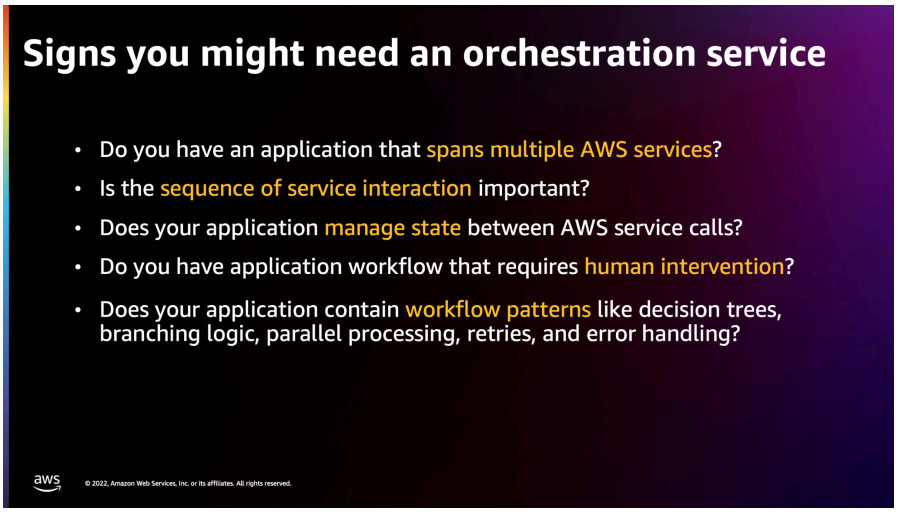
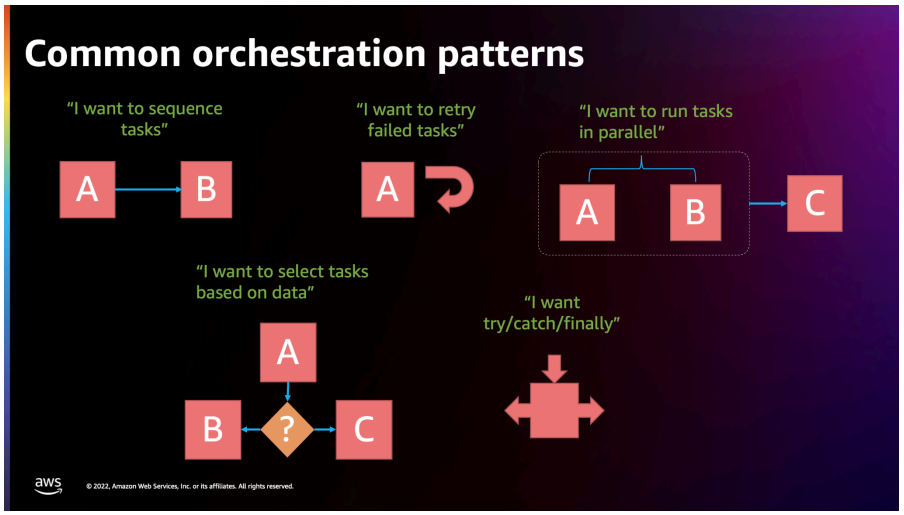
Fuentes:

- https://catalog.workshops.aws/stepfunctions/en-US
- https://catalog.workshops.aws/serverless-data-processing/en-US

The workshop materials cover:

- Amazon States Language (ASL)
- Task Orchestration Patterns
  - Request Response
  - Run a Job (.sync)
  - Wait for Callback (.waitForTaskToken)
- States (Task, Choice, Map, Parallel, Wait, Pass, Success, Fail)
- Standard Workflows and Express Workflows
- Workflow Studio
- Input and output processing
- API Gateway integration
- Error handling
- AWS SDK integrations
- AWS CDK deployments
- AWS SAM deