
  ]
}
```

4. Choose **Review policy**.
5. On the **Review policy** page, enter a name for the permission, and then choose **Create policy**.

Step 4: Add permission in execution role to assume the target role

Step Functions doesn’t automatically generate the AssumeRole policy for all cross-account service integrations. You must add the required permission in the state machine's execution role to allow it to assume a target IAM role in one or more AWS accounts.

1. Open your state machine's execution role in the IAM console at https://console.aws.amazon.com/iam/. To do this:

   a. Open the state machine that you created in Step 1 in the source account.

   b. On the **State machine detail** page, choose **IAM role ARN**.

2. On the **Permissions** tab, choose **Add permissions** and then choose **Create inline policy**.

3. Choose the **JSON** tab and replace the existing content with the following permission. Make sure to replace your Lambda function ARN.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Resource": "arn:aws:iam::111122223333:role/LambdaRole"  // The target role
 to be assumed
    }
  ]
}
```

4. Choose **Review policy**.

5. On the **Review policy** page, enter a name for the permission, and then choose **Create policy**.

Workshops for learning Step Functions

Workshop: The Step Functions Workshop

In this workshop, you will learn to use the primary features of Step Functions while building workflows. A series of interactive modules start by introducing you to basic workflows, task states, and error handling. You can continue to learn choice states for branch logic, map states for processing arrays, and parallel states for running multiple branches in parallel.

Workshop: Large-scale Data Processing with Step Functions

Learn how serverless technologies such as Step Functions and Lambda can simplify management and scaling, offload undifferentiated tasks, and address the challenges of large-scale distributed data processing. Along the way, you will work with distributed map for high concurrency processing. The workshop also presents best practices for optimizing your workflows, and practical use cases for claims processing, vulnerability scanning, and Monte Carlo simulation.

Deploy a state machine using a starter template for Step Functions

To deploy state machines for a variety of example use cases and patterns, you can choose one of the following starter templates in the AWS Step Functions console. These starter templates are ready-to-run sample projects that automatically create the workflow prototype and definition, and all related AWS resources for the project.

You can use these sample projects to deploy and run them as is, or use the workflow prototypes to build on them. If you build upon these projects, Step Functions creates the workflow prototype, but doesn't deploy the resources listed in the workflow definition.

When you deploy the sample projects, they provision a fully functional state machine, and create the related resources for the state machine to run. When you create a sample project, Step Functions uses AWS CloudFormation to create the related resources referenced by the state machine.

List of starter templates

- Perform selective checkpointing using Standard and Express workflows
- Build an AWS CodeBuild project using Step Functions
- Preprocess data and train a machine learning model with Amazon SageMaker AI
- Orchestrate AWS Lambda functions with Step Functions
- Start an Athena query and send a results notification
- Execute queries in sequence and parallel using Athena
- Query large datasets using an AWS Glue crawler
- Keep data in a target table updated with AWS Glue and Athena
- Create and manage an Amazon EKS cluster with a node group
- Interact with an API managed by API Gateway
- Call a microservice running on Fargate using API Gateway integration
- Send a custom event to an EventBridge event bus
- Invoke Synchronous Express Workflows through API Gateway
- Run an ETL/ELT workflow using Step Functions and the Amazon Redshift API
- Manage a batch job with AWS Batch and Amazon SNS
- Fan out batch jobs with Map state
- Run an AWS Batch job with Lambda

Manage a container task with Amazon ECS and 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.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.

3. Choose how to use the template:

   a. **Run a demo –** creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it –** provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Transfer data records with Lambda, DynamoDB, and Amazon SQS

This sample project demonstrates how to iteratively read items from an Amazon DynamoDB table and send these items to an Amazon SQS queue using a Step Functions state machine. Deploying this sample project will create a Step Functions state machine, a DynamoDB table, an AWS Lambda function, and an Amazon SQS queue.

In this project, Step Functions uses the Lambda function to populate the DynamoDB table. The state machine also uses a `for` loop to read each of the entries, and then sends each entry to an Amazon SQS queue.

Step 1: Create the state machine

1. Open the [Step Functions console](#) and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Poll for job status with Lambda and 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 Integrating services with Step Functions.

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 Integrating services with Step Functions.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

    > **Note**
    >
    > *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Create a task timer with Lambda and 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 workflow state 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 Integrating services with Step Functions.

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 Integrating services with Step Functions.

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

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:
   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.
   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Create a callback pattern example with Amazon SQS, Amazon SNS, and 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.

To learn how to implement the callback pattern in Step Functions, see Wait for a Callback with Task Token.

For more information about how AWS Step Functions can control other AWS services, see Integrating services with Step Functions.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Manage an Amazon EMR job

This sample project demonstrates Amazon EMR and AWS Step Functions integration. The project creates an Amazon EMR cluster, adds multiple steps and runs 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.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:
    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.
    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Run an EMR Serverless job

This sample project demonstrates how to create and start an EMR Serverless application and run multiple jobs within it.

This sample project creates the state machine, the supporting AWS resources, and configures the related IAM permissions. Explore this sample project to learn about running EMR Serverless jobs using Step Functions state machines, or use it as a starting point for your own projects.

> **Important**
>
> EMR Serverless does not have a free pricing tier. Running the sample project will incur costs. You can find pricing information on the [Amazon EMR Serverless pricing](#) page. In addition, the availability of EMR Serverless service integration is subject to the availability of EMR Serverless APIs. Because of this, this sample project might not work

correctly or be available in some AWS Regions. See the Other considerations topic for information about availability of EMR Serverless in AWS Regions.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo –** creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it –** provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Start a workflow within a workflow with Step Functions and Lambda

This sample project demonstrates how to use an AWS Step Functions state machine to start other state machine executions. For information about starting state machine executions from another state machine, see Start workflow executions from a task state in Step Functions.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Process data from a queue with a Map state in Step Functions

In this sample workflow, a Map workflow state state processes data from a queue, sending messages to subscribers and storing them in a database.

Step Functions uses an optimized integration to pull messages from an Amazon SQS queue. When messages are available, a Choice state passes an array of JSON messages to a Map state for processing. For each message, the state machine writes the message to DynamoDB, removes the message from the queue, and publishes the message to an Amazon SNS topic.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:
    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.
    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Subscribe to the Amazon SNS topic

> **Tip**
>
> Subscribe to the Amazon SNS topic and add items to the Amazon SQS queue **before** you run your state machine.

1. Open the [Amazon SNS console](https://console.aws.amazon.com/sns/home).
2. Choose **Topics** and find the topic that was created by the sample project.
3. Choose **Create subscription**, and for **Protocol**, choose **Email**.
4. Under **Endpoint**, enter your email address to subscribe to the topic.
5. Choose **Create subscription**.
6. Confirm the subscription in your email to activate the subscription.

Step 3: Add messages to the Amazon SQS queue

1. Open the [Amazon SQS console](https://console.aws.amazon.com/sqs/home).
2. Choose the queue that was created by the sample project.
3. Choose **Send and receive messages**, enter a message and choose **Send message**. Repeat this step to add several messages to the queue.

Step 4: Run the state machine

> **Tip**
>
> Queues in Amazon SNS are eventually consistent. You may need to wait a few minutes after sending messages to the queue before running your state machine.

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Process a CSV file from Amazon S3 using a Distributed Map

This sample project demonstrates how you can use the *Distributed Map state* to iterate over 10,000 rows of a CSV file that is generated using a Lambda function. The CSV file contains shipping information of customer orders and is stored in an Amazon S3 bucket. The Distributed Map iterates over a batch of 10 rows in the CSV file for data analysis.

The Distributed Map contains a Lambda function to detect any delayed orders. The Distributed Map also contains an Inline Map to process the delayed orders in a batch and returns these delayed orders in an array. For each delayed order, the Inline Map sends a message to an Amazon SQS queue. Finally, this sample project stores the Map Run results to another Amazon S3 bucket in your AWS account.

With Distributed Map, you can run up to 10,000 parallel child workflow executions at a time. In this sample project, the maximum concurrency of Distributed Map is set at 1000 that limits it to 1000 parallel child workflow executions.

This sample project creates the state machine, the supporting AWS resources, and configures the related IAM permissions. Explore this sample project to learn about using the Distributed Map for orchestrating large-scale, parallel workloads, or use it as a starting point for your own projects.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:
    a. **Run a demo –** creates a read-only state machine. After review, you can create the workflow and all related resources.
    b. **Build on it –** provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Process data in an Amazon S3 bucket with Distributed Map

This sample project demonstrates how you can use the *Distributed Map state* to process large-scale data, for example, analyze historical weather data and identify the weather station that has the highest average temperature on the planet each month. The weather data is recorded in over 12,000 CSV files, which in turn are stored in an Amazon S3 bucket.

This sample project includes two *Distributed Map state*s named **Distributed S3 copy NOA Data** and **ProcessNOAAData**. **Distributed S3 copy NOA Data** iterates over the CSV files in a public Amazon S3 bucket named **noaa-gsod-pds** and copies them to an Amazon S3 bucket in your AWS account. **ProcessNOAAData** iterates over the copied files and includes a Lambda function that performs the temperature analysis.

The sample project first checks the contents of the Amazon S3 bucket with a call to the ListObjectsV2 API action. Based on the number of keys returned in response to this call, the sample project takes one of the following decisions:

- If the key count is more than or equal to 1, the project transitions to the **ProcessNOAAData** state. This *Distributed Map state* includes a Lambda function named **TemperatureFunction** that finds the weather station that had the highest average temperature each month. This function returns a dictionary with `year-month` as the key and a dictionary that contains information about the weather station as the value.

- If the returned key count doesn't exceed 1, the **Distributed S3 copy NOA Data** state lists all objects from the public bucket **noaa-gsod-pds** and iteratively copies the individual objects to another bucket in your account in batches of 100. An Inline Map performs the iterative copying of the objects.

  After all objects are copied, the project transitions to the **ProcessNOAAData** state for processing the weather data.

The sample project finally transitions to a reducer Lambda function that performs a final aggregation of the results returned by the **TemperatureFunction** function and writes the results to an Amazon DynamoDB table.

With Distributed Map, you can run up to 10,000 parallel child workflow executions at a time. In this sample project, the maximum concurrency of **ProcessNOAAData** Distributed Map is set at 3000 that limits it to 3000 parallel child workflow executions.

This sample project creates the state machine, the supporting AWS resources, and configures the related IAM permissions. Explore this sample project to learn about using the Distributed Map for orchestrating large-scale, parallel workloads, or use it as a starting point for your own projects.

> **Important**
>
> This sample project is only available in the US East (N. Virginia) Region.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Train a machine learning model using Amazon SageMaker AI

This sample project demonstrates how to use SageMaker AI and AWS Step Functions to train a machine learning model and how to batch transform a test dataset.

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 AI service integration.

For more information about SageMaker AI and Step Functions service integrations, see the following:

- Integrating services with Step Functions
- Create and manage Amazon SageMaker AI jobs with Step Functions

> **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 AI Pricing.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

   > **Note**
   >
   > *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Tune the hyperparameters of a machine learning model in SageMaker AI

This sample project demonstrates using SageMaker AI to tune the hyperparameters of a machine learning model, and to batch transform a test dataset.

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 AI service integration. 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 AI.

For more information about SageMaker AI and Step Functions service integrations, see the following:

- Integrating services with Step Functions
- Create and manage Amazon SageMaker AI jobs with Step Functions

> **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 AI Pricing.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Perform AI prompt-chaining with Amazon Bedrock

This sample project demonstrates how you can integrate with Amazon Bedrock to perform AI prompt-chaining and build high-quality chatbots using Amazon Bedrock. The project chains together some prompts and resolves them in the sequence in which they're provided. Chaining of these prompts augments the ability of the language model being used to deliver a highly-curated response.

This sample project creates the state machine, the supporting AWS resources, and configures the related IAM permissions. Explore this sample project to learn about using Amazon Bedrock optimized service integration with Step Functions state machines, or use it as a starting point for your own projects.

Prerequisites

This sample project uses the Cohere Command large language model (LLM). To successfully run this sample project, you must add access to this LLM from the Amazon Bedrock console. To add the model access, do the following:

1. Open the Amazon Bedrock console.
2. On the navigation pane, choose **Model access**.
3. Choose **Manage model access**.
4. Select the check box next to **Cohere**.
5. Choose **Request access**. The **Access status** for **Cohere** model shows as **Access granted**.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo –** creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it –** provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Process high-volume messages from Amazon SQS with Step Functions 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:

- Choosing workflow type in Step Functions
- Integrating services with Step Functions
- Step Functions service quotas

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Trigger the state machine 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.


```
{
      "input":
 "QW5kIGxpa2UgdGhlIGJhc2VsZXNzIGZhYnJpYyBvZiB0aGlzIHZpc2lvbiwgVGhlIGNsb3VkLWNhcHBlZCB0b3dlc

  91cyBwYWxhY2VzLCBUaGUgc29sZW1uIHRlbXBsZXMsIHRoZSBncmVhdCBnbG9iZSBpdHNlbGbigJQgWWVhLCBhbGw

  ZXJpdOKAlHNoYWxsIGRpc3NvbHZlLCBBbmQgbGlrZSB0aGlzIGluc3Vic3RhbnRpYWwgcGFnZWFudCBmYWRlZCwgT

  FjayBiZWhpbmQuIFdlIGFyZSBzdWNoIHN0dWZmIEFzIGRyZWFtcyBhcmUgbWFkZSBvbiwgYW5kIG91ciBsaXR0bGU

  ZGVkIHdpdGggYSBzbGVlcC4gU2lyLCBJIGFtIHZleGVkLiBCZWFyIHdpdGggbXkgd2Vha25lc3MuIE15IG9sZCBic

  xlZC4gQmUgbm90IGRpc3R1cmJlZCB3aXRoIG15IGluZmlybWl0eS4gSWYgeW91IGJlIHBsZWFzZWQsIHJldGlyZSB

  QW5kIHRoZXJlIHJlcG9zZS4gQSB0dXJuIG9yIHR3byBJ4oCZbGwgd2FsayBUbyBzdGlsbCBteSBiZWF0aW5nIG1pb
}
```


5. Choose **Close**.
6. Open the [Step Functions console](https://console.aws.amazon.com/states/home).
7. Go to your [Amazon CloudWatch Logs log group](https://console.aws.amazon.com/cloudwatch/home) and inspect the logs. The name of the log group will look like **example-ExpressLogGroup-wJalrXUtnFEMI**.

Perform selective checkpointing using Standard and 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:

- Choosing workflow type in Step Functions
- Start workflow executions from a task state in Step Functions
- Integrating services with Step Functions

Step 1: Create the State Machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Build an AWS CodeBuild project using Step Functions

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 based on the results.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Preprocess data and train a machine learning model with Amazon SageMaker AI

This sample project demonstrates how to use SageMaker AI and AWS Step Functions to preprocess data and train a machine learning model.

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 AI service integration.

For more information about SageMaker AI and Step Functions service integrations, see the following:

- Integrating services with Step Functions
- Create and manage Amazon SageMaker AI jobs with Step Functions

> **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 AI Pricing.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo –** creates a read-only state machine. After review, you can create the workflow and all related resources.

b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

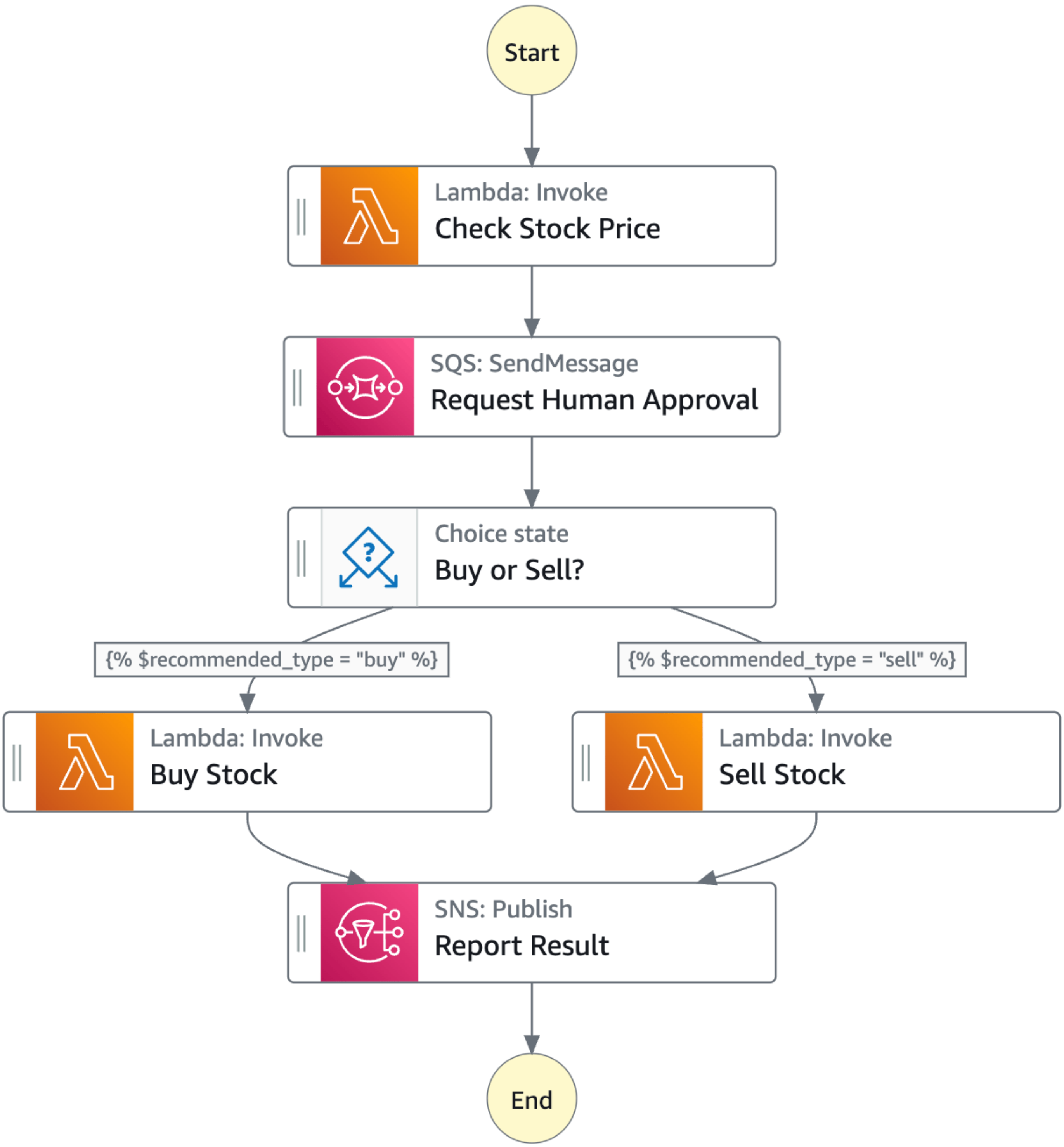
You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Orchestrate AWS Lambda functions with Step Functions

The **Orchestrate Lambda functions** template uses several Lambda functions in a sample stock trading workflow. One function checks a stock price, then a human is prompted to choose to buy or sell the stock. A choice state selects the next function based on the `recommended_type` variable to complete the purchase or sale. After either function finishes, the result of the trade is then published before reaching the end of the workflow.

To implement the human approval step, the workflow execution pauses until a unique TaskToken is returned. In this project, the workflow passes a message with the task token to an Amazon SQS queue. The message triggers another Lambda function that's configured to handle a callback based on the payload of the message. The workflow pauses until it receives the task token back from a

SendTaskSuccess API call. For more information about task tokens, see Wait for a Callback with Task Token.



Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:
   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.
   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

For more information about Step Functions service integrations, see Integrating services with Step Functions.

Start an Athena query and send a results notification

This sample project demonstrates how to use Step Functions and Amazon Athena to start an Athena query and send a notification with query results using Standard workflows.

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 and returns the results using an SNS topic.

For more information about Athena and Step Functions service integrations, see the following:

- Integrating services with Step Functions
- Run Athena queries with Step Functions

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Execute queries in sequence and parallel using Athena

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.

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.

Step 1: Create the state machine

1. Open the [Step Functions console]() and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Query large datasets using an AWS Glue crawler

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.

In this project, the Step Functions state machine invokes 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.

Step 1: Create the state machine

1. Open the [Step Functions console](https://console.aws.amazon.com/states/home?region=us-east-1#/) and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Keep data in a target table updated with AWS Glue and Athena

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.

In this project, the Step Functions state machine calls AWS Glue Catalog to verify if a target table exists in an Amazon S3 Bucket. If no table is found one, it will create a new table. Then, Step Functions runs an Athena query 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 into the target table.

Step 1: Create the state machine

1. Open the [Step Functions console]() and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.

3. Choose how to use the template:

   a. **Run a demo –** creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it –** provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Create and manage an Amazon EKS cluster with a node group

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.

For more information about Step Functions and Step Functions service integrations, see the following:

- Integrating services with Step Functions
- Create and manage Amazon EKS clusters with Step Functions

> **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.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo –** creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it –** provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

   > **Note**
   >
   > *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.

2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Interact with an API managed by API Gateway

This sample project demonstrates how to use Step Functions to make a call to API Gateway and checks whether the call succeeded.

For more information about API Gateway and Step Functions service integrations, see the following:

- Integrating services with Step Functions
- Create API Gateway REST APIs with Step Functions

Step 1: Create the state

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

   Note

   Standard charges apply for services deployed to your account.

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

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.

For more information about API Gateway and Step Functions service integrations, see the following:

- Integrating services with Step Functions
- Create API Gateway REST APIs with Step Functions

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo –** creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Send a custom event to an EventBridge event bus

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).

For more information about Step Functions and Step Functions service integrations, see the following:

- [Integrating services with Step Functions]
- [Add EventBridge events with Step Functions]

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.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

   Note

   Standard charges apply for services deployed to your account.

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Invoke Synchronous Express Workflows through API Gateway

This sample project demonstrates how to invoke Synchronous Express Workflows through Amazon API Gateway to manage an employee database.

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 in Step Functions.

> **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.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Run an ETL/ELT workflow using Step Functions and the Amazon Redshift 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.

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 guides:

- Integrating services with Step Functions
- Using the Amazon Redshift Data API
- Amazon Redshift Data API service
- Creating a Step Functions state machine that uses Lambda

For more information about IAM policies for Lambda and Amazon Redshift, see the following guides:

- IAM policies for calling AWS Lambda
- 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.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Manage a batch job with AWS Batch and 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.

Step 1: Create the state machine

1. Open the [Step Functions console]() and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

a. **Run a demo –** creates a read-only state machine. After review, you can create the workflow and all related resources.

b. **Build on it –** provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Fan out batch jobs with Map state

This sample project demonstrates how to use Step Functions's Map workflow state state to fan out AWS Batch jobs.

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 workflow state state.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

   a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

   b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Run an AWS Batch job 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.

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 or success of the previous one.

Step 1: Create the state machine

1. Open the Step Functions console and choose **Create state machine**.
2. Choose **Create from template** and find the related starter template. Choose **Next** to continue.
3. Choose how to use the template:

    a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

    b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)
4. Choose **Use template** to continue with your selection.

> **Note**
>
> *Standard charges apply for services deployed to your account.*

Step 2: Run the demo state machine

If you chose the **Run a demo** option, all related resources will be deployed and ready to run. If you chose the **Build on it** option, you might need to set placeholder values and create additional resources before you can run your custom workflow.

1. Choose **Deploy and run**.
2. Wait for the AWS CloudFormation stack to deploy. This can take up to 10 minutes.
3. After the **Start execution** option appears, review the **Input** and choose **Start execution**.

Congratulations!

You should now have a running demo of your state machine. You can choose states in the **Graph view** to review input, output, variables, definition, and events.

Developing workflows with Step Functions

We recommend starting to build workflows in the Step Functions console and Workflow Studio visual editor. You can start from a blank canvas or choose starter templates for common scenarios.

Building your workflows require the following tasks:

- Defining your workflow
- Running and debugging your workflow
- Deploying your workflow

You define a state machine in Amazon States Language. You can manually create your Amazon States Language definitions, but Workflow Studio will be featured in tutorials. With Workflow Studio, you can define, your machine definition, visualize and edit the steps, run and debug your workflow, and view the results all from within the Step Functions console.

> **Working with Workflow Studio in Visual Studio Code**
>
> With the AWS toolkit, you can use Workflow Studio from within VS Code to visualize, build, and even test individual states in your state machines. You provide state inputs and set variables, start the test, then you can see how your data is transformed. You can adjust the workflow and re-test. When finished, you can apply the changes to update the state machine. For more information, see Working with Workflow Studio in the AWS Toolkit for Visual Studio Code.

You can also use many Step Functions features from the AWS Command Line Interface (AWS CLI). For example, you can create a state machine and 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 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.

> **Validating state machine definitions**
>
> You can use the API to **validate** state machines and find potential problems before creating your workflow.
> To learn more about validating workflows, see ValidateStateMachineDefinition in the Step Functions API Reference.

To get started with minimal setup, you can follow the Creating a Lambda State Machine tutorial, which shows you how to define a workflow with a single step that calls a Lambda function, then run the workflow, and view the results.

Defining your workflow

The first step in developing your workflow is defining the steps in Amazon States Language. Depending on your preference and tool, you can define your Step Functions state machines in JSON, YAML, or as a stringified Amazon States Language (ASL) definition.

The following table shows ASL-based definition format support by tool.

AWS Tool	Supported format(s)
Step Functions Console	JSON
HTTPS Service API	Stringified ASL
AWS CLI	Stringified ASL
Step Functions Local	Stringified ASL
AWS Toolkit for Visual Studio Code	JSON, YAML
AWS SAM	JSON, YAML
AWS CloudFormation	JSON, YAML, Stringified ASL

YAML single line comments in the state machine definition of a template will not be carried forward into the created resource's definition. If you need to persist a comment, you should use the `Comment` property within the state machine definition. For information, see State machine structure.

With AWS CloudFormation and AWS SAM, you can upload your state machine definitions to Amazon S3 (JSON or YAML format) and provide the definition's Amazon S3 location in the template. For 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

```
{
  "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"
                }
              ]
            }
          }
        ]
      }
    }
  },
  "Outputs": {
    "StateMachineArn": {
      "Value": {
        "Ref": "MyStateMachine"
      }
    }
  }
}
```

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": "{\n  \"StartAt\": \"HelloWorld\",\n  \"States\": {\n
 \"HelloWorld\": {\n      \"Type\": \"Pass\",\n      \"End\": true\n    }\n  }\n}"
      }
    },
    "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"
      }
    }
  }
}
```

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:
    Value:
      Ref: MyStateMachine
```

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: MyStateMachinele
```

Develop workflows with 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. See the reference documentation for the current AWS SDKs and Tools for Amazon Web Services.

Develop workflows through HTTPS requests

Step Functions provides service operations that are accessible through HTTPS requests. You can use these operations to communicate directly with Step Functions from your own libraries. 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 details see the AWS Step Functions API Reference.

Develop workflows with the AWS Step Functions Data Science SDK

Data scientists can create workflows that process and publish machine learning models using SageMaker AI and Step Functions. You can also create multi-step machine learning workflows in Python that orchestrate AWS infrastructure at scale. 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. For more information, see: AWS Step Functions Data Science Project on Github, data science SDK documentation, and example Jupyter notebooks and SageMaker AI examples on GitHub.

Running and debugging your workflows

You can start workflows in a number of ways, including from the console, an API call (for example, from a Lambda function), from Amazon EventBridge and EventBridge Scheduler, from another Step Functions state machine. Running workflows can connect to third party services, use AWS SDKs, and manipulate data while running. Various tools exist to both run and debug the execution steps and data flowing through your state machine. The following sections provide additional resources for running and debugging your workflows.

To learn more about the ways to start state machine executions, see *Starting state machines*.

Choose an endpoint to run your workflows

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 do not 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 [AWS Step Functions Regions and Endpoints]() in the *AWS General Reference*.

Development with VS Code

With the AWS toolkit, you can use Workflow Studio from within VS Code to visualize, build, and even test individual states in your state machines. You can also use your SAM and CloudFormation definition substitutions. You provide state inputs and set variables, start the test, then you can see how your data is transformed. In the State definition tab, you can adjust the workflow and re-test. When finished, you can apply the changes to update the state machine. For more information, see Working with Step Functions and Working with Workflow Studio in the AWS Toolkit for Visual Studio Code.

Deploying your workflows

After you have defined and debugged your workflows, you'll probably want to deploy using Infrastructure as Code frameworks. You can choose to deploy your state machines using a variety of IaC options, including: AWS Serverless Application Model, AWS CloudFormation, AWS CDK, and Terraform.

AWS Serverless Application Model

You can use AWS Serverless Application Model with Step Functions to build workflows and deploy the infrastructure you need, including 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 Using AWS SAM to build Step Functions workflows.

AWS CloudFormation

You can use your state machine definitions directly in AWS CloudFormation templates.

For more information, see Using AWS CloudFormation to create a workflow in Step Functions.

AWS CDK

You can build Standard and Express state machines with AWS CDK.

To build a Standard workflow, see Using CDK to create a Standard workflow.

To build an Express workflow, see Using CDK to create an Express workflow.

Terraform

Terraform by HashiCorp is a framework for building applications using infrastructure as code (IaC). With Terraform, you can create state machines and use features, such as previewing infrastructure deployments and creating reusable templates. Terraform templates help you maintain and reuse the code by breaking it down into smaller chunks.

For more information, see Using Terraform to deploy state machines in Step Functions.

Developing workflows in Step Functions Workflow Studio

When editing a workflow in the AWS Step Functions console, you'll use a visual tool called Workflow Studio. With Workflow Studio, you can drag-and-drop states onto a canvas to build your workflows. You can add, edit, and configure states, set input and output filters, transform results, and set up error handling.

As you modify states in your workflow, Workflow Studio will validate and auto-generate the state machine definition. You can review the generated code, edit the configuration, and even modify the text definition with the built-in code editor. When you're finished, you can save your workflow, run it, and then examine the results.

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

You can also use Workflow Studio from **within** AWS Infrastructure Composer, a visual designer to create infrastructure as code with AWS Serverless Application Model and AWS CloudFormation. To discover the benefits of this approach, see Using Workflow Studio in Infrastructure Composer.

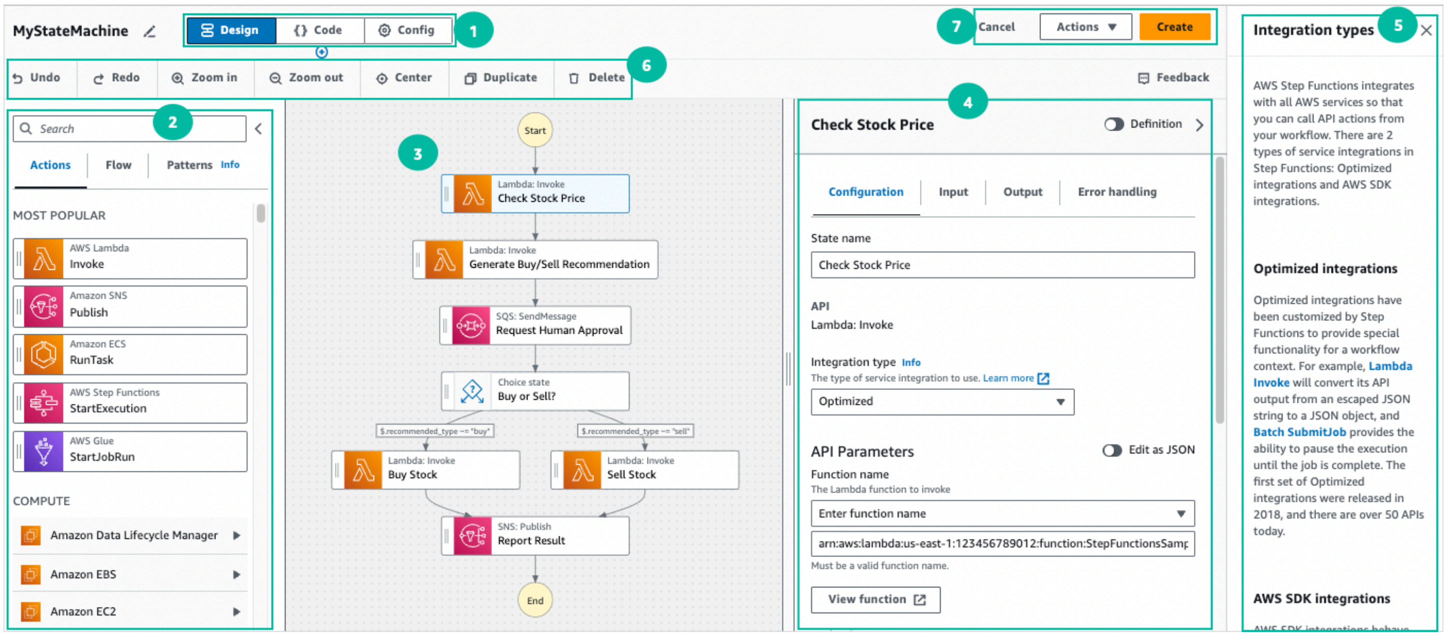
Workflow Studio has three modes: **Design**, **Code**, and **Config**. In *Design mode*, you can drag-and-drop states onto the canvas. *Code mode* provides a built-in code editor for editing your workflow definitions within the console. In *Config mode*, you can manage your workflow configuration.

> **Working with Workflow Studio in Visual Studio Code**
>
> With the AWS toolkit, you can use Workflow Studio from within VS Code to visualize, build, and even test individual states in your state machines. You provide state inputs and set variables, start the test, then you can see how your data is transformed. You can adjust the workflow and re-test. When finished, you can apply the changes to update the state

machine. For more information, see Working with Workflow Studio in the AWS Toolkit for Visual Studio Code.

Design mode

Design mode provides a graphical interface to visualize your workflows as you build their prototypes. The following image shows the states browser, workflow canvas, inspector, and contextual help panels in the **Design** mode of Workflow Studio.



1. Mode buttons switch between the three modes. You cannot switch modes if your ASL workflow definition is invalid.
2. The States browser contains the following three tabs:
   - The **Actions** tab provides a list of AWS APIs that you can drag and drop into your workflow graph in the canvas. Each action represents a Task workflow state state.
   - The **Flow** tab provides a list of flow states that you can drag and drop into your workflow graph in the canvas.
   - The **Patterns** tab provides several ready-to-use, reusable building blocks that you can use for a variety of use cases. For example, you can use these patterns to iteratively process data in an Amazon S3 bucket.
3. The Canvas and workflow graph is where you drag and drop states into your workflow graph, change the order of states, and select states to configure or view.

4. The [Inspector panel](#) panel is where you can view and edit the properties of any state you've selected on the canvas. Turn on the **Definition** toggle to view the Amazon States Language code for your workflow, with the currently selected state highlighted.

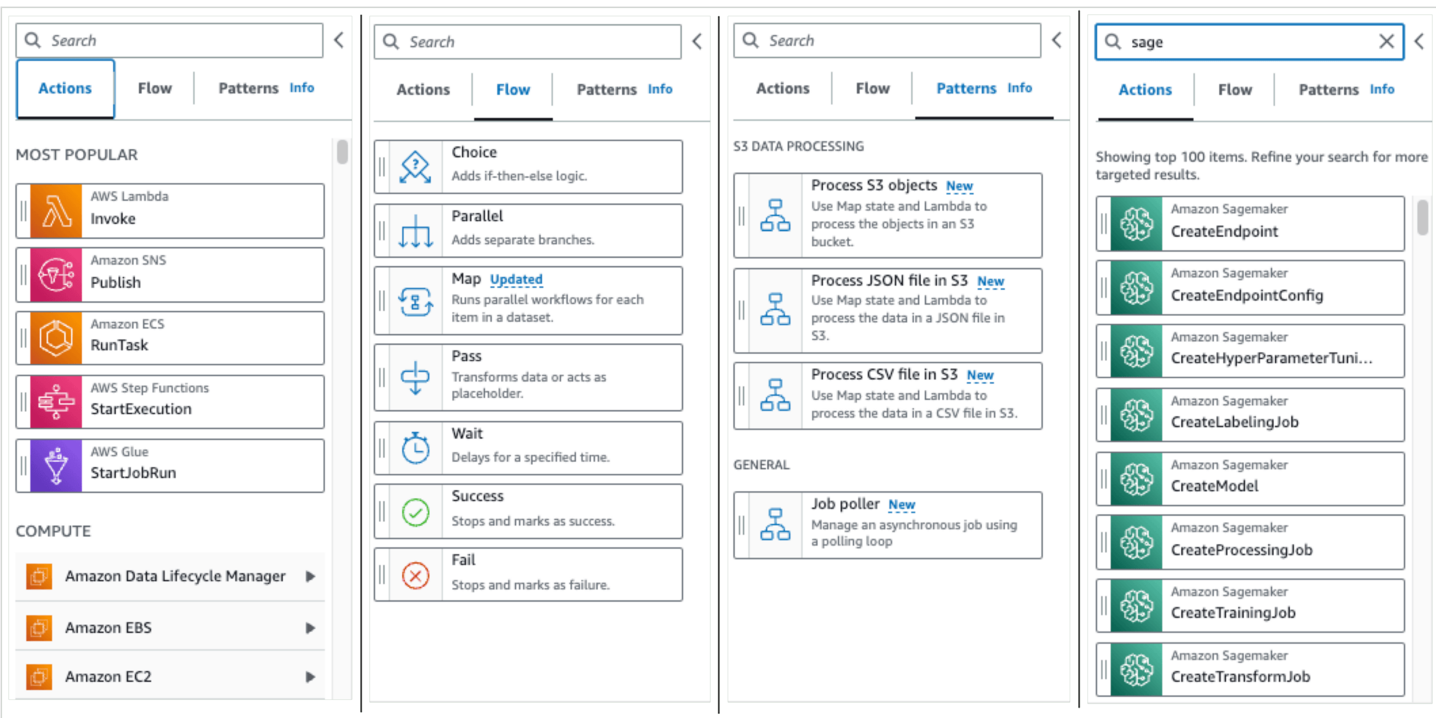
5. **Info** links open a panel with contextual information when you need help. These panels also include links to related topics in the Step Functions documentation.

6. Design toolbar – Contains a set of buttons to perform common actions, such as undo, delete, and zoom in.

7. Utility buttons – A set of buttons to perform tasks, such as saving your workflows or exporting their ASL definitions in a JSON or YAML file.

States browser

From the States browser, you can select states to drag and drop on to your workflow canvas. The **Actions** tab provides a list of task states that connect to 3rd party HTTP endpoints and AWS APIs. The **Flow** tab provides a list of states with which you can direct and control your workflow. Flow states include: Choice, Parallel, Map, Pass, Wait, Success, and Fail. The **Patterns** tab provides ready-to-use, reusable pre-defined building blocks. You can search among all state types with the search box at the top of the panel.

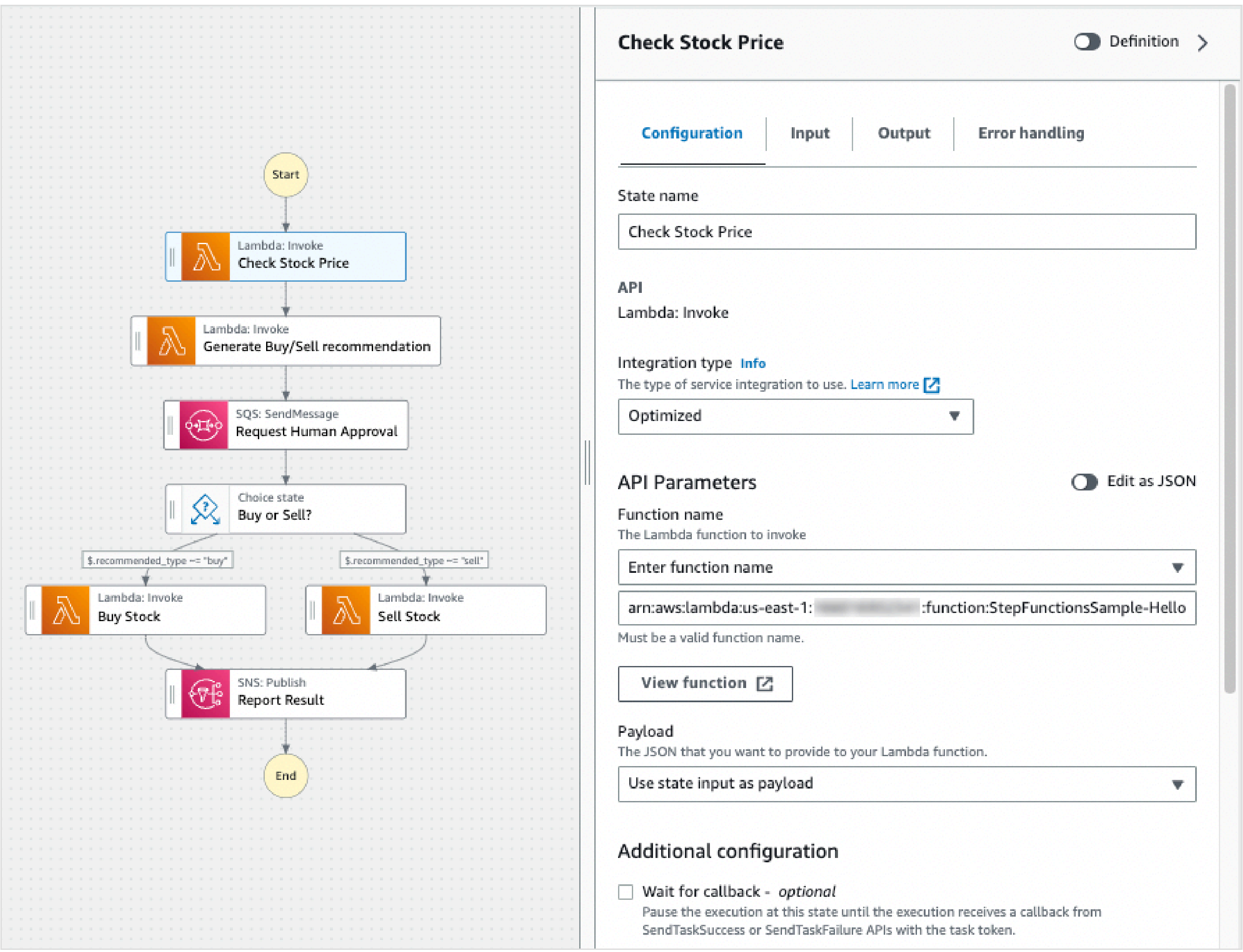


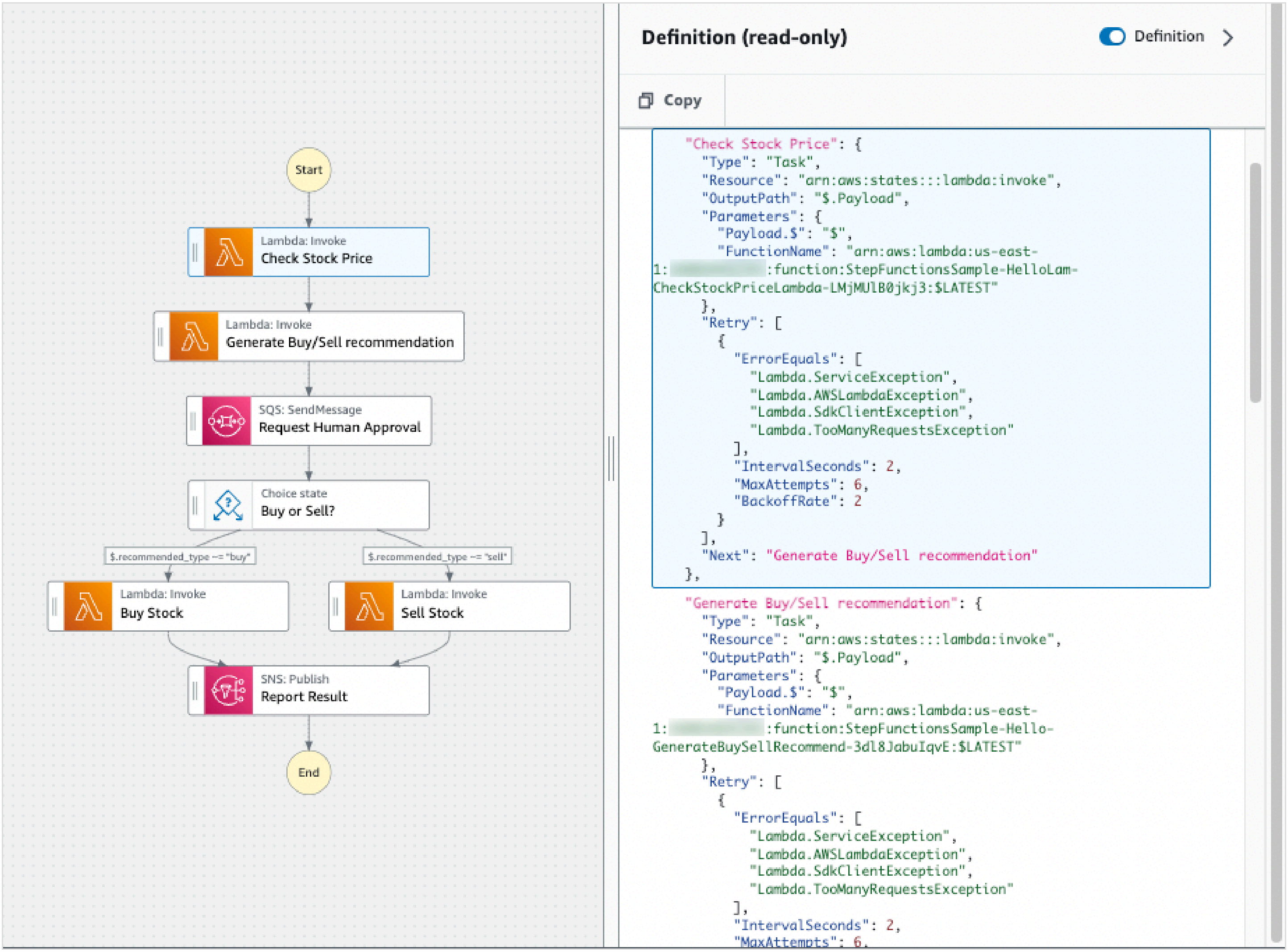
Canvas and workflow graph

After you choose a state to add to your workflow, you can drag it to the canvas and drop it into your workflow graph. You can also drag and drop states to move them within your workflow. If your workflow is large, you can zoom in or out to view different parts of your workflow graph in the canvas.

Inspector panel

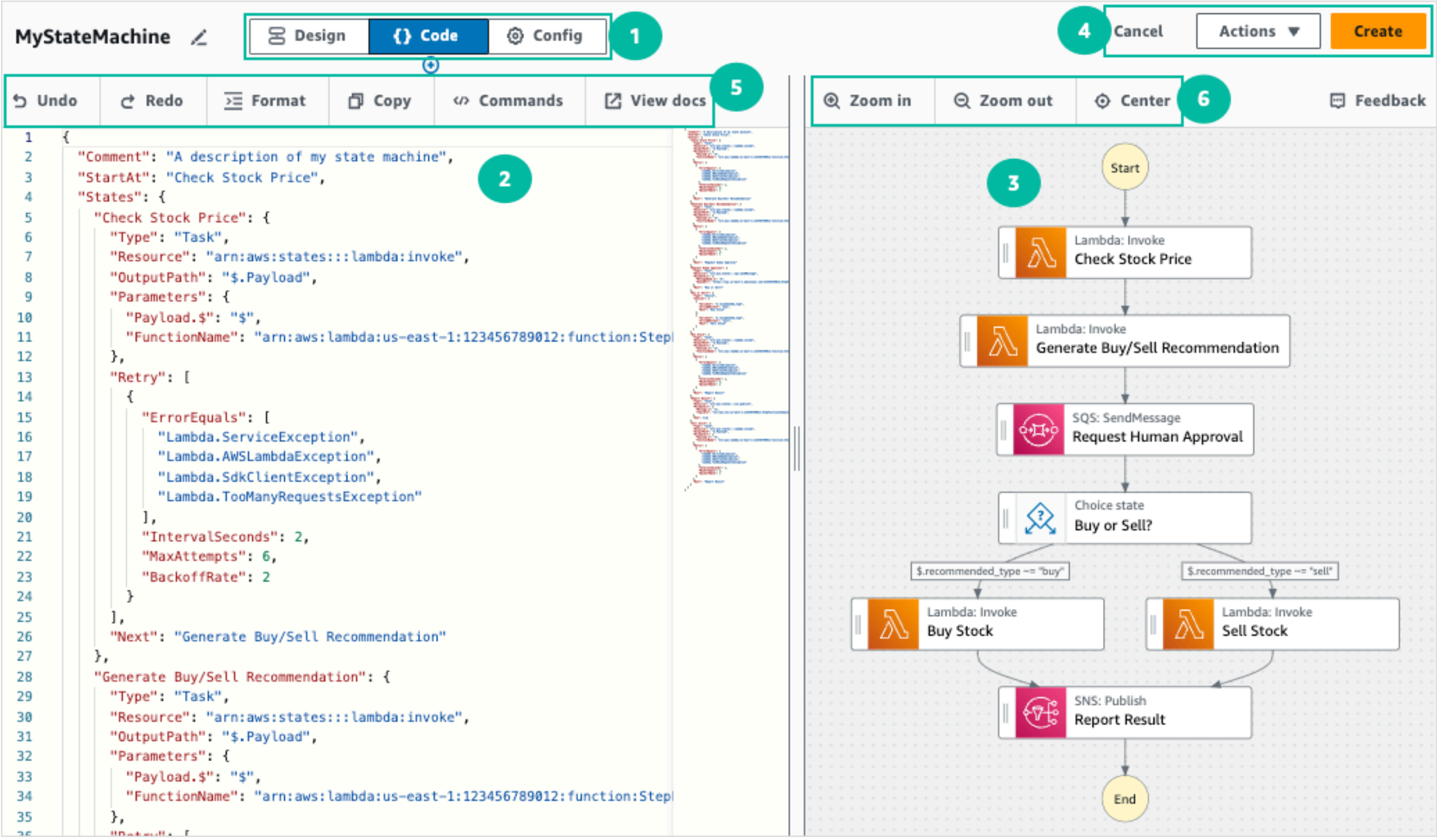
You can configure any states that you add to your workflow from the **Inspector** panel on the right. Choose the state you want to configure, and you will see its configuration options in the **Inspector** panel. To see the auto-generated [ASL definition]() for your workflow code, turn on the **Definition** toggle. The ASL definition associated with the state you've selected will appear highlighted.





Code mode

In **Code** mode of Workflow Studio, you can use an integrated code editor to view, write, and edit the [Using Amazon States Language to define Step Functions workflows] (ASL) definition of your workflows within the Step Functions console. The following screenshot shows the components in the **Code** mode.



1. Mode buttons switch between the three modes. You cannot switch modes if your ASL workflow definition is invalid.
2. The Code editor is where you write and edit the ASL definition of your workflows within the Workflow Studio. The code editor also provides features, such as syntax highlighting and auto-completion.
3. Graph visualization – Shows a real-time graphical visualization of your workflow.
4. Utility buttons – A set of buttons to perform tasks, such as saving your workflows or exporting their ASL definitions in a JSON or YAML file.
5. Code toolbar – Contains a set of buttons to perform common actions, such as undoing an action or formatting the code.
6. Graph toolbar – Contains a set of buttons to perform common actions, such as zooming in and zooming out the workflow graph.

Code editor

The code editor provides an IDE-like experience to write and edit your workflow definitions using JSON within the Workflow Studio. The code editor includes several features, such as syntax highlighting, auto-complete suggestions, ASL definition validation, and context-sensitive help display. As you update your workflow definition, the Graph visualization renders a real-time graph of your workflow. You can also see the updated workflow graph in the Design mode.

If you select a state in the Design mode or the graph visualization pane, the ASL definition of that state appears highlighted in the code editor. The ASL definition of your workflow is automatically updated if you reorder, delete, or add a state in the **Design** mode or the graph visualization pane.

The code editor can make suggestions to auto-complete fields and states.

- To see a list of fields you can include within a specific state, press **`Ctrl+Space`**.
- To generate a code snippet for a new state in your workflow press **`Ctrl+Space`** after the current state's definition.
- To display a list of all available commands and **keyboard shortcuts**, press **`F1`**.

Graph visualization

The graph visualization panel shows your workflow in a graphical format. When you write your workflow definitions in the Code editor of Workflow Studio, the graph visualization pane renders a real-time graph of your workflow.

As you reorder, delete, or duplicate a state in the graph visualization pane, the workflow definition in the Code editor is automatically updated. Similarly, as you update your workflow definitions, reorder, delete, or add a state in the Code editor, the visualization is automatically updated.

If the JSON in the ASL definition of your workflow is invalid, the graph visualization panel pauses the rendering and displays a status message at the bottom of the pane.

Config mode

In the **Config** mode of Workflow Studio, you can manage the general configuration of your state machines. In this mode, you can specify settings, such as the following:

- **Details**: Set the workflow **name** and **type**. Note that both **cannot** be changed after you create the state machine.
- **Permissions** : you can create a new role (recommended), choose an existing role, or enter an ARN for a specific role. If you select the option to create a new role, Step Functions creates an execution role for your state machines using least privileges. The generated IAM roles are valid

for the AWS Region in which you create the state machine. Prior to creation, you can review the permissions that Step Functions will automatically generate for your state machine.

- **Logging**: You can enable and set a log level for your state machine. Step Functions logs the execution history events based on your selection. You can optionally use a customer managed key to encrypt your logs. For more information about log levels, see Log levels for Step Functions execution events.

In **Additional configuration**, you can set one or more of the following **optional** configuration options:

- **Enable X-Ray tracing**: You can send traces to X-Ray for state machine executions, even when a trace ID is not passed by an upstream service. For more information, see Trace Step Functions request data in AWS X-Ray.
- **Publish version on creation**: A *version* is a numbered, immutable snapshot of a state machine that you can run. Choose this option to publish a version of your state machine while creating the state machine. Step Functions publishes version 1 as the first revision of the state machine. For more information about versions, see State machine versions in Step Functions workflows.
- **Encrypt with customer managed key** : You can provide a key that you mange directly to encrypt your data. For information, see Data at rest encryption
- **Tags**: Choose this box to add tags that can help you track and manage the costs associated with your resources, and provide better security in your IAM policies. For more information about tags, see Tagging state machines and activities in Step Functions.

Creating a workflow with Workflow Studio in Step Functions

Learn to create, edit, and run workflows using Step Functions Workflow Studio. After your workflow is ready, you can save, run, and export it.

In this topic

Create a state machine

In Workflow Studio, you can either choose a starter template or a blank template to create a workflow.

A starter template is a ready-to-run sample project that automatically creates the workflow prototype and definition, and deploys all the related AWS resources that your project needs to your AWS account. You can use these starter templates to deploy and run them as is, or use the workflow prototypes to build on them. For more information about starter templates, see Deploy a state machine using a starter template for Step Functions.

With a blank template, you use the Design or Code mode to create your custom workflow.

Create a state machine using a starter template

1. Open the Step Functions console and choose **Create state machine**.
2. In the **Choose a template** dialog box, do one of the following to choose a sample project:
   - Type **`Task Timer`** in the Search by keyword box, and then choose **Task Timer** from the search results.
   - Browse through the sample projects listed under **All** on the right pane, and then choose **Task Timer**.
3. Choose **Next** to continue.
4. Choose how to use the template:
5. Choose **Use template** to continue with your selection.

a. **Run a demo** – creates a read-only state machine. After review, you can create the workflow and all related resources.

b. **Build on it** – provides an editable workflow definition that you can review, customize, and deploy with your own resources. (Related resources, such as functions or queues, will **not** be created automatically.)

Create a workflow using a blank template

When you want to start from a clean canvas, create a workflow from the blank template.

1. Open the Step Functions console.

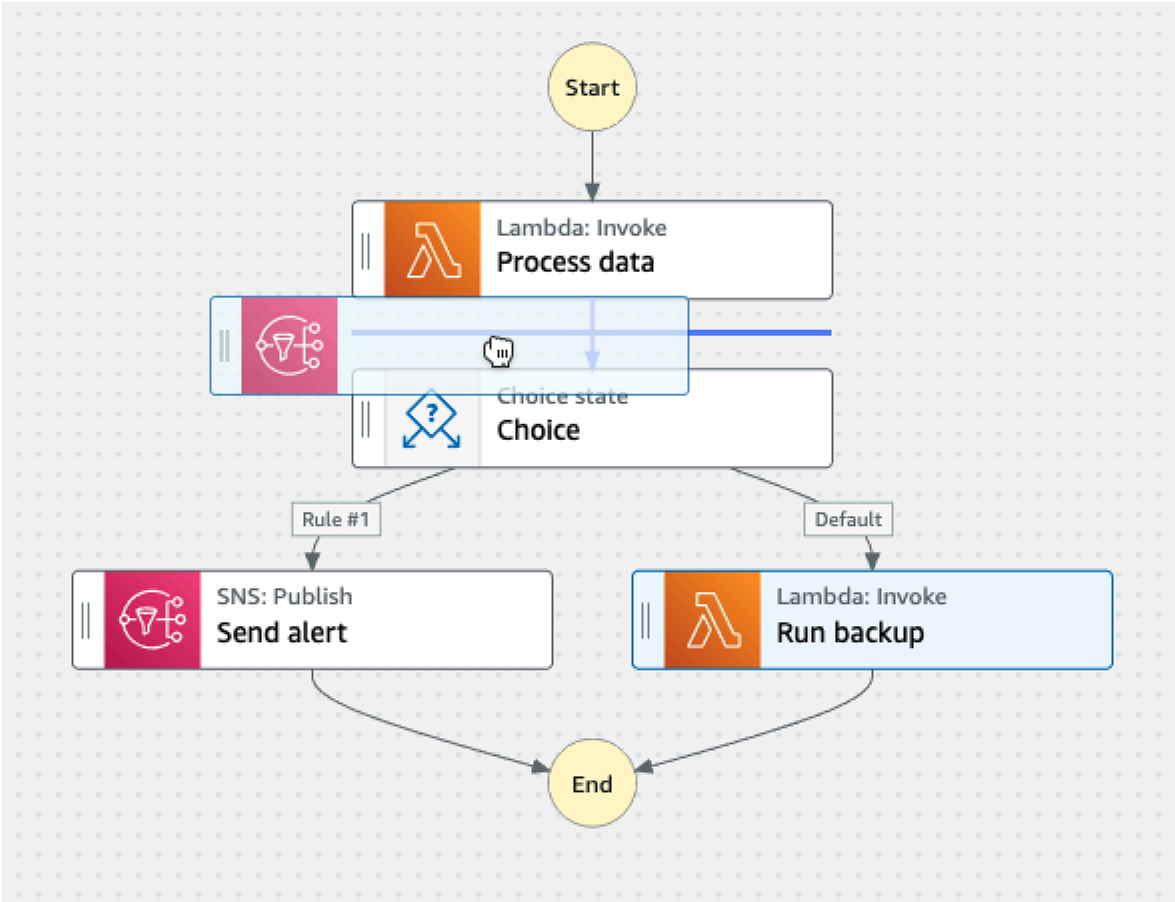
2. Choose **Create state machine**.
3. Choose **Create from blank**.
4. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.

   You can now start designing your workflow in Design mode or writing your workflow definition in Code mode.
5. Choose **Config** to manage the configuration of your workflow in the Config mode. For example, provide a name for your workflow and choose its type.

Design a workflow

When you know the name of the state you want to add, use the search box at the top of the States browser to find it. Otherwise, look for the state you need in the browser and add it onto the canvas.

You can reorder 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 to show where the state will be inserted into your workflow, as shown in the following screenshot:



After a state is dropped onto the canvas, its code is auto-generated and added inside the workflow definition. To see the definition, turn on the **Definition** toggle on the Inspector panel. You can choose Code mode to edit the definition with the built-in code editor.

After you drop a state onto the canvas, you can configure it in the Inspector panel panel on the right. This panel contains the **Configuration**, **Input**, **Output**, and **Error Handling** tabs for each of the state or API action that you place on the canvas. You configure the states you include in your workflows in the **Configuration** tab.

For example, the **Configuration** tab for Lambda Invoke API action provides the following options:

- **State name**: You can identify the state with a custom name or accept the default generated name.
- **API** shows which API action is used by the state.
- **Integration type**: You can choose the service integration type used to call API actions on other services.
- **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.
- **Payload** : you can choose to use the state input, a JSON object, or no payload to pass as the payload to your Lambda function. If you choose JSON, you can include both static values and values selected from the state input.
- (Optional) Some states will have an option to select **Wait for task to complete** or **Wait for callback**. When available, you can choose one of the following service integration patterns:
  - **No option selected**: Step Functions will use the Request Response 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) integration pattern.
  - **Wait for callback**: Step Functions will use the Wait for a Callback with Task Token integration pattern.
- (Optional) To access resources configured in different AWS accounts within your workflows, Step Functions provides cross-account access. **IAM role for cross-account access** provides options to:
  - **Provide IAM role ARN**: Specify the IAM role that contains appropriate resource access permissions. These resources are available in a target account, which is an AWS account to which you make cross-account calls.
  - **Get IAM role ARN at runtime from state input**: Specify a reference path to an existing key-value pair in the state’s JSON input which contains the IAM role.
- **Next state** lets you to select the state you want to transition to next.
- (Optional) **Comment** field will not affect the workflow, but you can be use it to annotate your workflow.

Some states will have additional generic configuration options. For example, the Amazon ECS `RunTask` state configuration contains an `API Parameters` field populated with placeholder values. For these states, you can replace the placeholder values with configurations that are suited to your needs.

To delete a state

You can press backspace, right-click and choose **Delete state**, or choose **Delete** on the Design toolbar.

Run your workflow

When your workflow is ready to go, you can run it and view its execution from the Step Functions console.

To run a workflow in Workflow Studio

1. In the **Design**, **Code**, or **Config** mode, choose **Execute.**

   The **Start execution** dialog box opens in a new tab.

2. In the **Start execution** dialog box, do the following:

   1. (Optional) Enter a custom execution name to override the generated default.

      > **Non-ASCII names and logging**
      >
      > Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

   2. (Optional) In the **Input** box, enter input values in JSON format to run your workflow.
   3. Choose **Start execution**.
   4. The Step Functions console directs you to a page that's titled with your execution ID, known as the *Execution Details* page. You can review the execution results as the workflow progresses and after it completes.

      To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the Step details pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

Edit your workflow

You can edit an existing workflow visually in the Design mode of Workflow Studio.

In the Step Functions console, choose the workflow you want to edit from the **State machines** page. The workflow opens in **Design** mode of Workflow Studio.

You can also edit the workflow definition in Code mode. Choose the **Code** button to view or edit the workflow definition in Workflow Studio.

> **Note**
>
> If you see errors in your workflow, you must fix them in **Design** mode. You can't switch to the **Code** or **Config** mode if any errors exist in your workflow.

When you save changes to your workflow, you have the option to also publish a new **version**. With versions, you can choose to run the original or alternate versions of your workflow. To learn more about managing workflows with versions, see State machine versions in Step Functions workflows

Export your workflow

You can export your workflow's Amazon States Language (ASL) definition and your workflow graph:

1. Choose your workflow in the Step Functions console.
2. On the *State machine detail* page, choose **Edit**.
3. Choose the **Actions** dropdown button, and then do one or both of the following:
    - To export the workflow graph to an SVG or PNG file, under **Export graph**, select the format you want.
    - To export the workflow definition as a JSON or YAML file, under **Export definition**, select the format you want.

Creating a workflow prototype with placeholders

You can use Workflow Studio or Workflow Studio in Infrastructure Composer to create prototypes of new workflows that contain *placeholder resources* which are named resources that do not exist yet.

To create a workflow prototype:

1. Sign in to the Step Functions console.

2. Choose **Create state machine**.
3. Choose **Create from blank**.
4. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.
5. The Design mode of Workflow Studio opens. Design your workflow in Workflow Studio. To include placeholder resources:
   a. Choose the state for which you want to include a placeholder resource, and then in **Configuration**:
      - For Lambda Invoke states, choose **Function name**, then choose **Enter function name**. You can also enter a custom name for your function.
      - For Amazon SQS Send Message states, choose **Queue URL**, then choose **Enter queue URL**. Enter a placeholder queue URL.
      - For Amazon SNS Publish states, from **Topic**, choose a topic ARN.
      - For all other states listed under **Actions**, you can use the default configuration.

      > **Note**
      >
      > If you see errors in your workflow, you must fix them in **Design** mode. You can't switch to the **Code** or **Config** mode if any errors exist in your workflow.

   b. (Optional) To view the auto-generated ASL definition of your workflow, choose **Definition**.
   c. (Optional) To update the workflow definition in Workflow Studio, choose the **Code** button.

      > **Note**
      >
      > If you see errors in your workflow definition, you must fix them in **Code** mode. You can't switch to the **Design** or **Config** mode if any errors exist in your workflow definition.

6. (Optional) To edit the state machine name, choose the edit icon next to the default state machine name of **MyStateMachine** and specify a name in the **State machine name** box.

   You can also switch to the Config mode to edit the default state machine name.
7. Specify your workflow settings, such as state machine type and its execution role.
8. Choose **Create**.

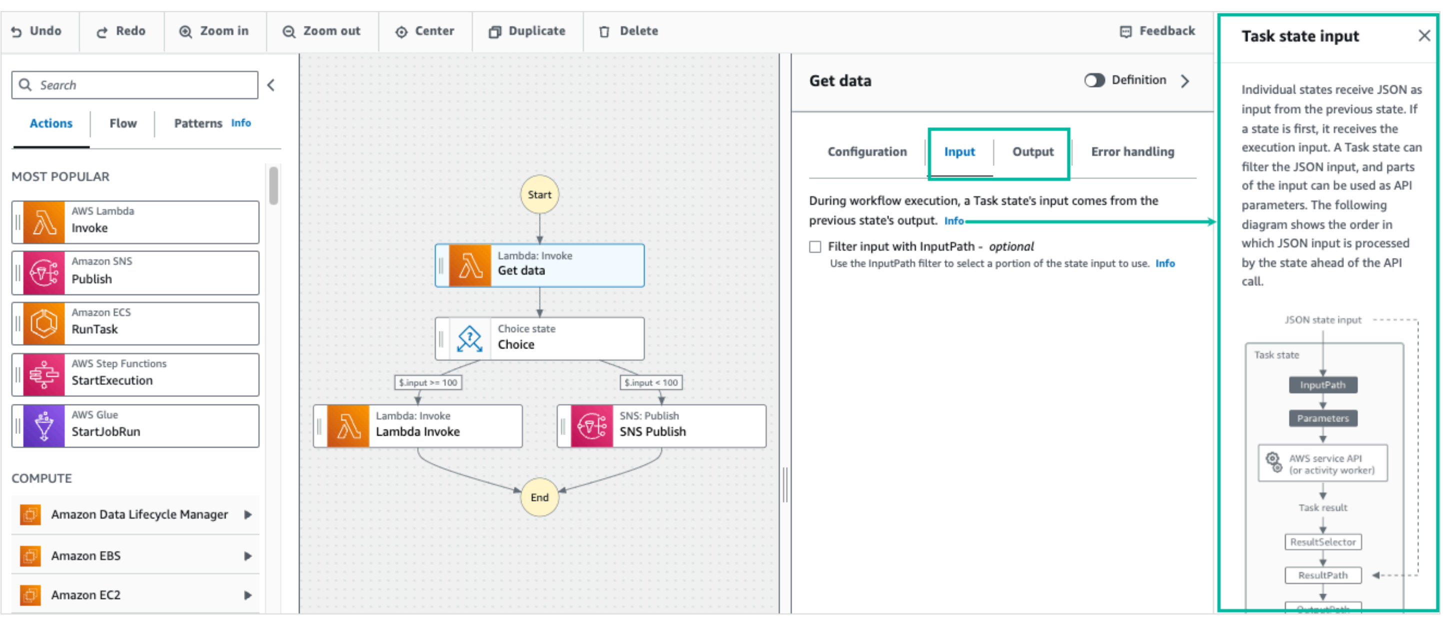
You've now created a new workflow with placeholder resources that can be used to prototype. You can export your workflow definition and the workflow graph.

- To export your workflow definition as a JSON or YAML file, in the **Design** or **Code** mode, choose the **Actions** dropdown button. Then, under **Export definition**, select the format you want to export. You can use this exported definition as the starting point for local development with the AWS Toolkit for Visual Studio Code.
- To export your workflow graph to an SVG or PNG file, in the **Design** or **Code** mode, choose the **Actions** dropdown button. Then, under **Export definition**, select the format you want.

Configure states inputs and outputs with Workflow Studio in Step Functions

> **Managing state and transforming data**
>
> Learn about Passing data between states with variables and Transforming data with JSONata.

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 Inspector panel 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 Processing input and output in Step Functions.

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` filter under the **Input** tab in the Inspector panel panel. The `InputPath` is a string, beginning with `$`, that identifies a specific JSON node. These are called reference paths, and they follow JsonPath syntax.

To filter the input:

- Choose **Filter input with InputPath**.
- Enter a valid JsonPath for the `InputPath` filter. For example, **`$.data`**.

Your `InputPath` filter will be added to your workflow.

Example Example 1: Use InputPath filter in Workflow Studio

Say the input to your state includes the following JSON data.

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

To apply the `InputPath` filter, 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.

```
{"a"}
```

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

```
[ 1, 2 ]
```

Parallel workflow state, Map workflow state, and Pass workflow state 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 Accessing execution data from the Context object in Step Functions 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 Example 2: Create custom JSON input for Parallel state

Say the following JSON data is the input to a Parallel state.

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

To select part of this input 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.

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

The following JSON data will be the result.

```
{
  "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** panel. For Task workflow state states, any output filters you select are processed in this order:

1. `ResultSelector`: Use this filter to manipulate the state's result. You can construct a new JSON object with parts of the result.
2. `Specifying state output using ResultPath in Step Functions`: Use this filter to select a combination of the state input and the task result to pass to the output.
3. `Filtering state output using OutputPath`: Use this filter to filter the JSON output to choose which information from the result will be passed to the next state.

Use ResultSelector

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

- Task workflow state states, which are all states listed in the **Actions** tab of the States browser.

- Map workflow state states, in the **Flow** tab of the States browser.
- Parallel workflow state states, in the **Flow** tab 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 Example to use ResultSelector filter

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.

```
{
  "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"
  }
}
```

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.

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

Using `ResultSelector` produces the following result.

```
{
 "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 Specifying state output using ResultPath in Step Functions.

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

- Task workflow state states, which are all states listed in the **Actions** tab of the States browser.
- Map workflow state states, in the **Flow** tab of the States browser.
- Parallel workflow state states, in the **Flow** tab of the States browser.
- Pass workflow state states, in the **Flow** tab 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 Example to use ResultPath filter

Say the following is the input to a Task state.

```
{
```

```
  "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 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.

```
{
  "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 need. The `OutputPath` is a string, beginning with `$`, that identifies nodes within JSON text.

Example Example to use OutputPath filter

Imagine a Lambda Invoke API call returns metadata in addition to the Lambda function's result.

```
{
  "ExecutedVersion": "$LATEST",
  "Payload": {
     "foo": "bar",
     "colors": [
          "red",
          "blue",
          "green"
     ],
     "car": {
          "year": 2008,
          "make": "Toyota",
          "model": "Matrix"
```

```
      }
    },
  "SdkHttpMetadata": {
    "AllHttpHeaders": {
      "X-Amz-Executed-Version": ["$LATEST"]
...
```

You can use `OutputPath` to filter out the additional metadata. By default, the value of **OutputPath** filter for Lambda Invoke states created through the Workflow Studio is `$.Payload`. This default value removes the additional metadata and returns an output equivalent to running the Lambda function directly.

The Lambda Invoke task result example and the value of `$.Payload` for the **Output** filter pass the following JSON data as the output.

```
{
 "foo": "bar",
 "colors": [
      "red",
      "blue",
      "green"
 ],
 "car": {
      "year": 2008,
      "make": "Toyota",
      "model": "Matrix"
 }
}
```

Note

The `OutputPath` filter is the last output filter to take effect, so if you use additional output filters such as `ResultSelector` or `ResultPath`, you should modify the default value of `$.Payload` for the `OutputPath` filter accordingly.

Set up execution roles with Workflow Studio in Step Functions

You can use Workflow Studio to set up execution roles for your workflows. Every Step Functions state machine requires an AWS Identity and Access Management (IAM) role which grants the state machine permission to perform actions on AWS services and resources or call HTTPS APIs. This role is called an *execution role*.

The execution role must contain IAM policies for each action, for example, policies that allow the state machine to invoke an AWS Lambda function, run an AWS Batch job, or call the Stripe API. Step Functions requires you to provide an execution role in the following cases:

- You create a state machine in the console, AWS SDKs or AWS CLI using the CreateStateMachine API.
- You test a state in the console, AWS SDKs, or AWS CLI using the TestState API.

Topics

- About auto-generated roles
- Automatically generating roles
- Resolving role generation problems
- Role for testing HTTP Tasks in Workflow Studio
- Role for testing an optimized service integration in Workflow Studio
- Role for testing an AWS SDK service integration in Workflow Studio
- Role for testing flow states in Workflow Studio

About auto-generated roles

When you create a state machine in the Step Functions console, Workflow Studio can automatically create an execution role for you which contains the necessary IAM policies. Workflow Studio analyzes your state machine definition and generates policies with the least privileges necessary to execute your workflow.

Workflow Studio can generate IAM policies for the following:

- HTTP Tasks that call HTTPS APIs.
- Task states that call other AWS services using optimized integrations, such as Lambda Invoke, DynamoDB GetItem, or AWS Glue StartJobRun.
- Task states that run nested workflows.
- Distributed Map states, including policies to start child workflow executions, list Amazon S3 buckets, and read or write S3 objects.

- X-Ray tracing. Every role that is auto-generated in Workflow Studio contains a policy which grants permissions for the state machine to send traces to X-Ray.
- Using CloudWatch Logs to log execution history in Step Functions when logging is enabled on the state machine.

Workflow Studio can't generate IAM policies for Task states that call other AWS services using AWS SDK integrations.

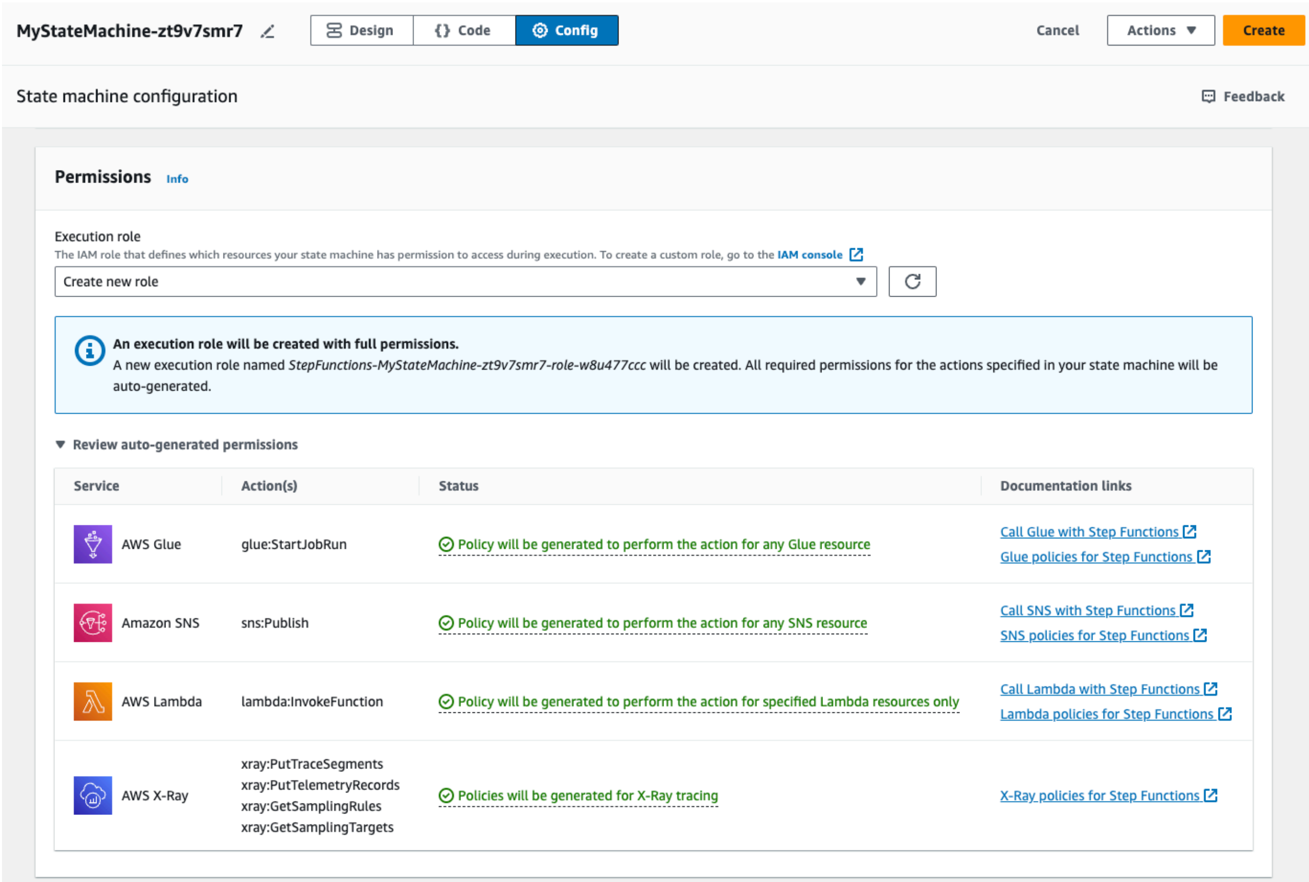
Automatically generating roles

1. Open the Step Functions console, choose **State machines** from the menu, then choose **Create state machine**.

   You can also update an existing state machine. Refer Step 4 if you're updating a state machine.
2. Choose **Create from blank**.
3. Name your state machine, then choose **Continue** to edit your state machine in Workflow Studio.
4. Choose the **Config** tab.
5. Scroll down to the **Permissions** section, and do the following:

   a. For **Execution role**, make sure you keep the default selection of **Create new role**.

      Workflow Studio automatically generates all the required IAM policies for every valid state in your state machine definition. It displays a banner in with the message, **An execution role will be created with full permissions.**



> **Tip**
>
> To review the permissions that Workflow Studio automatically generates for your state machine, choose **Review auto-generated permissions**.

> **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.

If Workflow Studio can't generate all the required IAM policies, it displays a banner with the message **Permissions for certain actions cannot be auto-generated. An IAM role will**

be created with partial permissions only. For information about how to add the missing permissions, see Resolving role generation problems.

b. Choose **Create** if you're creating a state machine. Otherwise, choose **Save**.

c. Choose **Confirm** in the dialog box that appears.

Workflow Studio saves your state machine and creates the new execution role.

Resolving role generation problems

Workflow Studio can't automatically generate an execution role with all the required permissions in the following cases:

- There are errors in your state machine. Make sure to resolve all validation errors in Workflow Studio. Also, make sure that you address any server-side errors you encounter in the course of saving.
- Your state machine contains tasks use AWS SDK integrations. Workflow Studio can't auto-generate IAM policies in this case. Workflow Studio displays a banner with the message, **Permissions for certain actions cannot be auto-generated. An IAM role will be created with partial permissions only.** In the **Review auto-generated permissions** table, choose the content in **Status** for more information about the policies your execution role is missing. Workflow Studio can still generate an execution role, but this role will not contain IAM policies for all actions. See the links under **Documentation links** to write your own policies and add them to the role after it is generated. These links are available even after you save the state machine.

Role for testing HTTP Tasks in Workflow Studio

Testing an HTTP Task state requires an execution role. If you don’t have a role with sufficient permissions, use one of the following options to create a role:

- **Auto-generate a role with Workflow Studio (recommended) –** This is the secure option. Close the **Test state** dialog box and follow the instructions in Automatically generating roles. This will require you to create or update your state machine first, then go back into Workflow Studio to test your state.
- **Use a role with Administrator access –** If you have permissions to create a role with full access to all services and resources in AWS, you can use that role to test any type of state in your workflow. To do this, you can create a Step Functions service role and add the AdministratorAccess policy to it in the IAM console https://console.aws.amazon.com/iam/.

Role for testing an optimized service integration in Workflow Studio

Task states that call optimized service integrations require an execution role. If you don't have a role with sufficient permissions, use one of the following options to create a role:

- **Auto-generate a role with Workflow Studio (recommended) –** This is the secure option. Close the **Test state** dialog box and follow the instructions in Automatically generating roles. This will require you to create or update your state machine first, then go back into Workflow Studio to test your state.
- **Use a role with Administrator access –** If you have permissions to create a role with full access to all services and resources in AWS, you can use that role to test any type of state in your workflow. To do this, you can create a Step Functions service role and add the AdministratorAccess policy to it in the IAM console https://console.aws.amazon.com/iam/.

Role for testing an AWS SDK service integration in Workflow Studio

Task states that call AWS SDK integrations require an execution role. If you don't have a role with sufficient permissions, use one of the following options to create a role:

- **Auto-generate a role with Workflow Studio (recommended) –** This is the secure option. Close the **Test state** dialog box and follow the instructions in Automatically generating roles. This will require you to create or update your state machine first, then go back into Workflow Studio to test your state. Do the following:
  1. Close the **Test state** dialog box
  2. Choose the **Config** tab to view the Config mode.
  3. Scroll down to the **Permissions** section.
  4. Workflow Studio displays a banner with the message, **Permissions for certain actions cannot be auto-generated. An IAM role will be created with partial permissions only.** Choose **Review auto-generated permissions**.
  5. The **Review auto-generated permissions** table displays a row that shows the action corresponding to the task state you want to test. See the links under **Documentation links** to write your own IAM policies into a custom role.
- **Use a role with Administrator access –** If you have permissions to create a role with full access to all services and resources in AWS, you can use that role to test any type of state in your workflow. To do this, you can create a Step Functions service role and add the AdministratorAccess policy to it in the IAM console https://console.aws.amazon.com/iam/.

Role for testing flow states in Workflow Studio

You require an execution role to test flow states in Workflow Studio. Flow states are those states that direct execution flow, such as Choice workflow state, Parallel workflow state, Map workflow state, Pass workflow state, Wait workflow state, Succeed workflow state, or Fail workflow state. The TestState API doesn't work with Map or Parallel states. Use one of the following options to create a role for testing a flow state:

- **Use any role in your AWS account (recommended) –** Flow states do not require any specific IAM policies, because they don't call AWS actions or resources. Therefore, you can use any IAM role in your AWS account.
  1. In the **Test state** dialog box, select any role from the **Execution role** dropdown list.
  2. If no roles appear in the dropdown list, do the following:
     a. In the IAM console https://console.aws.amazon.com/iam/, choose **Roles**.
     b. Choose a role from the list, and copy its ARN from the role details page. You will need to provide this ARN in the **Test state** dialog box.
     c. In the **Test state** dialog box, select **Enter a role ARN** from the **Execution role** dropdown list.
     d. Paste the ARN in **Role ARN**.
- **Use a role with Administrator access –** If you have permissions to create a role with full access to all services and resources in AWS, you can use that role to test any type of state in your workflow. To do this, you can create a Step Functions service role and add the AdministratorAccess policy to it in the IAM console https://console.aws.amazon.com/iam/.

Configure error handling with Workflow Studio in Step Functions

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

You can configure error handling within the Workflow Studio visual editor. By default, when a state reports an error, Step Functions causes the workflow 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 a workflow execution to fail. For more information, see Handling errors in Step Functions workflows. In Workflow Studio, configure error handling in the **Error handling** tab of the Inspector panel.

Retry on errors

You can add one or more rules to action states and the Parallel workflow state flow state to retry the task when an error occurs. These rules are called *retriers*. To add a retrier, choose the edit icon in **Retrier #1** box, then configure its options:

- (Optional) In the **Comment** field, add your comment. It will not affect the workflow, but can be used to annotate your workflow.
- Place the cursor in the **Errors** field and choose an error that will trigger the retrier, or enter a custom error name. You can choose or add multiple errors.
- (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 of 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.

> **Note**
>
> Not all error handling options are available for all states. Lambda Invoke has one retrier configured by default.

Catch errors

You can add one or more rules to action states and to the Parallel workflow state and Map workflow state flow states to catch an error. These rules are called *catchers*. To add a catcher, choose **Add new catcher**, then configure its options:

- (Optional) In the **Comment** field, add your comment. It will not affect the workflow, but can be used to annotate your workflow.

- Place the cursor in **Errors** field and choose an error that will trigger the catcher, or enter a custom error name. You can choose or add multiple errors.
- In the **Fallback state** field, choose a fallback state. This is the state that the workflow will move to next, after an error is caught.
- (Optional) In the **ResultPath** field, add a `ResultPath` filter to add the error to the original state input. The `ResultPath` must be a valid JsonPath. 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 about timeouts, see `TimeoutSeconds` in Task workflow state state.

HeartbeatSeconds

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 Task workflow state state.

Using Workflow Studio in Infrastructure Composer to build Step Functions workflows

Workflow Studio is available in Infrastructure Composer to help you design and build your workflows. Workflow Studio in Infrastructure Composer provides a visual infrastructure as code (IaC) environment that makes it easy for you to incorporate workflows in your serverless applications built using IaC tools, such as CloudFormation templates.

AWS Infrastructure Composer is a visual builder that helps you develop AWS SAM and AWS CloudFormation templates using a simple graphical interface. With Infrastructure Composer, you design an application architecture by dragging, grouping, and connecting AWS services in a visual canvas. Infrastructure Composer then creates an IaC template from your design that you can use to deploy your application with the AWS SAM Command Line Interface (AWS SAM CLI) or CloudFormation. To learn more about Infrastructure Composer, see What is Infrastructure Composer.

When you use Workflow Studio in Infrastructure Composer, Infrastructure Composer connects the individual workflow steps to AWS resources and generates the resource configurations in an AWS SAM template. Infrastructure Composer also adds the IAM permissions required for your workflow to run. Using Workflow Studio in Infrastructure Composer, you can create prototypes of your applications and turn them into production-ready applications.

When you use Workflow Studio in Infrastructure Composer, you can switch back and forth between the Infrastructure Composer canvas and Workflow Studio.

Topics

Using Workflow Studio in Infrastructure Composer to build a serverless workflow

1. Open the Infrastructure Composer console and choose **Create project** to create a project.
2. In the search field in the **Resources** palette, enter **`state machine`**.
3. Drag the **Step Functions State machine** resource onto the canvas.
4. Choose **Edit in Workflow Studio** to edit your state machine resource.

   The following animation shows how you can switch to the Workflow Studio for editing your state machine definition.
   An animation that illustrates how you can use Workflow Studio in Infrastructure Composer.

   The integration with Workflow Studio to edit state machines resources created in Infrastructure Composer is only available for `AWS::Serverless::StateMachine` resource. This integration is not available for templates that use the `AWS::StepFunctions::StateMachine` resource.

Dynamically reference resources using CloudFormation definition substitutions in Workflow Studio

In Workflow Studio, you can use CloudFormation definition substitutions in your workflow definition to dynamically reference resources that you've defined in your IaC template. You can add placeholder substitutions to your workflow definition using the `${dollar_sign_brace}` notation and they are replaced with actual values during the CloudFormation stack creation process. For more information about definition substitutions, see DefinitionSubstitutions in AWS SAM templates.

The following animation shows how you can add placeholder substitutions for the resources in your state machine definition.
Animation showing how to add placeholder substitutions for resources in your state machine.

Connect service integration tasks to enhanced component cards

You can connect the tasks that call optimized service integrations to enhanced component cards in Infrastructure Composer canvas. Doing this automatically maps any placeholder substitutions specified by the `${dollar_sign_brace}` notation in your workflow definition and the `DefinitionSubstitution` property for your `StateMachine` resource. It also adds the appropriate AWS SAM policies for the state machine.

If you map optimized service integration tasks with standard component cards, the connection line doesn't appear on the Infrastructure Composer canvas.

The following animation shows how you can connect an optimized task to an enhanced component card and view the changes in **Change Inspector**.
Animation showing how to connect tasks and optimized service integrations.

You can't connect AWS SDK integrations in your Task state with enhanced component cards or optimized service integrations with standard component cards. For these tasks, you can map the substitutions in the **Resource properties** panel in Infrastructure Composer canvas, and add policies in the AWS SAM template.

> **Tip**
>
> Alternatively, you can also map placeholder substitutions for your state machine under **Definition Substitutions** in the **Resource properties** panel. When you do this, you must add the required permissions for the AWS service your Task state calls in the state machine

execution role. For information about permissions your execution role might need, see Set up execution roles with Workflow Studio in Step Functions.

The following animation shows how you can manually update the placeholder substitution mapping in the **Resource properties** panel.
Animation showing how to update placeholder substitution mapping in the resource properties panel.

Import existing projects and sync them locally

You can open existing CloudFormation and AWS SAM projects in Infrastructure Composer to visualize them for better understanding and modify their designs. Using Infrastructure Composer's local sync feature, you can automatically sync and save your template and code files to your local build machine. Using the local sync mode can compliment your existing development flows. Make sure that your browser supports the File System Access API, which allows web applications to read, write, and save files in your local file system. We recommend using either Google Chrome or Microsoft Edge.

Export Step Functions workflows directly into AWS Infrastructure Composer

The AWS Step Functions console provides the ability to export a saved state machine workflow as a template that's recognized as an advanced IaC resource by Infrastructure Composer. This feature creates an IaC template as an AWS SAM schema and navigates you to Infrastructure Composer. For more information, see Exporting your workflow to IaC templates.

Unavailable Workflow Studio features in AWS Infrastructure Composer

When you use Workflow Studio in Infrastructure Composer, some of the Workflow Studio features are unavailable. In addition, the **API Parameters** section available in the Inspector panel panel supports CloudFormation definition substitutions. You can add the substitutions in the Code mode using the `${dollar_sign_brace}` notation. For more information about this notation, see DefinitionSubstitutions in AWS SAM templates.

The following list describes the Workflow Studio features that are unavailable when you use Workflow Studio in Infrastructure Composer:

- Starter templates – Starter templates are ready-to-run sample projects that automatically create the workflow prototypes and definitions. These templates deploys all the related AWS resources that your project needs to your AWS account.

- Config mode – This mode lets you manage the configuration of your state machines. You can update your state machine configurations in your IaC templates or use the **Resource properties** panel in Infrastructure Composer canvas. For information about updating configurations in the **Resource properties** panel, see Connect service integration tasks to enhanced component cards.
- TestState API
- Option to import or export workflow definitions from the **Actions** dropdown button in Workflow Studio. Instead, from the Infrastructure Composer **menu**, select **Open** > **Project folder**. Make sure that you've enabled the local sync mode to automatically save your changes in the Infrastructure Composer canvas directly to your local machine.
- **Execute** button. When you use Workflow Studio in Infrastructure Composer, Infrastructure Composer generates the IaC code for your workflow. Therefore, you must first deploy the template. Then, run the workflow in the console or through the AWS Command Line Interface (AWS CLI).

Using AWS SAM to build Step Functions workflows

You can use AWS Serverless Application Model with Step Functions to build workflows and deploy the infrastructure you need, including Lambda functions, APIs and events, to create serverless applications.

You can also use the AWS Serverless Application Model CLI in conjunction with the AWS Toolkit for Visual Studio Code as part of an integrated experience to build and deploy AWS Step Functions state machines. 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.

> **Tip**
>
> To deploy a sample serverless application that starts a Step Functions workflow using AWS SAM, see Deploy with AWS SAM in *The AWS Step Functions Workshop*.

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.

  AWS CloudFormation supports `DefinitionSubstitutions` that let you add dynamic references in your workflow definition to a value that you provide in your CloudFormation template. You can add dynamic references by adding substitutions to your workflow definition using the `${dollar_sign_brace}` notation. You also need to define these dynamic references in the `DefinitionSubstitutions` property for the StateMachine resource in your CloudFormation template. These substitutions are replaced with actual values during the CloudFormation stack creation process. For more information, see DefinitionSubstitutions in AWS SAM templates.
- 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 tutorial to learn how to use AWS SAM to create state machines.

Supported AWS SAM CLI functions include:

CLI Command	Description
sam init	Initializes a Serverless Application with an AWS SAM template. Can be used with a SAM template for Step Functions.
sam validate	Validates an AWS SAM template.

CLI Command	Description
sam package	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.
sam deploy	Deploys an AWS SAM application.
sam publish	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.

> **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.

DefinitionSubstitutions in AWS SAM templates

You can define state machines using CloudFormation templates with AWS SAM. Using AWS SAM, you can define the state machine inline in the template or in a separate file. The following AWS SAM template includes a state machine that simulates a stock trading workflow. This state machine invokes three Lambda functions to check the price of a stock and determine whether to buy or sell the stock. This transaction is then recorded in an Amazon DynamoDB table. The ARNs for the Lambda functions and DynamoDB table in the following template are specified using `DefinitionSubstitutions`.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: |
```

```
  step-functions-stock-trader
  Sample SAM Template for step-functions-stock-trader
Resources:
  StockTradingStateMachine:
    Type: AWS::Serverless::StateMachine
    Properties:
      DefinitionSubstitutions:
        StockCheckerFunctionArn: !GetAtt StockCheckerFunction.Arn
        StockSellerFunctionArn: !GetAtt StockSellerFunction.Arn
        StockBuyerFunctionArn: !GetAtt StockBuyerFunction.Arn
        DDBPutItem: !Sub arn:${AWS::Partition}:states:::dynamodb:putItem
        DDBTable: !Ref TransactionTable
      Policies:
        - DynamoDBWritePolicy:
            TableName: !Ref TransactionTable
        - LambdaInvokePolicy:
            FunctionName: !Ref StockCheckerFunction
        - LambdaInvokePolicy:
            FunctionName: !Ref StockBuyerFunction
        - LambdaInvokePolicy:
            FunctionName: !Ref StockSellerFunction
      DefinitionUri: statemachine/stock_trader.asl.json
  StockCheckerFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: functions/stock-checker/
      Handler: app.lambdaHandler
      Runtime: nodejs18.x
      Architectures:
        - x86_64
  StockSellerFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: functions/stock-seller/
      Handler: app.lambdaHandler
      Runtime: nodejs18.x
      Architectures:
        - x86_64
  StockBuyerFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: functions/stock-buyer/
      Handler: app.lambdaHandler
      Runtime: nodejs18.x
```

```
      Architectures:
        - x86_64
  TransactionTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
```

The following code is the state machine definition in the file `stock_trader.asl.json` which is used in the [Create a Step Functions state machine using AWS SAM](#) tutorial.This state machine definition contains several `DefinitionSubstitutions` denoted by the `${dollar_sign_brace}` notation. For example, instead of specifying a static Lambda function ARN for the `Check Stock Value` task, the substitution `${StockCheckerFunctionArn}` is used. This substitution is defined in the [DefinitionSubstitutions](#) property of the template. `DefinitionSubstitutions` is a map of key-value pairs for the state machine resource. In `DefinitionSubstitutions`, ${StockCheckerFunctionArn} maps to the ARN of the `StockCheckerFunction` resource using the CloudFormation intrinsic function [`!GetAtt`](#). When you deploy the AWS SAM template, the `DefinitionSubstitutions` in the template are replaced with the actual values.

```
{
    "Comment": "A state machine that does mock stock trading.",
    "StartAt": "Check Stock Value",
    "States": {
        "Check Stock Value": {
            "Type": "Task",
            "Resource": "arn:aws:states:::lambda:invoke",
            "OutputPath": "$.Payload",
            "Parameters": {
                "Payload.$": "$",
                "FunctionName": "${StockCheckerFunctionArn}"
            },
            "Next": "Buy or Sell?"
        },
        "Buy or Sell?": {
            "Type": "Choice",
            "Choices": [
                {
                    "Variable": "$.stock_price",
                    "NumericLessThanEquals": 50,
```

```
                    "Next": "Buy Stock"
                }
            ],
            "Default": "Sell Stock"
        },
        "Buy Stock": {
            "Type": "Task",
            "Resource": "arn:aws:states:::lambda:invoke",
            "OutputPath": "$.Payload",
            "Parameters": {
                "Payload.$": "$",
                "FunctionName": "${StockBuyerFunctionArn}"
            },
            "Retry": [
                {
                    "ErrorEquals": [
                        "Lambda.ServiceException",
                        "Lambda.AWSLambdaException",
                        "Lambda.SdkClientException",
                        "Lambda.TooManyRequestsException"
                    ],
                    "IntervalSeconds": 1,
                    "MaxAttempts": 3,
                    "BackoffRate": 2
                }
            ],
            "Next": "Record Transaction"
        },
        "Sell Stock": {
            "Type": "Task",
            "Resource": "arn:aws:states:::lambda:invoke",
            "OutputPath": "$.Payload",
            "Parameters": {
                "Payload.$": "$",
                "FunctionName": "${StockSellerFunctionArn}"
            },
            "Next": "Record Transaction"
        },
        "Record Transaction": {
            "Type": "Task",
            "Resource": "arn:aws:states:::dynamodb:putItem",
            "Parameters": {
                "TableName": "${DDBTable}",
                "Item": {
```

```
                    "Id": {
                        "S.$": "$.id"
                    },
                    "Type": {
                        "S.$": "$.type"
                    },
                    "Price": {
                        "N.$": "$.price"
                    },
                    "Quantity": {
                        "N.$": "$.qty"
                    },
                    "Timestamp": {
                        "S.$": "$.timestamp"
                    }
                }
            },
            "End": true
        }
    }
}
```

Next steps

You can learn more about using Step Functions with AWS SAM with the following resources:

- Complete the Create a Step Functions state machine using AWS SAM tutorial to create a state machine with AWS SAM.
- Specify a AWS::Serverless::StateMachine resource.
- Find AWS SAM Policy Templates to use.
- Use AWS Toolkit for Visual Studio Code with Step Functions.
- Review the AWS SAM CLI reference to learn more about the features available in AWS SAM.

You can also design and build your workflows in infrastructure as code (IaC) using visual builders, such as Workflow Studio in Infrastructure Composer. For more information, see Using Workflow Studio in Infrastructure Composer to build Step Functions workflows.

Using AWS CloudFormation to create a workflow in Step Functions

In this tutorial, you will create a AWS Lambda function using AWS CloudFormation. You'll use the AWS CloudFormation console and a YAML template to create a *stack* (IAM roles, the Lambda function, and the state machine). Then, you'll use the 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*.

Step 1: Set up your AWS CloudFormation template

Before you use the example templates, you should understand how to declare the different parts of an AWS CloudFormation template.

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"
```

JSON

```
          "LambdaExecutionRole": {
              "Type": "AWS::IAM::Role",
              "Properties": {
                  "AssumeRolePolicyDocument": {
```

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

To create a Lambda function

Define the following properties 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) =>
 {\n    callback(null, \"Hello World!\");\n};\n"
                },
                "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

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

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

JSON

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

```
                    ]
                }
            },
```

To create a Lambda state machine

Define the Lambda state machine.

YAML

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

JSON

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

```
                                "lambdaArn": {
                                    "Fn::GetAtt": [
                                        "MyLambdaFunction",
                                        "Arn"
                                    ]
                                }
                            }
                        ]
                    },
                    "RoleArn": {
                        "Fn::GetAtt": [
                            "StatesExecutionRole",
                            "Arn"
                        ]
                    }
                }
            }
```

Step 2: Use the AWS CloudFormation template to create a Lambda State Machine

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

```
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:
            Version: "2012-10-17"
            Statement:
              - Effect: Allow
                Action:
                  - "lambda:InvokeFunction"
                Resource: "*"

  MyStateMachine:
    Type: "AWS::StepFunctions::StateMachine"
    Properties:
      DefinitionString:
```

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

JSON

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

```
                    "Handler": "index.handler",
                    "Role": {
                        "Fn::GetAtt": [
                            "LambdaExecutionRole",
                            "Arn"
                        ]
                    },
                    "Code": {
                        "ZipFile": "exports.handler = (event, context, callback)
 => {\n    callback(null, \"Hello World!\");\n};\n"
                    },
                    "Runtime": "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\",\n  \"StartAt\": \"HelloWorld\",\n  \"States\":
 {\n    \"HelloWorld\": {\n      \"Type\": \"Task\",\n      \"Resource\":
 \"${lambdaArn}\",\n      \"End\": true\n    }\n  }\n}",
                            {
                                "lambdaArn": {
                                    "Fn::GetAtt": [
                                        "MyLambdaFunction",
                                        "Arn"
                                    ]
                                }
                            }
                        ]
                    },
                    "RoleArn": {
                        "Fn::GetAtt": [
                            "StatesExecutionRole",
                            "Arn"
                        ]
                    }
                }
            }
        }
    }
```

2. Open the [AWS CloudFormation console](#) and choose **Create Stack**.

3. On the **Select Template** page, 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.
2. On the ***`MyStateMachine-ABCDEFGHIJ1K`*** page, choose **New execution**.

   The **New execution** page is displayed.
3. (Optional) Enter a custom execution name to override the generated default.

   > **Non-ASCII names and logging**
   >
   > Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.
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**.

Using AWS CDK to create a Standard workflow in Step Functions

You can use the AWS Cloud Development Kit (AWS CDK) Infrastructure as Code (IAC) framework, to create an AWS Step Functions state machine that contains an AWS Lambda function.

You will define AWS infrastructure using one of the CDK's supported languages. After you define your infrastructure, you will synthesize your app to an AWS CloudFormation template and deploy it to your AWS account.

You will use this method to define a Step Functions state machine containing a Lambda function, and then run the state machine from the use the Step Functions AWS Management Console.

Before you begin this tutorial, you must set up your AWS CDK development environment as described in Getting Started With the AWS CDK - Prerequisites in the *AWS Cloud Development Kit (AWS CDK) Developer Guide*. 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 "Create a state machine with CloudFormation". 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 workflow state step to illustrate how to add additional steps to your state machine.

> **Tip**
>
> To deploy a sample serverless application that starts a Step Functions workflow using AWS CDK with TypeScript, see Deploy with AWS CDK in *The AWS Step Functions Workshop*.

Step 1: Set up your AWS CDK project

1. In your home directory, or another directory if you prefer, run the following command to create a directory for your new AWS CDK app.

   > **Important**
   >
   > 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 && cd step
   ```

   JavaScript

   ```
   mkdir step && cd step
   ```

   Python

   ```
   mkdir step && cd step
   ```

   Java

   ```
   mkdir step && cd step
   ```

   C#

   Make sure you've installed .NET version 6.0 or higher. For information, see [Supported versions](#).

   ```
   mkdir step && cd step
   ```

2. Initialize the app by using the **cdk init** command. Specify the desired template ("app") and programming language as shown in the following examples.

TypeScript

```
cdk init --language typescript
```

JavaScript

```
cdk init --language javascript
```

Python

```
cdk init --language python
```

After the project is initialized, activate the project's virtual environment and install the AWS CDK's baseline dependencies.

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
cdk init --language java
```

C#

```
cdk init --language csharp
```

Step 2: Use AWS CDK to create a 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

```
const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
    code: lambda.Code.fromInline(`
          exports.handler = (event, context, callback) => {
              callback(null, "Hello World!");
          };
      `),
    runtime: lambda.Runtime.NODEJS_18_X,
    handler: "index.handler",
    timeout: cdk.Duration.seconds(3)
});
```

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_18_X,
    handler: "index.handler",
    timeout: cdk.Duration.seconds(3)
});
```

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_18_X,
                handler="index.handler",
                timeout=Duration.seconds(25))
```

Java

```
final Function helloFunction = Function.Builder.create(this, "MyLambdaFunction")
        .code(Code.fromInline(
```

```
                "exports.handler = (event, context, callback) => { callback(null,
 'Hello World!' );}"))
        .runtime(Runtime.NODEJS_18_X)
        .handler("index.handler")
        .timeout(Duration.seconds(25))
        .build();
```

C#

```
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,
    Handler = "index.handler",
    Timeout = Duration.Seconds(25)
});
```

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 18.x), the function's entry point, and a timeout.

To create a state machine

Our state machine has two states: a Lambda function task, and a Succeed workflow state state. The function requires that we create a Step Functions the section called “Task” that invokes our function. This Task state is used as the first step in the state machine. The success state is added to the state machine using the Task state's `next()` method. The following code first invokes the function named `MyLambdaTask`, then uses the `next()` method to define a success state named `GreetedWorld`.

TypeScript

```
const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
```

```
  definition: new tasks.LambdaInvoke(this, "MyLambdaTask", {
    lambdaFunction: helloFunction
  }).next(new sfn.Succeed(this, "GreetedWorld"))
});
```

JavaScript

```
const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
  definition: new tasks.LambdaInvoke(this, "MyLambdaTask", {
    lambdaFunction: helloFunction
  }).next(new sfn.Succeed(this, "GreetedWorld"))
});
```

Python

```
state_machine = sfn.StateMachine(
                                 self, "MyStateMachine",
                                 definition=tasks.LambdaInvoke(
                                 self, "MyLambdaTask",
                                 lambda_function=hello_function)
                                 .next(sfn.Succeed(self, "GreetedWorld")))
```

Java

```
final StateMachine stateMachine = StateMachine.Builder.create(this,
 "MyStateMachine")
        .definition(LambdaInvoke.Builder.create(this, "MyLambdaTask")
            .lambdaFunction(helloFunction)
            .build()
            .next(new Succeed(this, "GreetedWorld")))
        .build();
```

C#

```
var stateMachine = new StateMachine(this, "MyStateMachine", new StateMachineProps {
    DefinitionBody = DefinitionBody.FromChainable(new LambdaInvoke(this,
 "MyLambdaTask", new LambdaInvokeProps
    {
        LambdaFunction = helloFunction
    })
    .Next(new Succeed(this, "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.

1. Update the stack as shown in the following examples.

   TypeScript

   Update `lib/step-stack.ts` with the following code.

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as sfn from 'aws-cdk-lib/aws-stepfunctions';
import * as tasks from 'aws-cdk-lib/aws-stepfunctions-tasks';

export class StepStack extends cdk.Stack {
  constructor(app: cdk.App, id: string) {
    super(app, id);

    const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
      code: lambda.Code.fromInline(`
          exports.handler = (event, context, callback) => {
              callback(null, "Hello World!");
          };
      `),
      runtime: lambda.Runtime.NODEJS_18_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.

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as sfn from 'aws-cdk-lib/aws-stepfunctions';
import * as tasks from 'aws-cdk-lib/aws-stepfunctions-tasks';

export class StepStack extends cdk.Stack {
  constructor(app, id) {
    super(app, id);

    const helloFunction = new lambda.Function(this, 'MyLambdaFunction', {
      code: lambda.Code.fromInline(`
          exports.handler = (event, context, callback) => {
              callback(null, "Hello World!");
          };
      `),
      runtime: lambda.Runtime.NODEJS_18_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"))
    });
  }
}
```

Python

Update `step/step_stack.py` with the following code.

```
from aws_cdk import (
    Duration,
    Stack,
    aws_stepfunctions as sfn,
    aws_stepfunctions_tasks as tasks,
    aws_lambda as lambda_
)
class StepStack(Stack):

    def __init__(self, scope: 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_18_X,
                handler="index.handler",
                timeout=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.

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.Duration;
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
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);

        final Function helloFunction = Function.Builder.create(this,
 "MyLambdaFunction")
                .code(Code.fromInline(
                        "exports.handler = (event, context, callback) =>
 { callback(null, 'Hello World!' );}"))
                .runtime(Runtime.NODEJS_18_X)
                .handler("index.handler")
                .timeout(Duration.seconds(25))
                .build();

        final StateMachine stateMachine = StateMachine.Builder.create(this,
 "MyStateMachine")
                .definition(LambdaInvoke.Builder.create(this, "MyLambdaTask")
                        .lambdaFunction(helloFunction)
                        .build()
                        .next(new Succeed(this, "GreetedWorld")))
                .build();
    }
}
```

C#

Update `src/Step/StepStack.cs` with the following code.

```
using Amazon.CDK;
using Constructs;
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_18_X,
                Handler = "index.handler",
                Timeout = Duration.Seconds(25)
            });

            var stateMachine = new StateMachine(this, "MyStateMachine", new
 StateMachineProps
            {
                DefinitionBody = DefinitionBody.FromChainable(new
 LambdaInvoke(this, "MyLambdaTask", new LambdaInvokeProps
                {
                    LambdaFunction = helloFunction
                })
                .Next(new Succeed(this, "GreetedWorld")))
            });
        }
    }
}
```

2. Save the source file, and then run the `cdk synth` command in the app's main directory.

   AWS CDK runs the app and synthesizes an AWS CloudFormation template from it. AWS CDK then displays the template.

   > **Note**
   >
   > If you used TypeScript to create your AWS CDK project, running the `cdk synth` command may return the following error.
   >
   > ```
   > TSError: # Unable to compile TypeScript:
   > bin/step.ts:7:33 - error TS2554: Expected 2 arguments, but got 3.
   > ```
   >
   > Modify the `bin/step.ts` file as shown in the following example to resolve this error.
   >
   > ```
   > #!/usr/bin/env node
   > import 'source-map-support/register';
   > import * as cdk from 'aws-cdk-lib';
   > import { StepStack } from '../lib/step-stack';
   > ```

```
const app = new cdk.App();
new StepStack(app, 'StepStack');
app.synth();
```

3. To 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 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 CDK.
2. On the state machine page, choose **Start execution**.

   The **Start execution** dialog box is displayed.
3. (Optional) Enter a custom execution name to override the generated default.

   > **Non-ASCII names and logging**
   >
   > Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

4. Choose **Start Execution**.

   Your state machine's execution starts, and a new page showing your running execution is displayed.
5. The Step Functions console directs you to a page that's titled with your execution ID. This page is known as the *Execution Details* page. On this page, you can review the execution results as the execution progresses or after it's complete.

   To review the execution results, choose individual states on the **Graph view**, and then choose the individual tabs on the [Step details](#) pane to view each state's details including input, output, and definition respectively. For details about the execution information you can view on the *Execution Details* page, see Execution details overview.

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 AWS CDK, see the AWS CDK Developer Guide.

For information about writing AWS CDK apps in your language of choice, see:

TypeScript

Working with AWS CDK in TypeScript

JavaScript

Working with AWS CDK in JavaScript

Python

Working with AWS CDK in Python

Java

Working with AWS CDK in Java

C#

Working with AWS CDK in C#

For more information about the AWS Construct Library modules used in this tutorial, see the following AWS CDK API Reference overviews:

- aws-lambda
- aws-stepfunctions
- aws-stepfunctions-tasks

Using AWS CDK to create an Express workflow in Step Functions

In this tutorial, you learn how to create an API Gateway REST API with a synchronous express state machine as the backend integration, using the AWS Cloud Development Kit (AWS CDK) Infrastructure as Code (IAC) framework.

You 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.

With AWS CDK is an Infrastructure as Code (IAC) framework, you define AWS infrastructure using a programming language. You define an app in one of the CDK's supported languages, synthesize the code into an AWS CloudFormation template, and then deploy the infrastructure to your AWS account.

You will use AWS CloudFormation 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 starting 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:

```
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

```
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language typescript
```

JavaScript

```
mkdir stepfunctions-rest-api
```

```
cd stepfunctions-rest-api
cdk init --language javascript
```

Python

```
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.

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language java
```

C#

```
mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language csharp
```

Go

```
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

```
npm install @aws-cdk/aws-stepfunctions @aws-cdk/aws-apigateway
```

JavaScript

```
npm install @aws-cdk/aws-stepfunctions @aws-cdk/aws-apigateway
```

Python

```
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.

```
        <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#

```
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

```
@aws-cdk/aws-stepfunctions
@aws-cdk/aws-apigateway
```

JavaScript

```
@aws-cdk/aws-stepfunctions
@aws-cdk/aws-apigateway
```

Python

```
aws_cdk.aws_stepfunctions
aws_cdk.aws_apigateway
```

Java

```
software.amazon.awscdk.services.apigateway.StepFunctionsRestApi
software.amazon.awscdk.services.stepfunctions.Pass
software.amazon.awscdk.services.stepfunctions.StateMachine
software.amazon.awscdk.services.stepfunctions.StateMachineType
```

C#

```
Amazon.CDK.AWS.StepFunctions
Amazon.CDK.AWS.APIGateway
```

Go

Add the following to `import` inside `stepfunctions-rest-api.go`.

```
"github.com/aws/aws-cdk-go/awscdk/awsapigateway"
"github.com/aws/aws-cdk-go/awscdk/awsstepfunctions"
```

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

```
const machineDefinition = new stepfunctions.Pass(this, 'PassState', {
    result: {value:"Hello!"},
})

const stateMachine = new stepfunctions.StateMachine(this, 'MyStateMachine', {
    definition: machineDefinition,
    stateMachineType: stepfunctions.StateMachineType.EXPRESS,
});
```

JavaScript

```
const machineDefinition = new sfn.Pass(this, 'PassState', {
    result: {value:"Hello!"},
})

const stateMachine = new sfn.StateMachine(this, 'MyStateMachine', {
    definition: machineDefinition,
    stateMachineType: stepfunctions.StateMachineType.EXPRESS,
});
```

Python

```
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)
```

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#

```
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

```
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: awsstepfunctions.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
{
    StateMachine = stateMachine,
})
```

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.

```
import * as cdk from 'aws-cdk-lib';
import * as stepfunctions from 'aws-cdk-lib/aws-stepfunctions'
import * as apigateway from 'aws-cdk-lib/aws-apigateway';


export class StepfunctionsRestApiStack extends cdk.Stack {
    constructor(scope: cdk.App, 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.

```
const cdk = require('@aws-cdk/core');
const stepfunctions = require('@aws-cdk/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 sfn.StateMachine(this, 'MyStateMachine', {
        definition: machineDefinition,
        stateMachineType: stepfunctions.StateMachineType.EXPRESS,
    });

    const api = new apigateway.StepFunctionsRestApi(this,
        'StepFunctionsRestApi', { stateMachine: stateMachine });

    }
}

module.exports = { StepStack }
```

Python

Update `stepfunctions_rest_api/stepfunctions_rest_api_stack.py` to read as follows.

```
from aws_cdk import App, Stack
from constructs import Construct
```

```
from aws_cdk import aws_stepfunctions as sfn
from aws_cdk import aws_apigateway as apigw

class StepfunctionsRestApiStack(Stack):

    def __init__(self, scope: 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.

```
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;

public 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();

        StepFunctionsRestApi api = StepFunctionsRestApi.Builder.create(this,
 "StepFunctionsRestApi")
                                   .stateMachine(stateMachine)
                                   .build();

    }
}
```

C#

Update `src/StepfunctionsRestApi/StepfunctionsRestApiStack.cs` to read as follows.

```
using Amazon.CDK;
using Amazon.CDK.AWS.StepFunctions;
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

Update `stepfunctions-rest-api.go` to read as follows.

```
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/v3"
    "github.com/aws/jsii-runtime-go"
)


type StepfunctionsRestApiGoStackProps struct {
    awscdk.StackProps
}

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")),
    })

    var stateMachine = awsstepfunctions.NewStateMachine(stack,
 jsii.String("StateMachine"), &awsstepfunctions.StateMachineProps{
        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("account-id"),
    //  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{
    //  Account: jsii.String(os.Getenv("CDK_DEFAULT_ACCOUNT")),
    //  Region:  jsii.String(os.Getenv("CDK_DEFAULT_REGION")),
    // }
}
```

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.

Step 3: Test the API Gateway

After you create your API Gateway REST API with Synchronous Express State Machine as the backend integration, you can test the API Gateway.

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 ANY method.
4. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
5. For **Method**, choose **POST**.
6. For **Request body**, copy the following request parameters.

```
{
    "key": "Hello"
}
```

7. 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.
   - **Log** shows 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 API ID and `<region>` with the region where your API is deployed.

```
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.

Using Terraform to deploy state machines in Step Functions

Terraform by HashiCorp is a framework for building applications using infrastructure as code (IaC). With Terraform, you can create state machines and use features, such as previewing infrastructure deployments and creating reusable templates. Terraform templates help you maintain and reuse the code by breaking it down into smaller chunks.

If you're familiar with Terraform, you can follow the development lifecycle described in this topic as a model for creating and deploying your state machines in Terraform. If you aren't familiar with Terraform, we recommend that you first complete the workshop Introduction to Terraform on AWS for getting acquainted with Terraform.

> **Tip**
>
> To deploy an example of a state machine built using Terraform, see Deploy with Terraform in *The AWS Step Functions Workshop*.

In this topic

- Prerequisites
- State machine development lifecycle with Terraform
- IAM roles and policies for your state machine

Prerequisites

Before you get started, make sure you complete the following prerequisites:

- Install Terraform on your machine. For information about installing Terraform, see Install Terraform.
- Install Step Functions Local on your machine. We recommend that you install the Step Functions Local Docker image to use Step Functions Local. For more information, see Testing state machines with Step Functions Local (unsupported).
- Install AWS SAM CLI. For installation information, see Installing the AWS SAM CLI in the *AWS Serverless Application Model Developer Guide*.
- Install the AWS Toolkit for Visual Studio Code to view the workflow diagram of your state machines. For installation information, see Installing the AWS Toolkit for Visual Studio Code in the *AWS Toolkit for Visual Studio Code User Guide*.

State machine development lifecycle with Terraform

The following procedure explains how you can use a state machine prototype that you build using Workflow Studio in the Step Functions console as a starting point for local development with Terraform and the AWS Toolkit for Visual Studio Code.

To view the complete example that discusses the state machine development with Terraform and presents the best practices in detail, see Best practices for writing Step Functions Terraform projects.

To start the development lifecycle of a state machine with Terraform

1. Bootstrap a new Terraform project with the following command.

   ```
   terraform init
   ```

2. Open the Step Functions console to create a prototype for your state machine.

3. In Workflow Studio, do the following:

   a. Create your workflow prototype.

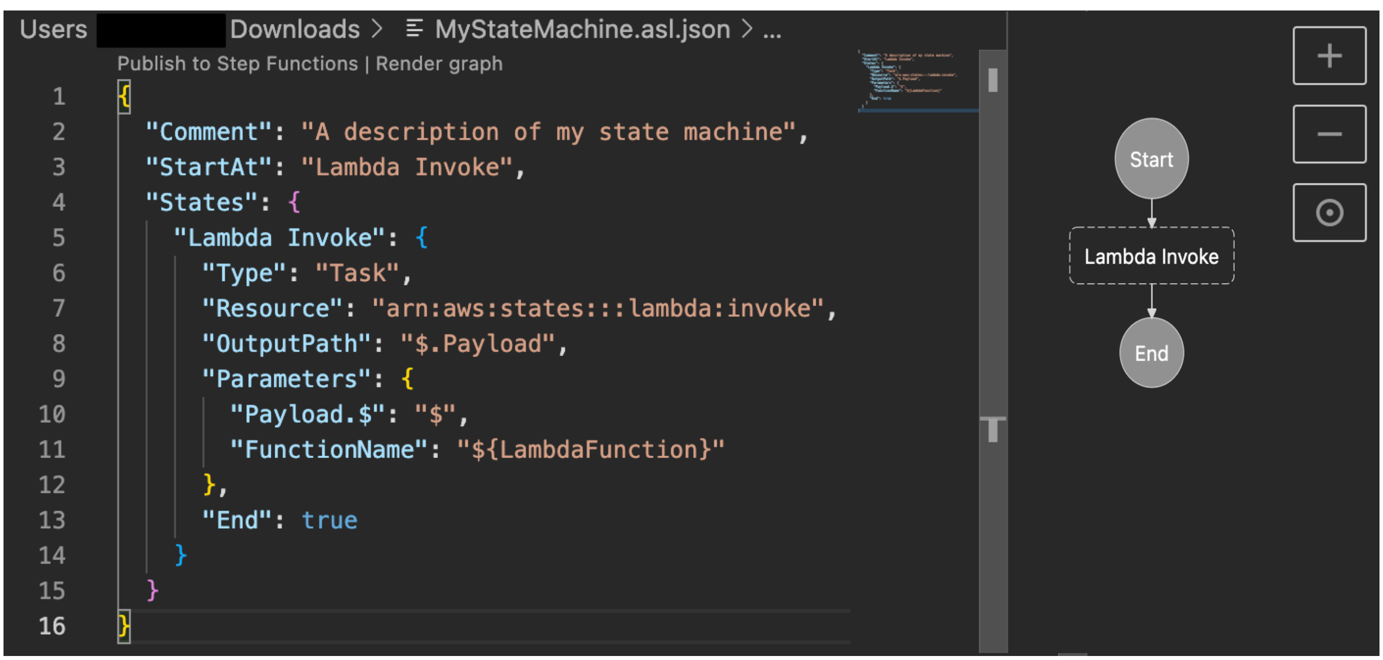
   b. Export the [Amazon States Language (ASL)](#) definition of your workflow. To do this, choose the **Import/Export** dropdownlist, and then select **Export JSON definition**.

4. Save the exported ASL definition within your project directory.

   You pass the exported ASL definition as an input parameter to the `aws_sfn_state_machine` Terraform resource that uses the `templatefile` function. This function is used inside the definition field that passes the exported ASL definition and any variable substitutions.

   > **Tip**
   >
   > Because the ASL definition file can contain lengthy blocks of text, we recommend you avoid the inline EOF method. This makes it easier to substitute parameters into your state machine definition.

5. (Optional) Update the ASL definition within your IDE and visualize your changes using the AWS Toolkit for Visual Studio Code.



   To avoid continuously exporting your definition and refactoring it into your project, we recommend that you make updates locally in you IDE and track these updates with [Git](#).

6. Test your workflow using [Step Functions Local](#).

> **Tip**
>
> You can also locally test service integrations with Lambda functions and API Gateway APIs in your state machine using AWS SAM CLI Local.

7. Preview your state machine and other AWS resources before deploying the state machine. To do this, run the following command.

```
terraform plan
```

8. Deploy your state machine from your local environment or through CI/CD pipelines using the following command.

```
terraform apply
```

9. (Optional) Clean up your resources and delete the state machine using the following command.

```
terraform destroy
```

IAM roles and policies for your state machine

Use the Terraform service integration policies to add necessary IAM permissions to your state machine, for example, permission to invoke Lambda functions. You can also define explicit roles and policies and associate them with your state machine.

The following IAM policy example grants your state machine access to invoke a Lambda function named *myFunction*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:myFunction"
```

```
    }
  ]
}
```

We also recommend using the [aws_iam_policy_document] data source when defining IAM policies for your state machines in Terraform. This helps you check if your policy is malformed and substitute any resources with variables.

The following IAM policy example uses the `aws_iam_policy_document` data source and grants your state machine access to invoke a Lambda function named *myFunction*.

```
data "aws_iam_policy_document" "state_machine_role_policy" {

  statement {
    effect = "Allow"

    actions = [
      "lambda:InvokeFunction"
    ]

    resources = ["${aws_lambda_function.function-1.arn}:*"]
  }

}
```

Tip

To view more advanced AWS architectural patterns deployed with Terraform, see Terraform examples at Serverless Land Workflows Collection.

Exporting your workflow to IaC templates

The AWS Step Functions console provides the ability to export and download saved workflows as AWS CloudFormation or AWS SAM (SAM) templates. For AWS Regions that support AWS Infrastructure Composer, it additionally provides the ability to export your workflows to Infrastructure Composer and navigates to the Infrastructure Composer console, where you can continue to work with the newly generated template.

Template configuration options

The following options are available with this feature. If you select to export and download an IaC template file, the console displays the options that apply to your saved state machine for selection. If you're exporting to Infrastructure Composer, the Step Functions console automatically implements the configurations that apply to your state machine.

- **Include IAM role created by console on your behalf –** This option exports the execution role policies. It constructs an IAM role in the template and attaches it to the state machine resource. This option is only applicable if the state machine has an execution role that's created by the console.
- **Include CloudWatch Log Group –** Constructs a CloudWatch log group in the template and attaches it to the state machine resource. This option is only applicable if the state machine has a CloudWatch log group attached to it and the log level is *not* set to `OFF`.
- **Replace resource references with DefinitionSubstitutions –** This option generates DefinitionSubstitutions for the following components:
  - Distributed Map S3 fields.
  - `Activity` resources. The export includes `Activity` resources in the AWS CloudFormation template for any `Run Activity` task. The export also provides `DefinitionSubstitutions` referencing the created `Activity` resources.
  - Any `ARN` or `S3URI` in the Payload field for all service integrations.
  - In addition to the `ARN` and `S3URI` fields, the export generates `DefinitionSubstitutions` for other frequently used service integration payload fields. The specific service integrations are the following:
    - `athena:startQueryExecution`
    - `batch:submitJob`
    - `dynamodb:getItem`, `dynamodb:updateItem`, `dynamodb:updateItem`, `dynamodb:deleteItem`
    - `ecs:runTask`
    - `glue:startJobRun`
    - `http:invoke`
    - `lambda:invoke`
    - `sns:publish`
    - `sqs:sendMessage`

- `states:startExecution`

Export and download your workflow's IaC template

To export your workflow into an IaC template file

1. Open the Step Functions console and select the state machine you want to work with. Make sure that any changes to the state machine are saved before you proceed to the next step.
2. Select **Export to CloudFormation or SAM template** from the **Actions** menu.
3. Select **Type** as either **SAM** or **CloudFormation** from the dialog box that appears.
   - If you selected the **CloudFormation** template, next choose either the **JSON** or **YAML** file format.
   - If you selected the **SAM** template, no formats choices are presented. The SAM template defaults to YAML file format.
4. Expand **Additional configurations**. By default all of the options are selected. Review and update the selection of options for your IaC template. The options are described in detail in the previous section titled Template configuration options.

   If an option doesn't apply to your specific workflow, then it won't display in the dialogue box.
5. Choose **Download** to export and download your generated IaC template file.

Export your workflow directly into AWS Infrastructure Composer

To export your workflow into Infrastructure Composer

1. Open the Step Functions console and select the state machine you want to work with. Make sure that any changes to the state machine are saved before you proceed to the next step.
2. Select **Export to Infrastructure Composer** from the **Actions** menu.
3. The **Export to Infrastructure Composer** dialog box displays. You can use the default name that displays in the **Transfer bucket name** field or enter a new name. Amazon S3 bucket names must be globally unique and follow the bucket naming rules.
4. Choose the **Confirm and create project** to export your workflow to Infrastructure Composer.
5. To save your project and workflow definition in Infrastructure Composer, activate local sync mode.

> **Note**
>
> If you've used the **Export to Infrastructure Composer** feature before and created an Amazon S3 bucket using the default name, Step Functions can re-use this bucket if it still exists. Accept the default bucket name in the dialog box to re-use the existing bucket.

Amazon S3 transfer bucket configuration

The Amazon S3 bucket that Step Functions creates to transfer your workflow automatically encrypts objects using the AES 256 encryption standard. Step Functions also configures the bucket to use the bucket owner condition to ensure that only your AWS account is able to add objects to the bucket.

The default bucket name uses the prefix `states-templates`, a 10-digit alphanumeric string, and the AWS Region you created your workflow in: `states-templates-`*`amzn-s3-demo-bucket`*`-`*`us-east-1`*. To avoid additional charges being added to your AWS account, we recommend that you delete the Amazon S3 bucket as soon as you have finished exporting your workflow to Infrastructure Composer.

Standard Amazon S3 pricing applies.

Required permissions

To use this Step Functions export feature with Infrastructure Composer, you need certain permissions to download an AWS SAM template and to write your template configuration to Amazon S3.

To download an AWS SAM template, you must have permission to use the following API actions:

- iam:GetPolicy
- iam:GetPolicyVersion
- iam:GetRole
- iam:GetRolePolicy
- iam:ListAttachedRolePolicies
- iam:ListRolePolicies
- iam:ListRoles

For Step Functions to write your function's configuration to Amazon S3, you must have permission to use the following API actions:

- S3:PutObject
- S3:CreateBucket
- S3:PutBucketEncryption

If you are unable to export your function's configuration to Infrastructure Composer, check that your account has the required permissions for these operations.

Starting state machine 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:

- Start an execution in the Step Functions console.

  You can start a state machine in the console, watch the execution, and debug failures.
- Call the StartExecution API action.
- Use Amazon EventBridge to start an execution in response to an event.
- Use Amazon EventBridge Scheduler to start a state machine execution on a schedule.
- Start a nested workflow execution from a Task state.
- Start an execution with Amazon API Gateway.

> **Tip**
>
> To learn how to monitor running executions, see the tutorial: the section called "Examine executions"

Start workflow executions from a task state in Step Functions

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. 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.

Tip

To deploy an example nested workflow, see Optimizing costs in *The AWS Step Functions Workshop*.

To start a new execution of a state machine, use a `Task` state similar to the following example:

```
{
   "Type":"Task",
   "Resource":"arn:aws:states:::states:startExecution",
   "Parameters":{
      "StateMachineArn":"arn:aws:states:region:account-id:stateMachine:HelloWorld",
      "Input":{
         "Comment":"Hello world!"
      },
   },
   "Retry":[
      {
        "ErrorEquals":[
            "StepFunctions.ExecutionLimitExceeded"
        ]
      }
   ],
   "End":true
}
```

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

Note

The `StartExecution` API action 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

- Handling errors in Step Functions workflows

Associate Workflow Executions

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

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

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

```
{
   "Type":"Task",
   "Resource":"arn:aws:states:::states:startExecution",
   "Parameters":{
      "StateMachineArn":"arn:aws:states:region:account-id:stateMachine:HelloWorld",
      "Input": {
        "Comment": "Hello world!",
        "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
       }
   },
   "End":true
}
```

For more information, see the following:

- Integrating services
- Passing parameters to a service API in Step Functions
- Accessing the Context object
- AWS Step Functions

Using Amazon EventBridge Scheduler to start a Step Functions state machine execution

Amazon EventBridge Scheduler is a serverless scheduler that allows you to create, run, and manage tasks from one central, managed service. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for failed API invocations.

For example, with EventBridge Scheduler, you can start a state machine execution on a schedule when a security related event occurs or to automate a data processing job.

This page explains how to use EventBridge Scheduler to start execution of a Step Functions state machine on a schedule.

Topics

- Set up the execution role
- Create a schedule
- Related resources

Set up the execution role

When you create a new schedule, EventBridge Scheduler must have permission to invoke its target API operation on your behalf. You grant these permissions to EventBridge Scheduler using an *execution role*. The permission policy you attach to your schedule's execution role defines the required permissions. These permissions depend on the target API you want EventBridge Scheduler to invoke.

When you use the EventBridge Scheduler console to create a schedule, as in the following procedure, EventBridge Scheduler automatically sets up an execution role based on your selected target. If you want to create a schedule using one of the EventBridge Scheduler SDKs, the AWS CLI, or AWS CloudFormation, you must have an existing execution role that grants the permissions EventBridge Scheduler requires to invoke a target. For more information about manually setting up an execution role for your schedule, see Setting up an execution role in the *EventBridge Scheduler User Guide*.

Create a schedule

To create a schedule by using the console

1. Open the Amazon EventBridge Scheduler console at https://console.aws.amazon.com/scheduler/home.
2. On the **Schedules** page, choose **Create schedule**.
3. On the **Specify schedule detail** page, in the **Schedule name and description** section, do the following:

   a. For **Schedule name**, enter a name for your schedule. For example, **`MyTestSchedule`**.

   b. (Optional) For **Description**, enter a description for your schedule. For example, **`My first schedule`**.

   c. For **Schedule group**, choose a schedule group from the dropdown list. If you don't have a group, choose **default**. To create a schedule group, choose **create your own schedule**.

   You use schedule groups to add tags to groups of schedules.

4. • Choose your schedule options.

Occurrence	Do this...	
One-time schedule A one-time schedule invokes a target only once at the date and time that you specify.	For **Date and time**, do the following: • Enter a valid date in `YYYY/MM/DD` format. • Enter a timestamp in 24-hour `hh:mm` format. • For **Timezone**, choose the timezone.	
Recurring schedule A recurring schedule invokes a target at a rate that you specify using a	a. For **Schedule type**, do one of the following: • To use a cron expression to define the schedule, choose	

Occurrence	Do this...	
cron expression or rate expression.	**Cron-based schedule** and enter the cron expression. • To use a rate expression to define the schedule, choose **Rate-based schedule** and enter the rate expression. For more informati on about cron and rate expressions, see [Schedule types on EventBridge Scheduler](#) in the *Amazon EventBridge Scheduler User Guide*. b. For **Flexible time window**, choose **Off** to turn off the option, or choose one of the pre-defined time windows. For example, if you choose **15 minutes** and you set a recurring schedule to invoke its target once every hour, the schedule runs within 15 minutes after the start of every hour.	

5. (Optional) If you chose **Recurring schedule** in the previous step, in the **Timeframe** section, do the following:

a. For **Timezone**, choose a timezone.
b. For **Start date and time**, enter a valid date in `YYYY/MM/DD` format, and then specify a timestamp in 24-hour `hh:mm` format.
c. For **End date and time**, enter a valid date in `YYYY/MM/DD` format, and then specify a timestamp in 24-hour `hh:mm` format.

6. Choose **Next**.
7. On the **Select target** page, choose the AWS API operation that EventBridge Scheduler invokes:

a. Choose **AWS Step Functions StartExecution**.
b. In the **StartExecution** section, select a state machine or choose **Create new state machine**.

Currently, you can't run Synchronous Express workflows on a schedule.

c. Enter a JSON payload for the execution. Even if your state machine doesn't require any JSON payload, you must still include input in JSON format as shown in the following example.

```
{
    "Comment": "sampleJSONData"
}
```

8. Choose **Next**.
9. On the **Settings** page, do the following:

a. To turn on the schedule, under **Schedule state**, toggle **Enable schedule**.
b. To configure a retry policy for your schedule, under **Retry policy and dead-letter queue (DLQ)**, do the following:

- Toggle **Retry**.
- For **Maximum age of event**, enter the maximum **hour(s)** and **min(s)** that EventBridge Scheduler must keep an unprocessed event.
- The maximum time is 24 hours.
- For **Maximum retries**, enter the maximum number of times EventBridge Scheduler retries the schedule if the target returns an error.

The maximum value is 185 retries.

With retry policies, if a schedule fails to invoke its target, EventBridge Scheduler re-runs the schedule. If configured, you must set the maximum retention time and retries for the schedule.

c. Choose where EventBridge Scheduler stores undelivered events.

Dead-letter queue (DLQ) option	**Do this...**
Don't store	Choose **None**.
Store the event in the same AWS account where you're creating the schedule	a. Choose **Select an Amazon SQS queue in my AWS account as a DLQ**. b. Choose the Amazon Resource Name (ARN) of the Amazon SQS queue.
Store the event in a different AWS account from where you're creating the schedule	a. Choose **Specify an Amazon SQS queue in other AWS accounts as a DLQ**. b. Enter the Amazon Resource Name (ARN) of the Amazon SQS queue.

d. To use a customer managed key to encrypt your target input, under **Encryption**, choose **Customize encryption settings (advanced)**.

If you choose this option, enter an existing KMS key ARN or choose **Create an AWS KMS key** to navigate to the AWS KMS console. For more information about how EventBridge Scheduler encrypts your data at rest, see Encryption at rest in the *Amazon EventBridge Scheduler User Guide*.

e. To have EventBridge Scheduler create a new execution role for you, choose **Create new role for this schedule**. Then, enter a name for **Role name**. If you choose this option, EventBridge Scheduler attaches the required permissions necessary for your templated target to the role.

10. Choose **Next**.
11. In the **Review and create schedule** page, review the details of your schedule. In each section, choose **Edit** to go back to that step and edit its details.
12. Choose **Create schedule**.

    You can view a list of your new and existing schedules on the **Schedules** page. Under the **Status** column, verify that your new schedule is **Enabled**.

To confirm that EventBridge Scheduler invoked the state machine, check the state machine's Amazon CloudWatch logs.

Related resources

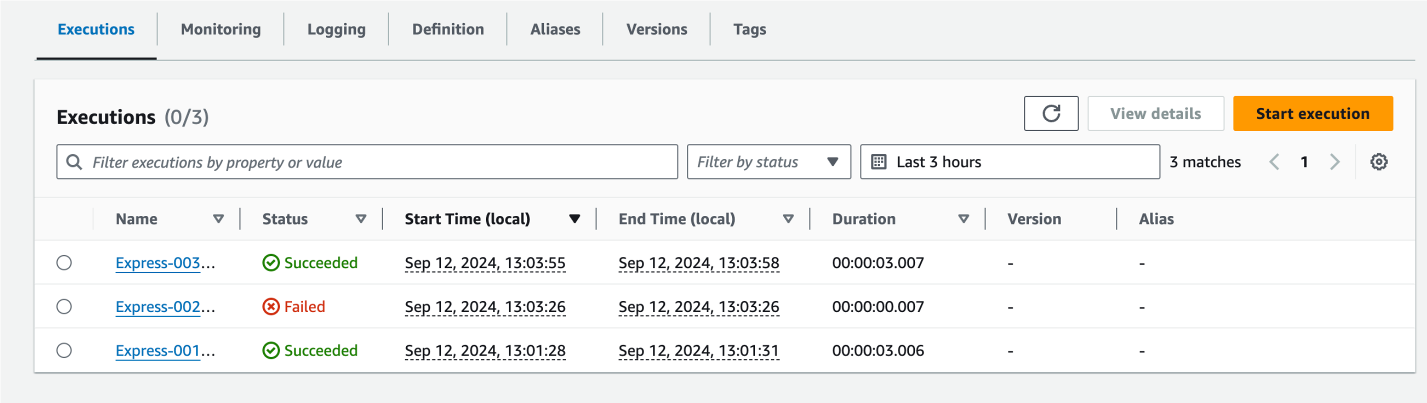
For more information about EventBridge Scheduler, see the following:

- EventBridge Scheduler User Guide
- EventBridge Scheduler API Reference
- EventBridge Scheduler Pricing

Viewing execution details in the Step Functions console

You can view in-progress and past executions of workflows in the *Executions* section of the Step Functions console.

In the *Executions* details, you can view the state machine's definition, execution status, ARN, number of state transitions, and the inputs and outputs for individual states in the workflow.



Standard workflow execution details are recorded in Step Functions, but the history of Express workflow executions are not. To record Express workflow executions, you must configure your Express state machines to send logs to Amazon CloudWatch. See Logging in CloudWatch Logs to set up logging for Step Functions.

The console experience to view both types of workflow executions is similar, but there are some limitations for Express workflows. See the section called “Standard and Express differences”.

Note

Because execution data for Express workflows are displayed using CloudWatch Logs Insights, scanning the logs will incur charges. By default, your log group only lists executions completed in the last three hours. If you specify a larger time range that includes more execution events, your costs will increase. For more information, see **Vended Logs** under the **Logs** tab on the CloudWatch Pricing page.

Execution details overview

The execution details link and page title use the unique execution ID generated by Step Functions or the custom ID you provided when starting the workflow. The *Execution Details* page includes metrics and the following options to manage your state machine:

- **Stop execution** – Stop an in-progress execution. (Unavailable for completed executions.)
- **Start new execution** – Start a new execution of your state machine
- **Redrive** – Redrive executions of Standard Workflows that did not complete successfully in the last 14 days, including failed, aborted, or timed out executions. For more information, see Redriving state machines.

- **Export –** Export the execution details in JSON format to share or perform offline analysis.

> **Viewing executions started with a version or alias**
>
> You can also view the executions started with a version or an alias in the Step Functions console. For more information, see Listing executions for versions and aliases.

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

1. Execution summary
2. Error message
3. View mode
4. Step details
5. Events

Execution summary

The *Execution summary* provides an overview of the execution details of your workflow, in the following 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 running the state machine execution. You can also 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 started by another state machine, you can view the link for the parent state machine on this tab.

If your state machine execution was redriven, this tab displays redrive related information, for example **Redrive count**.

Execution input and output

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

Definition

Shows the state machine's Amazon States Language definition.

Error message

If your state machine execution failed, the *Execution Details* page displays an error message. Choose **Cause** or **View step details** in the error message to view the reason for execution failure or the step that caused the error.

If you choose **View step details**, Step Functions highlights the step that caused the error in the Step details, Graph view, and Table view tabs. If the step is a Task, Map, or Parallel state for which you've defined retries, the **Step details** pane displays the **Retry** tab for the step. Additionally, if you've redriven the execution, you can see the retries and redrive execution details in the **Retries & redrives** tab of the **Step details** pane.

From the **Recover** dropdown button on this error message, you can either redrive your unsuccessful executions or start a new execution. For more information, see Redriving state machines.

The error message for a failed state machine execution will be displayed on the *Execution Details* page. The error message will also have a link to the step that caused the execution failure.

View mode

The *View mode* section contains two 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:

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 align the full workflow, or view the workflow in full-screen mode.

From the graph view, you can choose any step in your workflow to view details about its execution in the *Step details* component. When you chose a step in the **Graph view**, the **Table view** also shows that step. This is true in reverse as well. If you choose a step from **Table view**, the **Graph view** shows the same step.

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. This is because iterations are counted from zero. Likewise, choose **#3** from the dropdown list for the fourth iteration of the **Map** state.

   Alternatively, use the up arrow icon and down arrow icon controls to move between 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 to represent the iteration data.

3. (Optional) If one or more of your **Map** state iterations failed to execute, or the execution was stopped, you can view its 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. You can also 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.

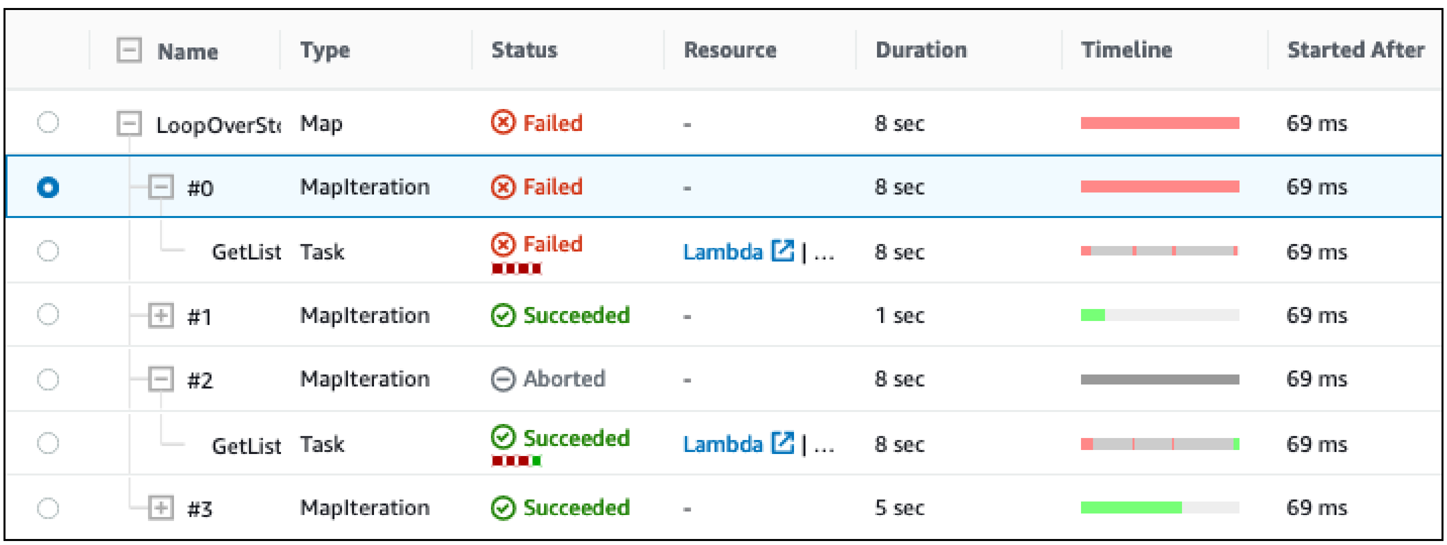
From this view, you can choose any state in your workflow to view details about its execution in the *Step details* component. When you chose a step in the **Table view**, the **Graph view** also shows that step. This is true in reverse as well. If you choose a step from **Graph view**, the **Table view** shows the same step.

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 **Status** or **Redrive attempt**. For more information, see Examine executions.

By default, this mode displays the **Name**, **Type**, **Status**, **Resource**, and **Started After** columns. 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 until they are changed again.

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.



Name	Type	Status	Resource	Duration	Timeline	Started After
LoopOverSt	Map	Failed	-	8 sec		69 ms
#0	MapIteration	Failed	-	8 sec		69 ms
GetList	Task	Failed	Lambda | ...	8 sec		69 ms
#1	MapIteration	Succeeded	-	1 sec		69 ms
#2	MapIteration	Aborted	-	8 sec		69 ms
GetList	Task	Succeeded	Lambda | ...	8 sec		69 ms
#3	MapIteration	Succeeded	-	5 sec		69 ms

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* 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.

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

Shows the input details of the selected state. If there is an error in the input, it is indicated with a error icon on the tab header. In addition, you can view the reason for the error in this tab.

You can also choose the **Advanced view** toggle button to see the input 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.

Output

Shows the output of the selected state. If there is an error in the output, it is indicated with an error icon on the tab header. In addition, you can view the reason for the error in the this tab.

You can also choose the **Advanced view** toggle button to see the 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 have defined a `Retry` field in your state machine's `Task` or `Parallel` state.

Shows the initial and subsequent retry attempts for a selected state in its original execution attempt. For the initial and all the subsequent failed attempts, choose the arrow icon next to **Type** 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.

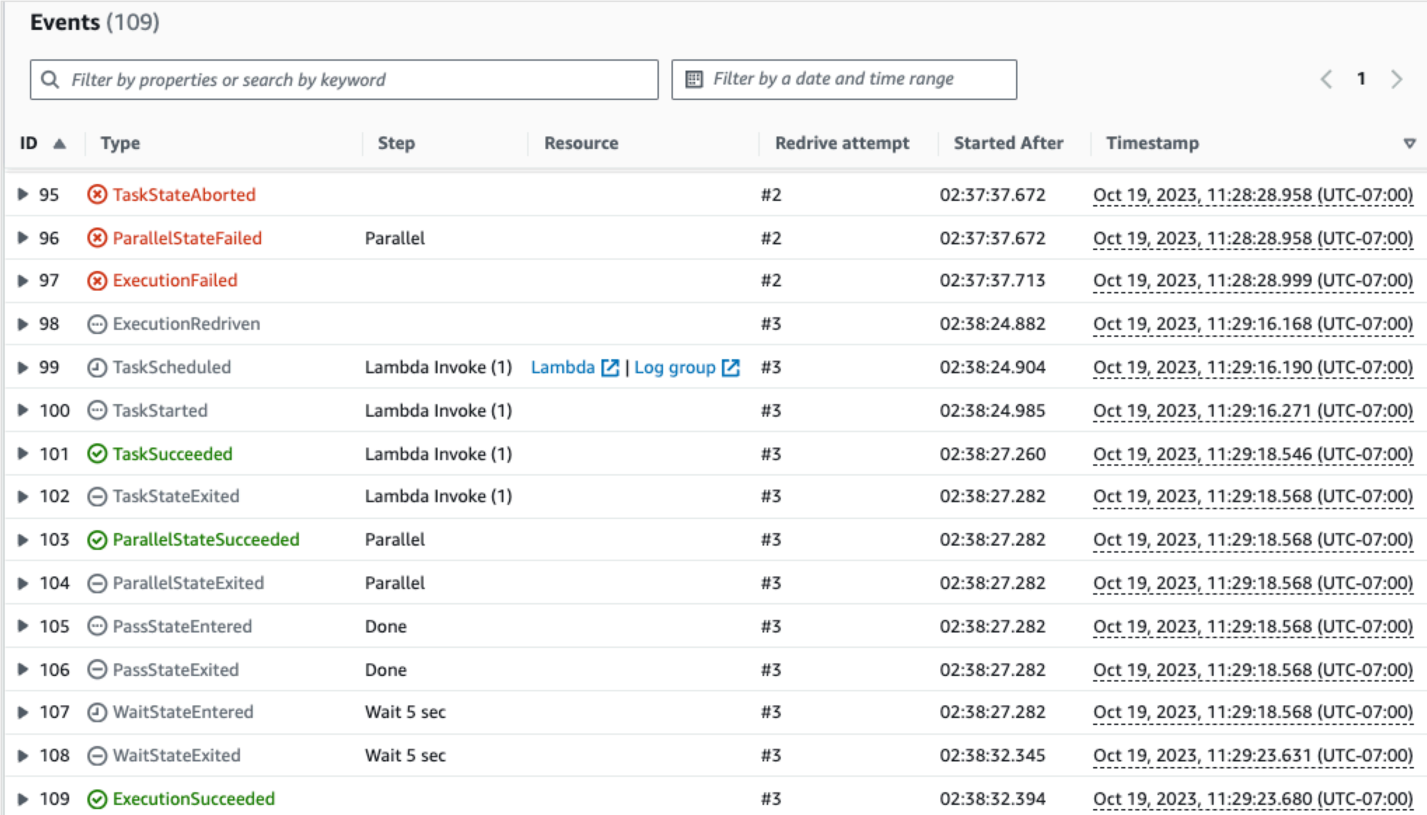
If you've redriven your execution, this tab header displays the name **Retries & redrives** and displays the retry attempt details for each redrive.

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 *Events* table.

Events

The **Events** table displays the complete history for the selected execution as a list of events spanning multiple pages. Each page contains up to 25 events. This section also displays the total event count, which can help you determine if you exceeded the maximum event history count of 25,000 events.



Events (109)

ID	Type	Step	Resource	Redrive attempt	Started After	Timestamp
95	TaskStateAborted			#2	02:37:37.672	Oct 19, 2023, 11:28:28.958 (UTC-07:00)
96	ParallelStateFailed	Parallel		#2	02:37:37.672	Oct 19, 2023, 11:28:28.958 (UTC-07:00)
97	ExecutionFailed			#2	02:37:37.713	Oct 19, 2023, 11:28:28.999 (UTC-07:00)
98	ExecutionRedriven			#3	02:38:24.882	Oct 19, 2023, 11:29:16.168 (UTC-07:00)
99	TaskScheduled	Lambda Invoke (1)	Lambda | Log group	#3	02:38:24.904	Oct 19, 2023, 11:29:16.190 (UTC-07:00)
100	TaskStarted	Lambda Invoke (1)		#3	02:38:24.985	Oct 19, 2023, 11:29:16.271 (UTC-07:00)
101	TaskSucceeded	Lambda Invoke (1)		#3	02:38:27.260	Oct 19, 2023, 11:29:18.546 (UTC-07:00)
102	TaskStateExited	Lambda Invoke (1)		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
103	ParallelStateSucceeded	Parallel		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
104	ParallelStateExited	Parallel		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
105	PassStateEntered	Done		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
106	PassStateExited	Done		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
107	WaitStateEntered	Wait 5 sec		#3	02:38:27.282	Oct 19, 2023, 11:29:18.568 (UTC-07:00)
108	WaitStateExited	Wait 5 sec		#3	02:38:32.345	Oct 19, 2023, 11:29:23.631 (UTC-07:00)
109	ExecutionSucceeded			#3	02:38:32.394	Oct 19, 2023, 11:29:23.680 (UTC-07:00)

By default, the results in the **Events** table 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 **Events** table, each event is color-coded to indicate its execution status. For example, events that failed appear in red. To view additional details about an event, choose the arrow icon next to the event ID. Once open, the event details show the input, output, and resource invocation for the event.

In addition, in the **Events** table, you can apply filters to limit the execution event history results that are displayed. You can choose properties such as **ID**, or **Redrive attempt**. For more information, see Examine executions.

Standard and Express console experience differences

Standard workflows

The execution histories for Standard Workflows are always available for executions completed in the last 90 days.

Express workflows

For Express workflows, the Step Functions console retrieves log data gathered through a CloudWatch Logs log group to show execution history. The histories for executions completed in the last **three hours** are available by default. You can customize the time range. If you specify a larger time range which includes more execution events, the cost to scan the logs will increase. For more information, see **Vended Logs** under the **Logs** tab on the CloudWatch Pricing page and Logging in CloudWatch Logs.

Considerations and limitations for viewing Express workflow executions

When viewing Express workflow executions on the Step Functions console, keep in mind the following considerations and limitations:

Availability of Express workflow execution details relies on Amazon CloudWatch Logs

For Express workflows, their execution history and detailed execution information are gathered through CloudWatch Logs Insights. This information is kept in the CloudWatch Logs log group that you specify when you create the state machine. The state machine's execution history is shown under the **Executions** tab on the Step Functions console.

> **Warning**
>
> If you delete the CloudWatch Logs for an Express workflow, it won't be listed under the **Executions** tab.

We recommend that you use the default log level of **ALL** for logging all execution event types. You can update the log level as required for your existing state machines when you edit them. For more information, see Using CloudWatch Logs to log execution history in Step Functions and Event log levels.

Partial Express workflow execution details are available if logging level is ERROR or FATAL

By default, the logging level for Express workflow executions is set to **ALL**. If you change the log level, the execution histories and execution details for completed executions won’t be affected. However, all new executions will emit logs based on the updated log level. For more information, see Using CloudWatch Logs to log execution history in Step Functions and Event log levels.

For example, if you change the log level from **ALL** to either **ERROR** or **FATAL**, the **Executions** tab on the Step Functions console only lists failed executions. In the **Event view** tab, the console shows only the event details for the state machine steps that failed.

We recommend that you use the default log level of **ALL** for logging all execution event types. You can update the log level as required for your existing state machines when you edit the state machine.

State machine definition for a prior execution can't be viewed after the state machine has been modified

State machine definitions for past executions are not stored for Express workflows. If you change your state machine definition, you can only view the state machine definition for executions using the most current definition.

For example, if you remove one or more steps from your state machine definition, Step Functions detects a mismatch between the definition and prior execution events. Because previous definitions are not stored for Express workflows, Step Functions can't display the state machine definition for executions run on an earlier version of the state machine definition. As a result, the **Definition**, **Graph view**, and **Table view** tabs are unavailable for executions run on previous versions of a state machine definition.

Restarting state machine executions with redrive in Step Functions

You can use redrive to restart executions of Standard Workflows that didn't complete successfully in the last 14 days. These include failed, aborted, or timed out executions.

When you redrive an execution, Step Functions continues the failed execution from the unsuccessful step and uses the same input. Step Functions preserves the results and execution history of the successful steps, which are not rerun when you redrive an execution. For example, say that your workflow contains two states: a Pass workflow state state followed by a Task workflow state state. If your workflow execution fails at the Task state, and you redrive the execution, the execution reschedules and then reruns the Task state.

Redriven executions use the same state machine definition and execution ARN that was used for the original execution attempt. If your original execution attempt was associated with a version, alias, or both, the redriven execution is associated with the same version, alias, or both. Even if you update your alias to point to a different version, the redriven execution continues to use the version associated with the original execution attempt. Because redriven executions use the same state machine definition, you must start a new execution if you update your state machine definition.

When you redrive an execution, the state machine level timeout, if defined, is reset to 0. For more information about state machine level timeout, see `TimeoutSeconds`.

Execution redrives are considered as state transitions. For information about how state transitions affect billing, see Step Functions Pricing.

Redrive eligibility for unsuccessful executions

You can redrive executions if your original execution attempt meets the following conditions:

- You started the execution on or after November 15, 2023. Executions that you started prior to this date aren't eligible for redrive.
- The execution status isn't `SUCCEEDED`.
- The workflow execution hasn't exceeded the redrivable period of 14 days. Redrivable period refers to the time during which you can redrive a given execution. This period starts from the day a state machine completes its execution.
- The workflow execution hasn't exceeded the maximum open time of one year. For information about state machine execution quotas, see Quotas related to state machine executions.
- The execution event history count is less than 24,999. Redriven executions append their event history to the existing event history. Make sure your workflow execution contains less than 24,999 events to accommodate the `ExecutionRedriven` history event and at least one other history event.

Redrive behavior of individual states

Depending on the state that failed in your workflow, the redrive behavior for all unsuccessful states varies. The following table describes the redrive behavior for all the states.

State name	Redrive execution behavior
Pass workflow state	If a preceding step fails or the state machine times out, the Pass state is exited and isn't executed on redrive.

State name	Redrive execution behavior
Task workflow state	Schedules and starts the Task state again. When you redrive an execution that reruns a Task state, the `TimeoutSeconds` for the state, if defined, is reset to 0. For more information about timeout, see Task state.
Choice workflow state	Reevaluates the Choice state rules.
Wait workflow state	If the state specifies `Timestamp` or `TimestampPath` that refers to a timestamp in the past, redrive causes the Wait state to be exited and enters the state specified in the `Next` field.
Succeed workflow state	Doesn't redrive state machine executions that enter the Succeed state.
Fail workflow state	Reenters the Fail state and fails again.
Parallel workflow state	Reschedules and redrives only those branches that failed or aborted. If the state failed because of a `States.DataLimitExceeded` error, the Parallel state is rerun, including the branches that were successful in the original execution attempt.
Inline Map state	Reschedules and redrives only those iterations that failed or aborted. If the state failed because of a `States.DataLimitExceeded` error, the Inline Map state is rerun, including the iterations that were successful in the original execution attempt.

State name	Redrive execution behavior
Distributed Map state	redrives the unsuccessful child workflow executions in a Map Run. For more information, see Redriving Map Runs in Step Functions executions. If the state failed because of a `States.DataLimitExceeded` error, the Distributed Map state is rerun. This includes the child workflows that were successful in the original execution attempt.

IAM permission to redrive an execution

Step Functions needs appropriate permission to redrive an execution. The following IAM policy example grants the least privilege required to your state machine for redriving an execution. Remember to replace the *italicized* text with your resource-specific information.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "states:RedriveExecution"
            ],
            "Resource": "arn:aws:states:us-east-1:123456789012:execution:myStateMachine:*"
        }
    ]
}
```

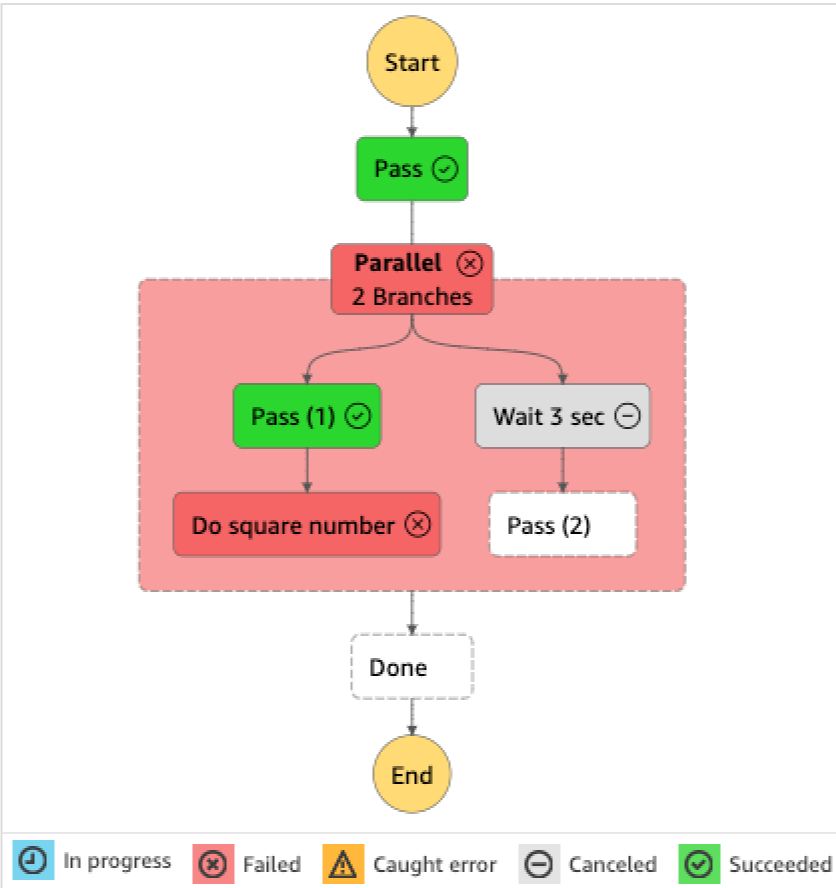
For an example of the permission you need to redrive a Map Run, see Example of IAM policy for redriving a Distributed Map.

Redriving executions in console

You can redrive eligible executions from the Step Functions console.

For example, imagine that you run a state machine and a parallel state fails to run.

The following image shows a **Lambda Invoke** step named **Do square number** inside a **Parallel** state has returned and failed. This caused the **Parallel** state to fail as well. The branches whose execution were in progress or not started are stopped and the state machine execution fails.



To redrive an execution from the console

1. Open the [Step Functions console](), and then choose an existing state machine that failed execution.
2. On the state machine detail page, under **Executions**, choose a failed execution instance.
3. Choose **Redrive**.
4. In the **Redrive** dialog box, choose **Redrive execution**.

> **Tip**
>
> If you're on the *Execution Details* page of a failed execution, do one of the following to redrive the execution:

- Choose **Recover**, and then select **Redrive from failure**.
- Choose **Actions**, and then select **Redrive**.

Notice that redrive uses the same state machine definition and ARN. It continues running the execution from the step that failed in the original execution attempt. In this example, it's the **Do square number** step and **Wait 3 sec** branch inside the **Parallel** state. After restarting the execution of these unsuccessful steps in the **Parallel** state, redrive will continue execution for the **Done** step.

5. Choose the execution to open the *Execution Details* page.

   On this page, you can view the results of the redriven execution. For example, in the Execution summary section, you can see **Redrive count**, which represents the number of times an execution has been redriven. In the **Events** section, you can see the redrive related execution events appended to the events of the original execution attempt. For example, the `ExecutionRedriven` event.

Redriving executions using API

You can redrive eligible executions using the RedriveExecution API. This API restarts unsuccessful executions of Standard Workflows from the step that failed, aborted, or timed out.

In the AWS Command Line Interface (AWS CLI), run the following command to redrive an unsuccessful state machine execution. Remember to replace the *italicized* text with your resource-specific information.

```
aws stepfunctions redrive-execution --execution-arn arn:aws:states:us-east-2:account-id:execution:myStateMachine:foo
```

Examining redriven executions

You can examine a redriven execution in the console or using the APIs: GetExecutionHistory and DescribeExecution.

Examine redriven executions on console

1. Open the [Step Functions console](), and then choose an existing state machine for which you've redriven an execution.
2. Open the *Execution Details* page.

   On this page, you can view the results of the redriven execution. For example, in the [Execution summary]() section, you can see **Redrive count**, which represents the number of times an execution has been redriven. In the **Events** section, you can see the redrive related execution events appended to the events of the original execution attempt. For example, the `ExecutionRedriven` event.

Examine redriven executions using APIs

If you've redriven a state machine execution, you can use one of the following APIs to view details about the redriven execution. Remember to replace the *`italicized`* text with your resource-specific information.

- GetExecutionHistory – Returns the history of the specified execution as a list of events. This API also returns the details about the redrive attempt of an execution, if available.

  In the AWS CLI, run the following command.

  ```
  aws stepfunctions get-execution-history --execution-arn arn:aws:states:us-east-2:account-id:execution:myStateMachine:foo
  ```

- DescribeExecution – Provides information about a state machine execution. This can be the state machine associated with the execution, the execution input and output, execution redrive details, if available, and relevant execution metadata.

  In the AWS CLI, run the following command.

  ```
  aws stepfunctions describe-execution --execution-arn arn:aws:states:us-east-2:account-id:execution:myStateMachine:foo
  ```

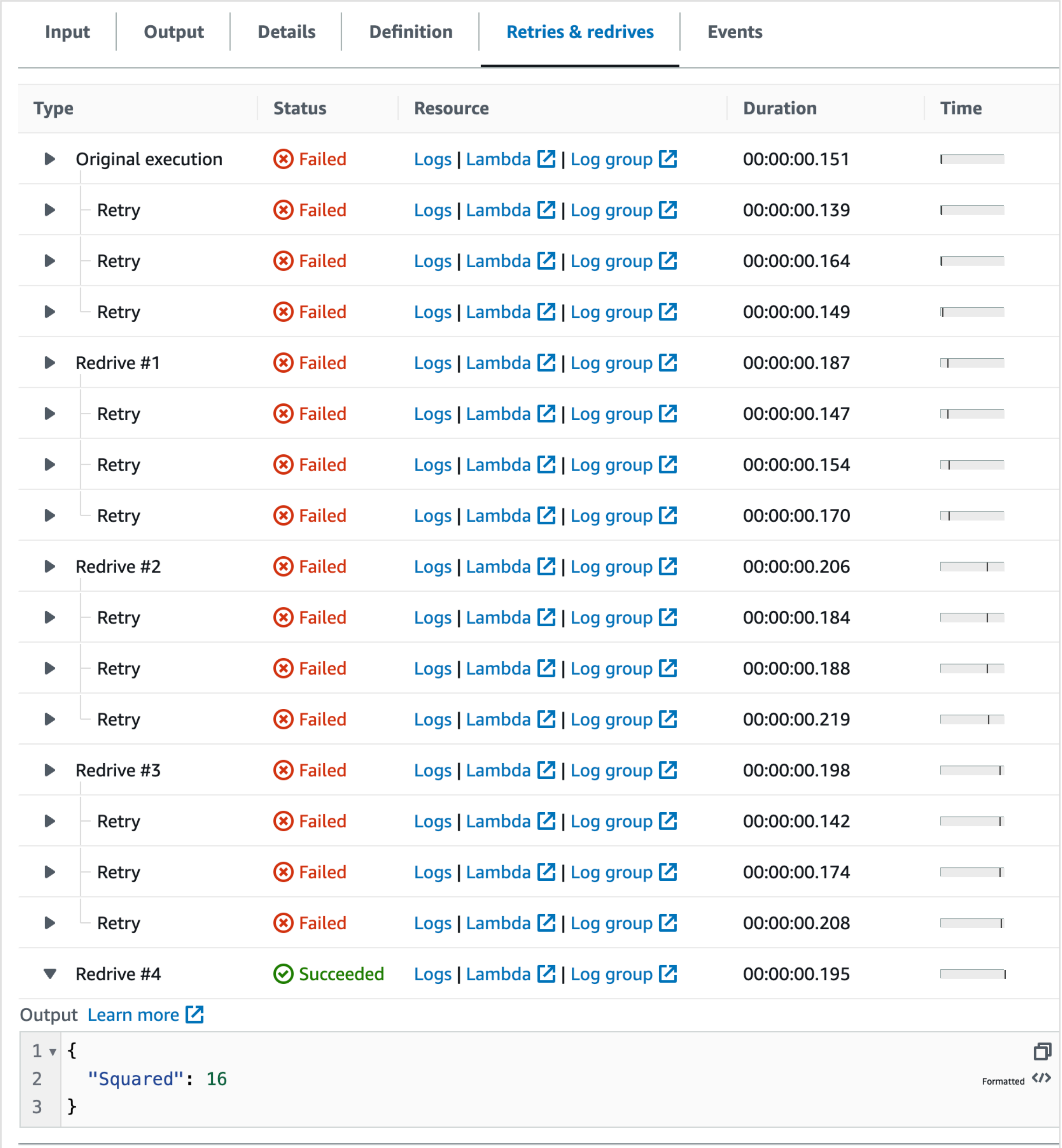
Retry behavior of redriven executions

If your redriven execution reruns a Task workflow state, Parallel workflow state, or Inline Map state, for which you have defined retries, the retry attempt count for these states is reset to 0 to allow for the maximum number of attempts on redrive. For a redriven execution, you can track individual retry attempts of these states using the console.

To examine the individual retry attempts in the console

1. On the *Execution Details* page of the Step Functions console, choose a state that was retried on redrive.
2. Choose the **Retries & redrives** tab.
3. Choose the arrow icon next to each retry attempt to view its details. If the retry attempt succeeded, you can view the results in **Output** that appears in a dropdown box.

The following image shows an example of the retries performed for a state in the original execution attempt and the redrives of that execution. In this image, three retries are performed in the original and redrive execution attempts. The execution succeeds in the fourth redrive attempt and returns an output of 16.



Type	Status	Resource	Duration	Time
Original execution	Failed	Logs | Lambda | Log group	00:00:00.151	
Retry	Failed	Logs | Lambda | Log group	00:00:00.139	
Retry	Failed	Logs | Lambda | Log group	00:00:00.164	
Retry	Failed	Logs | Lambda | Log group	00:00:00.149	
Redrive #1	Failed	Logs | Lambda | Log group	00:00:00.187	
Retry	Failed	Logs | Lambda | Log group	00:00:00.147	
Retry	Failed	Logs | Lambda | Log group	00:00:00.154	
Retry	Failed	Logs | Lambda | Log group	00:00:00.170	
Redrive #2	Failed	Logs | Lambda | Log group	00:00:00.206	
Retry	Failed	Logs | Lambda | Log group	00:00:00.184	
Retry	Failed	Logs | Lambda | Log group	00:00:00.188	
Retry	Failed	Logs | Lambda | Log group	00:00:00.219	
Redrive #3	Failed	Logs | Lambda | Log group	00:00:00.198	
Retry	Failed	Logs | Lambda | Log group	00:00:00.142	
Retry	Failed	Logs | Lambda | Log group	00:00:00.174	
Retry	Failed	Logs | Lambda | Log group	00:00:00.208	
Redrive #4	Succeeded	Logs | Lambda | Log group	00:00:00.195	

Output Learn more

```
{
  "Squared": 16
}
```

Formatted

Viewing a Distributed Map Run execution in Step Functions

The Step Functions console provides a *Map Run Details* page which displays all the information related to a *Distributed Map state* execution. For example, you can view the status of the *Distributed Map state*'s execution, the Map Run's ARN, and the statuses of the items processed in the child workflow executions started by the *Distributed Map state*. You can also view a list of all child workflow executions and access their details. If your Map Run was redriven, you will see redrive details in the Map Run execution summary too.

When you run a `Map` state in Distributed mode, Step Functions creates a Map Run resource. A Map Run refers to a set of child workflow executions that a *Distributed Map state* starts, and the runtime settings that control these executions. Step Functions assigns an Amazon Resource Name (ARN) to your Map Run. You can examine a Map Run in the Step Functions console. You can also invoke the `DescribeMapRun` API action.

Map Runs do **not** emit metrics to CloudWatch. However, child workflow executions of a Map Run **do** emit metrics to CloudWatch. These metrics will have a labelled State Machine ARN with the following format:

```
arn:partition:states:region:account:stateMachine:stateMachineName/MapRunLabel or UUID
```

The *Map Run Details* has three sections: *Map Run execution summary*, *Item processing status*, and *Listing executions*.

Map Run execution summary

The *Map Run Execution summary* provides an overview of the execution details of the *Distributed Map state*.

Details

Shows execution status of the *Distributed Map state*, the Map Run ARN, and type of the child workflow executions started by the *Distributed Map state*. You can view additional configurations, such as tolerated failure threshold for the Map Run and the maximum concurrency specified for child workflow executions.

Input and output

Shows the input received by the *Distributed Map state* and the corresponding output that it generates.

You can view the input dataset and its location, and the input filters applied to the individual data items in that dataset. If you export the output of the *Distributed Map state* execution, this tab shows the path to the Amazon S3 bucket that contains the execution results. Otherwise, it points you to the parent workflow's *Execution Details* page to view the execution output.

Error message

If your Map Run failed, the *Map Run Details* page displays an error message with the reason for failure.

From the **Recover** dropdown button on this error message, you can either redrive the unsuccessful child workflow executions started by this Map Run or start a new execution of the parent workflow.

See Redriving Map Runs to learn how to restart your workflow.

Item processing status

The **Item processing status** section displays the status of the items processed in a Map Run. For example, **Pending** indicates that a child workflow execution hasn't started processing the item yet.

Item statuses are dependent on the status of the child workflow executions processing the items. If a child workflow execution failed, times out, or if a user cancels the execution, Step Functions doesn't receive any information about the processing result of the items inside that child workflow execution. All items processed by that execution share the child workflow execution's status.

For example, say that you want to process 100 items in two child workflow executions, where each execution processes a batch of 50 items. If one of the executions fails and the other succeeds, you'll have 50 successful and 50 failed items.

The following table explains the types of processing statuses available for all items:

Status	Description
Pending	Indicates an item that the child workflow execution hasn't started processing. If a Map Run stops, fails, or a user cancels the execution before processing of an item starts, the item remains in **Pending** status.

Status	Description
	For example, if a Map Run fails with 10 unprocessed items, these 10 items remain in the **Pending** status.
Running	Indicates an item currently being processed by the child workflow execution.
Succeeded	Indicates that the child workflow execution successfully processed the item. A successful child workflow execution can't have any failed items. If one item in the dataset fails during execution, the entire child workflow execution fails.
Failed	Indicates that the child workflow execution either failed to process the item, or the execution timed out. If any one item processed by a child workflow execution fails, the entire child workflow execution fails. For example, consider a child workflow execution that processed 1000 items. If any one item in that dataset fails during execution , then Step Functions considers the entire child workflow execution as failed. When you redrive a Map Run, the count of items with this status is reset to 0.

Status	Description
Aborted	Indicates that the child workflow execution started processing the item, but either the user cancelled the execution, or Step Functions stopped the execution because the Map Run failed. For example, consider a **Running** child workflow execution that's processing 50 items. If the Map Run stops because of a failure or because a user cancelled the execution, the child workflow execution and the status of all 50 items changes to **Aborted**. If you use a child workflow execution of the **Express** type, you can't stop the execution. When you redrive a Map Run that starts child workflow executions of type Express, the count of items with this status is reset to 0. This is because Express child workflows are restarted using the StartExecution API action instead of being redriven.

Listing executions

The **Executions** section lists all of the child workflow executions for a specific Map Run. Use the **Search by exact execution name** field to search for a specific child workflow execution. To see details about a specific execution, select a child workflow execution from the list and choose the **View details** button to open its *Execution details* page.

You can also use the API or AWS CLI to list child workflow executions started by the Map Run:

- Using the API, call ListExecutions with the `mapRunArn` parameter set to the ARN of the parent workflow.

- Using the AWS CLI, call list-executions with the `map-run-arn` parameter set to the ARN of the parent workflow.

> **Important**
>
> The retention policy for child workflow executions is 90 days.
> Completed child workflow executions that are older will not be displayed in the **Executions** table, even if the *Distributed Map state* or parent workflow continues to run longer than the retention period. You can view execution details, including results, of these child workflow executions if you export the *Distributed Map state* output to an Amazon S3 bucket using `ResultWriter (Map)`.

> **Tip**
>
> Choose the refresh button to view the most current list of all child workflow executions.

Redriving Map Runs in Step Functions executions

You can restart unsuccessful child workflow executions in a Map Run by redriving your parent workflow. A redriven parent workflow redrives all the unsuccessful states, including Distributed Map. A parent workflow redrives unsuccessful states if there's no `<stateType>Exited` event corresponding to the `<stateType>Entered` event for a state when the parent workflow completed its execution. For example, if the event history doesn't contain the `MapStateExited` event for a `MapStateEntered` event, you can redrive the parent workflow to redrive all the unsuccessful child workflow executions in the Map Run.

A Map Run is either not started or fails in the original execution attempt when the state machine doesn't have the required permission to access the ItemReader (Map), ResultWriter (Map), or both. If the Map Run wasn't started in the original execution attempt of the parent workflow, redriving the parent workflow starts the Map Run for the first time. To resolve this, add the required permissions to your state machine role, and then redrive the parent workflow. If you redrive the parent workflow without adding the required permissions, it attempts to start a new Map Run run that will fail again. For information about the permissions that you might need, see IAM policies for using Distributed Map states.

Topics

Redrive eligibility for child workflows in a Map Run

You can redrive the unsuccessful child workflow executions in a Map Run if the following conditions are met:

- You started the parent workflow execution on or after November 15, 2023. Executions that you started prior to this date aren't eligible for redrive.
- You haven't exceeded the hard limit of 1000 redrives of a given Map Run. If you've exceeded this limit, you'll receive the `States.Runtime` error.
- The parent workflow is redrivable. If the parent workflow isn't redrivable, you can't redrive the child workflow executions in a Map Run. For more information about redrive eligibility of a workflow, see Redrive eligibility for unsuccessful executions.
- The child workflow executions of type Standard in your Map Run haven't exceeded the 25,000 execution event history limit. Child workflow executions that have exceeded the event history limit are counted towards the tolerated failure threshold and considered as failed. For more information about the redrive eligibility of an execution, see Redrive eligibility for unsuccessful executions.

A new Map Run is started and the existing Map Run isn't redriven in the following cases even if the Map Run failed in the original execution attempt:

- Map Run failed because of the `States.DataLimitExceeded` error.
- Map Run failed because of the JSON data interpolation error, `States.Runtime`. For example, you selected a non-existent JSON node in Filtering state output using OutputPath.

A Map Run can continue to run even after the parent workflow stops or times out. In these scenarios, the redrive doesn't happen immediately:

- Map Run might still be canceling in progress child workflow executions of type Standard, or waiting for child workflow executions of type Express to complete their executions.
- Map Run might still be writing results to the ResultWriter (Map), if you configured it to export results.

In these cases, the running Map Run completes its operations before attempting to redrive.

Child workflow execution redrive behavior

The redriven child workflow executions in a Map Run exhibit the behavior as described in the following table.

Express child workflow	Standard child workflow
All child workflow executions that failed or timed out in the original execution attempt are started using the StartExecution API action. The first state in ItemProcessor is run first.	All child workflow executions that failed, timed out, or canceled in the original execution attempt are redriven using the RedriveExecution API action. These child workflows are redriven from the last state in ItemProcessor that resulted in their unsuccess ful execution.
Unsuccessful executions can always be redriven. This is because Express child workflow executions are always started as a new execution using the StartExecution API action.	Unsuccessful Standard child workflow executions can't always be redriven. If an execution isn't redrivable, it won't be attempted again. The last error or output of the execution is permanent. This is possible when an execution exceeds 25,000 history events, or its redrivable period of 14 days has expired. A Standard child workflow execution might not be redrivable if the parent workflow execution has closed within 14 days, but the

Express child workflow	Standard child workflow
	child workflow execution closed earlier than 14 days.
Express child workflow executions use the same execution ARN as the original execution attempt, but you can't distinctly identify their individual redrives.	Standard child workflow executions use the same execution ARN as the original execution attempt. You can distinctly identify the individual redrives in the console and using APIs, such as GetExecutionHistory and DescribeExecution. For more information, see the section called "Examining redriven executions".

If you've redriven a Map Run, and it has reached its concurrency limit, the child workflow executions in that Map Run transition to the pending state. The execution status of the Map Run also transitions to the **Pending redrive** state. Until the specified concurrency limit can allow for more child workflow executions to run, the execution remains in the **Pending redrive** state.

For example, say that the concurrency limit of the Distributed Map in your workflow is 3000, and the number of child workflows to be rerun is 6000. This causes 3000 child workflows to run in parallel while the remaining 3000 workflows remain in the **Pending redrive** state. After the first batch of 3000 child workflows complete their execution, the remaining 3000 child workflows are run.

When a Map Run has completed its execution or is aborted, the count of child workflow executions in the **Pending redrive** state is reset to 0.

Scenarios of input used on Map Run redrive

Depending on how you provided input to the Distributed Map in the original execution attempt, a redriven Map Run will use the input as described in the following table.

Input in the original execution attempt	Input used on Map Run redrive
Input passed from a previous state or the execution input.	The redriven Map Run uses the same input.

Input in the original execution attempt	Input used on Map Run redrive
Input passed using ItemReader (Map) and the Map Run didn't start the child workflow executions because one of the following conditions is true: • Map Run failed with the `States.ItemReaderFailed` error. • Map Run failed with the `States.ResultWriterFailed` error. • The parent workflow execution was timed out or canceled before the Map Run was started.	The redriven Map Run uses the input in the Amazon S3 bucket.
Input passed using ItemReader. The Map Run failed after starting or attempting to start child workflow executions.	The redriven Map Run uses the same input provided in the original execution attempt.

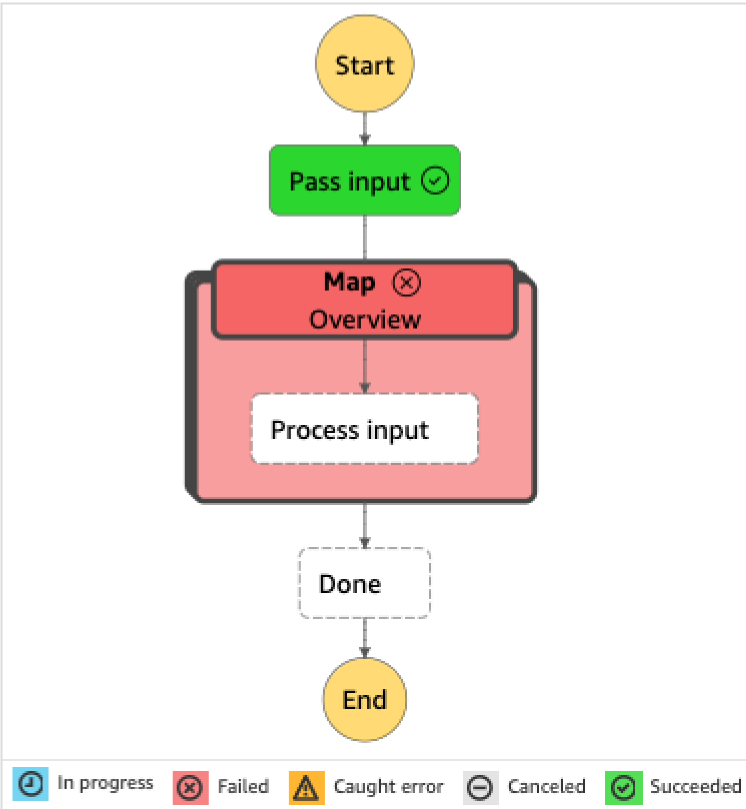
IAM permission to redrive a Map Run

Step Functions needs appropriate permission to redrive a Map Run. The following IAM policy example grants the least privilege required to your state machine for redriving a Map Run. Remember to replace the *`italicized`* text with your resource-specific information.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:RedriveExecution"
      ],
      "Resource": "arn:aws:states:us-
east-2:123456789012:execution:myStateMachineName/myMapRunLabel:*"
    }
  ]
}
```

Redriving Map Run in console

The following image shows the execution graph of a state machine that contains a Distributed Map. This execution failed because the Map Run failed. To redrive the Map Run, you must redrive the parent workflow.



To redrive a Map Run from the console

1. Open the [Step Functions console](https://console.aws.amazon.com/states/home?region=us-east-1#/), and then choose an existing state machine that contains a Distributed Map that failed execution.
2. On the state machine detail page, under **Executions**, choose a failed execution instance of this state machine.
3. Choose **Redrive**.
4. In the **Redrive** dialog box, choose **Redrive execution**.

> **Tip**
>
> You can also redrive a Map Run from the *Execution Details* or *Map Run Details* page. If you're on the *Execution Details* page, do one of the following to redrive the execution:

- Choose **Recover**, and then select **Redrive from failure**.
- Choose **Actions**, and then select **Redrive**.

If you're on the *Map Run Details* page, choose **Recover**, and then select **Redrive from failure**.

Notice that redrive uses the same state machine definition and ARN. It continues running the execution from the step that failed in the original execution attempt. In this example, it's the Distributed Map step named **Map** and the **Process input** step inside it. After restarting the unsuccessful child workflow executions of the Map Run, redrive will continue execution for the **Done** step.

5. From the *Execution Details* page, choose **Map Run** to see the details of the redriven Map Run.

   On this page, you can view the results of the redriven execution. For example, in the Map Run execution summary section, you can see **Redrive count**, which represents the number of times the Map Run has been redriven. In the **Events** section, you can see the redrive related execution events appended to the events of the original execution attempt. For example, the `MapRunRedriven` event.

After you've redriven a Map Run, you can examine its redrive details in the console or using the GetExecutionHistory and DescribeExecution API actions. For more information about examining a redriven execution, see Examining redriven executions.

Redriving Map Run using API

You can redrive an eligible Map Run using the RedriveExecution API on the parent workflow. This API restarts unsuccessful child workflow executions in a Map Run.

In the AWS Command Line Interface (AWS CLI), run the following command to redrive an unsuccessful state machine execution. Remember to replace the *italicized* text with your resource-specific information.

```
aws stepfunctions redrive-execution --execution-arn arn:aws:states:us-east-2:account-id:execution:myStateMachine:foo
```

After you have redriven a Map Run, you can examine its redrive details in the console or using the DescribeMapRun API action. To examine the redrive details of Standard workflow executions in a Map Run, you can use the GetExecutionHistory or DescribeExecution API action. For more information about examining a redriven execution, see the section called “Examining redriven executions”.

You can examine the redrive details of Express workflow executions in a Map Run on the Step Functions console if you've enabled logging on the parent workflow. For more information, see Using CloudWatch Logs to log execution history in Step Functions.

Processing input and output in Step Functions

> **Managing state with variables and JSONata**
>
> Step Functions recently added variables and JSONata to manage state and transform data. Learn more in the blog post Simplifying developer experience with variables and JSONata in AWS Step Functions

When a Step Functions execution receives JSON input, it passes that data to the first state in the workflow as input.

With JSONata, you can retrieve state input from `$states.input`. Your state machine executions also provide that initial input data in the Context object. You can retrieve the original state machine input at any point in your workflow from `$states.context.Execution.Input`.

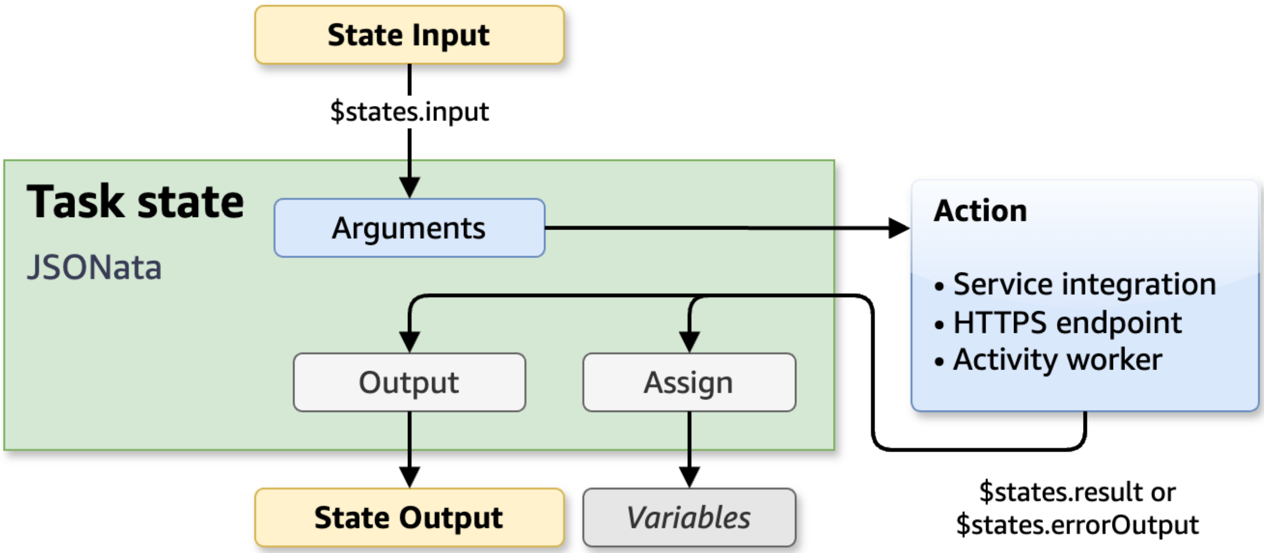
When states exit, their output is available to the *very* next state in your state machine. Your state inputs will pass through as state output by default, unless you **modify** the state output. For data that you might need in later steps, consider storing it in variables. For more info, see the section called "Passing data with variables".

> **QueryLanguage recommendation**
>
> For new state machines, we recommend the JSONata query language. In state machines that do not specify a query language, the state machine defaults to JSONPath for backward compatibility. You must opt-in to use JSONata for your state machines or individual states.

Processing input and output with JSONata

With JSONata expressions, you can select and transform data. In the `Arguments` field, you can customize the data sent to the action. The result can be transformed into custom state output in the `Output` field. You can also store data in variables in the `Assign` field. For more info, see Transforming data with JSONata.

The following diagram shows how JSON information moves through a JSONata task state.



Processing input and output with JSONPath

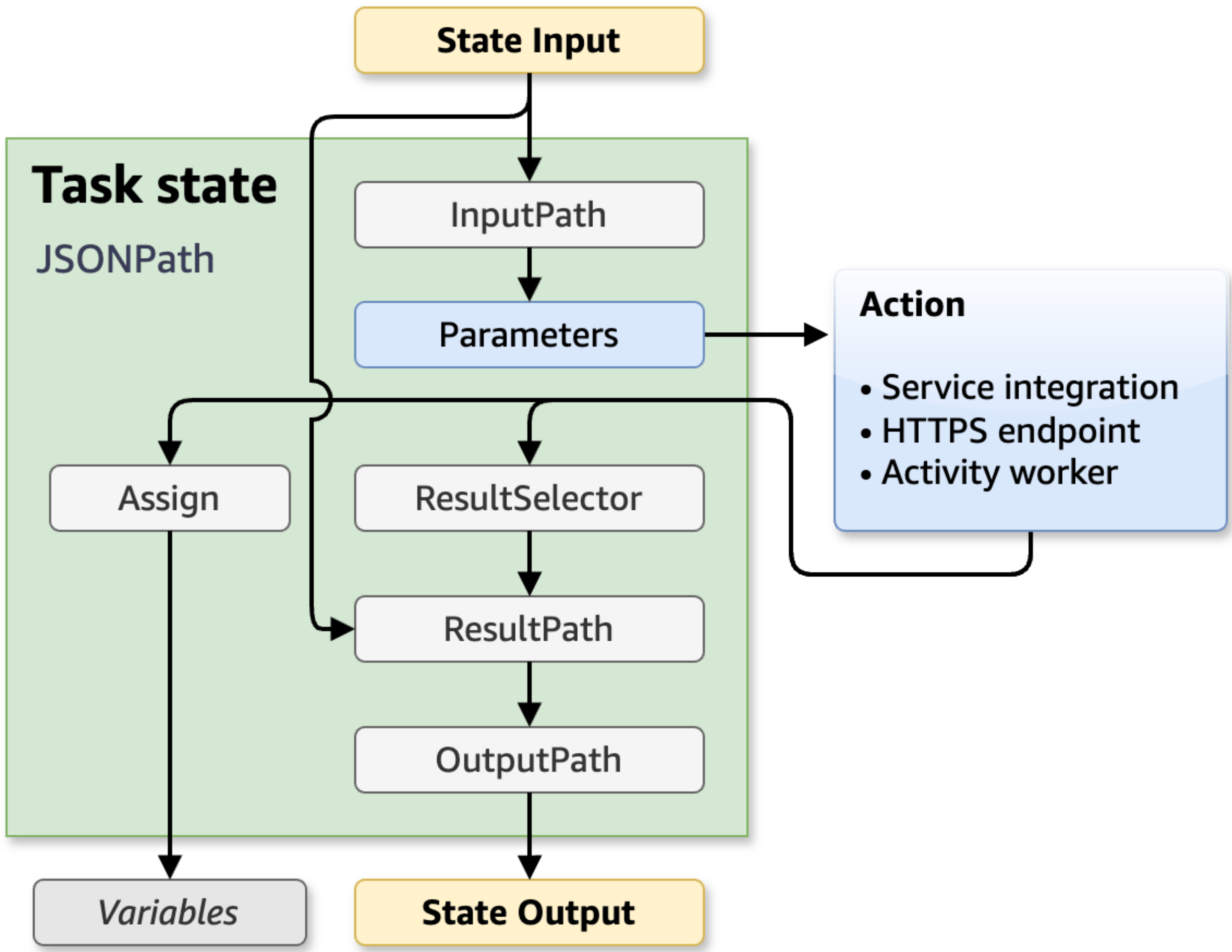
> **Managing state and transforming data**
>
> Learn about Passing data between states with variables and Transforming data with JSONata.

For state machines that use JSONPath, the following fields control the flow of data from state to state: `InputPath`, `Parameters`, `ResultSelector`, `ResultPath`, and `OutputPath`. Each JSONPath field can manipulate JSON as it moves through each state in your workflow.

JSONPath fields can use paths 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.

The following diagram shows how JSON information moves through a JSONPath task state. The `InputPath` selects the parts of the JSON input to pass to the task of the `Task` state (for example, an AWS Lambda function). You can adjust the data that is sent to your action in the `Parameters` field. Then, with `ResultSelector`, you can select portions of the action result to carry forward. `ResultPath` then selects the combination of state input and task results to pass to the output.

`OutputPath` can filter the JSON output to further limit the information that's passed to the output.



Topics

- Passing data between states with variables
- Transforming data with JSONata in Step Functions
- Accessing execution data from the Context object in Step Functions
- Using JSONPath paths
- Manipulate parameters in Step Functions workflows
- Example: Manipulating state data with paths in Step Functions workflows
- Specifying state output using ResultPath in Step Functions

- Map state input and output fields in Step Functions

Passing data between states with variables

Managing state with variables and JSONata

Step Functions recently added variables and JSONata to manage state and transform data. Learn more in the blog post Simplifying developer experience with variables and JSONata in AWS Step Functions

The following video link describes variables and JSONata in Step Functions with a DynamoDB example: Enhanced Data Flow in AWS Step Functions

With variables and state output, you can pass data between the steps of your workflow.

Using workflow variables, you can store data in a step and retrieve that data in future steps. For example, you could store an API response that contains data you might need later. Conversely, state output can only be used as input to the very next step.

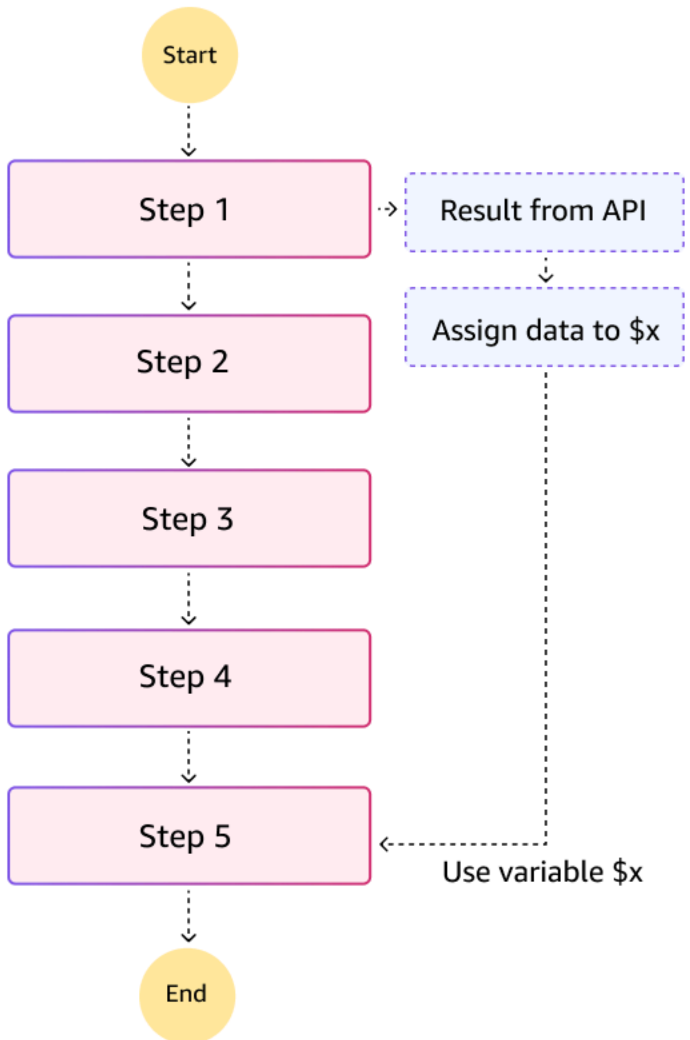
Conceptual overview of variables

With workflow variables, you can store data to reference later. For example, Step 1 might store the result from an API request so a part of that request can be re-used later in Step 5.

In the following scenario, the state machine fetches data from an API once. In Step 1, the workflow stores the returned API data (up to 256 KiB per state) in a variable ‘x’ to use in later steps.

Without variables, you would need to pass the data through output from Step 1 to Step 2 to Step 3 to Step 4 to use it in Step 5. What if those intermediate steps do not need the data? Passing data from state to state through outputs and input would be unnecessary effort.

With variables, you can store data and use it in any future step. You can also modify, rearrange, or add steps without disrupting the flow of your data. Given the flexibility of variables, you might only need to use **Output** to return data from Parallel and Map sub-workflows, and at the end of your state machine execution.



States that support variables

The following state types support `Assign` to declare and assign values to variables: *Pass, Task, Map, Parallel, Choice, Wait.*

To set a variable, provide a JSON object with variable names and values:

```
"Assign": {
  "productName": "product1",
  "count" : 42,
  "available" : true
}
```

To reference a variable, prepend the name with a dollar sign ($), for example, `$productName`.

Reserved variable : $states

Step Functions defines a single reserved variable called **$states**. In JSONata states, the following structures are assigned to `$states` for use in JSONata expressions:

```
# Reserved $states variable in JSONata states
$states = {
  "input":       // Original input to the state
  "result":      // API or sub-workflow's result (if successful)
  "errorOutput": // Error Output (only available in a Catch)
  "context":     // Context object
}
```

On state entry, Step Functions assigns the state input to **$states.input**. The value of `$states.input` can be used in all fields that accept JSONata expressions. `$states.input` always refers to the original state input.

For `Task`, `Parallel`, and `Map` states:

- **$states.result** refers to the API or sub-workflow's raw result if successful.
- **$states.errorOutput** refers to the Error Output if the API or sub-workflow failed.

  `$states.errorOutput` can be used in the `Catch` field's `Assign` or `Output`.

Attempting to access `$states.result` or `$states.errorOutput` in fields and states where they are not accessible will be caught at creation, update, or validation of the state machine.

The `$states.context` object provides your workflows information about their specific execution, such as `StartTime`, task token, and initial workflow input. To learn more, see Accessing execution data from the Context object in Step Functions .

Variable name syntax

Variable names follow the rules for Unicode Identifiers as described in Unicode® Standard Annex #31. The first character of a variable name must be a Unicode ID_Start character, and the second and subsequent characters must be Unicode ID_Continue characters. The maximum length of a variable name is 80.

The variable name convention is similar to rules for JavaScript and other programming languages.

Variable scope

Step Functions workflows avoid race conditions with variables by using a *workflow-local scope*.

Workflow-local scope includes all states inside a state machine's **States** field, but not states inside Parallel or Map states. States inside Parallel or Map states can refer to outer scope variables, but they create and maintain their own separate workflow-local variables and values.

`Parallel` branches and `Map` iterations can access variable values from **outer scopes**, but they do not have access to variable values from other concurrent branches or iterations. When handling errors, the `Assign` field in a `Catch` can assign values to variables in the outer scope, that is, the scope in which the Parallel/Map state exists.

Exception: **Distributed Map states** cannot currently reference variables in outer scopes.

A variable exists in a scope if any state in the scope assigns a value to it. To help avoid common errors, a variable assigned in an inner scope cannot have the same name as one assigned in an outer scope. For example, if the top-level scope assigns a value to a variable called `myVariable`, then no other scope (inside a `Map`, `Parallel`) can assign to `myVariable` as well.

Access to variables depends on the current scope. Parallel and Map states have their own scope, but can access variables in outer scopes.

When a Parallel or Map state completes, all of their variables will go out of scope and stop being accessible. Use the **Output field** to pass data out of Parallel branches and Map iterations.

Assign field in ASL

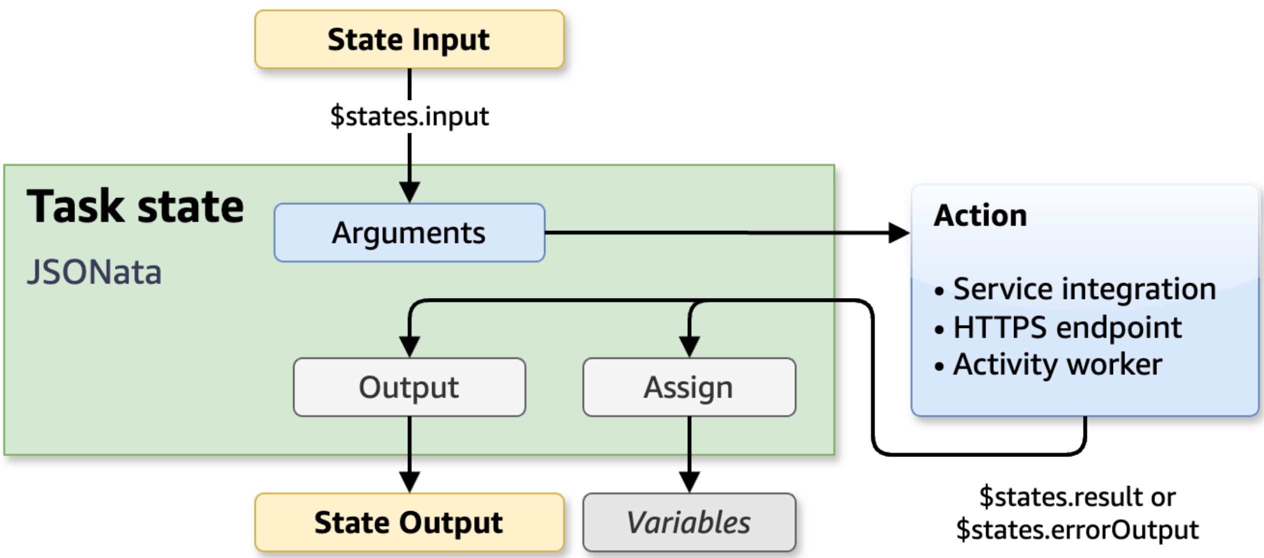
The `Assign` field in ASL is used to assign values to one or more variables. The `Assign` field is available at the top level of each state (except `Succeed` and `Fail`), inside `Choice` state rules, and inside `Catch` fields. For example:

```
# Example of Assign with JSONata
"Store inputs": {
    "Type": "Pass",
    "Next": "Get Current Price",
    "Comment": "Store the input desired price into a variable: $desiredPrice",
    "Assign": {
       "desiredPrice": "{% $states.input.desired_price %}",
       "maximumWait": "{% $states.input.max_days %}"
```

```
    }
},
```

The `Assign` field takes a JSON object. Each top-level field names a variable to assign. In the previous examples, the variable names are `desiredPrice` and `maximumWait`. When using JSONata, `{% ... %}` indicates a JSONata expression which might contain variables or more complex expressions. For more information about JSONata expressions, refer to the [JSONata.org documentation](https://jsonata.org).

When using **JSONata** as the query language, the following diagram shows how **Assign** and **Output** fields are processed in parallel. Note the implication: *assigning variable values will not affect state Output.*



The following JSONata example retrieves `order.product` from the state input. The variable `currentPrice` is set to a value from the result of the task.

```
# Example of Task with JSONata assignment from result
{
   "Type": "Task",
   ...
   "Assign": {
      "product": "{% $states.input.order.product %}",
```

```
        "currentPrice": "{% $states.result.Payload.current_price %}"
    },
    "Next": "the next state"
}
```

Note: You **cannot** assign a value to a part of a variable. For example, you can `"Assign":{"x":42}`, but you cannot `"Assign":{"x.y":42}` or `"Assign":{"x[2]":42}`.

Evaluation order in an assign field

All variable references in Step Functions states use the values as they were on **state entry**.

The previous fact is important to understand how the `Assign` field assigns values to one or more variables. First, new values are calculated, then Step Functions assigns the new values to the variables. The new variable values will be available starting with the **next** state. For example, consider the following `Assign` field:

```
# Starting values: $x=3, $a=6

"Assign": {
  "x": "{% $a %}",
  "nextX": "{% $x %}"
}

# Ending values: $x=6, $nextX=3
```

In the preceding example, the variable `x` is both assigned and referenced.

Remember, all expressions are ***evaluated first***, then assignments are made. And newly assigned values will be available in the **next** state.

Let's go through the example in detail. Assume that in a previous state, `$x` was assigned a value of three (3) and `$a` was assigned a value of six (6). The following steps describe the process:

1. All expressions are evaluated, using **current** values of all variables.

   The expression `"{% $a %}"` will evaluate to 6, and `"{% $x %}"` will evaluate to 3.
2. Next, assignments are made:

   `$x` will be assigned the value six (6)

`$nextX` will be assigned three (3)

Note: If `$x` had not been previously assigned, the example would **fail** because `$x` would be *undefined*.

In summary, Step Functions evaluates **all** expressions and then makes assignments. The order in which the variables occur in the `Assign` field does **not** matter.

Limits

The maximum size of a single variable is 256Kib, for both Standard and Express workflows.

The maximum combined size for all variables in a single `Assign` field is also 256Kib. For example, you could assign X and Y to 128KiB, but you could not assign both X and Y to 256KiB in the same `Assign` field.

The total size of all stored variables cannot exceed 10MiB per execution.

Using variables in JSONPath states

Variables are also available in states that use JSONPath for their query language.

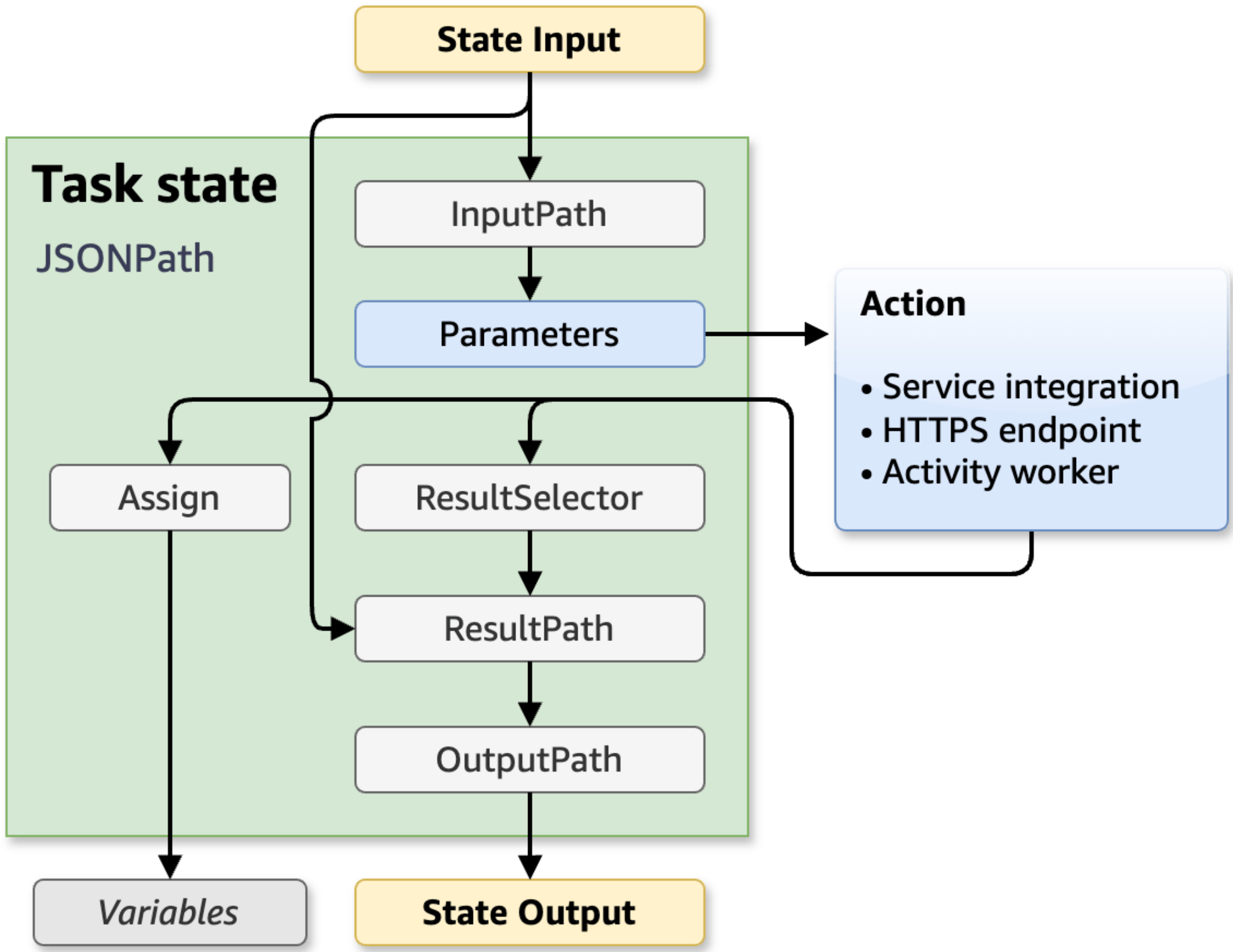
You can reference a variable in any field that accepts a JSONpath expression (`$.` or `$$.` syntax), with the exception of `ResultPath`, which specifies a location in state input to inject the state's result. Variables cannot be used in `ResultPath`.

In JSONPath, the $ symbol refers to the 'current' value and $$ represents the states Context object. JSONPath expressions can start with `$.` as in `$.customer.name`. You can access context with `$$.` as in `$$.Execution.Id`.

To reference a variable, you also use the $ symbol before a variable name, for example, `$x` or `$order.numItems`.

In **JSONPath** fields that accept intrinsic functions, variables can be used in the arguments, for example `States.Format('The order number is {}', $order.number)`.

The following digram illustrates how the assign step in a **JSONPath** task occurs in at the same time as the ResultSelector:



Assigning variables in JSONPath

JSONPath variable assignments behave similarly to payload templates. Fields that end with `.$` indicate the value is a JSONPath expression which Step Functions evaluates to a value during state machine execution (for example: `$.order..product` and `$.order.total`).

```
# Example of Assign with JSONPath
{
  "Type": "Task",
  ...
  "Assign": {
    "products.$": "$.order..product",
    "orderTotal.$": "$.order.total"
```

```
  },
  "Next": "the next state"
}
```

For JSONPath states, the value of `$` in an `Assign` field depends on the state type. In `Task`, `Map`, `Parallel` states, the `$` refers to the API/sub-workflow result. In `Choice` and `Wait` state, `$` refers to the *effective input*, which is the value after `InputPath` has been applied to the state input. For `Pass`, `$` refers to the result, whether generated by the `Result` field or the `InputPath`/`Parameters` fields.

The following JSONPath example assigns a JSON object to the `details` variable, the result of the JSONPath expression `$.result.code` to `resultCode`, and the result of the JSONPath expression `States.Format('Hello {}', $customer.name)` to `message`. If this was in a `Task` state, then `$` in `$.order.items` and `$.result.code` refers to the API result. The `startTime` variable is assigned with a value from the Context object, `$$.Execution.StartTime`.

```
"Assign": {
   "details": {
      "status": "SUCCESS",
      "lineItems.$": "$.order.items"
   },
   "resultCode.$": "$.result.code",
   "message.$": "States.Format('Hello {}', $customer.name)",
   "startTime.$": "$$.Execution.StartTime"
}
```

Transforming data with JSONata in Step Functions

With JSONata, you gain a powerful open source query and expression language to **select** and **transform** data in your workflows. For a brief introduction and complete JSONata reference, see JSONata.org documentation.

The following video link describes variables and JSONata in Step Functions with a DynamoDB example: Enhanced Data Flow in AWS Step Functions

You must opt-in to use the JSONata query and transformation language for existing workflows. When creating a workflow in the console, we recommend choosing JSONata for the top-level state machine `QueryLanguage`. For existing or new workflows that use JSONPath, the console provides an option to convert individual states to JSONata.

After selecting JSONata, your workflow fields will be reduced from five JSONPath fields (`InputPath`, `Parameters`, `ResultSelector`, `ResultPath`, and `OutputPath`) to only two fields: `Arguments` and `Output`. Also, you will **not** use `.$` on JSON object key names.

If you are new to Step Functions, you only need to know that JSONata expressions use the following syntax:

JSONata syntax: `"{% <JSONata expression> %}"`

The following code samples show a conversion from JSONPath to JSONata:

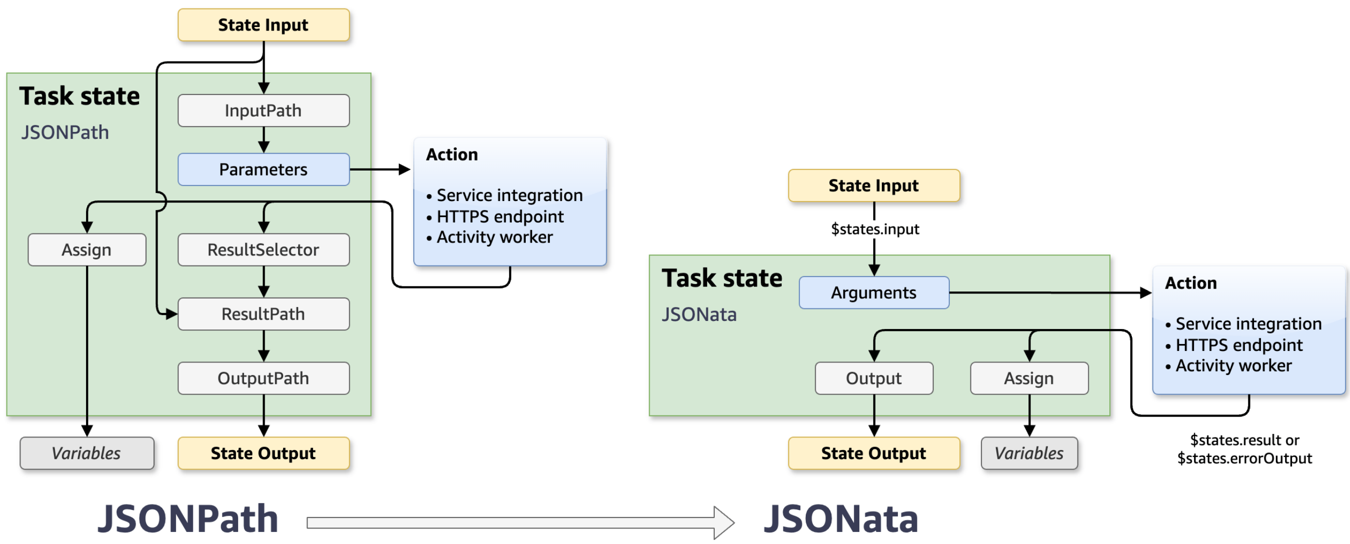
```
# Original sample using JSONPath
{
  "QueryLanguage": "JSONPath", // Set explicitly; could be set and inherited from top-level
  "Type": "Task",
  ...
  "Parameters": {
    "static": "Hello",
    "title.$": "$.title",
    "name.$": "$customerName",  // With $customerName declared as a variable
    "not-evaluated": "$customerName"
  }
}
```

```
# Sample after conversion to JSONata
{
  "QueryLanguage": "JSONata", // Set explicitly; could be set and inherited from top-level
  "Type": "Task",
  ...
  "Arguments": { // JSONata states do not have Parameters
    "static": "Hello",
    "title": "{% $states.input.title %}",
    "name": "{% $customerName %}",   // With $customerName declared as a variable
    "not-evaluated": "$customerName"
  }
}
```

Given input `{ "title" : "Doctor" }` and variable `customerName` assigned to `"María"`, both state machines will produce the following JSON result:

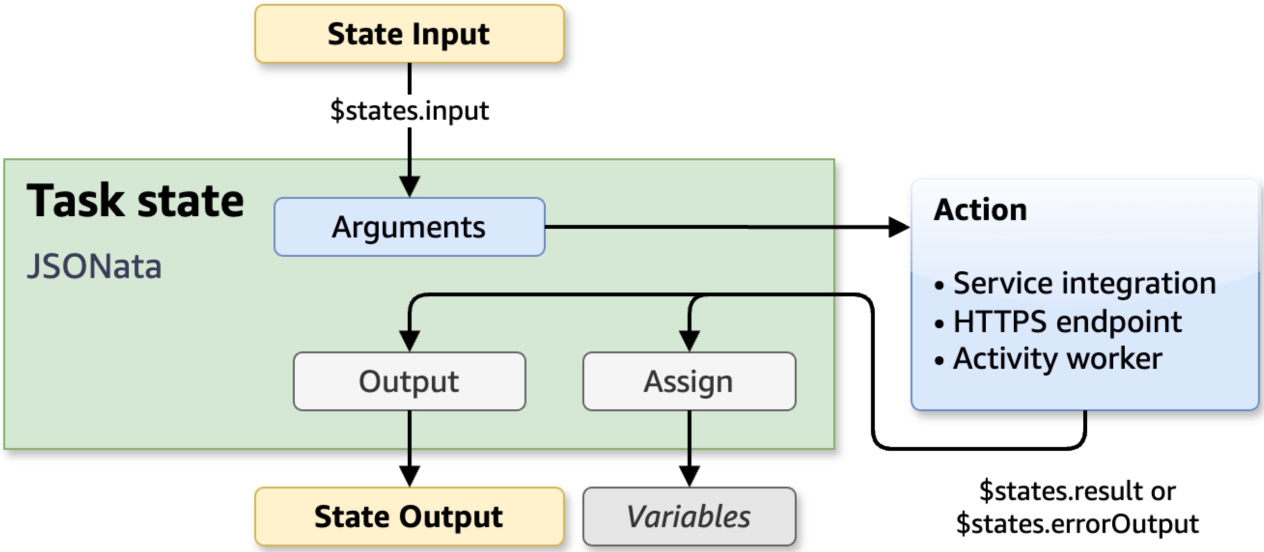
```
{
  "static": "Hello",
  "title": "Doctor",
  "name": "María",
  "not-evaluated": "$customerName"
 }
```

In the next diagram, you can see a graphical representation showing how converting JSONPath (left) to JSONata (right) will reduce the complexity of the steps in your state machines:



You can (optionally) select and transform data from the state input into **Arguments** to send to your integrated action. With JSONata, you can then (optionally) select and transform the **results** from the action for assigning to variables and for state **Output**.

Note: **Assign** and **Output** steps occur in **parallel**. If you choose to transform data during variable assignment, that transformed data will **not** be available in the Output step. You must reapply the JSONata transformation in the Output step.



QueryLanguage field

In your workflow ASL definitions, there is a `QueryLanguage` field at the top level of a state machine definition and in individual states. By setting `QueryLanguage` inside individual states, you can incrementally adopt JSONata in an existing state machine rather than upgrading the state machine all at once.

The `QueryLanguage` field can be set to `"JSONPath"` or `"JSONata"`. If the top-level `QueryLanguage` field is omitted, it defaults to `"JSONPath"`. If a state contains a state-level `QueryLanguage` field, Step Functions will use the specified query language for that state. If the state does not contain a `QueryLanguage` field, then it will use the query language specified in the top-level `QueryLanguage` field.

Writing JSONata expressions in JSON strings

When a string in the value of an ASL field, a JSON object field, or a JSON array element is surrounded by `{% %}` characters, that string will be evaluated as JSONata . Note, the string must start with `{%` with no leading spaces, and must end with `%}` with no trailing spaces. Improperly opening or closing the expression will result in a validation error.

Some examples:

- `"TimeoutSeconds" : "{% $timeout %}"`
- `"Arguments" : {"field1" : "{% $name %}"}` in a `Task` state
- `"Items": [1, "{% $two %}", 3]` in a `Map` state

Not all ASL fields accept JSONata. For example, each state's `Type` field must be set to a constant string. Similarly, the `Task` state's `Resource` field must be a constant string. The `Map` state `Items` field will accept a JSON array or a JSONata expression that must evaluate to an array.

Reserved variable : $states

Step Functions defines a single reserved variable called **`$states`**. In JSONata states, the following structures are assigned to `$states` for use in JSONata expressions:

```
# Reserved $states variable in JSONata states
$states = {
  "input":       // Original input to the state
  "result":      // API or sub-workflow's result (if successful)
  "errorOutput": // Error Output (only available in a Catch)
  "context":     // Context object
}
```

On state entry, Step Functions assigns the state input to **`$states.input`**. The value of `$states.input` can be used in all fields that accept JSONata expressions. `$states.input` always refers to the original state input.

For `Task`, `Parallel`, and `Map` states:

- **`$states.result`** refers to the API or sub-workflow's raw result if successful.
- **`$states.errorOutput`** refers to the Error Output if the API or sub-workflow failed.

  `$states.errorOutput` can be used in the `Catch` field's `Assign` or `Output`.

Attempting to access `$states.result` or `$states.errorOutput` in fields and states where they are not accessible will be caught at creation, update, or validation of the state machine.

The `$states.context` object provides your workflows information about their specific execution, such as `StartTime`, task token, and initial workflow input. To learn more, see Accessing execution data from the Context object in Step Functions .

Handling expression errors

At runtime, JSONata expression evaluation might fail for a variety of reasons, such as:

- **Type error** - An expression, such as `{% $x + $y %}`, will fail if `$x` or `$y` is not a number.
- **Type incompatibility** - An expression might evaluate to a type that the field will not accept. For example, the field `TimeoutSeconds` requires a numeric input, so the expression `{% $timeout %}` will fail if `$timeout` returns a string.
- **Value out of range** - An expression that produces a value that is outside the acceptable range for a field will fail. For example, an expression such as `{% $evaluatesToNegativeNumber %}` will fail in the `TimeoutSeconds` field.
- **Failure to return a result** - JSON cannot represent an undefined value expression, so the expression `{% $data.thisFieldDoesNotExist %}` would result in an error.

In each case, the interpreter will throw the error: `States.QueryEvaluationError`. Your Task, Map, and Parallel states can provide a `Catch` field to catch the error, and a `Retry` field to retry on the error.

Converting from JSONPath to JSONata

The following sections compare and explain the differences between code written with JSONPath and JSONata.

No more path fields

ASL requires developers use `Path` versions of fields, as in `TimeoutSecondsPath`, to select a value from the state data when using JSONPath. When you use JSONata, you no longer use `Path` fields because ASL will interpret `{% %}`-enclosed JSONata expressions automatically for you in non-Path fields, such as `TimeoutSeconds`.

- JSONPath legacy example: `"TimeoutSecondsPath": "$timeout"`
- JSONata : `"TimeoutSeconds": "{% $timeout %}"`

Similarly, the `Map` state `ItemsPath` has been replaced with the `Items` field which accepts a JSON array or a JSONata expression that must evaluate to an array.

JSON Objects

ASL uses the term *payload template* to describe a JSON object that can contain JSONPath expressions for `Parameters` and `ResultSelector` field values. ASL will not use the term payload template for JSONata because JSONata evaluation happens for all strings whether they occur on their own or inside a JSON object or a JSON array.

No more .$

ASL requires you to append '`.$`' to field names in payload templates to use JSONPath and Intrinsic Functions. When you specify `"QueryLanguage":"JSONata"`, you no longer use the '`.$`' convention for JSON object field names. Instead, you enclose JSONata expressions in `{% %}` characters. You use the same convention for all string-valued fields, regardless of how deeply the object is nested inside other arrays or objects.

Arguments and Output Fields

When the `QueryLanguage` is set to `JSONata`, the old I/O processing fields will be disabled (`InputPath`, `Parameters`, `ResultSelector`, `ResultPath` and `OutputPath`) and most states will get two new fields: `Arguments` and `Output`.

JSONata provides a simpler way to perform I/O transformations compared to the fields used with JSONPath. JSONata's features makes `Arguments` and `Output` more capable than the previous five fields with JSONPath. These new field names also help simplify your ASL and clarify the model for passing and returning values.

The `Arguments` and `Output` fields (and other similar fields such as `Map` state's `ItemSelector`) will accept either a JSON object such as:

```
"Arguments": {
    "field1": 42,
    "field2": "{% jsonata expression %}"
}
```

Or, you can use a JSONata expression directly, for example:

```
"Output": "{% jsonata expression %}"
```

Output can also accept any type of JSON value too, for example: `"Output":true`, `"Output":42`.

The `Arguments` and `Output` fields only support JSONata, so it is invalid to use them with workflows that use JSONPath. Conversely, `InputPath`, `Parameters`, `ResultSelector`, `ResultPath`, `OutputPath` , and other JSONPath fields are only supported in JSONPath, so it is invalid to use path-based fields when using JSONata as your top level workflow or state query language.

Pass state

The optional **Result** in a Pass state was previously treated as the *output* of a virtual task. With JSONata selected as the workflow or state query language, you can now use the new **Output** field.

Choice state

When using JSONPath, choice states have an input `Variable` and numerous comparison paths, such as the following `NumericLessThanEqualsPath` :

```
# JSONPath choice state sample, with Variable and comparison path
"Check Price": {
  "Type": "Choice",
  "Default": "Pause",
  "Choices": [
  {
    "Variable": "$.current_price.current_price",
    "NumericLessThanEqualsPath": "$.desired_price",
    "Next": "Send Notification"
  } ],
}
```

With JSONata, the choice state has a `Condition` where you can use a JSONata expression:

```
# Choice state after JSONata conversion
"Check Price": {
  "Type": "Choice",
  "Default": "Pause"
  "Choices": [
    {
      "Condition": "{% $current_price <= $states.input.desired_priced %}",
      "Next": "Send Notification"
    } ]
```

Note: Variables and comparison fields are only available for JSONPath. Condition is only available for JSONata.

JSONata examples

The following examples can be created in Workflow Studio to experiment with JSONata. You can create and execute the state machines, or use the **Test state** to pass in data and even modify the state machine definition.

Example: Input and Output

This example shows how to use `$states.input` to use the state input and the `Output` field to specify the state output when you opt into JSONata.

```
{
  "Comment": "Input and Output example using JSONata",
  "QueryLanguage": "JSONata",
  "StartAt": "Basic Input and Output",
  "States": {
    "Basic Input and Output": {
      "QueryLanguage": "JSONata",
      "Type": "Succeed",
      "Output": {
        "lastName": "{% 'Last=>' & $states.input.customer.lastName %}",
        "orderValue": "{% $states.input.order.total %}"
      }
    }
  }
}
```

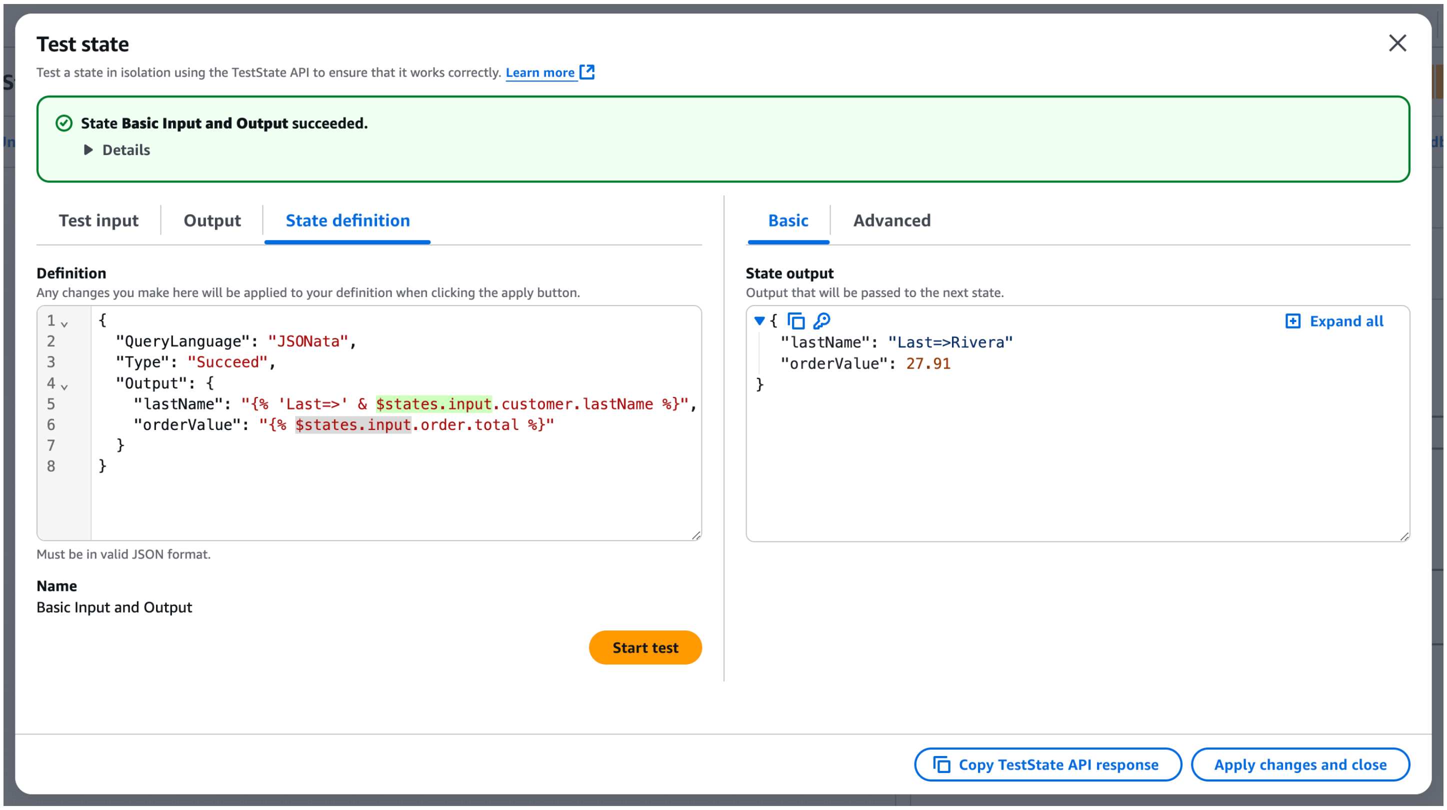
When the workflow is executed with the following as input:

```
{
  "customer": {
    "firstName": "Martha",
    "lastName": "Rivera"
  },
  "order": {
    "items": 7,
    "total": 27.91
  }
```

```
}
```

Test state or state machine execution will return the following JSON output:

```
{
  "lastName": "Last=>Rivera",
  "orderValue": 27.91
}
```



Example: Filtering with JSONata

You can filter your data with JSONata Path operators. For example, imagine you have a list of products for input, and you only want to process products that contain zero calories. You can create a state machine definition with the following ASL and test the `FilterDietProducts` state with the sample input that follows.

State machine definition for filtering with JSONata

```
{
  "Comment": "Filter products using JSONata",
  "QueryLanguage": "JSONata",
```

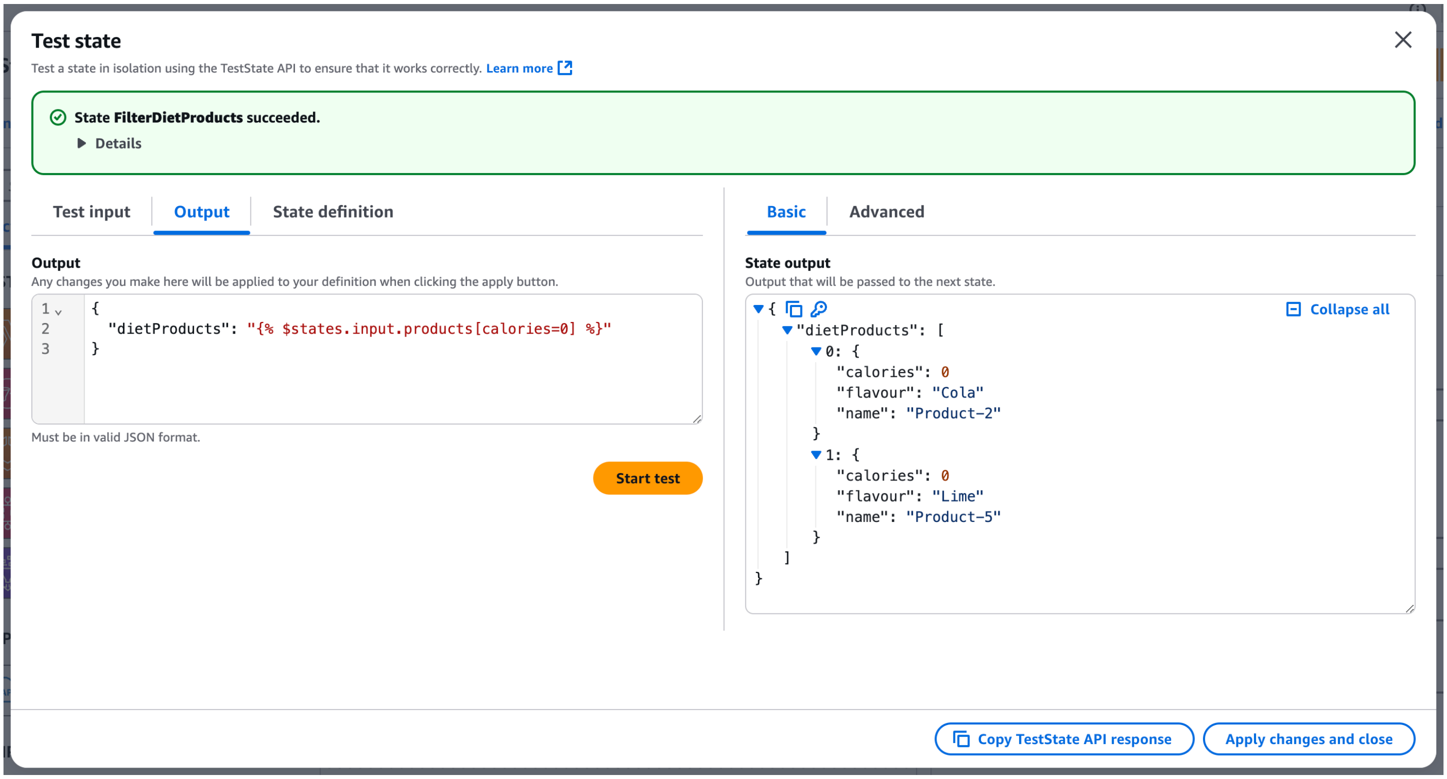
```
  "StartAt": "FilterDietProducts",
  "States": {
    "FilterDietProducts": {
      "Type": "Pass",
      "Output": {
        "dietProducts": "{% $states.input.products[calories=0] %}"
      },
      "End": true
    }
  }
}
```

Sample input for the test

```
{
  "products": [
    {
      "calories": 140,
      "flavour": "Cola",
      "name": "Product-1"
    },
    {
      "calories": 0,
      "flavour": "Cola",
      "name": "Product-2"
    },
    {
      "calories": 160,
      "flavour": "Orange",
      "name": "Product-3"
    },
    {
      "calories": 100,
      "flavour": "Orange",
      "name": "Product-4"
    },
    {
      "calories": 0,
      "flavour": "Lime",
      "name": "Product-5"
    }
  ]
}
```

Output from testing the step in your state machine

```
{
    "dietProducts": [
        {
            "calories": 0,
            "flavour": "Cola",
            "name": "Product-2"
        },
        {
            "calories": 0,
            "flavour": "Lime",
            "name": "Product-5"
        }
    ]
}
```



JSONata functions provided by Step Functions

JSONata contains function libraries for String, Numeric, Aggregation, Boolean, Array, Object, Date/Time, and High Order functions. Step Functions provides additional JSONata functions that you can use in your JSONata expressions. These built-in functions serve as replacements for Step

Functions intrinsic functions. Intrinsic functions are only available in states that use the JSONPath query language.

Note: Built-in JSONata functions that require integer values as parameters will automatically round down any non-integer numbers provided.

$partition - JSONata equivalent of `States.ArrayPartition` intrinsic function to partition a large array.

The first parameter is the array to partition, the second parameter is an integer representing the chunk size. The return value will be a two-dimensional array. The interpreter chunks the input array into multiple arrays of the size specified by chunk size. The length of the last array chunk may be less than the length of the previous array chunks if the number of remaining items in the array is smaller than the chunk size.

```
"Assign": {
  "arrayPartition": "{% $partition([1,2,3,4], $states.input.chunkSize) %}"
}
```

$range - JSONata equivalent of `States.ArrayRange` intrinsic function to generate an array of values.

This function takes three arguments. The first argument is an integer representing the first element of the new array, the second argument is an integer representing the final element of the new array, and the third argument is the delta value integer for the elements in the new array. The return value is a newly-generated array of values ranging from the first argument of the function to the second argument of the function with elements in between adjusted by the delta. The delta value can be positive or negative which will increment or decrement each element from the last until the end value is reached or exceeded.

```
"Assign": {
  "arrayRange": "{% $range(0, 10, 2) %}"
}
```

$hash - JSONata equivalent of the `States.Hash` intrinsic function to calculate the hash value of a given input.

This function takes two arguments. The first argument is the source string to be hashed. The second argument is a string representing the hashing algorithm to for the hash calculation.

The hashing algorithm must be one of the following values: `"MD5"`, `"SHA-1"`, `"SHA-256"`, `"SHA-384"`, `"SHA-512"`. The return value is a string of the calculated hash of the data.

This function was created because JSONata does not natively support the ability to calculate hashes.

```
"Assign": {
  "myHash": "{% $hash($states.input.content, $hashAlgorithmName) %}"
}
```

$random - JSONata equivalent of the `States.MathRandom` intrinsic function to return a random number n where `0 ≤ n < 1`.

The function takes an *optional* integer argument representing the seed value of the random function. If you use this function with the same seed value, it returns an identical number.

This overloaded function was created because the built-in JSONata function [`$random`] does not accept a seed value.

```
"Assign": {
   "randNoSeed": "{% $random() %}",
   "randSeeded": "{% $random($states.input.seed) %}"
}
```

$uuid - JSONata version of the `States.UUID` intrinsic function.

The function takes no arguments. This function return a v4 UUID.

This function was created because JSONata does not natively support the ability to generate UUIDs.

```
"Assign": {
  "uniqueId": "{% $uuid() %}"
}
```

$parse - JSONata function to deserialize JSON strings.

The function takes a stringified JSON as its only argument.

JSONata supports this functionality via `$eval`; however, `$eval` is not supported in Step Functions workflows.

```
"Assign": {
  "deserializedPayload": "{% $parse($states.input.json_string) %}"
}
```

Accessing execution data from the Context object in Step Functions

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

The Context object is an internal JSON structure that is available during an execution, and contains information about your state machine and execution. The context provides your workflows information about their specific execution. Your workflows can reference the Context object in a JSONata expression with `$states.context`.

Accessing the Context object

To access the Context object in JSONata

To access the Context object in JSONata states, use `$states.context` in a JSONata expression.

```
{
  "ExecutionID" : "{% $states.context.Execution.Id %}"
}
```

To access the Context object in JSONPath

To access the Context object in JSONPath, you first append `.$` to the end of the key to indicate the value is a path. Then, prepend the value with `$$.` to select a node in the Context object.

```
{
  "ExecutionID.$": "$$.Execution.Id"
}
```

JSONPath states can refer to the context (`$$.`) from the following JSONPath fields:

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

Context object fields

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

```
{
    "Execution": {
        "Id": "String",
        "Input": {},
        "Name": "String",
        "RoleArn": "String",
        "StartTime": "Format: ISO 8601",
        "RedriveCount": Number,
        "RedriveTime": "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. `RedriveTime` Context object is only available if you've redriven an execution. If you've redriven a Map Run, the

RedriveTime context object is only available for child workflows of type Standard. For a redriven Map Run with child workflows of type Express, RedriveTime isn't available.

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

```
{
    "Execution": {
        "Id":
 "arn:aws:states:region:123456789012:execution:stateMachineName:executionName",
        "Input": {
           "key": "value"
        },
        "Name": "executionName",
        "RoleArn": "arn:aws:iam::123456789012:role...",
        "StartTime": "2025-08-27T10:04:42Z"
    },
    "State": {
        "EnteredTime": "2025-08-27T10:04:42.001Z",
        "Name": "Test",
        "RetryCount": 3
    },
    "StateMachine": {
        "Id": "arn:aws:states:region:123456789012:stateMachine:stateMachineName",
        "Name": "stateMachineName"
    },
    "Task": {
        "Token": "h7XRiCdLtd/83p1E0dMccoxlzFhglsdkzpK9mBVKZsp7d9yrT1W"
    }
}
```

Timestamp format with fractional seconds

Step Functions follows the ISO8601 specification which states that output can be zero, three, six or nine digits digits as necessary. When a timestamp has zero fractional seconds, Step Functions removes the trailing zeros rather than pad the output.
If you create code that consumes Step Functions timestamps, your code must be able to process a variable number of fractional seconds.

Context object data for Map states

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

There are two additional items available in the Context object when processing a Map state: `Index` and `Value`. For each `Map` state iteration, `Index` contains the index number for the array item that is being currently processed, while `Value` contains the array item being processed. Within a `Map` state, the Context object includes the following data:

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

These are available only in a `Map` state, and can be specified in the `ItemSelector (Map)` field.

Note

You must define parameters from the Context object in the `ItemSelector` block of the main `Map` state, not within the states included in the `ItemProcessor` section.

Given a state machine using a JSONPath `Map` state, you can inject information from the Context object as follows.

```
{
  "StartAt": "ExampleMapState",
  "States": {
    "ExampleMapState": {
      "Type": "Map",
      "ItemSelector": {
        "ContextIndex.$": "$$.Map.Item.Index",
        "ContextValue.$": "$$.Map.Item.Value"
      },
```

```
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "INLINE"
        },
        "StartAt": "TestPass",
        "States": {
          "TestPass": {
            "Type": "Pass",
            "End": true
          }
        }
      },
      "End": true
    }
  }
}
```

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

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

The output for the execution returns the values of `Index` and `Value` items for each of the three iterations as follows:

```
[
  {
    "ContextIndex": 0,
    "ContextValue": {
      "who": "bob"
    }
  },
```

```
  {
    "ContextIndex": 1,
    "ContextValue": {
      "who": "meg"
    }
  },
  {

    "ContextIndex": 2,
    "ContextValue": {
      "who": "joe"
    }
  }
]
```

Using JSONPath paths

> **Managing state and transforming data**
>
> Learn about Passing data between states with variables and Transforming data with JSONata.

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, which is only available when the `QueryLanguage` is set to JSONPath. You can specify a path to access subsets of the input when specifying values for `InputPath`, `ResultPath`, and `OutputPath`.

You must use square bracket notation if your field name contains any character that is not included in the `member-name-shorthand` definition of the JsonPath ABNF rule. Therefore, to encode special characters, such as punctuation marks (excluding _), you must use square bracket notation. For example, `$.abc.['def ghi']`.

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.
- Functions such as `length()` aren't supported.

- Lexical operators, which are non-symbolic, such as `subsetof` aren't supported.
- Filtering by regular expression or by referencing another value in the JSON structure is not supported.
- The operators `@`, `,`, `:`, and `?` are not supported

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

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

The following reference paths would return the following.

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

Certain states use paths and reference paths to control the flow of a state machine or configure a state's settings or options. For more information, see Modeling workflow input and output path processing with data flow simulator and Using JSONPath effectively in AWS Step Functions.

Flattening an array of arrays

If the Parallel workflow state or Map workflow state state in your state machines return an array of arrays, you can transform them into a flat array with the ResultSelector field. You can include this field inside the Parallel or Map state definition to manipulate the result of these states.

To flatten arrays, use the syntax: `[*]` in the `ResultSelector` field as shown in the following example.

```
"ResultSelector": {
    "flattenArray.$": "$[*][*]"
  }
```

For examples that show how to flatten an array, see *Step 3* in the following tutorials:

- Processing batch data with a Lambda function in Step Functions
- Processing individual items with a Lambda function in Step Functions

Manipulate parameters in Step Functions workflows

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

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 Using JSONPath paths). With the `Parameters` field, you can pass a collection of key-value pairs, using either static values or selections 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.

InputPath

Use `InputPath` to select a portion of the state input.

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

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

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

You could apply the `InputPath`.

```
"InputPath": "$.dataset2",
```

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

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

> **Note**
>
> A path can yield a selection of values. Consider the following example.
>
> ```
> { "a": [1, 2, 3, 4] }
> ```
>
> If you apply the path `$.a[0:2]`, the following is the result.
>
> ```
> [ 1, 2 ]
> ```

Parameters

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.

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

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

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

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

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

In addition to the input, you can access a special JSON object, known as the Context object. The Context object includes information about your state machine execution. See [Accessing execution data from the Context object in Step Functions](#).

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:

- Passing parameters to a service API in Step Functions
- Integrating services

Amazon S3

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 implement one of the following methods:

- Use the *Distributed Map state* in your workflow so that the `Map` state can read input directly from Amazon S3 data sources. For more information, see Distributed mode.
- Parse the Amazon Resource Name (ARN) of the bucket in the `Payload` parameter to get the bucket name and key value. For more information, see Using Amazon S3 ARNs instead of passing large payloads in Step Functions.

Alternatively, you can adjust your implementation to pass smaller payloads in your executions.

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. Using the `ResultSelector` field, you can choose what parts of a state's result you want to pass to the `ResultPath` field.

> **Note**
>
> With the `ResultPath` field, you can add the output of the `ResultSelector` field to the original input.

`ResultSelector` is an optional field in the following states:

- Map workflow state
- Task workflow state
- Parallel workflow state

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:

```
{
  "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`:

```
"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:

```
{
  "OtherDataFromInput": {},
  "EMROutput": {
      "ClusterId": "AKIAIOSFODNN7EXAMPLE",
      "ResourceType": "elasticmapreduce",
  }
}
```

Flattening an array of arrays

If the Parallel workflow state or Map workflow state state in your state machines return an array of arrays, you can transform them into a flat array with the ResultSelector field. You can include this field inside the Parallel or Map state definition to manipulate the result of these states.

To flatten arrays, use the syntax: `[*]` in the `ResultSelector` field as shown in the following example.

```
"ResultSelector": {
    "flattenArray.$": "$[*][*]"
  }
```

For examples that show how to flatten an array, see *Step 3* in the following tutorials:

- Processing batch data with a Lambda function in Step Functions
- Processing individual items with a Lambda function in Step Functions

Example: Manipulating state data with paths in Step Functions workflows

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

This topic contains examples of how to manipulate state input and output JSON using the InputPath, ResultPath, and OutputPath fields.

Any state other than a Fail workflow state state or a Succeed workflow state state can include the input and output processing fields, such as `InputPath`, `ResultPath`, or `OutputPath`. Additionally, the Wait workflow state and Choice workflow state states don't support the `ResultPath` field. With these fields, you can use a JsonPath to filter the JSON data as it moves through your workflow.

You can also use the `Parameters` field to manipulate the JSON data as it moves through your workflow. For information about using `Parameters`, see Manipulate parameters in Step Functions workflows.

For example, start with the AWS Lambda function and state machine described in the Creating a Step Functions state machine that uses Lambda tutorial. Modify the state machine so that it includes the following `InputPath`, `ResultPath`, and `OutputPath`.

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:region:123456789012:function:HelloFunction",
      "InputPath": "$.lambda",
      "ResultPath": "$.data.lambdaresult",
      "OutputPath": "$.data",
      "End": true
    }
  }
```

```
}
```

Start an execution using the following input.

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

Assume that the `comment` and `extra` nodes can be discarded, but that you 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. Because you are not performing any other manipulation on the original input and the result using `OutputPath`, the output of the state now includes the result of the Lambda function with the original input.

```
{
  "comment": "An input comment.",
  "data": {
    "val1": 23,
    "val2": 17,
```

```
    "lambdaresult": "Hello, AWS Step Functions!"
  },
  "extra": "foo",
  "lambda": {
    "who": "AWS Step Functions"
  }
}
```

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

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

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

```
{
  "val1": 23,
  "val2": 17,
  "lambdaresult": "Hello, AWS Step Functions!"
}
```

In this `Task` state:

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

Example to manipulate original state machine input, result, and final output using JsonPath

Consider the following state machine that verifies an insurance applicant's identity and address.

> **Note**
>
> To view the complete example, see How to use JSON Path in Step Functions.

```
{
```

```
  "Comment": "Sample state machine to verify an applicant's ID and address",
  "StartAt": "Verify info",
  "States": {
    "Verify info": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "Verify identity",
          "States": {
            "Verify identity": {
              "Type": "Task",
              "Resource": "arn:aws:states:::lambda:invoke",
              "Parameters": {
                "Payload.$": "$",
                "FunctionName": "arn:aws:lambda:us-east-2:111122223333:function:check-
identity:$LATEST"
              },
              "End": true
            }
          }
        },
        {
          "StartAt": "Verify address",
          "States": {
            "Verify address": {
              "Type": "Task",
              "Resource": "arn:aws:states:::lambda:invoke",
              "Parameters": {
                "Payload.$": "$",
                "FunctionName": "arn:aws:lambda:us-east-2:111122223333:function:check-
address:$LATEST"
              },
              "End": true
            }
          }
        }
      ]
    }
  }
}
```

If you run this state machine using the following input, the execution fails because the Lambda functions that perform verification only expect the data that needs to be verified as input. Therefore, you must specify the nodes that contain the information to be verified using an appropriate JsonPath.

```
{
  "data": {
    "firstname": "Jane",
    "lastname": "Doe",
    "identity": {
      "email": "jdoe@example.com",
      "ssn": "123-45-6789"
    },
    "address": {
      "street": "123 Main St",
      "city": "Columbus",
      "state": "OH",
      "zip": "43219"
    },
    "interests": [
      {
        "category": "home",
        "type": "own",
        "yearBuilt": 2004
      },
      {
        "category": "boat",
        "type": "snowmobile",
        "yearBuilt": 2020
      },
      {
        "category": "auto",
        "type": "RV",
        "yearBuilt": 2015
      },
    ]
  }
}
```

To specify the node that the *`check-identity`* Lambda function must use, use the `InputPath` field as follows:

```
"InputPath": "$.data.identity"
```

And to specify the node that the *check-address* Lambda function must use, use the `InputPath` field as follows:

```
"InputPath": "$.data.address"
```

Now if you want to store the verification result within the original state machine input, use the `ResultPath` field as follows:

```
"ResultPath": "$.results"
```

However, if you only need the identity and verification results and discard the original input, use the `OutputPath` field as follows:

```
"OutputPath": "$.results"
```

For more information, see Processing input and output in Step Functions.

Filtering state output using OutputPath

With `OutputPath` you can select a portion of the state output to pass to the next state. With this approach, you can filter out unwanted information, and pass only the portion of JSON that you need.

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.

Specifying state output using ResultPath in Step Functions

Managing state and transforming data

This page refers to JSONPath. Step Functions recently added variables and JSONata to manage state and transform data.
Learn about Passing data with variables and Transforming data with JSONata.

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 workflow state
- Task workflow state
- Parallel workflow state
- Map workflow state

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, which limit scope so the path must identify only a single node in JSON. See Reference Paths in the Amazon States Language.

Use ResultPath to replace input with the task result

If you do not specify a `ResultPath`, the default behavior is the same as `"ResultPath": "$"`. The state will replace the entire state input with the result from the task.

```
# State Input
{
 "comment": "This is a test",
 "details": "Default example",
 "who" : "Step Functions"
}

# Path
"ResultPath": "$"

# Task result
"Hello, Step Functions!"

# State Output
```

```
"Hello, 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 action is to replace the entire input.

Discard the result and keep the original input

If you set `ResultPath` to `null`, the state will pass the **original input** to the output. The state's input payload will be copied directly to the output, with no regard for the task result.

```
# State Input
{
 "comment": "This is a test",
 "details": "Default example",
 "who" : "Step Functions"
}

# Path
"ResultPath": null

# Task result
"Hello, Step Functions!"

# State Output
{
 "comment": "This is a test",
 "details": "Default example",
 "who" : "Step Functions"
}
```

Use ResultPath to include the result with the input

If you specify a path for ResultPath, the state output will combine the state input and task result:

```
# State Input
{
 "comment": "This is a test",
```

```
 "details": "Default example",
 "who" : "Step Functions"
}

# Path
"ResultPath": "$.taskresult"

# Task result
"Hello, Step Functions!"

# State Output
{
 "comment": "This is a test",
 "details": "Default example",
 "who" : "Step Functions",
 "taskresult" : "Hello, Step Functions!"
}
```

You can also insert the result into a child node of the input. Set the `ResultPath` to the following.

```
"ResultPath": "$.strings.lambdaresult"
```

Given the following input:

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

The task result would be inserted as a child of the `strings` node in the input.

```
{
  "comment": "An input comment.",
  "strings": {
    "string1": "foo",
    "string2": "bar",
    "string3": "baz",
```

```
    "lambdaresult": "Hello, 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

If you specify an existing node for ResultPath, the task result will replace that existing node:

```
# State Input
{
 "comment": "This is a test",
 "details": "Default example",
 "who" : "Step Functions"
}

# Path
"ResultPath": "$.comment"

# Task result
"Hello, Step Functions!"

# State Output
{
 "comment": "Hello, Step Functions!",
 "details": "Default example",
 "who" : "Step Functions"
}
```

Use ResultPath to include both error and input in a `Catch`

In some cases, you might want to preserve the original input with the error. Use `ResultPath` in a `Catch` to include the error with the original input, instead of replacing it.

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

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

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

The state output when catching an error is the following.

```
{
  "foo": "bar",
  "error": {
    "Error": "Error here"
  }
}
```

For more information about error handling, see the following:

- Handling errors in Step Functions workflows
- Handling error conditions in a Step Functions state machine

Map state input and output fields in Step Functions

Managing state and transforming data

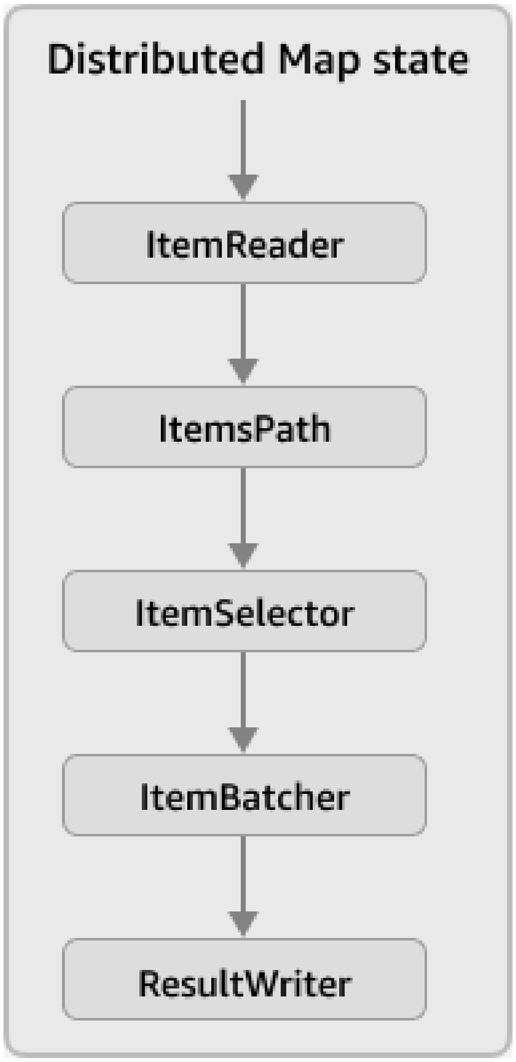
Learn about Passing data between states with variables and Transforming data with JSONata.

`Map` states concurrently iterate over a collection of items in a dataset, such as a JSON array, a list of Amazon S3 objects, or the rows of JSON Lines or a CSV file in an Amazon S3 bucket. It repeats a set of steps for each item in the collection. You can configure the input that the `Map` state receives and the output it generates using these fields. Step Functions applies each field in your *Distributed Map state* in the order shown in the following list and illustration:

Note

Based on your use case, you may not need to apply all of these fields.

1. ItemReader (Map)
2. ItemsPath (Map, JSONPath only)
3. ItemSelector (Map)
4. ItemBatcher (Map)
5. ResultWriter (Map)



Note

These `Map` state input and output fields are currently unavailable in the data flow simulator in the Step Functions console.

ItemReader (Map)

The `ItemReader` field is a JSON object, which specifies a dataset and its location. A *Distributed Map state* uses this dataset as its input.

The following example shows the syntax of the `ItemReader` field in a **JSONPath-based** workflow, for a dataset in a text delimited file that's stored in an Amazon S3 bucket.

```
"ItemReader": {
    "ReaderConfig": {
        "InputType": "CSV",
        "CSVHeaderLocation": "FIRST_ROW"
    },
    "Resource": "arn:aws:states:::s3:getObject",
    "Parameters": {
        "Bucket": "myBucket",
        "Key": "csvDataset/ratings.csv",
        "VersionId": "BcK42coT2jE1234VHLUvBV1yLNod2OEt"
    }
}
```

The following example shows that in JSONata-based workflows, `Parameters` is replaced with **Arguments**.

```
"ItemReader": {
    "ReaderConfig": {
        "InputType": "CSV",
        "CSVHeaderLocation": "FIRST_ROW"
    },
    "Resource": "arn:aws:states:::s3:getObject",
    "Arguments": {
        "Bucket": "amzn-s3-demo-bucket",
        "Key": "csvDataset/ratings.csv"
    }
}
```

Tip

In Workflow Studio, you specify the dataset and its location in the **Item source** field.

Contents

Contents of the ItemReader field

Depending on your dataset, the contents of the `ItemReader` field varies. For example, if your dataset is a JSON array passed from a previous step in the workflow, the `ItemReader` field is omitted. If your dataset is an Amazon S3 data source, this field contains the following sub-fields.

`ReaderConfig`

A JSON object that specifies the following details:

- `InputType`

  Accepts one of the following values: `CSV`, `JSON`, `JSONL`,`MANIFEST`.

  Specifies the type of Amazon S3 data source, such as a text delimited file (`CSV`), object, JSON file, JSON Lines, or an Amazon S3 inventory list. In Workflow Studio, you can select an input type from the **Amazon S3 item source** dropdown list under the **Item source** field.

- `CSVDelimiter`

  Specify this field only if you use `CSV` as `InputType`, which indicates a text delimited file. Accepts one of the following values: `COMMA` (default), `PIPE`, `SEMICOLON`, `SPACE`, `TAB`.

  > **Note**
  >
  > The `CSVDelimiter` field enables `ItemReader` more flexibility to support files that are delimited by other characters besides the comma. Therefore, assume that our references to CSV files in relation to `ItemReader` also include files that use delimiters accepted by the `CSVDelimiter` field.

- `CSVHeaderLocation`

  Required if `InputType` is `CSV`, which indicates a text delimited file with delimiters accepted by the `CSVDelimiter` field.

Accepts one of the following values to specify the location of the column header:

- `FIRST_ROW` – Use this option if the first line of the file is the header.
- `GIVEN` – Use this option to specify the header within the state machine definition. For example, if your file contains the following data.

  ```
  1,307,3.5,1256677221
  1,481,3.5,1256677456
  1,1091,1.5,1256677471
  ...
  ```

  Provide the following JSON array as a CSV header.

  ```
  "ItemReader": {
      "ReaderConfig": {
          "InputType": "CSV",
          "CSVHeaderLocation": "GIVEN",
          "CSVHeaders": [
              "userId",
              "movieId",
              "rating",
              "timestamp"
          ]
      }
  }
  ```

> **Important**
>
> Currently, Step Functions supports headers of up to 10 KiB for text delimited files.

> **Tip**
>
> In Workflow Studio, you can find this option under **Additional configuration** in the **Item source** field.

- `MaxItems`

Limits the number of data items passed to the `Map` state. For example, suppose that you provide a text delimited file that contains 1000 rows and specify a limit of 100. Then, the interpreter passes *only* 100 rows to the `Map` state. The `Map` state processes items in sequential order, starting after the header row.

By default, the `Map` state iterates over all the items in the specified dataset.

> **Note**
>
> Currently, you can specify a limit of up to 100,000,000. The *Distributed Map state* stops reading items beyond this limit.

> **Tip**
>
> In Workflow Studio, you can find this option under **Additional configuration** in the **Item source** field.

Alternatively, you can specify a reference path to an existing key-value pair in your *Distributed Map state* input. This path must resolve to a positive integer. You specify the reference path in the `MaxItemsPath` sub-field.

> **Important**
>
> You can specify either the `MaxItems` or the `MaxItemsPath` sub-field, but not both.

`Resource`

The Amazon S3 API action that Step Functions must invoke depending on the specified dataset.

`Parameters`

A JSON object that specifies the Amazon S3 bucket name and object key that the dataset is stored in. In this field, you can also provide the Amazon S3 object version, if the bucket has versioning enabled.

> **Important**
>
> Make sure that your Amazon S3 buckets are in the same AWS account and AWS Region as your state machine.
> Note that even though your state machine may be able to access files in buckets across different AWS accounts that are in the same AWS Region, Step Functions only supports state machines to list objects in S3 buckets that are in *both* the same AWS account and the same AWS Region as the state machine.

Examples of datasets

You can specify one of the following options as your dataset:

- JSON array from a previous step
- A list of Amazon S3 objects
- JSON file in an Amazon S3 bucket
- JSON Lines file in an Amazon S3 bucket
- CSV file in an Amazon S3 bucket
- Amazon S3 inventory list

> **Important**
>
> Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see IAM policies for datasets.

JSON array from a previous step

A *Distributed Map state* can accept a JSON input passed from a previous step in the workflow. This input must either be an array, or must contain an array within a specific node. To select a node that contains the array, you can use the `ItemsPath (Map, JSONPath only)` field.

To process individual items in the array, the *Distributed Map state* starts a child workflow execution for each array item. The following tabs show examples of the input passed to the `Map` state and the corresponding input to a child workflow execution.

> **Note**
>
> Step Functions omits the `ItemReader` field when your dataset is a JSON array from a previous step.

Input passed to the Map state

Consider the following JSON array of three items.

```
"facts": [
    {
        "verdict": "true",
        "statement_date": "6/11/2008",
        "statement_source": "speech"
    },
    {
        "verdict": "false",
        "statement_date": "6/7/2022",
        "statement_source": "television"
    },
    {
        "verdict": "mostly-true",
        "statement_date": "5/18/2016",
        "statement_source": "news"
    }
]
```

Input passed to a child workflow execution

The *Distributed Map state* starts three child workflow executions. Each execution receives an array item as input. The following example shows the input received by a child workflow execution.

```
{
  "verdict": "true",
  "statement_date": "6/11/2008",
  "statement_source": "speech"
}
```

Amazon S3 objects example

A *Distributed Map state* can iterate over the objects that are stored in an Amazon S3 bucket. When the workflow execution reaches the `Map` state, Step Functions invokes the [ListObjectsV2](#) API action, which returns an array of the Amazon S3 object metadata. In this array, each item contains data, such as **ETag** and **Key**, for the data stored in the bucket.

To process individual items in the array, the *Distributed Map state* starts a child workflow execution. For example, suppose that your Amazon S3 bucket contains 100 images. Then, the array returned after invoking the `ListObjectsV2` API action contains 100 items. The *Distributed Map state* then starts 100 child workflow executions to process each array item.

> **Note**
>
> - Currently, Step Functions also includes an item for each folder you create in a specific Amazon S3 bucket using the Amazon S3 console. This results in an extra child workflow execution started by the *Distributed Map state*. To avoid creating an extra child workflow execution for the folder, we recommend that you use the AWS CLI to create folders. For more information, see High-level Amazon S3 commands in the *AWS Command Line Interface User Guide*.
> - Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see IAM policies for datasets.

The following tabs show examples of the `ItemReader` field syntax and the input passed to a child workflow execution for this dataset.

ItemReader syntax

In this example, you've organized your data, which includes images, JSON files, and objects, within a prefix named `processData` in an Amazon S3 bucket named `amzn-s3-demo-bucket`.

```
"ItemReader": {
    "Resource": "arn:aws:states:::s3:listObjectsV2",
    "Parameters": {
        "Bucket": "amzn-s3-demo-bucket",
        "Prefix": "processData"
    }
}
```

Input passed to a child workflow execution

The *Distributed Map state* starts as many child workflow executions as the number of items present in the Amazon S3 bucket. The following example shows the input received by a child workflow execution.

```
{
  "Etag": "\"05704fbdccb224cb01c59005bebbad28\"",
  "Key": "processData/images/n02085620_1073.jpg",
  "LastModified": 1668699881,
  "Size": 34910,
  "StorageClass": "STANDARD"
}
```

JSON file in an Amazon S3 bucket

A *Distributed Map state* can accept a JSON file that's stored in an Amazon S3 bucket as a dataset. The JSON file must contain an array.

When the workflow execution reaches the `Map` state, Step Functions invokes the GetObject API action to fetch the specified JSON file. The `Map` state then iterates over each item in the array and starts a child workflow execution for each item. For example, if your JSON file contains 1000 array items, the `Map` state starts 1000 child workflow executions.

Note

- The execution input used to start a child workflow execution can't exceed 256 KiB. However, Step Functions supports reading an item of up to 8 MB from a text delimited file, JSON, or JSON Lines file if you then apply the optional `ItemSelector` field to reduce the item's size.
- Currently, Step Functions supports 10 GB as the maximum size of an individual file in Amazon S3.
- Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see IAM policies for datasets.

The following tabs show examples of the `ItemReader` field syntax and the input passed to a child workflow execution for this dataset.

For this example, imagine you have a JSON file named *factcheck.json*. You've stored this file within a prefix named *jsonDataset* in an Amazon S3 bucket. The following is an example of the JSON dataset.

```
[
  {
    "verdict": "true",
    "statement_date": "6/11/2008",
    "statement_source": "speech"
  },
  {
    "verdict": "false",
    "statement_date": "6/7/2022",
    "statement_source": "television"
  },
  {
    "verdict": "mostly-true",
    "statement_date": "5/18/2016",
    "statement_source": "news"
  },
  ...
]
```

ItemReader syntax

```
"ItemReader": {
    "Resource": "arn:aws:states:::s3:getObject",
    "ReaderConfig": {
        "InputType": "JSON"
    },
    "Parameters": {
        "Bucket": "amzn-s3-demo-bucket",
        "Key": "jsonDataset/factcheck.json"
    }
}
```

Input to a child workflow execution

The *Distributed Map state* starts as many child workflow executions as the number of array items present in the JSON file. The following example shows the input received by a child workflow execution.

```
{
  "verdict": "true",
  "statement_date": "6/11/2008",
  "statement_source": "speech"
}
```

JSON Lines file in an Amazon S3 bucket

A *Distributed Map state* can accept a JSON Lines file that's stored in an Amazon S3 bucket as a dataset.

> **Note**
>
> - The execution input used to start a child workflow execution can't exceed 256 KiB. However, Step Functions supports reading an item of up to 8 MB from a text delimited file, JSON, or JSON Lines file if you then apply the optional `ItemSelector` field to reduce the item's size.
> - Currently, Step Functions supports 10 GB as the maximum size of an individual file in Amazon S3.
> - Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see IAM policies for datasets.

The following tabs show examples of the `ItemReader` field syntax and the input passed to a child workflow execution for this dataset.

For this example, imagine you have a JSON Lines file named *`factcheck.jsonl`*. You've stored this file within a prefix named *`jsonlDataset`* in an Amazon S3 bucket. The following is an example of the file's contents.

```
{"verdict": "true", "statement_date": "6/11/2008", "statement_source": "speech"}
{"verdict": "false", "statement_date": "6/7/2022", "statement_source": "television"}
{"verdict": "mostly-true", "statement_date": "5/18/2016", "statement_source": "news"}
```

ItemReader syntax

```
"ItemReader": {
```

```
    "Resource": "arn:aws:states:::s3:getObject",
    "ReaderConfig": {
        "InputType": "JSONL"
    },
    "Parameters": {
        "Bucket": "amzn-s3-demo-bucket",
        "Key": "jsonlDataset/factcheck.jsonl"
    }
}
```

Input to a child workflow execution

The *Distributed Map state* starts as many child workflow executions as the number of lines present in the JSONL file. The following example shows the input received by a child workflow execution.

```
{
  "verdict": "true",
  "statement_date": "6/11/2008",
  "statement_source": "speech"
}
```

CSV file in an Amazon S3 bucket

Note

The `CSVDelimiter` field enables `ItemReader` more flexibility to support files that are delimited by other characters besides the comma. Therefore, assume that our references to CSV files in relation to `ItemReader` also include files that use delimiters accepted by the `CSVDelimiter` field.

A *Distributed Map state* can accept a text delimited file that's stored in an Amazon S3 bucket as a dataset. If you use a text delimited file as your dataset, you need to specify a column header. For information about how to specify a header, see Contents of the ItemReader field.

Step Functions parses text delimited files based on the following rules:

- The delimiter that separates fields is specified by `CSVDelimiter` in *ReaderConfig*. The delimiter defaults to `COMMA`.

- Newlines are a delimiter that separates **records**.
- Fields are treated as strings. For data type conversions, use the `States.StringToJson` intrinsic function in ItemSelector (Map).
- Double quotation marks (" ") are not required to enclose strings. However, strings that are enclosed by double quotation marks can contain commas and newlines without acting as record delimiters.
- You can preserve double quotes by repeating them.
- If the number of fields in a row is **less** than the number of fields in the header, Step Functions provides **empty strings** for the missing values.
- If the number of fields in a row is **more** than the number of fields in the header, Step Functions **skips** the additional fields.

For more information about how Step Functions parses a text delimited file, see Example of parsing an input CSV file.

When the workflow execution reaches the `Map` state, Step Functions invokes the GetObject API action to fetch the specified file. The `Map` state then iterates over each row in the file and starts a child workflow execution to process the items in each row. For example, suppose that you provide a text delimited file that contains 100 rows as input. Then, the interpreter passes each row to the `Map` state. The `Map` state processes items in serial order, starting after the header row.

Note

- The execution input used to start a child workflow execution can't exceed 256 KiB. However, Step Functions supports reading an item of up to 8 MB from a text delimited file, JSON, or JSON Lines file if you then apply the optional `ItemSelector` field to reduce the item's size.
- Currently, Step Functions supports 10 GB as the maximum size of an individual file in Amazon S3.
- Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see IAM policies for datasets.

The following tabs show examples of the `ItemReader` field syntax and the input passed to a child workflow execution for this dataset.

ItemReader syntax

For example, say that you have a CSV file named *ratings.csv*. Then, you've stored this file within a prefix that's named *csvDataset* in an Amazon S3 bucket.

```
{
  "ItemReader": {
    "ReaderConfig": {
      "InputType": "CSV",
      "CSVHeaderLocation": "FIRST_ROW",
      "CSVDelimiter": "PIPE"
    },
    "Resource": "arn:aws:states:::s3:getObject",
    "Parameters": {
      "Bucket": "amzn-s3-demo-bucket",
      "Key": "csvDataset/ratings.csv"
    }
  }
}
```

Input to a child workflow execution

The *Distributed Map state* starts as many child workflow executions as the number of rows present in the CSV file, excluding the header row, if in the file. The following example shows the input received by a child workflow execution.

```
{
  "rating": "3.5",
  "movieId": "307",
  "userId": "1",
  "timestamp": "1256677221"
}
```

S3 inventory example

A *Distributed Map state* can accept an Amazon S3 inventory manifest file that's stored in an Amazon S3 bucket as a dataset.

When the workflow execution reaches the `Map` state, Step Functions invokes the [GetObject](#) API action to fetch the specified Amazon S3 inventory manifest file. The `Map` state then iterates over the objects in the inventory to return an array of Amazon S3 inventory object metadata.

Note

- Currently, Step Functions supports 10 GB as the maximum size of an individual file in an Amazon S3 inventory report after decompression. However, Step Functions can process more than 10 GB if each individual file is under 10 GB.
- Step Functions needs appropriate permissions to access the Amazon S3 datasets that you use. For information about IAM policies for the datasets, see IAM policies for datasets.

The following is an example of an inventory file in CSV format. This file includes the objects named `csvDataset` and `imageDataset`, which are stored in an Amazon S3 bucket that's named `amzn-s3-demo-source-bucket`.

```
"amzn-s3-demo-source-bucket","csvDataset/","0","2022-11-16T00:27:19.000Z"
"amzn-s3-demo-source-bucket","csvDataset/
titles.csv","3399671","2022-11-16T00:29:32.000Z"
"amzn-s3-demo-source-bucket","imageDataset/","0","2022-11-15T20:00:44.000Z"
"amzn-s3-demo-source-bucket","imageDataset/
n02085620_10074.jpg","27034","2022-11-15T20:02:16.000Z"
...
```

Important

Currently, Step Functions doesn't support user-defined Amazon S3 inventory report as a dataset. You must also make sure that the output format of your Amazon S3 inventory report is CSV. For more information about Amazon S3 inventories and how to set them up, see Amazon S3 Inventory in the *Amazon S3 User Guide*.

The following example of an inventory manifest file shows the CSV headers for the inventory object metadata.

```
{
  "sourceBucket" : "amzn-s3-demo-source-bucket",
  "destinationBucket" : "arn:aws:s3:::amzn-s3-demo-inventory",
  "version" : "2016-11-30",
  "creationTimestamp" : "1668560400000",
  "fileFormat" : "CSV",
```

```
  "fileSchema" : "Bucket, Key, Size, LastModifiedDate",
  "files" : [ {
    "key" : "amzn-s3-demo-bucket/destination-prefix/
data/20e55de8-9c21-45d4-99b9-46c732000228.csv.gz",
    "size" : 7300,
    "MD5checksum" : "a7ff4a1d4164c3cd55851055ec8f6b20"
  } ]
}
```

The following tabs show examples of the `ItemReader` field syntax and the input passed to a child workflow execution for this dataset.

ItemReader syntax

```
{
  "ItemReader": {
    "ReaderConfig": {
      "InputType": "MANIFEST"
    },
    "Resource": "arn:aws:states:::s3:getObject",
    "Parameters": {
      "Bucket": "amzn-s3-demo-destination-bucket",
      "Key": "destination-prefix/amzn-s3-demo-bucket/config-id/YYYY-MM-DDTHH-MMZ/
manifest.json"
    }
  }
}
```

Input to a child workflow execution

```
{
  "LastModifiedDate": "2022-11-16T00:29:32.000Z",
  "Bucket": "amzn-s3-demo-source-bucket",
  "Size": "3399671",
  "Key": "csvDataset/titles.csv"
}
```

Depending on the fields you selected while configuring the Amazon S3 inventory report, the contents of your `manifest.json` file may vary from the example shown.

IAM policies for datasets

When you create workflows with the Step Functions console, Step Functions can automatically generate IAM policies based on the resources in your workflow definition. These policies include the least privileges necessary to allow the state machine role to invoke the `StartExecution` API action for the *Distributed Map state*. These policies also include the least privileges necessary Step Functions to access AWS resources, such as Amazon S3 buckets and objects and Lambda functions. We highly recommend that you include only those permissions that are necessary in your IAM policies. For example, if your workflow includes a `Map` state in Distributed mode, scope your policies down to the specific Amazon S3 bucket and folder that contains your dataset.

> **Important**
>
> If you specify an Amazon S3 bucket and object, or prefix, with a reference path to an existing key-value pair in your *Distributed Map state* input, make sure that you update the IAM policies for your workflow. Scope the policies down to the bucket and object names the path resolves to at runtime.

The following IAM policy examples grant the least privileges required to access your Amazon S3 datasets using the ListObjectsV2 and GetObject API actions.

Example IAM policy for Amazon S3 object as dataset

The following example shows an IAM policy that grants the least privileges to access the objects organized within *`processImages`* in an Amazon S3 bucket named *`amzn-s3-demo-bucket`*.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket"
            ],
            "Condition": {
                "StringLike": {
```

```
                    "s3:prefix": [
                        "processImages"
                    ]
                }
            }
        }
    ]
}
```

Example IAM policy for a CSV file as dataset

The following example shows an IAM policy that grants least privileges to access a CSV file named *ratings.csv*.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket/csvDataset/ratings.csv"
            ]
        }
    ]
}
```

Example IAM policy for an Amazon S3 inventory as dataset

The following example shows an IAM policy that grants least privileges to access an Amazon S3 inventory report.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
```

```
                "arn:aws:s3:::myDestination-prefix/amzn-s3-demo-bucket/myConfig-
id/YYYY-MM-DDTHH-MMZ/manifest.json",
                "arn:aws:s3:::myDestination-prefix/amzn-s3-demo-bucket/myConfig-id/
data/*"
            ]
        }
    ]
}
```

ItemsPath (Map, JSONPath only)

Managing state and transforming data

This page refers to JSONPath. Step Functions recently added variables and JSONata to manage state and transform data.
Learn about Passing data with variables and Transforming data with JSONata.

In JSONPath-based states, use the `ItemsPath` field to select an array within a JSON input provided to a `Map` state. The `Map` state repeats a set of steps for each item in the array. By default, the `Map` state sets `ItemsPath` to `$`, which selects the entire input. If the input to the `Map` state is a JSON array, it runs an iteration for each item in the array, passing that item to the iteration as input.

Note

You can use `ItemsPath` in the *Distributed Map state* only if you use a JSON input passed from a previous state in the workflow.

You can use the `ItemsPath` field to specify a location in the input that points to JSON array used for iterations. The value of `ItemsPath` must be a Reference Path, and that path must point to 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 it with `ItemsPath`. The following state machine definition specifies the `ThingsPiratesSay` array in the input using `ItemsPath`.It then runs an iteration of the `SayWord` pass state for each item in the `ThingsPiratesSay` array.

```
{
  "StartAt": "PiratesSay",
  "States": {
    "PiratesSay": {
      "Type": "Map",
      "ItemsPath": "$.ThingsPiratesSay",
      "ItemProcessor": {
         "StartAt": "SayWord",
         "States": {
           "SayWord": {
             "Type": "Pass",
             "End": true
           }
         }
```

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

When processing input, the Map state applies ItemsPath after InputPath. It operates on the effective input to the state after InputPath filters the input.

For more information on Map states, see the following:

- Map state
- Map state processing modes
- Repeat actions with Inline Map
- Inline Map state input and output processing

ItemSelector (Map)

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

By default, the effective input for the Map state is the set of individual data items present in the raw state input. The ItemSelector field lets you override the data items' values before they're passed on to the Map state. To override the values, specify a valid JSON input that contains a collection of key-value pairs. These pairs can be static values provided in your state machine definition, values selected from the state input using a path, or values accessed from the Context object.

If you specify key-value pairs using a path or Context object, the key name must end in .$.

Note

The ItemSelector field replaces the Parameters field within the Map state. If you use the Parameters field in your Map state definitions to create custom input, we highly recommend that you replace them with ItemSelector.

You can specify the `ItemSelector` field in both an *Inline Map state* and a *Distributed Map state*.

For example, consider the following JSON input that contains an array of three items within the `imageData` node. For each *Map state iteration*, an array item is passed to the iteration as input.

```
[
  {
    "resize": "true",
    "format": "jpg"
  },
  {
    "resize": "false",
    "format": "png"
  },
  {
    "resize": "true",
    "format": "jpg"
  }
]
```

Using the `ItemSelector` field, you can define a custom JSON input to override the original input as shown in the following example. Step Functions then passes this custom input to each *Map state iteration*. The custom input contains a static value for `size` and the value of a Context object data for `Map` state. The `$$.Map.Item.Value` Context object contains the value of each individual data item.

```
{
  "ItemSelector": {
    "size": 10,
    "value.$": "$$.Map.Item.Value"
  }
}
```

The following example shows the input received by one iteration of the *Inline Map state*:

```
{
  "size": 10,
  "value": {
    "resize": "true",
    "format": "jpg"
  }
```

```
}
```

Tip

For a complete example of a *Distributed Map state* that uses the `ItemSelector` field, see Copy large-scale CSV using Distributed Map.

ItemBatcher (Map)

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

The `ItemBatcher` field is a JSON object, which specifies to process a group of items in a single child workflow execution. Use batching when processing large CSV files or JSON arrays, or large sets of Amazon S3 objects.

The following example shows the syntax of the `ItemBatcher` field. In the following syntax, the maximum number of items that each child workflow execution should process is set to 100.

```
{
  "ItemBatcher": {
    "MaxItemsPerBatch": 100
  }
}
```

By default, each item in a dataset is passed as input to individual child workflow executions. For example, assume you specify a JSON file as input that contains the following array:

```
[
  {
    "verdict": "true",
    "statement_date": "6/11/2008",
    "statement_source": "speech"
  },
```

```
  {
    "verdict": "false",
    "statement_date": "6/7/2022",
    "statement_source": "television"
  },
  {
    "verdict": "true",
    "statement_date": "5/18/2016",
    "statement_source": "news"
  },
  ...
]
```

For the given input, each child workflow execution receives an array item as its input. The following example shows the input of a child workflow execution:

```
{
  "verdict": "true",
  "statement_date": "6/11/2008",
  "statement_source": "speech"
}
```

To help optimize the performance and cost of your processing job, select a batch size that balances the number of items against the items processing time. If you use batching, Step Functions adds the items to an **Items** array. It then passes the array as input to each child workflow execution. The following example shows a batch of two items passed as input to a child workflow execution:

```
{
  "Items": [
    {
      "verdict": "true",
      "statement_date": "6/11/2008",
      "statement_source": "speech"
    },
    {
      "verdict": "false",
      "statement_date": "6/7/2022",
      "statement_source": "television"
    }
  ]
}
```

> **Tip**
>
> To learn more about using the `ItemBatcher` field in your workflows, try the following tutorials and workshop:
>
> - Process an entire batch of data within a Lambda function
> - Iterate over items in a batch inside child workflow executions
> - Distributed map and related resources in *The AWS Step Functions Workshop*

Contents

Fields to specify item batching

To batch items, specify the maximum number of items to batch, the maximum batch size, or both. You must specify one of these values to batch items.

Max items per batch

Specifies the maximum number of items that each child workflow execution processes. The interpreter limits the number of items batched in the `Items` array to this value. If you specify both a batch number and size, the interpreter reduces the number of items in a batch to avoid exceeding the specified batch size limit.

If you don't specify this value but provide a value for maximum batch size, Step Functions processes as many items as possible in each child workflow execution without exceeding the maximum batch size in bytes.

For example, imagine you run an execution with an input JSON file that contains 1130 nodes. If you specify a maximum items value for each batch of 100, Step Functions creates 12 batches. Of these, 11 batches contain 100 items each, while the twelfth batch contains the remaining 30 items.

Alternatively, you can specify the maximum items for each batch as a reference path to an existing key-value pair in your *Distributed Map state* input. This path must resolve to a positive integer.

For example, given the following input:

```
{
  "maxBatchItems": 500
}
```

You can specify the maximum number of items to batch using a reference path (**JSONPath only**) as follows:

```
{
  ...
  "Map": {
    "Type": "Map",
    "MaxConcurrency": 2000,
    "ItemBatcher": {
      "MaxItemsPerBatchPath": "$.maxBatchItems"
    }
    ...
    ...
  }
}
```

For **JSONata-based** states, you can also provide a JSONata expression that evaluates to a positive integer.

Important

You can specify either the `MaxItemsPerBatch` or the `MaxItemsPerBatchPath` `(JSONPath only)` sub-field, but not both.

Max KiB per batch

Specifies the maximum size of a batch in bytes, up to 256 KiB. If you specify both a maximum batch number and size, Step Functions reduces the number of items in a batch to avoid exceeding the specified batch size limit.

Alternatively, you can specify the maximum batch size as a reference path to an existing key-value pair in your *Distributed Map state* input. This path must resolve to a positive integer.

> **Note**
>
> If you use batching and don't specify a maximum batch size, the interpreter processes as many items it can process up to 256 KiB in each child workflow execution.

For example, given the following input:

```
{
  "batchSize": 131072
}
```

You can specify the maximum batch size using a reference path as follows:

```
{
  ...
  "Map": {
    "Type": "Map",
    "MaxConcurrency": 2000,
    "ItemBatcher": {
      "MaxInputBytesPerBatchPath": "$.batchSize"
    }
    ...
    ...
  }
}
```

For **JSONata-based** states, you can also provide a JSONata expression that evaluates to a positive integer.

> **Important**
>
> You can specify either the `MaxInputBytesPerBatch` or the `MaxInputBytesPerBatchPath` (JSONPath only) sub-field, but not both.

Batch input

Optionally, you can also specify a fixed JSON input to include in each batch passed to each child workflow execution. Step Functions merges this input with the input for each individual child workflow executions. For example, given the following fixed input of a fact check date on an array of items:

```
"ItemBatcher": {
    "BatchInput": {
        "factCheck": "December 2022"
    }
}
```

Each child workflow execution receives the following as input:

```
{
  "BatchInput": {
    "factCheck": "December 2022"
  },
  "Items": [
    {
      "verdict": "true",
      "statement_date": "6/11/2008",
      "statement_source": "speech"
    },
    {
      "verdict": "false",
      "statement_date": "6/7/2022",
      "statement_source": "television"
    },
    ...
  ]
}
```

For **JSONata-based** states, you can provide JSONata expressions directly to BatchInput, or use JSONata expressions inside JSON objects or arrays.

ResultWriter (Map)

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

The `ResultWriter` field is a JSON object that provides options for the output results of the child workflow executions started by a Distributed Map state. You can specify different formatting options for the output results along with the Amazon S3 location to store them if you choose to export them. Step Functions doesn't export these results by default.

Contents

Contents of the ResultWriter field

The `ResultWriter` field contains the following sub-fields. The choice of fields determines how the output is formatted and whether it's exported to Amazon S3.

`ResultWriter`

A JSON object that specifies the following details:

- `Resource`

  The Amazon S3 API action that Step Functions invokes to export the execution results.

- `Parameters`

  A JSON object that specifies the Amazon S3 bucket name and prefix that stores the execution output.

- `WriterConfig`

  This field enables you to configure the following options.

  - `Transformation`
    - `NONE` - returns the output of the child workflow executions unchanged, in addition to the workflow metadata. Default when exporting the child workflow execution results to Amazon S3 and `WriterConfig` is not specified.
    - `COMPACT` - returns the output of the child workflow executions. Default when `ResultWriter` is not specified.

- `FLATTEN` - returns the output of the child workflow executions. If a child workflow execution returns an array, this option flattens the array, prior to returning the result to a state output or writing the result to an Amazon S3 object.

> **Note**
>
> If a child workflow execution fails, Step Functions returns its execution result unchanged. The results would be equivalent to having set `Transformation` to `NONE`.

- `OutputType`
  - `JSON` - formats the results as a JSON array.
  - `JSONL` - formats the results as JSON Lines.

Required field combinations

The `ResultWriter` field cannot be empty. You must specify one of these sets of sub-fields.

- `WriterConfig` - to preview the formatted output, without saving the results to Amazon S3.
- `Resource` and `Parameters` - to save the results to Amazon S3 without additional formatting.
- All three fields: `WriterConfig`, `Resource` and `Parameters` - to format the output and save it to Amazon S3.

Example configurations and transformation output

The following topics demonstrate the possible configuration settings for `ResultWriter` and examples of processed results from the different transformation options.

- [ResultWriter configurations](#)
- [Transformations](#)

Examples of ResultWriter configurations

The following examples demonstrate configurations with the possible combinations of the three fields: `WriterConfig`, `Resources` and `Parameters`.

Only *WriterConfig*

This example configures how the state output is presented in preview, with the output format and transformation specified in the `WriterConfig` field. Non-existent `Resource` and `Parameters` fields, which would have provided the Amazon S3 bucket specifications, imply the *state output* resource. The results are passed on to the next state.

```
"ResultWriter": {
    "WriterConfig": {
        "Transformation": "FLATTEN",
        "OutputType": "JSON"
    }
}
```

Only *Resources* and *Parameters*

This example exports the state output to the specified Amazon S3 bucket, without the additional formatting and transformation that the non-existent `WriterConfig` field would have specified.

```
"ResultWriter": {
    "Resource": "arn:aws:states:::s3:putObject",
    "Parameters": {
        "Bucket": "amzn-s3-demo-destination-bucket",
        "Prefix": "csvProcessJobs"
    }
```

All three fields: *WriterConfig*, *Resources* and *Parameters*

This example formats the state output according the specifications in the `WriterConfig` field. It also exports it to an Amazon S3 bucket according to the specifications in the `Resource` and `Parameters` fields.

```
"ResultWriter": {
     "WriterConfig": {
        "Transformation": "FLATTEN",
        "OutputType": "JSON"
    },
    "Resource": "arn:aws:states:::s3:putObject",
    "Parameters": {
        "Bucket": "amzn-s3-demo-destination-bucket",
        "Prefix": "csvProcessJobs"
    }
}
```

Examples of transformations

For these examples assume that each child workflow execution returns an output, which is an array of objects.

```
[
  {
    "customer_id": "145538",
    "order_id": "100000"
  },
  {
    "customer_id": "898037",
    "order_id": "100001"
  }
]
```

These examples demonstrate the formatted output for different `Transformation` values, with `OutputType` of `JSON`.

Transformation NONE

This is an example of the processed result when you use the `NONE` transformation. The output is unchanged, and it includes the workflow metadata.

```
[
    {
        "ExecutionArn": "arn:aws:states:region:account-id:execution:orderProcessing/
getOrders:da4e9fc7-abab-3b27-9a77-a277e463b709",
        "Input": ...,
        "InputDetails": {
            "Included": true
        },
        "Name": "da4e9fc7-abab-3b27-9a77-a277e463b709",
        "Output": "[{\"customer_id\":\"145538\",\"order_id\":\"100000\"},{\"customer_id
\":\"898037\",\"order_id\":\"100001\"}]",
        "OutputDetails": {
            "Included": true
        },
        "RedriveCount": 0,
        "RedriveStatus": "NOT_REDRIVABLE",
        "RedriveStatusReason": "Execution is SUCCEEDED and cannot be redriven",
        "StartDate": "2025-02-04T01:49:50.099Z",
```

```
        "StateMachineArn": "arn:aws:states:region:account-
id:stateMachine:orderProcessing/getOrders",
        "Status": "SUCCEEDED",
        "StopDate": "2025-02-04T01:49:50.163Z"
    },
    ...
    {
        "ExecutionArn": "arn:aws:states:region:account-id:execution:orderProcessing/
getOrders:f43a56f7-d21e-3fe9-a40c-9b9b8d0adf5a",
        "Input": ...,
        "InputDetails": {
            "Included": true
        },
        "Name": "f43a56f7-d21e-3fe9-a40c-9b9b8d0adf5a",
        "Output": "[{\"customer_id\":\"169881\",\"order_id\":\"100005\"},{\"customer_id
\":\"797471\",\"order_id\":\"100006\"}]",
        "OutputDetails": {
            "Included": true
        },
        "RedriveCount": 0,
        "RedriveStatus": "NOT_REDRIVABLE",
        "RedriveStatusReason": "Execution is SUCCEEDED and cannot be redriven",
        "StartDate": "2025-02-04T01:49:50.135Z",
        "StateMachineArn": "arn:aws:states:region:account-
id:stateMachine:orderProcessing/getOrders",
        "Status": "SUCCEEDED",
        "StopDate": "2025-02-04T01:49:50.227Z"
    }
]
```

Transformation COMPACT

This is an example of the processed result when you use the `COMPACT` transformation. Note that it's the combined output of the child workflow executions with the original array structure.

```
[
    [
        {
            "customer_id": "145538",
            "order_id": "100000"
        },
        {
            "customer_id": "898037",
```

```
            "order_id": "100001"
        }
    ],
    ...,

    [
        {
            "customer_id": "169881",
            "order_id": "100005"
        },
        {
            "customer_id": "797471",
            "order_id": "100006"
        }
    ]
]
```

Transformation FLATTEN

This is an example of the processed result when you use the `FLATTEN` transformation. Note that it's the combined output of the child workflow executions arrays flattened into one array.

```
[
    {
        "customer_id": "145538",
        "order_id": "100000"
    },
    {
        "customer_id": "898037",
        "order_id": "100001"
    },
    ...
    {
        "customer_id": "169881",
        "order_id": "100005"
    },
    {
        "customer_id": "797471",
        "order_id": "100006"
    }
]
```

Exporting to Amazon S3

> **Important**
>
> Make sure that the Amazon S3 bucket you use to export the results of a Map Run is under the same AWS account and AWS Region as your state machine. Otherwise, your state machine execution will fail with the `States.ResultWriterFailed` error.

Exporting the results to an Amazon S3 bucket is helpful if your output payload size exceeds 256 KiB. Step Functions consolidates all child workflow execution data, such as execution input and output, ARN, and execution status. It then exports executions with the same status to their respective files in the specified Amazon S3 location.

The following example, using **JSONPath**, shows the syntax of the `ResultWriter` field with `Parameters` to export the child workflow execution results. In this example, you store the results in a bucket named `amzn-s3-demo-destination-bucket` within a prefix called `csvProcessJobs`.

```
{
  "ResultWriter": {
    "Resource": "arn:aws:states:::s3:putObject",
    "Parameters": {
      "Bucket": "amzn-s3-demo-destination-bucket",
      "Prefix": "csvProcessJobs"
    }
  }
}
```

For **JSONata** states, `Parameters` will be replaced with `Arguments`.

```
{
  "ResultWriter": {
    "Resource": "arn:aws:states:::s3:putObject",
    "Arguments": {
      "Bucket": "amzn-s3-demo-destination-bucket",
      "Prefix": "csvProcessJobs"
    }
  }
}
```

> **Tip**
>
> In Workflow Studio, you can export the child workflow execution results by selecting **Export Map state results to Amazon S3**. Then, provide the name of the Amazon S3 bucket and prefix where you want to export the results to.

Step Functions needs appropriate permissions to access the bucket and folder where you want to export the results. For information about the required IAM policy, see IAM policies for ResultWriter.

If you export the child workflow execution results, the *Distributed Map state* execution returns the Map Run ARN and data about the Amazon S3 export location in the following format:

```
{
  "MapRunArn": "arn:aws:states:us-east-2:account-
id:mapRun:csvProcess/Map:ad9b5f27-090b-3ac6-9beb-243cd77144a7",
  "ResultWriterDetails": {
    "Bucket": "amzn-s3-demo-destination-bucket",
    "Key": "csvProcessJobs/ad9b5f27-090b-3ac6-9beb-243cd77144a7/manifest.json"
  }
}
```

Step Functions exports executions with the same status to their respective files. For example, if your child workflow executions resulted in 500 success and 200 failure results, Step Functions creates two files in the specified Amazon S3 location for the success and failure results. In this example, the success results file contains the 500 success results, while the failure results file contains the 200 failure results.

For a given execution attempt, Step Functions creates the following files in the specified Amazon S3 location depending on your execution output:

- `manifest.json` – Contains Map Run metadata, such as export location, Map Run ARN, and information about the result files.

  If you've redriven a Map Run, the `manifest.json` file, contains references to all the successful child workflow executions across all the attempts of a Map Run. However, this file contains references to the failed and pending executions for a specific redrive.

- `SUCCEEDED_n.json` – Contains the consolidated data for all successful child workflow executions. *n* represents the index number of the file. The index number starts from 0. For example, `SUCCEEDED_1.json`.
- `FAILED_n.json` – Contains the consolidated data for all failed, timed out, and aborted child workflow executions. Use this file to recover from failed executions. *n* represents the index of the file. The index number starts from 0. For example, `FAILED_1.json`.
- `PENDING_n.json` – Contains the consolidated data for all child workflow executions that weren't started because the Map Run failed or aborted. *n* represents the index of the file. The index number starts from 0. For example, `PENDING_1.json`.

Step Functions supports individual result files of up to 5 GB. If a file size exceeds 5 GB, Step Functions creates another file to write the remaining execution results and appends an index number to the file name. For example, if size of the `SUCCEEDED_0.json` file exceeds 5 GB, Step Functions creates `SUCCEEDED_1.json` file to record the remaining results.

If you didn't specify to export the child workflow execution results, the state machine execution returns an array of child workflow execution results as shown in the following example:

```
[
  {
    "statusCode": 200,
    "inputReceived": {
      "show_id": "s1",
      "release_year": "2020",
      "rating": "PG-13",
      "type": "Movie"
    }
  },
  {
    "statusCode": 200,
    "inputReceived": {
      "show_id": "s2",
      "release_year": "2021",
      "rating": "TV-MA",
      "type": "TV Show"
    }
  },
  ...
]
```

> **Note**
>
> If the returned output size exceeds 256 KiB, the state machine execution fails and returns a `States.DataLimitExceeded` error.

IAM policies for ResultWriter

When you create workflows with the Step Functions console, Step Functions can automatically generate IAM policies based on the resources in your workflow definition. These policies include the least privileges necessary to allow the state machine role to invoke the `StartExecution` API action for the *Distributed Map state*. These policies also include the least privileges necessary Step Functions to access AWS resources, such as Amazon S3 buckets and objects and Lambda functions. We highly recommend that you include only those permissions that are necessary in your IAM policies. For example, if your workflow includes a `Map` state in Distributed mode, scope your policies down to the specific Amazon S3 bucket and folder that contains your dataset.

> **Important**
>
> If you specify an Amazon S3 bucket and object, or prefix, with a reference path to an existing key-value pair in your *Distributed Map state* input, make sure that you update the IAM policies for your workflow. Scope the policies down to the bucket and object names the path resolves to at runtime.

The following IAM policy example grants the least privileges required to write your child workflow execution results to a folder named *`csvJobs`* in an Amazon S3 bucket using the `PutObject` API action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:ListMultipartUploadParts",
                "s3:AbortMultipartUpload"
```

```
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-destination-bucket/csvJobs/*"
            ]
        }
    ]
}
```

If the Amazon S3 bucket to which you're writing the child workflow execution result is encrypted using an AWS Key Management Service (AWS KMS) key, you must include the necessary AWS KMS permissions in your IAM policy. For more information, see IAM permissions for AWS KMS key encrypted Amazon S3 bucket.

How Step Functions parses input CSV files

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

Step Functions parses text delimited files based on the following rules:

- The delimiter that separates fields is specified by `CSVDelimiter` in *ReaderConfig*. The delimiter defaults to `COMMA`.
- Newlines are a delimiter that separates **records**.
- Fields are treated as strings. For data type conversions, use the `States.StringToJson` intrinsic function in ItemSelector (Map).
- Double quotation marks (" ") are not required to enclose strings. However, strings that are enclosed by double quotation marks can contain commas and newlines without acting as record delimiters.
- You can preserve double quotes by repeating them.
- If the number of fields in a row is **less** than the number of fields in the header, Step Functions provides **empty strings** for the missing values.
- If the number of fields in a row is **more** than the number of fields in the header, Step Functions **skips** the additional fields.

Example of parsing an input CSV file

Say that you have provided a CSV file named *myCSVInput.csv* that contains one row as input. Then, you've stored this file in an Amazon S3 bucket that's named *amzn-s3-demo-bucket*. The CSV file is as follows.

```
abc,123,"This string contains commas, a double quotation marks (""), and a newline (
)",{""MyKey"":""MyValue""},"[1,2,3]"
```

The following state machine reads this CSV file and uses [ItemSelector (Map)](#) to convert the data types of some of the fields.

```
{
  "StartAt": "Map",
  "States": {
    "Map": {
      "Type": "Map",
      "ItemProcessor": {
        "ProcessorConfig": {
          "Mode": "DISTRIBUTED",
          "ExecutionType": "STANDARD"
        },
        "StartAt": "Pass",
        "States": {
          "Pass": {
            "Type": "Pass",
            "End": true
          }
        }
      },
      "End": true,
      "Label": "Map",
      "MaxConcurrency": 1000,
      "ItemReader": {
        "Resource": "arn:aws:states:::s3:getObject",
        "ReaderConfig": {
          "InputType": "CSV",
          "CSVHeaderLocation": "GIVEN",
          "CSVHeaders": [
            "MyLetters",
            "MyNumbers",
            "MyString",
            "MyObject",
```

```
            "MyArray"
          ]
        },
        "Parameters": {
          "Bucket": "amzn-s3-demo-bucket",
          "Key": "myCSVInput.csv"
        }
      },
      "ItemSelector": {
        "MyLetters.$": "$$.Map.Item.Value.MyLetters",
        "MyNumbers.$": "States.StringToJson($$.Map.Item.Value.MyNumbers)",
        "MyString.$": "$$.Map.Item.Value.MyString",
        "MyObject.$": "States.StringToJson($$.Map.Item.Value.MyObject)",
        "MyArray.$": "States.StringToJson($$.Map.Item.Value.MyArray)"
      }
    }
  }
}
```

When you run this state machine, it produces the following output.

```
[
  {
    "MyNumbers": 123,
    "MyObject": {
      "MyKey": "MyValue"
    },
    "MyString": "This string contains commas, a double quote (\"), and a newline (\n)",
    "MyLetters": "abc",
    "MyArray": [
      1,
      2,
      3
    ]
  }
]
```

Integrating services with Step Functions

Learn how to integrate AWS services and call HTTPS APIs with Step Functions. With service integrations, your workflows can coordinate resources and orchestrate your business processes.

Depending on workflow type and availability, your workflows call services using one of three service integration patterns:

- Request a Response (default) - wait for HTTP response, then go to the next state
- Run a Job (`.sync`) - wait for the job to complete
- Wait for Callback (`.waitForTaskToken`) - pause a workflow until a task token is returned

To learn more, see Service integration patterns. And to learn more about controlling the flow of data to your integrated services, see the section called “Pass parameters”.

AWS SDK integrations

AWS SDK integrations work exactly like an API call using the AWS SDK.

Using AWS SDK integrations, your state machines can call over nine thousand API actions for over two hundred AWS services.

Example integrations you might use:

- Invoke a AWS Lambda function.
- Run an AWS Batch job and take different actions based on the results.
- Retrieve or updated items in 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 AI.
- Build workflows for executing Amazon EMR jobs.
- Launch another AWS Step Functions workflow execution.

Optimized integrations

In addition to standard integrations, Step Functions provides optimized integrations which provide enhanced functionality. Optimized integrations have been customized by Step Functions to provide an improved developer experience when integrating the service in a workflow context.

For example, the optimized Lambda `Invoke` automatically converts API output from escaped JSON to a JSON object which you can more easily use. Another example is how AWS `BatchSubmitJob` can pause execution until the batch job completes, which is a common scenario.

When possible, we **recommend** using the optimized integrations.

For the full list of optimized integrations, see the dedicated chapter for *Integrating optimized services*

Call HTTPS APIs

An HTTP Task is a type of Task workflow state state that you can use to call HTTPS APIs in your workflows. The API can be public, such as third-party SaaS applications like Stripe or Salesforce. You can also call private API, such as HTTPS-based applications in an Amazon Virtual Private Cloud.

For more information, see the section called “Call HTTPS APIs”.

Learning to use AWS service SDK integrations in Step Functions

With Step Functions' AWS SDK integration, your workflows can call almost any AWS service's API actions. The services or SDKs that are not available might be recently released, require customized configuration, or are not suitable for use in a workflow, such as SDKs for streaming audio or video.

Topics

- Using AWS SDK service integrations
- Supported AWS SDK service integrations
- Deprecated AWS SDK service integrations

Using AWS SDK service integrations

To use AWS SDK integrations, you specify the service name and API call and, optionally, a service integration pattern (For more information, see Service integration patterns.

> **Note**
>
> - Parameters in Step Functions are expressed in PascalCase, even if the native service API is in camelCase. For example, you could use the Step Functions API action `startSyncExecution` and specify its parameter as `StateMachineArn`.
> - For API actions that accept enumerated parameters, such as the `DescribeLaunchTemplateVersions` API action for Amazon EC2, specify a plural version of the parameter name. For example, specify `Filters` for the `Filter.N` parameter of the `DescribeLaunchTemplateVersions` API action.

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, you might use `arn:aws:states:::aws-sdk:ec2:describeInstances` to 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: *`ServiceName.ErrorName`*. Both the service name and error name are in Pascal case. You can also see the service name in the Task state's Resource field in lowercase. The target service's API reference documentation lists the potential error names.

Consider an example where you might use the AWS SDK integration `arn:aws:states:::aws-sdk:acmpca:deleteCertificateAuthority`. The AWS Private Certificate Authority API Reference indicates that the `DeleteCertificateAuthority` API action can result in an error named `ResourceNotFoundException`. To handle this error you would specify the Error `AcmPca.ResourceNotFoundException` in your Task state's Retriers or Catchers.

> **Note**
>
> Some AWS services don't include the *Exception* suffix in the The API Reference documentation. Despite this alternate naming convention, always include the *Exception* suffix for the potential error names in your AWS Step Functions integration. Do so even when the suffix is not already part of the error name provided by the service.

Consider another error name, this time the CreateBucket action. The *Amazon Simple Storage Service API Reference* lists the error `BucketAlreadyExists`. Note that it doesn't have the *Exception* suffix. To handle this error in Step Functions, refer to it as `S3.BucketAlreadyExists`**`Exception`**. The S3 service error naming convention differs from the errors in the previously mentioned AWS Private Certificate Authority API Reference. Regardless, in both cases you must include the *Exception* suffix for potential errors in the Step Functions integration.

For more information about error handling, see Handling errors in Step Functions workflows.

Step Functions cannot 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 How Step Functions generates IAM policies for integrated services for more information.

See the Gather Amazon S3 bucket info using AWS SDK service integrations tutorial for an example of how to use AWS SDK integrations.

Supported AWS SDK service integrations

The *Task state resource* (Resource) shows the syntax to call a specific API action for the service. The *Exception prefix* is present in the exceptions that are generated when you erroneously perform an AWS SDK service integration with Step Functions.

> **Important**
>
> New actions and updates to already supported actions, such as new parameters, will not be immediately available after service SDK updates.

Amazon A2I

Task state resource: `arn:aws:states:::aws-sdk:sagemakera2iruntime:`*`[apiAction]`*

Exception prefix: `SageMakerA2IRuntime`

API Gateway V1

Task state resource: `arn:aws:states:::aws-sdk:apigateway:`*`[apiAction]`*

Exception prefix: `ApiGateway`

API Gateway V2

Task state resource: `arn:aws:states:::aws-sdk:apigatewayv2:`*`[apiAction]`*

Exception prefix: `ApiGatewayV2`

AWS Account Management

Task state resource: `arn:aws:states:::aws-sdk:account:`*`[apiAction]`*

Exception prefix: `Account`

AWS Amplify

Task state resource: `arn:aws:states:::aws-sdk:amplify:`*`[apiAction]`*

Exception prefix: `Amplify`

Amplify Backend

Task state resource: `arn:aws:states:::aws-sdk:amplifybackend:`*`[apiAction]`*

Exception prefix: `AmplifyBackend`

Amplify UI Builder

Task state resource: `arn:aws:states:::aws-sdk:amplifyuibuilder:`*`[apiAction]`*

Exception prefix: `AmplifyUiBuilder`

AWS App Mesh

Task state resource: `arn:aws:states:::aws-sdk:appmesh:`*`[apiAction]`*

Exception prefix: `AppMesh`

AWS App Runner

Task state resource: `arn:aws:states:::aws-sdk:apprunner:`*`[apiAction]`*

Exception prefix: `AppRunner`

AWS AppConfig

Task state resource: `arn:aws:states:::aws-sdk:appconfig:`*`[apiAction]`*

Exception prefix: `AppConfig`

AWS AppConfig Data

Task state resource: `arn:aws:states:::aws-sdk:appconfigdata:`*`[apiAction]`*

Exception prefix: `AppConfigData`

AWS AppFabric

Task state resource: `arn:aws:states:::aws-sdk:appfabric:[apiAction]`

Exception prefix: `AppFabric`

AppIntegrations

Task state resource: `arn:aws:states:::aws-sdk:appintegrations:[apiAction]`

Exception prefix: `AppIntegrations`

Amazon AppStream

Task state resource: `arn:aws:states:::aws-sdk:appstream:[apiAction]`

Exception prefix: `AppStream`

AWS AppSync

Task state resource: `arn:aws:states:::aws-sdk:appsync:[apiAction]`

Exception prefix: `AppSync`

Amazon Appflow

Task state resource: `arn:aws:states:::aws-sdk:appflow:[apiAction]`

Exception prefix: `Appflow`

Application Auto Scaling

Task state resource: `arn:aws:states:::aws-sdk:applicationautoscaling:[apiAction]`

Exception prefix: `ApplicationAutoScaling`

Application Cost Profiler

Task state resource: `arn:aws:states:::aws-sdk:applicationcostprofiler:[apiAction]`

Exception prefix: `ApplicationCostProfiler`

Application Discovery Service

Task state resource: `arn:aws:states:::aws-sdk:applicationdiscovery:[apiAction]`

Exception prefix: `ApplicationDiscovery`

Unsupported operations: `DescribeExportConfigurations`, `ExportConfigurations`

Application Migration Service

Task state resource: `arn:aws:states:::aws-sdk:mgn:`*`[apiAction]`*

Exception prefix: `Mgn`

Amazon Athena

Task state resource: `arn:aws:states:::aws-sdk:athena:`*`[apiAction]`*

Exception prefix: `Athena`

Audit Manager

Task state resource: `arn:aws:states:::aws-sdk:auditmanager:`*`[apiAction]`*

Exception prefix: `AuditManager`

Amazon Aurora DSQL

Task state resource: `arn:aws:states:::aws-sdk:dsql:`*`[apiAction]`*

Exception prefix: `Dsql`

AWS Auto Scaling

Task state resource: `arn:aws:states:::aws-sdk:autoscalingplans:`*`[apiAction]`*

Exception prefix: `AutoScalingPlans`

B2B Data Interchange

Task state resource: `arn:aws:states:::aws-sdk:b2bi:`*`[apiAction]`*

Exception prefix: `B2Bi`

AWS Backup

Task state resource: `arn:aws:states:::aws-sdk:backup:`*`[apiAction]`*

Exception prefix: `Backup`

AWS Backup Gateway

Task state resource: `arn:aws:states:::aws-sdk:backupgateway:`*`[apiAction]`*

Exception prefix: `BackupGateway`

AWS Backup Search

Task state resource: `arn:aws:states:::aws-sdk:backupsearch:`*`[apiAction]`*

Exception prefix: `BackupSearch`

AWS Batch

Task state resource: `arn:aws:states:::aws-sdk:batch:`*`[apiAction]`*

Exception prefix: `Batch`

Amazon Bedrock

Task state resource: `arn:aws:states:::aws-sdk:bedrock:`*`[apiAction]`*

Exception prefix: `Bedrock`

Amazon Bedrock Agents

Task state resource: `arn:aws:states:::aws-sdk:bedrockagent:`*`[apiAction]`*

Exception prefix: `BedrockAgent`

Amazon Bedrock Runtime

Task state resource: `arn:aws:states:::aws-sdk:bedrockruntime:`*`[apiAction]`*

Exception prefix: `BedrockRuntime`

Unsupported operations: `InvokeModelWithResponseStream`, `ConverseStream`

Amazon Bedrock Runtime Agents

Task state resource: `arn:aws:states:::aws-sdk:bedrockagentruntime:`*`[apiAction]`*

Exception prefix: `BedrockAgentRuntime`

Unsupported operations: `InvokeAgent`, `InvokeFlow`, `InvokeInlineAgent`, `OptimizePrompt`, `RetrieveAndGenerateStream`

AWS Billing

Task state resource: `arn:aws:states:::aws-sdk:billing:`*`[apiAction]`*

Exception prefix: `Billing`

AWS Billing Conductor

Task state resource: `arn:aws:states:::aws-sdk:billingconductor:`*[apiAction]*

Exception prefix: `Billingconductor`

AWS Billing and Cost Management Pricing Calculator

Task state resource: `arn:aws:states:::aws-sdk:bcmpricingcalculator:`*[apiAction]*

Exception prefix: `BcmPricingCalculator`

Amazon Braket

Task state resource: `arn:aws:states:::aws-sdk:braket:`*[apiAction]*

Exception prefix: `Braket`

AWS Budgets

Task state resource: `arn:aws:states:::aws-sdk:budgets:`*[apiAction]*

Exception prefix: `Budgets`

Certificate Manager

Task state resource: `arn:aws:states:::aws-sdk:acm:`*[apiAction]*

Exception prefix: `Acm`

Certificate Manager PCA

Task state resource: `arn:aws:states:::aws-sdk:acmpca:`*[apiAction]*

Exception prefix: `AcmPca`

Amazon Chime

Task state resource: `arn:aws:states:::aws-sdk:chime:`*[apiAction]*

Exception prefix: `Chime`

Amazon Chime Identity

Task state resource: `arn:aws:states:::aws-sdk:chimesdkidentity:`*[apiAction]*

Exception prefix: `ChimeSdkIdentity`

Amazon Chime Media Pipelines

Task state resource: `arn:aws:states:::aws-sdk:chimesdkmediapipelines:[apiAction]`

Exception prefix: `ChimeSdkMediaPipelines`

Amazon Chime Meetings

Task state resource: `arn:aws:states:::aws-sdk:chimesdkmeetings:[apiAction]`

Exception prefix: `ChimeSdkMeetings`

Amazon Chime Messaging

Task state resource: `arn:aws:states:::aws-sdk:chimesdkmessaging:[apiAction]`

Exception prefix: `ChimeSdkMessaging`

Amazon Chime Voice

Task state resource: `arn:aws:states:::aws-sdk:chimesdkvoice:[apiAction]`

Exception prefix: `ChimeSdkVoice`

AWS Clean Rooms

Task state resource: `arn:aws:states:::aws-sdk:cleanrooms:[apiAction]`

Exception prefix: `CleanRooms`

AWS Clean Rooms ML

Task state resource: `arn:aws:states:::aws-sdk:cleanroomsml:[apiAction]`

Exception prefix: `CleanRoomsMl`

AWS Cloud Control

Task state resource: `arn:aws:states:::aws-sdk:cloudcontrol:[apiAction]`

Exception prefix: `CloudControl`

Cloud Directory

Task state resource: `arn:aws:states:::aws-sdk:clouddirectory:[apiAction]`

Exception prefix: `CloudDirectory`

AWS Cloud Map

Task state resource: `arn:aws:states:::aws-sdk:servicediscovery:`*`[apiAction]`*

Exception prefix: `ServiceDiscovery`

AWS Cloud9

Task state resource: `arn:aws:states:::aws-sdk:cloud9:`*`[apiAction]`*

Exception prefix: `Cloud9`

CloudFormation

Task state resource: `arn:aws:states:::aws-sdk:cloudformation:`*`[apiAction]`*

Exception prefix: `CloudFormation`

CloudFront

Task state resource: `arn:aws:states:::aws-sdk:cloudfront:`*`[apiAction]`*

Exception prefix: `CloudFront`

Amazon CloudFront KeyValueStore

Task state resource: `arn:aws:states:::aws-sdk:cloudfrontkeyvaluestore:`*`[apiAction]`*

Exception prefix: `CloudFrontKeyValueStore`

CloudHSM V1

Task state resource: `arn:aws:states:::aws-sdk:cloudhsm:`*`[apiAction]`*

Exception prefix: `CloudHsm`

CloudHSM V2

Task state resource: `arn:aws:states:::aws-sdk:cloudhsmv2:`*`[apiAction]`*

Exception prefix: `CloudHsmV2`

CloudSearch

Task state resource: `arn:aws:states:::aws-sdk:cloudsearch:`*`[apiAction]`*

Exception prefix: `CloudSearch`

CloudTrail

Task state resource: `arn:aws:states:::aws-sdk:cloudtrail:`*`[apiAction]`*

Exception prefix: `CloudTrail`

CloudTrail Data

Task state resource: `arn:aws:states:::aws-sdk:cloudtraildata:`*`[apiAction]`*

Exception prefix: `CloudTrailData`

CloudWatch

Task state resource: `arn:aws:states:::aws-sdk:cloudwatch:`*`[apiAction]`*

Exception prefix: `CloudWatch`

CloudWatch Application Insights

Task state resource: `arn:aws:states:::aws-sdk:applicationinsights:`*`[apiAction]`*

Exception prefix: `ApplicationInsights`

Amazon CloudWatch Application Signals

Task state resource: `arn:aws:states:::aws-sdk:applicationsignals:`*`[apiAction]`*

Exception prefix: `ApplicationSignals`

CloudWatch Internet Monitor

Task state resource: `arn:aws:states:::aws-sdk:internetmonitor:`*`[apiAction]`*

Exception prefix: `InternetMonitor`

CloudWatch Logs

Task state resource: `arn:aws:states:::aws-sdk:cloudwatchlogs:`*`[apiAction]`*

Exception prefix: `CloudWatchLogs`

Unsupported operations: `StartLiveTail`

CloudWatch Observability Access Manager

Task state resource: `arn:aws:states:::aws-sdk:oam:`*`[apiAction]`*

Exception prefix: `Oam`

CloudWatch Observability Admin Service

Task state resource: `arn:aws:states:::aws-sdk:observabilityadmin:`*`[apiAction]`*

Exception prefix: `ObservabilityAdmin`

CloudWatch RUM

Task state resource: `arn:aws:states:::aws-sdk:rum:`*`[apiAction]`*

Exception prefix: `Rum`

CloudWatch Synthetics

Task state resource: `arn:aws:states:::aws-sdk:synthetics:`*`[apiAction]`*

Exception prefix: `Synthetics`

CodeArtifact

Task state resource: `arn:aws:states:::aws-sdk:codeartifact:`*`[apiAction]`*

Exception prefix: `Codeartifact`

CodeBuild

Task state resource: `arn:aws:states:::aws-sdk:codebuild:`*`[apiAction]`*

Exception prefix: `CodeBuild`

Amazon CodeCatalyst

Task state resource: `arn:aws:states:::aws-sdk:codecatalyst:`*`[apiAction]`*

Exception prefix: `CodeCatalyst`

CodeCommit

Task state resource: `arn:aws:states:::aws-sdk:codecommit:`*`[apiAction]`*

Exception prefix: `CodeCommit`

AWS CodeConnections

Task state resource: `arn:aws:states:::aws-sdk:codeconnections:`*`[apiAction]`*

Exception prefix: `CodeConnections`

CodeDeploy

Task state resource: `arn:aws:states:::aws-sdk:codedeploy:`*`[apiAction]`*

Exception prefix: `CodeDeploy`

Unsupported operations: `BatchGetDeploymentInstances`, `GetDeploymentInstance`, `ListDeploymentInstances`, `SkipWaitTimeForInstanceTermination`

CodeGuru Profiler

Task state resource: `arn:aws:states:::aws-sdk:codeguruprofiler:`*`[apiAction]`*

Exception prefix: `CodeGuruProfiler`

CodeGuru Reviewer

Task state resource: `arn:aws:states:::aws-sdk:codegurureviewer:`*`[apiAction]`*

Exception prefix: `CodeGuruReviewer`

CodeGuru Security

Task state resource: `arn:aws:states:::aws-sdk:codegurusecurity:`*`[apiAction]`*

Exception prefix: `CodeGuruSecurity`

CodePipeline

Task state resource: `arn:aws:states:::aws-sdk:codepipeline:`*`[apiAction]`*

Exception prefix: `CodePipeline`

AWS CodeStar Connections

Task state resource: `arn:aws:states:::aws-sdk:codestarconnections:`*`[apiAction]`*

Exception prefix: `CodeStarConnections`

AWS CodeStar Notifications

Task state resource: `arn:aws:states:::aws-sdk:codestarnotifications:`*`[apiAction]`*

Exception prefix: `CodestarNotifications`

Cognito Identity Pools

Task state resource: `arn:aws:states:::aws-sdk:cognitoidentity:[apiAction]`

Exception prefix: `CognitoIdentity`

Cognito Sync

Task state resource: `arn:aws:states:::aws-sdk:cognitosync:[apiAction]`

Exception prefix: `CognitoSync`

Cognito User Pools

Task state resource: `arn:aws:states:::aws-sdk:cognitoidentityprovider:[apiAction]`

Exception prefix: `CognitoIdentityProvider`

Amazon Comprehend

Task state resource: `arn:aws:states:::aws-sdk:comprehend:[apiAction]`

Exception prefix: `Comprehend`

Amazon Comprehend Medical

Task state resource: `arn:aws:states:::aws-sdk:comprehendmedical:[apiAction]`

Exception prefix: `ComprehendMedical`

Unsupported operations: `DetectEntities`

Compute Optimizer

Task state resource: `arn:aws:states:::aws-sdk:computeoptimizer:[apiAction]`

Exception prefix: `ComputeOptimizer`

AWS Config

Task state resource: `arn:aws:states:::aws-sdk:config:[apiAction]`

Exception prefix: `Config`

Amazon Connect

Task state resource: `arn:aws:states:::aws-sdk:connect:[apiAction]`

Exception prefix: `Connect`

Amazon Connect Campaigns

Task state resource: `arn:aws:states:::aws-sdk:connectcampaigns:`*`[apiAction]`*

Exception prefix: `ConnectCampaigns`

Amazon Connect Campaigns V2

Task state resource: `arn:aws:states:::aws-sdk:connectcampaignsv2:`*`[apiAction]`*

Exception prefix: `ConnectCampaignsV2`

Amazon Connect Cases

Task state resource: `arn:aws:states:::aws-sdk:connectcases:`*`[apiAction]`*

Exception prefix: `ConnectCases`

Amazon Connect Contact Lens

Task state resource: `arn:aws:states:::aws-sdk:connectcontactlens:`*`[apiAction]`*

Exception prefix: `ConnectContactLens`

Amazon Connect Customer Profiles

Task state resource: `arn:aws:states:::aws-sdk:customerprofiles:`*`[apiAction]`*

Exception prefix: `CustomerProfiles`

Amazon Connect Participant

Task state resource: `arn:aws:states:::aws-sdk:connectparticipant:`*`[apiAction]`*

Exception prefix: `ConnectParticipant`

Amazon Connect Voice ID

Task state resource: `arn:aws:states:::aws-sdk:voiceid:`*`[apiAction]`*

Exception prefix: `VoiceId`

Amazon Connect Wisdom

Task state resource: `arn:aws:states:::aws-sdk:wisdom:`*`[apiAction]`*

Exception prefix: `Wisdom`

AWS Control Catalog

Task state resource: `arn:aws:states:::aws-sdk:controlcatalog:[apiAction]`

Exception prefix: `ControlCatalog`

AWS Control Tower

Task state resource: `arn:aws:states:::aws-sdk:controltower:[apiAction]`

Exception prefix: `ControlTower`

AWS Cost Explorer

Task state resource: `arn:aws:states:::aws-sdk:costexplorer:[apiAction]`

Exception prefix: `CostExplorer`

Cost Optimization Hub

Task state resource: `arn:aws:states:::aws-sdk:costoptimizationhub:[apiAction]`

Exception prefix: `CostOptimizationHub`

AWS Cost and Usage Report

Task state resource: `arn:aws:states:::aws-sdk:costandusagereport:[apiAction]`

Exception prefix: `CostAndUsageReport`

Data Automation for Amazon Bedrock

Task state resource: `arn:aws:states:::aws-sdk:bedrockdataautomation:[apiAction]`

Exception prefix: `BedrockDataAutomation`

AWS Data Exchange

Task state resource: `arn:aws:states:::aws-sdk:dataexchange:[apiAction]`

Exception prefix: `DataExchange`

Unsupported operations: `SendApiAsset`

AWS Data Exports

Task state resource: `arn:aws:states:::aws-sdk:bcmdataexports:[apiAction]`

Exception prefix: `BcmDataExports`

Amazon Data Lifecycle Manager

Task state resource: `arn:aws:states:::aws-sdk:dlm:`*`[apiAction]`*

Exception prefix: `Dlm`

Data Pipeline

Task state resource: `arn:aws:states:::aws-sdk:datapipeline:`*`[apiAction]`*

Exception prefix: `DataPipeline`

DataSync

Task state resource: `arn:aws:states:::aws-sdk:datasync:`*`[apiAction]`*

Exception prefix: `DataSync`

Amazon DataZone

Task state resource: `arn:aws:states:::aws-sdk:datazone:`*`[apiAction]`*

Exception prefix: `DataZone`

AWS Database Migration Service

Task state resource: `arn:aws:states:::aws-sdk:databasemigration:`*`[apiAction]`*

Exception prefix: `DatabaseMigration`

AWS Deadline Cloud

Task state resource: `arn:aws:states:::aws-sdk:deadline:`*`[apiAction]`*

Exception prefix: `Deadline`

Detective

Task state resource: `arn:aws:states:::aws-sdk:detective:`*`[apiAction]`*

Exception prefix: `Detective`

DevOps Guru

Task state resource: `arn:aws:states:::aws-sdk:devopsguru:`*`[apiAction]`*

Exception prefix: `DevOpsGuru`

Device Farm

Task state resource: `arn:aws:states:::aws-sdk:devicefarm:`*`[apiAction]`*

Exception prefix: `DeviceFarm`

Direct Connect

Task state resource: `arn:aws:states:::aws-sdk:directconnect:`*`[apiAction]`*

Exception prefix: `DirectConnect`

Unsupported operations: `AllocateConnectionOnInterconnect`, `DescribeConnectionLoa`, `DescribeConnectionsOnInterconnect`, `DescribeInterconnectLoa`

Directory Service

Task state resource: `arn:aws:states:::aws-sdk:directory:`*`[apiAction]`*

Exception prefix: `Directory`

AWS Directory Service Data

Task state resource: `arn:aws:states:::aws-sdk:directoryservicedata:`*`[apiAction]`*

Exception prefix: `DirectoryServiceData`

Amazon DocumentDB

Task state resource: `arn:aws:states:::aws-sdk:docdb:`*`[apiAction]`*

Exception prefix: `DocDb`

Amazon DocumentDB Elastic Clusters

Task state resource: `arn:aws:states:::aws-sdk:docdbelastic:`*`[apiAction]`*

Exception prefix: `DocDbElastic`

DynamoDB

Task state resource: `arn:aws:states:::aws-sdk:dynamodb:`*`[apiAction]`*

Exception prefix: `DynamoDb`

DynamoDB Accelerator

Task state resource: `arn:aws:states:::aws-sdk:dax:`*`[apiAction]`*

Exception prefix: `Dax`

DynamoDB Streams

Task state resource: `arn:aws:states:::aws-sdk:dynamodbstreams:[apiAction]`

Exception prefix: `DynamoDbStreams`

Amazon EBS

Task state resource: `arn:aws:states:::aws-sdk:ebs:[apiAction]`

Exception prefix: `Ebs`

Amazon EC2

Task state resource: `arn:aws:states:::aws-sdk:ec2:[apiAction]`

Exception prefix: `Ec2`

EC2 Auto Scaling

Task state resource: `arn:aws:states:::aws-sdk:autoscaling:[apiAction]`

Exception prefix: `AutoScaling`

EC2 Image Builder

Task state resource: `arn:aws:states:::aws-sdk:imagebuilder:[apiAction]`

Exception prefix: `Imagebuilder`

AWS EC2 Instance Connect

Task state resource: `arn:aws:states:::aws-sdk:ec2instanceconnect:[apiAction]`

Exception prefix: `Ec2InstanceConnect`

Amazon ECR

Task state resource: `arn:aws:states:::aws-sdk:ecr:[apiAction]`

Exception prefix: `Ecr`

Amazon ECR Public

Task state resource: `arn:aws:states:::aws-sdk:ecrpublic:[apiAction]`

Exception prefix: `EcrPublic`

Amazon ECS

Task state resource: `arn:aws:states:::aws-sdk:ecs:`*`[apiAction]`*

Exception prefix: `Ecs`

Amazon EFS

Task state resource: `arn:aws:states:::aws-sdk:efs:`*`[apiAction]`*

Exception prefix: `Efs`

Unsupported operations: `CreateTags`

Amazon EKS

Task state resource: `arn:aws:states:::aws-sdk:eks:`*`[apiAction]`*

Exception prefix: `Eks`

Amazon EKS Auth

Task state resource: `arn:aws:states:::aws-sdk:eksauth:`*`[apiAction]`*

Exception prefix: `EksAuth`

Amazon EMR

Task state resource: `arn:aws:states:::aws-sdk:emr:`*`[apiAction]`*

Exception prefix: `Emr`

Unsupported operations: `DescribeJobFlows`

Amazon EMR Containers

Task state resource: `arn:aws:states:::aws-sdk:emrcontainers:`*`[apiAction]`*

Exception prefix: `EmrContainers`

Amazon EMR Serverless

Task state resource: `arn:aws:states:::aws-sdk:emrserverless:`*`[apiAction]`*

Exception prefix: `EmrServerless`

ElastiCache

Task state resource: `arn:aws:states:::aws-sdk:elasticache:`*`[apiAction]`*

Exception prefix: `ElastiCache`

Elastic Beanstalk

Task state resource: `arn:aws:states:::aws-sdk:elasticbeanstalk:`*`[apiAction]`*

Exception prefix: `ElasticBeanstalk`

Elastic Disaster Recovery

Task state resource: `arn:aws:states:::aws-sdk:drs:`*`[apiAction]`*

Exception prefix: `Drs`

Elastic Inference

Task state resource: `arn:aws:states:::aws-sdk:elasticinference:`*`[apiAction]`*

Exception prefix: `ElasticInference`

Elastic Load Balancing V1

Task state resource: `arn:aws:states:::aws-sdk:elasticloadbalancing:`*`[apiAction]`*

Exception prefix: `ElasticLoadBalancing`

Elastic Load Balancing V2

Task state resource: `arn:aws:states:::aws-sdk:elasticloadbalancingv2:`*`[apiAction]`*

Exception prefix: `ElasticLoadBalancingV2`

Elastic Transcoder

Task state resource: `arn:aws:states:::aws-sdk:elastictranscoder:`*`[apiAction]`*

Exception prefix: `ElasticTranscoder`

Unsupported operations: `TestRole`

Amazon ElasticSearch

Task state resource: `arn:aws:states:::aws-sdk:elasticsearch:`*`[apiAction]`*

Exception prefix: `Elasticsearch`

AWS End User Messaging Social

Task state resource: `arn:aws:states:::aws-sdk:socialmessaging:`*`[apiAction]`*

Exception prefix: `SocialMessaging`

AWS Entity Resolution

Task state resource: `arn:aws:states:::aws-sdk:entityresolution:`*`[apiAction]`*

Exception prefix: `EntityResolution`

Amazon EventBridge

Task state resource: `arn:aws:states:::aws-sdk:eventbridge:`*`[apiAction]`*

Exception prefix: `EventBridge`

EventBridge Pipes

Task state resource: `arn:aws:states:::aws-sdk:pipes:`*`[apiAction]`*

Exception prefix: `Pipes`

EventBridge Scheduler

Task state resource: `arn:aws:states:::aws-sdk:scheduler:`*`[apiAction]`*

Exception prefix: `Scheduler`

EventBridge Schema Registry

Task state resource: `arn:aws:states:::aws-sdk:schemas:`*`[apiAction]`*

Exception prefix: `Schemas`

Evidently

Task state resource: `arn:aws:states:::aws-sdk:evidently:`*`[apiAction]`*

Exception prefix: `Evidently`

AWS FIS

Task state resource: `arn:aws:states:::aws-sdk:fis:`*`[apiAction]`*

Exception prefix: `Fis`

Amazon FSx

Task state resource: `arn:aws:states:::aws-sdk:fsx:`*`[apiAction]`*

Exception prefix: `FSx`

FinSpace Data

Task state resource: `arn:aws:states:::aws-sdk:finspacedata:`*`[apiAction]`*

Exception prefix: `FinspaceData`

FinSpace Management

Task state resource: `arn:aws:states:::aws-sdk:finspace:`*`[apiAction]`*

Exception prefix: `Finspace`

Firewall Manager

Task state resource: `arn:aws:states:::aws-sdk:fms:`*`[apiAction]`*

Exception prefix: `Fms`

Amazon Forecast

Task state resource: `arn:aws:states:::aws-sdk:forecast:`*`[apiAction]`*

Exception prefix: `Forecast`

Amazon Forecast Query

Task state resource: `arn:aws:states:::aws-sdk:forecastquery:`*`[apiAction]`*

Exception prefix: `Forecastquery`

Amazon Fraud Detector

Task state resource: `arn:aws:states:::aws-sdk:frauddetector:`*`[apiAction]`*

Exception prefix: `FraudDetector`

AWS Free Tier

Task state resource: `arn:aws:states:::aws-sdk:freetier:`*`[apiAction]`*

Exception prefix: `FreeTier`

Amazon GameLift

Task state resource: `arn:aws:states:::aws-sdk:gamelift:`*`[apiAction]`*

Exception prefix: `GameLift`

AWS Glue

Task state resource: `arn:aws:states:::aws-sdk:glue:`*`[apiAction]`*

Exception prefix: `Glue`

AWS Glue DataBrew

Task state resource: `arn:aws:states:::aws-sdk:databrew:`*`[apiAction]`*

Exception prefix: `DataBrew`

AWS Ground Station

Task state resource: `arn:aws:states:::aws-sdk:groundstation:`*`[apiAction]`*

Exception prefix: `GroundStation`

Amazon GuardDuty

Task state resource: `arn:aws:states:::aws-sdk:guardduty:`*`[apiAction]`*

Exception prefix: `GuardDuty`

AWS Health

Task state resource: `arn:aws:states:::aws-sdk:health:`*`[apiAction]`*

Exception prefix: `Health`

AWS Health Imaging

Task state resource: `arn:aws:states:::aws-sdk:medicalimaging:`*`[apiAction]`*

Exception prefix: `MedicalImaging`

Amazon HealthLake

Task state resource: `arn:aws:states:::aws-sdk:healthlake:`*`[apiAction]`*

Exception prefix: `HealthLake`

Amazon Honeycode

Task state resource: `arn:aws:states:::aws-sdk:honeycode:`*`[apiAction]`*

Exception prefix: `Honeycode`

IAM

Task state resource: `arn:aws:states:::aws-sdk:iam:`*`[apiAction]`*

Exception prefix: `Iam`

IAM Access Analyzer

Task state resource: `arn:aws:states:::aws-sdk:accessanalyzer:`*`[apiAction]`*

Exception prefix: `AccessAnalyzer`

IAM Roles Anywhere

Task state resource: `arn:aws:states:::aws-sdk:rolesanywhere:`*`[apiAction]`*

Exception prefix: `RolesAnywhere`

Amazon IVS

Task state resource: `arn:aws:states:::aws-sdk:ivs:`*`[apiAction]`*

Exception prefix: `Ivs`

Amazon IVS Chat

Task state resource: `arn:aws:states:::aws-sdk:ivschat:`*`[apiAction]`*

Exception prefix: `Ivschat`

Amazon IVS RealTime

Task state resource: `arn:aws:states:::aws-sdk:ivsrealtime:`*`[apiAction]`*

Exception prefix: `IvsRealTime`

Incident Manager

Task state resource: `arn:aws:states:::aws-sdk:ssmincidents:`*`[apiAction]`*

Exception prefix: `SsmIncidents`

Incident Manager Contacts

Task state resource: `arn:aws:states:::aws-sdk:ssmcontacts:`*`[apiAction]`*

Exception prefix: `SsmContacts`

Amazon Inspector Scan

Task state resource: `arn:aws:states:::aws-sdk:inspectorscan:`*`[apiAction]`*

Exception prefix: `InspectorScan`

Amazon Inspector V1

Task state resource: `arn:aws:states:::aws-sdk:inspector:`*`[apiAction]`*

Exception prefix: `Inspector`

Amazon Inspector V2

Task state resource: `arn:aws:states:::aws-sdk:inspector2:`*`[apiAction]`*

Exception prefix: `Inspector2`

AWS Invoicing

Task state resource: `arn:aws:states:::aws-sdk:invoicing:`*`[apiAction]`*

Exception prefix: `Invoicing`

AWS IoT

Task state resource: `arn:aws:states:::aws-sdk:iot:`*`[apiAction]`*

Exception prefix: `Iot`

Unsupported operations: `AttachPrincipalPolicy`, `ListPrincipalPolicies`, `DetachPrincipalPolicy`, `ListPolicyPrincipals`, `DetachPrincipalPolicy`

AWS IoT Analytics

Task state resource: `arn:aws:states:::aws-sdk:iotanalytics:`*`[apiAction]`*

Exception prefix: `IoTAnalytics`

AWS IoT Device Advisor

Task state resource: `arn:aws:states:::aws-sdk:iotdeviceadvisor:`*`[apiAction]`*

Exception prefix: `IotDeviceAdvisor`

Unsupported operations: `ListTestCases`

AWS IoT Events

Task state resource: `arn:aws:states:::aws-sdk:iotevents:`*[apiAction]*

Exception prefix: `IotEvents`

AWS IoT Events Data

Task state resource: `arn:aws:states:::aws-sdk:ioteventsdata:`*[apiAction]*

Exception prefix: `IotEventsData`

AWS IoT Fleet Hub

Task state resource: `arn:aws:states:::aws-sdk:iotfleethub:`*[apiAction]*

Exception prefix: `IoTFleetHub`

AWS IoT FleetWise

Task state resource: `arn:aws:states:::aws-sdk:iotfleetwise:`*[apiAction]*

Exception prefix: `IoTFleetWise`

AWS IoT Greengrass V1

Task state resource: `arn:aws:states:::aws-sdk:greengrass:`*[apiAction]*

Exception prefix: `Greengrass`

AWS IoT Greengrass V2

Task state resource: `arn:aws:states:::aws-sdk:greengrassv2:`*[apiAction]*

Exception prefix: `GreengrassV2`

AWS IoT Jobs Data

Task state resource: `arn:aws:states:::aws-sdk:iotjobsdataplane:`*[apiAction]*

Exception prefix: `IotJobsDataPlane`

AWS IoT Secure Tunneling

Task state resource: `arn:aws:states:::aws-sdk:iotsecuretunneling:`*[apiAction]*

Exception prefix: `IoTSecureTunneling`

AWS IoT SiteWise

Task state resource: `arn:aws:states:::aws-sdk:iotsitewise:[apiAction]`

Exception prefix: `IoTSiteWise`

Unsupported operations: `InvokeAssistant`

AWS IoT Things Graph

Task state resource: `arn:aws:states:::aws-sdk:iotthingsgraph:[apiAction]`

Exception prefix: `IoTThingsGraph`

AWS IoT TwinMaker

Task state resource: `arn:aws:states:::aws-sdk:iottwinmaker:[apiAction]`

Exception prefix: `IoTTwinMaker`

AWS IoT Wireless

Task state resource: `arn:aws:states:::aws-sdk:iotwireless:[apiAction]`

Exception prefix: `IotWireless`

AWS KMS

Task state resource: `arn:aws:states:::aws-sdk:kms:[apiAction]`

Exception prefix: `Kms`

Amazon Kendra

Task state resource: `arn:aws:states:::aws-sdk:kendra:[apiAction]`

Exception prefix: `Kendra`

Amazon Kendra Intelligent Ranking

Task state resource: `arn:aws:states:::aws-sdk:kendraranking:[apiAction]`

Exception prefix: `KendraRanking`

Amazon Keyspaces

Task state resource: `arn:aws:states:::aws-sdk:keyspaces:[apiAction]`

Exception prefix: `Keyspaces`

Kinesis Data Analytics V1

Task state resource: `arn:aws:states:::aws-sdk:kinesisanalytics:[apiAction]`

Exception prefix: `KinesisAnalytics`

Kinesis Data Analytics V2

Task state resource: `arn:aws:states:::aws-sdk:kinesisanalyticsv2:[apiAction]`

Exception prefix: `KinesisAnalyticsV2`

Kinesis Data Firehose

Task state resource: `arn:aws:states:::aws-sdk:firehose:[apiAction]`

Exception prefix: `Firehose`

Kinesis Data Streams

Task state resource: `arn:aws:states:::aws-sdk:kinesis:[apiAction]`

Exception prefix: `Kinesis`

Unsupported operations: `SubscribeToShard`

Kinesis Video Signaling Channels

Task state resource: `arn:aws:states:::aws-sdk:kinesisvideosignaling:[apiAction]`

Exception prefix: `KinesisVideoSignaling`

Kinesis Video Streams

Task state resource: `arn:aws:states:::aws-sdk:kinesisvideo:[apiAction]`

Exception prefix: `KinesisVideo`

Kinesis Video Streams Archived Media

Task state resource: `arn:aws:states:::aws-sdk:kinesisvideoarchivedmedia:[apiAction]`

Exception prefix: `KinesisVideoArchivedMedia`

Kinesis Video Streams Media

Task state resource: `arn:aws:states:::aws-sdk:kinesisvideomedia:[apiAction]`

Exception prefix: `KinesisVideoMedia`

Kinesis Video WebRTC Storage

Task state resource: `arn:aws:states:::aws-sdk:kinesisvideowebrtcstorage:[apiAction]`

Exception prefix: `KinesisVideoWebRtcStorage`

AWS Lake Formation

Task state resource: `arn:aws:states:::aws-sdk:lakeformation:[apiAction]`

Exception prefix: `LakeFormation`

AWS Lambda

Task state resource: `arn:aws:states:::aws-sdk:lambda:[apiAction]`

Exception prefix: `Lambda`

Unsupported operations: `InvokeAsync`, `InvokeWithResponseStream`

AWS Launch Wizard

Task state resource: `arn:aws:states:::aws-sdk:launchwizard:[apiAction]`

Exception prefix: `LaunchWizard`

Amazon Lex Model Building V1

Task state resource: `arn:aws:states:::aws-sdk:lexmodelbuilding:[apiAction]`

Exception prefix: `LexModelBuilding`

Amazon Lex Model Building V2

Task state resource: `arn:aws:states:::aws-sdk:lexmodelsv2:[apiAction]`

Exception prefix: `LexModelsV2`

Amazon Lex Runtime V1

Task state resource: `arn:aws:states:::aws-sdk:lexruntime:[apiAction]`

Exception prefix: `LexRuntime`

Amazon Lex Runtime V2

Task state resource: `arn:aws:states:::aws-sdk:lexruntimev2:[apiAction]`

Exception prefix: `LexRuntimeV2`

Unsupported operations: `StartConversation`

AWS License Manager

Task state resource: `arn:aws:states:::aws-sdk:licensemanager:[apiAction]`

Exception prefix: `LicenseManager`

License Manager Linux Subscriptions

Task state resource: `arn:aws:states:::aws-sdk:licensemanagerlinuxsubscriptions:[apiAction]`

Exception prefix: `LicenseManagerLinuxSubscriptions`

License Manager User Subscriptions

Task state resource: `arn:aws:states:::aws-sdk:licensemanagerusersubscriptions:[apiAction]`

Exception prefix: `LicenseManagerUserSubscriptions`

Amazon Lightsail

Task state resource: `arn:aws:states:::aws-sdk:lightsail:[apiAction]`

Exception prefix: `Lightsail`

Amazon Location

Task state resource: `arn:aws:states:::aws-sdk:location:[apiAction]`

Exception prefix: `Location`

Amazon Location Service Maps V2

Task state resource: `arn:aws:states:::aws-sdk:geomaps:[apiAction]`

Exception prefix: `GeoMaps`

Amazon Location Service Places V2

Task state resource: `arn:aws:states:::aws-sdk:geoplaces:[apiAction]`

Exception prefix: `GeoPlaces`

Amazon Location Service Routes V2

Task state resource: `arn:aws:states:::aws-sdk:georoutes:`*`[apiAction]`*

Exception prefix: `GeoRoutes`

Lookout for Equipment

Task state resource: `arn:aws:states:::aws-sdk:lookoutequipment:`*`[apiAction]`*

Exception prefix: `LookoutEquipment`

Lookout for Metrics

Task state resource: `arn:aws:states:::aws-sdk:lookoutmetrics:`*`[apiAction]`*

Exception prefix: `LookoutMetrics`

Lookout for Vision

Task state resource: `arn:aws:states:::aws-sdk:lookoutvision:`*`[apiAction]`*

Exception prefix: `LookoutVision`

Amazon MQ

Task state resource: `arn:aws:states:::aws-sdk:mq:`*`[apiAction]`*

Exception prefix: `Mq`

Amazon MSK

Task state resource: `arn:aws:states:::aws-sdk:kafka:`*`[apiAction]`*

Exception prefix: `Kafka`

Amazon MSK Connect

Task state resource: `arn:aws:states:::aws-sdk:kafkaconnect:`*`[apiAction]`*

Exception prefix: `KafkaConnect`

Amazon MWAA

Task state resource: `arn:aws:states:::aws-sdk:mwaa:`*`[apiAction]`*

Exception prefix: `Mwaa`

Amazon Macie V2

Task state resource: `arn:aws:states:::aws-sdk:macie2:[apiAction]`

Exception prefix: `Macie2`

MailManager

Task state resource: `arn:aws:states:::aws-sdk:mailmanager:[apiAction]`

Exception prefix: `MailManager`

AWS Mainframe Modernization

Task state resource: `arn:aws:states:::aws-sdk:m2:[apiAction]`

Exception prefix: `M2`

AWS Mainframe Modernization Application Testing

Task state resource: `arn:aws:states:::aws-sdk:apptest:[apiAction]`

Exception prefix: `AppTest`

Managed Blockchain

Task state resource: `arn:aws:states:::aws-sdk:managedblockchain:[apiAction]`

Exception prefix: `ManagedBlockchain`

Managed Blockchain Query

Task state resource: `arn:aws:states:::aws-sdk:managedblockchainquery:[apiAction]`

Exception prefix: `ManagedBlockchainQuery`

Amazon Managed Grafana

Task state resource: `arn:aws:states:::aws-sdk:grafana:[apiAction]`

Exception prefix: `Grafana`

AWS Marketplace Catalog

Task state resource: `arn:aws:states:::aws-sdk:marketplacecatalog:[apiAction]`

Exception prefix: `MarketplaceCatalog`

AWS Marketplace Commerce Analytics

Task state resource: `arn:aws:states:::aws-sdk:marketplacecommerceanalytics:[apiAction]`

Exception prefix: `MarketplaceCommerceAnalytics`

AWS Marketplace Entitlement Service

Task state resource: `arn:aws:states:::aws-sdk:marketplaceentitlement:[apiAction]`

Exception prefix: `MarketplaceEntitlement`

AWS Marketplace Metering

Task state resource: `arn:aws:states:::aws-sdk:marketplacemetering:[apiAction]`

Exception prefix: `MarketplaceMetering`

AWS Marketplace Reporting Service

Task state resource: `arn:aws:states:::aws-sdk:marketplacereporting:[apiAction]`

Exception prefix: `MarketplaceReporting`

Amazon Mechanical Turk

Task state resource: `arn:aws:states:::aws-sdk:mturk:[apiAction]`

Exception prefix: `MTurk`

MediaConnect

Task state resource: `arn:aws:states:::aws-sdk:mediaconnect:[apiAction]`

Exception prefix: `MediaConnect`

MediaConvert

Task state resource: `arn:aws:states:::aws-sdk:mediaconvert:[apiAction]`

Exception prefix: `MediaConvert`

MediaLive

Task state resource: `arn:aws:states:::aws-sdk:medialive:[apiAction]`

Exception prefix: `MediaLive`

MediaPackage V1

Task state resource: `arn:aws:states:::aws-sdk:mediapackage:`*`[apiAction]`*

Exception prefix: `MediaPackage`

Unsupported operations: `RotateChannelCredentials`

MediaPackage V2

Task state resource: `arn:aws:states:::aws-sdk:mediapackagev2:`*`[apiAction]`*

Exception prefix: `MediaPackageV2`

MediaPackage VOD

Task state resource: `arn:aws:states:::aws-sdk:mediapackagevod:`*`[apiAction]`*

Exception prefix: `MediaPackageVod`

MediaStore

Task state resource: `arn:aws:states:::aws-sdk:mediastore:`*`[apiAction]`*

Exception prefix: `MediaStore`

MediaTailor

Task state resource: `arn:aws:states:::aws-sdk:mediatailor:`*`[apiAction]`*

Exception prefix: `MediaTailor`

Amazon MemoryDB

Task state resource: `arn:aws:states:::aws-sdk:memorydb:`*`[apiAction]`*

Exception prefix: `MemoryDb`

Migration Hub

Task state resource: `arn:aws:states:::aws-sdk:migrationhub:`*`[apiAction]`*

Exception prefix: `MigrationHub`

Migration Hub Home Region

Task state resource: `arn:aws:states:::aws-sdk:migrationhubconfig:`*`[apiAction]`*

Exception prefix: `MigrationHubConfig`

Migration Hub Orchestrator

Task state resource: `arn:aws:states:::aws-sdk:migrationhuborchestrator:[apiAction]`

Exception prefix: `MigrationHubOrchestrator`

Migration Hub Refactor Spaces

Task state resource: `arn:aws:states:::aws-sdk:migrationhubrefactorspaces:[apiAction]`

Exception prefix: `MigrationHubRefactorSpaces`

Migration Hub Strategy Recommendations

Task state resource: `arn:aws:states:::aws-sdk:migrationhubstrategy:[apiAction]`

Exception prefix: `MigrationHubStrategy`

Amazon Neptune

Task state resource: `arn:aws:states:::aws-sdk:neptune:[apiAction]`

Exception prefix: `Neptune`

Amazon Neptune Graph

Task state resource: `arn:aws:states:::aws-sdk:neptunegraph:[apiAction]`

Exception prefix: `NeptuneGraph`

Network Firewall

Task state resource: `arn:aws:states:::aws-sdk:networkfirewall:[apiAction]`

Exception prefix: `NetworkFirewall`

Network Flow Monitor

Task state resource: `arn:aws:states:::aws-sdk:networkflowmonitor:[apiAction]`

Exception prefix: `NetworkFlowMonitor`

Network Manager

Task state resource: `arn:aws:states:::aws-sdk:networkmanager:[apiAction]`

Exception prefix: `NetworkManager`

Network Monitor

Task state resource: `arn:aws:states:::aws-sdk:networkmonitor:[apiAction]`

Exception prefix: `NetworkMonitor`

Amazon Omics

Task state resource: `arn:aws:states:::aws-sdk:omics:[apiAction]`

Exception prefix: `Omics`

Amazon OpenSearch

Task state resource: `arn:aws:states:::aws-sdk:opensearch:[apiAction]`

Exception prefix: `OpenSearch`

Amazon OpenSearch Ingestion

Task state resource: `arn:aws:states:::aws-sdk:osis:[apiAction]`

Exception prefix: `Osis`

OpenSearch Serverless

Task state resource: `arn:aws:states:::aws-sdk:opensearchserverless:[apiAction]`

Exception prefix: `OpenSearchServerless`

OpsWorks

Task state resource: `arn:aws:states:::aws-sdk:opsworks:[apiAction]`

Exception prefix: `OpsWorks`

OpsWorks CM

Task state resource: `arn:aws:states:::aws-sdk:opsworkscm:[apiAction]`

Exception prefix: `OpsWorksCm`

AWS Organizations

Task state resource: `arn:aws:states:::aws-sdk:organizations:[apiAction]`

Exception prefix: `Organizations`

AWS Outposts

Task state resource: `arn:aws:states:::aws-sdk:outposts:[apiAction]`

Exception prefix: `Outposts`

AWS Panorama

Task state resource: `arn:aws:states:::aws-sdk:panorama:[apiAction]`

Exception prefix: `Panorama`

AWS Parallel Computing Service

Task state resource: `arn:aws:states:::aws-sdk:pcs:[apiAction]`

Exception prefix: `Pcs`

Partner Central Selling API

Task state resource: `arn:aws:states:::aws-sdk:partnercentralselling:[apiAction]`

Exception prefix: `PartnerCentralSelling`

Payment Cryptography

Task state resource: `arn:aws:states:::aws-sdk:paymentcryptography:[apiAction]`

Exception prefix: `PaymentCryptography`

Payment Cryptography Data

Task state resource: `arn:aws:states:::aws-sdk:paymentcryptographydata:[apiAction]`

Exception prefix: `PaymentCryptographyData`

Amazon Personalize

Task state resource: `arn:aws:states:::aws-sdk:personalize:[apiAction]`

Exception prefix: `Personalize`

Amazon Personalize Events

Task state resource: `arn:aws:states:::aws-sdk:personalizeevents:[apiAction]`

Exception prefix: `PersonalizeEvents`

Amazon Personalize Runtime

Task state resource: `arn:aws:states:::aws-sdk:personalizeruntime:[apiAction]`

Exception prefix: `PersonalizeRuntime`

Amazon Pinpoint

Task state resource: `arn:aws:states:::aws-sdk:pinpoint:[apiAction]`

Exception prefix: `Pinpoint`

Amazon Pinpoint Email Service

Task state resource: `arn:aws:states:::aws-sdk:pinpointemail:[apiAction]`

Exception prefix: `PinpointEmail`

Amazon Pinpoint SMS and Voice V1

Task state resource: `arn:aws:states:::aws-sdk:pinpointsmsvoice:[apiAction]`

Exception prefix: `PinpointSmsVoice`

Amazon Pinpoint SMS and Voice V2

Task state resource: `arn:aws:states:::aws-sdk:pinpointsmsvoicev2:[apiAction]`

Exception prefix: `PinpointSmsVoiceV2`

Amazon Polly

Task state resource: `arn:aws:states:::aws-sdk:polly:[apiAction]`

Exception prefix: `Polly`

AWS Price List

Task state resource: `arn:aws:states:::aws-sdk:pricing:[apiAction]`

Exception prefix: `Pricing`

AWS Private 5G

Task state resource: `arn:aws:states:::aws-sdk:privatenetworks:[apiAction]`

Exception prefix: `PrivateNetworks`

Private CA Connector for Active Directory

Task state resource: `arn:aws:states:::aws-sdk:pcaconnectorad:`*`[apiAction]`*

Exception prefix: `PcaConnectorAd`

Private CA Connector for SCEP

Task state resource: `arn:aws:states:::aws-sdk:pcaconnectorscep:`*`[apiAction]`*

Exception prefix: `PcaConnectorScep`

Amazon Prometheus

Task state resource: `arn:aws:states:::aws-sdk:amp:`*`[apiAction]`*

Exception prefix: `Amp`

AWS Proton

Task state resource: `arn:aws:states:::aws-sdk:proton:`*`[apiAction]`*

Exception prefix: `Proton`

Amazon Q Apps

Task state resource: `arn:aws:states:::aws-sdk:qapps:`*`[apiAction]`*

Exception prefix: `QApps`

Amazon Q Business

Task state resource: `arn:aws:states:::aws-sdk:qbusiness:`*`[apiAction]`*

Exception prefix: `QBusiness`

Unsupported operations: `Chat`

Amazon Q Connect

Task state resource: `arn:aws:states:::aws-sdk:qconnect:`*`[apiAction]`*

Exception prefix: `QConnect`

Amazon QLDB

Task state resource: `arn:aws:states:::aws-sdk:qldb:`*`[apiAction]`*

Exception prefix: `Qldb`

Amazon QLDB Session

Task state resource: `arn:aws:states:::aws-sdk:qldbsession:`*`[apiAction]`*

Exception prefix: `QldbSession`

Amazon QuickSight

Task state resource: `arn:aws:states:::aws-sdk:quicksight:`*`[apiAction]`*

Exception prefix: `QuickSight`

Amazon RDS

Task state resource: `arn:aws:states:::aws-sdk:rds:`*`[apiAction]`*

Exception prefix: `Rds`

Amazon RDS Data

Task state resource: `arn:aws:states:::aws-sdk:rdsdata:`*`[apiAction]`*

Exception prefix: `RdsData`

Unsupported operations: `ExecuteSql`

Amazon RDS Performance Insights

Task state resource: `arn:aws:states:::aws-sdk:pi:`*`[apiAction]`*

Exception prefix: `Pi`

Recycle Bin for EBS

Task state resource: `arn:aws:states:::aws-sdk:rbin:`*`[apiAction]`*

Exception prefix: `Rbin`

Amazon Redshift

Task state resource: `arn:aws:states:::aws-sdk:redshift:`*`[apiAction]`*

Exception prefix: `Redshift`

Amazon Redshift Data

Task state resource: `arn:aws:states:::aws-sdk:redshiftdata:`*`[apiAction]`*

Exception prefix: `RedshiftData`

Amazon Redshift Serverless

Task state resource: `arn:aws:states:::aws-sdk:redshiftserverless:`*`[apiAction]`*

Exception prefix: `RedshiftServerless`

Amazon Rekognition

Task state resource: `arn:aws:states:::aws-sdk:rekognition:`*`[apiAction]`*

Exception prefix: `Rekognition`

Resilience Hub

Task state resource: `arn:aws:states:::aws-sdk:resiliencehub:`*`[apiAction]`*

Exception prefix: `Resiliencehub`

AWS Resource Access Manager

Task state resource: `arn:aws:states:::aws-sdk:ram:`*`[apiAction]`*

Exception prefix: `Ram`

AWS Resource Explorer

Task state resource: `arn:aws:states:::aws-sdk:resourceexplorer2:`*`[apiAction]`*

Exception prefix: `ResourceExplorer2`

Resource Groups

Task state resource: `arn:aws:states:::aws-sdk:resourcegroups:`*`[apiAction]`*

Exception prefix: `ResourceGroups`

Resource Groups Tagging

Task state resource: `arn:aws:states:::aws-sdk:resourcegroupstaggingapi:`*`[apiAction]`*

Exception prefix: `ResourceGroupsTaggingApi`

AWS RoboMaker

Task state resource: `arn:aws:states:::aws-sdk:robomaker:`*`[apiAction]`*

Exception prefix: `RoboMaker`

Route 53

Task state resource: `arn:aws:states:::aws-sdk:route53:`*`[apiAction]`*

Exception prefix: `Route53`

Route 53 ARC Zonal Shift

Task state resource: `arn:aws:states:::aws-sdk:arczonalshift:`*`[apiAction]`*

Exception prefix: `ArcZonalShift`

Route 53 Domains

Task state resource: `arn:aws:states:::aws-sdk:route53domains:`*`[apiAction]`*

Exception prefix: `Route53Domains`

Route 53 Profiles

Task state resource: `arn:aws:states:::aws-sdk:route53profiles:`*`[apiAction]`*

Exception prefix: `Route53Profiles`

Route 53 Recovery Control Config

Task state resource: `arn:aws:states:::aws-sdk:route53recoverycontrolconfig:`*`[apiAction]`*

Exception prefix: `Route53RecoveryControlConfig`

Route 53 Recovery Readiness

Task state resource: `arn:aws:states:::aws-sdk:route53recoveryreadiness:`*`[apiAction]`*

Exception prefix: `Route53RecoveryReadiness`

Route 53 Resolver

Task state resource: `arn:aws:states:::aws-sdk:route53resolver:`*`[apiAction]`*

Exception prefix: `Route53Resolver`

Route 53 Routing Control

Task state resource: `arn:aws:states:::aws-sdk:route53recoverycluster:`*`[apiAction]`*

Exception prefix: `Route53RecoveryCluster`

Runtime for Amazon Bedrock Data Automation

Task state resource: `arn:aws:states:::aws-sdk:bedrockdataautomationruntime:[apiAction]`

Exception prefix: `BedrockDataAutomationRuntime`

Amazon S3

Task state resource: `arn:aws:states:::aws-sdk:s3:[apiAction]`

Exception prefix: `S3`

Unsupported operations: `SelectObjectContent`

Amazon S3 Control

Task state resource: `arn:aws:states:::aws-sdk:s3control:[apiAction]`

Exception prefix: `S3Control`

Unsupported operations: `SelectObjectContent`

Amazon S3 Glacier

Task state resource: `arn:aws:states:::aws-sdk:glacier:[apiAction]`

Exception prefix: `Glacier`

Amazon S3 Tables

Task state resource: `arn:aws:states:::aws-sdk:s3tables:[apiAction]`

Exception prefix: `S3Tables`

Amazon S3 on Outposts

Task state resource: `arn:aws:states:::aws-sdk:s3outposts:[apiAction]`

Exception prefix: `S3Outposts`

Amazon SES V1

Task state resource: `arn:aws:states:::aws-sdk:ses:[apiAction]`

Exception prefix: `Ses`

Amazon SES V2

Task state resource: `arn:aws:states:::aws-sdk:sesv2:[apiAction]`

Exception prefix: `SesV2`

Amazon SNS

Task state resource: `arn:aws:states:::aws-sdk:sns:[apiAction]`

Exception prefix: `Sns`

Amazon SQS

Task state resource: `arn:aws:states:::aws-sdk:sqs:[apiAction]`

Exception prefix: `Sqs`

AWS SSO

Task state resource: `arn:aws:states:::aws-sdk:sso:[apiAction]`

Exception prefix: `Sso`

AWS SSO

Task state resource: `arn:aws:states:::aws-sdk:identitystore:[apiAction]`

Exception prefix: `Identitystore`

AWS SSO Admin

Task state resource: `arn:aws:states:::aws-sdk:ssoadmin:[apiAction]`

Exception prefix: `SsoAdmin`

AWS SSO OIDC

Task state resource: `arn:aws:states:::aws-sdk:ssooidc:[apiAction]`

Exception prefix: `SsoOidc`

Amazon SWF

Task state resource: `arn:aws:states:::aws-sdk:swf:[apiAction]`

Exception prefix: `Swf`

SageMaker

Task state resource: `arn:aws:states:::aws-sdk:sagemaker:[apiAction]`

Exception prefix: `SageMaker`

SageMaker Edge Manager

Task state resource: `arn:aws:states:::aws-sdk:sagemakeredge:[apiAction]`

Exception prefix: `SagemakerEdge`

SageMaker Feature Store

Task state resource: `arn:aws:states:::aws-sdk:sagemakerfeaturestoreruntime:[apiAction]`

Exception prefix: `SageMakerFeatureStoreRuntime`

SageMaker Geospatial

Task state resource: `arn:aws:states:::aws-sdk:sagemakergeospatial:[apiAction]`

Exception prefix: `SageMakerGeospatial`

SageMaker Metrics

Task state resource: `arn:aws:states:::aws-sdk:sagemakermetrics:[apiAction]`

Exception prefix: `SageMakerMetrics`

SageMaker Runtime

Task state resource: `arn:aws:states:::aws-sdk:sagemakerruntime:[apiAction]`

Exception prefix: `SageMakerRuntime`

Unsupported operations: `InvokeEndpointWithResponseStream`

AWS Savings Plans

Task state resource: `arn:aws:states:::aws-sdk:savingsplans:[apiAction]`

Exception prefix: `Savingsplans`

AWS Secrets Manager

Task state resource: `arn:aws:states:::aws-sdk:secretsmanager:[apiAction]`

Exception prefix: SecretsManager

AWS Security Hub

Task state resource: arn:aws:states:::aws-sdk:securityhub:*[apiAction]*

Exception prefix: SecurityHub

Security Incident Response

Task state resource: arn:aws:states:::aws-sdk:securityir:*[apiAction]*

Exception prefix: SecurityIr

Amazon Security Lake

Task state resource: arn:aws:states:::aws-sdk:securitylake:*[apiAction]*

Exception prefix: SecurityLake

Unsupported operations: GetDatalake, GetDatalakeAutoEnable, GetDatalakeExceptionsExpiry, GetDatalakeExceptionsSubscription, GetDatalakeStatus, CreateSubscriptionNotificationConfiguration, CreateDatalake, CreateDatalakeAutoEnable, CreateDatalakeDelegatedAdmin, CreateDatalakeExceptionsSubscription, DeleteDatalake, UpdateDatalake, UpdateSubscriptionNotificationConfiguration, UpdateDatalakeExceptionsExpiry, UpdateDatalakeExceptionsSubscription, DeleteDatalakeAutoEnable, DeleteDatalakeDelegatedAdmin, DeleteDatalakeExceptionsSubscription, DeleteSubscriptionNotificationConfiguration, ListDatalakeExceptions

AWS Security Token Service

Task state resource: arn:aws:states:::aws-sdk:sts:*[apiAction]*

Exception prefix: Sts

Unsupported operations: AssumeRole, AssumeRoleWithSAML, AssumeRoleWithWebIdentity

AWS Server Migration Service

Task state resource: arn:aws:states:::aws-sdk:sms:*[apiAction]*

Exception prefix: Sms

AWS Serverless Application Repository

Task state resource: `arn:aws:states:::aws-sdk:serverlessapplicationrepository:[apiAction]`

Exception prefix: `ServerlessApplicationRepository`

AWS Service Catalog

Task state resource: `arn:aws:states:::aws-sdk:servicecatalog:[apiAction]`

Exception prefix: `ServiceCatalog`

AWS Service Catalog App Registry

Task state resource: `arn:aws:states:::aws-sdk:servicecatalogappregistry:[apiAction]`

Exception prefix: `ServiceCatalogAppRegistry`

Service Quotas

Task state resource: `arn:aws:states:::aws-sdk:servicequotas:[apiAction]`

Exception prefix: `ServiceQuotas`

AWS Shield

Task state resource: `arn:aws:states:::aws-sdk:shield:[apiAction]`

Exception prefix: `Shield`

Unsupported operations: `DeleteSubscription`

AWS Signer

Task state resource: `arn:aws:states:::aws-sdk:signer:[apiAction]`

Exception prefix: `Signer`

AWS SimSpace Weaver

Task state resource: `arn:aws:states:::aws-sdk:simspaceweaver:[apiAction]`

Exception prefix: `SimSpaceWeaver`

AWS Snow Device Management

Task state resource: `arn:aws:states:::aws-sdk:snowdevicemanagement:[apiAction]`

Exception prefix: `SnowDeviceManagement`

AWS Snowball

Task state resource: `arn:aws:states:::aws-sdk:snowball:`*`[apiAction]`*

Exception prefix: `Snowball`

AWS Step Functions

Task state resource: `arn:aws:states:::aws-sdk:sfn:`*`[apiAction]`*

Exception prefix: `Sfn`

AWS Storage Gateway

Task state resource: `arn:aws:states:::aws-sdk:storagegateway:`*`[apiAction]`*

Exception prefix: `StorageGateway`

AWS Supply Chain

Task state resource: `arn:aws:states:::aws-sdk:supplychain:`*`[apiAction]`*

Exception prefix: `SupplyChain`

AWS Support

Task state resource: `arn:aws:states:::aws-sdk:support:`*`[apiAction]`*

Exception prefix: `Support`

AWS Support App

Task state resource: `arn:aws:states:::aws-sdk:supportapp:`*`[apiAction]`*

Exception prefix: `SupportApp`

Systems Manager

Task state resource: `arn:aws:states:::aws-sdk:ssm:`*`[apiAction]`*

Exception prefix: `Ssm`

AWS Systems Manager QuickSetup

Task state resource: `arn:aws:states:::aws-sdk:ssmquicksetup:`*`[apiAction]`*

Exception prefix: `SsmQuickSetup`

Systems Manager for SAP

Task state resource: `arn:aws:states:::aws-sdk:ssmsap:`*`[apiAction]`*

Exception prefix: `SsmSap`

Tax Settings

Task state resource: `arn:aws:states:::aws-sdk:taxsettings:`*`[apiAction]`*

Exception prefix: `TaxSettings`

AWS Telco Network Builder

Task state resource: `arn:aws:states:::aws-sdk:tnb:`*`[apiAction]`*

Exception prefix: `Tnb`

Amazon Textract

Task state resource: `arn:aws:states:::aws-sdk:textract:`*`[apiAction]`*

Exception prefix: `Textract`

Timestream InfluxDB

Task state resource: `arn:aws:states:::aws-sdk:timestreaminfluxdb:`*`[apiAction]`*

Exception prefix: `TimestreamInfluxDb`

Amazon Timestream Query

Task state resource: `arn:aws:states:::aws-sdk:timestreamquery:`*`[apiAction]`*

Exception prefix: `TimestreamQuery`

Amazon Timestream Write

Task state resource: `arn:aws:states:::aws-sdk:timestreamwrite:`*`[apiAction]`*

Exception prefix: `TimestreamWrite`

Amazon Transcribe

Task state resource: `arn:aws:states:::aws-sdk:transcribe:`*`[apiAction]`*

Exception prefix: `Transcribe`

AWS Transfer Family

Task state resource: `arn:aws:states:::aws-sdk:transfer:[apiAction]`

Exception prefix: `Transfer`

Amazon Translate

Task state resource: `arn:aws:states:::aws-sdk:translate:[apiAction]`

Exception prefix: `Translate`

Trusted Advisor

Task state resource: `arn:aws:states:::aws-sdk:trustedadvisor:[apiAction]`

Exception prefix: `TrustedAdvisor`

AWS User Notifications Contacts

Task state resource: `arn:aws:states:::aws-sdk:notificationscontacts:[apiAction]`

Exception prefix: `NotificationsContacts`

Amazon VPC Lattice

Task state resource: `arn:aws:states:::aws-sdk:vpclattice:[apiAction]`

Exception prefix: `VpcLattice`

Verified Permissions

Task state resource: `arn:aws:states:::aws-sdk:verifiedpermissions:[apiAction]`

Exception prefix: `VerifiedPermissions`

AWS WAF V1

Task state resource: `arn:aws:states:::aws-sdk:waf:[apiAction]`

Exception prefix: `Waf`

AWS WAF V1 Regional

Task state resource: `arn:aws:states:::aws-sdk:wafregional:[apiAction]`

Exception prefix: `WafRegional`

AWS WAF V2

Task state resource: `arn:aws:states:::aws-sdk:wafv2:[apiAction]`

Exception prefix: `Wafv2`

AWS Well-Architected Tool

Task state resource: `arn:aws:states:::aws-sdk:wellarchitected:[apiAction]`

Exception prefix: `WellArchitected`

Amazon WorkDocs

Task state resource: `arn:aws:states:::aws-sdk:workdocs:[apiAction]`

Exception prefix: `WorkDocs`

Amazon WorkMail

Task state resource: `arn:aws:states:::aws-sdk:workmail:[apiAction]`

Exception prefix: `WorkMail`

Amazon WorkMail Message Flow

Task state resource: `arn:aws:states:::aws-sdk:workmailmessageflow:[apiAction]`

Exception prefix: `WorkMailMessageFlow`

Amazon WorkSpaces

Task state resource: `arn:aws:states:::aws-sdk:workspaces:[apiAction]`

Exception prefix: `WorkSpaces`

Amazon WorkSpaces Thin Client

Task state resource: `arn:aws:states:::aws-sdk:workspacesthinclient:[apiAction]`

Exception prefix: `WorkSpacesThinClient`

Amazon WorkSpaces Web

Task state resource: `arn:aws:states:::aws-sdk:workspacesweb:[apiAction]`

Exception prefix: `WorkSpacesWeb`

AWS X-Ray

Task state resource: `arn:aws:states:::aws-sdk:xray:[apiAction]`

Exception prefix: `XRay`

re:Post Private

Task state resource: `arn:aws:states:::aws-sdk:repostspace:[apiAction]`

Exception prefix: `Repostspace`

Deprecated AWS SDK service integrations

The following AWS SDK service integrations are now deprecated:

- AWS Mobile
- Amazon Macie
- AWS IoT RoboRunner

Discover service integration patterns in Step Functions

For service integrations, you can specify various integration patterns to control how your state machine interacts with the integrated AWS services :

- the section called "Request Response" - Call a service and Step Functions will progress to the next state immediately after it receives an HTTP response.
- the section called "Run a Job (.sync)" - Call a service and have Step Functions wait for a job to complete.
- the section called "Wait for Callback" - 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.

Integration pattern support

Standard Workflows and Express Workflows support the same **integrations** but not the same **integration patterns**.

- **Standard Workflows** support *Request Response* integrations. Certain services support *Run a Job (.sync)*, or *Wait for Callback (.waitForTaskToken)* , and both in some cases. See the following optimized integrations table for details.
- **Express Workflows** only support *Request Response* integrations.

To help decide between the two types, see Choosing workflow type in Step Functions.

AWS SDK integrations in Step Functions

Integrated service	Request Response	Run a Job - *.sync*	Wait for Callback - *.waitForTaskToken*
Over two hundred services	Standard & Express	*Not supported*	Standard

Optimized integrations in Step Functions

Integrated service	Request Response	Run a Job - *.sync*	Wait for Callback - *.waitForTaskToken*
Amazon API Gateway	Standard & Express	*Not supported*	Standard
Amazon Athena	Standard & Express	Standard	*Not supported*
AWS Batch	Standard & Express	Standard	*Not supported*
Amazon Bedrock	Standard & Express	Standard	Standard
AWS CodeBuild	Standard & Express	Standard	*Not supported*
Amazon DynamoDB	Standard & Express	*Not supported*	*Not supported*
Amazon ECS/Fargate	Standard & Express	Standard	Standard
Amazon EKS	Standard & Express	Standard	Standard
Amazon EMR	Standard & Express	Standard	*Not supported*

Integrated service	Request Response	Run a Job - *.sync*	Wait for Callback - *.waitForTaskToken*
Amazon EMR on EKS	Standard & Express	Standard	*Not supported*
Amazon EMR Serverless	Standard & Express	Standard	*Not supported*
Amazon EventBridge	Standard & Express	*Not supported*	Standard
AWS Glue	Standard & Express	Standard	*Not supported*
AWS Glue DataBrew	Standard & Express	Standard	*Not supported*
AWS Lambda	Standard & Express	*Not supported*	Standard
AWS Elemental MediaConvert	Standard & Express	Standard	*Not supported*
Amazon SageMaker AI	Standard & Express	Standard	*Not supported*
Amazon SNS	Standard & Express	*Not supported*	Standard
Amazon SQS	Standard & Express	*Not supported*	Standard
AWS Step Functions	Standard & Express	Standard	Standard

Request Response

When you specify a service in the `"Resource"` string of your task state, and you *only* provide the resource, Step Functions will wait for an HTTP response and then progress to the next state. Step Functions will not wait for a job to complete.

The following example shows how you can publish an Amazon SNS topic.

```
"Send message to SNS": {
    "Type":"Task",
    "Resource":"arn:aws:states:::sns:publish",
    "Parameters": {
```

```
        "TopicArn":"arn:aws:sns:region:123456789012:myTopic",
        "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.

> **Tip**
>
> To deploy a sample workflow that uses the Request Response service integration pattern, see [Integrate a service](#) in the getting started tutorial in this guide, or in the [Request Response module](#) in *The AWS Step Functions Workshop*.

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": {
        "JobDefinition": "arn:aws:batch:us-east-2:123456789012:job-definition/
testJobDefinition",
        "JobName": "testJob",
        "JobQueue": "arn:aws:batch:us-east-2:123456789012:job-queue/testQueue"
    },
    "Next": "NEXT_STATE"
}
```

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 with AWS Batch and Amazon SNS.

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.

When you use the `.sync` service integration pattern, Step Functions uses polling that consumes your assigned quota and events to monitor a job's status. For `.sync` invocations within the same account, Step Functions uses EventBridge events and polls the APIs that you specify in the `Task` state. For cross-account `.sync` invocations, Step Functions only uses polling. For example, for `states:StartExecution.sync`, Step Functions performs polling on the DescribeExecution API and uses your assigned quota.

> **Tip**
>
> To deploy an example workflow that uses the .sync integration pattern, see Run a Job (.sync) in *The AWS Step Functions Workshop*.

To see a list of what integrated services support waiting for a job to complete (`.sync`), see Integrating services with Step Functions.

> **Note**
>
> Service integrations that use the `.sync` or `.waitForTaskToken` patterns require additional IAM permissions. For more information, see How Step Functions generates IAM policies for integrated services.

In some cases, you may want Step Functions to continue your workflow before the job is fully complete. You can achieve this in the same way as when using the Wait for a Callback with Task Token service integration pattern. To do this, pass a task token to your job, then return it using a `SendTaskSuccess` or `SendTaskFailure` API call. Step Functions will use the data you provide in that call to complete the task, stop monitoring the job, and continue the workflow.

Wait for a Callback with 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 until the workflow execution reaches the one year service quota (see, Quotas related to state throttling), 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 service integrations. 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.

> **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.

> **Tip**
>
> To deploy an example workflow that uses a callback task token integration pattern, see Callback with Task Token in *The AWS Step Functions Workshop*.

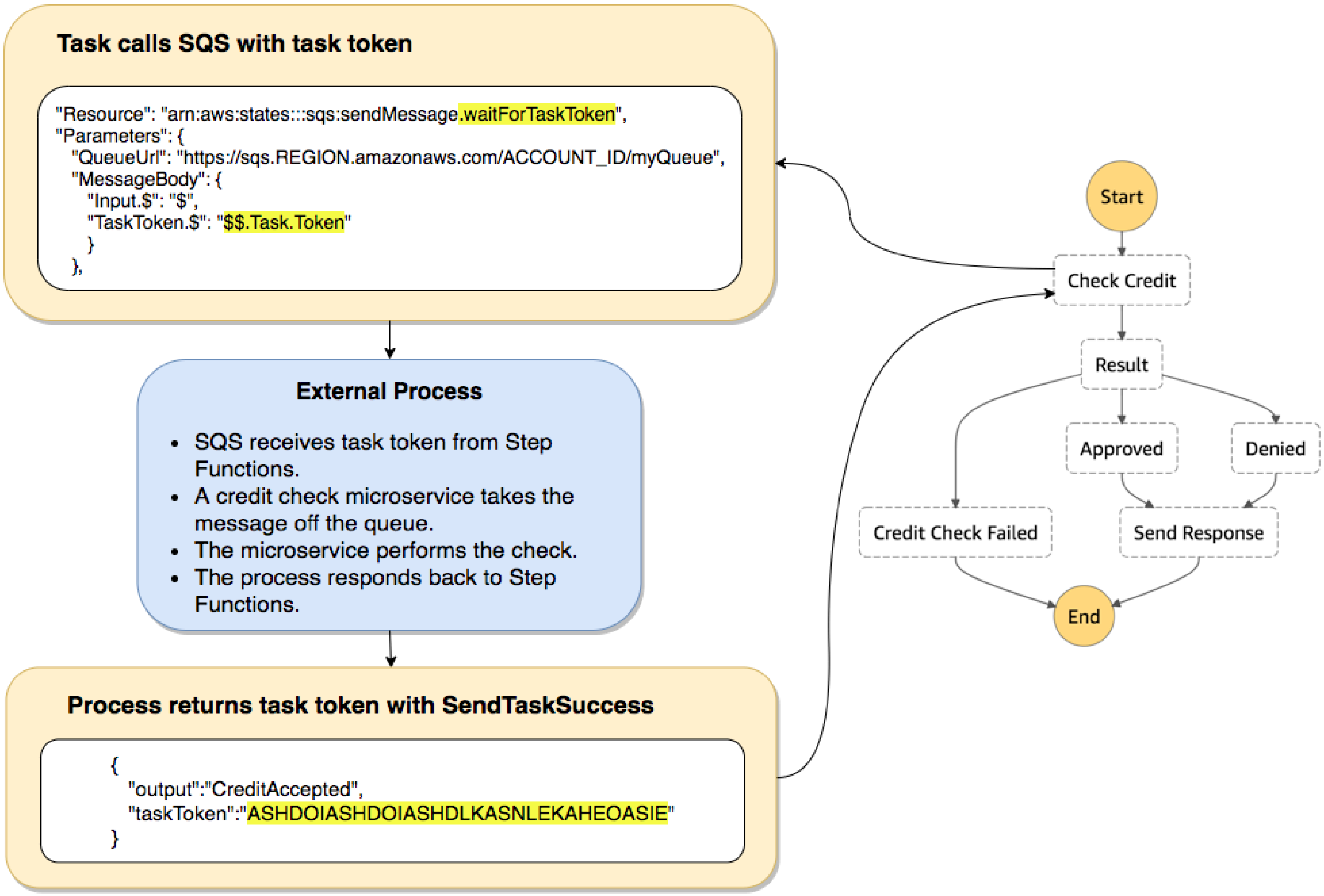
To see a list of what integrated services support waiting for a task token (`.waitForTaskToken`), see Integrating services with Step Functions.

Topics

- Task Token Example
- Get a Token from the Context object
- Configure a Heartbeat Timeout for a Waiting Task

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.



The "`Resource`" field of the task definition that references Amazon SQS includes `.waitForTaskToken` appended to the end.

```
"Send message to SQS": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
    "Parameters": {
        "QueueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/myQueue",
        "MessageBody": {
            "Message": "Hello from Step Functions!",
            "TaskToken.$": "$$.Task.Token"
        }
    },
    "Next": "NEXT_STATE"
}
```

This tells Step Functions to pause and wait for the task token. When you specify a resource using `.waitForTaskToken`, the task token can be accessed in the `"Parameters"` field of your state definition with a special path designation (`$$.Task.Token`). The initial `$$.` designates that the path accesses the Context object, 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.

Get a Token from the Context object

The Context object is an internal JSON object that contains information about your execution. Like state input, it can be accessed with a path from the `"Parameters"` field during an execution. When accessed from within a task definition, it includes information about the specific execution, including the task token.

```
{
    "Execution": {
        "Id": "arn:aws:states:region:account-id:execution:stateMachineName:executionName",
        "Input": {
            "key": "value"
        },
        "Name": "executionName",
        "RoleArn": "arn:aws:iam::account-id:role...",
        "StartTime": "2019-03-26T20:14:13.192Z"
    },
    "State": {
        "EnteredTime": "2019-03-26T20:14:13.192Z",
        "Name": "Test",
        "RetryCount": 3
    },
    "StateMachine": {
        "Id": "arn:aws:states:region:account-id:stateMachine:stateMachineName",
```

```
        "Name": "name"
    },
    "Task": {
        "Token": "h7XRiCdLtd/83p1E0dMccoxlzFhglsdkzpK9mBVKZsp7d9yrT1W"
    }
}
```

You can access the task token by using a special path from inside the `"Parameters"` field of your task definition. To access the input or the Context object, you first specify that the parameter will be a path by appending a `.$` to the parameter name. The following specifies nodes from both the input and the Context object in a `"Parameters"` specification.

```
"Parameters": {
    "Input.$": "$",
    "TaskToken.$": "$$.Task.Token"
},
```

In both cases, appending `.$` to the parameter name tells Step Functions to expect a path. In the first case, `"$"` is a path that includes the entire input. In the second case, `$$.` specifies that the path will access the Context object, and `$$.Task.Token` sets the parameter to the value of the task token in the Context object of a running execution.

In the Amazon SQS example, `.waitForTaskToken` in the `"Resource"` field tells Step Functions to wait for the task token to be returned. The `"TaskToken.$": "`**`$$.Task.Token`**`"` parameter passes that token as a part of the Amazon SQS message.

```
"Send message to SQS": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
    "Parameters": {
        "QueueUrl": "https://sqs.us-east-2.amazonaws.com/123456789012/myQueue",
        "MessageBody": {
            "Message": "Hello from Step Functions!",
            "TaskToken.$": "$$.Task.Token"
        }
    },
    "Next": "NEXT_STATE"
}
```

For more information about the Context object, see [Accessing execution data from the Context object in Step Functions](#) in the [Processing input and output](#) 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). To avoid stuck executions you can configure a heartbeat timeout interval in your state machine definition. Use the `HeartbeatSeconds` 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 Create a callback pattern example with Amazon SQS, Amazon SNS, and Lambda.

Call HTTPS APIs in Step Functions workflows

An HTTP Task is a type of Task workflow state state that you can use to call HTTPS APIs in your workflows. The API can be public, such as third-party SaaS applications like Stripe or Salesforce. You can also call private API, such as HTTPS-based applications in an Amazon Virtual Private Cloud.

For authorization and network connectivity, an HTTP Task requires an EventBridge connection.

To call an HTTPS API, use the Task state with the `arn:aws:states:::http:invoke` resource. Then, provide the API endpoint configuration details, such as the API URL, method you want to use, and connection details.

If you use Workflow Studio to build your state machine that contains an HTTP Task, Workflow Studio automatically generates an execution role with IAM policies for the HTTP Task. For more information, see Role for testing HTTP Tasks in Workflow Studio.

> **Note**
>
> HTTP Task currently only supports public domain names with publicly trusted certificates for HTTPS endpoints when using private APIs. HTTP Task does not support mutual TLS (mTLS).

Topics

Connectivity for an HTTP Task

An HTTP Task requires an EventBridge connection, which securely manages the authentication credentials of an API provider. A *connection* defines the authorization method and credentials to use in connecting to a given API. If you are connecting to a private API, such as a private API in an Amazon Virtual Private Cloud (Amazon VPC), you can also use the connection to define secure point-to-point network connectivity. Using a connection helps you avoid hard-coding secrets, such as API keys, into your state machine definition. An EventBridge connection supports the Basic, OAuth, and API Key authorization schemes.

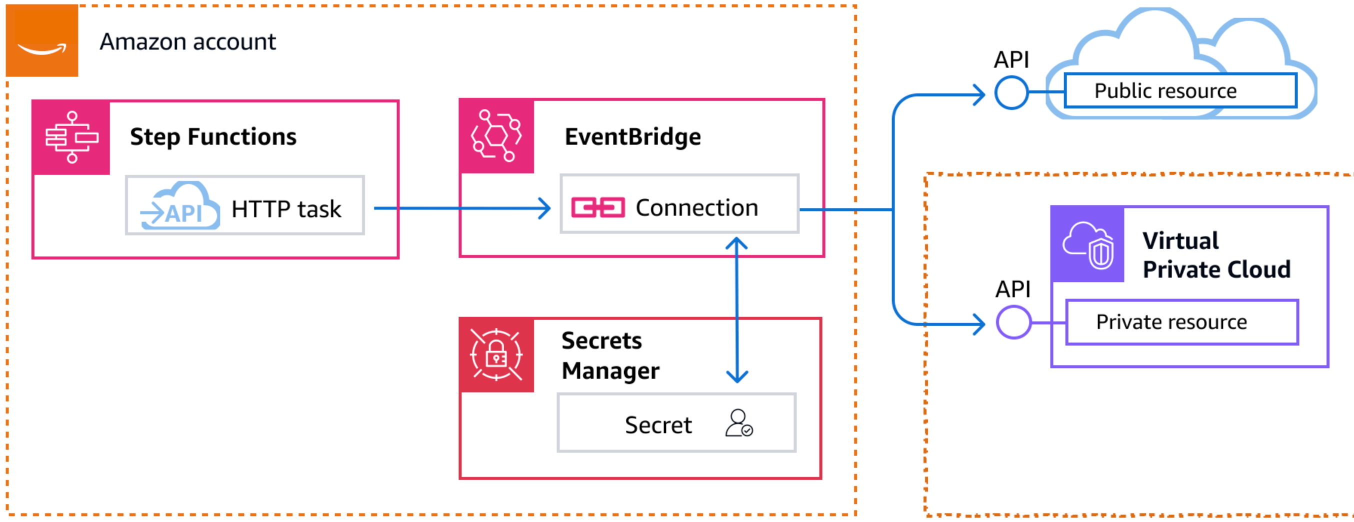
When you create an EventBridge connection, you provide your authorization and network connectivity details. You can also include the header, body, and query parameters that are required for authorization with an API. You must include the connection ARN in any HTTP Task that calls an HTTPS API.

When you create a connection, EventBridge creates a *secret* in AWS Secrets Manager. In this secret, EventBridge stores the connection and authorization parameters in an encrypted form. To successfully create or update a connection, you must use an AWS account that has permission to use Secrets Manager. For more information about the IAM permissions your state machine needs to access an EventBridge connection, see IAM permissions to run an HTTP Task.

The following image shows how Step Functions handles authorization for HTTPS API calls using an EventBridge connection. The EventBridge connection manages credentials of an HTTPS API provider. EventBridge creates a *secret* in Secrets Manager to store the connection and authorization parameters in an encrypted form. In the case of private APIs, EventBridge also stores network connectivity configurations.

Timeouts for connections

HTTP task requests will timeout after 60 seconds.



HTTP Task definition

The ASL definition represents an HTTP Task with `http:invoke` resource. The following HTTP Task definition invokes a public Stripe API that returns a list of all customers.

```
"Call HTTPS API": {
  "Type": "Task",
  "Resource": "arn:aws:states:::http:invoke",
  "Parameters": {
    "ApiEndpoint": "https://api.stripe.com/v1/customers",
    "Authentication": {
      "ConnectionArn": "arn:aws:events:region:account-id:connection/
Stripe/81210c42-8af1-456b-9c4a-6ff02fc664ac"
    },
    "Method": "GET"
  },
  "End": true
}
```

HTTP Task fields

An HTTP Task includes the following fields in its definition.

`Resource` (Required)

To specify a task type, provide its ARN in the `Resource` field. For an HTTP Task, you specify the `Resource` field as follows.

```
"Resource": "arn:aws:states:::http:invoke"
```

`Parameters` (Required)

Contains the `ApiEndpoint`, `Method`, and `ConnectionArn` fields that provide information about the HTTPS API you want to call. `Parameters` also contains optional fields, such as `Headers` and `QueryParameters`.

You can specify a combination of static JSON and JsonPath syntax as `Parameters` in the `Parameters` field. For more information, see Passing parameters to a service API in Step Functions.

To specify the EventBridge connection, use either the `Authentication` *or* `InvocationConfig` field.

`ApiEndpoint` (Required)

Specifies the URL of the HTTPS API you want to call. To append query parameters to the URL, use the `QueryParameters` field. The following example shows how you can call a Stripe API to fetch the list of all customers.

```
"ApiEndpoint":"https://api.stripe.com/v1/customers"
```

You can also specify a reference path using the JsonPath syntax to select the JSON node that contains the HTTPS API URL. For example, say you want to call one of Stripe's APIs using a specific customer ID. Imagine that you've provided the following state input.

```
{
    "customer_id": "1234567890",
    "name": "John Doe"
}
```

To retrieve the details of this customer ID using a Stripe API, specify the `ApiEndpoint` as shown in the following example. This example uses an intrinsic function and a reference path.

```
"ApiEndpoint.$":"States.Format('https://api.stripe.com/v1/customers/{}',
 $.customer_id)"
```

At runtime, Step Functions resolves the value of `ApiEndpoint` as follows.

```
https://api.stripe.com/v1/customers/1234567890
```

Method (Required)

Specifies the HTTP method you want to use for calling an HTTPS API. You can specify one of these methods in your HTTP Task: GET, POST, PUT, DELETE, PATCH, OPTIONS, or HEAD.

For example, to use the GET method, specify the `Method` field as follows.

```
"Method": "GET"
```

You can also use a reference path to specify the method at runtime. For example, **`"Method.$": "$.myHTTPMethod"`**.

Authentication (Conditional)

You must specify either `Authentication` *or* `InvocationConfig`.

Contains the `ConnectionArn` field that specifies how to authenticate a public HTTPS API call. Step Functions supports authentication for a specified `ApiEndpoint` using the connection resource of Amazon EventBridge.

ConnectionArn (Required)

Specifies the EventBridge connection ARN.

An HTTP Task requires an EventBridge connection, which securely manages the authorization credentials of an API provider. A *connection* specifies the authorization type and credentials to use for authorizing an HTTPS API. Using a connection helps you avoid hard-coding secrets, such as API keys, into your state machine definition. In a connection, you can also specify `Headers`, `QueryParameters`, and `RequestBody` parameters.

For more information, see Connectivity for an HTTP Task.

The following example shows how you can specify the `Authentication` field in your HTTP Task definition.

```
"Authentication": {
  "ConnectionArn": "arn:aws:events:us-east-2:account-id:connection/
Stripe/81210c42-8af1-456b-9c4a-6ff02fc664ac"
}
```

`InvocationConfig` (Conditional)

You must specify either `Authentication` *or* `InvocationConfig`.

Contains the authorization and network connectivity configuration for a private HTTPS API call. Step Functions supports connection for a specified `ApiEndpoint` using the connection resource of Amazon EventBridge. For more information, see [Connecting to private APIs](#) in the *Amazon EventBridge User Guide*.

`ConnectionArn` (Required)

Specifies the EventBridge connection ARN.

An HTTP Task requires an [EventBridge connection](#), which securely manages the authorization credentials of an API provider. A *connection* specifies the authorization type and credentials to use for authorizing an HTTPS API. For private APIs, the connection also defines secure point-to-point network connectivity. Using a connection helps you avoid hard-coding secrets, such as API keys, into your state machine definition. In a connection, you can also specify [`Headers`](#), [`QueryParameters`](#), and [`RequestBody`](#) parameters.

For more information, see [Connectivity for an HTTP Task](#).

The following example shows how you can specify an `InvocationConfig` field in your HTTP Task definition.

```
"InvocationConfig": {
  "ConnectionArn": "arn:aws:events:region:account-id:connection/connection-id"
}
```

`Headers` (Optional)

Provides additional context and metadata to the API endpoint. You can specify headers as a string or JSON array.

You can specify headers in the EventBridge connection and the `Headers` field in an HTTP Task. We recommend that you do not include authentication details to your API providers in the `Headers` field. We recommend that you include these details into your EventBridge connection.

Step Functions adds the headers that you specify in the EventBridge connection to the headers that you specify in the HTTP Task definition. If the same header keys are present in the definition and connection, Step Functions uses the corresponding values specified in the EventBridge connection for those headers. For more information about how Step Functions performs data merging, see Merging EventBridge connection and HTTP Task definition data.

The following example specifies a header that will be included in an HTTPS API call: `content-type`.

```
"Headers": {
  "content-type": "application/json"
}
```

You can also use a reference path to specify the headers at runtime. For example, **`"Headers.$": "$.myHTTPHeaders"`**.

Step Functions sets the `User-Agent`, `Range`, and `Host` headers. Step Functions sets the value of the `Host` header based on the API you're calling. The following is an example of these headers.

```
User-Agent: Amazon|StepFunctions|HttpInvoke|region,
Range: bytes=0-262144,
Host: api.stripe.com
```

You can't use the following headers in your HTTP Task definition. If you use these headers, the HTTP Task fails with the `States.Runtime` error.

- A-IM
- Accept-Charset
- Accept-Datetime
- Accept-Encoding
- Authorization
- Cache-Control
- Connection

- Content-Encoding
- Content-MD5
- Date
- Expect
- Forwarded
- From
- Host
- HTTP2-Settings
- If-Match
- If-Modified-Since
- If-None-Match
- If-Range
- If-Unmodified-Since
- Max-Forwards
- Origin
- Pragma
- Proxy-Authorization
- Referer
- Server
- TE
- Trailer
- Transfer-Encoding
- Upgrade
- Via
- Warning
- x-forwarded-*
- x-amz-*
- x-amzn-*

`QueryParameters` (Optional)

Inserts key-value pairs at the end of an API URL. You can specify query parameters as a string, JSON array, or a JSON object. Step Functions automatically URL-encodes query parameters when it calls an HTTPS API.

For example, say that you want to call the Stripe API to search for customers that do their transactions in US dollars (USD). Imagine that you've provided the following `QueryParameters` as state input.

```
"QueryParameters": {
  "currency": "usd"
}
```

At runtime, Step Functions appends the `QueryParameters` to the API URL as follows.

```
https://api.stripe.com/v1/customers/search?currency=usd
```

You can also use a reference path to specify the query parameters at runtime. For example, **`"QueryParameters.$": "$.myQueryParameters"`**.

If you’ve specified query parameters in your EventBridge connection, Step Functions adds these query parameters to the query parameters that you specify in the HTTP Task definition. If the same query parameters keys are present in the definition and connection, Step Functions uses the corresponding values specified in the EventBridge connection for those headers. For more information about how Step Functions performs data merging, see Merging EventBridge connection and HTTP Task definition data.

`Transform` (Optional)

Contains the `RequestBodyEncoding` and `RequestEncodingOptions` fields. By default, Step Functions sends the request body as JSON data to an API endpoint.

If your API provider accepts `form-urlencoded` request bodies, use the `Transform` field to specify URL-encoding for the request bodies. You must also specify the `content-type` header as `application/x-www-form-urlencoded`. Step Functions then automatically URL-encodes your request body.

`RequestBodyEncoding`

Specifies URL-encoding of your request body. You can specify one these values: `NONE` or `URL_ENCODED`.

- `NONE` – The HTTP request body will be the serialized JSON of the `RequestBody` field. This is the default value.
- `URL_ENCODED` – The HTTP request body will be the URL-encoded form data of the `RequestBody` field.

`RequestEncodingOptions`

Determines the encoding option to use for arrays in your request body if you set `RequestBodyEncoding` to `URL_ENCODED`.

Step Functions supports the following array encoding options. For more information about these options and their examples, see Applying URL-encoding on request body.

- `INDICES` – Encodes arrays using the index value of array elements. By default, Step Functions uses this encoding option.
- `REPEAT` – Repeats a key for each item in an array.
- `COMMAS` – Encodes all the values in a key as a comma-delimited list of values.
- `BRACKETS` – Repeats a key for each item in an array and appends a bracket, [], to the key to indicate that it is an array.

The following example sets URL-encoding for the request body data. It also specifies to use the `COMMAS` encoding option for arrays in the request body.

```
"Transform": {
  "RequestBodyEncoding": "URL_ENCODED",
  "RequestEncodingOptions": {
    "ArrayFormat": "COMMAS"
  }
}
```

`RequestBody` (Optional)

Accepts JSON data that you provide in the state input. In `RequestBody`, you can specify a combination of static JSON and JsonPath syntax. For example, say that you provide the following state input:

```
{
    "CardNumber": "1234567890",
    "ExpiryDate": "09/25"
}
```

To use these values of `CardNumber` and `ExpiryDate` in your request body at runtime, you can specify the following JSON data in your request body.

```
"RequestBody": {
  "Card": {
    "Number.$": "$.CardNumber",
    "Expiry.$": "$.ExpiryDate",
    "Name": "John Doe",
    "Address": "123 Any Street, Any Town, USA"
  }
}
```

If the HTTPS API you want to call requires `form-urlencoded` request bodies, you must specify URL-encoding for your request body data. For more information, see Applying URL-encoding on request body.

Merging EventBridge connection and HTTP Task definition data

When you invoke an HTTP Task, you can specify data in your EventBridge connection and your HTTP Task definition. This data includes `Headers`, `QueryParameters`, and `RequestBody` parameters. Before calling an HTTPS API, Step Functions merges the request body with the connection body parameters in all cases except if your request body is a string and the connection body parameters is non-empty. In this case, the HTTP Task fails with the `States.Runtime` error.

If there are any duplicate keys specified in the HTTP Task definition and the EventBridge connection, Step Functions overwrites the values in the HTTP Task with the values in the connection.

The following list describes how Step Functions merges data before calling an HTTPS API:

- **Headers** – Step Functions adds any headers you specified in the connection to the headers in the `Headers` field of the HTTP Task. If there is a conflict between the header keys, Step Functions uses the values specified in the connection for those headers. For example, if you specified the `content-type` header in both the HTTP Task definition and EventBridge connection, Step Functions uses the `content-type` header value specified in the connection.
- **Query parameters** – Step Functions adds any query parameters you specified in the connection to the query parameters in the `QueryParameters` field of the HTTP Task. If there is a conflict between the query parameter keys, Step Functions uses the values specified in the connection for those query parameters. For example, if you specified the `maxItems` query parameter in both

the HTTP Task definition and EventBridge connection, Step Functions uses the `maxItems` query parameter value specified in the connection.

- **Body parameters**
  - Step Functions adds any request body values specified in the connection to the request body in the `RequestBody` field of the HTTP Task. If there is a conflict between the request body keys, Step Functions uses the values specified in the connection for the request body. For example, say that you specified a `Mode` field in the `RequestBody` of both the HTTP Task definition and EventBridge connection. Step Functions uses the `Mode` field value you specified in the connection.
  - If you specify the request body as a string instead of a JSON object, and the EventBridge connection also contains request body, Step Functions can't merge the request body specified in both these places. It fails the HTTP Task with the `States.Runtime` error.

  Step Functions applies all transformations and serializes the request body after it completes the merging of the request body.

The following example sets the `Headers`, `QueryParameters`, and `RequestBody` fields in both the HTTP Task and EventBridge connection.

HTTP Task definition

```
{
  "Comment": "Data merging example for HTTP Task and EventBridge connection",
  "StartAt": "ListCustomers",
  "States": {
    "ListCustomers": {
      "Type": "Task",
      "Resource": "arn:aws:states:::http:invoke",
      "Parameters": {
        "Authentication": {
          "ConnectionArn": "arn:aws:events:region:account-
id:connection/Example/81210c42-8af1-456b-9c4a-6ff02fc664ac"
        },
        "ApiEndpoint": "https:/example.com/path",
        "Method": "GET",
        "Headers": {
          "Request-Id": "my_request_id",
          "Header-Param": "state_machine_header_param"
        },
        "RequestBody": {
```

```
          "Job": "Software Engineer",
          "Company": "AnyCompany",
          "BodyParam": "state_machine_body_param"
        },
        "QueryParameters": {
          "QueryParam": "state_machine_query_param"
        }
      }
    }
  }
}
```

EventBridge connection

```
{
  "AuthorizationType": "API_KEY",
  "AuthParameters": {
    "ApiKeyAuthParameters": {
      "ApiKeyName": "ApiKey",
      "ApiKeyValue": "key_value"
    },
    "InvocationHttpParameters": {
      "BodyParameters": [
        {
          "Key": "BodyParam",
          "Value": "connection_body_param"
        }
      ],
      "HeaderParameters": [
        {
          "Key": "Header-Param",
          "Value": "connection_header_param"
        }
      ],
      "QueryStringParameters": [
        {
          "Key": "QueryParam",
          "Value": "connection_query_param"
        }
      ]
    }
  }
}
```

In this example, duplicate keys are specified in the HTTP Task and EventBridge connection. Therefore, Step Functions overwrites the values in the HTTP Task with the values in the connection. The following code snippet shows the HTTP request that Step Functions sends to the HTTPS API.

```
POST /path?QueryParam=connection_query_param HTTP/1.1
Apikey: key_value
Content-Length: 79
Content-Type: application/json; charset=UTF-8
Header-Param: connection_header_param
Host: example.com
Range: bytes=0-262144
Request-Id: my_request_id
User-Agent: Amazon|StepFunctions|HttpInvoke|region

{"Job":"Software Engineer","Company":"AnyCompany","BodyParam":"connection_body_param"}
```

Applying URL-encoding on request body

By default, Step Functions sends the request body as JSON data to an API endpoint. If your HTTPS API provider requires `form-urlencoded` request bodies, you must specify URL-encoding for the request bodies. Step Functions then automatically URL-encodes the request body based on the URL-encoding option you select.

You specify URL-encoding using the `Transform` field. This field contains the `RequestBodyEncoding` field that specifies whether or not you want to apply URL-encoding for your request bodies. When you specify the `RequestBodyEncoding` field, Step Functions converts your JSON request body to `form-urlencoded` request body before calling the HTTPS API. You must also specify the `content-type` header as `application/x-www-form-urlencoded` because APIs that accept URL-encoded data expect the `content-type` header.

To encode arrays in your request body, Step Functions provides the following array encoding options.

- `INDICES` – Repeats a key for each item in an array and appends a bracket, [], to the key to indicate that it is an array. This bracket contains the index of the array element. Adding the index helps you specify the order of the array elements. By default, Step Functions uses this encoding option.

  For example, if your request body contains the following array.

```
{"array": ["a","b","c","d"]}
```

Step Functions encodes this array to the following string.

```
array[0]=a&array[1]=b&array[2]=c&array[3]=d
```

- `REPEAT` – Repeats a key for each item in an array.

  For example, if your request body contains the following array.

  ```
  {"array": ["a","b","c","d"]}
  ```

  Step Functions encodes this array to the following string.

  ```
  array=a&array=b&array=c&array=d
  ```

- `COMMAS` – Encodes all the values in a key as a comma-delimited list of values.

  For example, if your request body contains the following array.

  ```
  {"array": ["a","b","c","d"]}
  ```

  Step Functions encodes this array to the following string.

  ```
  array=a,b,c,d
  ```

- `BRACKETS` – Repeats a key for each item in an array and appends a bracket, [], to the key to indicate that it is an array.

  For example, if your request body contains the following array.

  ```
  {"array": ["a","b","c","d"]}
  ```

  Step Functions encodes this array to the following string.

  ```
  array[]=a&array[]=b&array[]=c&array[]=d
  ```

IAM permissions to run an HTTP Task

Your state machine execution role must have the following permissions for an HTTP Task to call an HTTPS API:

- `states:InvokeHTTPEndpoint`
- `events:RetrieveConnectionCredentials`
- `secretsmanager:GetSecretValue`
- `secretsmanager:DescribeSecret`

The following IAM policy example grants the least privileges required to your state machine role for calling Stripe APIs. This IAM policy also grants permission to the state machine role to access a specific EventBridge connection, including the secret for this connection that is stored in Secrets Manager.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Statement1",
            "Effect": "Allow",
            "Action": "states:InvokeHTTPEndpoint",
            "Resource": "arn:aws:states:us-
east-2:123456789012:stateMachine:myStateMachine",
            "Condition": {
                "StringEquals": {
                    "states:HTTPMethod": "GET"
                },
                "StringLike": {
                    "states:HTTPEndpoint": "https://api.stripe.com/*"
                }
            }
        },
        {
            "Sid": "Statement2",
            "Effect": "Allow",
            "Action": [
                "events:RetrieveConnectionCredentials"
            ],
```

```
            "Resource": "arn:aws:events:us-
east-2:123456789012:connection/oauth_connection/aeabd89e-d39c-4181-9486-9fe03e6f286a"
        },
        {
            "Sid": "Statement3",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue",
                "secretsmanager:DescribeSecret"
            ],
            "Resource": "arn:aws:secretsmanager:*:*:secret:events!connection/*"
        }
    ]
}
```

HTTP Task example

The following state machine definition shows an HTTP Task that includes the `Headers`, `QueryParameters`, `Transform`, and `RequestBody` parameters. The HTTP Task calls a Stripe API, https://api.stripe.com/v1/invoices, to generate an invoice. The HTTP Task also specifies URL-encoding for the request body using the `INDICES` encoding option.

Make sure that you've created an EventBridge connection. The following example shows a connection created using BASIC auth type.

```
{
    "Type": "BASIC",
    "AuthParameters": {
        "BasicAuthParameters": {
            "Password": "myPassword",
            "Username": "myUsername"
         },
    }
}
```

Remember to replace the *italicized* text with your resource-specific information.

```
{
  "Comment": "A state machine that uses HTTP Task",
  "StartAt": "CreateInvoiceAPI",
  "States": {
    "CreateInvoiceAPI": {
```

```
      "Type": "Task",
      "Resource": "arn:aws:states:::http:invoke",
      "Parameters": {
        "ApiEndpoint": "https://api.stripe.com/v1/invoices",
        "Method": "POST",
        "Authentication": {
          "ConnectionArn": ""arn:aws:events:region:account-id:connection/
Stripe/81210c42-8af1-456b-9c4a-6ff02fc664ac"
        },
        "Headers": {
          "Content-Type": "application/x-www-form-urlencoded"
        },
        "RequestBody": {
          "customer.$": "$.customer_id",
          "description": "Monthly subscription",
          "metadata": {
            "order_details": "monthly report data"
          }
        },
        "Transform": {
          "RequestBodyEncoding": "URL_ENCODED",
          "RequestEncodingOptions": {
            "ArrayFormat": "INDICES"
          }
        }
      },
      "Retry": [
        {
          "ErrorEquals": [
            "States.Http.StatusCode.429",
            "States.Http.StatusCode.503",
            "States.Http.StatusCode.504",
            "States.Http.StatusCode.502"
          ],
          "BackoffRate": 2,
          "IntervalSeconds": 1,
          "MaxAttempts": 3,
          "JitterStrategy": "FULL"
        }
      ],
      "Catch": [
        {
          "ErrorEquals": [
            "States.Http.StatusCode.404",
```

```
            "States.Http.StatusCode.400",
            "States.Http.StatusCode.401",
            "States.Http.StatusCode.409",
            "States.Http.StatusCode.500"
          ],
          "Comment": "Handle all non 200 ",
          "Next": "HandleInvoiceFailure"
        }
      ],
      "End": true
    }
  }
}
```

To run this state machine, provide the customer ID as the input as shown in the following example:

```
{
    "customer_id": "1234567890"
}
```

The following example shows the HTTP request that Step Functions sends to the Stripe API.

```
POST /v1/invoices HTTP/1.1
Authorization: Basic <base64 of username and password>
Content-Type: application/x-www-form-urlencoded
Host: api.stripe.com
Range: bytes=0-262144
Transfer-Encoding: chunked
User-Agent: Amazon|StepFunctions|HttpInvoke|region

description=Monthly%20subscription&metadata%5Border_details%5D=monthly%20report
%20data&customer=1234567890
```

Testing an HTTP Task

You can use the TestState API through the console, SDK, or the AWS CLI to test an HTTP Task. The following procedure describes how to use the TestState API in the Step Functions console. You can iteratively test the API request, response, and authentication details until your HTTP Task is working as expected.

Test an HTTP Task state in Step Functions console

1. Open the Step Functions console.
2. Choose **Create state machine** to start creating a state machine or choose an existing state machine that contains an HTTP Task.

   Refer Step 4 if you're testing the task in an existing state machine.
3. In the Design mode of Workflow Studio, configure an HTTP Task visually. Or choose the Code mode to copy-paste the state machine definition from your local development environment.
4. In Design mode, choose **Test state** in the Inspector panel panel of Workflow Studio.
5. In the **Test state** dialog box, do the following:

   a. For **Execution role**, choose an execution role to test the state. If you don't have a role with sufficient permissions for an HTTP Task, see Role for testing HTTP Tasks in Workflow Studio to create a role.

   b. (Optional) Provide any JSON input that your selected state needs for the test.

   c. For **Inspection level**, keep the default selection of **INFO**. This level shows you the status of the API call and the state output. This is useful for quickly checking the API response.

   d. Choose **Start test**.

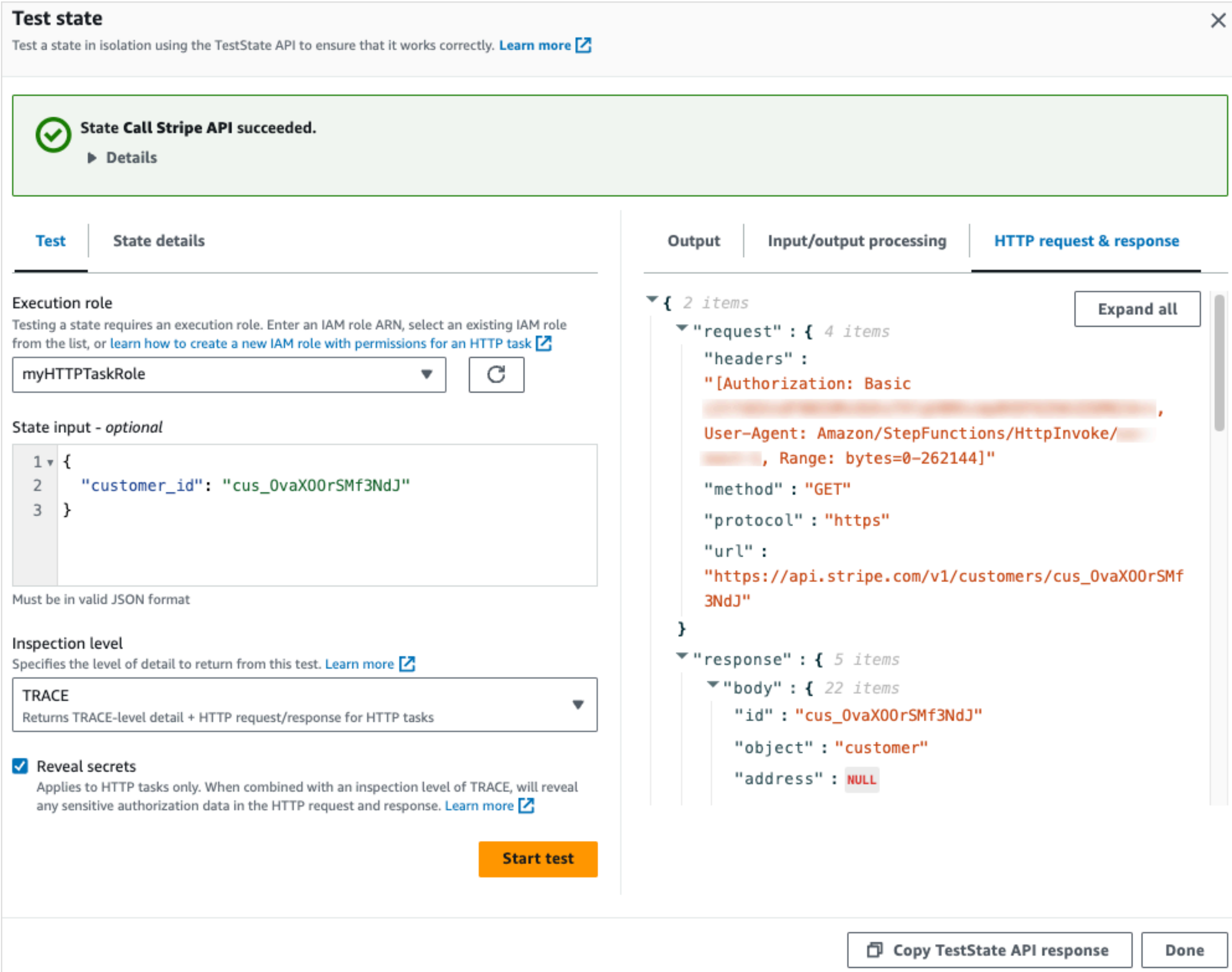
   e. If the test succeeds, the state output appears on the right side of the **Test state** dialog box. If the test fails, an error appears.

      In the **State details** tab of the dialog box, you can see the state definition and a link to your EventBridge connection.

   f. Change the **Inspection level** to **TRACE**. This level shows you the raw HTTP request and response, and is useful for verifying headers, query parameters, and other API-specific details.

   g. Choose the **Reveal secrets** checkbox. In combination with **TRACE**, this setting lets you see the sensitive data that the EventBridge connection inserts, such as API keys. The IAM user identity that you use to access the console must have permission to perform the `states:RevealSecrets` action. Without this permission, Step Functions throws an access denied error when you start the test. For an example of an IAM policy that sets the `states:RevealSecrets` permission, see IAM permissions for using TestState API.

The following image shows a test for an HTTP Task that succeeds. The **Inspection level** for this state is set to **TRACE**. The **HTTP request & response** tab in the following image shows the result of the HTTPS API call.



h. Choose **Start test**.

i. If the test succeeds, you can see your HTTP details under the **HTTP request & response** tab.

Unsupported HTTP Task responses

An HTTP Task fails with the `States.Runtime` error if one of the following conditions is true for the response returned:

- The response contains a content-type header of `application/octet-stream`, `image/*`, `video/*`, or `audio/*`.
- The response can't be read as a valid string. For example, binary or image data.

Connection errors

If EventBridge encounters an issue when connecting to the specified API during workflow execution, Step Functions raises the error in your workflow. Connection errors are prefixed with `Events.ConnectionResource.`.

These errors include:

- `Events.ConnectionResource.InvalidConnectionState`
- `Events.ConnectionResource.InvalidPrivateConnectionState`
- `Events.ConnectionResource.AccessDenied`
- `Events.ConnectionResource.ResourceNotFound`
- `Events.ConnectionResource.AuthInProgress`
- `Events.ConnectionResource.ConcurrentModification`
- `Events.ConnectionResource.InternalError`

Passing parameters to a service API in Step Functions

> **Managing state and transforming data**
>
> Learn about Passing data between states with variables and Transforming data with JSONata.

Use the `Parameters` field in a `Task` state to control what parameters are passed to a service API.

Inside the `Parameters` field, you must use the plural form of the array parameters in an API action. For example, if you use the Filter field of the `DescribeSnapshots` API action for integrating with Amazon EC2, you must define the field as `Filters`. If you don't use the plural form, Step Functions returns the following error:

```
The field Filter is not supported by Step Functions.
```

Pass static JSON as parameters

You can include a JSON object directly in your state machine definition to pass as a parameter to a resource.

For example, to set the `RetryStrategy` parameter for the `SubmitJob` API for AWS Batch, you could include the following in your parameters.

```
"RetryStrategy": {
  "attempts": 5
}
```

You can also pass multiple parameters with static JSON. As a more complete example, the following are the `Resource` and `Parameters` fields of the specification of a task that publishes to an Amazon SNS topic named *`myTopic`*.

```
"Resource": "arn:aws:states:::sns:publish",
  "Parameters": {
     "TopicArn": "arn:aws:sns:us-east-2:account-id:myTopic",
     "Message": "test message",
     "MessageAttributes": {
       "my attribute no 1": {
         "DataType": "String",
         "StringValue": "value of my attribute no 1"
       },
       "my attribute no 2": {
         "DataType": "String",
         "StringValue": "value of my attribute no 2"
       }
     }
  },
```

Pass state input as parameters using Paths

You can pass portions of the state input as parameters by using [paths](). 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, end the parameter name with `.$`. For example, if your state input contains text within a node named `message`, you could pass that text as a parameter using a path.

Consider the following state input:

```
{
  "comment": "A message in the state input",
  "input": {
    "message": "foo",
    "otherInfo": "bar"
  },
  "data": "example"
}
```

To pass the value of the node named `message` as a parameter named `myMessage`, specify the following syntax:

```
"Parameters": {"myMessage.$": "$.input.message"},
```

Step Functions then passes the value `foo` as a parameter.

For more information about using parameters in Step Functions, see the following:

- Processing input and output
- Manipulate parameters in Step Functions workflows

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:

- Accessing execution data from the Context object in Step Functions
- Accessing the Context object
- Get a Token from the Context object

Integrating optimized services with Step Functions

Your workflow can call optimized services directly using the `Resource` field of a `Task` state. The following topics explain the supported APIs, parameters, and request/response syntax in Amazon States Language for coordinating AWS services.

Depending on workflow type and availability, your workflows call services using one of three service integration patterns:

- Request a Response (default) - wait for HTTP response, then go to the next state
- Run a Job (`.sync`) - wait for the job to complete
- Wait for Callback (`.waitForTaskToken`) - pause a workflow until a task token is returned

Standard Workflows and Express Workflows support the same **integrations** but not the same **integration patterns**.

- **Standard Workflows** support *Request Response* integrations. Certain services support *Run a Job (.sync)*, or *Wait for Callback (.waitForTaskToken)* , and both in some cases. See the following optimized integrations table for details.
- **Express Workflows** only support *Request Response* integrations.

To help decide between the two types, see Choosing workflow type in Step Functions.

AWS SDK integrations in Step Functions

Integrated service	Request Response	Run a Job - *.sync*	Wait for Callback - *.waitForTaskToken*
Over two hundred services	Standard & Express	*Not supported*	Standard

Optimized integrations in Step Functions

Integrated service	Request Response	Run a Job - *.sync*	Wait for Callback - *.waitForTaskToken*
Amazon API Gateway	Standard & Express	*Not supported*	Standard
Amazon Athena	Standard & Express	Standard	*Not supported*
AWS Batch	Standard & Express	Standard	*Not supported*
Amazon Bedrock	Standard & Express	Standard	Standard
AWS CodeBuild	Standard & Express	Standard	*Not supported*
Amazon DynamoDB	Standard & Express	*Not supported*	*Not supported*
Amazon ECS/Fargate	Standard & Express	Standard	Standard
Amazon EKS	Standard & Express	Standard	Standard
Amazon EMR	Standard & Express	Standard	*Not supported*
Amazon EMR on EKS	Standard & Express	Standard	*Not supported*
Amazon EMR Serverless	Standard & Express	Standard	*Not supported*
Amazon EventBridge	Standard & Express	*Not supported*	Standard
AWS Glue	Standard & Express	Standard	*Not supported*
AWS Glue DataBrew	Standard & Express	Standard	*Not supported*
AWS Lambda	Standard & Express	*Not supported*	Standard
AWS Elemental MediaConvert	Standard & Express	Standard	*Not supported*
Amazon SageMaker AI	Standard & Express	Standard	*Not supported*
Amazon SNS	Standard & Express	*Not supported*	Standard

Integrated service	Request Response	Run a Job - *.sync*	Wait for Callback - *.waitForTaskToken*
Amazon SQS	Standard & Express	*Not supported*	Standard
AWS Step Functions	Standard & Express	Standard	Standard

Create API Gateway REST APIs with Step Functions

Learn how to use Amazon API Gateway to create, publish, maintain, and monitor HTTP and REST APIs with Step Functions. 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.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of Optimized API Gateway integration

- `apigateway:invoke:` has no equivalent in the AWS SDK service integration. Instead, the Optimized API Gateway service calls your API Gateway endpoint directly.

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. The Step Functions API Gateway REST API integration that provides the option for Edge-optimized APIs are not supported.
- 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`
  - `PATCH`
  - `HEAD`
  - `OPTIONS`

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 only 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` – default value: `false`
  - With the default setting, any **null** values in the request input state will **not** be sent to your API. In the following example, the `category` field will **not** be included in the request, unless `AllowNullValues` is set to `true` in your state machine definition.

```
{
  "NewPet": {
```

```
        "type": "turtle",
        "price": 123,
        "category": null
    }
}
```

> **Note**
>
> By default, fields with **null** values in the request input state will **not** be sent to your API. You can force null values to be sent to your API by setting `AllowNullValues` to `true` in your state machine definition.

- `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`
> - `Content-md5`
> - `Expect`
> - `Host`
> - `Max-Forwards`
> - `Proxy-Authenticate`
> - `Server`
> - `TE`

- `Transfer-Encoding`
- `Trailer`
- `Upgrade`
- `Via`
- `Www-Authenticate`

The following code example shows how to invoke API Gateway using Step Functions.

```
{
    "Type": "Task",
    "Resource":"arn:aws:states:::apigateway:invoke",
    "Arguments": {
        "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.

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, 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 Task Token (`.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:

```
{
    "Type": "Task",
    "Resource":"arn:aws:states:::apigateway:invoke.waitForTaskToken",
    "Arguments": {
        "ApiEndpoint": "example.execute-api.us-east-1.amazonaws.com",
        "Method": "POST",
        "Headers": {
            "TaskToken": "{% $states.context.Task.Token %}"
        },
        "Stage": "prod",
        "Path": "bills/add",
        "QueryParameters": {},
        "RequestBody": {
            "billId": "my-new-bill"
        },
        "AuthType": "IAM_ROLE"
    }
}
```

Output format

The following output parameters are provided:

Name	Type	Description
`ResponseBody`	`JSON` or `String`	The response body of the API call.
`Headers`	`JSON`	The response headers.
`StatusCode`	`Integer`	The HTTP status code of the response.
`StatusText`	`String`	The status text of the response.

An example response:

```
{
    "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:

```
{
    "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.

IAM policies for calls to Amazon API Gateway

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

Resources:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "execute-api:Invoke"
            ],
            "Resource": [
                "arn:aws:execute-api:us-east-1:123456789012:ENDPOINT/STAGE/GET/pets",
                "arn:aws:execute-api:us-east-1:123456789012:ENDPOINT/STAGE/POST/pets"
            ],
            "Effect": "Allow"
        }
    ]
}
```

The following code example shows a resource policy for calling API Gateway.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "states.amazonaws.com"
            },
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:us-east-1:123456789012:myApi-id/stage-name/HTTP-VERB/resource-path-specifier",
            "Condition": {
                "StringEquals": {
                    "aws:SourceArn": [
                        "<SourceStateMachineArn>"
                    ]
                }
            }
        }
    ]
```

```
}
```

Run Athena queries with Step Functions

You can integrate AWS Step Functions with Amazon Athena to start and stop query execution and get query results with Step Functions. 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. This page lists the supported Athena APIs and provides an example `Task` state to start an Athena query.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of Optimized Athena integration

- The Run a Job (.sync) integration pattern is supported.
- There are no specific optimizations for the Request Response integration pattern.
- The Wait for a Callback with Task Token integration pattern is not supported.

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.

API	Request Response	Run a Job (.sync)
`StartQueryExecution`	Supported	Supported
`StopQueryExecution`	Supported	*Not supported*
`GetQueryExecution`	Supported	*Not supported*
`GetQueryResults`	Supported	*Not supported*

The following includes a Task state that starts an Athena query.

```
"Start an Athena query": {
  "Type": "Task",
  "Resource": "arn:aws:states:::athena:startQueryExecution.sync",
  "Arguments": {
    "QueryString": "SELECT * FROM \"myDatabase\".\"myTable\" limit 1",
    "WorkGroup": "primary",
    "ResultConfiguration": {
       "OutputLocation": "s3://amzn-s3-demo-bucket"
    }
  },
  "Next": "Get results of the query"
}
```

Optimized Amazon Athena APIs:

- `StartQueryExecution`
- `StopQueryExecution`
- `GetQueryExecution`
- `GetQueryResults`

Quota for input or result data

When sending or receiving data between services, the maximum input or result for a task is 256 KiB of data as a UTF-8 encoded string. See Quotas related to state machine executions.

IAM policies for calling Amazon Athena

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

> **Note**
>
> In addition to IAM policies, you might need to use AWS Lake Formation to grant access to data in services, such as Amazon S3 and the AWS Glue Data Catalog. For more information, see [Use Athena to query data registered with AWS Lake Formation](#).

StartQueryExecution

Static resources

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement":[
    {
        "Effect": "Allow",
        "Action": [
            "athena:startQueryExecution",
            "athena:stopQueryExecution",
            "athena:getQueryExecution",
            "athena:getDataCatalog",
            "athena:GetWorkGroup",
            "athena:BatchGetQueryExecution",
            "athena:GetQueryResults",
            "athena:ListQueryExecutions"
        ],
        "Resource": [
            "arn:aws:athena:us-east-1:123456789012:workgroup/myWorkGroup",
            "arn:aws:athena:us-east-1:123456789012: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-1:123456789012:catalog",
                "arn:aws:glue:us-east-1:123456789012:database/*",
                "arn:aws:glue:us-east-1:123456789012:table/*",
                "arn:aws:glue:us-east-1:123456789012: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:us-east-1:123456789012:workgroup/myWorkGroup",
            "arn:aws:athena:us-east-1:123456789012: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-1:123456789012:catalog",
            "arn:aws:glue:us-east-1:123456789012:database/*",
            "arn:aws:glue:us-east-1:123456789012:table/*",
            "arn:aws:glue:us-east-1:123456789012:userDefinedFunction/*"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "lakeformation:GetDataAccess"
        ],
        "Resource": [
            "*"
        ]
    }
]
}
```

Dynamic resources

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
```

```
    "Statement":[
    {
        "Effect": "Allow",
        "Action": [
            "athena:startQueryExecution",
            "athena:stopQueryExecution",
            "athena:getQueryExecution",
            "athena:getDataCatalog",
            "athena:GetWorkGroup",
            "athena:BatchGetQueryExecution",
            "athena:GetQueryResults",
            "athena:ListQueryExecutions"
        ],
        "Resource": [
            "arn:aws:athena:us-east-1:123456789012:workgroup/*",
            "arn:aws:athena:us-east-1:123456789012: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-1:123456789012:catalog",
                "arn:aws:glue:us-east-1:123456789012:database/*",
                "arn:aws:glue:us-east-1:123456789012:table/*",
                "arn:aws:glue:us-east-1:123456789012: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:us-east-1:123456789012:workgroup/*",
                "arn:aws:athena:us-east-1:123456789012: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-1:123456789012:catalog",
                "arn:aws:glue:us-east-1:123456789012:database/*",
                "arn:aws:glue:us-east-1:123456789012:table/*",
                "arn:aws:glue:us-east-1:123456789012:userDefinedFunction/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "lakeformation:GetDataAccess"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

StopQueryExecution

Resources

```
{
    "Version": "2012-10-17",
    "Statement":[
    {
        "Effect": "Allow",
        "Action": [
            "athena:stopQueryExecution"
        ],
        "Resource": [
            "arn:aws:athena:us-east-1:123456789012:workgroup/*"
        ]
    }
    ]
}
```

GetQueryExecution

Resources

```
{
    "Version": "2012-10-17",
    "Statement":[
    {
        "Effect": "Allow",
        "Action": [
            "athena:getQueryExecution"
        ],
        "Resource": [
            "arn:aws:athena:us-east-1:123456789012:workgroup/*"
        ]
    }
    ]
}
```

GetQueryResults

Resources

```
{
    "Version": "2012-10-17",
    "Statement":[
    {
        "Effect": "Allow",
        "Action": [
            "athena:getQueryResults"
        ],
        "Resource": [
            "arn:aws:athena:us-east-1:123456789012:workgroup/*"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "s3:GetObject"
        ],
        "Resource": [
            "arn:aws:s3:::*"
        ]
    }
    ]
}
```

Run AWS Batch workloads with Step Functions

You can integrate Step Functions with AWS Batch to run batch computing workloads in the AWS cloud. This page lists the supported AWS Batch APIs and provides an example `Task` state to perform a batch-processing task.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

> **Key features of Optimized AWS Batch integration**
>
> - The Run a Job (.sync) integration pattern is available.
>
> Note that there are no specific optimizations for the Request Response or Wait for a Callback with Task Token integration patterns.

The following shows an example `Task` state that submits an AWS Batch job and waits for it to complete. Many of the arguments shown are optional.

```
"Submit Batch Job": {
    "Type": "Task",
    "Resource": "arn:aws:states:::batch:submitJob.sync",
    "Arguments": {
        "JobName": "BATCH_NAME",
        "JobQueue": "BATCH_QUEUE_ARN",
        "JobDefinition": "BATCH_JOB_DEFINITION_ARN",
        "ArrayProperties": {
        "Size": 10
        },
        "ContainerOverrides": {
        "ResourceRequirements": [
            {
            "Type": "VCPU",
            "Value": "4"
            }
        ]
        },
        "DependsOn": [
        {
```

```
            "JobId": "myJobId",
            "Type": "SEQUENTIAL"
        }
        ],
        "PropagateTags": true,
        "Arguments": {
        "Key1": "value1",
        "Key2": 100
        },
        "RetryStrategy": {
        "Attempts": 1
        },
        "Tags": {
        "Tag": "TAG"
        },
        "Timeout": {
        "AttemptDurationSeconds": 10
        }
    }
}
```

Optimized AWS Batch APIs:

- [SubmitJob]

> **Parameters in Step Functions are expressed in PascalCase**
>
> Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

IAM policies for calling AWS Batch

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

Because job ids for `SubmitJob` and `TerminateJob` are generated and therefore only known at runtime, you cannot create a policy that restricts access based on a specific resource.

> **Tip for fine grained access**
>
> To add fine grained access to `SubmitJob` and `TerminateJob`, consider using tags for jobs and creating a policy that limits access based on your tags. In addition, the job queue, definition, and consumable resources can be restricted for `SubmitJob` using known resources.

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "batch:SubmitJob",
                "batch:DescribeJobs",
                "batch:TerminateJob"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
               "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventsForBatchJobsRule"
            ]
        }
    ]
}
```

Request Response

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "batch:SubmitJob"
            ],
            "Resource": "*"
        }
    ]
}
```

Invoke and customize Amazon Bedrock models with Step Functions

You can integrate Step Functions with Amazon Bedrock to invoke a specified Amazon Bedrock model and create a fine-tuning job to customize a model. This page lists the optimized Amazon Bedrock APIs and provides an example `Task` state to extract the result of a model invocation.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

> **Tip**
>
> To deploy an example workflow that integrates with Amazon Bedrock, see Perform AI prompt-chaining with Amazon Bedrock.

Amazon Bedrock service integration APIs

To integrate AWS Step Functions with Amazon Bedrock, you can use the following APIs. These APIs are similar to the corresponding Amazon Bedrock APIs, except *InvokeModel* has additional request fields.

Amazon Bedrock API - CreateModelCustomizationJob

Creates a fine-tuning job to customize a base model. You can invoke the Step Functions integration API with **CreateModelCustomizationJob** for *Request Response*, or **CreateModelCustomizationJob.sync** for *Run a Job (.sync)* integration patterns. There are no differences in the fields for the API calls.

Amazon Bedrock API - InvokeModel

Invokes the specified Amazon Bedrock model to run inference using the input you provide in the request body. You use `InvokeModel` to run inference for text models, image models, and embedding models.

The Amazon Bedrock service integration API request body for *InvokeModel* includes the following additional parameters.

- `Body` – Specifies input data in the format specified in the content-type request header. `Body` contains parameters specific to the target model.

  If you use the `InvokeModel` API, you must specify the `Body` parameter. Step Functions doesn't validate the input you provide in `Body`.

  When you specify `Body` using the Amazon Bedrock optimized integration, you can specify a payload of up to 256 KiB. If your payload exceeds 256 KiB, we recommend that you use `Input`.
- `Input` – Specifies the source to retrieve the input data from. This optional field is specific to Amazon Bedrock optimized integration with Step Functions. In this field, you can specify an `S3Uri`.

  You can specify either `Body` in the Parameters or `Input`, but not both.

  When you specify `Input` without specifying `ContentType`, the content type of the input data source becomes the value for `ContentType`.
- `Output` – Specifies the destination where the API response is written. This optional field is specific to Amazon Bedrock optimized integration with Step Functions. In this field, you can specify an `S3Uri`.

  If you specify this field, the API response body is replaced with a reference to the Amazon S3 location of the original output.

The following example shows the syntax for InvokeModel API for Amazon Bedrock integration.

```
{
    "ModelId": String,  // required
    "Accept": String,  // default: application/json
    "ContentType": String,  // default: application/json
    "Input": {  // not from Bedrock API
        "S3Uri": String
    },
    "Output": {  // not from Bedrock API
        "S3Uri": String
    }
}
```

Task state definition for Amazon Bedrock integration

The following Task state definition shows how you can integrate with Amazon Bedrock in your state machines. This example shows a Task state that extracts the full result of model invocation specified by the path, `result_one`. This is based on [Inference parameters for foundation models](). This example uses the Cohere Command large language model (LLM).

```
{
  "Type": "Task",
  "Resource": "arn:aws:states:::bedrock:invokeModel",
  "Arguments": {
    "ModelId": "cohere.command-text-v14",
    "Body": {
      "prompt": "{% states.input.prompt_one %}",
      "max_tokens": 20
    },
    "ContentType": "application/json",
    "Accept": "*/*"
  },
  "End": true
}
```

IAM policies for calling Amazon Bedrock

When you create a state machine using the console, Step Functions automatically creates an execution role for your state machine with the least privileges required. These automatically generated IAM roles are valid for the AWS Region in which you create the state machine.

We recommend that when you create IAM policies, do not include wildcards in the policies. As a security best practice, you should scope your policies down as much as possible. You should use dynamic policies only when certain input parameters are not known during runtime.

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

IAM policy examples for Amazon Bedrock integration

The following section describes the IAM permissions you need based on the Amazon Bedrock API that you use for a specific foundation or provisioned model. This section also contains examples of policies that grant full access.

Remember to replace the *italicized* text with your resource-specific information.

- IAM policy example to access a specific foundation model using InvokeModel
- IAM policy example to access a specific provisioned model using InvokeModel
- Full access IAM policy example to use InvokeModel
- IAM policy example to access a specific foundation model as a base model
- IAM policy example to access a specific custom model as a base model
- Full access IAM policy example to use CreateModelCustomizationJob.sync
- IAM policy example to access a specific foundation model using CreateModelCustomizationJob.sync
- IAM policy example to access a custom model using CreateModelCustomizationJob.sync
- Full access IAM policy example to use CreateModelCustomizationJob.sync

IAM policy example to access a specific foundation model using InvokeModel

The following is an IAM policy example for a state machine that accesses a specific foundation model named `amazon.titan-text-express-v1` using the InvokeModel API action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Sid": "InvokeModel1",
            "Action": [
                "bedrock:InvokeModel"
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1::foundation-model/amazon.titan-text-express-
v1"
            ]
        }
    ]
}
```

IAM policy example to access a specific provisioned model using InvokeModel

The following is an IAM policy example for a state machine that accesses a specific provisioned model named `c2oi931ulksx` using the InvokeModel API action.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Sid": "InvokeModel1",
      "Action": [
        "bedrock:InvokeModel"
      ],
      "Resource": [
        "arn:aws:bedrock:us-east-1:123456789012:provisioned-model/c2oi931ulksx"
      ]
    }
  ]
}
```

Full access IAM policy example to use InvokeModel

The following is an IAM policy example for a state machine that provides full access when you use the InvokeModel API action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Sid": "InvokeModel1",
            "Action": [
                "bedrock:InvokeModel"
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1::foundation-model/*",
                "arn:aws:bedrock:us-east-1:123456789012:provisioned-model/*"
            ]
        }
    ]
}
```

IAM policy example to access a specific foundation model as a base model

The following is an IAM policy example for a state machine to access a specific foundation model named `amazon.titan-text-express-v1` as a base model using the CreateModelCustomizationJob API action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob1",
            "Action": [
                "bedrock:CreateModelCustomizationJob"
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1::foundation-model/amazon.titan-text-express-
v1",
                "arn:aws:bedrock:us-east-1:123456789012:custom-model/*",
                "arn:aws:bedrock:us-east-1:123456789012:model-customization-job/*"
            ]
        },
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob2",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::123456789012:role/myRole"
            ]
```

```
        }
    ]
}
```

IAM policy example to access a specific custom model as a base model

The following is an IAM policy example for a state machine to access a specific custom model as a base model using the CreateModelCustomizationJob API action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob1",
            "Action": [
                "bedrock:CreateModelCustomizationJob"
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1:123456789012:custom-model/*",
                "arn:aws:bedrock:us-east-1:123456789012:model-customization-job/*"
            ]
        },
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob2",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::123456789012:role/myRoleName"
            ]
        }
    ]
}
```

Full access IAM policy example to use CreateModelCustomizationJob.sync

The following is an IAM policy example for a state machine that provides full access when you use the CreateModelCustomizationJob API action.

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob1",
            "Action": [
                "bedrock:CreateModelCustomizationJob"
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1::foundation-model/*",
                "arn:aws:bedrock:us-east-1:123456789012:custom-model/*",
                "arn:aws:bedrock:us-east-1:123456789012:model-customization-job/*"
            ]
        },
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob2",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::123456789012:role/myRole"
            ]
        }
    ]
}
```

IAM policy example to access a specific foundation model using CreateModelCustomizationJob.sync

The following is an IAM policy example for a state machine to access a specific foundation model named `amazon.titan-text-express-v1` using the [CreateModelCustomizationJob.sync](#) API action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob1",
            "Action": [
                "bedrock:CreateModelCustomizationJob"
            ],
            "Resource": [
```

```
                "arn:aws:bedrock:us-east-1::foundation-model/amazon.titan-text-express-
v1",
                "arn:aws:bedrock:us-east-1:123456789012:custom-model/*",
                "arn:aws:bedrock:us-east-1:123456789012:model-customization-job/*"
            ]
        },
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob2",
            "Action": [
                "bedrock:GetModelCustomizationJob",
                "bedrock:StopModelCustomizationJob"
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1:123456789012:model-customization-job/*"
            ]
        },
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob3",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::123456789012:role/myRole"
            ]
        }
    ]
}
```

IAM policy example to access a custom model using CreateModelCustomizationJob.sync

The following is an IAM policy example for a state machine to access a custom model using the CreateModelCustomizationJob.sync API action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob1",
            "Action": [
                "bedrock:CreateModelCustomizationJob"
```

```
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1:123456789012:custom-model/*",
                "arn:aws:bedrock:us-east-1:123456789012:model-customization-job/*"
            ]
        },
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob2",
            "Action": [
                "bedrock:GetModelCustomizationJob",
                "bedrock:StopModelCustomizationJob"
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1:123456789012:model-customization-job/*"
            ]
        },
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob3",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::123456789012:role/myRole"
            ]
        }
    ]
}
```

Full access IAM policy example to use CreateModelCustomizationJob.sync

The following is an IAM policy example for a state machine that provides full access when you use the CreateModelCustomizationJob.sync API action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob1",
            "Action": [
                "bedrock:CreateModelCustomizationJob"
```

```
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1::foundation-model/*",
                "arn:aws:bedrock:us-east-1:123456789012:custom-model/*",
                "arn:aws:bedrock:us-east-1:123456789012:model-customization-job/*"
            ]
        },
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob2",
            "Action": [
                "bedrock:GetModelCustomizationJob",
                "bedrock:StopModelCustomizationJob"
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1:123456789012:model-customization-job/*"
            ]
        },
        {
            "Effect": "Allow",
            "Sid": "CreateModelCustomizationJob3",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::123456789012:role/myRole"
            ]
        }
    ]
}
```

Manage AWS CodeBuild builds with Step Functions

You can integrate Step Functions with AWS CodeBuild to start, stop, and manage builds. This page lists the supported CodeBuild APIs you can use with Step Functions.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

With the Step Functions integration with AWS CodeBuild you can 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.

Key features of Optimized CodeBuild integration

- The Run a Job (.sync) 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.

Not all APIs support all integration patterns, as shown in the following table.

API	Request Response	Run a Job (.sync)
StartBuild	Supported	Supported
StopBuild	Supported	*Not supported*
BatchDeleteBuilds	Supported	*Not supported*
BatchGetReports	Supported	*Not supported*
StartBuildBatch	Supported	Supported
StopBuildBatch	Supported	*Not supported*
RetryBuildBatch	Supported	Supported
DeleteBuildBatch	Supported	*Not supported*

Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

Optimized CodeBuild APIs

- `StartBuild`
- `StopBuild`
- `BatchDeleteBuilds`
- `BatchGetReports`
- `StartBuildBatch`
- `StopBuildBatch`
- `RetryBuildBatch`
- `DeleteBuildBatch`

Note

When using JSONPath, you can use the recursive descent operator (`..`) to provide parameters for `BatchDeleteBuilds`. With the returned array, you can transform the `Arn` field from `StartBuild` into a plural `Ids` parameter, as shown in the following example.

```
"BatchDeleteBuilds": {
    "Type": "Task",
    "Resource": "arn:aws:states:::codebuild:batchDeleteBuilds",
    "Arguments": {
        "Ids.$": "$.Build..Arn"
    },
    "Next": "MyNextState"
},
```

IAM policies for calling AWS CodeBuild

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

Resources:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "sns:Publish"
            ],
            "Resource": [
                "arn:aws:sns:sa-east-1:123456789012:StepFunctionsSample-
CodeBuildExecution1111-2222-3333-wJalrXUtnFEMI-SNSTopic-bPxRfiCYEXAMPLEKEY"
            ],
            "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:sa-east-1:123456789012:rule/
StepFunctionsGetEventForCodeBuildStartBuildRule"
            ],
            "Effect": "Allow"
```

```
        }
    ]
}
```

StartBuild

Static resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuild",
        "codebuild:StopBuild",
        "codebuild:BatchGetBuilds"
      ],
      "Resource": [
        "arn:aws:codebuild:us-east-1:123456789012:project/myProjectName"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventForCodeBuildStartBuildRule"
      ]
    }
  ]
}
```

Request Response

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuild"
      ],
      "Resource": [
        "arn:aws:codebuild:us-east-1:123456789012:project/myProjectName"
      ]
    }
  ]
}
```

Dynamic resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuild",
        "codebuild:StopBuild",
        "codebuild:BatchGetBuilds"
      ],
      "Resource": [
        "arn:aws:codebuild:us-east-1:*:project/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
```

```
        "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventForCodeBuildStartBuildRule"
      ]
    }
  ]
}
```

Request Response

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StartBuild"
      ],
      "Resource": [
        "arn:aws:codebuild:us-east-1:*:project/*"
      ]
    }
  ]
}
```

StopBuild

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StopBuild"
      ],
      "Resource": [
        "arn:aws:codebuild:us-east-1:123456789012:project/myProjectName"
      ]
    }
  ]
```

```
}
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:StopBuild"
      ],
      "Resource": [
        "arn:aws:codebuild:us-east-1:*:project/*"
      ]
    }
  ]
}
```

BatchDeleteBuilds

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:BatchDeleteBuilds"
      ],
      "Resource": [
        "arn:aws:codebuild:us-east-1:123456789012:project/myProjectName"
      ]
    }
  ]
}
```

Dynamic resources

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:BatchDeleteBuilds"
      ],
      "Resource": [
        "arn:aws:codebuild:us-east-1:*:project/*"
      ]
    }
  ]
}
```

BatchGetReports

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:BatchGetReports"
      ],
      "Resource": [
        "arn:aws:codebuild:us-east-1:123456789012:report-group/myReportName"
      ]
    }
  ]
}
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "codebuild:BatchGetReports"
```

```
      ],
      "Resource": [
        "arn:aws:codebuild:us-east-1:*:report-group/*"
      ]
    }
  ]
}
```

StartBuildBatch

Static resources

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:StartBuildBatch",
                "codebuild:StopBuildBatch",
                "codebuild:BatchGetBuildBatches"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/myProjectName"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventForCodeBuildStartBuildBatchRule"
            ]
        }
    ]
}
```

Request Response

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:StartBuildBatch"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/myProjectName"
            ]
        }
    ]
}
```

Dynamic resources

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:StartBuildBatch",
                "codebuild:StopBuildBatch",
                "codebuild:BatchGetBuildBatches"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
```

```
            ],
            "Resource": [
                "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventForCodeBuildStartBuildBatchRule"
            ]
        }
    ]
}
```

Request Response

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:StartBuildBatch"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/*"
            ]
        }
    ]
}
```

StopBuildBatch

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:StopBuildBatch"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/myProjectName"
            ]
```

```
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:StopBuildBatch"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/*"
            ]
        }
    ]
}
```

RetryBuildBatch

Static resources

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:RetryBuildBatch",
                "codebuild:StopBuildBatch",
                "codebuild:BatchGetBuildBatches"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/myProjectName"
            ]
        }
    ]
```

```
}
```

Request Response

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:RetryBuildBatch"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/myProjectName"
            ]
        }
    ]
}
```

Dynamic resources

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:RetryBuildBatch",
                "codebuild:StopBuildBatch",
                "codebuild:BatchGetBuildBatches"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/*"
            ]
        }
    ]
}
```

Request Response

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:RetryBuildBatch"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/*"
            ]
        }
    ]
}
```

DeleteBuildBatch

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:DeleteBuildBatch"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/myProjectName"
            ]
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "codebuild:DeleteBuildBatch"
            ],
            "Resource": [
                "arn:aws:codebuild:us-east-1:123456789012:project/*"
            ]
        }
    ]
}
```

Perform DynamoDB CRUD operations with Step Functions

You can integrate Step Functions with DynamoDB to perform CRUD operations on a DynamoDB table. This page lists the supported DynamoDB APIs and provides an example `Task` state to retrieve an item from DynamoDB.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of optimized DynamoDB integration

- There is no specific optimization for the Request Response integration pattern.
- Wait for a Callback with Task Token integration pattern is not supported.
- Only `GetItem`, `PutItem`, `UpdateItem`, and `DeleteItem` API actions are available through optimized integration. Other API actions, such as `CreateTable` are available using the DynamoDB AWS SDK integration.

The following is an example `Task` state that retrieves a message from DynamoDB.

```
"Read next Message from DynamoDB": {
    "Type": "Task",
    "Resource": "arn:aws:states:::dynamodb:getItem",
    "Arguments": {
        "TableName": "DYNAMO_DB_TABLE_NAME",
        "Key": {
            "MessageId": {"S": "{% $List[0] %}"}
```

```
        }
    }
```

To see this state in a working example, see the Transfer data records with Lambda, DynamoDB, and Amazon SQS starter template.

Exception prefix differences

When standard DynamoDB connections experience an error, the exception prefix will be `DynamoDb` (mixed case).
For optimized integrations, the exception prefix will be `DynamoDB` (uppercase DB).

Quota for input or result data

When sending or receiving data between services, the maximum input or result for a task is 256 KiB of data as a UTF-8 encoded string. See Quotas related to state machine executions.

Optimized DynamoDB APIs

- `GetItem`
- `PutItem`
- `DeleteItem`
- `UpdateItem`

Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

IAM policies for calling DynamoDB

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:123456789012:table/myTableName"
            ]
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem"
            ],
            "Resource": "*"
        }
    ]
}
```

For more information about the IAM policies for all DynamoDB API actions, see IAM policies with DynamoDB in the *Amazon DynamoDB Developer Guide*. Additionally, for information about the IAM policies for PartiQL for DynamoDB, see IAM policies with PartiQL for DynamoDB in the *Amazon DynamoDB Developer Guide*.

Run Amazon ECS or Fargate tasks with Step Functions

Learn how to integrate Step Functions with Amazon ECS or Fargate to run and manage tasks. In Amazon ECS, a task is the fundamental unit of computation. Tasks are defined by a task definition that specifies how a Docker container should be run, including the container image, CPU and memory limits, network configuration, and other parameters. This page lists the available Amazon ECS API actions and provides instructions on how to pass data to an Amazon ECS task using Step Functions.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of Optimized Amazon ECS/Fargate integration

- The Run a Job (.sync) integration pattern is supported.
- `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, which is the same as non-optimized integrations.
  - **Run a Job or Task Token**: If a non-empty `Failures` field is encountered, the task is failed with an `AmazonECS.Unknown` error.

Optimized Amazon ECS/Fargate APIs

- `RunTask` starts a new task using the specified task definition.

Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

Passing Data to an Amazon ECS Task

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

You can use `overrides` to override the default command for a container, and pass input to your Amazon ECS tasks. See `ContainerOverride`. In the example, we have used JsonPath to pass values to the `Task` from the input to the `Task` state.

The following includes a `Task` state that runs an Amazon ECS task and waits for it to complete.

```
{
 "StartAt": "Run an ECS Task and wait for it to complete",
 "States": {
   "Run an ECS Task and wait for it to complete": {
     "Type": "Task",
     "Resource": "arn:aws:states:::ecs:runTask.sync",
     "Arguments": {
                "Cluster": "cluster-arn",
                "TaskDefinition": "job-id",
                "Overrides": {
                    "ContainerOverrides": [
                        {
                            "Name": "container-name",
                            "Command": "{% $state.input.commands %}"
                        }
                    ]
                }
            },
     "End": true
    }
  }
}
```

The Command line in ContainerOverrides passes the commands from the state input to the container.

In the previous example state machine, given the following input, each of the commands would be passed as a container override:

```
{
  "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 Task Token.

```
{
   "StartAt":"Manage ECS task",
   "States":{
      "Manage ECS task":{
         "Type":"Task",
         "Resource":"arn:aws:states:::ecs:runTask.waitForTaskToken",
         "Arguments":{
            "LaunchType":"FARGATE",
            "Cluster":"cluster-arn",
            "TaskDefinition":"job-id",
            "Overrides":{
               "ContainerOverrides":[
                  {
                     "Name":"container-name",
                     "Environment":[
                        {
                           "Name" : "TASK_TOKEN_ENV_VARIABLE",
                           "Value" : "{% $states.context.Task.Token %}"
                        }
                     ]
                  }
               ]
            }
         },
         "End":true
      }
```

```
    }
}
```

IAM policies for calling Amazon ECS/AWS Fargate

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

Because the value for `TaskId` is not known until the task is submitted, Step Functions creates a more privileged `"Resource": "*"` policy.

> **Note**
>
> You can only stop Amazon Elastic Container Service (Amazon ECS) tasks that were started by Step Functions, despite the `"*"` IAM policy.

Run a Job (.sync)

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask"
            ],
            "Resource": [
                "arn:aws:ecs:region:
account-id:task-definition/taskDefinition:revisionNumber"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ecs:StopTask",
                "ecs:DescribeTasks"
```

```
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
               "arn:aws:events:region:
account-id:rule/StepFunctionsGetEventsForECSTaskRule"
            ]
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask",
                "ecs:StopTask",
                "ecs:DescribeTasks"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
               "arn:aws:events:region:
```

```
account-id:rule/StepFunctionsGetEventsForECSTaskRule"
            ]
        }
    ]
}
```

Request Response and Callback (.waitForTaskToken)

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask"
            ],
            "Resource": [
                "arn:aws:ecs:region:
account-id:task-definition/taskDefinition:revisionNumber"
            ]
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask"
            ],
            "Resource": "*"
        }
    ]
}
```

If your scheduled Amazon ECS tasks require the use of a task execution role, a task role, or a task role override, then you must add `iam:PassRole` permissions for each task execution role, task role, or task role override to the CloudWatch Events IAM role of the calling entity, which in this case is Step Functions.

Create and manage Amazon EKS clusters with Step Functions

Learn how to integrate Step Functions with Amazon EKS to manage Kubernetes clusters. 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. For more information, see Creating an Amazon EKS cluster in the Amazon EKS User Guide.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of Optimized Amazon EKS integration

- The Run a Job (.sync) integration pattern is supported.
- There are no specific optimizations for the Request Response integration pattern.
- The Wait for a Callback with Task Token integration pattern is not supported.

Note

The Step Functions EKS integration supports only Kubernetes APIs with public endpoint access. By default, EKS clusters API server endpoints have public access. For more information, see Amazon EKS cluster endpoint access control 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) in Service Integration Patterns.

> **Quota for input or result data**
>
> When sending or receiving data between services, the maximum input or result for a task is 256 KiB of data as a UTF-8 encoded string. See Quotas related to state machine executions.

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.

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 “Job poller” 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.

```
{
  ...
  "logs": {
    "pods": {
      "pod1": {
        "containers": {
          "container1": {
            "log": <log>
```

```
          },
          ...
        }
      },
      ...
    }
  }
```

  - 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:

```
{
```

```
  "StartAt": "Run a job on EKS",
  "States": {
    "Run a job on EKS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:runJob.sync",
      "Arguments": {
        "ClusterName": "MyCluster",
        "CertificateAuthority": "ANPAJ2UCCR6DPCEXAMPLE",
        "Endpoint": "https://AKIAIOSFODNN7EXAMPLE.yl4.us-east-1.eks.amazonaws.com",
        "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-2000",
                    "image": "perl",
                    "command": [ "perl" ],
                    "args": [
                      "-Mbignum=bpi",
                      "-wle",
                      "print bpi(2000)"
                    ]
                  }
                ],
                "restartPolicy": "Never"
              }
            }
          }
        }
      },
      "End": true
```

```
    }
  }
}
```

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.

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:

    ```
    "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`.

```
{
  "StartAt": "Call EKS",
  "States": {
    "Call EKS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:call",
      "Arguments": {
        "ClusterName": "MyCluster",
        "CertificateAuthority": "ANPAJ2UCCR6DPCEXAMPLE",
        "Endpoint": "https://444455556666.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.

```
{
  "StartAt": "Call EKS",
  "States": {
    "Call EKS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:call",
      "Arguments": {
        "ClusterName": "MyCluster",
        "CertificateAuthority": "ANPAJ2UCCR6DPCEXAMPLE",
        "Endpoint": "https://444455556666.yl4.us-east-1.eks.amazonaws.com",
        "Method": "DELETE",
        "Path": "/apis/batch/v1/namespaces/default/jobs/example-job",
        "QueryParameters": {
          "propagationPolicy": [
            "Foreground"
          ]
        }
      },
      "End": true
    }
  }
}
```

Optimized 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 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 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*.

  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`
  - Request syntax
  - Response syntax

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 node IAM role to Amazon EKS. For more information, see Using Service-Linked Roles in the *Amazon EKS User Guide*.

- `DeleteNodegroup`
  - Request syntax
  - Response syntax

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",
      "Arguments": {
        "Name": "MyCluster",
        "ResourcesVpcConfig": {
          "SubnetIds": [
            "subnet-053e7c47012341234",
            "subnet-027cfea4b12341234"
          ]
        },
        "RoleArn": "arn:aws:iam::account-id: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",
      "Arguments": {
        "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",
      "Arguments": {
        "ClusterName": "MyCluster",
        "FargateProfileName": "MyFargateProfile",
        "PodExecutionRoleArn": "arn:aws:iam::account-id:role/
MyFargatePodExecutionRole",
        "Selectors": [{
            "Namespace": "my-namespace",
            "Labels": { "my-label": "my-value" }
          }]
      },
      "End": true
    }
  }
}
```

The following includes a Task state that deletes a Fargate profile.

```
{
  "StartAt": "DeleteFargateProfile.sync",
  "States": {
    "DeleteFargateProfile.sync": {
      "Type": "Task",
```

```
      "Resource": "arn:aws:states:::eks:deleteFargateProfile.sync",
      "Arguments": {
        "ClusterName": "MyCluster",
        "FargateProfileName": "MyFargateProfile"
      },
      "End": true
    }
  }
}
```

The following includes a Task state that creates a node group.

```
{
  "StartAt": "CreateNodegroup.sync",
  "States": {
    "CreateNodegroup.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:createNodegroup.sync",
      "Arguments": {
        "ClusterName": "MyCluster",
        "NodegroupName": "MyNodegroup",
        "NodeRole": "arn:aws:iam::account-id:role/MyNodeInstanceRole",
        "Subnets": ["subnet-09fb51df01234", "subnet-027cfea4b1234"]
      },
      "End": true
    }
  }
}
```

The following includes a Task state that deletes a node group.

```
{
  "StartAt": "DeleteNodegroup.sync",
  "States": {
    "DeleteNodegroup.sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:deleteNodegroup.sync",
      "Arguments": {
        "ClusterName": "MyCluster",
        "NodegroupName": "MyNodegroup"
      },
      "End": true
    }
```

```
  }
}
```

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 user `arn:aws:iam::account-id:user/my-user` and the IAM role `arn:aws:iam::account-id:role/my-role`.

```
{
  "StartAt": "Add authorized user",
  "States": {
    "Add authorized user": {
      "Type": "Task",
      "Resource": "arn:aws:states:::eks:call",
      "Arguments": {
        "ClusterName": "MyCluster",
        "CertificateAuthority": "LS0tLS1CRUd...UtLS0tLQo=",
        "Endpoint": "https://444455556666.yl4.region.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::account-id:user/my-user\",
\"username\": \"my-user\", \"groups\": [ \"system:masters\" ] } ]",
            "mapRoles": "[{ \"rolearn\": \"arn:aws:iam::account-id: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::account-id:role/service-role/my-role`. This **service-role** path token should not be included when listing the role in `aws-auth`.

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"
}
```

IAM policies for calling Amazon EKS

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

`CreateCluster`

Resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "eks:CreateCluster"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "eks:DescribeCluster",
                "eks:DeleteCluster"
            ],
```

```
            "Resource": "arn:aws:eks:sa-east-1:444455556666:cluster/*"
        },
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": [
                "arn:aws:iam::444455556666:role/StepFunctionsSample-EKSClusterManag-
EKSServiceRole-ANPAJ2UCCR6DPCEXAMPLE"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "eks.amazonaws.com"
                }
            }
        }
    ]
}
```

CreateNodeGroup

Resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeSubnets",
                "eks:CreateNodegroup"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "eks:DescribeNodegroup",
                "eks:DeleteNodegroup"
            ],
            "Resource": "arn:aws:eks:sa-east-1:444455556666:nodegroup/*"
        },
```

```
        {
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "iam:ListAttachedRolePolicies"
            ],
            "Resource": "arn:aws:iam::444455556666:role/*"
        },
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": [
                "arn:aws:iam::444455556666:role/StepFunctionsSample-EKSClusterMan-
NodeInstanceRole-ANPAJ2UCCR6DPCEXAMPLE"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "eks.amazonaws.com"
                }
            }
        }
    ]
}
```

DeleteCluster

Resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "eks:DeleteCluster",
                "eks:DescribeCluster"
            ],
            "Resource": [
                "arn:aws:eks:sa-east-1:444455556666:cluster/ExampleCluster"
            ]
        }
    ]
}
```

DeleteNodegroup

Resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "eks:DeleteNodegroup",
                "eks:DescribeNodegroup"
            ],
            "Resource": [
                "arn:aws:eks:sa-east-1:444455556666:nodegroup/ExampleCluster/
ExampleNodegroup/*"
            ]
        }
    ]
}
```

For more information about using Amazon EKS with Step Functions, see Create and manage Amazon EKS clusters with Step Functions.

Create and manage Amazon EMR clusters with Step Functions

Learn how to integrate AWS Step Functions with Amazon EMR using 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.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of Optimized Amazon EMR 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.
- The Run a Job (.sync) integration pattern is supported.

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) 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. See Amazon EMR documentation for limitations in special regions.

> **Note**
>
> For integration with Amazon EMR, Step Functions has a hard-coded 60 seconds job polling frequency for the first 10 minutes and 300 seconds after that.

Optimized Amazon EMR APIs

The following table describes the differences between each Amazon EMR service integration API and corresponding Amazon EMR APIs.

Amazon EMR Service Integration API	Corresponding EMR API	Differences
createCluster Creates and starts running a cluster (job flow).	runJobFlow	`createCluster` uses the same request syntax as runJobFlow, except for the following:

Amazon EMR Service Integration API	Corresponding EMR API	Differences

Amazon EMR is linked directly to a unique type of IAM role known as a service-linked 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 `Instances.KeepJobFlowAliveWhenNoSteps` is mandatory, and must have the Boolean value TRUE. - 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. Response is this:
		`{ "ClusterId": "string" }`
		 Amazon EMR uses this:
		`{ "JobFlowId": "string"`

Amazon EMR Service Integration API	Corresponding EMR API	Differences
		`}`
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"] }`

terminateCluster
 	
Shuts down a cluster (job flow).
	[terminateJobFlows]()	Request uses this:
		`{ "ClusterId": "string" }`
		Amazon EMR uses this:
		`{ "JobFlowIds": ["string"] }`

Amazon EMR Service Integration API	Corresponding EMR API	Differences
terminateCluster.sync Shuts down a cluster (job flow).	[terminateJobFlows]()	The same as `terminateCluster` , but waits for the cluster to terminate.

addStep
 	
Adds a new step to a running cluster.
 	
Optionally, you can also specify the [`ExecutionRoleArn`]() parameter while using this API.
	[addJobFlowSteps]()	Request uses the key `"ClusterId"` . Amazon EMR uses `"JobFlowId"` . Request uses a single step.
		
		`{ "Step": <"StepConfig object"> }`
		
		Amazon EMR uses this:
		
		`{ "Steps": [<StepConfig objects>] }`
		
		Response is this:
		
		`{ "StepId": "string" }`
		
		Amazon EMR returns this:
		
		`{ "StepIds": [<strings >] }`

Amazon EMR Service Integration API	Corresponding EMR API	Differences
addStep.sync Adds a new step to a running cluster. Optionally, you can also specify the [`ExecutionRoleArn`](#) parameter while using this API.	[addJobFlowSteps](#)	The same as `addStep`, but waits for the step to complete.

Amazon EMR Service Integration API	Corresponding EMR API	Differences

cancelStep
 	
Cancels a pending step in a running cluster.
	[cancelSteps](#)	Request uses this:
		`{`
		`  "StepId": "string"`
		`}`
		
		Amazon EMR uses this:
		`{`
		`  "StepIds": [<strings >]`
		`}`
		
		Response is this:
		`{`
		`  "CancelStepsInfo": <CancelStepsInfo object>`
		`}`
		
		Amazon EMR uses this:
		`{`
		`  "CancelStepsInfoList": [<CancelStepsInfo objects>]`
		`}`

Amazon EMR Service Integration API	Corresponding EMR API	Differences
modifyInstanceFleetByName Modifies the target On-Demand and target Spot capacities for the instance fleet with the specified `InstanceFleetName` .	modifyInstanceFleet	Request is the same as for `modifyInstanceFleet` , except for the following: • The field `Instance.InstanceFleetId` is not allowed. • At runtime the `InstanceFleetId` is determined automatically by the service integration by calling `ListInstanceFleets` and parsing the result.

<table>
<tr><th>Amazon EMR Service Integration API</th><th>Corresponding EMR API</th><th>Differences</th></tr>
<tr><td> modifyInstanceGroupByName Modifies the number of nodes and configuration settings of an instance group. </td><td>modifyInstanceGroups</td><td> Request is this:
<pre><code>{
  "ClusterId":
 "string",
  "InstanceGroup":
 <InstanceGroupModi
fyConfig object>
}</code></pre>
Amazon EMR uses a list:
<pre><code>{
  "ClusterId":
 ["string"],
  "InstanceGroups":
 [<InstanceGroupMod
ifyConfig objects>]
}</code></pre>
Within the <code>InstanceGroupModifyConfig</code> object, the field <code>InstanceGroupId</code> is not allowed.
A new field, <code>InstanceGroupName</code>, has been added. At runtime the <code>InstanceGroupId</code> is determined automatically by the service integration by calling <code>ListInstanceGroups</code> and parsing the result. </td></tr>
</table>

Workflow example

The following includes a Task state that creates a cluster.

```
"Create_Cluster": {
    "Type": "Task",
    "Resource": "arn:aws:states:::elasticmapreduce:createCluster.sync",
    "Arguments": {
        "Name": "MyWorkflowCluster",
        "VisibleToAllUsers": true,
        "ReleaseLabel": "emr-5.28.0",
        "Applications": [
            {
                "Name": "Hive"
            }
        ],
        "ServiceRole": "EMR_DefaultRole",
        "JobFlowRole": "EMR_EC2_DefaultRole",
        "LogUri": "s3n://aws-logs-account-id-us-east-1/elasticmapreduce/",
        "Instances": {
            "KeepJobFlowAliveWhenNoSteps": true,
            "InstanceFleets": [
                {
                    "InstanceFleetType": "MASTER",
                    "Name": "MASTER",
                    "TargetOnDemandCapacity": 1,
                    "InstanceTypeConfigs": [
                        {
                            "InstanceType": "m4.xlarge"
                        }
                    ]
                },
                {
                    "InstanceFleetType": "CORE",
                    "Name": "CORE",
                    "TargetOnDemandCapacity": 1,
                    "InstanceTypeConfigs": [
                        {
                            "InstanceType": "m4.xlarge"
                        }
                    ]
                }
            ]
```

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

The following includes a `Task` state that enables termination protection.

```
"Enable_Termination_Protection": {
    "Type": "Task",
    "Resource": "arn:aws:states:::elasticmapreduce:setClusterTerminationProtection",
    "Arguments": {
        "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",
    "Arguments": {
        "ClusterId": "{% $ClusterId %}",
        "ExecutionRoleArn": "arn:aws:iam::account-id:role/myEMR-execution-role",
        "Step": {
            "Name": "The first step",
            "ActionOnFailure": "TERMINATE_CLUSTER",
            "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://<amzn-s3-demo-bucket>/MyHiveQueryResults/"
                ]
            }
```

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

The following includes a `Task` state that cancels a step.

```
"Cancel_Step_One": {
    "Type": "Task",
    "Resource": "arn:aws:states:::elasticmapreduce:cancelStep",
    "Arguments": {
        "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",
    "Arguments": {
        "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",
    "Arguments": {
        "ClusterId": "j-account-id3",
        "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",
    "Arguments": {
        "ClusterId": "j-account-id3",
        "InstanceFleetName": "MyCoreFleet",
        "InstanceFleet": {
            "TargetOnDemandCapacity": 8,
            "TargetSpotCapacity": 0
        }
    },
    "End": true
}
```

IAM policies for calling Amazon EMR

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

addStep

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elasticmapreduce:AddJobFlowSteps",
                "elasticmapreduce:DescribeStep",
                "elasticmapreduce:CancelSteps"
            ],
            "Resource": [
                "arn:aws:elasticmapreduce:us-east-1:123456789012:cluster/clusterId"
            ]
```

```
        }
    ]
}
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:AddJobFlowSteps",
        "elasticmapreduce:DescribeStep",
        "elasticmapreduce:CancelSteps"
      ],
      "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
    }
  ]
}
```

cancelStep

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "elasticmapreduce:CancelSteps",
            "Resource": [
                "arn:aws:elasticmapreduce:us-east-1:123456789012:cluster/myCluster-id"
            ]
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "elasticmapreduce:CancelSteps",
            "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
        }
    ]
}
```

createCluster

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:RunJobFlow",
        "elasticmapreduce:DescribeCluster",
        "elasticmapreduce:TerminateJobFlows"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": [
        "arn:aws:iam::123456789012:role/myRoleName"
      ]
    }
  ]
}
```

setClusterTerminationProtection

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": "elasticmapreduce:SetTerminationProtection",
            "Resource": [
                "arn:aws:elasticmapreduce:us-east-1:123456789012:cluster/myCluster-id"
            ]
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "elasticmapreduce:SetTerminationProtection",
            "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
        }
    ]
}
```

modifyInstanceFleetByName

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elasticmapreduce:ModifyInstanceFleet",
                "elasticmapreduce:ListInstanceFleets"
            ],
            "Resource": [
                "arn:aws:elasticmapreduce:us-east-1:123456789012:cluster/myCluster-id"
            ]
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elasticmapreduce:ModifyInstanceFleet",
                "elasticmapreduce:ListInstanceFleets"
            ],
            "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
        }
    ]
}
```

modifyInstanceGroupByName

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elasticmapreduce:ModifyInstanceGroups",
                "elasticmapreduce:ListInstanceGroups"
            ],
            "Resource": [
                "arn:aws:elasticmapreduce:us-east-1:123456789012:cluster/myCluster-id"
            ]
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": [
                "elasticmapreduce:ModifyInstanceGroups",
                "elasticmapreduce:ListInstanceGroups"
            ],
            "Resource": "*"
        }
    ]
}
```

terminateCluster

Static resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:TerminateJobFlows",
        "elasticmapreduce:DescribeCluster"
      ],
      "Resource": [
        "arn:aws:elasticmapreduce:us-east-1:123456789012:cluster/myCluster-id"
      ]
    }
  ]
}
```

Dynamic resources

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:TerminateJobFlows",
        "elasticmapreduce:DescribeCluster"
      ],
      "Resource": "arn:aws:elasticmapreduce:*:*:cluster/*"
    }
  ]
```

```
}
```

Create and manage Amazon EMR clusters on EKS with AWS Step Functions

Learn how to integrate AWS Step Functions with Amazon EMR on EKS using 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.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

How the Optimized Amazon EMR on EKS integration is different than the Amazon EMR on EKS AWS SDK integration

- The Run a Job (.sync) integration pattern is supported.
- There are no specific optimizations for the Request Response integration pattern.
- The Wait for a Callback with Task Token integration pattern is not supported.

Note

For integration with Amazon EMR, Step Functions has a hard-coded 60 seconds job polling frequency for the first 10 minutes and 300 seconds after that.

API	Request response	Run a job (.sync)
CreateVirtualCluster	Supported	*Not supported*
DeleteVirtualCluster	Supported	Supported
StartJobRun	Supported	Supported

Supported Amazon EMR on EKS APIs:

> **Quota for input or result data**
>
> When sending or receiving data between services, the maximum input or result for a task is 256 KiB of data as a UTF-8 encoded string. See Quotas related to state machine executions.

- `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": {
  "Type": "Task",
  "Resource": "arn:aws:states:::emr-containers:createVirtualCluster",
  "Arguments": {
    "Name": "MyVirtualCluster",
    "ContainerProvider": {
      "Id": "EKSClusterName",
      "Type": "EKS",
      "Info": {
        "EksInfo": {
          "Namespace": "Namespace"
        }
      }
    }
  },
```

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

The following includes a `Task` state that submits a job to a virtual cluster and waits for it to complete.

```
"Submit_Job": {
    "Type": "Task",
    "Resource": "arn:aws:states:::emr-containers:startJobRun.sync",
    "Arguments": {
      "Name": "MyJobName",
      "VirtualClusterId": "{% $VirtualClusterId %}",
      "ExecutionRoleArn": "arn:aws:iam::<accountId>:role/job-execution-role",
      "ReleaseLabel": "emr-6.2.0-latest",
      "JobDriver": {
        "SparkSubmitJobDriver": {
          "EntryPoint": "s3://<amzn-s3-demo-bucket>/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://<amzn-s3-demo-logging-bucket1>"
              }
            }
          },
          "Tags": {
            "taskType": "jobName"
          }
        },
        "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",
  "Arguments": {
    "Id": "{% $states.input.VirtualClusterId %}",
  },
  "End": true
}
```

To learn about configuring IAM permissions when using Step Functions with other AWS services, see How Step Functions generates IAM policies for integrated services.

Create and manage Amazon EMR Serverless applications with Step Functions

Learn how to create, start, stop, and delete applications on EMR Serverless using Step Functions. This page lists the supported APIs and provides example `Task` states to perform common use cases.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of Optimized EMR Serverless integration

- The Optimized EMR Serverless service integration has a customized set of APIs that wrap the underlying EMR Serverless APIs. Because of this customization, the optimized EMR Serverless integration differs significantly from the AWS SDK service integration.
- In addition, the optimized EMR Serverless integration supports Run a Job (.sync) integration pattern.
- The Wait for a Callback with Task Token integration pattern is **not** supported.

EMR Serverless service integration APIs

To integrate AWS Step Functions with EMR Serverless, you can use the following six EMR Serverless service integration APIs. These service integration APIs are similar to the corresponding EMR Serverless APIs, with some differences in the fields that are passed and in the responses that are returned.

The following table describes the differences between each EMR Serverless service integration API and its corresponding EMR Serverless API.

EMR Serverless service integration API	Corresponding EMR Serverless API	Differences
createApplication Creates an application. EMR Serverless is linked to a unique type of IAM role known as a service-linked role. For `createApplication` and `createApplication.sync` to work, you must have configured the necessary permissions to create the service-linked role `AWSServiceRoleForA`	CreateApplication	None

EMR Serverless service integration API	Corresponding EMR Serverless API	Differences
`mazonEMRServerless .` For more information about this, including a statement you can add to your IAM permissions policy, see [Using service-linked roles for EMR Serverless]().		
createApplication.sync Creates an application.	[CreateApplication]()	No differences between the requests and responses of the EMR Serverless API and EMR Serverless service integration API. However, *createApplication.sync* waits for the application to reach the `CREATED` state.
startApplication Starts a specified application and initializes the application's initial capacity if configured.	[StartApplication]()	The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integration API response includes the following data. `{ "ApplicationId": "string" }`

<table>
<tr><th>EMR Serverless service integration API</th><th>Corresponding EMR Serverless API</th><th>Differences</th></tr>
<tr><td> startApplication.sync Starts a specified applicati on and initializes the initial capacity if configured. </td><td>StartApplication</td><td> The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integrati on API response includes the following data. <pre><code>{
  "ApplicationId":
 "string"
}</code></pre> Also, startApplication.sync waits for the application to reach the <code>STARTED</code> state. </td></tr>
<tr><td> stopApplication Stops a specified application and releases initial capacity if configured. All scheduled and running jobs must be completed or cancelled before stopping an applicati on. </td><td>StopApplication</td><td> The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integrati on API response includes the following data. <pre><code>{
  "ApplicationId":
 "string"
}</code></pre></td></tr>
</table>

<table>
<tr><th>EMR Serverless service integration API</th><th>Corresponding EMR Serverless API</th><th>Differences</th></tr>
<tr><td>stopApplication.sync

Stops a specified application and releases initial capacity if configured. All scheduled and running jobs must be completed or cancelled before stopping an application.</td><td>StopApplication</td><td>The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integration API response includes the following data.

<code>{ "ApplicationId": "string" }</code>

Also, stopApplication.sync waits for the application to reach the <code>STOPPED</code> state.</td></tr>
<tr><td>deleteApplication

Deletes an application. An application must be in the <code>STOPPED</code> or <code>CREATED</code> state in order to be deleted.</td><td>DeleteApplication</td><td>The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integration API response includes the following data.

<code>{ "ApplicationId": "string" }</code></td></tr>
</table>

EMR Serverless service integration API	Corresponding EMR Serverless API	Differences
deleteApplication.sync Deletes an application. An application must be in the `STOPPED` or `CREATED` state in order to be deleted.	[DeleteApplication]()	The EMR Serverless API response doesn't contain any data, but the EMR Serverless service integration API response includes the following data. `{ "ApplicationId": "string" }` Also, *stopApplication.sync* waits for the application to reach the `TERMINATED` state.
startJobRun Starts a job run.	[StartJobRun]()	None
startJobRun.sync Starts a job run.	[StartJobRun]()	No differences between the requests and responses of the EMR Serverless API and EMR Serverless service integration API. However, *startJobRun.sync* waits for the application to reach the `SUCCESS` state.
cancelJobRun Cancels a job run.	[CancelJobRun]()	None

EMR Serverless service integration API	Corresponding EMR Serverless API	Differences
cancelJobRun.sync Cancels a job run.	CancelJobRun	No differences between the requests and responses of the EMR Serverless API and EMR Serverless service integration API. However, *cancelJobRun.sync* waits for the application to reach the `CANCELLED` state.

EMR Serverless integration use cases

For the Optimized EMR Serverless service integration, we recommend that you create a single application, and then use that application to run multiple jobs. For example, in a single state machine, you can include multiple startJobRun requests, all of which use the same application. The following Task workflow state state examples show use cases to integrate EMR Serverless APIs with Step Functions. For information about other use cases of EMR Serverless, see What is Amazon EMR Serverless.

> **Tip**
>
> To deploy an example of a state machine that integrates with EMR Serverless for running multiple jobs;, see Run an EMR Serverless job.

- Create an application
- Start an application
- Stop an application
- Delete an application
- Start a job in an application
- Cancel a job in an application

To learn about configuring IAM permissions when using Step Functions with other AWS services, see [How Step Functions generates IAM policies for integrated services](#).

In the examples shown in the following use cases, replace the *italicized* text with your resource-specific information. For example, replace *yourApplicationId* with the ID of your EMR Serverless application, such as `00yv7iv71inak893`.

Create an application

The following Task state example creates an application using the *createApplication.sync* service integration API.

```
"Create_Application": {
    "Type": "Task",
    "Resource": "arn:aws:states:::emr-serverless:createApplication.sync",
    "Arguments": {
        "Name": "MyApplication",
        "ReleaseLabel": "emr-6.9.0",
        "Type": "SPARK"
    },
    "End": true
}
```

Start an application

The following Task state example starts an application using the *startApplication.sync* service integration API.

```
"Start_Application": {
    "Type": "Task",
    "Resource": "arn:aws:states:::emr-serverless:startApplication.sync",
    "Arguments": {
        "ApplicationId": "yourApplicationId"
    },
    "End": true
}
```

Stop an application

The following Task state example stops an application using the *stopApplication.sync* service integration API.

```
"Stop_Application": {
    "Type": "Task",
    "Resource": "arn:aws:states:::emr-serverless:stopApplication.sync",
    "Arguments": {
        "ApplicationId": "yourApplicationId"
    },
    "End": true
}
```

Delete an application

The following Task state example deletes an application using the *deleteApplication.sync* service integration API.

```
"Delete_Application": {
    "Type": "Task",
    "Resource": "arn:aws:states:::emr-serverless:deleteApplication.sync",
    "Arguments": {
        "ApplicationId": "yourApplicationId"
    },
    "End": true
}
```

Start a job in an application

The following Task state example starts a job in an application using the *startJobRun.sync* service integration API.

```
"Start_Job": {
    "Type": "Task",
    "Resource": "arn:aws:states:::emr-serverless:startJobRun.sync",
    "Arguments": {
        "ApplicationId": "yourApplicationId",
        "ExecutionRoleArn": "arn:aws:iam::account-id:role/myEMRServerless-execution-
role",
        "JobDriver": {
            "SparkSubmit": {
                "EntryPoint": "s3://<amzn-s3-demo-bucket>/sample.py",
                "EntryPointArguments": ["1"],
                "SparkSubmitParameters": "--conf spark.executor.cores=4 --conf
 spark.executor.memory=4g --conf spark.driver.cores=2 --conf spark.driver.memory=4g --
conf spark.executor.instances=1"
```

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

Cancel a job in an application

The following Task state example cancels a job in an application using the *cancelJobRun.sync* service integration API.

```
"Cancel_Job": {
    "Type": "Task",
    "Resource": "arn:aws:states:::emr-serverless:cancelJobRun.sync",
    "Arguments": {
        "ApplicationId": "{% $states.input.ApplicationId %}",
        "JobRunId": "{% $states.input.JobRunId %}"
    },
    "End": true
}
```

IAM policies for calling Amazon EMR Serverless

When you create a state machine using the console, Step Functions automatically creates an execution role for your state machine with the least privileges required. These automatically generated IAM roles are valid for the AWS Region in which you create the state machine.

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

We recommend that when you create IAM policies, do not include wildcards in the policies. As a security best practice, you should scope your policies down as much as possible. You should use dynamic policies only when certain input parameters are not known during runtime.

Further, administrator users should be careful when granting non-administrator users execution roles for running the state machines. We recommend that you include passRole policies in the execution roles if you're creating policies on your own. We also recommend that you add the `aws:SourceARN` and `aws:SourceAccount` context keys in the execution roles.

IAM policy examples for EMR Serverless integration with Step Functions

- IAM policy example for CreateApplication
- IAM policy example for StartApplication
- IAM policy example for StopApplication
- IAM policy example for DeleteApplication
- IAM policy example for StartJobRun
- IAM policy example for CancelJobRun

IAM policy example for CreateApplication

The following is an IAM policy example for a state machine with a CreateApplication Task workflow state state.

> **Note**
>
> You need to specify the CreateServiceLinkedRole permissions in your IAM policies during the creation of the first ever application in your account. Thereafter, you need not add this permission. For information about CreateServiceLinkedRole, see CreateServiceLinkedRole in the https://docs.aws.amazon.com/IAM/latest/APIReference/.

Static and dynamic resources for the following policies are the same.

Run a Job (.sync)

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "emr-serverless:CreateApplication"
            ],
            "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/*"
            ]
        },
        {
```

```
            "Effect": "Allow",
            "Action": [
                "emr-serverless:GetApplication",
                "emr-serverless:DeleteApplication"
            ],
            "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventsForEMRServerlessApplicationRule"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::123456789012:role/aws-service-role/ops.emr-
serverless.amazonaws.com/AWSServiceRoleForAmazonEMRServerless*",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": "ops.emr-serverless.amazonaws.com"
                }
            }
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",

   "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": [
                "emr-serverless:CreateApplication"
            ],
            "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::123456789012:role/aws-service-role/ops.emr-
serverless.amazonaws.com/AWSServiceRoleForAmazonEMRServerless*",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": "ops.emr-serverless.amazonaws.com"
                }
            }
        }
    ]
}
```

IAM policy example for StartApplication

Static resources

The following are IAM policy examples for static resources when you use a state machine with a StartApplication Task workflow state state.

Run a Job (.sync)

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StartApplication",
               "emr-serverless:GetApplication",
               "emr-serverless:StopApplication"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId"
```

```
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-
east-1:123456789012:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
            ]
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StartApplication"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId"
            ]
        }
    ]
}
```

Dynamic resources

The following are IAM policy examples for dynamic resources when you use a state machine with a StartApplication Task workflow state state.

Run a Job (.sync)

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StartApplication",
               "emr-serverless:GetApplication",
               "emr-serverless:StopApplication"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-
east-1:123456789012:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
            ]
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StartApplication"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
```

```
            ]
        }
    ]
}
```

IAM policy example for StopApplication

Static resources

The following are IAM policy examples for static resources when you use a state machine with a StopApplication [Task workflow state](#) state.

Run a Job (.sync)

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StopApplication",
               "emr-serverless:GetApplication"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-
east-1:123456789012:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
            ]
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StopApplication"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId"
            ]
        }
    ]
}
```

Dynamic resources

The following are IAM policy examples for dynamic resources when you use a state machine with a StopApplication Task workflow state state.

Run a Job (.sync)

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StopApplication",
               "emr-serverless:GetApplication"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
```

```
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-
east-1:123456789012:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
            ]
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StopApplication"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
            ]
        }
    ]
}
```

IAM policy example for DeleteApplication

Static resources

The following are IAM policy examples for static resources when you use a state machine with a DeleteApplication Task workflow state state.

Run a Job (.sync)

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
```

```
            "Action": [
                "emr-serverless:DeleteApplication",
                "emr-serverless:GetApplication"
             ],
            "Resource": [
                 "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId"
             ]
        },
        {
             "Effect": "Allow",
             "Action": [
                 "events:PutTargets",
                 "events:PutRule",
                 "events:DescribeRule"
             ],
             "Resource": [
                 "arn:aws:events:us-
east-1:123456789012:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
             ]
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "emr-serverless:DeleteApplication"
             ],
            "Resource": [
                 "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId"
             ]
        }
    ]
}
```

Dynamic resources

The following are IAM policy examples for dynamic resources when you use a state machine with a DeleteApplication Task workflow state state.

Run a Job (.sync)

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:DeleteApplication",
               "emr-serverless:GetApplication"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-
east-1:123456789012:rule/StepFunctionsGetEventsForEMRServerlessApplicationRule"
            ]
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
```

```
                "emr-serverless:DeleteApplication"
             ],
            "Resource": [
                 "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
             ]
        }
    ]
}
```

IAM policy example for StartJobRun

Static resources

The following are IAM policy examples for static resources when you use a state machine with a StartJobRun [Task workflow state](#) state.

Run a Job (.sync)

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StartJobRun"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": [
                "arn:aws:iam::123456789012:role/jobExecutionRoleArn"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "emr-serverless.amazonaws.com"
                }
            }
        },
```

```
        {
            "Effect": "Allow",
            "Action": [
                "emr-serverless:GetJobRun",
                "emr-serverless:CancelJobRun"
            ],
            "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId/jobruns/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-
east-1:123456789012:rule/StepFunctionsGetEventsForEMRServerlessJobRule"
            ]
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StartJobRun"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId"
            ]
        },
        {
            "Effect": "Allow",
```

```
                "Action": "iam:PassRole",
                "Resource": [
                    "arn:aws:iam::123456789012:role/jobExecutionRoleArn"
                ],
                "Condition": {
                    "StringEquals": {
                        "iam:PassedToService": "emr-serverless.amazonaws.com"
                    }
                }
            }
        ]
}
```

Dynamic resources

The following are IAM policy examples for dynamic resources when you use a state machine with a StartJobRun [Task workflow state](#) state.

Run a Job (.sync)

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StartJobRun",
               "emr-serverless:GetJobRun",
               "emr-serverless:CancelJobRun"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": [
                "arn:aws:iam::123456789012:role/jobExecutionRoleArn"
            ],
            "Condition": {
                "StringEquals": {
```

```
                    "iam:PassedToService": "emr-serverless.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-
east-1:123456789012:rule/StepFunctionsGetEventsForEMRServerlessJobRule"
            ]
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:StartJobRun"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": [
                "arn:aws:iam::123456789012:role/jobExecutionRoleArn"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "emr-serverless.amazonaws.com"
                }
```

```
            }
        }
    ]
}
```

IAM policy example for CancelJobRun

Static resources

The following are IAM policy examples for static resources when you use a state machine with a CancelJobRun Task workflow state state.

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "emr-serverless:CancelJobRun",
                "emr-serverless:GetJobRun"
             ],
            "Resource": [
                 "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId/jobruns/jobRunId"
            ]
        },
        {
             "Effect": "Allow",
             "Action": [
                 "events:PutTargets",
                 "events:PutRule",
                 "events:DescribeRule"
             ],
             "Resource": [
                 "arn:aws:events:us-
east-1:123456789012:rule/StepFunctionsGetEventsForEMRServerlessJobRule"
            ]
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:CancelJobRun"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/applicationId/jobruns/jobRunId"
            ]
        }
    ]
}
```

Dynamic resources

The following are IAM policy examples for dynamic resources when you use a state machine with a CancelJobRun Task workflow state state.

Run a Job (.sync)

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:CancelJobRun",
               "emr-serverless:GetJobRun"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
```

```
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-
east-1:123456789012:rule/StepFunctionsGetEventsForEMRServerlessJobRule"
            ]
        }
    ]
}
```

Request Response

```
{
   "Version": "2012-10-17",
   "Statement": [
        {
           "Effect": "Allow",
           "Action": [
               "emr-serverless:CancelJobRun"
            ],
           "Resource": [
                "arn:aws:emr-serverless:us-east-1:123456789012:/applications/*"
            ]
        }
    ]
}
```

Add EventBridge events with Step Functions

Step Functions provides a service integration API for integrating with Amazon EventBridge. Learn how to build event-driven applications by sending custom events directly from Step Functions workflows.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of Optimized EventBridge 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`.

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:

- Adding Amazon EventBridge events with PutEvents in the EventBridge User Guide.
- Wait for a Callback with Task Token in Service Integration Patterns.

The following includes a `Task` that sends a custom event:

```
{
  "Type": "Task",
  "Resource": "arn:aws:states:::events:putEvents",
  "Arguments": {
    "Entries": [
      {
        "Detail": {
          "Message": "MyMessage"
        },
        "DetailType": "MyDetailType",
        "EventBusName": "MyEventBus",
        "Source": "my.source"
      }
    ]
  },
```

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

> **Quota for input or result data**
>
> When sending or receiving data between services, the maximum input or result for a task is 256 KiB of data as a UTF-8 encoded string. See Quotas related to state machine executions.

Optimized EventBridge API

Supported EventBridge API and syntax include:

- `PutEvents`

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.

IAM policies for calling EventBridge

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

PutEvents

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "events:PutEvents"
            ],
            "Resource": [
                "arn:aws:events:us-east-1:123456789012:event-bus/my-project-eventbus"
            ],
            "Effect": "Allow"
        }
    ]
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "events:PutEvents"
            ],
            "Resource": "arn:aws:events:*:*:event-bus/*"
        }
    ]
}
```

Start an AWS Glue job with Step Functions

Learn to use Step Functions to start a job run on AWS Glue. This page lists the supported API actions and provides an example `Task` state to start a AWS Glue job.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of Optimized AWS Glue integration

- The Run a Job (.sync) integration pattern is available.
- The `JobName` field is extracted from the request and inserted into the response, which normally only contains `JobRunID`.

The following includes a `Task` state that starts an AWS Glue job.

```
"Glue StartJobRun": {
      "Type": "Task",
      "Resource": "arn:aws:states:::glue:startJobRun.sync",
      "Arguments": {
        "JobName": "GlueJob-JTrR05l98qMG"
      },
      "Next": "ValidateOutput"
    },
```

Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

Optimized AWS Glue APIs

- `StartJobRun`

IAM policies for calling AWS Glue

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

AWS Glue does not have resource-based control.

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "glue:StartJobRun",
                "glue:GetJobRun",
                "glue:GetJobRuns",
                "glue:BatchStopJobRun"
            ],
            "Resource": "*"
        }
    ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "glue:StartJobRun"
            ],
            "Resource": "*"
        }
    ]
}
```

Start AWS Glue DataBrew jobs with Step Functions

Learn how you can use the DataBrew integration to add data cleaning and data normalization steps into your analytics and machine learning workflows with Step Functions.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

The following includes a `Task` state that starts a request-response DataBrew job.

```
"DataBrew StartJobRun": {
            "Type": "Task",
            "Resource": "arn:aws:states:::databrew:startJobRun",
            "Arguments": {
               "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",
           "Arguments": {
              "Name": "sample-proj-job-1"
           },
           "Next": "NEXT_STATE"
          },
```

Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

Supported DataBrew APIs

- StartJobRun

IAM policies for calling DataBrew

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "databrew:startJobRun",
                "databrew:listJobRuns",
                "databrew:stopJobRun"
            ],
            "Resource": [
                "arn:aws:databrew:us-east-1:123456789012:job/*"
            ]
        }
    ]
}
```

Request Response

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "databrew:startJobRun"
            ],
            "Resource": [
```

```
                    "arn:aws:databrew:us-east-1:123456789012:job/*"
                ]
            }
        ]
}
```

Invoke an AWS Lambda function with Step Functions

Learn how to use Step Functions to invoke Lambda functions either synchronously or asynchronously as part of an event-driven serverless application.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of Optimized Lambda integration

- The `Payload` field of the response is parsed from escaped Json to Json.
- If the response contains the field `FunctionError` or an exception is raised within the Lambda function, the task fails.

For more information about managing state input, output, and results, see Processing input and output in Step Functions.

Optimized Lambda APIs

- `Invoke`

Workflow Examples

The following includes a `Task` state that invokes a Lambda function.

```
{
   "StartAt":"CallLambda",
   "States":{
      "CallLambda":{
         "Type":"Task",
```

```
            "Resource":"arn:aws:states:::lambda:invoke",
            "Arguments":{
               "FunctionName":"arn:aws:lambda:region:account-id:function:MyFunction"
            },
            "End":true
         }
      }
}
```

The following includes a `Task` state that implements the [callback](#) service integration pattern.

```
{
   "StartAt":"GetManualReview",
   "States":{
      "GetManualReview":{
         "Type":"Task",
         "Resource":"arn:aws:states:::lambda:invoke.waitForTaskToken",
         "Arguments":{
            "FunctionName":"arn:aws:lambda:region:account-id:function:get-model-review-
decision",
            "Payload":{
               "model":"{% $states.input.my-model %}",
               "TaskToken": "{% $states.context.Task.Token %}"
            },
            "Qualifier":"prod-v1"
         },
         "End":true
      }
   }
}
```

When you invoke a Lambda function, the execution will wait for the function to complete. If you invoke the Lambda function with a callback task, the heartbeat timeout does not start counting until after the Lambda function has completed executing and returned a result. As long as the Lambda function executes, the heartbeat timeout is not enforced.

It is also possible to call Lambda asynchronously using the `InvocationType` parameter, as seen in the following example:

> **Note**
>
> For asynchronous invocations of Lambda functions, the heartbeat timeout period starts immediately.

```
{

  "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",
      "Arguments": {
        "FunctionName": "arn:aws:lambda:region:account-id:function:echo",
        "InvocationType": "Event"
      },
      "End": true
    }
  }
}
```

When the Task result is returned, the function output is nested inside a dictionary of metadata. For example:

```
{

   "ExecutedVersion":"$LATEST",
   "Payload":"FUNCTION OUTPUT",
   "SdkHttpMetadata":{
      "HttpHeaders":{
         "Connection":"keep-alive",
         "Content-Length":"4",
         "Content-Type":"application/json",
         "Date":"Fri, 26 Mar 2021 07:42:02 GMT",
         "X-Amz-Executed-Version":"$LATEST",
         "x-amzn-Remapped-Content-Length":"0",
         "x-amzn-RequestId":"0101aa0101-1111-111a-aa55-1010aaa1010",
         "X-Amzn-Trace-Id":"root=1-1a1a000a2a2-fe0101aa10ab;sampled=0"
      },
      "HttpStatusCode":200
```

```
   },
   "SdkResponseMetadata":{
      "RequestId":"6b3bebdb-9251-453a-ae45-512d9e2bf4d3"
   },
   "StatusCode":200
}
```

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 `.waitForTaskToken`, and the task result contains only the function output.

```
{
   "StartAt":"CallFunction",
   "States":{
      "CallFunction": {
         "Type":"Task",
         "Resource":"arn:aws:lambda:region:account-id:function:HelloFunction",
         "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

IAM policies for calling AWS Lambda

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

In the following example, a state machine with two AWS Lambda task states which call `function1` and `function2`, the autogenerated policy includes `lambda:Invoke` permission for both functions.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "arn:aws:lambda:us-east-1:123456789012:function:myFn1",
                "arn:aws:lambda:us-east-1:123456789012:function:myFn2"
            ]
        }
    ]
}
```

Create an AWS Elemental MediaConvert job with Step Functions

Learn how to use Step Functions to create an AWS Elemental MediaConvert job using the `CreateJob` API.

> **Experiment with Step Functions and MediaConvert**
>
> Learn how to use the MediaConvert optimized integration in a workflow that detects and removes SMTPE color bars of unknown length from the beginning of a video clip. Read the blog post from Apr, 12, 2024: *Low code workflows with AWS Elemental MediaConvert*

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

> **Key features of Optimized MediaConvert integration**
>
> - The Run a Job (.sync) and Request Response integration patterns are supported.
> - Step Functions will add the following custom tag to MediaConvert jobs: `ManagedByService: AWSStepFunctions`

- There is no specific optimization for Wait for a Callback with Task Token integration patterns.

The following includes a `Task` state that submits a MediaConvert job and waits for it to complete.

```
{
    "StartAt": "MediaConvert_CreateJob",
    "States": {
        "MediaConvert_CreateJob": {
        "Type": "Task",
        "Resource": "arn:aws:states:::mediaconvert:createJob.sync",
        "Arguments": {
            "Role": "arn:aws:iam::111122223333:role/Admin",
            "Settings": {
            "OutputGroups": [
                {
                "Outputs": [
                    {
                    "ContainerSettings": {
                        "Container": "MP4"
                    },
                    "VideoDescription": {
                        "CodecSettings": {
                        "Codec": "H_264",
                        "H264Settings": {
                            "MaxBitrate": 1000,
                            "RateControlMode": "QVBR",
                            "SceneChangeDetect": "TRANSITION_DETECTION"
                        }
                        }
                    },
                    "AudioDescriptions": [
                        {
                        "CodecSettings": {
                            "Codec": "AAC",
                            "AacSettings": {
                            "Bitrate": 96000,
                            "CodingMode": "CODING_MODE_2_0",
                            "SampleRate": 48000
                            }
                        }
                        }
```

```
                    ]
                    }
                ],
                "OutputGroupSettings": {
                    "Type": "FILE_GROUP_SETTINGS",
                    "FileGroupSettings": {
                    "Destination": "s3://amzn-s3-demo-destination-bucket/"
                    }
                }
                }
            ],
            "Inputs": [
                {
                "AudioSelectors": {
                    "Audio Selector 1": {
                    "DefaultSelection": "DEFAULT"
                    }
                },
                "FileInput": "s3://amzn-s3-demo-bucket/DOC-EXAMPLE-SOURCE_FILE"
                }
            ]
            }
        },
        "End": true
        }
    }
}
```

Parameters in Step Functions are expressed in PascalCase

Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

Optimized MediaConvert APIs

- `CreateJob`
  - Request syntax
  - Supported parameters:

  - Role (Required)
  - Settings (Required)
  - CreateJobRequest (Optional)
- Response syntax – see **CreateJobResponse schema**

IAM policies for calling AWS Elemental MediaConvert

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

The IAM policy for `GetJob` and `CancelJob` actions are scoped to only permit access to jobs with the `ManagedByService: AWSStepFunctions` tag.

> **Tag-based policy**
>
> Modifying the autogenerated `ManagedByService: AWSStepFunctions` tag will cause state machine executions to fail.

Run a Job (.sync)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "MediaConvertCreateJob",
            "Effect": "Allow",
            "Action": [
                "mediaconvert:CreateJob"
            ],
            "Resource": [
                "arn:aws:mediaconvert:us-east-1:123456789012:queues/*",
                "arn:aws:mediaconvert:us-east-1:123456789012:jobTemplates/*",
                "arn:aws:mediaconvert:us-east-1:123456789012:presets/*"
            ]
        },
        {
```

```
            "Sid": "MediaConvertManageJob",
            "Effect": "Allow",
            "Action": [
                "mediaconvert:GetJob",
                "mediaconvert:CancelJob"
            ],
            "Resource": "arn:aws:mediaconvert:us-east-1:123456789012:jobs/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/ManagedByService": "AWSStepFunctions"
                }
            }
        },
        {
            "Sid": "IamPassRole",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::123456789012:role/myRoleName"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "mediaconvert.amazonaws.com"
                    ]
                }
            }
        },
        {
            "Sid": "EventBridgeManageRule",
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
                "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventsForMediaConvertJobRule"
            ]
        }
    ]
```

```
}
```

Request Response

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "MediaConvertCreateJob",
            "Effect": "Allow",
            "Action": [
                "mediaconvert:CreateJob"
            ],
            "Resource": [
                "arn:aws:mediaconvert:us-east-1:123456789012:queues/*",
                "arn:aws:mediaconvert:us-east-1:123456789012:jobTemplates/*",
                "arn:aws:mediaconvert:us-east-1:123456789012:presets/*"
            ]
        },
        {
            "Sid": "IamPassRole",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::123456789012:role/myRoleName"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "mediaconvert.amazonaws.com"
                    ]
                }
            }
        }
    ]
}
```

Create and manage Amazon SageMaker AI jobs with Step Functions

Learn how to use Step Functions to create and manage jobs on SageMaker AI. This page lists the supported SageMaker AI API actions and provides example `Task` states to create SageMaker AI transform, training, labeling, and processing jobs.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

Key features of Optimized SageMaker AI integration

- The Run a Job (.sync) integration pattern is supported.
- There are no specific optimizations for the Request Response integration pattern.
- The Wait for a Callback with Task Token integration pattern is not supported.

Optimized SageMaker AI APIs

- `CreateEndpoint`
- `CreateEndpointConfig`
- `CreateHyperParameterTuningJob` - Supports the `.sync` integration pattern.
- `CreateLabelingJob` - Supports the `.sync` integration pattern.
- `CreateModel`
- `CreateProcessingJob` - Supports the `.sync` integration pattern.
- `CreateTrainingJob` - Supports the `.sync` integration pattern.
- `CreateTransformJob` - Supports the `.sync` integration pattern.
- `UpdateEndpoint`

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].

SageMaker AI Transform Job Example

The following includes a `Task` state that creates an Amazon SageMaker AI transform job, specifying the Amazon S3 location for `DataSource` and `TransformOutput`.

```
{
"SageMaker CreateTransformJob": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sagemaker:createTransformJob.sync",
  "Arguments": {
    "ModelName": "SageMakerCreateTransformJobModel-9iFBKsYti9vr",
    "TransformInput": {
      "CompressionType": "None",
      "ContentType": "text/csv",
      "DataSource": {
        "S3DataSource": {
          "S3DataType": "S3Prefix",
          "S3Uri": "s3://amzn-s3-demo-source-bucket1/TransformJobDataInput.txt"
        }
      }
    },
    "TransformOutput": {
      "S3OutputPath": "s3://amzn-s3-demo-source-bucket1/TransformJobOutputPath"
    },
    "TransformResources": {
      "InstanceCount": 1,
      "InstanceType": "ml.m4.xlarge"
    },
    "TransformJobName": "sfn-binary-classification-prediction"
  },
  "Next": "ValidateOutput"
},
```

SageMaker AI Training Job Example

The following includes a `Task` state that creates an Amazon SageMaker AI training job.

```
{
   "SageMaker CreateTrainingJob":{
      "Type":"Task",
      "Resource":"arn:aws:states:::sagemaker:createTrainingJob.sync",
      "Arguments":{
         "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://amzn-s3-demo-destination-bucket1/doc-search/model"
         },
         "StoppingCondition":{
            "MaxRuntimeInSeconds":100000
         },
         "RoleArn":"arn:aws:iam::account-id:role/docsearch-stepfunction-iam-role",
         "InputDataConfig":[
            {
               "ChannelName":"train",
               "DataSource":{
                  "S3DataSource":{
                     "S3DataType":"S3Prefix",
```

```
                    "S3Uri":"s3://amzn-s3-demo-destination-bucket1/doc-search/interim-
data/training-data/",
                    "S3DataDistributionType":"FullyReplicated"
                 }
              }
           }
        ]
     },
     "Retry":[
        {
           "ErrorEquals":[
              "SageMaker.AmazonSageMakerException"
           ],
           "IntervalSeconds":1,
           "MaxAttempts":100,
           "BackoffRate":1.1
        },
        {
           "ErrorEquals":[
              "SageMaker.ResourceLimitExceededException"
           ],
           "IntervalSeconds":60,
           "MaxAttempts":5000,
           "BackoffRate":1
        },
        {
           "ErrorEquals":[
              "States.Timeout"
           ],
           "IntervalSeconds":1,
           "MaxAttempts":5,
           "BackoffRate":1
        }
     ],
     "Catch":[
        {
           "ErrorEquals":[
              "States.ALL"
           ],
           "Next":"Sagemaker Training Job Error"
        }
     ],
     "Next":"Delete Interim Data Job"
  }
```

```
}
```

SageMaker AI Labeling Job Example

The following includes a `Task` state that creates an Amazon SageMaker AI labeling job.

```
{
  "StartAt": "SageMaker CreateLabelingJob",
  "TimeoutSeconds": 3600,
  "States": {
    "SageMaker CreateLabelingJob": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sagemaker:createLabelingJob.sync",
      "Arguments": {
        "HumanTaskConfig": {
          "AnnotationConsolidationConfig": {
            "AnnotationConsolidationLambdaArn":
 "arn:aws:lambda:region:123456789012:function:ACS-TextMultiClass"
          },
          "NumberOfHumanWorkersPerDataObject": 1,
          "PreHumanTaskLambdaArn": "arn:aws:lambda:region:123456789012:function:PRE-
TextMultiClass",
          "TaskDescription": "Classify the following text",
          "TaskKeywords": [
            "tc",
            "Labeling"
          ],
          "TaskTimeLimitInSeconds": 300,
          "TaskTitle": "Classify short bits of text",
          "UiConfig": {
            "UiTemplateS3Uri": "s3://amzn-s3-demo-bucket/TextClassification.template"
          },
          "WorkteamArn": "arn:aws:sagemaker:region:123456789012:workteam/private-crowd/
ExampleTesting"
        },
        "InputConfig": {
          "DataAttributes": {
            "ContentClassifiers": [
              "FreeOfPersonallyIdentifiableInformation",
              "FreeOfAdultContent"
            ]
```

```
          },
          "DataSource": {
            "S3DataSource": {
              "ManifestS3Uri": "s3://amzn-s3-demo-bucket/manifest.json"
            }
          }
        },
        "LabelAttributeName": "Categories",
        "LabelCategoryConfigS3Uri": "s3://amzn-s3-demo-bucket/labelcategories.json",
        "LabelingJobName": "example-job-name",
        "OutputConfig": {
          "S3OutputPath": "s3://amzn-s3-demo-bucket/output"
        },
        "RoleArn": "arn:aws:iam::123456789012:role/service-role/AmazonSageMaker-
ExecutionRole",
        "StoppingConditions": {
          "MaxHumanLabeledObjectCount": 10000,
          "MaxPercentageOfInputDatasetLabeled": 100
        }
      },
      "Next": "ValidateOutput"
    },
    "ValidateOutput": {
        "Type": "Choice",
        "Choices": [
            {
                "Next": "Success",
                "Condition": "{% $states.input.LabelingJobArn != '' %}"
            }
        ],
        "Default": "Fail"
        },
        "Success": {
            "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 AI Processing Job Example

The following includes a `Task` state that creates an Amazon SageMaker AI processing job.

```
{
  "StartAt": "SageMaker CreateProcessingJob Sync",
  "TimeoutSeconds": 3600,
  "States": {
    "SageMaker CreateProcessingJob Sync": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sagemaker:createProcessingJob.sync",
      "Arguments": {
        "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::account-id: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."
      }
    }
  }
```

IAM policies for calling Amazon SageMaker AI

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

> **Note**
>
> For these examples, *roleArn* refers to the Amazon Resource Name (ARN) of the IAM role that SageMaker AI 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.

CreateTrainingJob

Static resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",
        "sagemaker:DescribeTrainingJob",
        "sagemaker:StopTrainingJob"
```

```
      ],
      "Resource": [
        "arn:aws:sagemaker:us-east-1:123456789012:training-job/myJobName*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/MyExampleRole"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventsForSageMakerTrainingJobsRule"
      ]
    }
  ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:us-east-1:123456789012:training-job/myJobName*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/MyExampleRole"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}
```

Dynamic resources

.sync or .waitForTaskToken

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob",
        "sagemaker:DescribeTrainingJob",
        "sagemaker:StopTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:us-east-1:123456789012:training-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/MyExampleRole"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
```

```
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventsForSageMakerTrainingJobsRule"
      ]
    }
  ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTrainingJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:us-east-1:123456789012:training-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/MyExampleRole"
      ],
      "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 AI. 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:us-east-1:123456789012:transform-job/myJobName*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
```

```
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/MyExampleRole"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventsForSageMakerTransformJobsRule"
      ]
    }
  ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTransformJob"
      ],
      "Resource": [
```

```
        "arn:aws:sagemaker:us-east-1:123456789012:transform-job/myJobName*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/MyExampleRole"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}
```

Dynamic resources

Run a Job (.sync)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTransformJob",
        "sagemaker:DescribeTransformJob",
```

```
        "sagemaker:StopTransformJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:us-east-1:123456789012:transform-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/MyExampleRole"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "events:PutTargets",
        "events:PutRule",
        "events:DescribeRule"
      ],
      "Resource": [
        "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventsForSageMakerTransformJobsRule"
      ]
    }
  ]
}
```

Request Response and Callback (.waitForTaskToken)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:CreateTransformJob"
      ],
      "Resource": [
        "arn:aws:sagemaker:us-east-1:123456789012:transform-job/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:ListTags"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/MyExampleRole"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "sagemaker.amazonaws.com"
        }
      }
    }
  ]
}
```

Publish messages to an Amazon SNS topic with Step Functions

Learn how to use Step Functions to publish messages to an Amazon SNS topic. This page lists the supported Amazon SNS API actions and provides example `Task` states to publish message to Amazon SNS.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

> **Key features of Optimized Amazon SNS integration**
>
> There are no specific optimizations for the Request Response or Wait for a Callback with Task Token integration patterns.

The following includes a `Task` state that publishes to an Amazon Simple Notification Service (Amazon SNS) topic.

```
{
 "StartAt": "Publish to SNS",
 "States": {
   "Publish to SNS": {
     "Type": "Task",
     "Resource": "arn:aws:states:::sns:publish",
     "Arguments": {
       "TopicArn": "arn:aws:sns:region:account-id:myTopic",
       "Message": "{% states.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:

```
{
  "message": "Hello world",
  "SNSDetails": {
    "attribute1": "some value",
    "attribute2": "some other value",
  }
}
```

The following sets values using JSONata expressions for the `StringValue` fields:

```
"MessageAttributes": {
  "my_attribute_no_1": {
      "DataType": "String",
      "StringValue": "{% $states.input.SNSDetails.attribute1 %}"
  },
  "my_attribute_no_2": {
      "DataType": "String",
      "StringValue": "{% $states.input.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 Task Token](#).

```
{
   "StartAt":"Send message to SNS",
   "States":{
      "Send message to SNS":{
         "Type":"Task",
         "Resource":"arn:aws:states:::sns:publish.waitForTaskToken",
         "Arguments":{
            "TopicArn":"arn:aws:sns:region:account-id:myTopic",
            "Message":{
               "Input":"{% states.input.message %}",
               "TaskToken": "{% $states.context.Task.Token %}"
            }
         },
```

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

Optimized Amazon SNS APIs

- Publish

> **Parameters in Step Functions are expressed in PascalCase**
>
> Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

> **Quota for input or result data**
>
> When sending or receiving data between services, the maximum input or result for a task is 256 KiB of data as a UTF-8 encoded string. See Quotas related to state machine executions.

IAM policies for calling Amazon SNS

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
```

```
            ],
            "Resource": [
                "arn:aws:sns:us-east-1:123456789012:myTopicName"
            ]
        }
    ]
}
```

Resources based on a Path, or publishing to `TargetArn` or `PhoneNumber`

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": "*"
        }
    ]
}
```

Send messages to an Amazon SQS queue with Step Functions

You can to send messages to an Amazon SQS queue using the following Amazon SQS API actions and example `Task` state code for Step Functions workflows.

To learn about integrating with AWS services in Step Functions, see Integrating services and Passing parameters to a service API in Step Functions.

To learn more about receiving messages in Amazon SQS, see Receive and Delete Your Message in the *Amazon Simple Queue Service Developer Guide*.

The following sample includes a `Task` state (JSONata) that sends an Amazon Simple Queue Service (Amazon SQS) message with optional **MessageAttributes**:

```
{
 "StartAt": "Send to SQS",
```

```
  "States": {
    "Send to SQS": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage",
      "Arguments": {
        "QueueUrl": "https://sqs.us-east-1.amazonaws.com/account-id/myQueue",
        "MessageBody": "{% $states.input.message %}",
        "MessageAttributes": {
          "my_attribute_no_1": {
            "DataType": "String",
            "StringValue": "attribute1"
          },
          "my_attribute_no_2": {
            "DataType": "String",
            "StringValue": "attribute2"
          }
        }
      },
      "End": true
     }
   }
}
```

The following state machine 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 Task Token.

```
{
   "StartAt":"Send message to SQS",
   "States":{
      "Send message to SQS":{
         "Type":"Task",
         "Resource":"arn:aws:states:::sqs:sendMessage.waitForTaskToken",
         "Arguments":{
            "QueueUrl":"https://sqs.us-east-1.amazonaws.com/account-id/myQueue",
            "MessageBody":{
               "Input" : "{% $states.input.message %}",
               "MyTaskToken" : "{% $states.context.Task.Token %}"
            }
         },
         "End":true
      }
   }
}
```

Optimized Amazon SQS APIs

- [SendMessage]

> **Parameters in Step Functions are expressed in PascalCase**
>
> Even if the native service API is in camelCase, for example the API action `startSyncExecution`, you specify parameters in PascalCase, such as: `StateMachineArn`.

> **Quota for input or result data**
>
> When sending or receiving data between services, the maximum input or result for a task is 256 KiB of data as a UTF-8 encoded string. See Quotas related to state machine executions.

IAM policies for calling Amazon SQS

The following example templates show how AWS Step Functions generates IAM policies based on the resources in your state machine definition. For more information, see How Step Functions generates IAM policies for integrated services and Discover service integration patterns in Step Functions.

Static resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sqs:SendMessage"
            ],
            "Resource": [
                "arn:aws:sqs:us-east-1:123456789012:myQueueName"
            ]
        }
    ]
```

```
}
```

Dynamic resources

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sqs:SendMessage"
            ],
            "Resource": "*"
        }
    ]
}
```

Start a new AWS Step Functions state machine from a running execution

Step Functions integrates with its own API as a service integration. Learn how to use 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 to reduce the complexity of your main workflows and to reuse common processes.

Key features of Optimized Step Functions integration

- The Run a Job (.sync) integration pattern is available.

For more information, see the following:

- Start from a Task
- Integrating services
- Passing parameters to a service API in Step Functions

Optimized Step Functions APIs

- [StartExecution]

Workflow Examples

The following includes a `Task` state that starts an execution of another state machine and waits for it to complete.

```
{
   "Type":"Task",
   "Resource":"arn:aws:states:::states:startExecution.sync:2",
   "Arguments":{
      "Input":{
        "Comment": "Hello world!"
       },
      "StateMachineArn":"arn:aws:states:region:account-id:stateMachine:HelloWorld",
      "Name":"ExecutionName"
   },
   "End":true
}
```

The following includes a `Task` state that starts an execution of another state machine.

```
{
   "Type":"Task",
   "Resource":"arn:aws:states:::states:startExecution",
   "Arguments":{
      "Input":{
        "Comment": "Hello world!"
       },
      "StateMachineArn":"arn:aws:states:region:account-id:stateMachine:HelloWorld",
      "Name":"ExecutionName"
   },
   "End":true
}
```

The following includes a `Task` state that implements the callback service integration pattern.

```
{
```

```
    "Type":"Task",
    "Resource":"arn:aws:states:::states:startExecution.waitForTaskToken",
    "Arguments":{
       "Input":{
         "Comment": "Hello world!",
         "token": "{% $states.context.Task.Token %}"
        },
       "StateMachineArn":"arn:aws:states:region:account-id:stateMachine:HelloWorld",
       "Name":"ExecutionName"
    },
    "End":true
}
```

To associate a nested workflow execution with the parent execution that started it, pass a specially named parameter that includes the execution ID pulled from the Context object. 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.

```
{
    "Type":"Task",
    "Resource":"arn:aws:states:::states:startExecution.sync",
    "Arguments":{
       "Input":{
         "Comment": "Hello world!",
         "AWS_STEP_FUNCTIONS_STARTED_BY_EXECUTION_ID.$": "$$.Execution.Id"
        },
       "StateMachineArn":"arn:aws:states:region:account-id:stateMachine:HelloWorld",
       "Name":"ExecutionName"
    },
    "End":true
}
```

Nested state machines return the following:

Resource	Output
startExecution.sync	String
startExecution.sync:2	JSON

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:

```
{
   <other fields>
   "Output": "{ \"MyKey\": \"MyValue\" }"
}
```

For startExecution.sync:2:

```
{
   <other fields>
   "Output": {
      "MyKey": "MyValue"
   }
}
```

Configuring IAM permissions for nested state machines

A parent state machine determines if a child state machine has completed execution using polling and events. Polling requires permission for `states:DescribeExecution` while events sent through EventBridge to Step Functions require permissions for `events:PutTargets`, `events:PutRule`, and `events:DescribeRule`. If these permissions are missing from your IAM role, there may be a delay before a parent state machine becomes aware of the completion of the child state machine's execution.

For a state machine that calls `StartExecution` for a single nested workflow execution, use an IAM policy that limits permissions to that state machine.

IAM policies for calling nested Step Functions workflows

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:us-east-1:123456789012:stateMachine:myStateMachineName"
            ]
        }
    ]
}
```

For more information, see the following:

- Integrating services with Step Functions
- Passing parameters to a service API in Step Functions
- Start a new AWS Step Functions state machine from a running execution

Synchronous

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "states:StartExecution"
            ],
            "Resource": [
                "arn:aws:states:us-
east-1:123456789012:stateMachine:stateMachineName"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "states:DescribeExecution",
                "states:StopExecution"
            ],
            "Resource": [
               "arn:aws:states:us-
east-1:123456789012:execution:myStateMachineName:*"
```

```
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "events:PutTargets",
                "events:PutRule",
                "events:DescribeRule"
            ],
            "Resource": [
               "arn:aws:events:us-east-1:123456789012:rule/
StepFunctionsGetEventsForStepFunctionsExecutionRule"
            ]
        }
    ]
}
```

Asynchronous

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "states:StartExecution"
            ],
            "Resource": [
                "arn:aws:states:us-
east-1:123456789012:stateMachine:myStateMachineName"
            ]
        }
    ]
}
```

ARN types required

In the policy for **Synchronous**, note that `states:StartExecution` requires a state machine ARN whereas `states:DescribeExecution` and `states:StopExecution` require an execution ARN.

If you mistakenly combine all three actions, the JSON will be valid but the IAM policy will be incorrect. An incorrect policy can cause stuck workflows and/or access issues during workflow execution.

For more information about nested workflow executions, see Start workflow executions from a task state in Step Functions.

Security in AWS Step Functions

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to AWS Step Functions, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Step Functions. The following topics show you how to configure Step Functions to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Step Functions resources.

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 and encryption in 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. You are also responsible for 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 users with AWS IAM Identity Center or 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 require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- 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 sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 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) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text 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 text 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.

With customer managed AWS KMS keys, you can secure customer data that includes **protected health information (PHI)** from unauthorized access. Step Functions is integrated with CloudTrail, so you can view and audit the most recent events in the CloudTrail console in the event history.

Topics

- Data at rest encryption in Step Functions
- Data in transit encryption in Step Functions

Data at rest encryption in Step Functions

Read the blog

Read about customer managed keys in [Strengthening data security with a customer-managed AWS KMS key]

AWS Step Functions always encrypts your data at rest using transparent server-side encryption. Encryption of data at rest by default reduces the operational overhead and complexity involved in protecting sensitive data. You can build security-sensitive applications that meet strict encryption compliance and regulatory requirements.

Although you can't disable this layer of encryption or select an alternate encryption type, you can add a second layer of encryption over the existing AWS owned encryption keys by choosing a customer managed key when you create your state machine and activity resources:

- **Customer managed keys** — Step Functions supports the use of a symmetric customer managed key that you create, own, and manage to add a second layer of encryption over the existing AWS owned encryption. Because you have full control of this layer of encryption, you can perform such tasks as:
  - Establishing and maintaining key policies
  - Establishing and maintaining IAM policies and grants
  - Enabling and disabling key policies
  - Rotating key cryptographic material
  - Adding tags
  - Creating key aliases
  - Scheduling keys for deletion

  For information, see customer managed key in the *AWS Key Management Service Developer Guide*.

You can encrypt your data using a **customer-managed key** for AWS Step Functions state machines and activities. You can configure a symmetric AWS KMS key and data key reuse period when creating or updating a **State Machine**, and when creating an **Activity**. The execution history and state machine definition will be encrypted with the key applied to the State Machine. Activity inputs will be encrypted with the key applied to the Activity.

With customer managed AWS KMS keys, you can secure customer data that includes **protected health information (PHI)** from unauthorized access. Step Functions is integrated with CloudTrail, so you can view and audit the most recent events in the CloudTrail console in the event history.

For information on AWS KMS, see What is AWS Key Management Service?

> **Note**
>
> Step Functions automatically enables encryption at rest using AWS owned keys at no charge. However, AWS KMS charges apply when using a customer managed key. For information about pricing, see AWS Key Management Service pricing.

Encrypting with a customer managed key

Step Functions decrypts payload data with your customer managed AWS KMS key before passing it to another service to perform a task. The data is encrypted in transit using Transport Layer Security (TLS).

When data is returned from an integrated service, Step Functions encrypts the data with your customer managed AWS KMS key. You can use the same key to consistently apply encryption across many AWS services.

You can use a customer managed key with the following resources:

- **State Machine** - both Standard and Express workflow types
- **Activity**

You can specify the data key by entering a **KMS key ID**, which Step Functions uses to encrypt your data.

- **KMS key ID** — key identifier for an AWS KMS customer managed key in the form of a key ID, key ARN, alias name, or alias ARN.

Create a State Machine with a customer managed key

Prerequisite: Before you can create a state machine with customer managed AWS KMS keys, your user or role must have AWS KMS permissions to `DescribeKey` and `GenerateDataKey`.

You can perform the following steps in the AWS console, through the API, or by provisioning infrastructure through AWS CloudFormation resources. (CloudFormation examples are presented later in this guide.)

Step 1: Create AWS KMS key

You can create a symmetric customer managed key with the AWS KMS console or AWS KMS APIs.

To create a symmetric customer managed key

Follow the steps for [Creating symmetric customer managed key](#) in the *AWS Key Management Service Developer Guide*.

> **Note**
>
> *Optional*: When creating a key, you may choose **Key administrators**. The selected users or roles will be granted access manage the key, such as enabling or disabling the key through the API. You may also choose **Key users**. These users or roles will be granted the ability to use the AWS KMS key in cryptographic operations.

Step 2: Set AWS KMS key policy

Key policies control access to your customer managed key. Every customer managed key must have exactly one key policy, which contains statements that determine who can use the key and how they can use it. When you create your customer managed key, you can specify a key policy. For information, see [Managing access to customer managed keys](#) in the *AWS Key Management Service Developer Guide*.

The following is an example AWS KMS key policy from console, without **Key administrators** or **Key users**:

```
{
  "Version": "2012-10-17",
  "Id": "key-consolepolicy-1",
  "Statement": [
```

```
    {
      "Sid": "Enable IAM User Permissions for the key",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:root"
      },
      "Action": "kms:*",
      "Resource": "*"
    }
  ]
}
```

See the *AWS Key Management Service Developer Guide* for information about specifying permissions in a policy and troubleshooting key access.

Step 3: Add key policy to encrypt CloudWatch logs

Step Functions is integrated with CloudWatch for logging and monitoring. When you enable server-side encryption for your state machine using your own KMS key and enable CloudWatch Log integration, you must allow **`delivery.logs.amazonaws.com`** to do `kms:Decrypt` action from your AWS KMS key policy:

```
{
  "Sid": "Enable log service delivery for integrations",
  "Effect": "Allow",
  "Principal": {
    "Service": "delivery.logs.amazonaws.com"
  },
  "Action": "kms:Decrypt",
  "Resource": "*"
}
```

If you enable state machine encryption with a AWS KMS key, and your state machine has CloudWatch Logs integration enabled, the state machine's execution role needs the following policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowKMSPermissionForCloudWatchLogGroup",
      "Effect": "Allow",
```

```
      "Action": "kms:GenerateDataKey",
      "Resource": "arn:aws:kms:us-east-1:123456789012:key/keyId",
      "Condition": {
        "StringEquals": {
          "kms:EncryptionContext:SourceArn": "arn:aws:logs:us-east-1:123456789012:*"
        }
      }
    }
  ]
}
```

Step 4: Encrypt CloudWatch Log Group *(Optional)*

You can enable encryption of the logs in a CloudWatch Log Group using your own AWS KMS key. To do that, you must also add the following policy to that AWS KMS key.

> **Note**
>
> You can choose the same or different AWS KMS keys to encrypt your logs and your state machine definitions.

```
{
  "Id": "key-consolepolicy-logging",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Enable log service for a single log group",
      "Effect": "Allow",
      "Principal": {
        "Service": "logs.us-east-1.amazonaws.com"
      },
      "Action": [
        "kms:Encrypt*",
        "kms:Decrypt*",
        "kms:ReEncrypt*",
        "kms:GenerateDataKey*",
        "kms:Describe*"
      ],
      "Resource": "*",
      "Condition": {
        "ArnEquals": {
```

```
          "kms:EncryptionContext:aws:logs:arn": "arn:aws:logs:us-
east-1:123456789012:log-group:LOG_GROUP_NAME"
        }
      }
    }
  ]
}
```

Note

The `Condition` section restricts the AWS KMS key to a single log group ARN.

Note

See [CloudWatch logs documentation](#) to learn more about setting permissions on the AWS KMS key for your log group.

Step 5: Create state machine

After you have created a key and set up the policy, you can use the key to create a new state machine.

When creating the state machine, choose **Additional configuration**, then choose to encrypt with customer managed key. You can then select your key and set the data key reuse period from 1 min to 15 minutes.

Optionally, you can enable logging by setting a log level and choosing to encrypt the log group with your AWS KMS key.

Note

You can only enable encryption on a **new log group** in the Step Functions console. To learn how to associate a AWS KMS key with an existing log group, see [Associate a AWS KMS key with a log group](#).

To successfully start an execution for Standard workflows and Asynchronous Express workflows with customer managed keys, your execution role requires `kms:Decrypt` and `kms:GenerateDataKey` permissions. The execution role for Synchronous Express execution requires `kms:Decrypt`. When you create a state machine in the console and choose **Create a new role**, these permissions are automatically included for you.

The following is an example execution role policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowKMSPermissionsForStepFunctionsWorkflowExecutions",
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey"
      ],
      "Resource": [
        "arn:aws:kms:us-east-1:123456789012:key/keyId"
      ],
      "Condition": {
        "StringEquals": {
          "kms:EncryptionContext:aws:states:stateMachineArn": [
            "arn:aws:states:us-east-1:123456789012:stateMachine:stateMachineName"
          ]
        }
      }
    }
  ]
}
```

Step 6: Invoke state machine encrypted with your AWS KMS key

You can invoke your encrypted state machine as you normally would, and your data will be encrypted with your customer managed key.

Create an Activity with a customer managed key

Creating an Step Functions Activity with a customer managed key is similar to creating a state machine with a customer managed key. Before you can create a activity with customer managed AWS KMS keys, your user or role only needs AWS KMS permissions to `DescribeKey`. During creation of the Activity, you choose the key and set the encryption configuration parameters.

Note that Step Functions Activity resources remain **immutable**. You cannot update the `encryptionConfiguration` for an activity ARN of an existing activity; you must create a new Activity resource. Callers to the Activity API endpoints must have `kms:DescribeKey` permissions to successfully create an activity with a AWS KMS key.

When customer managed key encryption is enabled on an Activity Task, the state machine execution role will require `kms:GenerateDataKey` and `kms:Decrypt` permission for the activity key. If you are creating this state machine from the Step Functions console, the auto role creation feature will add these permissions.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowKMSPermissionsForStepFunctionsActivities",
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey"
      ],
      "Resource": [
        "arn:aws:kms:us-east-1:123456789012:key/keyId"
      ],
      "Condition": {
        "StringEquals": {
          "kms:EncryptionContext:aws:states:activityArn": [
            "arn:aws:states:us-east-1:123456789012:activity:activityName"
          ]
        }
      }
    }
  ]
}
```

Scope down AWS KMS permission policies with conditions

You can use the *encryption context* in key policies and IAM policies as `conditions` to control access to your symmetric customer managed key. To limit the use of a AWS KMS key to requests from Step Functions on behalf of a specific role, you can use the `kms:ViaService` condition.

Scoping with encryption context

An encryption context is an optional set of key-value pairs that contain additional contextual information about the data.

AWS KMS uses the encryption context as additional authenticated data to support authenticated encryption. When you include an encryption context in a request to encrypt data, AWS KMS binds the encryption context to the encrypted data. To decrypt data, you include the same encryption context in the request.

Step Functions provides an encryption context in AWS KMS cryptographic operations, where the key is `aws:states:stateMachineArn` for State Machines or `aws:states:activityArn` for Activities, and the value is the resource Amazon Resource Name (ARN).

Example

```
"encryptionContext": {"aws:states:stateMachineArn": "arn:aws:states:region:account-
id:stateMachine:stateMachineName"}
```

Example

```
"encryptionContext": {"aws:states:activityArn": "arn:aws:states:region:account-
id:activity:activityName"}
```

The following example shows how to limit the use of a AWS KMS key for execution roles to specific state machines with `kms:EncryptionContext` and the `aws:states:stateMachineArn` context key:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowKeyManagement",
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey"
      ],
      "Resource": [
        "arn:aws:kms:us-east-1:123456789012:key/keyId"
```

```
      ],
      "Condition": {
        "StringEquals": {
          "kms:EncryptionContext:aws:states:stateMachineArn": "arn:aws:states:us-east-1:123456789012:stateMachine:stateMachineName"
        }
      }
    }
  ]
}
```

Scoping with kms:ViaService

The `kms:ViaService` condition key limits use of an AWS Key Management Service key to requests from specified AWS services.

The following example policy uses the `kms:ViaService` condition to allow the AWS KMS key to be used for specific actions only when the request originates from Step Functions in the `ca-central-1` region, acting on behalf of the ExampleRole:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Allow access for Key Administrators in a region",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:role/ExampleRole"
      },
      "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "kms:ViaService": "states.us-east-1.amazonaws.com"
        }
      }
    }
  ]
}
```

> **Note**
>
> The `kms:ViaService` condition is only applicable when AWS KMS permissions are required by the API caller (for example, `CreateStateMachine`, `CreateActivity`, `GetActivityTask`, etc.). Adding `kms:ViaService` condition to an **execution role** can prevent a new execution from starting or cause a running execution to fail.

Required permissions for API callers

To call Step Functions API actions that return encrypted data, callers need AWS KMS permissions. Alternatively, some API actions have an option (`METADATA_ONLY`) to return only metadata, removing the requirement for AWS KMS permissions. Refer to the Step Functions API for information.

For an execution to successfully complete when using customer managed key encryption, the execution role needs to be granted `kms:GenerateDataKey` and `kms:Decrypt` permissions for AWS KMS keys used by the state machine.

The following table shows the AWS KMS permissions you need to provide to Step Functions API callers for the APIs using a **State Machine's AWS KMS key**. You can provide the permissions to the key policy or IAM policy for the role.

APIs using State Machine's AWS KMS key	Required by Caller
CreateStateMachine	kms:DescribeKey, kms:GenerateDataKey
UpdateStateMachine	kms:DescribeKey, kms:GenerateDataKey
DescribeStateMachine	kms:Decrypt
DescribeStateMachineForExecution	kms:Decrypt
StartExecution	--
StartSyncExecution	kms:Decrypt
SendTaskSuccess	--
SendTaskFailure	--

StopExecution	--
RedriveExecution	--
DescribeExecution	kms:Decrypt
GetExecutionHistory	kms:Decrypt

The following table shows the AWS KMS permissions you need to provide to Step Functions API callers for the APIs using an **Activity's AWS KMS key**. You can provide the permissions in the key policy or IAM policy for the role.

APIs using Activity's AWS KMS key	Required by Caller
CreateActivity	kms:DescribeKey
GetActivityTask	kms:Decrypt

When do I grant permissions to the Caller or the Execution role?

When an IAM role or user calls the Step Functions API, the Step Functions service calls AWS KMS on behalf of the API caller. In this case, you must grant AWS KMS permission to the API caller. When an execution role calls AWS KMS directly, you must grant AWS KMS permissions on the execution role.

AWS CloudFormation resources for encryption configuration

AWS CloudFormation resource types for Step Functions can provision state machine and activity resources with encryption configurations.

By default, Step Functions provides transparent server-side encryption. Both `AWS::StepFunctions::Activity` and `AWS::StepFunctions::StateMachine` accept an optional `EncryptionConfiguration` property which can configure a customer managed AWS KMS key for server-side encryption.

Prerequisite: Before you can create a state machine with customer managed AWS KMS keys, your user or role must have AWS KMS permissions to `DescribeKey` and `GenerateDataKey`.

Updates to StateMachine requires No interruption. Updates to Activity resources requires: Replacement.

To declare an **`EncryptionConfiguration`** property in your AWS CloudFormation template, use the following syntax:

JSON

```
{
  "KmsKeyId" : String,
  "KmsDataKeyReusePeriodSeconds" : Integer,
  "Type" : String
}
```

YAML

```
KmsKeyId: String
KmsDataKeyReusePeriodSeconds: Integer
Type: String
```

Properties

- **Type** - Encryption option for the state machine or activity. *Allowed values*: `CUSTOMER_MANAGED_KMS_KEY` | `AWS_OWNED_KEY`
- **KmsKeyId** - Alias, alias ARN, key ID, or key ARN of the symmetric encryption AWS KMS key that encrypts the data key. To specify a AWS KMS key in a different AWS account, the customer must use the key ARN or alias ARN. For information regarding kmsKeyId, see KeyId in AWS KMS docs.
- **KmsDataKeyReusePeriodSeconds** - Maximum duration for which SFN will reuse data keys. When the period expires, Step Functions will call `GenerateDataKey`. This setting can only be set when **Type** is `CUSTOMER_MANAGED_KMS_KEY`. The value can range from 60-900 seconds. Default is 300 seconds.

AWS CloudFormation examples

Example: StateMachine with customer managed key

```
AWSTemplateFormatVersion: '2010-09-09'
```

```
Description: An example template for a Step Functions State Machine.
Resources:
  MyStateMachine:
    Type: AWS::StepFunctions::StateMachine
    Properties:
      StateMachineName: HelloWorld-StateMachine
      Definition:
        StartAt: PassState
        States:
          PassState:
            Type: Pass
            End: true
      RoleArn: !Sub "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/example"
      EncryptionConfiguration:
        KmsKeyId: !Ref MyKmsKey
        KmsDataKeyReusePeriodSeconds: 100
        Type: CUSTOMER_MANAGED_KMS_KEY

  MyKmsKey:
    Type: AWS::KMS::Key
    Properties:
      Description: Symmetric KMS key used for encryption/decryption
```

Example: Activity with customer managed key

```
AWSTemplateFormatVersion: '2010-09-09'
Description: An example template for a Step Functions Activity.
Resources:
  Activity:
    Type: AWS::StepFunctions::Activity
    Properties:
      Name: ActivityWithKmsEncryption
      EncryptionConfiguration:
        KmsKeyId: !Ref MyKmsKey
        KmsDataKeyReusePeriodSeconds: 100
        Type: CUSTOMER_MANAGED_KMS_KEY

  MyKmsKey:
    Type: AWS::KMS::Key
    Properties:
      Description: Symmetric KMS key used for encryption/decryption
```

Updating encryption for an Activity requires creating a new resource

Activity configuration is immutable, and resource names must be unique. To set customer managed keys for encryption, you must create a **new Activity**. If you attempt to change the configuration in your CFN template for an existing activity, you will receive an `ActivityAlreadyExists` exception.

To update your activity to include customer managed keys, set a new activity name within your CFN template. The following shows an example that creates a new activity with a customer managed key configuration:

Existing activity definition

```
AWSTemplateFormatVersion: '2010-09-09'
  Description: An example template for a new Step Functions Activity.
  Resources:
    Activity:
    Type: AWS::StepFunctions::Activity
    Properties:
      Name: ActivityName
      EncryptionConfiguration:
        Type: AWS_OWNED_KEY
```

New activity definition

```
AWSTemplateFormatVersion: '2010-09-09'
  Description: An example template for a Step Functions Activity.
  Resources:
    Activity:
      Type: AWS::StepFunctions::Activity
      Properties:
        Name: ActivityWithKmsEncryption
        EncryptionConfiguration:
          KmsKeyId: !Ref MyKmsKey
          KmsDataKeyReusePeriodSeconds: 100
          Type: CUSTOMER_MANAGED_KMS_KEY

    MyKmsKey:
      Type: AWS::KMS::Key
      Properties:
        Description: Symmetric KMS key used for encryption/decryption
```

Monitoring your encryption key usage

When you use an AWS KMS customer managed key to encrypt your Step Functions resources, you can use CloudTrail to track requests that Step Functions sends to AWS KMS.

You can also use the encryption context in audit records and logs to identify how the customer managed key is being used. The encryption context also appears in logs generated by AWS CloudTrail.

The following examples are CloudTrail events for `Decrypt`, `DescribeKey`, and `GenerateDataKey` to monitor AWS KMS operations called by Step Functions to access data encrypted by your customer managed key:

Decrypt

When you access an encrypted state machine or activity, Step Functions calls the `Decrypt` operation to use the stored encrypted data key to access the encrypted data.

The following example event records the `Decrypt` operation:

```
{
  "eventVersion": "1.09",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "111122223333:Sampleuser01",
    "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
    "accountId": "111122223333",
    "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "111122223333:Sampleuser01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
        "accountId": "111122223333",
        "userName": "Admin"
      },
      "attributes": {
        "creationDate": "2024-07-05T21:06:27Z",
        "mfaAuthenticated": "false"
      }
    },
    "invokedBy": "states.amazonaws.com"
  },
```

```
  "eventTime": "2024-07-05T21:12:21Z",
  "eventSource": "kms.amazonaws.com",
  "eventName": "Decrypt",
  "awsRegion": "aa-example-1",
  "sourceIPAddress": "states.amazonaws.com",
  "userAgent": "states.amazonaws.com",
  "requestParameters": {
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT",
        "keyId": "arn:aws:kms:aa-example-1:111122223333:key/a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "encryptionContext": {
            "aws:states:stateMachineArn": "arn:aws:states:aa-example-1:111122223333:stateMachine:example1"
        }
    },
    "responseElements": null,
    "requestID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
    "eventID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
    "readOnly": true,
    "resources": [
        {
            "accountId": "111122223333",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:aa-example-1:111122223333:key/a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```

DescribeKey

Step Functions uses the `DescribeKey` operation to verify if the AWS KMS customer managed key associated with your State Machine or Activity exists in the account and region.

The following example event records the `DescribeKey`operation:

```
{
  "eventVersion": "1.09",
  "userIdentity": {
    "type": "AssumedRole",
```

```
    "principalId": "111122223333:Sampleuser01",
    "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
    "accountId": "111122223333",
    "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "111122223333:Sampleuser01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
        "accountId": "111122223333",
        "userName": "Admin"
      },
      "attributes": {
        "creationDate": "2024-07-05T21:06:27Z",
        "mfaAuthenticated": "false"
      }
    },
    "invokedBy": "states.amazonaws.com"
  },
  "eventTime": "2024-07-05T21:12:21Z",
  "eventSource": "kms.amazonaws.com",
  "eventName": "DescribeKey",
  "awsRegion": "aa-example-1",
  "sourceIPAddress": "states.amazonaws.com",
  "userAgent": "states.amazonaws.com",
  "requestParameters": {
    "keyId": "arn:aws:kms:aa-example-1:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
  },
  "responseElements": null,
  "requestID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
  "eventID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
  "readOnly": true,
  "resources": [
    {
      "accountId": "111122223333",
      "type": "AWS::KMS::Key",
      "ARN": "arn:aws:kms:aa-example-1:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "111122223333",
```

```
  "eventCategory": "Management",
  "sessionCredentialFromConsole": "true"
}
```

GenerateDataKey

When you enable an AWS KMS customer managed key for your State Machine or Activity, Step Functions sends a `GenerateDataKey` request to get a data key to the encrypt state machine definition or execution data.

The following example event records the `GenerateDataKey`operation:

```
{
  "eventVersion": "1.09",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "111122223333:Sampleuser01",
    "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
    "accountId": "111122223333",
    "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "111122223333:Sampleuser01",
        "arn": "arn:aws:iam::111122223333:role/Admin",
        "accountId": "111122223333",
        "userName": "Admin"
      },
      "attributes": {
        "creationDate": "2024-07-05T21:06:27Z",
        "mfaAuthenticated": "false"
      }
    },
    "invokedBy": "states.amazonaws.com"
  },
  "eventTime": "2024-07-05T21:12:21Z",
  "eventSource": "kms.amazonaws.com",
  "eventName": "GenerateDataKey",
  "awsRegion": "aa-example-1",
  "sourceIPAddress": "states.amazonaws.com",
  "userAgent": "states.amazonaws.com",
  "requestParameters": {
    "keySpec": "AES_256",
```

```
    "encryptionContext": {
      "aws:states:stateMachineArn": "arn:aws:states:aa-
example-1:111122223333:stateMachine:example1"
    },
    "keyId": "arn:aws:kms:aa-example-1:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
  },
  "responseElements": null,
  "requestID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
  "eventID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
  "readOnly": true,
  "resources": [
    {
      "accountId": "111122223333",
      "type": "AWS::KMS::Key",
      "ARN": "arn:aws:kms:aa-example-1:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "111122223333",
  "eventCategory": "Management"
}
```


FAQs

What happens if my key is marked for deletion or deleted in AWS KMS?

If the key is deleted or marked for deletion in AWS KMS, any related running executions will fail. New executions cannot be started until you remove or change the key associated with the workflow. After a AWS KMS key is deleted, all encrypted data associated with the workflow execution will remain encrypted and can no longer be decrypted, making the data ***unrecoverable***.

What happens if a AWS KMS key is disabled in AWS KMS?

If a AWS KMS key is disabled in AWS KMS, any related running executions will fail. New executions cannot be started. You can no longer decrypt the data encrypted under that disabled AWS KMS key until it is re-enabled.

What happens to Execution Status change events sent to EventBridge?

Execution Input, Output, Error, and Cause will not be included for execution status change events for workflows that are encrypted using your customer managed AWS KMS key.

Learn more

For information about data encryption at rest, see AWS Key Management Service concepts and security best practices for AWS Key Management Service in the *AWS Key Management Service Developer Guide*.

Data in transit encryption in Step Functions

Step Functions encrypts data in transit between the service and other integrated AWS services (see Integrating services with Step Functions). All data that passes between Step Functions and integrated services is encrypted using Transport Layer Security (TLS).

Identity and Access Management in Step Functions

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. For example, you will learn how to provide AWS Step Functions with credentials with permissions to access AWS resources, such as retrieving event data from other AWS resources.

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Step Functions resources. IAM is an AWS service that you can use with no additional charge.

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Step Functions.

Service user – If you use the Step Functions service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Step Functions features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Step Functions, see Troubleshooting identity and access issues in Step Functions.

Service administrator – If you're in charge of Step Functions resources at your company, you probably have full access to Step Functions. It's your job to determine which Step Functions features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Step Functions, see How AWS Step Functions works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Step Functions. To view example Step Functions identity-based policies that you can use in IAM, see Identity-based policy examples for AWS Step Functions.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the

principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries –** A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs) –** SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs) –** RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies –** Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic 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.

- Creating an IAM role for your state machine in Step Functions
- Creating granular permissions for non-admin users in Step Functions
- Creating Amazon VPC endpoints for Step Functions
- How Step Functions generates IAM policies for integrated services
- IAM policies for using Distributed Map states

How AWS Step Functions works with IAM

Before you use IAM to manage access to Step Functions, learn what IAM features are available to use with Step Functions.

The following table lists IAM features that you can use with AWS Step Functions:

IAM feature	Step Functions support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys (service-specific)	Yes

IAM feature	Step Functions support
ACLs	No
ABAC (tags in policies)	Partial
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	No

To get a high-level view of how Step Functions and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Step Functions

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Step Functions

To view examples of Step Functions identity-based policies, see Identity-based policy examples for AWS Step Functions.

Resource-based policies within Step Functions

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for Step Functions

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Step Functions actions, see Resources Defined by AWS Step Functions in the *Service Authorization Reference*.

Policy actions in Step Functions use the following prefix before the action:

```
states
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
          "states:action1",
          "states:action2"
             ]
```

To view examples of Step Functions identity-based policies, see Identity-based policy examples for AWS Step Functions.

Policy resources for Step Functions

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Step Functions resource types and their ARNs, see Actions Defined by AWS Step Functions in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Resources Defined by AWS Step Functions.

To view examples of Step Functions identity-based policies, see Identity-based policy examples for AWS Step Functions.

Policy condition keys for Step Functions

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of Step Functions condition keys, see Condition Keys for AWS Step Functions in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Resources Defined by AWS Step Functions.

If your policy must depend on the Step Functions service principal name, we recommend you check for the existence or non-existence of `states.amazonaws.com` in the `aws:PrincipalServiceNamesList` multivalued context key, rather than the `aws:PrincipalServiceName` condition key. The `aws:PrincipalServiceName` condition key contains only one entry from the list of service principal names and it may not always be `states.amazonaws.com`. The following condition block demonstrates checking for the existence of `states.amazonaws.com`.

```
{
    "Condition": {
        "ForAnyValue:StringEquals": {
            "aws:PrincipalServiceNamesList": "states.amazonaws.com"
        }
    }
}
```

To view examples of Step Functions identity-based policies, see Identity-based policy examples for AWS Step Functions.

ACLs in Step Functions

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with Step Functions

Supports ABAC (tags in policies): Partial

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with Step Functions

Supports temporary credentials: Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see AWS services that work with IAM in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see Switch from a user to an IAM role (console) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM.

Cross-service principal permissions for Step Functions

Supports forward access sessions (FAS): Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

Service roles for Step Functions

Supports service roles: Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Warning**
>
> Changing the permissions for a service role might break Step Functions functionality. Edit service roles only when Step Functions provides guidance to do so.

Service-linked roles for Step Functions

Supports service-linked roles: No

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for AWS Step Functions

By default, users and roles don't have permission to create or modify Step Functions resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Step Functions, including the format of the ARNs for each of the resource types, see Actions, Resources, and Condition Keys for AWS Step Functions in the *Service Authorization Reference*.

Topics

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Step Functions resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.

- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Step Functions console

To access the AWS Step Functions console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Step Functions resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the Step Functions console, also attach the Step Functions *ConsoleAccess* or *ReadOnly* AWS managed policy to the entities. For more information, see [Adding permissions to a user](#) in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policies for AWS Step Functions

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

AWS managed policy: AWSStepFunctionsConsoleFullAccess

You can attach the `AWSStepFunctionsConsoleFullAccess` policy to your IAM identities.

This policy grants *`administrator`*permissions that allow a user access to use the Step Functions console. For a full console experience, a user may also need iam:PassRole permission on other IAM roles that can be assumed by the service.

AWS managed policy: AWSStepFunctionsReadOnlyAccess

You can attach the `AWSStepFunctionsReadOnlyAccess` policy to your IAM identities.

This policy grants *`read-only`* permissions that allow a user or role to list and describe state machines, activities, executions, activities, tags, MapRuns, and state machine alias and versions. This policy also grants permission to check the syntax of state machine definitions that you provide.

AWS managed policy: AWSStepFunctionsFullAccess

You can attach the `AWSStepFunctionsFullAccess` policy to your IAM identities.

This policy grants *`full`* permissions to a user or role to use the Step Functions API. For full access, a user must have *`iam:PassRole`* permission on at least one IAM role that can be assumed by the service.

Step Functions updates to AWS managed policies

View details about updates to AWS managed policies for Step Functions since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Step Functions Document history page.

Change	Description	Date
AWSStepFunctionsReadOnlyAccess – Update to an existing policy	Step Functions added new permissions to allow calling `states:ValidateStateMachineDefinition` API action to check the syntax of state machine definitions that you provide.	April 25, 2024
AWSStepFunctionsReadOnlyAccess – Update to an existing policy	Step Functions added new permissions to allow listing and reading data related to: Tags (ListTagsForResource), Distributed Map (ListMapRuns, DescribeMapRun), Versions and Aliases (DescribeStateMachineAlias, ListStateMachineAliases, ListStateMachineVersions).	April 02, 2024
Step Functions started tracking changes	Step Functions started tracking changes for its AWS managed policies.	April 02, 2024

Creating an IAM role for your state machine in Step Functions

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

The Tutorials for learning Step Functions 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 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 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

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). This type of impersonation can happen cross-account and cross-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.

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*.

We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that Step Functions gives another service to access your resources. Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

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 of the resource, or if you're specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcard characters (*) 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":{
           "ArnLike":{
              "aws:SourceArn":"arn:aws:states:us-east-1:111122223333:stateMachine:*"
           },
           "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 Integrating services with Step Functions.

> **Note**
>
> For examples of IAM policies created by the Step Functions console, see How Step Functions generates IAM policies for integrated services.

Creating granular permissions for non-admin users in Step Functions

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

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

State Machine-Level Permissions

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

[Execution-Level Permissions](#)

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

[Activity-Level Permissions](#)

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

Service-Level Permissions

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

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "states:ListStateMachines",
                "states:ListActivities",
                "states:CreateStateMachine",
                "states:CreateActivity",
                "states:ValidateStateMachineDefinition"
            ],
            "Resource": [
                "arn:aws:states:*:*:*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::123456789012:role/my-execution-role"
            ]
        }
    ]
}
```

State Machine-Level Permissions

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

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeStateMachine",
        "states:StartExecution",
        "states:DeleteStateMachine",
        "states:ListExecutions",
        "states:UpdateStateMachine",
        "states:TestState",
        "states:RevealSecrets"
      ],
      "Resource": [
        "arn:aws:states:*:*:stateMachine:StateMachinePrefix*"
      ]
    }
  ]
}
```

Execution-Level Permissions

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

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeExecution",
```

```
        "states:DescribeStateMachineForExecution",
        "states:GetExecutionHistory",
        "states:StopExecution"
      ],
      "Resource": [
        "arn:aws:states:*:*:execution:*:ExecutionPrefix*"
      ]
    }
  ]
}
```

Activity-Level Permissions

This permission level applies to all the API actions that act on a specific activity or on a particular instance of it. These API operations require the ARN of the activity or the token of the instance as part of the request, such as `DeleteActivity`, `DescribeActivity`, `GetActivityTask`, and `SendTaskHeartbeat`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeActivity",
        "states:DeleteActivity",
        "states:GetActivityTask",
        "states:SendTaskHeartbeat"
      ],
      "Resource": [
        "arn:aws:states:*:*:activity:ActivityPrefix*"
      ]
    }
  ]
}
```

Accessing resources in other AWS accounts in Step Functions

Step Functions provides cross-account access to resources configured in different AWS accounts in your workflows. Using Step Functions service integrations, you can invoke any cross-account AWS resource even if that AWS service does not support resource-based policies or cross-account calls.

For example, assume you own two AWS accounts, called Development and Testing, in the same AWS Region. Using cross-account access, your workflow in the Development account can access resources, such as Amazon S3 buckets, Amazon DynamoDB tables, and Lambda functions that are available in the Testing account.

> **Important**
>
> IAM roles and resource-based policies delegate access across accounts only within a single partition. For example, assume that you have an account in US West (N. California) in the standard `aws` partition. You also have an account in China (Beijing) in the `aws-cn` partition. You can't use an Amazon S3 resource-based policy in your account in China (Beijing) to allow access for users in your standard `aws` account.

For more information about cross-account access, see Cross-account policy evaluation logic in the *IAM User Guide*.

Although each AWS account maintains complete control over its own resources, with Step Functions, you can reorganize, swap, add, or remove steps in your workflows without the need to customize any code. You can do this even as the processes change or applications evolve.

You can also invoke executions of nested state machines so they're available across different accounts. Doing so efficiently separates and isolates your workflows. When you use the `.sync` service integration pattern in your workflows that access another Step Functions workflow in a different account, Step Functions uses polling that consumes your assigned quota. For more information, see Run a Job (.sync).

> **Note**
>
> Currently, cross-Region AWS SDK integration and cross-Region AWS resource access aren't available in Step Functions.

Key cross-account resource concepts

Execution role

An IAM role that Step Functions uses to run code and access AWS resources, such as the AWS Lambda function's Invoke action.

Service integration

The AWS SDK integration API actions that can be called from within a `Task` state in your workflows.

source account

An AWS account that owns the state machine and has started its execution.

target account

An AWS account to which you make cross-account calls.

target role

An IAM role in the target account that the state machine assumes for making calls to resources that the target account owns.

Run a Job (.sync)

A service integration pattern used to call services, such as AWS Batch. It also makes a Step Functions state machine wait for a job to complete before progressing to the next state. To indicate that Step Functions should wait, append the `.sync` suffix in the `Resource` field in your `Task` state definition.

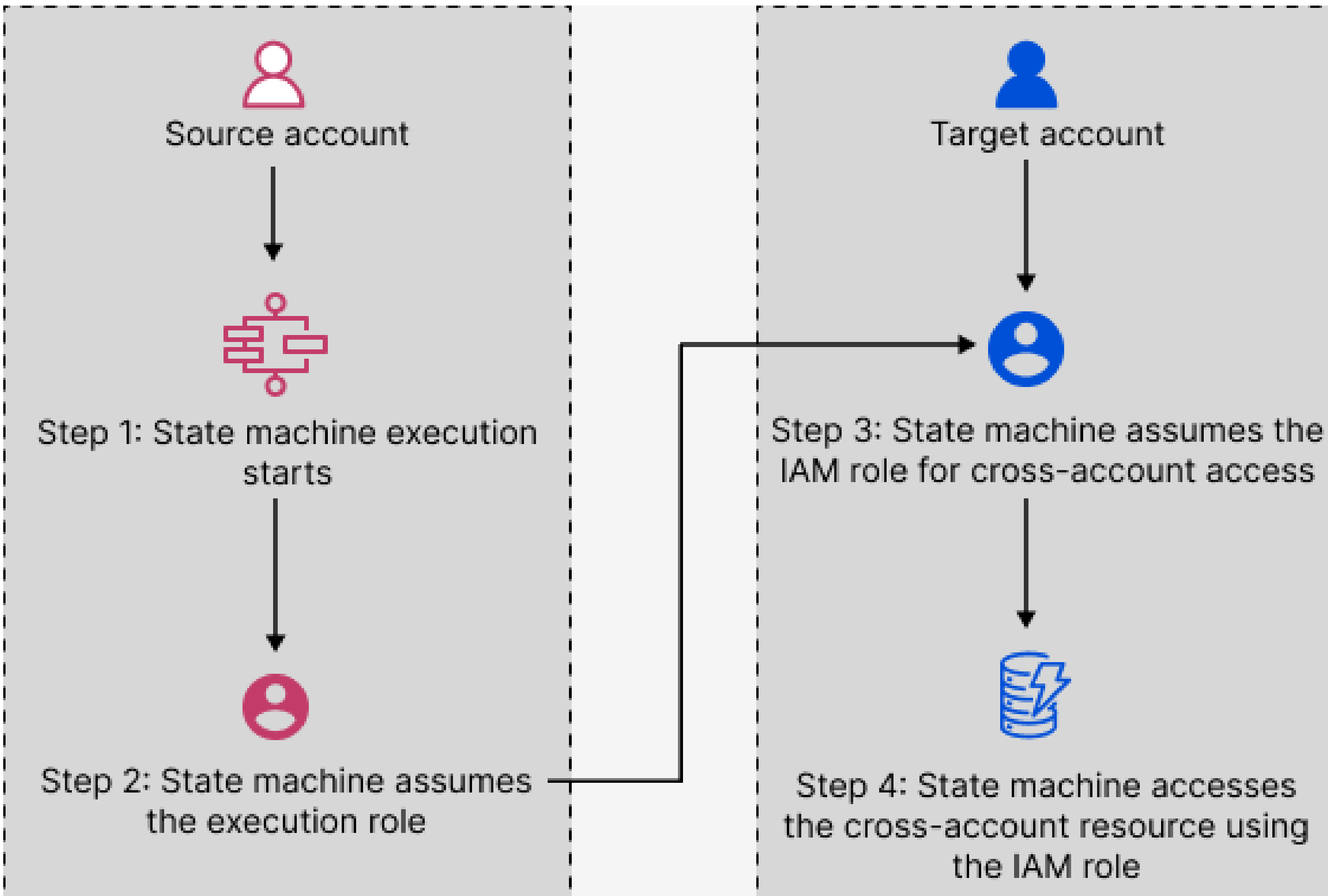
Invoking cross-account resources

To invoke a cross-account resource in your workflows, do the following:

1. Create an IAM role in the target account that contains the resource. This role grants the source account, containing the state machine, permissions to access the target account's resources.
2. In the `Task` state's definition, specify the target IAM role to be assumed by the state machine before invoking the cross-account resource.
3. Modify the trust policy in the target IAM role to allow the source account to assume this role temporarily. The trust policy must include the Amazon Resource Name (ARN) of the state machine defined in the source account. Also, define the appropriate permissions in the target IAM role to call the AWS resource.
4. Update the source account’s execution role to include the required permission for assuming the target IAM role.

For an example, see Accessing cross-account AWS resources in Step Functions in the tutorials.

> **Note**
>
> You can configure your state machine to assume an IAM role for accessing resources from multiple AWS accounts. However, a state machine can assume only one IAM role at a given time.



Cross-account access for .sync integration pattern

When you use the [.sync](#) service integration patterns in your workflows, Step Functions polls the invoked cross-account resource to confirm the task is complete. This causes a slight delay between the actual task completion time and the time when Step Functions recognizes the task as complete. The target IAM role needs the required permissions for a `.sync` invocation to complete this polling loop. To do this, the target IAM role must have a trust policy that allows the source account to assume it. Additionally, the target IAM role needs the required permissions to complete the polling loop.

> **Note**
>
> For nested Express Workflows, `arn:aws:states:::states:startExecution.sync` isn't currently supported. Use `arn:aws:states:::aws-sdk:sfn:startSyncExecution` instead.

Trust policy update for .sync calls

Update the trust policy of your target IAM role as shown in the following example. The `sts:ExternalId` field further controls who can assume the role. The state machine's name must include only characters that the AWS Security Token Service `AssumeRole` API supports. For more information, see AssumeRole in the *AWS Security Token Service API Reference*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Principal": {
        "AWS": "arn:aws:iam::sourceAccountID:role/InvokeRole",
      },
      "Condition": {
        "StringEquals": {
          "sts:ExternalId": "arn:aws:states:us-
east-2:sourceAccountID:stateMachine:stateMachineName"
        }
      }
    }
  ]
}
```

Permissions required for .sync calls

To grant the permissions required for your state machine, update the required permissions for the target IAM role. For more information, see the section called "IAM Policies for integrated services". The Amazon EventBridge permissions from the example policies aren't required. For example, to start a state machine, add the following permissions.

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": [
        "arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachineName"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeExecution",
        "states:StopExecution"
      ],
      "Resource": [
        "arn:aws:states:us-east-1:123456789012:execution:myStateMachineName:*"
      ]
    }
  ]
}
```

Creating 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):

```
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
- Creating granular permissions for non-admin users in Step Functions
- Controlling Access to Services with VPC Endpoints

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 user to create state machines, and denies all other users permission to delete state machines. The example policy also grants all users execution permission.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
              "states:ListExecutions", "states:StartExecution", "states:StopExecution",
 "states:DescribeExecution"
            ],
            "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 permissions for non-admin users in Step Functions
- Controlling Access to Services with VPC Endpoints

How Step Functions generates 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:

- For **optimized integrations**, Step Functions will create a policy with all the necessary permissions and roles for your state machine.

  Tip: You can see example policies in each of the service pages under Integrating optimized services.
- For **standard integrations** integrations, Step Functions will create an IAM role with partial permissions.

  You must add any missing role policies that your state machine needs to interact with the service.

Dynamic and static resources

Static resources are defined **directly** in the task state of your state machine. When you include the information about the resources you want to call directly in your task states, Step Functions can create an IAM role for only those resources.

Dynamic resources are **passed** as input when starting your state machine, or as input to an individual state, and accessed using JSONata or a JSONPath. When you are passing dynamic resources to your task, Step Functions cannot automatically scope-down the permissions, so Step Functions will create a more permissive policy which specifies:`"Resource": "*"`.

Additional permissions for tasks using .sync

Tasks that use the Run a Job (.sync) pattern require additional permissions for monitoring and receiving a response from the API of connected services.

Step Functions uses two approaches 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. For example, for Amazon ECS the state machine must have allow permission for `ecs:DescribeTasks`, for AWS Glue the state machine requires allow permissions for `glue:GetJobRun`. If the necessary permissions are missing from the role, Step Functions may be unable to determine the status of your job. One reason for using the polling method is because some service integrations do not support EventBridge events, and some services only send events on a best-effort basis.

Alternatively, you might use events sent from AWS services to Amazon EventBridge. Events are routed to Step Functions by EventBridge with a managed rule, so the role requires permissions for `events:PutTargets`, `events:PutRule`, and `events:DescribeRule`. If these permissions are missing from the 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.

Troubleshooting stuck .sync workflows

For Run a Job (.sync) tasks that support **both** polling and events, your task may complete properly using events, even when the role lacks the required permissions for polling.

In the previous scenario, you might not notice the polling permissions are missing or incorrect. In the rare case that an 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 in EventBridge that is responsible for forwarding events to Step Functions.

> **Note**
>
> Because managed rules are shared by all state machines in your account, you should use a test or development account to avoid unintentional impact to 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.

Permissions for cancelling workflows

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.

Cancelling a task 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.

> **Learn more about integration patterns**
>
> To learn about synchronous tasks, see Discover service integration patterns in Step Functions.

IAM policies for Activities-only Step Functions state machines

For a state machine that has only `Activity` tasks, or no tasks at all, use an IAM policy that denies access to all actions and resources.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": "*",
            "Resource": "*"
        }
    ]
}
```

For more information about using `Activity` tasks, see Learn about Activities in Step Functions.

IAM policies for using Distributed Map states

When you create workflows with the Step Functions console, Step Functions can automatically generate IAM policies based on the resources in your workflow definition. These policies include the least privileges necessary to allow the state machine role to invoke the `StartExecution` API action for the *Distributed Map state*. These policies also include the least privileges necessary Step Functions to access AWS resources, such as Amazon S3 buckets and objects and Lambda functions. We highly recommend that you include only those permissions that are necessary in your IAM policies. For example, if your workflow includes a `Map` state in Distributed mode, scope your policies down to the specific Amazon S3 bucket and folder that contains your dataset.

Important

If you specify an Amazon S3 bucket and object, or prefix, with a reference path to an existing key-value pair in your *Distributed Map state* input, make sure that you update the IAM policies for your workflow. Scope the policies down to the bucket and object names the path resolves to at runtime.

Example of IAM policy for running a Distributed Map state

When you include a *Distributed Map state* in your workflows, Step Functions needs appropriate permissions to allow the state machine role to invoke the `StartExecution` API action for the *Distributed Map state*.

The following IAM policy example grants the least privileges required to your state machine role for running the *Distributed Map state*.

> **Note**
>
> Make sure that you replace *stateMachineName* with the name of the state machine in which you're using the *Distributed Map state*. For example, `arn:aws:states:region:account-id:stateMachine:mystateMachine`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:StartExecution"
      ],
      "Resource": [
        "arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachineName"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "states:DescribeExecution"
      ],
      "Resource": "arn:aws:states:us-
east-1:123456789012:execution:myStateMachineName:*"
    }
  ]
}
```

Example of IAM policy for redriving a Distributed Map

You can restart unsuccessful child workflow executions in a Map Run by redriving your parent workflow. A redriven parent workflow redrives all the unsuccessful states, including Distributed Map. Make sure that your execution role has the least privileges necessary to allow it to invoke the `RedriveExecution` API action on the parent workflow.

The following IAM policy example grants the least privileges required to your state machine role for redriving a *Distributed Map state*.

> **Note**
>
> Make sure that you replace *stateMachineName* with the name of the state machine in which you're using the *Distributed Map state*. For example, `arn:aws:states:region:account-id:stateMachine:mystateMachine`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:RedriveExecution"
      ],
      "Resource": "arn:aws:states:us-
east-2:123456789012:execution:myStateMachineName/myMapRunLabel:*"
    }
  ]
}
```

Examples of IAM policies for reading data from Amazon S3 datasets

The following IAM policy examples grant the least privileges required to access your Amazon S3 datasets using the ListObjectsV2 and GetObject API actions.

Example IAM policy for Amazon S3 object as dataset

The following example shows an IAM policy that grants the least privileges to access the objects organized within *processImages* in an Amazon S3 bucket named *amzn-s3-demo-bucket*.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket"
```

```
            ],
            "Condition": {
                "StringLike": {
                    "s3:prefix": [
                        "processImages"
                    ]
                }
            }
        }
    ]
}
```

Example IAM policy for a CSV file as dataset

The following example shows an IAM policy that grants least privileges to access a CSV file named *ratings.csv*.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket/csvDataset/ratings.csv"
            ]
        }
    ]
}
```

Example IAM policy for an Amazon S3 inventory as dataset

The following example shows an IAM policy that grants least privileges to access an Amazon S3 inventory report.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::myDestination-prefix/amzn-s3-demo-bucket/myConfig-
id/YYYY-MM-DDTHH-MMZ/manifest.json",
                "arn:aws:s3:::myDestination-prefix/amzn-s3-demo-bucket/myConfig-id/
data/*"
            ]
        }
    ]
}
```

Example of IAM policy for writing data to an Amazon S3 bucket

The following IAM policy example grants the least privileges required to write your child workflow execution results to a folder named *csvJobs* in an Amazon S3 bucket using the `PutObject` API action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:ListMultipartUploadParts",
                "s3:AbortMultipartUpload"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-destination-bucket/csvJobs/*"
            ]
        }
    ]
}
```

IAM permissions for AWS KMS key encrypted Amazon S3 bucket

Distributed Map state uses multipart uploads to write the child workflow execution results to an Amazon S3 bucket. If the bucket is encrypted using an AWS Key Management Service (AWS KMS) key, you must also include permissions in your IAM policy to perform the `kms:Decrypt`, `kms:Encrypt`, and `kms:GenerateDataKey` actions on the key. These permissions are required because Amazon S3 must decrypt and read data from the encrypted file parts before it completes the multipart upload.

The following IAM policy example grants permission to the `kms:Decrypt`, `kms:Encrypt`, and `kms:GenerateDataKey` actions on the key used to encrypt your Amazon S3 bucket.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": [
      "kms:Decrypt",
      "kms:Encrypt",
      "kms:GenerateDataKey"
    ],
    "Resource": [
      "arn:aws:kms:us-east-1:123456789012:key/111aa2bb-333c-4d44-5555-a111bb2c33dd"
    ]
  }
}
```

For more information, see [Uploading a large file to Amazon S3 with encryption using an AWS KMS key]() in the *AWS Knowledge Center*.

If your IAM user or role is in the same AWS account as the KMS key, then you must have these permissions on the key policy. If your IAM user or role belongs to a different account than the KMS key, then you must have the permissions on both the key policy and your IAM user or role.

Creating tag-based IAM policies in Step Functions

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`.

```
{
    "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 tag-based authorization, state machine execution resources as shown in the following example inherit the tags associated with a state machine.

```
arn:partition:states:region:account-id:execution:<StateMachineName>:<ExecutionId>
```

When you call DescribeExecution or other APIs in which you specify the execution resource ARN, Step Functions uses tags associated with the state machine to accept or deny the request while performing tag-based authorization. This helps you allow or deny access to state machine executions at the state machine level.

For more information about tagging, see the following:

- Tagging state machines and activities in Step Functions
- Controlling Access Using IAM Tags

Troubleshooting identity and access issues in Step Functions

Use the following information to help you diagnose and fix common issues that you might encounter when working with Step Functions and IAM.

Topics

- I am not authorized to perform an action in Step Functions
- I am not authorized to perform iam:PassRole

- I want to allow people outside of my AWS account to access my Step Functions resources

I am not authorized to perform an action in Step Functions

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the mateojackson user tries to use the console to view details about a fictional *my-example-widget* resource but does not have the fictional states:*GetWidget* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 states:GetWidget on resource: my-example-widget
```

In this case, Mateo's policy must be updated to allow him to access the *my-example-widget* resource using the states:*GetWidget* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the iam:PassRole action, your policies must be updated to allow you to pass a role to Step Functions.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named marymajor tries to use the console to perform an action in Step Functions. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the iam:PassRole action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Step Functions resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Step Functions supports these features, see How AWS Step Functions works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Compliance validation for 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 on Amazon Web Services – 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 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 Step Functions

As a managed service, AWS Step Functions is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well‐Architected Framework*.

You use AWS published API calls to access Step Functions through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). 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 the AWS API operations from any network location, but Step Functions doesn't 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.

Logging and monitoring AWS Step Functions service performance

Learn how to log and monitor Step Functions to maintain the reliability, availability, and performance of Step Functions and your AWS solutions.

There are several tools available to use with Step Functions:

Topics

Tip

To deploy a sample workflow and learn how to monitor metrics, logs, and traces of the workflow execution, see Adding observability in *The AWS Step Functions Workshop*.

Monitoring Step Functions metrics using Amazon 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 debug 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 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.

We recommend monitoring Activity and Task failures to establish a baseline. When performance falls outside your baseline metric, set an alert so you can research the root cause.

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

- `ActivitiesStarted`
- `ActivitiesTimedOut`
- `ExecutionsStarted`
- `ExecutionsTimedOut`

Step Functions metrics for CloudWatch

Step Functions provides the following types of metrics to Amazon CloudWatch. You can use these metrics to track your state machines and activities and to set alarms on threshold values. You can view metrics using the AWS Management Console.

CloudWatch metrics delivery

CloudWatch metrics are delivered on a best-effort basis.

The completeness and timeliness of metrics are not guaranteed. The data point for a particular request might be returned with a timestamp that is later than when the request was actually processed. The data point for a minute might be delayed before being available through CloudWatch, or it might not be delivered at all. CloudWatch request metrics give you an idea of the state machine executions in near-real time. It is not meant to be a complete accounting of all execution-related metrics.

It follows from the best-effort nature of this feature that the reports available at the Billing & Cost Management Dashboard might include one or more access requests that do not appear in the execution metrics.

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.

Step Functions emits two `ExecutionsStarted` metrics for every state machine execution. This causes the SampleCount statistic for the `ExecutionsStarted` metric to show the value of *2* for every state machine execution. The SampleCount statistic shows `ExecutionStarted=1` and `ExecutionStarted=0` when the execution completes.

> **Tip**
>
> We recommend selecting **Sum** as the display statistic for metrics that report a count in the CloudWatch console.

Execution metrics

The `AWS/States` namespace includes the following metrics for all Step Functions executions. These are dimensionless metrics that apply across your account in a region.

Metric	Description
`OpenExecutionCount`	Approximate number of currently *open executions*—workflows that are currently in progress in your account. The intent is to provide insight into when your workflows are approaching the **maximum** execution limit, to avoid **Execution LimitExceeded** errors when calling `StartExecution` or `RedriveExecution` for Standard Workflows. `OpenExecutionCount` is an approximate number of open workflows. This metric will be lower than observed running workflow count. Running open workflow count lower than 10,000 may show zero open executions. For an alarm to notify if you are nearing your `OpenExecutionLimit`, we recommend using the Maximum statistic with a threshold of 100K or higher since the default open workflow limit is 1,000,000 executions.
`OpenExecutionLimit`	Maximum number of open executions. For more information, see Quotas related to accounts. *This limit does not apply to Express Workflows.*

Execution metrics for all state machines

All state machines emit metrics. The `ExecutionThrottled` metric will only be emitted in the case of throttled execution. These metrics will include a `StateMachineArn` to identify a specific state machine.

Metric	Description
`ExecutionTime`	Interval, in milliseconds, between the time the execution starts and the time it closes.

Metric	Description
`ExecutionThrottled`	Number of `StateEntered` events and retries that have been throttled. This is related to `StateTransition` throttling. For more information, see Quotas related to state throttling.
`ExecutionsAborted`	Number of aborted or terminated executions.
`ExecutionsFailed`	Number of failed executions.
`ExecutionsStarted`	Number of started executions.
`ExecutionsSucceeded`	Number of successfully completed executions.
`ExecutionsTimedOut`	Number of executions that time out for any reason.

Execution metrics for Express Workflows

The `AWS/States` namespace includes the following metrics for Step Functions Express Workflows' executions.

Metric	Description
`ExpressExecutionMemory`	The total memory consumed by an Express Workflow.
`ExpressExecutionBilledDuration`	The duration for which an Express Workflow is charged.
`ExpressExecutionBilledMemory`	The amount of consumed memory for which an Express Workflow is charged.

Redrive execution metrics for Standard Workflows

When you redrive a state machine execution, Step Functions emits the following metrics.

For all redriven executions, the `Executions*` metric is emitted. For example, say a redriven execution aborts. This execution will emit non-zero datapoints for both `RedrivenExecutionsAborted` and `ExecutionsAborted`.

Metric	Description
ExecutionsRedriven	Number of redriven executions.
RedrivenExecutionsAborted	Number of redriven executions that are canceled or terminated.
RedrivenExecutionsTimedOut	Number of redriven executions that time out for any reason.
RedrivenExecutionsSucceeded	Number of redriven executions that completed successfully.
RedrivenExecutionsFailed	Number of redriven executions that failed.

Dimension for Step Functions execution metrics

Dimension	Description
StateMachineArn	The Amazon Resource Name (ARN) of the state machine for the execution in question.

Dimensions for executions with version

Dimension	Description
StateMachineArn	The Amazon Resource Name (ARN) of the state machine whose execution was started by a version.
Version	State machine version used to start the execution.

Dimensions for executions with an alias

Dimension	Description
`StateMachineArn`	The Amazon Resource Name (ARN) of the state machine whose execution was started by an alias.
`Alias`	State machine alias used to start the execution.

Resource count metrics for versions and aliases

The `AWS/States` namespace includes the following metrics for the count of versions and aliases of a state machine.

Metric	Description
AliasCount	Number of aliases created for the state machine. You can create up to 100 aliases for each state machine.
VersionCount	Number of versions published for the state machine. You can publish up to 1000 versions of a state machine.

Dimension for resource count metrics for versions and aliases

Dimension	Description
`ResourceArn`	The Amazon Resource Name (ARN) of the state machine with a version or an alias.

Activity Metrics

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

Metric	Description
ActivityRunTime	Interval, in milliseconds, between the time the activity starts and the time it closes.
ActivityScheduleTime	Interval, in milliseconds, for which the activity stays in the schedule state.
ActivityTime	Interval, in milliseconds, between the time the activity is scheduled and the time it closes.
ActivitiesFailed	Number of failed activities.
ActivitiesHeartbeatTimedOut	Number of activities that time out due to a heartbeat timeout.
ActivitiesScheduled	Number of scheduled activities.
ActivitiesStarted	Number of started activities.
ActivitiesSucceeded	Number of successfully completed activities.
ActivitiesTimedOut	Number of activities that time out on close.

Dimension for Step Functions Activity Metrics

Dimension	Description
ActivityArn	The ARN of the activity.

Lambda Function Metrics

The `AWS/States` namespace includes the following metrics for Lambda functions which are referred to **directly** in the Resource field of a Task state definition. You might find these metrics in legacy state machines. In modern state machines, we recommend using the Optimized Lambda integration which emits Service Integration Metrics.

Metric	Description
`LambdaFunctionRunTime`	Interval, in milliseconds, between the time the Lambda function starts and the time it closes.
`LambdaFunctionScheduleTime`	Interval, in milliseconds, for which the Lambda function stays in the schedule state.
`LambdaFunctionTime`	Interval, in milliseconds, between the time the Lambda function is scheduled and the time it closes.
`LambdaFunctionsFailed`	Number of failed Lambda functions.
`LambdaFunctionsScheduled`	Number of scheduled Lambda functions.
`LambdaFunctionsStarted`	Number of started Lambda functions.
`LambdaFunctionsSucceeded`	Number of successfully completed Lambda functions.
`LambdaFunctionsTimedOut`	Number of Lambda functions that time out on close.

Dimension for Step Functions Lambda Function Metrics

Dimension	Description
`LambdaFunctionArn`	The ARN of the Lambda function.

> **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 an AWS Lambda function with Step Functions.

Service Integration Metrics

The AWS/States namespace includes the following metrics for Step Functions service integrations. For more information, see Integrating services with Step Functions.

Metric	Description
ServiceIntegrationRunTime	Interval, in milliseconds, between the time the Service Task starts and the time it closes.
ServiceIntegrationScheduleTime	Interval, in milliseconds, for which the Service Task stays in the schedule state.
ServiceIntegrationTime	Interval, in milliseconds, between the time the Service Task is scheduled and the time it closes.
ServiceIntegrationsFailed	Number of failed Service Tasks.
ServiceIntegrationsScheduled	Number of scheduled Service Tasks.
ServiceIntegrationsStarted	Number of started Service Tasks.
ServiceIntegrationsSucceeded	Number of successfully completed Service Tasks.
ServiceIntegrationsTimedOut	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`	Count of requests that have been throttled.
`ProvisionedBucketSize`	Count of available requests per second.
`ProvisionedRefillRate`	Count of requests per second that are allowed into the bucket.
`ConsumedCapacity`	Count of requests per second.

Dimension for Step Functions Service Metrics

Dimension	Description
`ServiceMetric`	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`	Count of requests that have been throttled.

Metric	Description
`ProvisionedBucketSize`	Count of available requests per second.
`ProvisionedRefillRate`	Count of requests per second that are allowed into the bucket.
`ConsumedCapacity`	Count of requests per second.

Dimension for Step Functions API Metrics

Dimension	Description
`APIName`	Filters data to an API of the specified API name.

Viewing Step Functions metrics in CloudWatch

You can use the CloudWatch console to view Step Functions metrics for executions, activities, functions, and service integrations.

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 to display the metric details.

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

Setting alarms for Step Functions metrics in CloudWatch

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 the bell-shaped icon next to a metric on the list to display the **Create Alarm** page.
6. Enter the values for the **Alarm threshold** and **Actions**, and then choose **Create Alarm**.

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

Automating Step Functions event delivery with EventBridge

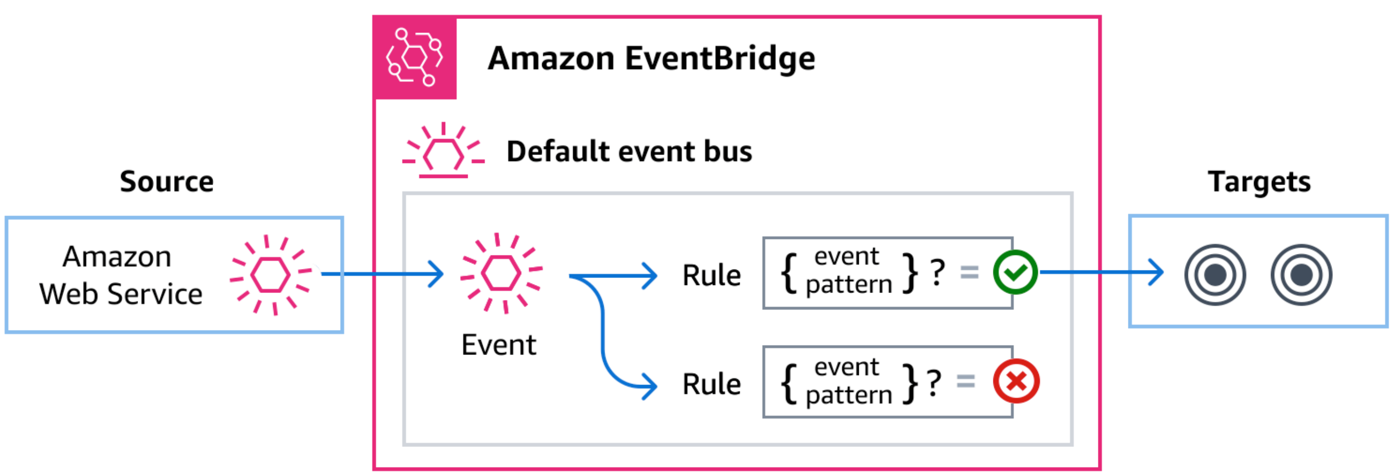
With EventBridge, you can select events from Step Functions standard workflows, to send to other services for additional processing. This technique provides a flexible way to loosely connect components and monitor your resources.

Amazon EventBridge is a serverless service that connects application components together to build scalable event-driven applications. Event-driven architecture is a style of building loosely-coupled software systems that work together by emitting and responding to *events*. Events represent a change in state, or an update.

By using EventBridge to deliver Step Functions events to other services, you can monitor your standard workflows without continuously calling the DescribeExecution API to get the status. Status changes in state machine executions are sent to EventBridge automatically. You can use those events to target services. For example, events might invoke a AWS Lambda function, publish a message to Amazon Simple Notification Service (Amazon SNS) topic, or even run another SFN workflow.

How event delivery works

Step Functions generates and sends events to the default EventBridge *event bus* which is automatically provisioned in every AWS account. An event bus is a router that receives events and delivers them to zero or more destinations, or *targets*. Targets are other AWS services. You can specify rules for the event bus that compare events against the rule's *event pattern*. When the event matches a pattern, the event bus sends the event to the specified target(s). The following diagram shows this process:



> **Standard versus Express workflows**
>
> Only standard workflows emit events to EventBridge. To monitor the execution of express workflows, you can use CloudWatch Logs. See Logging in CloudWatch Logs.

Step Functions events

Step Functions sends the following events to the default EventBridge event bus automatically. Events that match a rule's event pattern are delivered to the specified targets on a best-effort basis. Events might be delivered out of order.

For more information, see EventBridge events in the *Amazon EventBridge User Guide.*

Event detail type	Description
Execution Status Change	Represents a change in the status of a state machine execution.

Delivering Step Functions events using EventBridge

To have the EventBridge default event bus send Step Functions events to a target, you must create a rule. Each rule contains an event pattern, which EventBridge matches against each event received on the event bus. If the event data matches the specified event pattern, EventBridge delivers that event to the rule's target(s).

For comprehensive instructions on creating event bus rules, see Creating rules that react to events in the *EventBridge User Guide*.

You can also create an event bus rule for a specific state machine from the Step Functions console:

- On the **Details** page of a state machine, choose **Actions**, and then choose **Create *EventBridge* rule**.

  The EventBridge console opens to the **Create rule** page, with the state machine selected as the event source for the rule.
- Follow the procedure detailed in Creating rules that react to events in the *EventBridge User Guide*.

Creating event patterns that match Step Functions events

Each event pattern is a JSON object that contains:

- A `source` attribute that identifies the service sending the event. For Step Functions events, the source is `aws.states`.
- (Optional): A `detail-type` attribute that contains an array of the event types to match.
- (Optional): A `detail` attribute containing any other event data on which to match.

For example, the following event pattern matches against all Execution Status Change events from Step Functions:

```
{
  "source": ["aws.states"],
  "detail-type": ["Step Functions Execution Status Change"]
}
```

While the following example matches against a specific execution associated with a specific state machine, when that execution fails or times out:

```
{
  "source": ["aws.states"],
  "detail-type": ["Step Functions Execution Status Change"],
  "detail": {
    "status": ["FAILED", "TIMED_OUT"],
    "stateMachineArn": ["arn:aws:states:region:account-id:stateMachine:state-machine"],
    "executionArn": ["arn:aws:states:region:account-id:execution:state-machine-
name:execution-name"]
  }
}
```

For more information on writing event patterns, see Event patterns in the *EventBridge User Guide*.

Triggering Step Functions state machines using events

You can also specify a Step Functions state machine as a target for EventBridge event bus rule. 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 Amazon EventBridge targets in the *Amazon EventBridge User Guide*.

Step Functions events detail reference

All events from AWS services have a common set of fields containing metadata about the event, such as the AWS service that is the source of the event, the time the event was generated, the account and region in which the event took place, and others. For definitions of these general fields, see Event structure reference in the *Amazon EventBridge User Guide*.

In addition, each event has a `detail` field that contains data specific to that particular event.

When using EventBridge to select and manage Step Functions events, it's useful to keep the following in mind:

- The `source` field for all events from Step Functions is set to `aws.states`.
- The `detail-type` field specifies the event type.

  For example, `Step Functions Execution Status Change`.
- The `detail` field contains the data that is specific to that particular event.

For information on constructing event patterns that enable rules to match Step Functions events, see Event patterns in the *Amazon EventBridge User Guide*.

For more information on events and how EventBridge processes them, see Amazon EventBridge events in the *Amazon EventBridge User Guide*.

Execution Status Change

Represents a change in the status of a state machine execution.

The `source` and `detail-type` fields are included below because they contain specific values for Step Functions events. For definitions of the other metadata fields that are included in all events, see Event structure reference in the *Amazon EventBridge User Guide*.

Event structure

```
{
  . . .,
  "detail-type": "Step Functions Execution Status Change",
  "source"": "aws.states",
  . . .,
  "detail"": {
    "executionArn"" : "string",
```

```
    "input" : "string",
    "inputDetails" : {
      "included" : "boolean"
    },
    "name" : "string",
    "output" : "string",
    "outputDetails" : {
      "included" : "boolean"
    },
    "startDate" : "integer",
    "stateMachineArn" : "string",
    "stopDate" : "integer",
    "status" : "RUNNING | SUCCEEDED | FAILED | TIMED_OUT | ABORTED | PENDING_REDRIVE"
  }
}
```

Remarks

An Execution Status Change event can contain an input property in its definition. For some events, an Execution Status Change event can also contain an output property in its definition.

- If the combined escaped input and escaped output sent to EventBridge exceeds 248 KiB, then the input will be excluded. Similarly, if the escaped output exceeds 248 KiB, then the output will be excluded. This is a result of 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, use [DescribeExecution] to see the full input and output.

  `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.
  - Use Amazon S3 ARNs. For information about using ARNs, see [the section called “Using Amazon S3 to pass large data”].

Examples

Example Execution Status Change: execution started

```
{
```

```
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "account-id",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-2",
    "resources": [
      "arn:aws:states:us-east-2:account-id:execution:state-machine-name:execution-name"
    ],
    "detail": {
        "executionArn": "arn:aws:states:us-east-2:account-id:execution:state-machine-
name:execution-name",
        "stateMachineArn": "arn:aws::states:us-east-2:account-id:stateMachine:state-
machine",
        "name": "execution-name",
        "status": "RUNNING",
        "startDate": 1551225271984,
        "stopDate":  null,
        "input": "{}",
        "inputDetails": {
             "included": true
        },
        "output": null,
        "outputDetails": null
    }
}
```

Example Execution Status Change: execution succeeded

```
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "account-id",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-2",
    "resources": [
      "arn:aws:states:us-east-2:account-id:execution:state-machine-name:execution-name"
    ],
    "detail": {
```

```
        "executionArn": "arn:aws:states:us-east-2:account-id:execution:state-machine-
name:execution-name",
        "stateMachineArn": "arn:aws:states:us-east-2:account-id:stateMachine:state-
machine",
        "name": "execution-name",
        "status": "SUCCEEDED",
        "startDate": 1547148840101,
        "stopDate": 1547148840122,
        "input": "{}",
        "inputDetails": {
             "included": true
        },
        "output": "\"Hello World!\"",
        "outputDetails": {
             "included": true
        }
    }
}
```

Example Execution Status Change: execution failed

```
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "account-id",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-2",
    "resources": [
      "arn:aws:states:us-east-2:account-id:execution:state-machine-name:execution-name"
    ],
    "detail": {
        "executionArn": "arn:aws:states:us-east-2:account-id:execution:state-machine-
name:execution-name",
        "stateMachineArn": "arn:aws:states:us-east-2:account-id:stateMachine:state-
machine",
        "name": "execution-name",
        "status": "FAILED",
        "startDate": 1551225146847,
        "stopDate": 1551225151881,
        "input": "{}",
        "inputDetails": {
```

```
            "included": true
        },
        "output": null,
        "outputDetails": null
    }
}
```

Example Execution Status Change: timed-out

```
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "account-id",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-2",
    "resources": [
      "arn:aws:states:us-east-2:account-id:execution:state-machine-name:execution-name"
    ],
    "detail": {
        "executionArn": "arn:aws:states:us-east-2:account-id:execution:state-machine-
name:execution-name",
        "stateMachineArn": "arn:aws:states:us-east-2:account-id:stateMachine:state-
machine",
        "name": "execution-name",
        "status": "TIMED_OUT",
        "startDate": 1551224926156,
        "stopDate": 1551224927157,
        "input": "{}",
        "inputDetails": {
            "included": true
        },
        "output": null,
        "outputDetails": null
```

Example Execution Status Change: aborted

```
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
```

```
    "source": "aws.states",
    "account": "account-id",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-2",
    "resources": [
      "arn:aws:states:us-east-2:account-id:execution:state-machine-name:execution-name"
    ],
    "detail": {
        "executionArn": "arn:aws:states:us-east-2:account-id:execution:state-machine-
name:execution-name",
        "stateMachineArn": "arn:aws:states:us-east-2:account-id:stateMachine:state-
machine",
        "name": "execution-name",
        "status": "ABORTED",
        "startDate": 1551225014968,
        "stopDate": 1551225017576,
        "input": "{}",
        "inputDetails": {
             "included": true
        },
        "output": null,
        "outputDetails": null
    }
}
```

Recording Step Functions API calls with AWS CloudTrail

AWS Step Functions is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures all API calls for Step Functions as events. The calls captured include calls from the Step Functions console and code calls to the Step Functions API operations. 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, when it was made, and additional details.

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 user or user credentials.
- Whether the request was made on behalf of an IAM Identity Center user.
- Whether the request was made with temporary security credentials for a role or federated user.

- Whether the request was made by another AWS service.

CloudTrail is active in your AWS account when you create the account and you automatically have access to the CloudTrail **Event history**. The CloudTrail **Event history** provides a viewable, searchable, downloadable, and immutable record of the past 90 days of recorded management events in an AWS Region. For more information, see Working with CloudTrail Event history in the *AWS CloudTrail User Guide*. There are no CloudTrail charges for viewing the **Event history**.

For an ongoing record of events in your AWS account past 90 days, create a trail or a CloudTrail Lake event data store.

CloudTrail trails

A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. All trails created using the AWS Management Console are multi-Region. You can create a single-Region or a multi-Region trail by using the AWS CLI. Creating a multi-Region trail is recommended because you capture activity in all AWS Regions in your account. If you create a single-Region trail, you can view only the events logged in the trail's AWS Region. For more information about trails, see Creating a trail for your AWS account and Creating a trail for an organization in the *AWS CloudTrail User Guide*.

You can deliver one copy of your ongoing management events to your Amazon S3 bucket at no charge from CloudTrail by creating a trail, however, there are Amazon S3 storage charges. For more information about CloudTrail pricing, see AWS CloudTrail Pricing. For information about Amazon S3 pricing, see Amazon S3 Pricing.

CloudTrail Lake event data stores

CloudTrail Lake lets you run SQL-based queries on your events. CloudTrail Lake converts existing events in row-based JSON format to Apache ORC format. ORC is a columnar storage format that is optimized for fast retrieval of data. Events are aggregated into *event data stores*, which are immutable collections of events based on criteria that you select by applying advanced event selectors. The selectors that you apply to an event data store control which events persist and are available for you to query. For more information about CloudTrail Lake, see Working with AWS CloudTrail Lake in the *AWS CloudTrail User Guide*.

CloudTrail Lake event data stores and queries incur costs. When you create an event data store, you choose the pricing option you want to use for the event data store. The pricing option determines the cost for ingesting and storing events, and the default and maximum retention period for the event data store. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

Data events in CloudTrail

Data events provide information about the resource operations performed on or in a resource (for example, reading or writing to an Amazon S3 object). These are also known as data plane operations. Data events are often high-volume activities. By default, CloudTrail doesn’t log data events. The CloudTrail **Event history** doesn't record data events.

Additional charges apply for data events. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

You can log data events for the Step Functions resource types by using the CloudTrail console, AWS CLI, or CloudTrail API operations. For more information about how to log data events, see Logging data events with the AWS Management Console and Logging data events with the AWS Command Line Interface in the *AWS CloudTrail User Guide*.

The following table lists the Step Functions resource types for which you can log data events. The **Data event type** column shows the value to choose from the **Data event type** list on the CloudTrail console. The **resources.type value** column shows the `resources.type` value, which you would specify when configuring advanced event selectors using the AWS CLI or CloudTrail APIs. The **Data APIs logged to CloudTrail** column shows the API calls logged to CloudTrail for the resource type.

You can configure advanced event selectors to filter on the `eventName`, `readOnly`, and `resources.ARN` fields to log only those events that are important to you. For more information about these fields, see AdvancedFieldSelector in the *AWS CloudTrail API Reference*.

Data event type	resources.type value	Data APIs logged to CloudTrail
Step Functions state machine	`AWS::StepFunctions::StateMachine`	• InvokeHTTPEndpoint • StartSyncExecution
Step Functions activity	`AWS::StepFunctions::Activity`	• GetActivityTask

Management events in CloudTrail

Management events provide information about management operations that are performed on resources in your AWS account. These are also known as control plane operations. By default, CloudTrail logs management events.

State Machine

- CreateStateMachine
- ListStateMachines
- DescribeStateMachine
- UpdateStateMachine
- DeleteStateMachine
- ValidateStateMachineDefinition
- TestState

State Machine Alias

- CreateStateMachineAlias
- ListStateMachineAliases
- DescribeStateMachineAlias
- UpdateStateMachineAlias
- DeleteStateMachineAlias

State Machine Version

- ListStateMachineVersions
- PublishStateMachineVersion
- DeleteStateMachineVersion

Executions

- StartExecution
- StartSyncExecution

- RedriveExecution
- ListExecutions
- DescribeExecution
- GetExecutionHistory
- DescribeStateMachineForExecution
- StopExecution

Activity

- CreateActivity
- ListActivities
- DescribeActivity
- DeleteActivity
- GetActivityTask

Task Token

- SendTaskSuccess
- SendTaskHeartbeat
- SendTaskFailure

MapRun

- ListMapRuns
- DescribeMapRun
- UpdateMapRun

Tags

- ListTagsForResource
- TagResource
- UntagResource

Event examples

An event represents a single request from any source and includes information about the requested API operation, the date and time of the operation, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so events don't appear in any specific order.

The following example shows a CloudTrail **data event** that demonstrates `InvokeHTTPEndpoint`.

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "accountId": "account-id",
        "invokedBy": "states.amazonaws.com"
    },
    "eventTime": "2024-05-01T01:23:45Z",
    "eventSource": "states.amazonaws.com",
    "eventName": "InvokeHTTPEndpoint",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "states.amazonaws.com",
    "userAgent": "states.amazonaws.com",
    "requestParameters": null,
    "responseElements": null,
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "readOnly": false,
    "resources": [
        {
            "accountId": "account-id",
            "type": "AWS::StepFunctions::StateMachine",
            "ARN": "arn:aws:states:region:account-id:stateMachine:ExampleStateMachine"
        }
    ],
    "eventType": "AwsServiceEvent",
    "managementEvent": false,
    "recipientAccountId": "account-id",
    "serviceEventDetails": {
        "httpMethod": "GET",
        "httpEndpoint": "https://example.com"
    },
    "eventCategory": "Data"
}
```

The following example shows a CloudTrail **management event** that demonstrates the `CreateStateMachine` operation.


```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDAJYDLDBVBI4EXAMPLE",
        "arn": "arn:aws:iam::account-id:user/test-user",
        "accountId": "account-id",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "test-user"
    },
    "eventTime": "2024-05-01T01:23:45Z",
    "eventSource": "states.amazonaws.com",
    "eventName": "CreateStateMachine",
    "awsRegion": "region",
    "sourceIPAddress": "AWS Internal",
    "userAgent": "AWS Internal",
    "requestParameters": {
        "name": "MyStateMachine",
        "definition": "HIDDEN_DUE_TO_SECURITY_REASONS",
        "roleArn": "arn:aws:iam::account-id:role/MyStateMachineRole",
        "type": "STANDARD",
        "loggingConfiguration": {
            "level": "OFF",
            "includeExecutionData": false
        },
        "tags": [],
        "tracingConfiguration": {
            "enabled": false
        },
        "publish": false
    },
    "responseElements": {
        "stateMachineArn": "arn:aws:states:region:account-
id:stateMachine:MyStateMachine",
        "creationDate": "May 1, 2024 1:23:45 AM"
    },
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "readOnly": false,
    "eventType": "AwsApiCall",
```

```
    "managementEvent": true,
    "recipientAccountId": "account-id",
    "eventCategory": "Management"
}
```

For information about CloudTrail record contents, see CloudTrail record contents in the *AWS CloudTrail User Guide*.

Using CloudWatch Logs to log execution history in Step Functions

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.

> **Log delivery guarantees**
>
> Amazon CloudWatch Logs are delivered on a best-effort basis. The completeness and timeliness of log entries are not guaranteed. If you require guaranteed workflow history in Express Workflows, we recommend that you implement workflow steps to record data in an appropriate data storage service such as Amazon DynamoDB. Alternatively, you might consider using **Standard Workflows** for guaranteed execution history.

> **Pricing information**
>
> When you configure logging, CloudWatch Logs charges will apply and you will be billed at the vended logs rate. For more information, see **Vended Logs** under the **Logs** tab on the CloudWatch Pricing page.

Configure logging

When you create a Standard Workflow using the Step Functions console, that state machine will **not** be configured to send logs to CloudWatch Logs. When you create an Express Workflow using the Step Functions console, that state machine will by default be configured to send logs 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.

You can optionally configure customer managed AWS KMS keys to encrypt your logs. See Data at rest encryption for details and permission settings.

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 248 KiB, 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 "Using Amazon S3 to pass large data".

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. As shown in the following example, you need to specify * in the `Resource` field. CloudWatch API actions, such as CreateLogDelivery and DescribeLogGroups, do not support Resource types defined by Amazon CloudWatch Logs. For more information, see Actions defined by Amazon CloudWatch Logs.

- For information about CloudWatch resources, see CloudWatch Logs resources and operations in the *Amazon CloudWatch User Guide*.
- For information about the permissions you need to set up sending logs to CloudWatch Logs, see User permissions in the section titled *Logs sent to CloudWatch Logs*.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogDelivery",
                "logs:CreateLogStream",
                "logs:GetLogDelivery",
                "logs:UpdateLogDelivery",
                "logs:DeleteLogDelivery",
                "logs:ListLogDeliveries",
                "logs:PutLogEvents",
                "logs:PutResourcePolicy",
                "logs:DescribeResourcePolicies",
                "logs:DescribeLogGroups"
            ],
            "Resource": "*"
        }
    ]
}
```

Troubleshooting state machine logging to CloudWatch Logs

If your state machine cannot send logs to CloudWatch Logs, try the following steps:

1. Verify your state machine's execution role has permission to log to CloudWatch Logs.

When you call [CreateStateMachine] or [UpdateStateMachine] API endpoints, make sure the IAM role specified in the `roleArn` parameter provides the necessary permissions, shown in the preceding IAM policy example.

2. Verify the CloudWatch Logs resource policy does not exceed the 5,120 character limit.

   If the policy exceeds the character limit, prefix your log group names with `/aws/vendedlogs/states` to grant permissions to your state machines and avoid the limit. When you create a log group in the Step Functions console, the suggested log group names are already prefixed with `/aws/vendedlogs/states`. For more information on logging best practices, see CloudWatch Logs resource policy size limits.

Log levels for Step Functions execution events

Log levels range from `ALL` to `ERROR` to `FATAL` to `OFF`. All event types are logged for `ALL`, no event types are logged when set to `OFF`. For `ERROR` and `FATAL`, see the following table.

For more information about the execution data displayed for Express Workflow executions based on these **Log levels**, see Standard and Express console experience differences.

Event Type	ALL	ERROR	FATAL	OFF
ChoiceStateEntered	Logged	*Not logged*	*Not logged*	*Not logged*
ChoiceStateExited	Logged	*Not logged*	*Not logged*	*Not logged*
ExecutionAborted	Logged	Logged	Logged	*Not logged*
ExecutionFailed	Logged	Logged	Logged	*Not logged*
ExecutionStarted	Logged	*Not logged*	*Not logged*	*Not logged*
ExecutionSucceeded	Logged	*Not logged*	*Not logged*	*Not logged*

Event Type	ALL	ERROR	FATAL	OFF
ExecutionTimedOut	Logged	Logged	Logged	*Not logged*
FailStateEntered	Logged	Logged	*Not logged*	*Not logged*
LambdaFunctionFailed	Logged	Logged	*Not logged*	*Not logged*
LambdaFunctionScheduled	Logged	*Not logged*	*Not logged*	*Not logged*
LambdaFunctionScheduleFailed	Logged	Logged	*Not logged*	*Not logged*
LambdaFunctionStarted	Logged	*Not logged*	*Not logged*	*Not logged*
LambdaFunctionStartFailed	Logged	Logged	*Not logged*	*Not logged*
LambdaFunctionSucceeded	Logged	*Not logged*	*Not logged*	*Not logged*
LambdaFunctionTimedOut	Logged	Logged	*Not logged*	*Not logged*
MapIterationAborted	Logged	Logged	*Not logged*	*Not logged*
MapIterationFailed	Logged	Logged	*Not logged*	*Not logged*
MapIterationStarted	Logged	*Not logged*	*Not logged*	*Not logged*
MapIterationSucceeded	Logged	*Not logged*	*Not logged*	*Not logged*

Event Type	ALL	ERROR	FATAL	OFF
MapRunAborted	Logged	Logged	*Not logged*	*Not logged*
MapRunFailed	Logged	Logged	*Not logged*	*Not logged*
MapStateAborted	Logged	Logged	*Not logged*	*Not logged*
MapStateEntered	Logged	*Not logged*	*Not logged*	*Not logged*
MapStateExited	Logged	*Not logged*	*Not logged*	*Not logged*
MapStateFailed	Logged	Logged	*Not logged*	*Not logged*
MapStateStarted	Logged	*Not logged*	*Not logged*	*Not logged*
MapStateSucceeded	Logged	*Not logged*	*Not logged*	*Not logged*
ParallelStateAborted	Logged	Logged	*Not logged*	*Not logged*
ParallelStateEntered	Logged	*Not logged*	*Not logged*	*Not logged*
ParallelStateExited	Logged	*Not logged*	*Not logged*	*Not logged*
ParallelStateFailed	Logged	Logged	*Not logged*	*Not logged*
ParallelStateStarted	Logged	*Not logged*	*Not logged*	*Not logged*
ParallelStateSucceeded	Logged	*Not logged*	*Not logged*	*Not logged*

Event Type	ALL	ERROR	FATAL	OFF
PassStateEntered	Logged	*Not logged*	*Not logged*	*Not logged*
PassStateExited	Logged	*Not logged*	*Not logged*	*Not logged*
SucceedStateEntered	Logged	*Not logged*	*Not logged*	*Not logged*
SucceedStateExited	Logged	*Not logged*	*Not logged*	*Not logged*
TaskFailed	Logged	Logged	*Not logged*	*Not logged*
TaskScheduled	Logged	*Not logged*	*Not logged*	*Not logged*
TaskStarted	Logged	*Not logged*	*Not logged*	*Not logged*
TaskStartFailed	Logged	Logged	*Not logged*	*Not logged*
TaskStateAborted	Logged	Logged	*Not logged*	*Not logged*
TaskStateEntered	Logged	*Not logged*	*Not logged*	*Not logged*
TaskStateExited	Logged	*Not logged*	*Not logged*	*Not logged*
TaskSubmitFailed	Logged	Logged	*Not logged*	*Not logged*
TaskSubmitted	Logged	*Not logged*	*Not logged*	*Not logged*
TaskSucceeded	Logged	*Not logged*	*Not logged*	*Not logged*
TaskTimedOut	Logged	Logged	*Not logged*	*Not logged*
WaitStateAborted	Logged	Logged	*Not logged*	*Not logged*

Event Type	ALL	ERROR	FATAL	OFF
WaitState Entered	Logged	*Not logged*	*Not logged*	*Not logged*
WaitStateExited	Logged	*Not logged*	*Not logged*	*Not logged*

Trace Step Functions request data in AWS X-Ray

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 100 KiB for Step Functions state machines. If more than 100 KiB 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.

Important

Step Functions doesn't support X-Ray tracing for the child workflow executions started by a *Distributed Map state* because it's easy to exceed the Trace document size limit for such executions.

Topics

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.

Choose **Next**.

3. On the **Specify details** page, configure your state machine.
4. Choose **Enable X-Ray tracing**.

   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. For more information about the permissions that you need, see the following topic.

IAM policies using AWS X-Ray in Step Functions

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.

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.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingRules",
                "xray:GetSamplingTargets"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
```

```
}
```

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 IAM policies using AWS X-Ray in Step Functions.

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 AWS 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.

Integrate upstream services

To integrate the execution of Step Functions workflows, such as Express, Synchronous, and Standard workflows, with an upstream service you need to set the `traceHeader`. This is automatically done for you if you are using a HTTP API in API Gateway. However, if you're using a Lambda function and/or an SDK, you need to set the `traceHeader` on the `StartExecution` or `StartSyncExecution` API calls yourself.

You must specify the `traceHeader` format as `\p{ASCII}#`. Additionally, to let Step Functions use the same trace ID, you must specify the format as `Root={TRACE_ID};Sampled={1 or 0}`. If you're using a Lambda function, replace the `TRACE_ID` with the trace ID in your current segment and set the Sampled field as `1` if your sampling mode is true and `0` if your sampling mode is false. Providing the trace ID in this format ensures that you'll get a complete trace.

The following is an example written in Python to showcase how to specify the `traceHeader`.

```
state_machine = config.get_string_paramter("STATE_MACHINE_ARN")
    if (xray_recorder.current_subsegment() is not None and
        xray_recorder.current_subsegment().sampled) :
        trace_id = "Root={};Sampled=1".format(
            xray_recorder.current_subsegment().trace_id
```

```
        )
    else:
        trace_id = "Root=not enabled;Sampled=0"
    LOGGER.info("trace %s", trace_id)

    # execute it
    response = states.start_sync_execution(
        stateMachineArn=state_machine,
        input=event['body'],
        name=context.aws_request_id,
        traceHeader=trace_id
    )
    LOGGER.info(response)
```

X-Ray trace in header or payload

For X-Ray traces, all AWS services use the `X-Amzn-Trace-Id` header from the HTTP request. Using the header is the preferred mechanism to identify a trace. `StartExecution` and `StartSyncExecution` API operations can also use `traceHeader` from the body of the request payload. If **both** sources are provided, Step Functions will use the **header value** (preferred) over the value in the request body.

Concepts

The X-Ray console

In the AWS X-Ray console, you can view service maps and traces for requests that your applications serve when X-Ray is enabled for your state machine.

See [Viewing the X-Ray console](#) 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. You can analyze how your state machine is performing to 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 AI
- 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.

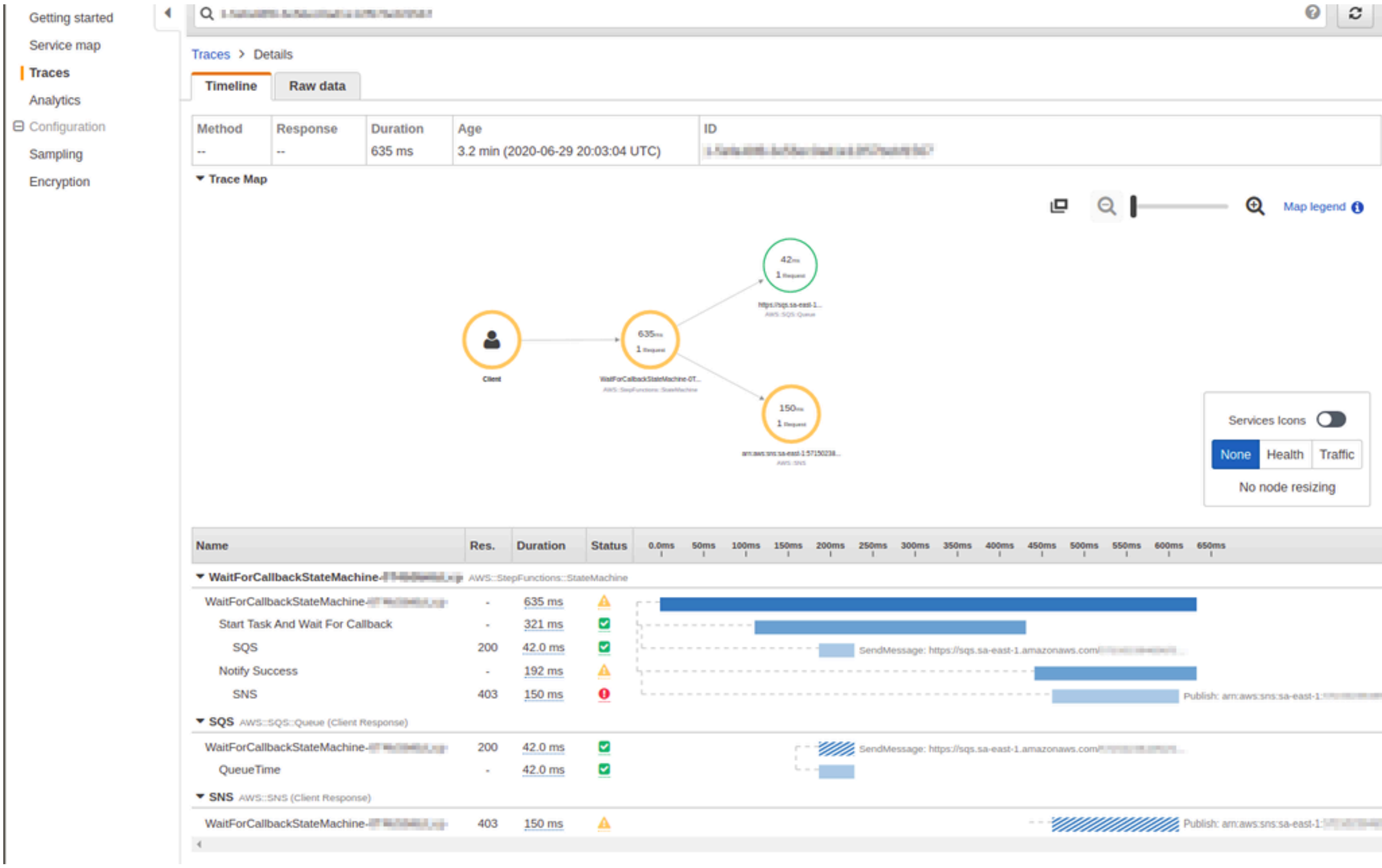
After you have enabled X-Ray for your state machine, you can view tracing information for its executions in the X-Ray console.

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 Create a callback pattern example with Amazon SQS, Amazon SNS, and Lambda sample project are shown.

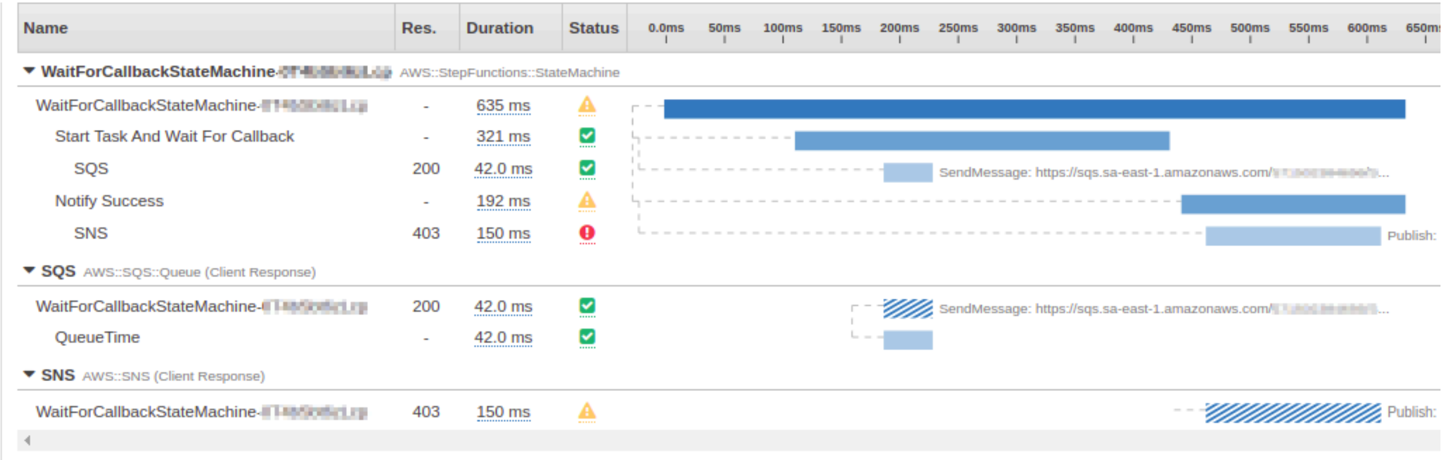
Traces

After 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.

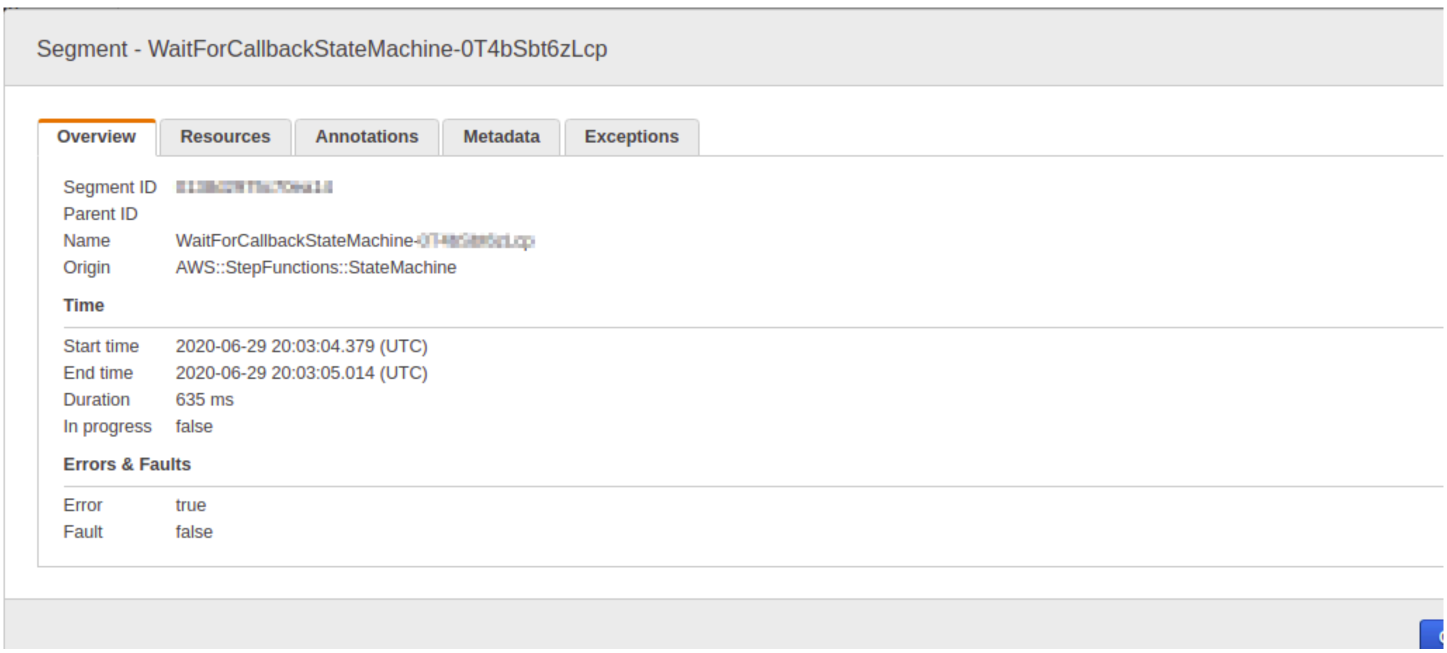


Service map

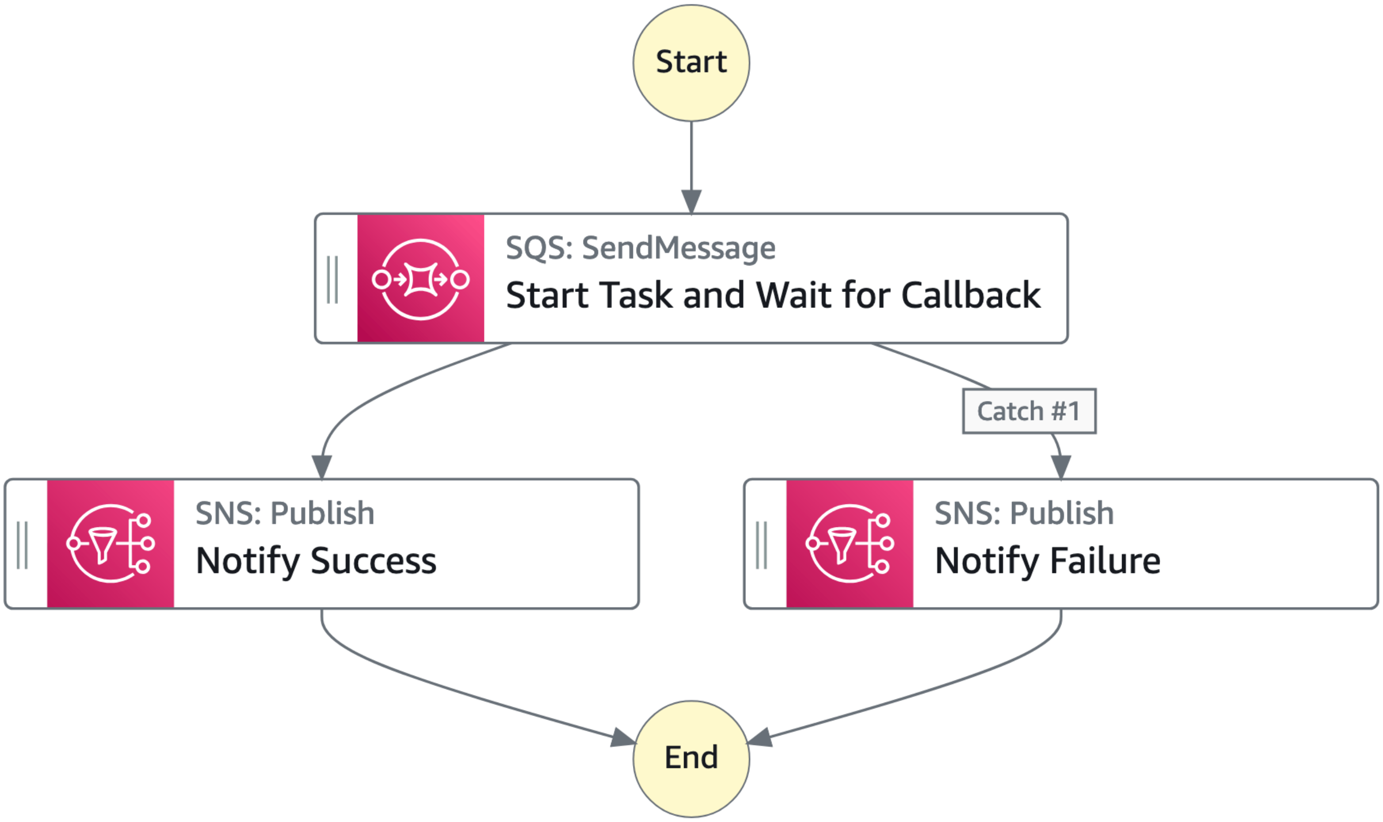
The service map in the X-Ray console helps you to identify services where errors are occurring, where there are connections with high latency, or see traces for requests that were unsuccessful.



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.



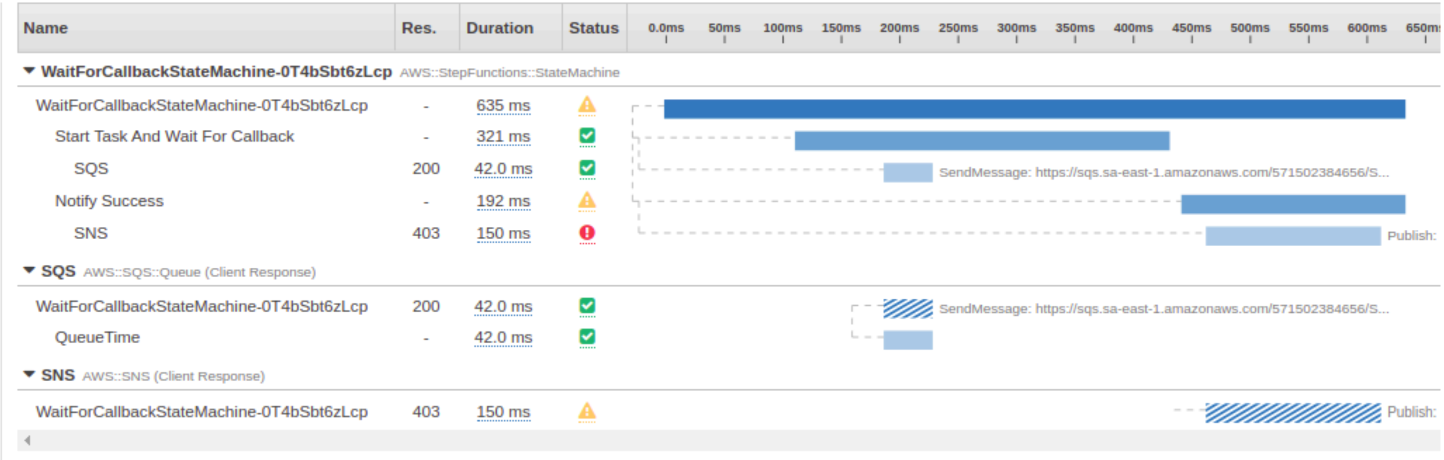
You can see how the X-Ray service map correlates to the state machine. There is a service map node for each service integration that is called by Step Functions, provided it supports X-Ray.



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.

You will be 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 shows 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. You can explore, analyze, and visualize traces through interactive response time and time-series graphs to help 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.

- 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.
- 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 KMS 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 Activating and deactivating AWS STS in an AWS Region in the *IAM User Guide*.

Setting up Step Functions event notification using AWS User Notifications

You can use AWS User Notifications to set up delivery channels to get notified about AWS Step Functions events. You receive a notification when an event matches a rule that you specify. You can receive notifications for events through multiple channels, including email, Amazon Q Developer in chat applications chat notifications, or AWS Console Mobile Application push notifications. You can also see notifications in the Console Notifications Center. User Notifications supports aggregation, which can reduce the number of notifications you receive during specific events.

Testing and debugging Step Functions state machines

Step Functions provides the following ways to test and debug state machines:

Test with Test State in console and API

In the Step Functions console, you can test an individual state with **Test State**. You provide the state definition and inputs in the console, then Step Functions runs the state and shows the outputs, all without creating a state machine.

Or, you can use the TestState API to test an individual state. You provide the definition of a single state, and the API will execute the state and report results, also without creating an actual state machine.

See Testing with TestState through the TestState API to test your states.

Data flow simulator (unsupported)

Data flow simulator is a console tool that was built to test JSONPath syntax. The data flow simulator is **unsupported**.

See Testing with TestState through the TestState API to test your states.

Step Functions Local (unsupported)

With AWS Step Functions Local, a downloadable version of Step Functions, you can test applications with Step Functions running in your own development environment.

Step Functions Local does **not** provide feature parity. For example, there is no support for optimized service integrations, cross-account access, or distributed map.

Step Functions Local is unsupported

Step Functions Local does **not** provide feature parity and is **unsupported**.
You might consider third party solutions that emulate Step Functions for testing purposes.

Using TestState API to test a state in Step Functions

The TestState API accepts the definition of a single state and executes it. You can test a state without creating a state machine or updating an existing state machine.

Using the TestState API, you can test the following:

- A state's input and output processing data flow.
- An AWS service integration with other AWS services request and response
- An HTTP Task request and response

To test a state, you can also use the Step Functions console, AWS Command Line Interface (AWS CLI), or the SDK.

The `TestState` API assumes an IAM role which must contain the required IAM permissions for the resources your state accesses. For information about the permissions a state might need, see IAM permissions for using TestState API.

Topics

Considerations about using the TestState API

Using the TestState API, you can test only one state at a time. The states that you can test include the following:

- All Task types, except Activities
- Pass workflow state
- Wait workflow state
- Choice workflow state

- Succeed workflow state
- Fail workflow state

While using the `TestState` API, keep in mind the following considerations.

- The TestState API doesn't include support for the following:
  - Task workflow state states that use the following resource types:
    - Activity
    - Service integration patterns of type `.sync` or `.waitForTaskToken`
  - Parallel workflow state state
  - Map workflow state state
- A test can run for up to five minutes. If a test exceeds this duration, it fails with the `States.Timeout` error.

Using inspection levels in TestState API

To test a state using the TestState API, you provide the definition of that state. The test then returns an output. For each state, you can specify the amount of detail you want to view in the test results. These details provide additional information about the state you're testing. For example, if you've used any input and output data processing filters, such as `InputPath` or `ResultPath` in a state, you can view the intermediate and final data processing results.

Step Functions provides the following levels to specify the details you want to view:

- INFO
- DEBUG
- TRACE

All these levels also return the `status` and `nextState` fields. `status` indicates the status of the state execution. For example, `SUCCEEDED`, `FAILED`, `RETRIABLE`, and `CAUGHT_ERROR`. `nextState` indicates the name of the next state to transition to. If you haven't defined a next state in your definition, this field returns an empty value.

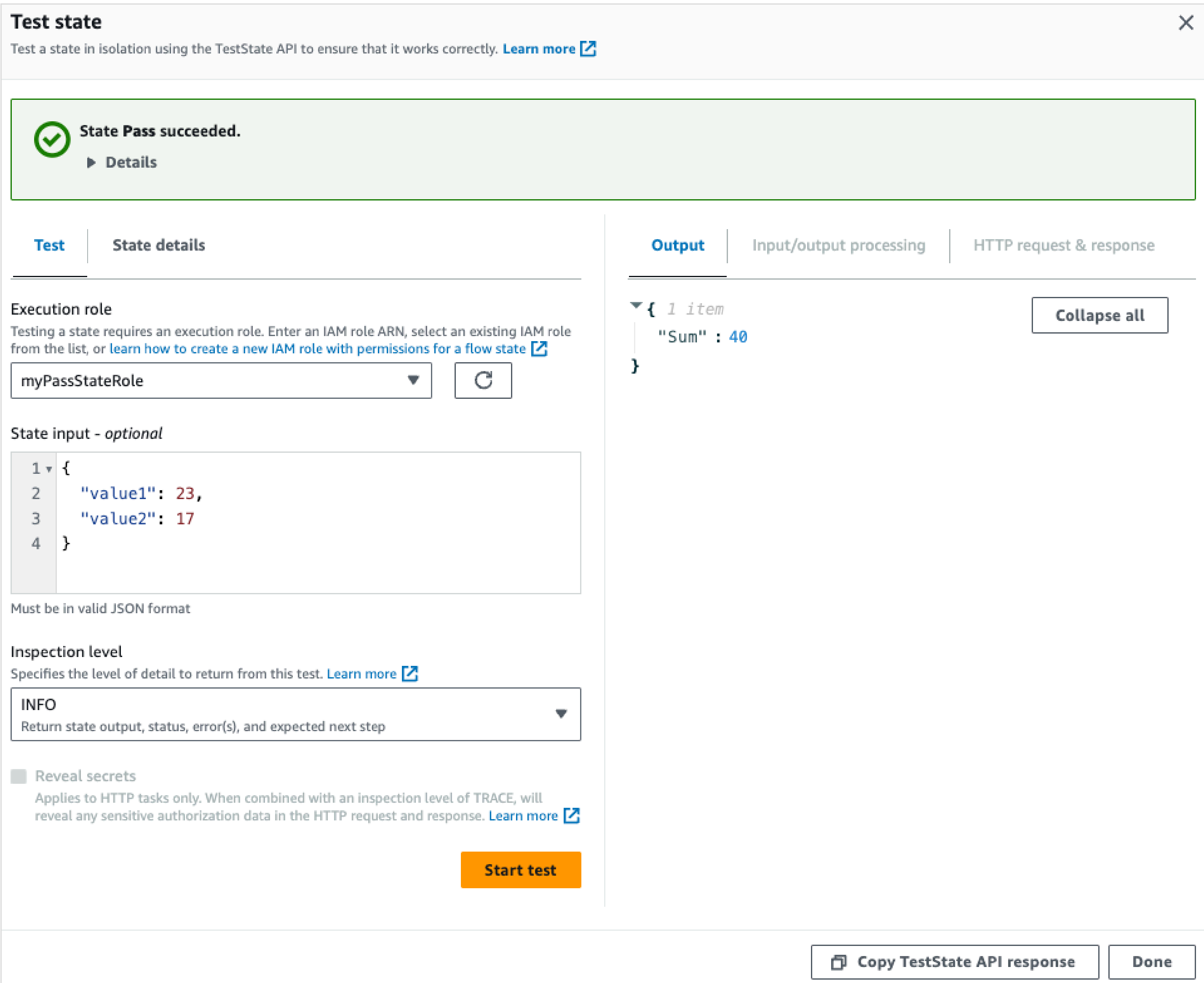
For information about testing a state using these inspection levels in the Step Functions console and AWS CLI, see Testing a state (Console) and Testing a state using AWS CLI.

INFO inspectionLevel

If the test succeeds, this level shows the state output. If the test fails, this level shows the error output. By default, Step Functions sets **Inspection level** to **INFO** if you don't specify a level.

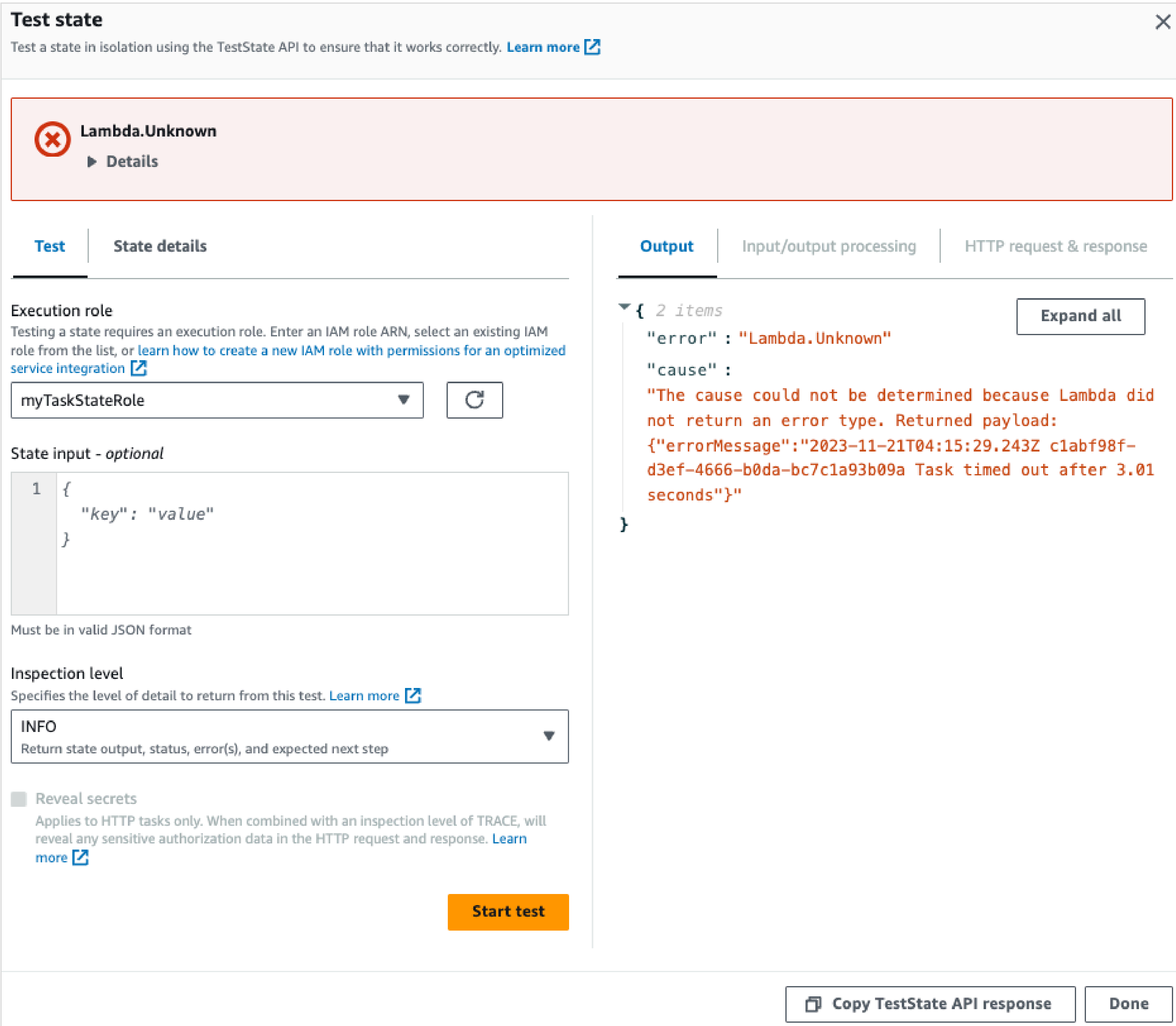
Example of test with INFO level that succeeds

The following image shows a test for a Pass state that succeeds. The **Inspection level** for this state is set to **INFO** and the output for the state appears in the **Output** tab.



Example of test with INFO level that fails

The following image shows a test that failed for a Task state when the **Inspection level** is set to **INFO**. The **Output** tab shows the error output that includes the error name and a detailed explanation of the cause for that error.



DEBUG inspectionLevel

If the test succeeds, this level shows the state output and the result of input and output data processing.

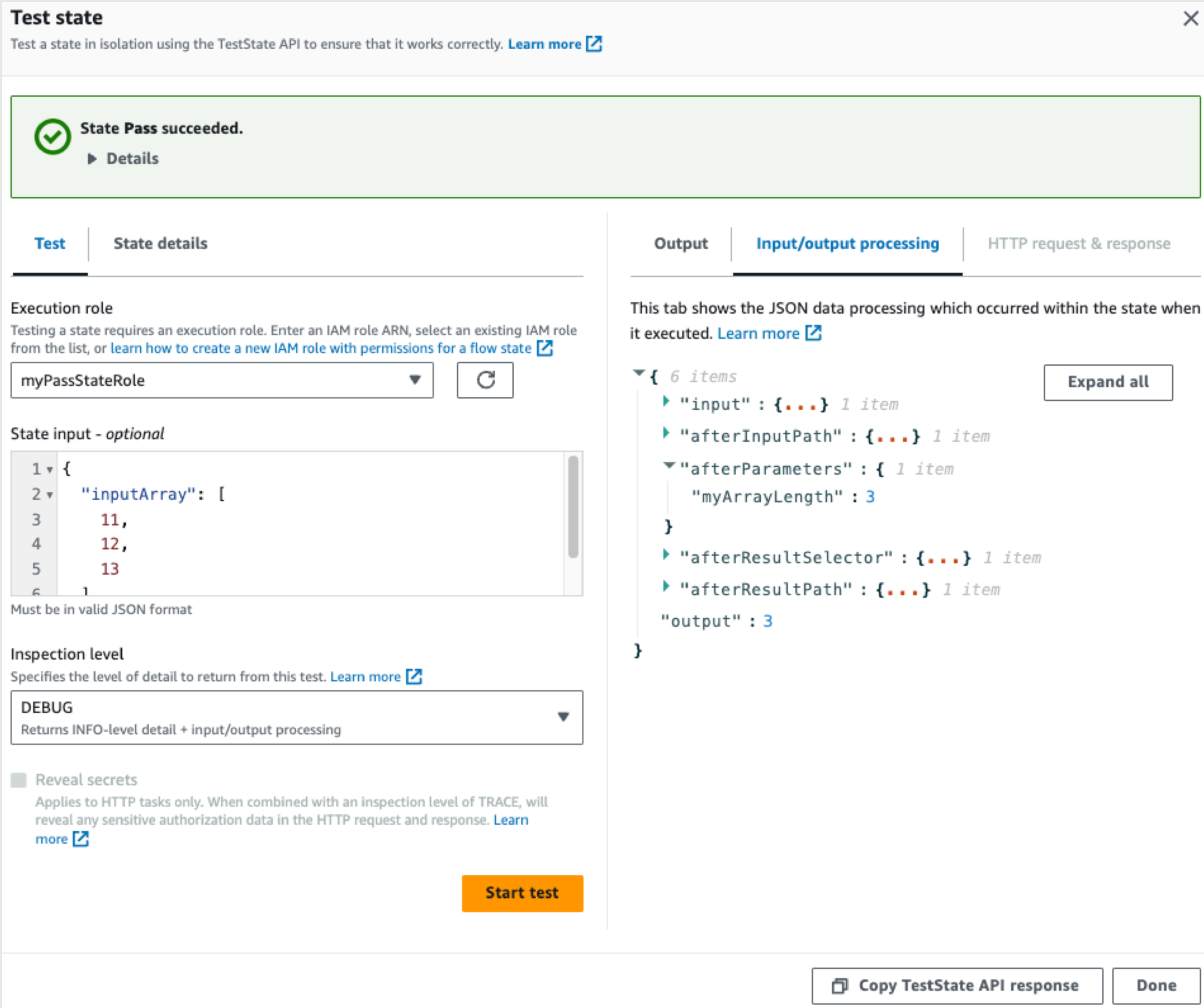
If the test fails, this level shows the error output. This level shows the intermediate data processing results up to the point of failure. For example, say that you tested a Task state that invokes a Lambda function. Imagine that you had applied the InputPath, Parameters, Specifying state output using ResultPath in Step Functions, and Filtering state output using OutputPath filters to the Task state. Say that the invocation failed. In this case, the DEBUG level shows data processing results based on the application of the filters in the following order:

- `input` – Raw state input
- `afterInputPath` – Input after Step Functions applies the `InputPath` filter.
- `afterParameters` – The effective input after Step Functions applies the `Parameters` filter.

The diagnostic information available in this level can help you troubleshoot issues related to a service integration or input and output data processing flow that you might have defined.

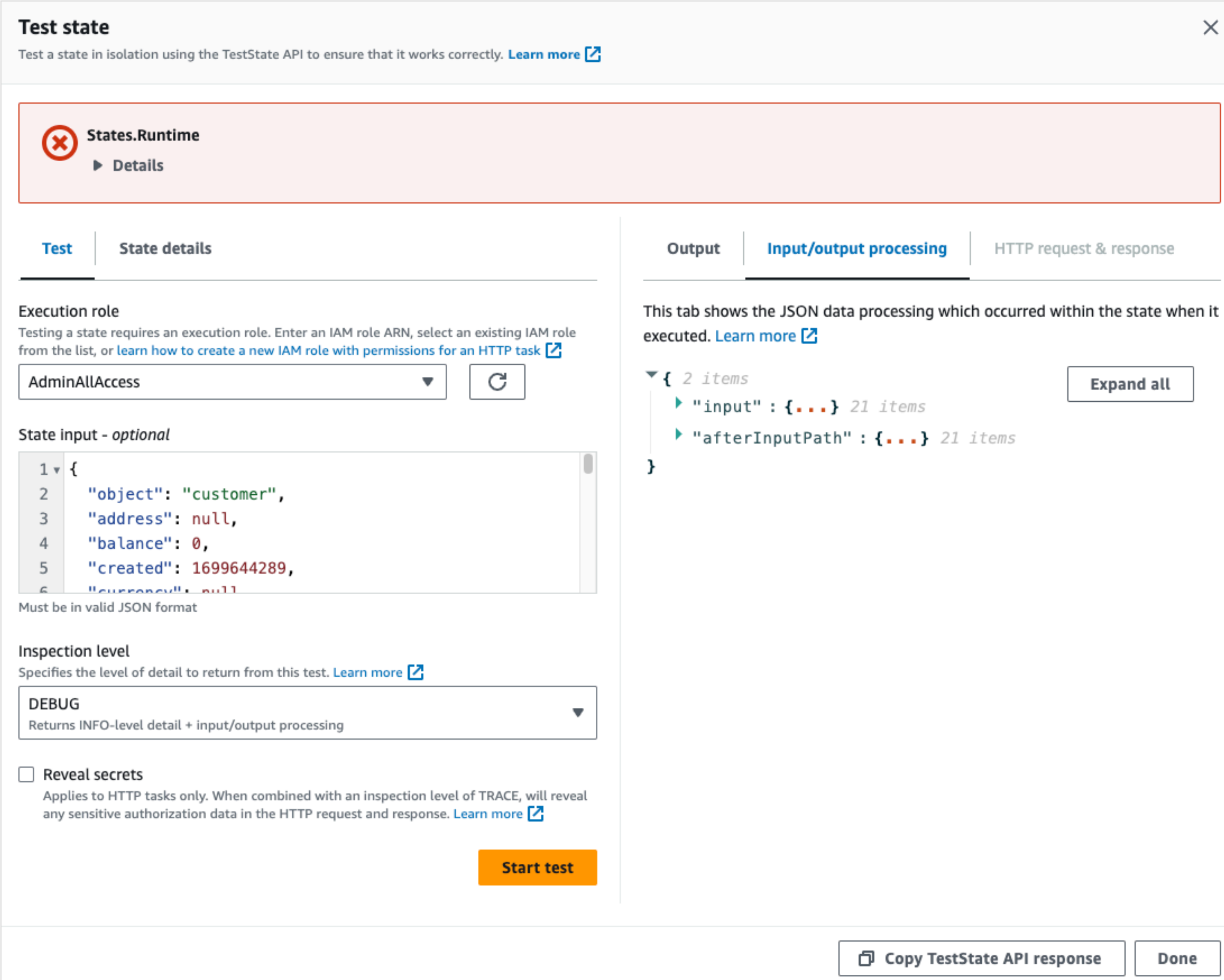
Example of test with DEBUG level that succeeds

The following image shows a test for a Pass state that succeeds. The **Inspection level** for this state is set to **DEBUG**. The **Input/output processing** tab in the following image shows the result of the application of `Parameters` on the input provided for this state.



Example of test with DEBUG level that fails

The following image shows a test that failed for a Task state when the **Inspection level** is set to **DEBUG**. The **Input/output processing** tab in the following image shows the input and output data processing result for the state up to the point of its failure.



TRACE inspectionLevel

Step Functions provides the **TRACE** level to test an HTTP Task. This level returns information about the HTTP request that Step Functions makes and response that a HTTPS API returns. The response might contain information, such as headers and request body. In addition, you can view the state output and result of input and output data processing in this level.

If the test fails, this level shows the error output.

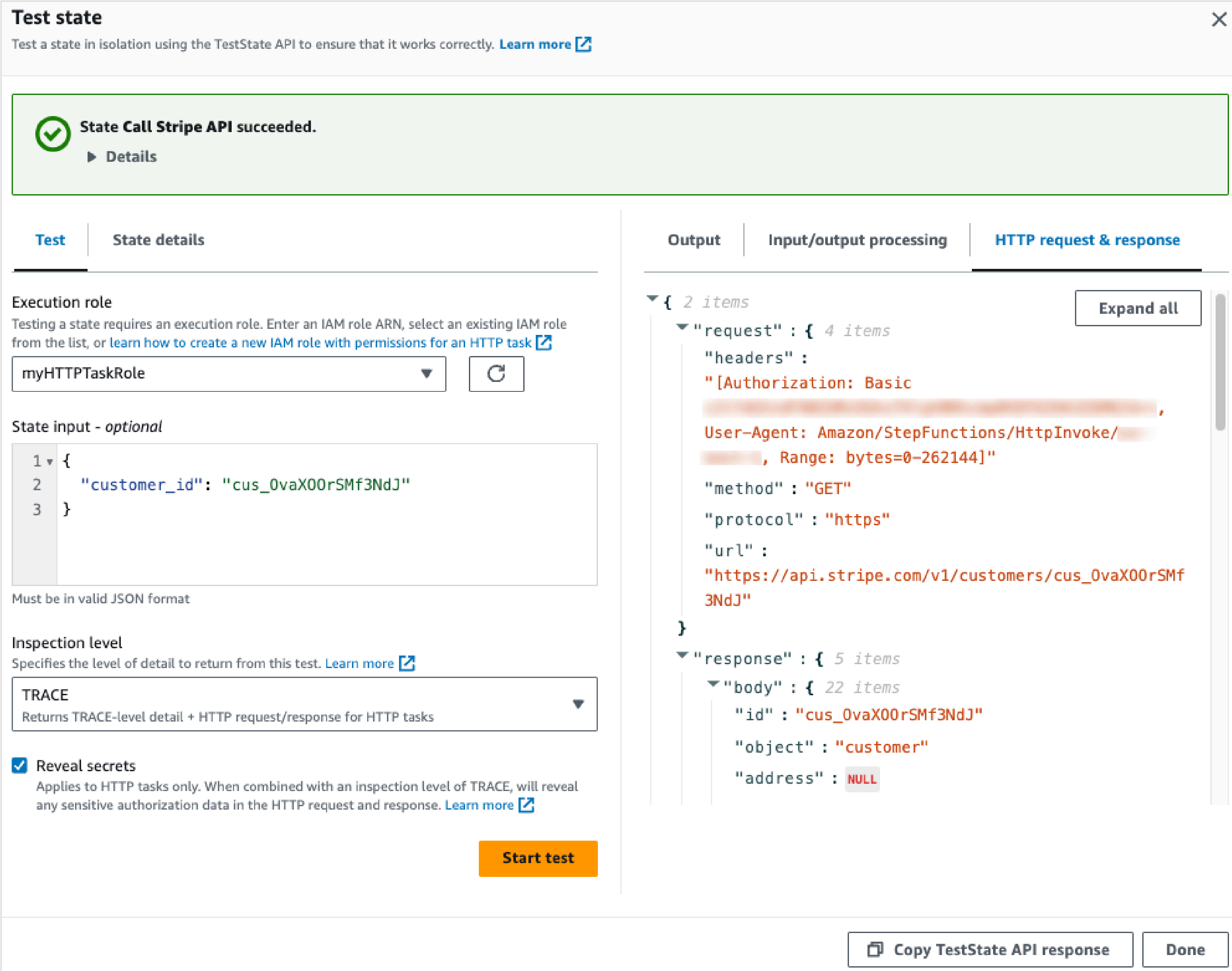
This level is only applicable for HTTP Task. Step Functions throws an error if you use this level for other state types.

When you set the **Inspection level** to **TRACE**, you can also view secrets included in the EventBridge connection. To do this, you must set the `revealSecrets` parameter to `true` in the TestState API. In addition, you must make sure that the IAM user that calls the TestState API has permission for the `states:RevealSecrets` action. For an example of IAM policy that sets the `states:RevealSecrets` permission, see IAM permissions for using TestState API. Without this permission, Step Functions throws an access denied error.

If you set the `revealSecrets` parameter to `false`, Step Functions omits all secrets in the HTTP request and response data.

Example of test with TRACE level that succeeds

The following image shows a test for an HTTP Task that succeeds. The **Inspection level** for this state is set to **TRACE**. The **HTTP request & response** tab in the following image shows the result of the HTTPS API call.



IAM permissions for using TestState API

The IAM user that calls the `TestState` API must have permissions to perform the `states:TestState` and `iam:PassRole` actions. In addition, if you set the revealSecrets parameter to `true`, you must make sure that the IAM user has permissions to perform the `states:RevealSecrets` action. Without this permission, Step Functions throws an access denied error.

You must also make sure that your execution role contains the required IAM permissions for the resources your state is accessing. For information about the permissions a state might need, see Managing execution roles.

The following IAM policy example sets the `states:TestState`, `iam:PassRole`, and `states:RevealSecrets` permissions.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "states:TestState",
        "states:RevealSecrets",
        "iam:PassRole"
      ],
      "Resource": "*"
    }
  ]
}
```

Testing a state (Console)

You can test a state in the console and check the state output or input and output data processing flow. For an HTTP Task, you can test the raw HTTP request and response.

To test a state

1. Open the Step Functions console.
2. Choose **Create state machine** to start creating a state machine or choose an existing state machine.
3. In the Design mode of Workflow Studio, choose a state that you want to test.
4. Choose **Test state** in the Inspector panel panel of Workflow Studio.
5. In the **Test state** dialog box, do the following:

    a. For **Execution role**, choose an execution role to test the state. Make sure that you have the required IAM permissions for the state that you want to test.

    b. (Optional) Provide any JSON input that your selected state needs for the test.

    c. For **Inspection level**, select one of the following options based on the values you want to view:

- INFO – Shows the state output in the **Output** tab if the test succeeds. If the test fails, **INFO** shows the error output, which includes the error name and a detailed explanation of the cause for that error. By default, Step Functions sets **Inspection level** to **INFO** if you don't select a level.
- DEBUG – Shows the state output and the result of input and output data processing if the test succeeds. If the test fails, **DEBUG** shows the error output, which includes the error name and a detailed explanation of the cause for that error.
- TRACE – Shows the raw HTTP request and response, and is useful for verifying headers, query parameters, and other API-specific details. This option is only available for the HTTP Task.

  Optionally, you can choose to select **Reveal secrets**. In combination with **TRACE**, this setting lets you see the sensitive data that the EventBridge connection inserts, such as API keys. The IAM user identity that you use to access the console must have permission to perform the `states:RevealSecrets` action. Without this permission, Step Functions throws an access denied error when you start the test. For an example of an IAM policy that sets the `states:RevealSecrets` permission, see IAM permissions for using TestState API.

d. Choose **Start test**.

Testing a state using AWS CLI

You can test a supported state using the TestState API in the AWS CLI. This API accepts the definition of a state and executes it.

For each state, you can specify the amount of detail you want to view in the test results. These details provide additional information about the state's execution, including its input and output data processing result and HTTP request and response information. The following examples showcase the different inspection levels you can specify for the TestState API. Remember to replace the *`italicized`* text with your resource-specific information.

This section contains the following examples that describe how you can use the different inspection levels that Step Functions provides in the AWS CLI:

- Using INFO inspectionLevel
- Using DEBUG inspectionLevel

- Using TRACE inspectionLevel
- Using jq utility in AWS CLI to filter and print the HTTP response that TestState API returns

Example 1: Using INFO inspectionLevel to test a Choice state

To test a state using the `INFO` inspectionLevel in the AWS CLI, run the `test-state` command as shown in the following example.

```
aws stepfunctions test-state \
    --definition '{"Type": "Choice", "Choices": [{"Variable": "$.number",
 "NumericEquals": 1, "Next": "Equals 1"}, {"Variable": "$.number", "NumericEquals": 2,
 "Next": "Equals 2"}], "Default": "No Match"}' \
    --role-arn arn:aws:iam::account-id:role/myRole \
    --input '{"number": 2}'
```

This example uses a Choice state to determine the execution path for the state based on the numeric input you provide. By default, Step Functions sets the `inspectionLevel` to `INFO` if you don't set a level.

Step Functions returns the following output.

```
{
    "output": "{\"number\": 2}",
    "nextState": "Equals 2",
    "status": "SUCCEEDED"
}
```

Example 2: Using DEBUG inspectionLevel to debug input and output data processing in a Pass state

To test a state using the `DEBUG` inspectionLevel in the AWS CLI, run the `test-state` command as shown in the following example.

```
aws stepfunctions test-state \
    --definition '{"Type": "Pass", "InputPath": "$.payload", "Parameters": {"data": 1},
 "ResultPath": "$.result", "OutputPath": "$.result.data", "Next": "Another State"}' \
    --role-arn arn:aws:iam::account-id:role/myRole \
    --input '{"payload": {"foo": "bar"}}' \
    --inspection-level DEBUG
```

This example uses a Pass workflow state state to showcase how Step Functions filters and manipulates input JSON data using the input and output data processing filters. This example uses these filters: `InputPath`, `Parameters`, `Specifying state output using ResultPath in Step Functions`, and `Filtering state output using OutputPath`.

Step Functions returns the following output.

```
{
    "output": "1",
    "inspectionData": {
        "input": "{\"payload\": {\"foo\": \"bar\"}}",
        "afterInputPath": "{\"foo\":\"bar\"}",
        "afterParameters": "{\"data\":1}",
        "afterResultSelector": "{\"data\":1}",
        "afterResultPath": "{\"payload\":{\"foo\":\"bar\"},\"result\":{\"data\":1}}"
    },
    "nextState": "Another State",
    "status": "SUCCEEDED"
}
```

Example 3: Using TRACE inspectionLevel and revealSecrets to inspect the HTTP request sent to a HTTPS API

To test an HTTP Task using the TRACE inspectionLevel along with the revealSecrets parameter in the AWS CLI, run the `test-state` command as shown in the following example.

```
aws stepfunctions test-state \
    --definition '{"Type": "Task", "Resource": "arn:aws:states:::http:invoke",
 "Parameters": {"Method": "GET", "Authentication": {"ConnectionArn":
 "arn:aws:events:region:account-
id:connection/MyConnection/0000000-0000-0000-0000-000000000000"}, "ApiEndpoint":
 "https://httpbin.org/get", "Headers": {"definitionHeader": "h1"}, "RequestBody":
 {"message": "Hello from Step Functions!"}, "QueryParameters": {"queryParam": "q1"}},
 "End": true}' \
    --role-arn arn:aws:iam::account-id:role/myRole \
    --inspection-level TRACE \
    --reveal-secrets
```

This example tests if the HTTP Task calls the specified HTTPS API, `https://httpbin.org/`. It also shows the HTTP request and response data for the API call.

```
{
    "output": "{\"Headers\":{\"date\":[\"Tue, 21 Nov 2023 00:06:17 GMT\"],
\"access-control-allow-origin\":[\"*\"],\"content-length\":[\"620\"],\"server\":
[\"gunicorn/19.9.0\"],\"access-control-allow-credentials\":[\"true\"],\"content-
type\":[\"application/json\"]},\"ResponseBody\":{\"args\":{\"QueryParam1\":
\"QueryParamValue1\",\"queryParam\":\"q1\"},\"headers\":{\"Authorization
\":\"Basic XXXXXXXX\",\"Content-Type\":\"application/json; charset=UTF-8\",
\"Customheader1\":\"CustomHeaderValue1\",\"Definitionheader\":\"h1\",\"Host\":
\"httpbin.org\",\"Range\":\"bytes=0-262144\",\"Transfer-Encoding\":\"chunked\",
\"User-Agent\":\"Amazon|StepFunctions|HttpInvoke|region\",\"X-Amzn-Trace-Id\":
\"Root=1-0000000-0000-0000-0000-000000000000\"},\"origin\":\"12.34.567.891\",\"url\":
\"https://httpbin.org/get?queryParam=q1&QueryParam1=QueryParamValue1\"},\"StatusCode
\":200,\"StatusText\":\"OK\"}",
    "inspectionData": {
        "input": "{}",
        "afterInputPath": "{}",
        "afterParameters": "{\"Method\":\"GET\",\"Authentication\":{\"ConnectionArn
\":\"arn:aws:events:region:account-id:connection/foo/a59c10f0-a315-4c1f-
be6a-559b9a0c6250\"},\"ApiEndpoint\":\"https://httpbin.org/get\",\"Headers\":
{\"definitionHeader\":\"h1\"},\"RequestBody\":{\"message\":\"Hello from Step Functions!
\"},\"QueryParameters\":{\"queryParam\":\"q1\"}}",
        "result": "{\"Headers\":{\"date\":[\"Tue, 21 Nov 2023 00:06:17 GMT\"],
\"access-control-allow-origin\":[\"*\"],\"content-length\":[\"620\"],\"server\":
[\"gunicorn/19.9.0\"],\"access-control-allow-credentials\":[\"true\"],\"content-
type\":[\"application/json\"]},\"ResponseBody\":{\"args\":{\"QueryParam1\":
\"QueryParamValue1\",\"queryParam\":\"q1\"},\"headers\":{\"Authorization
\":\"Basic XXXXXXXX\",\"Content-Type\":\"application/json; charset=UTF-8\",
\"Customheader1\":\"CustomHeaderValue1\",\"Definitionheader\":\"h1\",\"Host\":
\"httpbin.org\",\"Range\":\"bytes=0-262144\",\"Transfer-Encoding\":\"chunked\",
\"User-Agent\":\"Amazon|StepFunctions|HttpInvoke|region\",\"X-Amzn-Trace-Id\":
\"Root=1-0000000-0000-0000-0000-000000000000\"},\"origin\":\"12.34.567.891\",\"url\":
\"https://httpbin.org/get?queryParam=q1&QueryParam1=QueryParamValue1\"},\"StatusCode
\":200,\"StatusText\":\"OK\"}",
        "afterResultSelector": "{\"Headers\":{\"date\":[\"Tue, 21 Nov 2023
 00:06:17 GMT\"],\"access-control-allow-origin\":[\"*\"],\"content-length\":
[\"620\"],\"server\":[\"gunicorn/19.9.0\"],\"access-control-allow-credentials
\":[\"true\"],\"content-type\":[\"application/json\"]},\"ResponseBody\":{\"args
\":{\"QueryParam1\":\"QueryParamValue1\",\"queryParam\":\"q1\"},\"headers\":
{\"Authorization\":\"Basic XXXXXXXX\",\"Content-Type\":\"application/json;
 charset=UTF-8\",\"Customheader1\":\"CustomHeaderValue1\",\"Definitionheader\":\"h1\",
\"Host\":\"httpbin.org\",\"Range\":\"bytes=0-262144\",\"Transfer-Encoding\":\"chunked
\",\"User-Agent\":\"Amazon|StepFunctions|HttpInvoke|region\",\"X-Amzn-Trace-Id\":
\"Root=1-0000000-0000-0000-0000-000000000000\"},\"origin\":\"12.34.567.891\",\"url\":
```

```
\"https://httpbin.org/get?queryParam=q1&QueryParam1=QueryParamValue1\"},\"StatusCode
\":200,\"StatusText\":\"OK\"}",
        "afterResultPath": "{\"Headers\":{\"date\":[\"Tue, 21 Nov 2023 00:06:17
 GMT\"],\"access-control-allow-origin\":[\"*\"],\"content-length\":[\"620\"],
\"server\":[\"gunicorn/19.9.0\"],\"access-control-allow-credentials\":[\"true\"],
\"content-type\":[\"application/json\"]},\"ResponseBody\":{\"args\":{\"QueryParam1\":
\"QueryParamValue1\",\"queryParam\":\"q1\"},\"headers\":{\"Authorization\":
\"Basic XXXXXXXX\",\"Content-Type\":\"application/json; charset=UTF-8\",
\"Customheader1\":\"CustomHeaderValue1\",\"Definitionheader\":\"h1\",\"Host\":
\"httpbin.org\",\"Range\":\"bytes=0-262144\",\"Transfer-Encoding\":\"chunked\",
\"User-Agent\":\"Amazon|StepFunctions|HttpInvoke|region\",\"X-Amzn-Trace-Id\":
\"Root=1-0000000-0000-0000-0000-000000000000\"},\"origin\":\"12.34.567.891\",\"url\":
\"https://httpbin.org/get?queryParam=q1&QueryParam1=QueryParamValue1\"},\"StatusCode
\":200,\"StatusText\":\"OK\"}",
        "request": {
            "protocol": "https",
            "method": "GET",
            "url": "https://httpbin.org/get?
queryParam=q1&QueryParam1=QueryParamValue1",
            "headers": "[definitionHeader: h1, Authorization: Basic XXXXXXXX,
 CustomHeader1: CustomHeaderValue1, User-Agent: Amazon|StepFunctions|HttpInvoke|region,
 Range: bytes=0-262144]",
            "body": "{\"message\":\"Hello from Step Functions!\",\"BodyKey1\":
\"BodyValue1\"}"
        },
        "response": {
            "protocol": "https",
            "statusCode": "200",
            "statusMessage": "OK",
            "headers": "[date: Tue, 21 Nov 2023 00:06:17 GMT, content-type:
 application/json, content-length: 620, server: gunicorn/19.9.0, access-control-allow-
origin: *, access-control-allow-credentials: true]",
            "body": "{\n  \"args\": {\n    \"QueryParam1\": \"QueryParamValue1\",
 \n    \"queryParam\": \"q1\"\n  }, \n  \"headers\": {\n    \"Authorization
\": \"Basic XXXXXXXX\", \n    \"Content-Type\": \"application/json;
 charset=UTF-8\", \n    \"Customheader1\": \"CustomHeaderValue1\", \n
  \"Definitionheader\": \"h1\", \n    \"Host\": \"httpbin.org\", \n
 \"Range\": \"bytes=0-262144\", \n    \"Transfer-Encoding\": \"chunked\",
 \n    \"User-Agent\": \"Amazon|StepFunctions|HttpInvoke|region\", \n
 \"X-Amzn-Trace-Id\": \"Root=1-0000000-0000-0000-0000-000000000000\"\n  },
 \n  \"origin\": \"12.34.567.891\", \n  \"url\": \"https://httpbin.org/get?
queryParam=q1&QueryParam1=QueryParamValue1\"\n}\n"
        }
    },
```

```
    "status": "SUCCEEDED"
}
```

Example 4: Using jq utility to filter and print the response that TestState API returns

The TestState API returns JSON data as escaped strings in its response. The following AWS CLI example extends Example 3 and uses the `jq` utility to filter and print the HTTP response that the TestState API returns in a human-readable format. For information about `jq` and its installation instructions, see jq on *GitHub*.

```
aws stepfunctions test-state \
    --definition '{"Type": "Task", "Resource": "arn:aws:states:::http:invoke",
 "Parameters": {"Method": "GET", "Authentication": {"ConnectionArn":
 "arn:aws:events:region:account-
id:connection/MyConnection/0000000-0000-0000-0000-000000000000"}, "ApiEndpoint":
 "https://httpbin.org/get", "Headers": {"definitionHeader": "h1"}, "RequestBody":
 {"message": "Hello from Step Functions!"}, "QueryParameters": {"queryParam": "q1"}},
 "End": true}' \
    --role-arn arn:aws:iam::account-id:role/myRole \
    --inspection-level TRACE \
    --reveal-secrets \
    | jq '.inspectionData.response.body | fromjson'
```

The following example shows the output returned in a human-readable format.

```
{
  "args": {
    "QueryParam1": "QueryParamValue1",
    "queryParam": "q1"
  },
  "headers": {
    "Authorization": "Basic XXXXXXXX",
    "Content-Type": "application/json; charset=UTF-8",
    "Customheader1": "CustomHeaderValue1",
    "Definitionheader": "h1",
    "Host": "httpbin.org",
    "Range": "bytes=0-262144",
    "Transfer-Encoding": "chunked",
    "User-Agent": "Amazon|StepFunctions|HttpInvoke|region",
    "X-Amzn-Trace-Id": "Root=1-0000000-0000-0000-0000-000000000000"
```

```
  },
  "origin": "12.34.567.891",
  "url": "https://httpbin.org/get?queryParam=q1&QueryParam1=QueryParamValue1"
}
```

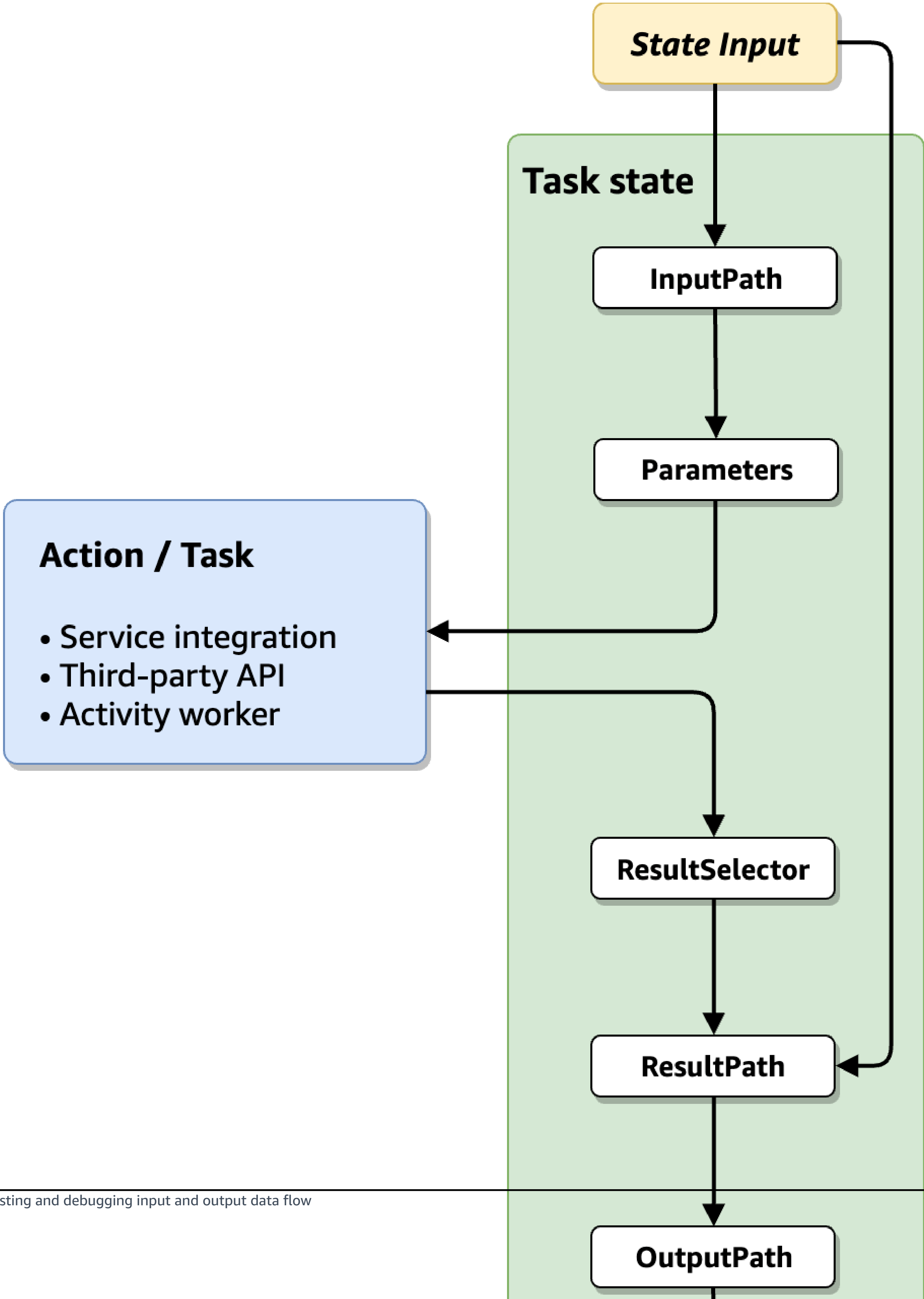
Testing and debugging input and output data flow

The `TestState` API is helpful for testing and debugging the data that flows through your workflow. This section provides some key concepts and explains how to use the TestState for this purpose.

Key concepts

In Step Functions, the process of filtering and manipulating JSON data as it passes through the states in your state machine is called *input and output processing*. For information about how this works, see Processing input and output in Step Functions.

All the state types in the Amazon States Language (ASL) (Task, Parallel, Map, Pass, Wait, Choice, Succeed, and Fail) share a set of common fields for filtering and manipulating the JSON data that passes through them. These fields are: InputPath, Parameters, ResultSelector, Specifying state output using ResultPath in Step Functions, and Filtering state output using OutputPath. Support for each field varies across states. At runtime, Step Functions applies each field in a specific order. The following diagram shows the order in which these fields are applied to the data inside a Task state:


State Input
Task state
InputPath
Parameters
Action / Task
• Service integration
• Third-party API
• Activity worker
ResultSelector
ResultPath
OutputPath

The following list describes the order of application of the input and output processing fields shown in the diagram.

1. *State input* is the JSON data passed to the current state from a previous state.
2. InputPath filters a portion of the raw state input.
3. Parameters configures the set of values to pass to the Task.
4. The task performs work and returns a result.
5. ResultSelector selects a set of values to keep from the task result.
6. Specifying state output using ResultPath in Step Functions combines the result with the raw state input, or replaces the result with it.
7. Filtering state output using OutputPath filters a portion of the output to pass along to the next state.
8. *State output* is the JSON data passed from the current state to the next state.

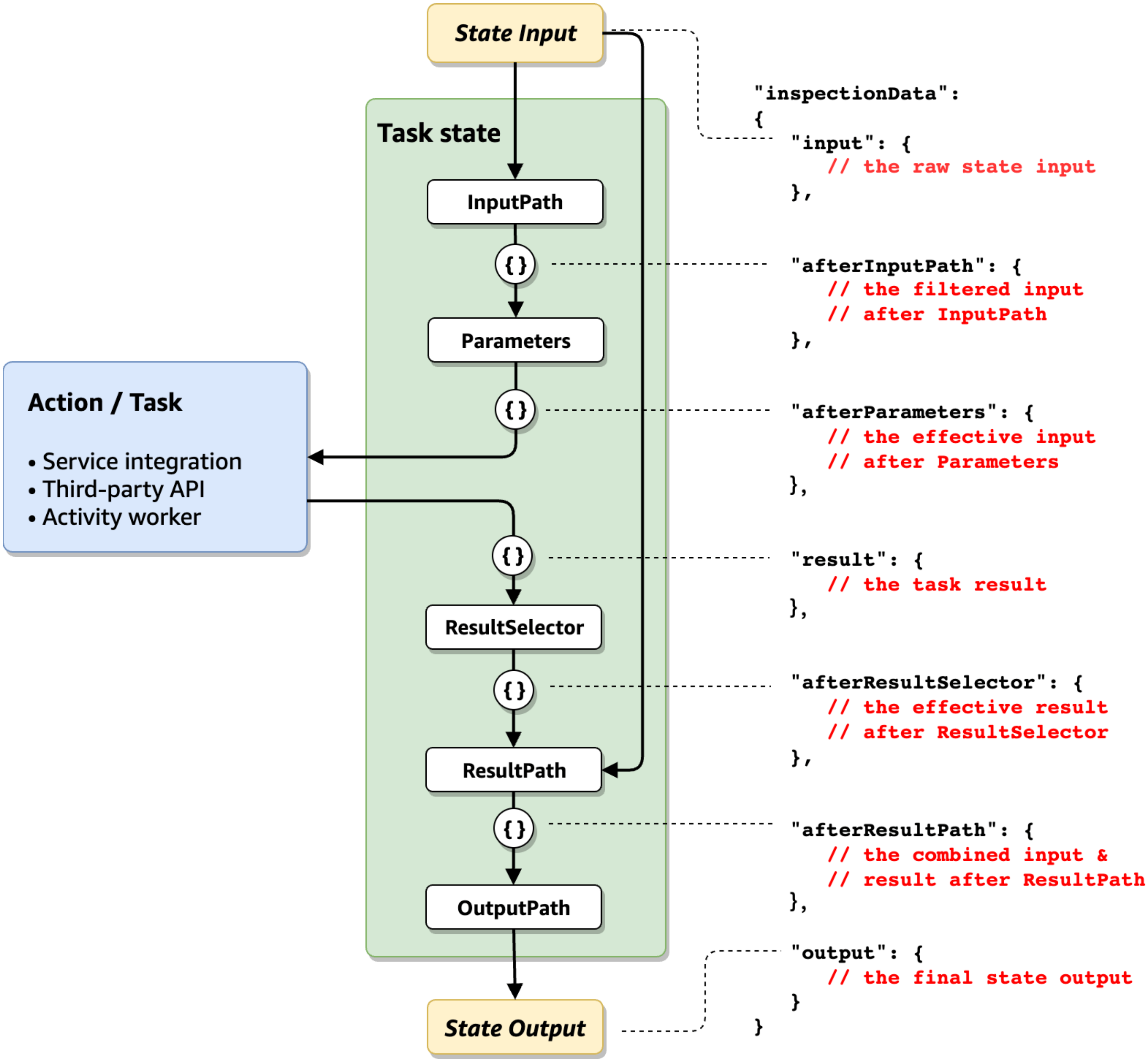
These input and output processing fields are optional. If you don't use any of these fields in your state definition, the task will consume the raw state input, and return the task result as the state output.

Using TestState to inspect input and output processing

When you call the `TestState` API and set the `inspectionLevel` parameter to `DEBUG`, the API response includes an object called `inspectionData`. This object contains fields to help you inspect how data was filtered or manipulated within the state when it was executed. The following example shows the `inspectionData` object for a Task state.

```
"inspectionData":   {
  "input": string,
  "afterInputPath": string,
  "afterParameters": string,
  "result": string,
  "afterResultSelector": string,
  "afterResultPath": string,
  "output": string
}
```

In this example, each field that contains the `after` prefix, shows the data after a particular field was applied. For example, `afterInputPath` shows the effect of applying the `InputPath` field to filter the raw state input. The following diagram maps each ASL definition field to its corresponding field in the `inspectionData` object:



For examples of using the TestState API to debug input and output processing, see the following:

- Testing a state using the DEBUG inspection level in the Step Functions console
- Testing a state using the DEBUG inspection level in the AWS CLI

Testing state machines with Step Functions Local (unsupported)

Step Functions Local is unsupported

Step Functions Local does **not** provide feature parity and is **unsupported**.
You might consider third party solutions that emulate Step Functions for testing purposes.

With AWS Step Functions Local, a downloadable version of Step Functions, you can test applications with Step Functions running in your own development environment.

When 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.
- Making calls directly to an AWS service from Step Functions Local.
- Mocking the response from service integrations.

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

You should only use Step Functions Local for testing and never to process sensitive information.

Topics

Setting Up Step Functions Local (Downloadable Version) in Docker

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

To get the Docker image for Step Functions Local, 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
- Credentials and configuration for Docker

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.

Download Links	Checksum
.tar.gz	.tar.gz.md5

Download Links	Checksum
.zip	.zip.md5

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

```
$ java -jar StepFunctionsLocal.jar -v
Step Function Local
Version: 2.0.0
Build: 2024-05-18
```

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:region:account-id:express:test:example-ExpressLogGroup-
wJalrXUtnFEMI
```

the log file will be:

```
aws/states/log-group-name/arn:aws:states:region:account-id:express:test:example-
ExpressLogGroup-wJalrXUtnFEMI.log
```

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

When you configure the Step Functions Local container to use an override endpoint such as Lambda Endpoint and Batch Endpoint, and make calls to that endpoint, Step Functions Local doesn't use the credentials you specify. Setting these endpoint overrides is optional.

Option	Command Line	Environment
Account	-account, --aws-account	AWS_ACCOUNT_ID
Region	-region, --aws-region	AWS_DEFAULT_REGION
Wait Time Scale	-waitTimeScale, --wait-time-scale	WAIT_TIME_SCALE
Lambda Endpoint	-lambdaEndpoint, --lambda-endpoint	LAMBDA_ENDPOINT
Batch Endpoint	-batchEndpoint, --batch-endpoint	BATCH_ENDPOINT

Option	Command Line	Environment
DynamoDB Endpoint	-dynamoDBEndpoint, --dynamodb-endpoint	DYNAMODB_ENDPOINT
ECS Endpoint	-ecsEndpoint, --ecs-endpoint	ECS_ENDPOINT
Glue Endpoint	-glueEndpoint, --glue-endpoint	GLUE_ENDPOINT
SageMaker Endpoint	-sageMakerEndpoint, --sagemaker-endpoint	SAGE_MAKER_ENDPOINT
SQS Endpoint	-sqsEndpoint, --sqs-endpoint	SQS_ENDPOINT
SNS Endpoint	-snsEndpoint, --sns-endpoint	SNS_ENDPOINT
Step Functions Endpoint	-stepFunctionsEndpoint, --step-functions-endpoint	STEP_FUNCTIONS_ENDPOINT

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.

```
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.

```
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,
    "stateMachineArn": "arn:aws:states:region:account-id:stateMachine:HelloWorld"
}
```

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

```
aws stepfunctions --endpoint-url http://localhost:8083 start-execution --state-machine-arn arn:aws:states:region:account-id:stateMachine:HelloWorld
```

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 Tutorial: Testing workflows using Step Functions and AWS SAM CLI Local.

Tutorial: Testing workflows using Step Functions and AWS SAM CLI Local

> **Step Functions Local is unsupported**
>
> Step Functions Local does **not** provide feature parity and is **unsupported**.
> You might consider third party solutions that emulate Step Functions for testing purposes.

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 state machines with Step Functions Local (unsupported)
- Set Up AWS SAM

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:

```
sam local start-api
```

This launches a local instance of your Lambda function. You should see output similar to the following:

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

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

Step 4: Start Step Functions Local

JAR File

If 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 account-id, region
 us-east-1
```

Docker

If you're using the Docker version of Step Functions Local, launch Step Functions with the following command:

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

For information about installing the Docker version of Step Functions, see Setting Up Step Functions Local (Downloadable Version) in Docker.

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.

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

After Step Functions Local is running, create a state machine that references the `HelloWorldFunction` that you initialized in Step 1: Set Up AWS SAM.

```
aws stepfunctions --endpoint http://localhost:8083 create-state-machine --definition
 "{\
  \"Comment\": \"A Hello World example of the Amazon States Language using an AWS
 Lambda Local function\",\
  \"StartAt\": \"HelloWorld\",\
  \"States\": {\
    \"HelloWorld\": {\
      \"Type\": \"Task\",\
      \"Resource\": \"arn:aws:lambda:region:account-id:function:HelloWorldFunction\",\
      \"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.

```
{
    "creationDate": 1548805711.403,
    "stateMachineArn": "arn:aws:states:region:account-id:stateMachine:HelloWorld"
}
```

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:

```
aws stepfunctions --endpoint http://localhost:8083 start-execution --state-machine
 arn:aws:states:region:account-id:stateMachine:HelloWorld --name test
```

This starts an execution named `test` of your `HelloWorld` state machine.

```
{
    "startDate": 1548810641.52,
    "executionArn": "arn:aws:states:region:account-id: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:

```
aws stepfunctions --endpoint http://localhost:8083 describe-execution --execution-arn
 arn:aws:states:region:account-id:execution:HelloWorld:test
```

Calling `describe-execution` for an execution provides more complete details, similar to the following output:

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

Using mocked service integrations for testing in Step Functions Local

Step Functions Local is unsupported

Step Functions Local does **not** provide feature parity and is **unsupported**.
You might consider third party solutions that emulate Step Functions for testing purposes.

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](#).

This topic uses several concepts which are defined in the following list:

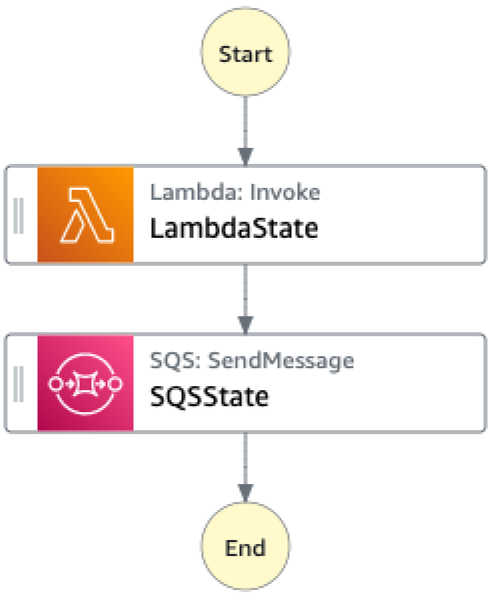
- Mocked Service Integrations - Refers to Task states configured to use mocked responses instead of performing actual service calls.
- Mocked Responses - Refers to mock data that Task states can be configured to use.
- Test Cases - Refers to state machine executions configured to use mocked service integrations.
- Mock Configuration File - Refers to mock configuration file that contains JSON, which defines mocked service integrations, mocked responses, and test cases.

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`.

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 Mock configuration file structure.

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 and sending a message to Amazon SQS. In this example, the `LambdaSQSIntegration` 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.

   ```
   {
   ```

```
  "StateMachines":{
    "LambdaSQSIntegration":{
      "TestCases":{
        "HappyPath":{
          "LambdaState":"MockedLambdaSuccess",
          "SQSState":"MockedSQSSuccess"
        },
        "RetryPath":{
          "LambdaState":"MockedLambdaRetry",
          "SQSState":"MockedSQSSuccess"
        },
        "HybridPath":{
          "LambdaState":"MockedLambdaSuccess"
        }
      }
    }
  },
  "MockedResponses":{
    "MockedLambdaSuccess":{
      "0":{
        "Return":{
          "StatusCode":200,
          "Payload":{
            "StatusCode":200,
            "body":"Hello from Lambda!"
          }
        }
      }
    },
    "LambdaMockedResourceNotReady":{
      "0":{
        "Throw":{
          "Error":"Lambda.ResourceNotReadyException",
          "Cause":"Lambda resource is not ready."
        }
      }
    },
    "MockedSQSSuccess":{
      "0":{
        "Return":{
          "MD5OfMessageBody":"3bcb6e8e-7h85-4375-b0bc-1a59812c6e51",
          "MessageId":"3bcb6e8e-8b51-4375-b0bc-1a59812c6e51"
        }
      }
```

```
    },
    "MockedLambdaRetry":{
      "0":{
        "Throw":{
          "Error":"Lambda.ResourceNotReadyException",
          "Cause":"Lambda resource is not ready."
        }
      },
      "1-2":{
        "Throw":{
          "Error":"Lambda.TimeoutException",
          "Cause":"Lambda timed out."
        }
      },
      "3":{
        "Return":{
          "StatusCode":200,
          "Payload":{
            "StatusCode":200,
            "body":"Hello from Lambda!"
          }
        }
      }
    }
  }
}
```

State machine definition

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`.

```
{
  "Comment":"This state machine is called: LambdaSQSIntegration",
  "StartAt":"LambdaState",
  "States":{
    "LambdaState":{
      "Type":"Task",
      "Resource":"arn:aws:states:::lambda:invoke",
      "Parameters":{
        "Payload.$":"$",
        "FunctionName":"HelloWorldFunction"
```

```
      },
      "Retry":[
        {
          "ErrorEquals":[
            "States.ALL"
          ],
          "IntervalSeconds":2,
          "MaxAttempts":3,
          "BackoffRate":2
        }
      ],
      "Next":"SQSState"
    },
    "SQSState":{
      "Type":"Task",
      "Resource":"arn:aws:states:::sqs:sendMessage",
      "Parameters":{
        "QueueUrl":"https://sqs.us-east-1.amazonaws.com/account-id/myQueue",
        "MessageBody.$":"$"
      },
      "End": true
    }
  }
}
```

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:

```
"0":{
  "Return":{
    "StatusCode":200,
    "Payload":{
      "StatusCode":200,
      "body":"Hello from Lambda!"
    }
  }
}
```

- The `SQSState` will return the following value:

```
"0":{
  "Return":{
    "MD5OfMessageBody":"3bcb6e8e-7h85-4375-b0bc-1a59812c6e51",
    "MessageId":"3bcb6e8e-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:

```
"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:

```
"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.

```
"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.

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.

   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.

```
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.

```
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 `Step FunctionsLocal.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.

```
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 `Step FunctionsLocal.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.

```
aws stepfunctions create-state-machine \
    --endpoint http://localhost:8083 \
    --definition "{\"Comment\":\"Thisstatemachineiscalled:LambdaSQSIntegration
\",\"StartAt\":\"LambdaState\",\"States\":{\"LambdaState\":{\"Type\":\"Task\",
```

```
\"Resource\":\"arn:aws:states:::lambda:invoke\",\"Parameters\":{\"Payload.$\":\"$
\",\"FunctionName\":\"arn:aws:lambda:region:account-id:function:HelloWorldFunction
\"},\"Retry\":[{\"ErrorEquals\":[\"States.ALL\"],\"IntervalSeconds\":2,
\"MaxAttempts\":3,\"BackoffRate\":2}],\"Next\":\"SQSState\"},\"SQSState\":
{\"Type\":\"Task\",\"Resource\":\"arn:aws:states:::sqs:sendMessage\",\"Parameters
\":{\"QueueUrl\":\"https://sqs.us-east-1.amazonaws.com/account-id/myQueue\",
\"MessageBody.$\":\"$\"},\"End\":true}}}" \
    --name "LambdaSQSIntegration" --role-arn "arn:aws:iam::account-id:role/service-
role/LambdaSQSIntegration"
```

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](#). 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.

```
aws stepfunctions start-execution \
    --endpoint http://localhost:8083 \
    --name executionWithHappyPathMockedServices \
    --state-machine arn:aws:states:region:account-
id: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:

```
{
   "startDate":"2022-01-28T15:03:16.981000-05:00",
```

```
    "executionArn":"arn:aws:states:region:account-
id:execution:LambdaSQSIntegration:executionWithHappyPathMockedServices"
}
```

4. Check the execution's results by making a `ListExecutions`, `DescribeExecution`, or `GetExecutionHistory` API call.

```
aws stepfunctions get-execution-history \
    --endpoint http://localhost:8083 \
    --execution-arn arn:aws:states:region:account-
id:execution:LambdaSQSIntegration:executionWithHappyPathMockedServices
```

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. 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.

```
{
    "events": [
        ...
        {
            "timestamp": "2021-12-02T19:39:48.988000+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!\\
\"\"}",
                "outputDetails": {
                    "truncated": false
                }
            }
        },
        ...
            "timestamp": "2021-12-02T19:39:49.464000+00:00",
            "type": "TaskStateEntered",
            "id": 7,
            "previousEventId": 6,
            "stateEnteredEventDetails": {
                "name": "SQSState",
                "input": "{\"statusCode\":200,\"body\":\"\\\"Hello from Lambda!\\
\"\"}",
                "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 in Step Functions

> **Step Functions Local is unsupported**
>
> Step Functions Local does **not** provide feature parity and is **unsupported**.
> You might consider third party solutions that emulate Step Functions for testing purposes.

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:

Mock configuration file structure

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`.

```
{
  "StateMachines":{
    "LambdaSQSIntegration":{
      "TestCases":{
        "HappyPath":{
          "LambdaState":"MockedLambdaSuccess",
          "SQSState":"MockedSQSSuccess"
        },
        "RetryPath":{
          "LambdaState":"MockedLambdaRetry",
          "SQSState":"MockedSQSSuccess"
        },
        "HybridPath":{
          "LambdaState":"MockedLambdaSuccess"
```

```
        }
      }
    }
  },
  "MockedResponses":{
    "MockedLambdaSuccess":{
      "0":{
        "Return":{
          "StatusCode":200,
          "Payload":{
            "StatusCode":200,
            "body":"Hello from Lambda!"
          }
        }
      }
    },
    "LambdaMockedResourceNotReady":{
      "0":{
        "Throw":{
          "Error":"Lambda.ResourceNotReadyException",
          "Cause":"Lambda resource is not ready."
        }
      }
    },
    "MockedSQSSuccess":{
      "0":{
        "Return":{
          "MD5OfMessageBody":"3bcb6e8e-7h85-4375-b0bc-1a59812c6e51",
          "MessageId":"3bcb6e8e-8b51-4375-b0bc-1a59812c6e51"
        }
      }
    },
    "MockedLambdaRetry":{
      "0":{
        "Throw":{
          "Error":"Lambda.ResourceNotReadyException",
          "Cause":"Lambda resource is not ready."
        }
      },
      "1-2":{
        "Throw":{
          "Error":"Lambda.TimeoutException",
          "Cause":"Lambda timed out."
        }
```

```
      },
      "3":{
        "Return":{
          "StatusCode":200,
          "Payload":{
            "StatusCode":200,
            "body":"Hello from Lambda!"
          }
        }
      }
    }
  }
}
```

Mock configuration field reference

The following sections explain the top-level object fields that you must define in your mock configuration.

- StateMachines
- MockedResponses

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:

```
"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`:

```
"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.

```
"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:

```
"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] 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:

```
"Throw": {
  "Error": "Lambda.TimeoutException",
  "Cause": "Lambda timed out."
}
```

Manage continuous deployments with versions and aliases in Step Functions

You can use Step Functions to manage continuous deployments of your workflows through state machine *versions* and *aliases*. A *version* is a numbered, immutable snapshot of a state machine that you can run. An *alias* is a pointer for up to two versions of a state machine.

You can maintain multiple versions of your state machines and manage their deployment in your production workflow. With aliases, you can route traffic between different workflow versions and gradually deploy those workflows to the production environment.

Additionally, you can start state machine executions using a version or an alias. If you don't use a version or alias when you start a state machine execution, Step Functions uses the latest revision of the state machine definition.

> **State machine revision**
>
> A state machine can have one or more revisions. When you update a state machine using the [UpdateStateMachine](#) API action, it creates a new state machine revision. A *revision* is an immutable, read-only snapshot of a state machine's definition and configuration. You can't start a state machine execution from a revision, and revisions don't have an ARN. Revisions have a `revisionId`, which is a universally unique identifier (UUID).

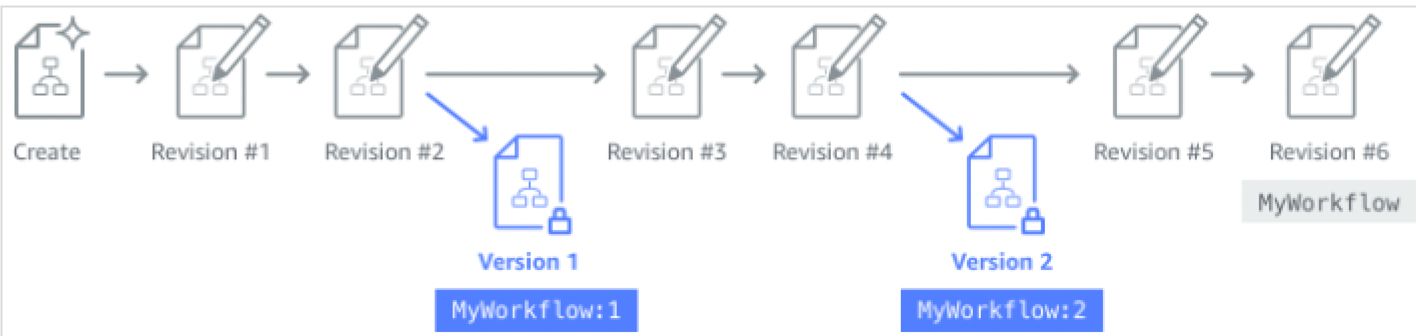
Contents

State machine versions in Step Functions workflows

A *version* is a numbered, **immutable** snapshot of a state machine. You publish versions from the most recent revision made to that state machine. Each version has a unique Amazon Resource Name (ARN) which is a combination of state machine ARN and the version number separated by a colon (:). The following example shows the format of a state machine version ARN.

```
arn:partition:states:region:account-id:stateMachine:myStateMachine:1
```

To start using state machine versions, you must publish the first version. After you publish a version, you can invoke the StartExecution API action with the version ARN. You can't edit a version, but you can update a state machine and publish a new version. You can also publish multiple versions of your state machine.



When you publish a new version of your state machine, Step Functions assigns it a version number. Version numbers start at 1 and increase monotonically for each new version. Version numbers aren't reused for a given state machine. If you delete version 10 of your state machine and then publish a new version, Step Functions publishes it as version 11.

The following properties are the same for all versions of a state machine:

- All versions of a state machine share the same type (Standard or Express).
- You can't change the name or creation date of a state machine between versions.
- Tags apply globally to state machines. You can manage tags for state machines using the TagResource and UntagResource API actions.

State machines also contain properties that are a part of each version and revision, but these properties can differ between two given versions or revisions. These properties include State machine definition, IAM role, tracing configuration, and logging configuration.

Publishing a state machine version (Console)

You can publish up to 1000 versions of a state machine. To request an increase to this soft limit, use the **Support Center** page in the AWS Management Console. You can manually delete unused versions from the console or by invoking the DeleteStateMachineVersion API action.

To publish a state machine version

1. Open the Step Functions console, and then choose an existing state machine.
2. On the **State machine detail** page, choose **Edit**.
3. Edit the state machine definition as required, and then choose **Save**.
4. Choose **Publish version**.
5. (Optional) In the **Description** field of the dialog box that appears, enter a brief description about the state machine version.
6. Choose **Publish**.

> **Note**
>
> When you publish a new version of your state machine, Step Functions assigns it a version number. Version numbers start at 1 and increase monotonically for each new version. Version numbers aren't reused for a given state machine. If you delete version 10 of your state machine and then publish a new version, Step Functions publishes it as version 11.

Managing versions with Step Functions API operations

Step Functions provides the following API operations to publish and manage state machine versions:

- PublishStateMachineVersion – Publishes a version from the current revision of a state machine.
- UpdateStateMachine – Publishes a new state machine version if you update a state machine and set the `publish` parameter to `true` in the same request.
- CreateStateMachine – Publishes the first revision of the state machine if you set the `publish` parameter to `true`.
- ListStateMachineVersions – Lists versions for the specified state machine ARN.

- DescribeStateMachine – Returns the state machine version details for a version ARN specified in `stateMachineArn`.
- DeleteStateMachineVersion – Deletes a state machine version.

To publish a new version from the current revision of a state machine called *myStateMachine* using the AWS Command Line Interface, use the `publish-state-machine-version` command:

```
aws stepfunctions publish-state-machine-version --state-machine-arn
 arn:aws:states:region:account-id:stateMachine:myStateMachine
```

The response returns the `stateMachineVersionArn`. For example, the previous command returns a response of`arn:aws:states:region:account-id:stateMachine:myStateMachine:1`.

Note

When you publish a new version of your state machine, Step Functions assigns it a version number. Version numbers start at 1 and increase monotonically for each new version. Version numbers aren't reused for a given state machine. If you delete version 10 of your state machine and then publish a new version, Step Functions publishes it as version 11.

Running a state machine version from the console

To start using state machine versions, you must first publish a version from the current state machine revision. To publish a version, use the Step Functions console or invoke the PublishStateMachineVersion API action. You can also invoke the UpdateStateMachineAlias API action with an optional parameter named `publish` to update a state machine and publish its version.

You can start executions of a version by using the console or by invoking the StartExecution API action and providing the version ARN. You can also use an alias to start executions of a version. Based on its routing configuration, an alias routes traffic to a specific version.

If you start a state machine execution without using a version, Step Functions uses the most recent revision of the state machine for the execution. For information about how Step Functions associates an execution with a version, see Associating executions with a version or alias.

To start an execution using a state machine version

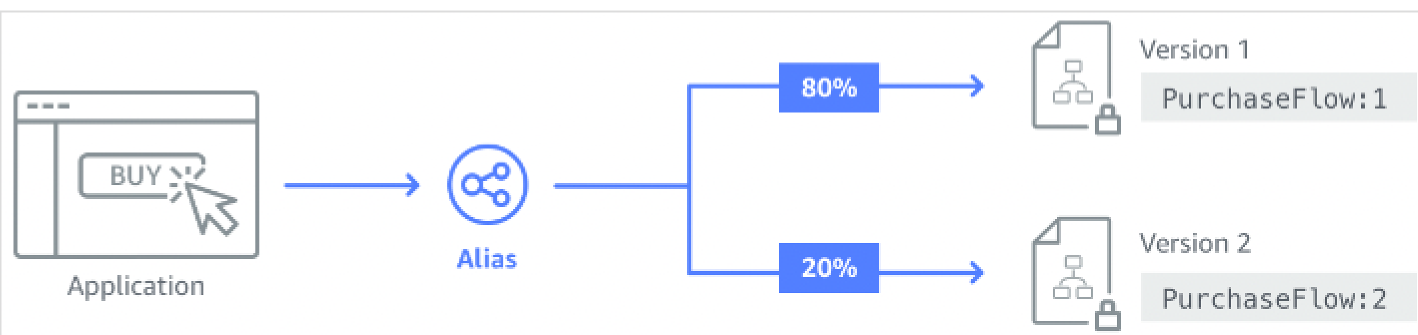
1. Open the [Step Functions console](), and then choose an existing state machine that you've published one or more versions for. To learn how to publish a version, see [Publishing a state machine version (Console)]().
2. On the **State machine detail** page, choose the **Versions** tab.
3. In the **Versions** section, do the following:
   a. Select the version that you want to start the execution with.
   b. Choose **Start execution**.
4. (Optional) In the **Start execution** dialog box, enter a name for the execution.
5. (Optional) , enter the execution input, and then choose **Start execution**.

State machine aliases in Step Functions workflows

An *alias* is a pointer for up to two versions of the same state machine. You can create multiple aliases for your state machines. Each alias has a unique Amazon Resource Name (ARN). The alias ARN is a combination of the state machine's ARN and the alias name, separated by a colon (:). The following example shows the format of a state machine alias ARN.

```
arn:partition:states:region:account-id:stateMachine:myStateMachine:aliasName
```

You can use an alias to [route traffic]() between one of the two state machine versions. You can also create an alias that points to a single version. Aliases can only point to state machine versions. You can't use an alias to point to another alias. You can also update an alias to point to a different version of the state machine.



Contents

- Managing aliases with Step Functions API operations
- Alias routing configuration
- Running a state machine using an alias (Console)

Creating a state machine alias (Console)

You can create up to 100 aliases for each state machine by using the Step Functions console or by invoking the CreateStateMachineAlias API action. To request an increase to this soft limit, use the **Support Center** page in the AWS Management Console. Delete unused aliases from the console or by invoking the DeleteStateMachineAlias API action.

To create a state machine alias

1. Open the Step Functions console, and then choose an existing state machine.
2. On the **State machine detail** page, choose the **Aliases** tab.
3. Choose **Create new alias**.
4. On the **Create alias** page, do the following:
   a. Enter an **Alias name**.
   b. (Optional) Enter a **Description** for the alias.
5. To configure routing on the alias, see Alias routing configuration.
6. Choose **Create alias**.

Managing aliases with Step Functions API operations

Step Functions provides the following API operations that you can use to create and manage state machine aliases or get information about the aliases:

- CreateStateMachineAlias – Creates an alias for a state machine.
- DescribeStateMachineAlias – Returns details about a state machine alias.
- ListStateMachineAliases – Lists aliases for the specified state machine ARN.
- UpdateStateMachineAlias – Updates the configuration of an existing state machine alias by modifying its `description` or `routingConfiguration`.
- DeleteStateMachineAlias – Deletes a state machine version.

To create an alias named *PROD* that points to version 1 of a state machine named *myStateMachine* using the AWS Command Line Interface, use the `create-state-machine-alias` command.

```
aws stepfunctions create-state-machine-alias --name PROD  --routing-
configuration "[{\"stateMachineVersionArn\":\"arn:aws:states:region:account-
id:stateMachine:myStateMachine:1\",\"weight\":100}]"
```

Alias routing configuration

You can use an alias to route execution traffic between two versions of a state machine. For example, say you want to launch a new version of your state machine. You can reduce the risks involved in deploying the new version by configuring routing on an alias. By configuring routing, you can send most of your traffic to an earlier, tested version of your state machine. The new version can then receive a smaller percentage, until you can confirm that it's safe to roll forward the new version.

To define routing configuration, make sure that you publish both state machine versions that your alias points to. When you start an execution from an alias, Step Functions randomly chooses the state machine version to run from the versions specified in the routing configuration. It bases this choice on the traffic percentage that you assign to each version in the alias routing configuration.

To configure routing configuration on an alias

- On the **Create alias** page, under **Routing configuration**, do the following:

    a. For **Version**, choose the first state machine version that the alias points to.

    b. Select the **Split traffic between two versions** check box.

    > **Tip**
    >
    > To point to a single version, clear the **Split traffic between two versions** check box.

    c. For **Version**, choose the second version that the alias must point to.

    d. In the **Traffic percentage** fields, specify the percentage of traffic to route to each version. For example, enter **60** and **40** to route 60 percent of the execution traffic to the first version and 40 percent traffic to the second version.

The combined traffic percentages must equal to 100 percent.

Running a state machine using an alias (Console)

You can start state machine executions with an alias from either the console or by invoking the StartExecution API action with the alias' ARN. Step Functions then runs the version specified by the alias. By default, if you don't specify a version or alias when you start a state machine execution, Step Functions uses the most recent revision.

To start a state machine execution using an alias

1. Open the Step Functions console, then choose an existing state machine that you've created an alias for. For information about creating an alias, see Creating a state machine alias (Console).
2. On the **State machine detail** page, choose the **Aliases** tab.
3. In the **Aliases** section, do the following:
   a. Select the alias that you want to start the execution with.
   b. Choose **Start execution**.
4. (Optional) In the **Start execution** dialog box, enter a name for the execution.
5. If required, enter the execution input, and then choose **Start execution**.

Authorization for versions and aliases in Step Functions workflows

To invoke Step Functions API actions with a version or an alias, you need appropriate permissions. To authorize a version or an alias to invoke an API action, Step Functions uses the state machine's ARN instead of using the version ARN or alias ARN. You can also scope down the permissions for a specific version or alias. For more information, see Scoping down permissions.

You can use the following IAM policy example of a state machine named *myStateMachine* to invoke the CreateStateMachineAlias API action to create a state machine alias.

```
{
  "Version": "2012-10-17",
  "Statement": [
```

```
    {
      "Effect": "Allow",
      "Action": "states:CreateStateMachineAlias",
      "Resource": "arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine"
    }
  ]
}
```

When you set permissions to allow or deny access to API actions using state machine versions or aliases, consider the following:

- If you use the `publish` parameter of the CreateStateMachine and UpdateStateMachine API actions to publish a new state machine version, you also need the `ALLOW` permission on the PublishStateMachineVersion API action.
- The DeleteStateMachine API action deletes all versions and aliases associated with a state machine.

Scoping down permissions for a version or alias

You can use a qualifier to further scope down the authorization permission needed by a version or an alias. A qualifier refers to a version number or an alias name. You use the qualifier to qualify a state machine. The following example is a state machine ARN that uses an alias named `PROD` as the qualifier.

```
arn:aws:states:region:account-id:stateMachine:myStateMachine:PROD
```

For more information about qualified and unqualified ARNs, see Associating executions with a version or alias.

You scope down the permissions using the optional context key named `states:StateMachineQualifier` in an IAM policy's `Condition` statement. For example, the following IAM policy for a state machine named `myStateMachine` denies access to invoke the DescribeStateMachine API action with an alias named as `PROD` or the version `1`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
```

```
      "Action": "states:DescribeStateMachine",
      "Resource": "arn:aws:states:us-east-1:123456789012:stateMachine:myStateMachine",
      "Condition": {
        "ForAnyValue:StringEquals": {
          "states:StateMachineQualifier": [
            "PROD",
            "1"
          ]
        }
      }
    }
  ]
}
```

The following list specifies the API actions on which you can scope down the permissions with the `StateMachineQualifier` context key.

- CreateStateMachineAlias
- DeleteStateMachineAlias
- DeleteStateMachineVersion
- DescribeStateMachine
- DescribeStateMachineAlias
- ListExecutions
- ListStateMachineAliases
- StartExecution
- StartSyncExecution
- UpdateStateMachineAlias

How Step Functions associates executions with a version or alias

Step Functions associates an execution with a version or alias based on the Amazon Resource Name (ARN) that you use to invoke the StartExecution API action. Step Functions performs this action at the execution start time.

You can start a state machine execution using a qualified or an unqualified ARN.

- **Qualified ARN** – Refers to a state machine ARN suffixed with a version number or an alias name.

  The following qualified ARN example refers to version 3 of a state machine named `myStateMachine`.

  ```
  arn:aws:states:region:account-id:stateMachine:myStateMachine:3
  ```

  The following qualified ARN example refers to an alias named `PROD` of a state machine named `myStateMachine`.

  ```
  arn:aws:states:region:account-id:stateMachine:myStateMachine:PROD
  ```

- **Unqualified ARN** – Refers to a state machine ARN without a version number or an alias name suffix.

  ```
  arn:aws:states:region:account-id:stateMachine:myStateMachine
  ```

For example, if your qualified ARN refers to version 3, Step Functions associates the execution with this version. It doesn't associate the execution with any aliases that point to the version 3.

If your qualified ARN refers to an alias, Step Functions associates the execution with that alias and the version to which the alias points. An execution can only be associated with one alias.

> **Note**
>
> If you start an execution with an unqualified ARN, Step Functions doesn't associate that execution with a version even if the version uses the same state machine revision. For example, if version 3 uses the latest revision, but you start an execution with an unqualified ARN, Step Functions doesn't associate that execution with the version 3.

Viewing executions started with a version or an alias

Step Functions provides the following ways in which you can view the executions started with a version or an alias:

Using API actions

You can view all the executions associated with a version or an alias by invoking the DescribeExecution and ListExecutions API actions. These API actions return the ARN of the version or alias that was used to start the execution. These actions also return other details including status and ARN of the execution.

You can also provide a state machine alias ARN or version ARN to list the executions associated with a specific alias or version.

The following example response of the ListExecutions API action shows the ARN of the alias used to start a state machine execution named *myFirstExecution*.

The *italicized* text in the following code snippet represents resource-specific information.

```
{
    "executions": [
        {
            "executionArn": "arn:aws:states:region:account-
id:execution:myStateMachine:myFirstExecution",
            "stateMachineArn": "arn:aws:states:region:account-
id:stateMachine:myStateMachine",
            "stateMachineAliasArn": "arn:aws:states:region:account-
id:stateMachine:myStateMachine:PROD",
            "name": "myFirstExecution",
            "status": "SUCCEEDED",
            "startDate": "2023-04-20T23:07:09.477000+00:00",
            "stopDate": "2023-04-20T23:07:09.732000+00:00"
        }
    ]
}
```

Using Step Functions console

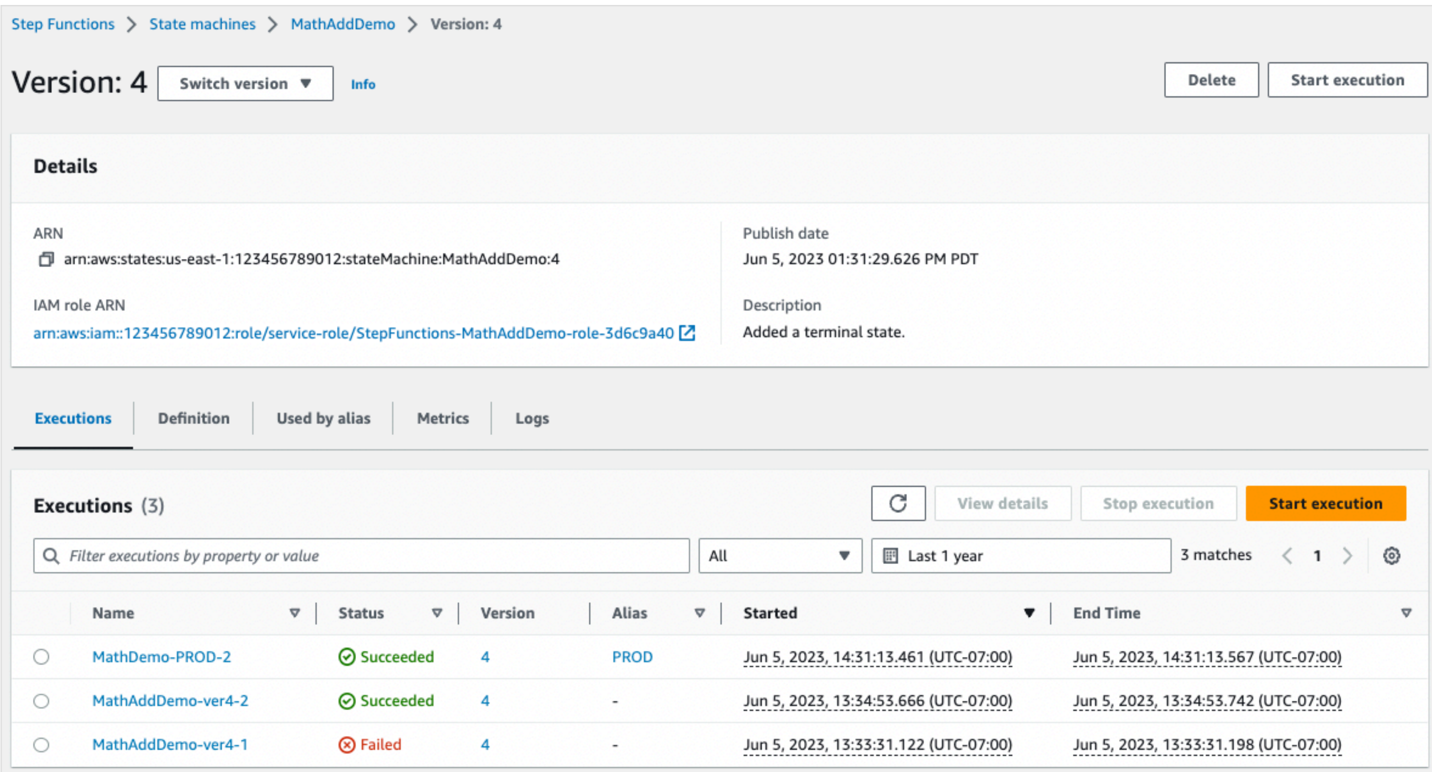
You can also view the executions started by a version or an alias from the Step Functions console. The following procedure shows how you can view the executions started with a specific version:

1. Open the Step Functions console, and then choose an existing state machine for which you've published a version or created an alias. This example shows how to view the executions started with a specific state machine version.
2. Choose the **Versions** tab, and then choose a version from the **Versions** list.

> **Tip**
>
> Filter by property or value box to search for a specific version.

3. On the *Version details page*, you can see a list of all the in-progress and past state machine executions started with the selected version.

The following image shows the *Version Details* console page. This page lists executions started by the version *4* of a state machine named *MathAddDemo*. This list also displays an execution that was started by an alias named *PROD*. This alias routed the execution traffic to version *4*.



Using CloudWatch metrics

For each state machine execution that you start with a [Qualified ARN](), Step Functions emits additional metrics with the same name and value as the metrics emitted currently. These additional metrics contain dimensions for each of the version identifier and alias name with which you start an execution. With these metrics, you can monitor state machine executions at the version level and determine when a rollback scenario might be necessary. You can also [create Amazon CloudWatch alarms]() based on these metrics.

Step Functions emits the following metrics for executions that you start with an alias or a version:

- `ExecutionTime`
- `ExecutionsAborted`
- `ExecutionsFailed`
- `ExecutionsStarted`
- `ExecutionsSucceeded`
- `ExecutionsTimedOut`

If you started the execution with a version ARN, Step Functions publishes the metric with the `StateMachineArn` and a second metric with `StateMachineArn` and `Version` dimensions.

If you started the execution with an alias ARN, Step Functions emits the following metrics:

- Two metrics for the unqualified ARN and version.
- A metric with the `StateMachineArn` and `Alias` dimensions.

Example: Alias and version deployment in Step Functions

The following example of the Canary deployment technique shows how you can deploy a new state machine version with the AWS Command Line Interface. In this example, the alias you create routes 20 percent of execution traffic to the new version. It then routes the remaining 80 percent the earlier version. To deploy a new state machine version and shift execution traffic with an alias, complete the following steps:

1. **Publish a version from the current state machine revision.**

   Use the **publish-state-machine-version** command in the AWS CLI to publish a version from the current revision of a state machine called *`myStateMachine`*:

   ```
   aws stepfunctions publish-state-machine-version --state-machine-arn
    arn:aws:states:region:account-id:stateMachine:myStateMachine
   ```

   The response returns the `stateMachineVersionArn` of the version that you published. For example, `arn:aws:states:region:account-id:stateMachine:myStateMachine:1`.

2. **Create an alias that points to the state machine version.**

Use the **create-state-machine-alias** command to create an alias named *PROD* that points to version 1 of *myStateMachine*:

```
aws stepfunctions create-state-machine-alias --name PROD --routing-
configuration "[{\"stateMachineVersionArn\":\"arn:aws:states:region:account-
id:stateMachine:myStateMachine:1\",\"weight\":100}]"
```

3. **Verify that executions started by the alias use correct published version.**

   Start a new execution of *myStateMachine* by providing the ARN of the alias **PROD** in the **start-execution** command:

```
aws stepfunctions start-execution
  --state-machine-arn arn:aws:states:region:account-
id:stateMachineAlias:myStateMachine:PROD
  --input "{}"
```

   If you provide the state machine ARN in the [StartExecution](#) request, it uses the most recent [revision](#) of the state machine instead of the version specified in your alias for starting the execution.

4. **Update the state machine definition and publish a new version.**

   Update *myStateMachine* and publish its new version. To do this, use the optional `publish` parameter of the **update-state-machine** command:

```
aws stepfunctions update-state-machine
    --state-machine-arn arn:aws:states:region:account-id:stateMachine:myStateMachine
    --definition $UPDATED_STATE_MACHINE_DEFINITION
    --publish
```

   The response returns the `stateMachineVersionArn` for the new version. For example, `arn:aws:states:region:account-id:stateMachine:myStateMachine:2`.

5. **Update the alias to point to both the versions and set the alias' [routing configuration](#).**

   Use the **update-state-machine-alias** command to update the routing configuration of the alias PROD. Configure the alias so that 80 percent of the execution traffic goes to version 1 and the remaining 20 percent goes to version 2:

```
aws stepfunctions update-state-machine-alias --state-machine-alias-arn
 arn:aws:states:region:account-id:stateMachineAlias:myStateMachine:PROD --routing-
configuration "[{\"stateMachineVersionArn\":\"arn:aws:states:region:account-
id:stateMachine:myStateMachine:1\",\"weight\":80}, {\"stateMachineVersionArn\":
\"arn:aws:states:region:account-id:stateMachine:myStateMachine:2\",\"weight\":20}]"
```

6. **Replace version 1 with version 2.**

   After you verify that your new state machine version works correctly, you can deploy the new state machine version. To do this, update the alias again to assign 100 percent of execution traffic to the new version.

   Use the **update-state-machine-alias** command to set the routing configuration of the alias PROD to 100 percent for version 2:

```
aws stepfunctions update-state-machine-alias --state-machine-alias-arn
 arn:aws:states:region:account-id:stateMachineAlias:myStateMachine:PROD --routing-
configuration "[{\"stateMachineVersionArn\":\"arn:aws:states:region:account-
id:stateMachine:myStateMachine:2\",\"weight\":100}]"
```

> **Tip**
>
> To roll back the deployment of version 2, edit the alias' routing configuration to shift 100 percent of traffic to the newly deployed version.

```
aws stepfunctions update-state-machine-alias
  --state-machine-alias-arn arn:aws:states:region:account-
id:stateMachineAlias:myStateMachine:PROD
  --routing-configuration "[{\"stateMachineVersionArn\":
\"arn:aws:states:region:account-id:stateMachine:myStateMachine:1\",\"weight
\":100}]"
```

You can use versions and aliases to perform other types of deployments. For instance, you can perform a *rolling deployment* of a new version of your state machine. To do so, gradually increase the weighted percentage in the routing configuration of the alias that points to the new version.

You can also use versions and aliases to perform a *blue/green deployment*. To do so, create an alias named `green` that runs the current version 1 of your state machine. Then, create another alias named `blue` that runs the new version, for example, *2*. To test the new version, send execution traffic to the `blue` alias. When you're confident that your new version works correctly, update the `green` alias to point to your new version.

Perform gradual deployment of state machine versions in Step Functions

A rolling deployment is a deployment strategy that slowly replaces previous versions of an application with new versions of an application. To perform a rolling deployment of a state machine version, gradually send an increasing amount of execution traffic to the new version. The amount of traffic and rate of increase are parameters that you configure.

You can perform rolling deployment of a version using one of the following options:

- Step Functions console – Create an alias that points to two versions of the same state machine. For this alias, you configure the routing configuration to shift traffic between the two versions. For more information about using the console to roll out versions, see Versions and Aliases.
- **Scripts for AWS CLI and SDK** – Create a shell script using the AWS CLI or the AWS SDK. For more information, see the following sections for using AWS CLI and AWS SDK.
- **AWS CloudFormation templates** – Use the `AWS::StepFunctions::StateMachineVersion` and `AWS::StepFunctions::StateMachineAlias` resources to publish multiple state machine versions and create an alias to point to one or two of these versions.

Use the AWS CLI to deploy a new state machine version

The example script in this section shows how you can use the AWS CLI to gradually shift traffic from a previous state machine version to a new state machine version. You can either use this example script or update it according to your requirements.

This script shows a Canary deployment for deploying a new state machine version using an alias. The following steps outline the tasks that the script performs:

1. If the `publish_revision` parameter is set to true, publish the most recent revision as the next version of the state machine. This version becomes the new, live version if the deployment succeeds.

If you set the `publish_revision` parameter to false, the script deploys the last published version of the state machine.

2. Create an alias if it doesn't exist yet. If the alias doesn't exist, point 100 percent of traffic for this alias to the new version, and then exit the script.
3. Update the routing configuration of the alias to shift a small percentage of traffic from the previous version to the new version. You set this canary percentage with the `canary_percentage` parameter.
4. By default, monitor the configurable CloudWatch alarms every 60 seconds. If any of these alarms set off, rollback the deployment immediately by pointing 100 percent of traffic to the previous version.

   After every time interval, in seconds, defined in `alarm_polling_interval`, continue monitoring the alarms. Continue monitoring until the time interval defined in `canary_interval_seconds` has passed.
5. If no alarms were set off during `canary_interval_seconds`, shift 100 percent of traffic to the new version.
6. If the new version deploys successfully, delete any versions older than the number specified in the `history_max` parameter.

```
#!/bin/bash
#
# AWS StepFunctions example showing how to create a canary deployment with a
# State Machine Alias and versions.
#
# Requirements: AWS CLI installed and credentials configured.
#
# A canary deployment deploys the new version alongside the old version, while
# routing only a small fraction of the overall traffic to the new version to
# see if there are any errors. Only once the new version has cleared a testing
# period will it start receiving 100% of traffic.
#
# For a Blue/Green or All at Once style deployment, you can set the
# canary_percentage to 100. The script will immediately shift 100% of traffic
# to the new version, but keep on monitoring the alarms (if any) during the
# canary_interval_seconds time interval. If any alarms raise during this period,
# the script will automatically rollback to the previous version.
#
```

```
# Step Functions allows you to keep a maximum of 1000 versions in version history
# for a state machine. This script has a version history deletion mechanism at
# the end, where it will delete any versions older than the limit specified.
#
# For an example that also demonstrates linear (or rolling) deployments, see the
 following:
# https://github.com/aws-samples/aws-stepfunctions-examples/blob/main/gradual-deploy/
sfndeploy.py

set -euo pipefail

# ******************************************************************************
# you can safely change the variables in this block to your values
state_machine_name="my-state-machine"
alias_name="alias-1"
region="us-east-1"

# array of cloudwatch alarms to poll during the test period.
# to disable alarm checking, set alarm_names=()
alarm_names=("alarm1" "alarm name with a space")

# true to publish the current revision as the next version before deploy.
# false to deploy the latest version from the state machine's version history.
publish_revision=true

# true to force routing configuration update even if the current routing
# for the alias does not have a 100% routing config.
# false will abandon deploy attempt if current routing config not 100% to a
# single version.
# Be careful when you combine this flag with publish_revision - if you just
# rerun the script you might deploy the newly published revision from the
# previous run.
force=false

# percentage of traffic to route to the new version during the test period
canary_percentage=10

# how many seconds the canary deployment lasts before full deploy to 100%
canary_interval_seconds=300

# how often to poll the alarms
alarm_polling_interval=60

# how many versions to keep in history. delete versions prior to this.
```

```
# set to 0 to disable old version history deletion.
history_max=0
# ******************************************************************************

######################################
# Update alias routing configuration.
#
# If you don't specify version 2 details, will only create 1 routing entry. In
# this case the routing entry weight must be 100.
#
# Globals:
#   alias_arn
# Arguments:
#   1. version 1 arn
#   2. version 1 weight
#   3. version 2 arn (optional)
#   4. version 2 weight (optional)
######################################
function update_routing() {
  if [[ $# -eq 2 ]]; then
    local routing_config="[{\"stateMachineVersionArn\": \"$1\", \"weight\":$2}]"
  elif [[ $# -eq 4 ]]; then
    local routing_config="[{\"stateMachineVersionArn\": \"$1\", \"weight\":$2},
 {\"stateMachineVersionArn\": \"$3\", \"weight\":$4}]"
  else
    echo "You have to call update_routing with either 2 or 4 input arguments." >&2
    exit 1
  fi

  ${aws} update-state-machine-alias --state-machine-alias-arn ${alias_arn} --routing-
configuration "${routing_config}"
}

# ******************************************************************************
# pre-run validation
if [[ (("${#alarm_names[@]}" -gt 0)) ]]; then
  alarm_exists_count=$(aws cloudwatch describe-alarms --alarm-names "${alarm_names[@]}"
 --alarm-types "CompositeAlarm" "MetricAlarm" --query "length([MetricAlarms,
 CompositeAlarms][])" --output text)

  if [[ (("${#alarm_names[@]}" -ne "${alarm_exists_count}")) ]]; then
    echo All of the alarms to monitor do not exist in CloudWatch: $(IFS=,; echo
 "${alarm_names[*]}") >&2
    echo Only the following alarm names exist in CloudWatch:
```

```
    aws cloudwatch describe-alarms --alarm-names "${alarm_names[@]}" --alarm-types
 "CompositeAlarm" "MetricAlarm" --query "join(', ', [MetricAlarms, CompositeAlarms]
[].AlarmName)" --output text
    exit 1
  fi
fi

if [[ (("${history_max}" -gt 0)) && (("${history_max}" -lt 2)) ]]; then
  echo The minimum value for history_max is 2. This is the minimum number of older
 state machine versions to be able to rollback in the future. >&2
  exit 1
fi
# ****************************************************************************
# main block follows

account_id=$(aws sts get-caller-identity --query Account --output text)

sm_arn="arn:aws:states:${region}:${account_id}:stateMachine:${state_machine_name}"

# the aws command we'll be invoking a lot throughout.
aws="aws stepfunctions"

# promote the latest revision to the next version
if [[ "${publish_revision}" = true ]]; then
  new_version=$(${aws} publish-state-machine-version --state-machine-arn=$sm_arn --
query stateMachineVersionArn --output text)
  echo Published the current revision of state machine as the next version with arn:
 ${new_version}
else
  new_version=$(${aws} list-state-machine-versions --state-machine-arn ${sm_arn} --max-
results 1 --query "stateMachineVersions[0].stateMachineVersionArn" --output text)
  echo "Since publish_revision is false, using the latest version from the state
 machine's version history: ${new_version}"
fi

# find the alias if it exists
alias_arn_expected="${sm_arn}:${alias_name}"
alias_arn=$(${aws} list-state-machine-aliases --state-machine-arn
 ${sm_arn} --query "stateMachineAliases[?stateMachineAliasArn==\`
${alias_arn_expected}\`].stateMachineAliasArn" --output text)

if [[ "${alias_arn_expected}" == "${alias_arn}" ]]; then
  echo Found alias ${alias_arn}
```

```
  echo Current routing configuration is:
  ${aws} describe-state-machine-alias --state-machine-alias-arn "${alias_arn}" --query
 routingConfiguration
else
  echo Alias does not exist. Creating alias ${alias_arn_expected} and routing 100%
 traffic to new version ${new_version}

  ${aws} create-state-machine-alias --name "${alias_name}" --routing-configuration
 "[{\"stateMachineVersionArn\": \"${new_version}\", \"weight\":100}]"

  echo Done!
  exit 0
fi

# find the version to which the alias currently points (the current live version)
old_version=$(${aws} describe-state-machine-alias --state-machine-alias-arn $alias_arn
 --query "routingConfiguration[?weight==\`100\`].stateMachineVersionArn" --output text)

if [[ -z "${old_version}" ]]; then
  if [[ "${force}" = true ]]; then
    echo Force setting is true. Will force update to routing config for alias to point
 100% to new version.
    update_routing "${new_version}" 100

    echo Alias ${alias_arn} now pointing 100% to ${new_version}.
    echo Done!
    exit 0
  else
    echo Alias ${alias_arn} does not have a routing config entry with 100% of the
 traffic. This means there might be a deploy in progress, so not starting another
 deploy at this time. >&2
    exit 1
  fi
fi

if [[ "${old_version}" == "${new_version}" ]]; then
  echo The alias already points to this version. No update necessary.
  exit 0
fi

echo Switching ${canary_percentage}% to new version ${new_version}
(( old_weight = 100 - ${canary_percentage} ))
update_routing "${new_version}" ${canary_percentage} "${old_version}" ${old_weight}
```

```
echo New version receiving ${canary_percentage}% of traffic.
echo Old version ${old_version} is still receiving ${old_weight}%.

if [[ ${#alarm_names[@]} -eq 0 ]]; then
  echo No alarm_names set. Skipping cloudwatch monitoring.
  echo Will sleep for ${canary_interval_seconds} seconds before routing 100% to new
 version.
  sleep ${canary_interval_seconds}
  echo Canary period complete. Switching 100% of traffic to new version...
else
  echo Checking if alarms fire for the next ${canary_interval_seconds} seconds.

  (( total_wait = canary_interval_seconds + $(date +%s) ))

  now=$(date +%s)
  while [[ ((${now} -lt ${total_wait})) ]]; do
    alarm_result=$(aws cloudwatch describe-alarms --alarm-names "${alarm_names[@]}"
 --state-value ALARM --alarm-types "CompositeAlarm" "MetricAlarm" --query "join(', ',
 [MetricAlarms, CompositeAlarms][].AlarmName)" --output text)

    if [[ ! -z "${alarm_result}" ]]; then
      echo The following alarms are in ALARM state: ${alarm_result}. Rolling back
 deploy. >&2
      update_routing "${old_version}" 100

      echo Rolled back to ${old_version}
      exit 1
    fi

    echo Monitoring alarms...no alarms have triggered.
    sleep ${alarm_polling_interval}
    now=$(date +%s)
  done

  echo No alarms detected during canary period. Switching 100% of traffic to new
 version...
fi

update_routing "${new_version}" 100

echo Version ${new_version} is now receiving 100% of traffic.

if [[ (("${history_max}" -eq 0 ))]]; then
```

```
  echo Version History deletion is disabled. Remember to prune your history, the
 default limit is 1000 versions.
  echo Done!
  exit 0
fi

echo Keep the last ${history_max} versions. Deleting any versions older than that...

# the results are sorted in descending order of the version creation time
version_history=$(${aws} list-state-machine-versions --state-
machine-arn ${sm_arn} --max-results 1000 --query "join(\`\"\\n\"\`,
 stateMachineVersions[].stateMachineVersionArn)" --output text)

counter=0

while read line; do
  ((counter=${counter} + 1))

  if [[ (( ${counter} -gt ${history_max})) ]]; then
    echo Deleting old version ${line}
    ${aws} delete-state-machine-version --state-machine-version-arn ${line}
  fi
done <<< "${version_history}"

echo Done!
```

Use the AWS SDK to deploy a new state machine version

The example script at aws-stepfunctions-examples shows how to use the AWS SDK for Python to gradually shift traffic from a previous version to a new version of a state machine. You can either use this example script or update it according to your requirements.

The script shows the following deployment strategies:

- **Canary** – Shifts traffic in two increments.

  In the first increment, a small percentage of traffic, for example, 10 percent is shifted to the new version. In the second increment, before a specified time interval in seconds gets over, the remaining traffic is shifted to the new version. The switch to the new version for the remaining traffic takes place only if no CloudWatch alarms are set off during the specified time interval.
- **Linear or Rolling** – Shifts traffic to the new version in equal increments with an equal number of seconds between each increment.

For example, if you specify the increment percent as **20** with an `--interval` of **600** seconds, this deployment increases traffic by 20 percent every 600 seconds until the new version receives 100 percent of the traffic.

This deployment immediately rolls back the new version if any CloudWatch alarms are set off.

- **All at Once or Blue/Green** – Shifts 100 percent of traffic to the new version immediately. This deployment monitors the new version and rolls it back automatically to the previous version if any CloudWatch alarms are set off.

Use AWS CloudFormation to deploy a new state machine version

The following CloudFormation template example publishes two versions of a state machine named *MyStateMachine*. It creates an alias named *PROD*, which points to both these versions, and then deploys the version 2.

In this example, 10 percent of traffic is shifted to the version 2 every five minutes until this version receives 100 percent of the traffic. This example also shows how you can set CloudWatch alarms. If any of the alarms you set go into the `ALARM` state, the deployment fails and rolls back immediately.

```
MyStateMachine:
  Type: AWS::StepFunctions::StateMachine
  Properties:
    Type: STANDARD
    StateMachineName: MyStateMachine
    RoleArn: arn:aws:iam::account-id:role/myIamRole
    Definition:
      StartAt: PassState
      States:
        PassState:
          Type: Pass
          Result: Result
          End: true

MyStateMachineVersionA:
  Type: AWS::StepFunctions::StateMachineVersion
  Properties:
    Description: Version 1
    StateMachineArn: !Ref MyStateMachine

MyStateMachineVersionB:
```

```
  Type: AWS::StepFunctions::StateMachineVersion
  Properties:
    Description: Version 2
    StateMachineArn: !Ref MyStateMachine

PROD:
  Type: AWS::StepFunctions::StateMachineAlias
  Properties:
    Name: PROD
    Description: The PROD state machine alias taking production traffic.
    DeploymentPreference:
      StateMachineVersionArn: !Ref MyStateMachineVersionB
      Type: LINEAR
      Percentage: 10
      Interval: 5
      Alarms:
        # A list of alarms that you want to monitor. If any of these alarms trigger,
 rollback the deployment immediately by pointing 100 percent of traffic to the previous
 version.
        - !Ref CloudWatchAlarm1
        - !Ref CloudWatchAlarm2
```

Handling errors in Step Functions workflows

All states, except `Pass` and `Wait` states, can encounter runtime errors. Errors can happen for various reasons, including the following:

- **State machine definition issues** - such as a Choice state without a matching rule
- **Task failures** - such as an exception in a AWS Lambda function
- **Transient issues** - such as network partition events

When a state reports an error, AWS Step Functions defaults to failing the **entire** state machine execution. Step Functions also has more advanced error handling features. You can set up your state machine to catch errors, retry failed states, and gracefully implement error handling protocols.

> **Tip**
>
> To deploy an example of a workflow that includes error handling, see Error Handling in *The AWS Step Functions Workshop*.

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**
>
> The States.ALL error type must appear alone in a Catcher and cannot catch the `States.DataLimitExceeded` terminal error or `Runtime` error types.
> For more information, see `States.DataLimitExceeded` and `States.Runtime`.

`States.DataLimitExceeded`

Reported due to the following conditions:

- 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 Step Functions service quotas.

> **Note**
>
> `DataLimitExceeded` is a terminal error which cannot be caught by the `States.ALL` error type.

`States.ExceedToleratedFailureThreshold`

A `Map` state failed because the number of failed items exceeded the threshold specified in the state machine definition. For more information, see Setting failure thresholds for Distributed Map states in Step Functions.

`States.HeartbeatTimeout`

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

> **Note**
>
> HeartbeatTimeout is only available inside the `Catch` and `Retry` fields.

`States.Http.Socket`

Occurs when an HTTP task times about after 60 seconds. See the section called "Quotas related to HTTP Task".

`States.IntrinsicFailure`

Reserved for future use. Intrinsic functions processing errors are reported with the `States.Runtime` error name.

States.ItemReaderFailed

A `Map` state failed because it couldn't read from the item source specified in the `ItemReader` field. For more information, see [ItemReader (Map)].

States.NoChoiceMatched

Reserved for future use. If no choice is matched, the error is reported with the `States.Runtime` error name.

States.ParameterPathFailure

Reserved for future use. Parameter processing errors are reported with the `States.Runtime` error name.

States.Permissions

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

States.ResultPathMatchFailure

Step Functions failed to apply a state's `ResultPath` field to the input the state received.

States.ResultWriterFailed

A `Map` state failed because it couldn't write results to the destination specified in the `ResultWriter` field. For more information, see [ResultWriter (Map)].

States.Runtime

An execution failed due to some exception that it couldn't process. 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 isn't retriable, and will always cause the execution to fail. A retry or catch on `States.ALL` won't 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.

Additionally, if a state machine runs longer than the specified `TimeoutSeconds` value, the execution fails with a `States.Timeout` error.

States can report errors with other names. However, these error names can't 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 transient Lambda service exceptions.

Note

Unhandled errors in Lambda runtimes were historically reported only as `Lambda.Unknown`. In newer runtimes, timeouts are reported as `Sandbox.Timedout` in the error output. When Lambda exceeds the maximum number of invocations, the reported error will be `Lambda.TooManyRequestsException`.
Match on `Lambda.Unknown`, `Sandbox.Timedout`, and `States.TaskFailed` to handle possible errors. You can also use `States.ALL`, but it must be alone and at the end of the list.
For more information about Lambda `Handled` and `Unhandled` errors, see `FunctionError` in the AWS Lambda Developer Guide.

Retrying after an error

`Task`, `Parallel`, and `Map` 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.

When one of these states reports an error and there's a `Retry` field, Step Functions scans through the retriers in the order listed in the array. When the error name appears in the value of a retrier's `ErrorEquals` field, the state machine makes retry attempts as defined in the `Retry` field.

If your redriven execution reruns a Task workflow state, Parallel workflow state, or Inline Map state, for which you have defined retries, the retry attempt count for these states is reset to 0 to allow for the maximum number of attempts on redrive. For a redriven execution, you can track individual retry attempts of these states using the console. For more information, see Retry behavior of redriven executions in Restarting state machine executions with redrive in Step Functions.

A retrier contains the following fields:

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

`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)

A positive integer that represents the number of seconds before the first retry attempt (1 by default). `IntervalSeconds` has a maximum value of 99999999.

`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 is never retried. `MaxAttempts` has a maximum value of 99999999.

`BackoffRate` (Optional)

The multiplier by which the retry interval denoted by `IntervalSeconds` increases after each retry attempt. By default, the `BackoffRate` value increases by `2.0`.

For example, say your `IntervalSeconds` is 3, `MaxAttempts` is 3, and `BackoffRate` is 2. The first retry attempt takes place three seconds after the error occurs. The second retry takes place six seconds after the first retry attempt. While the third retry takes place 12 seconds after the second retry attempt.

`MaxDelaySeconds` (Optional)

A positive integer that sets the maximum value, in seconds, up to which a retry interval can increase. This field is helpful to use with the `BackoffRate` field. The value you specify in this field limits the exponential wait times resulting from the backoff rate multiplier applied to each consecutive retry attempt. You must specify a value greater than 0 and less than 31622401 for `MaxDelaySeconds`.

If you don't specify this value, Step Functions doesn't limit the wait times between retry attempts.

`JitterStrategy` (Optional)

A string that determines whether or not to include jitter in the wait times between consecutive retry attempts. Jitter reduces simultaneous retry attempts by spreading these out over a randomized delay interval. This string accepts `FULL` or `NONE` as its values. The default value is `NONE`.

For example, say you have set `MaxAttempts` as 3, `IntervalSeconds` as 2, and `BackoffRate` as 2. The first retry attempt takes place two seconds after the error occurs. The second retry takes place four seconds after the first retry attempt and the third retry takes place eight seconds after the second retry attempt. If you set `JitterStrategy` as `FULL`, the first retry interval is randomized between 0 and 2 seconds, the second retry interval is randomized between 0 and 4 seconds, and the third retry interval is randomized between 0 and 8 seconds.

Retry field examples

This section includes the following `Retry` field examples.

- Retry with BackoffRate
- Retry with MaxDelaySeconds
- Retry all errors except States.Timeout
- Complex retry scenario

Example 1 – Retry with BackoffRate

The following example of a `Retry` makes two retry attempts with the first retry taking place after waiting for three seconds. Based on the `BackoffRate` you specify, Step Functions increases the interval between each retry until the maximum number of retry attempts is reached. In the following example, the second retry attempt starts after waiting for three seconds after the first retry.

```
"Retry": [ {
   "ErrorEquals": [ "States.Timeout" ],
   "IntervalSeconds": 3,
   "MaxAttempts": 2,
   "BackoffRate": 1
```

```
} ]
```

Example 2 – Retry with MaxDelaySeconds

The following example makes three retry attempts and limits the wait time resulting from `BackoffRate` at 5 seconds. The first retry takes place after waiting for three seconds. The second and third retry attempts take place after waiting for five seconds after the preceding retry attempt because of the maximum wait time limit set by `MaxDelaySeconds`.

```
"Retry": [ {
    "ErrorEquals": [ "States.Timeout" ],
    "IntervalSeconds": 3,
    "MaxAttempts": 3,
    "BackoffRate":2,
    "MaxDelaySeconds": 5,
    "JitterStrategy": "FULL"
} ]
```

Without `MaxDelaySeconds`, the second retry attempt would take place six seconds after the first retry, and the third retry attempt would take place 12 seconds after the second retry.

Example 3 – Retry all errors except States.Timeout

The reserved name `States.ALL` that appears in a retrier's `ErrorEquals` field is a wildcard that matches any error name. It must appear alone in the `ErrorEquals` array and must appear in the last retrier in the `Retry` array. The name `States.TaskFailed` also acts a wildcard and matches any error except for `States.Timeout`.

The following example of a `Retry` field retries any error except `States.Timeout`.

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

Example 4 – Complex retry scenario

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

Consider the following `Task` state.

```
"X": {
   "Type": "Task",
   "Resource": "arn:aws:states:region:123456789012:task:X",
   "Next": "Y",
   "Retry": [ {
      "ErrorEquals": [ "ErrorA", "ErrorB" ],
      "IntervalSeconds": 1,
      "BackoffRate": 2.0,
      "MaxAttempts": 2
   }, {
      "ErrorEquals": [ "ErrorC" ],
      "IntervalSeconds": 5
   } ],
   "Catch": [ {
      "ErrorEquals": [ "States.ALL" ],
      "Next": "Z"
   } ]
}
```

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

- The first two errors match the first retrier and cause waits of one and two seconds.
- The third error matches the second retrier and causes a wait of five seconds.
- The fourth error also matches the first retrier. However, it already reached its maximum of two retries (`MaxAttempts`) for that particular error. Therefore, that retrier fails and the execution redirects the workflow to the `Z` state through the `Catch` field.

Fallback states

`Task`, `Map` and `Parallel` states can each 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 that determines what input the catcher sends 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.

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

Note

Each catcher can specify multiple errors to handle.

Error output

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

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

- It takes the results of that state's execution and overwrites either all of, or a portion 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 the state's execution is the error output.

Thus, for the first catcher in the example, the catcher adds the error output to the input as a field named `error-info` if there isn't already a field with this name in the input. Then, the catcher sends the entire input to `RecoveryState`. For the second catcher, the error output overwrites the input and the catcher only sends the error output to `EndState`.

> **Note**
>
> If you don't specify the `ResultPath` field, it defaults to `$`, which selects and overwrites the entire input.

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

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 `Cause` string to JSON. The following example of a `Pass` state with intrinsic functions shows how to convert a `Cause` string to JSON.

```
"Handle escaped JSON with JSONtoString": {
  "Type": "Pass",
  "Parameters": {
```

```
    "Cause.$": "States.StringToJson($.Cause)"
  },
  "Next": "Pass State with Pass Processing"
},
```

State machine 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 `error`. In the state machine examples that follow, this Lambda function is named `FailFunction`. For information about creating a Lambda function, see [Step 1: Create a Lambda function](#) section.

```
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 `sleep10`.

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

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

Handling a failure using Retry

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

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:region:123456789012:function:FailFunction",
         "Retry": [ {
            "ErrorEquals": ["HandledError"],
            "IntervalSeconds": 1,
            "MaxAttempts": 2,
            "BackoffRate": 2.0
         } ],
      "End": true
      }
   }
}
```

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

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:region:123456789012:function:FailFunction",
         "Retry": [ {
            "ErrorEquals": ["States.TaskFailed"],
            "IntervalSeconds": 1,
            "MaxAttempts": 2,
            "BackoffRate": 2.0
         } ],
         "End": true
      }
   }
}
```

Note

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

Handling a failure using Catch

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

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:region:123456789012:function:FailFunction",
         "Catch": [ {
            "ErrorEquals": ["HandledError"],
            "Next": "fallback"
         } ],
         "End": true
      },
      "fallback": {
         "Type": "Pass",
         "Result": "Hello, AWS Step Functions!",
         "End": true
      }
   }
}
```

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

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
```

```
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:region:123456789012:function:FailFunction",
         "Catch": [ {
            "ErrorEquals": ["States.TaskFailed"],
            "Next": "fallback"
         } ],
      "End": true
      },
      "fallback": {
         "Type": "Pass",
         "Result": "Hello, AWS Step Functions!",
         "End": true
      }
   }
}
```

Handling a timeout using Retry

This state machine uses a `Retry` field to retry a `Task` state that times out, based on the timeout value specified in `TimeoutSeconds`. Step Functions retries the Lambda function invocation in this `Task` state twice, with an exponential backoff between retries.

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:region:123456789012:function:sleep10",
         "TimeoutSeconds": 2,
         "Retry": [ {
            "ErrorEquals": ["States.Timeout"],
            "IntervalSeconds": 1,
            "MaxAttempts": 2,
            "BackoffRate": 2.0
         } ],
         "End": true
      }
   }
}
```

Handling a timeout using Catch

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

```
{
   "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda
 function",
   "StartAt": "HelloWorld",
   "States": {
      "HelloWorld": {
         "Type": "Task",
         "Resource": "arn:aws:lambda:region:123456789012:function:sleep10",
         "TimeoutSeconds": 2,
         "Catch": [ {
            "ErrorEquals": ["States.Timeout"],
            "Next": "fallback"
         } ],
         "End": true
      },
      "fallback": {
         "Type": "Pass",
         "Result": "Hello, AWS Step Functions!",
         "End": true
      }
   }
}
```

Note

You can preserve the state input and the error by using `ResultPath`. See [Use ResultPath to include both error and input in a `Catch`](#).

Troubleshooting issues in Step Functions

If you encounter difficulties when working with Step Functions, use the following troubleshooting resources.

The following topics provide troubleshooting advice for errors and issues that you might encounter related to Step Functions state machines, service integrations, activities, and workflows. If you find an issue that is not listed here, you can use the **Feedback** button on this page to report it.

For more troubleshooting advice and answers to common support questions, visit the AWS Knowledge Center.

Topics

General troubleshooting

I'm unable to create a state machine.

The IAM role associated with the state machine might not have sufficient permissions. 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. For example, you could use something similar to the following:

Task A output:

```
{
    "sample": "test"
}
```

Task B:

```
{
    "JsonPathSample.$": "$.sample"
}
```

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 `ExecutionThrottled` metric in the Execution metrics 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 in each AWS Region. If you exceed this limit, Step Functions throws an `ExecutionLimitExceeded` error. This limit does not apply to Express Workflows. You can use the `OpenExecutionCount` to track when you are approaching the `OpenExecutionLimit` and create alarms to proactively notify you in that event. `OpenExecutionCount` is an approximate number of open workflows. For more information, see Execution metrics.

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 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 .sync.

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.

Accessing the Lambda function with cross-account support

If cross-account access of AWS resources is available in your Region, use the following method to invoke a Lambda function from another account.

To invoke a cross-account resource in your workflows, do the following:

1. Create an IAM role in the target account that contains the resource. This role grants the source account, containing the state machine, permissions to access the target account's resources.
2. In the `Task` state's definition, specify the target IAM role to be assumed by the state machine before invoking the cross-account resource.
3. Modify the trust policy in the target IAM role to allow the source account to assume this role temporarily. The trust policy must include the Amazon Resource Name (ARN) of the state machine defined in the source account. Also, define the appropriate permissions in the target IAM role to call the AWS resource.
4. Update the source account's execution role to include the required permission for assuming the target IAM role.

For an example, see Accessing cross-account AWS resources in Step Functions in the tutorials.

Note

You can configure your state machine to assume an IAM role for accessing resources from multiple AWS accounts. However, a state machine can assume only one IAM role at a given time.

For an example of a `Task` state definition that specifies a cross-account resource, see Task state's Credentials field examples.

Accessing the Lambda function without cross-account support

If cross-account access of AWS resources is unavailable in your Region, use the following method to 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`.

```
{
   "StartAt":"CallLambda",
   "States":{
      "CallLambda":{
         "Type":"Task",
         "Resource":"arn:aws:states:::lambda:invoke",
         "Parameters":{
            "FunctionName":"arn:aws:lambda:region:account-id:function:my-function"
         },
         "End":true
      }
   }
}
```

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.

```
{
   "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 Using timeouts to avoid stuck Step Functions workflow executions.

If your state machine is stuck in the ActivityScheduled event, it indicates that your activity worker fleet has issues or is under-scaled. You should monitor the `ActivityScheduleTime` CloudWatch metric and set an alarm when that time increases. However, to time out any stuck state machine executions in which the `Activity` state doesn't transition to the `ActivityStarted` state, define a timeout at state machine-level. To do this, specify a `TimeoutSeconds` field at the beginning of the state machine definition, outside of the `States` field.

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 use to make the API call. To receive a response, the timeout must have a value higher than the duration of the Express Workflow executions.

I'm unable to see the execution history in order to troubleshoot Express Workflow failures.

Express Workflows don't record execution history in AWS Step Functions. Instead, you must turn on CloudWatch logging. After logging is turned on, you can use CloudWatch Logs Insights queries to review your Express Workflow executions. You can also view execution history for Express Workflow executions on the Step Functions console if you choose the **Enable** button in the **Executions** tab. For more information, see Viewing execution details in the Step Functions console.

To list executions based on duration:

```
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  by
 execution_arn
| sort duration_in_ms desc
```

To list failed and cancelled executions:

```
fields ispresent(execution_arn) as isRes | filter type in ["ExecutionFailed",
 "ExecutionAborted", "ExecutionTimedOut"]
```

Best practices for Step Functions

Managing state and transforming data

Learn about Passing data between states with variables and Transforming data with JSONata.

The following topics are best practices to help you manage and optimize your Step Functions workflows.

List of best practices

- Optimizing costs using Express Workflows
- Tagging state machines and activities in Step Functions
- Using timeouts to avoid stuck Step Functions workflow executions
- Using Amazon S3 ARNs instead of passing large payloads in Step Functions
- Starting new executions to avoid reaching the history quota in Step Functions
- Handle transient Lambda service exceptions
- Avoiding latency when polling for activity tasks
- CloudWatch Logs resource policy size limits

Optimizing costs using Express Workflows

Step Functions determines pricing for Standard and Express workflows based on the workflow type you use to build your state machines. To optimize the cost of your serverless workflows, you can follow either or both of the following recommendations:

For information about how choosing a Standard or Express workflow type affects billing, see AWS Step Functions Pricing.

Nest Express workflows inside Standard workflows

Step Functions runs workflows that have a finite duration and number of steps. Some workflows may complete execution within a short period of time. Others may require a combination of both long-running and high-event-rate workflows. With Step Functions, you can build large, complex workflows out of multiple smaller, simpler workflows.

For example, to build an order processing workflow, you can include all non-idempotent actions into a Standard workflow. This could include actions, such as approving order through human interaction and processing payments. You can then combine a series of idempotent actions, such as sending payment notifications and updating product inventory, in an Express workflow. You can nest this Express workflow within the Standard workflow. In this example, the Standard workflow is known as the *parent state machine*. The nested Express workflow is known as a *child state machine*.

Migrate Standard workflows to Express workflows

You should consider migrating your Standard workflows to Express workflows if they meet the following requirements:

- Your workflow must complete execution within five minutes.
- Your workflow conforms to an *at-least-once* execution model, which means each step in the workflow may run more than exactly once.
- Your workflow does **not** use the `.waitForTaskToken` or `.sync` service integration patterns.

> **Important**
>
> Express workflows use Amazon CloudWatch Logs to record execution histories. You will incur additional costs when using CloudWatch Logs.

To migrate a Standard workflow to an Express workflow using the console

1. Open the Step Functions console.
2. On the **State machines** page, choose a Standard type state machine to open it.

   > **Tip**
   >
   > From the **Any type** dropdown list, choose **Standard** to filter the state machines list and view only Standard workflows.

3. Choose **Copy to new**.

Workflow Studio opens in Design mode displaying workflow of the state machine you selected.

4. (Optional) Update the workflow design.
5. Specify a name for your state machine. To do this, choose the edit icon next to the default state machine name of **MyStateMachine**. Then, in **State machine configuration**, specify a name in the **State machine name** box.
6. (Optional) In **State machine configuration**, specify other workflow settings, such as state machine type and its execution role.

   Make sure that for **Type**, you choose **Express**. Keep all the other default selections on **State machine settings**.

   > **Note**
   >
   > If you're migrating a Standard workflow previously defined in AWS CDK or AWS SAM, you must change the value of `Type` and `Resource` name.

7. In the **Confirm role creation** dialog box, choose **Confirm** to continue.

   You can also choose **View role settings** to go back to **State machine configuration**.

   > **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.

For more information about best practices and guidelines when you manage cost-optimization for your workflows, see Building cost-effective AWS Step Functions workflows.

Tagging state machines and activities in Step Functions

AWS Step Functions supports tagging state machines (both Standard and Express) and activities. Tags can help you track and manage your resources and provide better security in your AWS Identity and Access Management (IAM) policies. After tagging Step Functions resources, you can manage them with AWS Resource Groups. To learn how, see the AWS Resource Groups User Guide.

For tag-based authorization, state machine execution resources as shown in the following example inherit the tags associated with a state machine.

```
arn:partition:states:region:account-id:execution:<StateMachineName>:<ExecutionId>
```

When you call DescribeExecution or other APIs in which you specify the execution resource ARN, Step Functions uses tags associated with the state machine to accept or deny the request while performing tag-based authorization. This helps you allow or deny access to state machine executions at the state machine level.

To review the restrictions related to resource tagging, see Restrictions related to tagging.

Tagging for Cost Allocation

You can use cost allocation tags to identify the purpose of a state machine and reflect that organization in your AWS bill. Sign up to get your AWS account bill to include the tag keys and values. See Setting Up a Monthly Cost Allocation Report in the *AWS Billing User Guide* for details on setting up reports.

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

Resource	Key	Value
`StateMachine1`	`Cost Center`	`34567`
	`Application`	`Image processing`
`StateMachine2`	`Cost Center`	`34567`
	`Application`	`Rekognition processing`

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`.

```
{
    "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.

Managing tags in the Step Functions console

You can view and manage tags for your state machines in the Step Functions console. From the **Details** page of a state machine, select **Tags**.

Managing tags with Step Functions API Actions

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

- `ListTagsForResource`
- `TagResource`
- `UntagResource`

Using timeouts to avoid stuck Step Functions workflow 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:

```
"ActivityState": {
  "Type": "Task",
  "Resource": "arn:aws:states:region:account-id:activity:HelloWorld",
  "TimeoutSeconds": 300,
  "Next": "NextState"
}
```

If you use a [callback with a task token (.waitForTaskToken)](), we recommend that you use heartbeats and add the `HeartbeatSeconds` field in your `Task` state definition. You can set `HeartbeatSeconds` to be less than the task timeout so if your workflow fails with a heartbeat error then you know it's because of the task failure instead of the task taking a long time to complete.

```
{
  "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/account-id/push-based-queue"
      },
      "ResultPath": "$.SQS",
      "End": true
    }
  }
```

```
}
```

For more information, see Task workflow state 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 in Amazon States Language for Step Functions workflows.

Using Amazon S3 ARNs instead of passing large payloads in Step Functions

Executions that pass large payloads of data between states can be terminated. If the data you are passing between states might grow to over 256 KiB, 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.

In the following example, a state machine passes input to an AWS Lambda function, which processes a JSON file in an Amazon S3 bucket. After you run this state machine, the Lambda function reads the contents of the JSON file, and returns the file contents as output.

Create the Lambda function

The following Lambda function named *`pass-large-payload`* reads the contents of a JSON file stored in a specific Amazon S3 bucket.

Note

After you create this Lambda function, make sure you provide its IAM role the appropriate permission to read from an Amazon S3 bucket. For example, attach the **AmazonS3ReadOnlyAccess** permission to the Lambda function's role.

```
import json
import boto3
import io
```

```
import os

s3 = boto3.client('s3')

def lambda_handler(event, context):
    event = event['Input']
    final_json = str()

    s3 = boto3.resource('s3')
    bucket = event['bucket'].split(':')[-1]
    filename = event['key']
    directory = "/tmp/{}".format(filename)

    s3.Bucket(bucket).download_file(filename, directory)

    with open(directory, "r") as jsonfile:

        final_json = json.load(jsonfile)

    os.popen("rm -rf /tmp")

    return final_json
```

Create the state machine

The following state machine invokes the Lambda function you previously created.

```
{
   "StartAt":"Invoke Lambda function",
   "States":{
      "Invoke Lambda function":{
         "Type":"Task",
         "Resource":"arn:aws:states:::lambda:invoke",
         "Parameters":{
            "FunctionName":"arn:aws:lambda:us-east-2:123456789012:function:pass-large-
payload",
            "Payload":{
               "Input.$":"$"
            }
         },
         "OutputPath": "$.Payload",
         "End":true
      }
   }
```

```
}
```

Rather than pass a large amount of data in the input, you could save that data in an Amazon S3 bucket, and pass the Amazon Resource Name (ARN) of the bucket in the `Payload` parameter to get the bucket name and key value. Your Lambda function can then use that ARN to access the data directly. The following is example input for the state machine execution, where the data is stored in `data.json` in an Amazon S3 bucket named *amzn-s3-demo-large-payload-json*.

```
{
  "key": "data.json",
  "bucket": "arn:aws:s3:::amzn-s3-demo-large-payload-json"
}
```

Starting new executions to avoid reaching the history quota in Step Functions

AWS Step Functions has a hard quota of 25,000 entries in the execution event history. When an execution reaches 24,999 events, it waits for the next event to happen.

- If the event number 25,000 is `ExecutionSucceeded`, the execution finishes successfully.
- If the event number 25,000 isn't `ExecutionSucceeded`, the `ExecutionFailed` event is logged and the state machine execution fails because of reaching the history limit

To avoid reaching this quota for long-running executions, you can try one of the following workarounds:

- Use the Map state in Distributed mode. In this mode, the `Map` state runs each iteration as a child workflow execution, which enables high concurrency of up to 10,000 parallel child workflow executions. Each child workflow execution has its own, separate execution history from that of the parent workflow.
- Start a new state machine execution directly from the `Task` state of a running execution. To start such nested workflow executions, use Step Functions' `StartExecution` API action in the parent state machine along with the necessary parameters. For more information about using nested workflows, see Start workflow executions from a task state in Step Functions or Using a Step Functions API action to continue a new execution tutorial.

> **Tip**
>
> To deploy an example nested workflow, see Optimizing costs in *The AWS Step Functions Workshop*.

- 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 Using a Lambda function to continue a new execution in Step Functions tutorial.

Handle transient Lambda service exceptions

AWS Lambda can occasionally experience transient service errors. In this case, invoking Lambda results in a 500 error, such as `ClientExecutionTimeoutException`, `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.ClientExecutionTimeoutException",
 "Lambda.ServiceException", "Lambda.AWSLambdaException", "Lambda.SdkClientException"],
   "IntervalSeconds": 2,
   "MaxAttempts": 6,
   "BackoffRate": 2
} ]
```

> **Note**
>
> Unhandled errors in Lambda runtimes were historically reported only as `Lambda.Unknown`. In newer runtimes, timeouts are reported as `Sandbox.Timedout` in the error output. When Lambda exceeds the maximum number of invocations, the reported error will be `Lambda.TooManyRequestsException`.
> Match on `Lambda.Unknown`, `Sandbox.Timedout`, and `States.TaskFailed` to handle possible errors. You can also use `States.ALL`, but it must be alone and at the end of the list.

For more information about Lambda `Handled` and `Unhandled` errors, see `FunctionError` in the AWS Lambda Developer Guide.

For more information, see the following:

- Retrying after an error
- Handling error conditions in a Step Functions state machine
- Lambda Invoke Errors

Avoiding 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.

For more information on activities and activity workers see Learn about Activities in Step Functions.

CloudWatch Logs resource policy size limits

When you create a state machine with logging, or update an existing state machine to enable logging, Step Functions must update your CloudWatch Logs resource policy with the log group that you specify. CloudWatch Logs resource policies are limited to 5,120 characters.

When CloudWatch Logs detects that a policy approaches the size limit, CloudWatch Logs automatically enables logging for log groups that start with `/aws/vendedlogs/`.

You can prefix your CloudWatch Logs log group names with `/aws/vendedlogs/` to avoid the CloudWatch Logs resource policy size limit. If you create a log group in the Step Functions console, the suggested log group name will already be prefixed with `/aws/vendedlogs/states`.

CloudWatch Logs also has a quota of 10 resource policies per region, per account. If you try to enable logging on a state machine that already has 10 CloudWatch Logs resource policies in a region for an account, the state machine will not be created or updated. For more information about logging quotes, see CloudWatch Logs quotas.

If you are having trouble sending logs to CloudWatch Logs, see Troubleshooting state machine logging to CloudWatch Logs. To learn more about logging in general, see Enable logging from AWS services.

Step Functions service quotas

AWS Step Functions provide default service quotas for state machine parameters, such as the number of API actions during a time period or the number of state machines that you can define. Quotas are designed to prevent misconfigured state machine from consuming all of the resources of the system, although many do not have hard limits.

To request a service quota increase, you can do one of the following:

- Use the Service Quotas console at https://console.aws.amazon.com/servicequotas/home. For information about requesting a quota increase using the Service Quotas console, see Requesting a quota increase in the *Service Quotas User Guide*.
- 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

Names of state machines, executions, and activity tasks must not exceed 80 characters in length. These names 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 accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such characters will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.

Quotas related to accounts

Resource	Default quota	Can be increased to
Maximum number of registered state machines	100,000	150,000
Maximum number of registered activities	100,000	150,000
Maximum size of state machine definition	1 MB	Hard quota
Maximum request size	1 MB per request. This is the total data size per Step Functions API request, including the request header and all other associated request data.	Hard quota

Resource	Default quota	Can be increased to
Maximum open executions per account	1,000,000 executions for each AWS account in each AWS Region. Exceeding this limit will cause an `ExecutionLimitExceeded` error. This doesn't apply to Express Workflows.	*Millions*
Maximum number of open Map Runs	1000 This quota applies to *Distributed Map state*. An open Map Run is a Map Run that has started, but hasn't yet completed. Scheduled Map Runs wait at the MapRunStarted event until the total number of open Map Runs is less than the quota.	Hard quota
Maximum redrives of a Map Run.	1000 This quota applies to *Distributed Map state*.	Hard quota
Maximum number of parallel Map Run child executions	10,000	Hard quota

Quotas related to HTTP Task

HTTP Tasks are throttled using a token bucket scheme to maintain the Step Functions service bandwidth.

Resource	Bucket size	Refill rate per second
HTTP Task	300	300

Resource	Default quota
HTTP Task duration — time to send an HTTP request and receive a response	60 seconds (Hard quota)

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.

	Standard		Express	
Service metric	**Bucket size**	**Refill rate per second**	**Bucket size**	**Refill rate per second**
`StateTransition` — *US East (N. Virginia), US West (Oregon), and Europe (Ireland)*	5,000	5,000	Unlimited	Unlimited

	Standard		Express	
Service metric	Bucket size	Refill rate per second	Bucket size	Refill rate per second
`StateTransition` — *All other regions*	800	800	Unlimited	Unlimited

Quotas related to API action throttling

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

	Standard		Express	
API name	Bucket size	Refill rate per second	Bucket size	Refill rate per second
`StartExecution` — *US East (N. Virginia), US West (Oregon), and Europe (Ireland)*	1,300	300	6,000	6,000
`StartExecution` — *All other regions*	800	150	6,000	6,000

Quota related to TestState API

API name	Quota	Can be increased to
TestState	1 transaction per second (TPS)	Hard quota

Other quotas

The following are soft quotas and can be increased.

	US East (N. Virginia), US West (Oregon), and Europe (Ireland)		All other regions	
API name	Bucket size	Refill rate per second	Bucket size	Refill rate per second
`CreateActivity`	100	1	100	1
`CreateStateMachine`	100	1	100	1
`CreateStateMachineAlias`	100	1	100	1
`DeleteActivity`	100	1	100	1
`DeleteStateMachine`	100	1	100	1
`DeleteStateMachineAlias`	100	1	100	1

	US East (N. Virginia), US West (Oregon), and Europe (Ireland)		All other regions	
API name	**Bucket size**	**Refill rate per second**	**Bucket size**	**Refill rate per second**
`DeleteStateMachineVersion`	100	1	100	1
`DescribeActivity`	200	1	200	1
`DescribeExecution`	300	15	250	10
`DescribeMapRun`	200	1	200	1
`DescribeStateMachine`	200	20	200	20
`DescribeStateMachineAlias`	200	1	200	1
`DescribeStateMachineForExecution`	200	1	200	1
`GetActivityTask`	3,000	500	1,500	300
`GetExecutionHistory`	400	20	400	20

	US East (N. Virginia), US West (Oregon), and Europe (Ireland)		All other regions	
API name	Bucket size	Refill rate per second	Bucket size	Refill rate per second
`ListActivities`	100	10	100	5
`ListExecutions`	200	5	100	2
`ListMapRuns`	100	1	100	1
`ListStateMachineAliases`	100	1	100	1
`ListStateMachines`	100	5	100	5
`ListStateMachineVersions`	100	1	100	1
`ListTagsForResource`	100	1	100	1
`PublishStateMachineVersion`	100	1	100	1
`RedriveExecution`	1,300	300	800	150
`SendTaskFailure`	3,000	500	1,500	300

	US East (N. Virginia), US West (Oregon), and Europe (Ireland)		All other regions	
API name	**Bucket size**	**Refill rate per second**	**Bucket size**	**Refill rate per second**
`SendTaskHeartbeat`	3,000	500	1,500	300
`SendTaskSuccess`	3,000	500	1,500	300
StartSyncExecution	Synchronous Express execution API calls don't contribute to existing account capacity limits. Step Functions provides capacity on demand and automatically scales with sustained workload. Surges in workload may be throttled until capacity is available. If you experience throttling, try again after some time. For information about Synchronous Express workflows, see Synchronous and Asynchronous Express Workflows in Step Functions.			
`StopExecution`	1,000	200	500	25
`TagResource`	200	1	200	1
`UntagResource`	200	1	200	1
`UpdateMapRun`	100	1	100	1
`UpdateStateMachine`	100	1	100	1
`UpdateStateMachineAlias`	100	1	100	1

	US East (N. Virginia), US West (Oregon), and Europe (Ireland)		All other regions	
API name	**Bucket size**	**Refill rate per second**	**Bucket size**	**Refill rate per second**
`ValidateStateMachineDefinition`	100	1	100	1

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, except for the *Execution history retention time* quota.

Quota	Standard	Express
Maximum execution time	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.	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.
Maximum execution history size	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 Starting new executions to avoid reaching the history quota in Step Functions.	Unlimited.
Maximum execution idle time	1 year	5 minutes

Quota	Standard	Express
	Constrained by maximum execution time.	Constrained by maximum execution time.
Execution history retention time	90 days after an execution is closed. After this time, you can no longer retrieve or view the execution history. There is no further quota for the number of closed executions that Step Functions retains. To meet compliance, organizational, or regulatory requirements, you can reduce the execution history retention period to 30 days by sending a quota request. To do this, use the AWS Support Center Console and create a new case. The change to reduce the retention period to 30 days is applicable for each account in a Region.	To see execution history, Amazon CloudWatch Logs logging must be configured. For more information, see Using CloudWatch Logs to log execution history in Step Functions.

Quota	Standard	Express
Execution redrivable period	14 days Hard quota applies to *Distributed Map state*. Redrivable period refers to the time during which you can redrive a given Standard Workflow execution. This period starts from the day a state machine completes its execution.	Redrive is not supported for Express workflows.

Quotas related to task executions

The following table describes quotas related to task executions. These are all **hard quotas** that cannot be changed.

Quota	Standard	Express
Maximum task execution time	1 year — Constrained by maximum execution time.	5 minutes — Constrained by maximum execution time.
Maximum time Step Functions keeps a task in the queue	1 year — Constrained by maximum execution time.	5 minutes — Constrained by maximum execution time.
Maximum activity pollers per Amazon Resource Name (ARN)	1,000 pollers calling `GetActivityTask` per ARN. Exceeding this quota results in this error: *"The maximum number of workers concurrently polling for activity tasks has been reached."*	Does not apply to Express Workflows.

Quota	Standard	Express
Maximum input or output size for a task, state, or execution	256 KiB 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 .	256 KiB 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 .

Quotas related to versions and aliases

Resource	Default quota
Maximum number of published state machine **versions**	1000 per state machine
Maximum number of state machine **aliases**	100 per state machine

To request an increase to soft limits for published state machine versions and aliases, use the **Support Center** page in the AWS Management Console.

Restrictions related to tagging

The following tagging restrictions can **not** be modified or increased.

- **Prefix restriction** — Do not use the `aws:` prefix in your tag names or values because it is reserved for AWS use only. You cannot edit or delete tag names or values with an `aws:` prefix. Tags with the `aws:` prefix do not count against your tags per resource quota.
- **Character restrictions** — Tags may only contain Unicode letters, digits, whitespace, or the following symbols: `_ . : / = + - @`

Restriction	Description
Maximum number of tags per resource	50
Maximum key length	128 Unicode characters in UTF-8
Maximum value length	256 Unicode characters in UTF-8

Recent feature launches

The following table lists dates and links to announcements for recent Step Functions feature releases:

Launch date	Feature description
2025-02-07	AWS Step Functions expands data source and output options for Distributed Map.
2024-11-22	Simplifying developer experience with variables and JSONata in AWS Step Functions
2024-11-14	IaC exports to AWS SAM templates, CloudFormation templates, and to Infrastructure Composer.
2024-06-25	Encrypt workflows, logs, and activities with AWS KMS customer managed keys.
2023-11-26	Invoke HTTPS endpoints and test states with TestState API.
2023-11-15	Restarting state machine executions with redrive in Step Functions
2023-10-12	Add optimized integration for Amazon EMR Serverless
2023-09-07	Enhance error handling
2023-08-31	Workflow Studio enhancements to streamline the authoring experience
2023-06-22	Versions and aliases
2023-06-16	Add seven AWS SDK integrations, including VPC Lattice
2022-12-01	Orchestrate large-scale parallel workflows for data processing with *Distributed Map state*

Document history

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

Change	Description	Date changed
Documentation-only update	Simplified and re-organized the integrating services topics based on customer feedback. To see the updates, check out the *Integrating services* chapter.	Sep 4, 2025
Updates	Step Functions will now auto-create roles and policy for optimized integrations with MediaConvert. For integrations with MediaConvert, Step Functions will now automatically create the necessary roles and policies required by your state machine. To learn more, see the section called “AWS Elemental MediaConvert” and *Integrating optimized services*.	March 14, 2025
New feature	Step Functions expands data source and output options for Distributed Map. Distributed map can process data from JSON Lines (JSONL) and a broader range of delimited file formats, such as semicolon-delimited files and tab-delimited files. Additionally, Distributed Map offers output transform ations for greater control over result formatting. To learn more, ItemReader (Map) and ResultWriter (Map).	February 7, 2025
Documentation-only update	Replaced the Getting started tutorial with content from workshop presented at re:Invent 2024.	Dec 23, 2024
New feature	Manage state and transform data with Step Functions workflow variables and JSONata. With variables, you can pass data between the steps of your workflows. With JSONata, you gain an open source	November 22, 2024

Change	Description	Date changed
	query and expression language to select and transform data in your workflows. To learn more, see Passing data between states with variables and Transforming data with JSONata in Step Functions.	
New feature	Step Functions adds Infrastructure as Code (IaC) template generation The AWS Step Functions console provides the ability to export and download saved workflows as AWS CloudFormation or AWS SAM (SAM) templates. For AWS Regions that support AWS Infrastructure Composer, it additionally provides the ability to export your workflows to Infrastructure Composer and navigates to the Infrastructure Composer console, where you can continue to work with the newly generated template. To learn more, see Exporting your workflow to IaC templates.	November 14, 2024
New feature	Step Functions adds the option to use AWS KMS and customer managed keys to encrypt your data You can add another layer of security by choosing a customer managed key to encrypt workflows, activities, and logs. To learn more, see Data at rest encryption in Step Functions.	July 25, 2024
Updates	Document structure update With page view data and depth analysis, documentation sections were restructured to increase visibility of important topics. The navigation was updated to reduce overall depth. Related topics were consolidated. Redirects were added so that bookmarks should lead to the updated locations. Send feedback if you notice errors or omissions after this massive update. Thank you!	July 24, 2024

Change	Description	Date changed
Updates	AWS managed policy updates - new permission: `states:ValidateStateMachineDefinition` Added information about new permission to check the syntax of a state machine that you provide. To learn more, see AWS managed policies for AWS Step Functions.	April 29, 2024
New feature	Step Functions adds optimized integration for AWS Elemental MediaConvert AWS Elemental MediaConvert provides broadcast-grade video and audio file transcoding, which customers can automate with code to suit their media workflows. With the optimized integration for AWS Step Functions in MediaConvert, it is now possible to orchestrate using the low-code visual tool Workflow Studio. To learn more, see the documentation to Manage AWS Elemental MediaConvert with Step Functions.	April 12, 2024
Updates	AWS managed policy updates - Update to an existing policy: `AWSStepFunctionsReadOnlyAccess` Added information about new read-only permissions for tags, distributed maps, and versions and aliases. To learn more, see AWS managed policies for AWS Step Functions.	April 02, 2024

Change	Description	Date changed
Updates	Step Functions adds support for Open Workflow metrics With open workflow metrics, you now have account-l evel visibility into the number of standard workflows in progress as well as your open workflow limit. You can manage workloads across all workflows, regardles s of how they're started, to ensure smooth workflow operations. You can set CloudWatch alarms to monitor your workflows and proactively receive alerts as you approach your limits. Once alerted, you can effective ly manage your workflows by taking actions such as stopping specific workflows or requesting a limit increase. Open workflow metrics is available to use in CloudWatch for standard workflows with no additional configuration required. To learn more, see Execution metrics.	February 29, 2024
Updates	Service integration additions and updates. For the list of new and updated AWS SDK integrations, see Learning to use AWS service SDK integrations in Step Functions. For the full list of services, see Supported AWS SDK service integrations.	January 18, 2024
New feature	Use Workflow Studio in Infrastructure Composer to build serverless workflows using AWS CloudFormation templates. For more information, see Using Workflow Studio in Infrastructure Composer to build Step Functions workflows.	November 27, 2023
New feature	Step Functions now lets you directly invoke public HTTPS endpoints and test individual states using a new Test State API. For more information, see: • Call HTTPS APIs in Step Functions workflows • Using TestState API to test a state in Step Functions	November 26, 2023

Change	Description	Date changed
New feature	Step Functions now integrates with Amazon Bedrock. For more information, see the following topics: • Invoke and customize Amazon Bedrock models with Step Functions • IAM permissions for Amazon Bedrock • Perform AI prompt-chaining with Amazon Bedrock • Integrating services with Step Functions	November 26, 2023
New feature	Step Functions now lets you redrive workflow execution s of type Standard from their point of failure. For more information, see Restarting state machine executions with redrive in Step Functions and Redriving Map Runs in Step Functions executions.	November 15, 2023
Documentation-only update	Published a new topic that explains how to run state machines on a schedule using Amazon EventBridge Scheduler. For more information, see Using Amazon EventBridge Scheduler to start a Step Functions state machine execution.	October 16, 2023
New feature	Step Functions now integrates with Amazon EMR Serverless. For more information, see the following topics: • Create and manage Amazon EMR Serverless applicati ons with Step Functions • Run an EMR Serverless job • Integrating services with Step Functions • Integrating services with Step Functions	October 12, 2023
Documentation-only update	Added information about running state machines on a schedule using Amazon EventBridge Scheduler. For more information, see Using EventBridge Scheduler.	October 05, 2023

Change	Description	Date changed
Update	Reorganized and updated the *Distributed Map state* topics for clarity, brevity, and establishing a clear journey map for new users. For more information, see Using Map state in Distributed mode for large-scale parallel workloads in Step Functions.	October 6, 2023
Fixes	Fixed code samples in a tutorial to use AWS CDK v2. For more information, see Using AWS CDK to create a Standard workflow in Step Functions.	September 19, 2023
Update	Added information about the enhanced error handling capabilities that Step Functions has introduced to identify errors clearly and implement retries with greater control. For more information, see Fail workflow state and Retrying after an error.	September 07, 2023
Update	Step Functions has added enhancements to Workflow Studio for streamlining workflow authoring experience. For more information, see Developing workflows in Step Functions Workflow Studio.	August 31, 2023
Documentation-only update	Added information about twice the actual metric count reported for the `ExecutionsStarted` metric. For more information, see Metrics that report a count.	July 25, 2023
Documentation-only update	Step Functions has added two new sample projects that demonstrate the following common use cases for the *Distributed Map state*: • Processing a CSV file • Processing data in an Amazon S3 bucket	July 17, 2023
Documentation-only update	Published a new topic about deploying state machines using Terraform. For more information, see Using Terraform to deploy state machines in Step Functions.	July 5, 2023

Change	Description	Date changed
Documentation-only update	Updated the following procedures to match changes to the Amazon EventBridge interface. • Automate event delivery • Starting a Step Functions workflow in response to events	June 26, 2023
New feature	Step Functions now provides the ability to create multiple state machine versions and aliases for improved resiliency while deploying serverless workflows. For more informati on, see Manage continuous deployments with versions and aliases in Step Functions.	June 22, 2023
Documentation-only update	Improved the description of `TimeoutSeconds` and `HeartbeatSeconds` fields to describe how they're different from each other. For more information, see Task state fields.	June 22, 2023
Documentation-only update	Published a new section that describes how to flatten an array of arrays typically returned as result for Parallel and Map states. For more information, see Flattening an array of arrays.	June 20, 2023
Update	Step Functions has expanded support for AWS SDK integrations by adding seven AWS services and 468 new API actions. For more information, see Supported AWS SDK service integrations and Learning to use AWS service SDK integrations in Step Functions.	June 16, 2023
Documentation-only update	Published a new topic that lists the AWS Regions in which recently launched Step Functions features are available. For more information, see Recent feature launches.	June 16, 2023

Change	Description	Date changed
Documentation-only update	Step Functions now includes a section about AWS User Notifications, an AWS service that acts as a central location for your AWS notifications in the AWS Management Console. For more information, see Events using User Notifications.	May 4, 2023
Documentation-only update	Added a new section that explains about the permissio ns needed to write child workflow execution results to an Amazon S3 bucket encrypted with an AWS Key Management Service (AWS KMS) key. For more informati on, see IAM permissions for AWS KMS key encrypted Amazon S3 bucket.	April 29, 2023
Documentation-only update	Added a new topic that explains about the Data flow simulator feature. For more information, see Data flow simulator (unsupported).	April 14, 2023
Quota update	Added information about default quota of 1000 for open Map Runs in each account. For more information, see Quotas related to accounts.	April 05, 2023
Documentation-only update	Added a Note about unavailability of X-Ray tracing for the *Distributed Map state*. For more information, see Trace Step Functions request data in AWS X-Ray.	March 21, 2023
Documentation-only update	Added information about how Step Functions handles tag-based authorization. For more information, see Tagging state machines and activities in Step Functions and Creating tag-based IAM policies in Step Functions.	March 15, 2023
Documentation-only update	Added information about how Step Functions parses CSV files used as input in *Distributed Map state*. For more information, see CSV file in an Amazon S3 bucket.	March 14, 2023

Change	Description	Date changed
Documentation-only update	Added information about how Step Functions handles cross-account invocations for the Run a Job (.sync) pattern. For more information, see Run a Job (.sync).	March 01, 2023
Documentation-only update	Reduce the history retention period of your completed workflow executions from 90 days to 30 days. For more information about adjusting the retention period, see Execution guarantees in Step Functions workflows and Quotas related to state machine executions.	February 21, 2023
Update	Step Functions has expanded support for AWS SDK integrations by adding 35 AWS services and 1100 new API actions. For more information, see Supported AWS SDK service integrations and Learning to use AWS service SDK integrations in Step Functions.	February 17, 2023
Documentation-only update	Published a Getting Started tutorial series that walks you through the process of creating a workflow for credit card application using Step Functions. For more information, see Learn how to get started with Step Functions.	December 30, 2022
New feature	Step Functions adds support to orchestrate large-scale parallel workflows for data processing using a new Distributed mode for `Map` state. For more information, see Using Map state in Distributed mode for large-scale parallel workloads in Step Functions.	December 01, 2022

Change	Description	Date changed
New feature	Step Functions now supports access to cross-account AWS resources configured in other accounts. For more information, see • Accessing resources in other AWS accounts in Step Functions • Accessing cross-account AWS resources in Step Functions • Task state	November 18, 2022
Update	Step Functions now provides a new console experience for viewing and debugging Express workflow executions. For more information see: • Standard and Express console experience differences • Viewing execution details in the Step Functions console	October 18, 2022
Update	Added support to optionally specify the `ExecutionRoleArn` parameter while using the `addStep` and `addStep.sync` APIs for the Amazon EMR optimized service integration. For more information, see Create and manage Amazon EMR clusters with Step Functions.	September 20, 2022
Documentation-only update	Added a new topic that provides recommendations about optimizing cost while building serverless workflows using Step Functions. For more information, see Optimizing costs using Express Workflows.	September 15, 2022

Change	Description	Date changed
Update	Step Functions adds support for 14 new intrinsic functions for performing data processing tasks, such as array manipulations, data encoding and decoding, hash calculations, JSON data manipulation, math function operations, and unique identifier generation. **Documentation-only update:** Grouped all the existing and newly introduced intrinsic functions into the following categories based on the type of data processing task they help you perform: • Intrinsics for arrays • Intrinsics for data encoding and decoding • Intrinsic for hash calculation • Intrinsics for JSON data manipulation • Intrinsics for Math operations • Intrinsic for String operation • Intrinsic for unique identifier generation • Intrinsic for generic operation For more information, see Intrinsic functions for JSONPath states in Step Functions .	August 31, 2022
Update	Step Functions has expanded support for AWS SDK integrations by adding three more AWS services – AWS Billing Conductor, Amazon GameSparks, and Amazon Pinpoint SMS and Voice V2. For more information, see Learning to use AWS service SDK integrations in Step Functions.	July 26, 2022

Change	Description	Date changed
Documentation-only update	Added a new topic to include a summary of all the updates made to AWS SDK integrations supported by Step Functions. For more information, see Learning to use AWS service SDK integrations in Step Functions	July 26, 2022
Documentation-only update	*AWS Step Functions Developer Guide* now includes details about the execution metrics that are emitted specifica lly for Express Workflows. For more information, see Execution metrics for Express Workflows.	June 09, 2022

Change	Description	Date changed
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 informati on 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. **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 execution details in the Step Functions console • Examining state machine executions in Step Functions	May 09, 2022

Change	Description	Date changed
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	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. • 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.	April 19, 2022
New feature	Step Functions Local now supports AWS SDK integration and mocking of service integrations. For more information, see: • Using mocked service integrations for testing in Step Functions Local	January 28, 2022
New feature	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 (AWS CDK). For more information, see: • Using AWS CDK to create an Express workflow in Step Functions	December 10, 2021

Change	Description	Date changed
Update	Step Functions has added three new sample projects that demonstrate the integration of Step Functions and Amazon Athena's upgraded console. For more informati on, see: • Execute queries in sequence and parallel using Athena • Query large datasets using an AWS Glue crawler • Keep data in a target table updated with AWS Glue and Athena	November 22, 2021
New feature	Step Functions has added Amazon VPC endpoints support for Synchronous Express Workflows. For more informati on, see: • Creating Amazon VPC endpoints for Step Functions	November 15, 2021
Update	AWS Step Functions has added three new sample projects that demonstrate how to use the Step Functions AWS Batch integration. For more information, see: • Fan out batch jobs with Map state • Run an AWS Batch job with Lambda • Manage a batch job with AWS Batch and Amazon SNS	October 14, 2021
New feature	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: • Learning to use AWS service SDK integrations in Step Functions • Gather Amazon S3 bucket info using AWS SDK service integrations	September 30, 2021

Change	Description	Date changed
New feature	AWS Step Functions has added a visual workflow designer, the AWS Step Functions Workflow Studio. For more information, see: • Developing workflows in Step Functions Workflow Studio	June 17, 2021
Update	AWS Step Functions has added four new APIs, `StartBuildBatch` , `StopBuildBatch` , `RetryBuildBatch` and `DeleteBuildBatch` , to the CodeBuild integration. For more information, see: • Manage AWS CodeBuild builds with Step Functions	June 4, 2021
New feature	AWS Step Functions now integrates with Amazon EventBridge. For more information, see: • Add EventBridge events with Step Functions • IAM policies for Step Functions and IAM policies for calling EventBridge • A sample project that shows how to Send a custom event to an EventBridge event bus	May 14, 2021
Update	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: • Run an ETL/ELT workflow using Step Functions and the Amazon Redshift API	April 16, 2021
New feature	AWS Step Functions has a new data flow simulator in the console. For more information, see: • Data flow simulator (unsupported)	April 8, 2021

Change	Description	Date changed
New feature	AWS Step Functions now integrates with Amazon EMR on EKS. For more information, see: • Create and manage Amazon EMR clusters on EKS with AWS Step Functions	March 29, 2021
Update	YAML support for state machine definitions has been added to AWS Toolkit for Visual Studio Code and AWS CloudFormation. For more information, see: • AWS Toolkit for Visual Studio Code	March 4, 2021
New feature	AWS Step Functions now integrates with AWS Glue DataBrew. For more information, see: • Start AWS Glue DataBrew jobs with Step Functions • What is AWS Glue DataBrew? in the DataBrew developer guide.	January 6, 2021
New feature	AWS Step Functions Synchronous Express Workflows are now available, giving you an easy way to orchestrate microservices. For more information, see: • Synchronous and Asynchronous Express Workflows in Step Functions • A sample project that shows how to Invoke Synchronous Express Workflows through API Gateway • The StartSyncExecution API documentation.	November 24, 2020

Change	Description	Date changed
New feature	AWS Step Functions now integrates with Amazon API Gateway. For more information, see: • Create API Gateway REST APIs with Step Functions • IAM policies for Step Functions and IAM policies for calls to Amazon API Gateway • A sample project that shows how to Interact with an API managed by API Gateway	November 17, 2020
New feature	AWS Step Functions now integrates with Amazon Elastic Kubernetes Service. For more information, see: • Create and manage Amazon EKS clusters with Step Functions • IAM policies for Step Functions and IAM policies for calling Amazon EKS • A sample project that shows how to Create and manage an Amazon EKS cluster with a node group	November 16, 2020
New feature	AWS Step Functions now integrates with Amazon Athena. For more information, see: • Run Athena queries with Step Functions • IAM policies for Step Functions and IAM policies for calling Amazon Athena • A sample project that shows how to Start an Athena query and send a results notification	October 22, 2020

Change	Description	Date changed
New feature	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: • Trace Step Functions request data in AWS X-Ray • IAM policies for Step Functions and IAM policies using AWS X-Ray in Step Functions • AWS Step Functions API Reference ◦ TracingConfiguration	September 14, 2020

Change	Description	Date changed

Update	AWS Step Functions now supports payload sizes up to 256 KiB of data as a UTF-8 encoded string. This lets you process larger payloads in both Standard and Express workflows.
	
	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:
	
	• *Service quotas*
	• the section called "Using Amazon S3 to pass large data"
	• `States.DataLimitExceeded`
	• the section called "CloudWatch Logs payloads"
	• AWS Step Functions API Reference
	  ◦ CloudWatchEventsExecutionDataDetails
	  ◦ HistoryEventExecutionDataDetails
	  ◦ GetExecutionHistory
	  ◦ ActivityScheduledEventDetails
	  ◦ ActivitySucceededEventDetails
	  ◦ CloudWatchEventsExecutionDataDetails
	  ◦ ExecutionSucceededEventDetails
	  ◦ LambdaFunctionScheduledEventDetails
	  ◦ ExecutionSucceededEventDetails
	  ◦ StateEnteredEventDetails
	  ◦ StateExitedEventDetails
	  ◦ TaskSubmittedEventDetails
	  ◦ TaskSucceededEventDetails
		September 3, 2020

Change	Description	Date changed

Update	The Amazon States Language has been updated as follows:
	
	• Choice Rules (JSONata) has added
	  ◦ 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.
	  ◦ Four other new operators have been added, IsBoolean , IsNumeric, IsString and IsTimestamp.
	  ◦ 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.
	  ◦ Wildcard pattern matching to support string comparison against patterns with one or more wildcards.
	  ◦ Comparison between two variables for supported comparison operators.
	• 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 workflow state state for more information.
	• The new ResultSelector field provides a way to manipulate a state's result before `ResultPath` is applied. The `ResultSelector` field is an optional field in the Map workflow state, Parallel workflow state, and Task workflow state states.
	• Intrinsic functions for JSONPath states in Step Functions have been added to allow basic operation s without `Task` states. Intrinsic functions can be used
		August 13, 2020

Change	Description	Date changed
	within the `Parameters` and `ResultSelector` fields.	
Update	AWS Step Functions now supports the Amazon SageMaker AI `CreateProcessingJob` API call. For more information, see: • Create and manage Amazon SageMaker AI jobs with Step Functions • Preprocess data and train a machine learning model with Amazon SageMaker AI, a sample project that demonstrates `CreateProcessingJob` .	August 4, 2020
New feature	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: • Using AWS SAM to build Step Functions workflows • AWS::Serverless::StateMachine • AWS SAM Policy Templates	May 27, 2020
New feature	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: • Start a new AWS Step Functions state machine from a running execution	May 19, 2020

Change	Description	Date changed
New feature	AWS Step Functions now integrates with AWS CodeBuild. For more information, see: • Integrating services with Step Functions • Manage AWS CodeBuild builds with Step Functions • Integrating services with Step Functions	May 5, 2020
New feature	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.	March 31, 2020
Update	You can now configure logging to Amazon CloudWatch Logs for Standard workflows. For more information, see: • Using CloudWatch Logs to log execution history in Step Functions	February 25, 2020
New feature	AWS Step Functions can now be accessed without requiring a public IP address, directly from Amazon Virtual Private Cloud (VPC). For more information, see: • Creating Amazon VPC endpoints for Step Functions	December 23, 2019

Change	Description	Date changed

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.
	
	• Choosing workflow type in Step Functions
	    • Execution guarantees in Step Functions workflows
	• Integrating services with Step Functions
	    • Integrating services with Step Functions
	• Process high-volume messages from Amazon SQS with Step Functions Express workflows
	• Perform selective checkpointing using Standard and Express workflows
	• Step Functions service quotas
	    • Step Functions service quotas
	• Using CloudWatch Logs to log execution history in Step Functions
	• AWS Step Functions API Reference
	    • CreateStateMachine
	    • UpdateStateMachine
	    • DescribeStateMachine
	    • DescribeStateMachineForExecution
	    • StopExecution
	    • DescribeExecution
	    • GetExecutionHistory
	    • ListExecutions
	    • ListStateMachines
		December 3, 2019

Change	Description	Date changed
	• StartExecution • CloudWatchLogsLogGroup • LogDestination • LoggingConfiguration	
New feature	AWS Step Functions now integrates with Amazon EMR. For more information, see: • Integrating services with Step Functions • Create and manage Amazon EMR clusters with Step Functions • Integrating services with Step Functions	November 19, 2019

Update	AWS Step Functions has released the AWS Step Functions Data Science SDK. For more information, see the following.
	• Project on Github
	• SDK Documentation
	• The following Example Notebooks, which are available in the SageMaker AI console and the related GitHub project.
	  • `hello_world_workflow.ipynb`
	  • `machine_learning_workflow_abalone.ipynb`
	  • `training_pipeline_pytorch_mnist.ipynb`
		November 7, 2019

Change	Description	Date changed
Update	Step Functions now supports more API actions for Amazon SageMaker AI, and includes two new sample projects to demonstrate the functionality. For more information, see the following. • Create and manage Amazon SageMaker AI jobs with Step Functions • Integrating services with Step Functions • Train a machine learning model using Amazon SageMaker AI • Tune the hyperparameters of a machine learning model in SageMaker AI	October 3, 2019
New feature	Step Functions supports starting new workflow execution s by calling `StartExecution` as an integrated service API. See: • Start workflow executions from a task state in Step Functions • Start a new AWS Step Functions state machine from a running execution • Integrating services with Step Functions • IAM Policies for Starting Step Functions Workflow Executions	August 12, 2019

Change	Description	Date changed
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: • Discover service integration patterns in Step Functions • Wait for a Callback with Task Token • Create a callback pattern example with Amazon SQS, Amazon SNS, and Lambda • Integrating services with Step Functions • Deploying a workflow that waits for human approval in Step Functions • Service Integration Metrics Step Functions now provides a way to access dynamic information about your current execution directly in the `"Parameters"` field of a state definition. See: • Accessing execution data from the Context object in Step Functions • Pass Context object nodes as parameters	May 23, 2019
New feature	Step Functions supports CloudWatch Events for execution status changes, see: • Automating Step Functions event delivery with EventBridge	May 8, 2019
New feature	Step Functions supports IAM permissions using tags. For more information, see: • Tagging state machines and activities in Step Functions • Creating tag-based IAM policies in Step Functions	March 5, 2019

Change	Description	Date changed
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 state machines with Step Functions Local (unsupported).	February 4, 2019
New feature	AWS Step Functions is now available in the Beijing and Ningxia regions.	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 state machines and activities in Step Functions.	January 7, 2019
New feature	AWS Step Functions is now available in the Europe (Paris), and South America (São Paulo) regions.	December 13, 2018
New feature	AWS Step Functions is now available the Europe (Stockhol m) region.	December 12, 2018
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: • Integrating services with Step Functions • Passing parameters to a service API in Step Functions • Integrating services with Step Functions	November 29, 2018
Update	Improved the description of `TimeoutSeconds` and `HeartbeatSeconds` in the documentation for task states. See Task workflow state.	October 24, 2018

Change	Description	Date changed
Update	Improved the description for the *Maximum execution history size* limit and provided a link to the related best practices topic. • Quotas related to state machine executions • Starting new executions to avoid reaching the history quota in Step Functions	October 17, 2018
Update	Added a new tutorial to the AWS Step Functions documentation: See Starting a Step Functions workflow in response to events.	September 25, 2018
Update	Removed the entry *Maximum executions displayed in Step Functions console* from the limits documentation. See Step Functions service quotas.	September 13, 2018
Update	Added a best practices topic to the AWS Step Functions documentation on improving latency when polling for activity tasks. See Avoiding latency when polling for activity tasks.	August 30, 2018
Update	Improved the AWS Step Functions topic on activities and activity workers. See Learn about Activities in Step Functions.	August 29, 2018
Update	Improved the AWS Step Functions topic on CloudTrail integration. See Recording Step Functions API calls with AWS CloudTrail.	August 7, 2018
Update	Added JSON examples to AWS CloudFormation tutorial. See Using AWS CloudFormation to create a workflow in Step Functions.	June 23, 2018
Update	Added a new topic on handling Lambda service errors. See Handle transient Lambda service exceptions.	June 20, 2018

Change	Description	Date changed
New feature	AWS Step Functions is now available the Asia Pacific (Mumbai) region.	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
Update	Improved documentation on error handling for `Parallel` states. See Error Handling.	June 20, 2018
Update	Improved documentation about Input and Output processing in Step Functions. Learn how to use `InputPath`, `ResultPath`, and `OutputPath` to control the flow of JSON through your workflows, states, and tasks. See: • Processing input and output in Step Functions • Specifying state output using ResultPath in Step Functions	June 7, 2018
Update	Improved code examples for parallel states. See Parallel workflow state.	June 4, 2018
New feature	You can now monitor API and Service metrics in CloudWatch. See Monitoring Step Functions metrics using Amazon CloudWatch.	May 25, 2018

Change	Description	Date changed
Update	`StartExecution` , `StopExecution` , and `StateTransition` now have increased throttling limits in the following regions: • US East (N. Virginia) • US West (Oregon) • Europe (Ireland) For more information see Step Functions service quotas.	May 16, 2018
New feature	AWS Step Functions is now available the US West (N. California) and Asia Pacific (Seoul) regions. See AWS Services by Region for a list of supported regions.	May 5, 2018
Update	Updated procedures and images to match changes to the interface.	April 25, 2018
Update	Added a new tutorial that shows how to start a new execution to continue your work. See Continue long-running workflows using Step Functions API (recommended). This tutorial describes a design pattern that can help avoid some service limitations. See Starting new executions to avoid reaching the history quota in Step Functions.	April 19, 2018
Update	Improved introduction to states documentation by adding conceptual information about state machines. See Discovering workflow states to use in Step Functions.	March 9, 2018

Change	Description	Date changed

New feature	• When you create a new state machine, you must acknowledge that AWS Step Functions will create an IAM role which allows access to your Lambda functions.
	• Updated the following tutorials to reflect the minor changes in the state machine creation workflow:
	  • Creating a Step Functions state machine that uses Lambda
	  • Creating an Activity state machine using Step Functions
	  • Handling error conditions in a Step Functions state machine
	  • Iterate a loop with a Lambda function in Step Functions
		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.	February 6, 2018
Update	Added a new tutorial describing a design pattern that uses a Lambda function to iterate a count. See Creating a Step Functions state machine that uses Lambda.	January 31, 2018
Update	Updated content on IAM permissions to include `DescribeStateMachineForExecution` and `UpdateStateMachine` APIs. See Creating granular permissions for non-admin users in Step Functions.	January 26, 2018

Change	Description	Date changed
Update	Added newly available regions: Canada (Central), Asia Pacific (Singapore).	January 25, 2018
Update	Updated tutorials and procedures to reflect that IAM allows you to select *Step Functions* as a role.	January 24, 2018
Update	Added a new *Best Practices* topic that suggests not passing large payloads between states. See Using Amazon S3 ARNs instead of passing large payloads in Step Functions.	January 23, 2018
Update	Corrected procedures to match updated interface for creating a state machine: • Creating a Step Functions state machine that uses Lambda • Creating an Activity state machine using Step Functions • Handling error conditions in a Step Functions state machine	January 17, 2018

Change	Description	Date changed
New Feature	You can use *Sample Projects* to quickly provision state machines and all related AWS resources. See [Deploy a state machine using a starter template for Step Functions](), Available sample projects include: • Poll for job status with Lambda and AWS Batch • Create a task timer with Lambda and Amazon SNS **Note** These sample projects and related documentation replace tutorials that described implementing the same functionality.	January 11, 2018
Update	Added a *Best Practices* section that includes information on avoiding stuck executions. See Best practices for Step Functions.	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

Change	Description	Date changed
Update	Added information related to resource names: **Note** Step Functions accepts names for state machines, executions, activities, and labels that contain non-ASCII characters. Because such character s will not work with Amazon CloudWatch, we recommend using only ASCII characters so you can track metrics in CloudWatch.	December 6, 2017
Update	Improved security overview information and added a topic on granular IAM permissions. See Security in AWS Step Functions and Creating granular permissions for non-admin users in Step Functions.	November 27, 2017

Update	• Added new screenshots for state machine execution results to reflect changes in the Step Functions console. Rewrote the Lambda instructions in the following tutorials to reflect changes in the Lambda console:
	  • Creating a Step Functions state machine that uses Lambda
	  • Creating a Job Status Poller
	  • Creating a Task Timer
	  • Handling error conditions in a Step Functions state machine
	• Corrected and clarified information about creating state machines in the following sections:
	  • Creating an Activity state machine using Step Functions
		October 6, 2017

Change	Description	Date changed
Update	Rewrote the IAM instructions in the following sections to reflect changes in the IAM console: • Creating an IAM role for your state machine in Step Functions • Creating a Step Functions state machine that uses Lambda • Creating a Job Status Poller • Creating a Task Timer • Handling error conditions in a Step Functions state machine • Creating a Step Functions API using API Gateway	October 5, 2017
Update	Rewrote the State Machine Data section.	September 28, 2017
New feature	The limits related to API action throttling are increased for all regions where Step Functions is available.	September 18, 2017
Update	• Corrected and clarified information about starting new executions in all tutorials. • Corrected and clarified information in the Quotas related to accounts section.	September 14, 2017
Update	Rewrote the following tutorials to reflect changes in the Lambda console: • Creating a Step Functions state machine that uses Lambda • Handling error conditions in a Step Functions state machine • Creating a Job Status Poller	August 28, 2017

Change	Description	Date changed
New feature	Step Functions is available in Europe (London).	August 23, 2017
New feature	The visual workflows of state machines let you zoom in, zoom out, and center the graph.	August 21, 2017
New feature	**Important** An execution can't use the name of another execution for 90 days. When you make multiple `StartExecution` calls with the same name, the new execution doesn't run. For more information, see the name request parameter of the `StartExecution` API action in the *AWS Step Functions API Reference*.	August 18, 2017
Update	Added information about an alternative way of passing the state machine ARN to the Creating a Step Functions API using API Gateway 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 metrics using Amazon CloudWatch. • Added the Quotas related to state throttling section.	August 3, 2017
Update	Updated the instructions in the Step 1: Create an IAM Role for API Gateway section.	July 18, 2017

Change	Description	Date changed
Update	Corrected and clarified information in the Choice workflow state 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 • Using AWS CloudFormation to create a workflow in Step Functions • Creating an Activity state machine using Step Functions • Handling error conditions in a Step Functions state machine	June 22, 2017
Update	Corrected and clarified information in the following sections: • Handling error conditions in a Step Functions state machine • Discovering workflow states to use in Step Functions • Handling errors in Step Functions workflows	June 21, 2017
Update	Rewrote all tutorials to match the Step Functions console refresh.	June 12, 2017
New feature	Step Functions is available in Asia Pacific (Sydney).	June 8, 2017
Update	Restructured the Using Amazon States Language to define Step Functions workflows section.	June 7, 2017
Update	Corrected and clarified information in the Creating an Activity state machine using Step Functions section.	June 6, 2017

Change	Description	Date changed
Update	Corrected the code examples in the State machine examples using Retry and using Catch section.	June 5, 2017
Update	Restructured this guide using AWS documentation standards.	May 31, 2017
Update	Corrected and clarified information in the Parallel workflow state section.	May 25, 2017
Update	Merged the Paths and Filters sections into the Processing input and output in Step Functions section.	May 24, 2017
Update	Corrected and clarified information in the Monitoring Step Functions metrics using Amazon CloudWatch section.	May 15, 2017
Update	Updated the `GreeterActivities.java` worker code in the Creating an Activity state machine using Step Functions tutorial.	May 9, 2017
Update	Added an introductory video to the What is Step Functions? section.	April 19, 2017
Update	Corrected and clarified information in the following tutorials: • Creating a Step Functions state machine that uses Lambda • Creating an Activity state machine using Step Functions • Handling error conditions in a Step Functions state machine	April 19, 2017
Update	Added information about Lambda templates to the Creating a Step Functions state machine that uses Lambda and Handling error conditions in a Step Functions state machine tutorials.	April 6, 2017

Change	Description	Date changed
Update	Changed the "Maximum input or result data size" limit to "Maximum input or result data size for a task, state, or execution" (32,768 characters). For more information, see Quotas related to task executions.	March 31, 2017
New feature	• Step Functions supports executing state machines by setting Step Functions as Amazon CloudWatch Events targets.	March 21, 2017
New feature	• Step Functions allows Lambda function error handling as the preferred error handling method. • Updated the Handling error conditions in a Step Functions state machine tutorial and the Handling errors in Step Functions workflows section.	March 16, 2017
New feature	Step Functions is available in Europe (Frankfurt).	March 7, 2017
Update	Reorganized the topics in the table of contents and updated the following tutorials: • Creating a Step Functions state machine that uses Lambda • Creating an Activity state machine using Step Functions • Handling error conditions in a Step Functions state machine	February 23, 2017
New feature	• The **State Machines** page of the Step Functions console includes the **Copy to New** and **Delete** buttons. • Updated the screenshots to match the console changes.	February 23, 2017

Change	Description	Date changed
New feature	• Step Functions supports creating APIs using API Gateway. • Added the Creating a Step Functions API using API Gateway tutorial.	February 14, 2017
New feature	• Step Functions supports integration with AWS CloudFormation. • Added the Using AWS CloudFormation to create a workflow in Step Functions tutorial.	February 10, 2017
Update	Clarified the current behavior of the `ResultPath` and `OutputPath` fields in relation to `Parallel` states.	February 6, 2017
Update	• Clarified state machine naming restrictions in tutorials. • Corrected some code examples.	January 5, 2017
Update	Updated Lambda function examples to use the latest programming model.	December 9, 2016
Initial release	Initial release of AWS Step Functions.	December 1, 2016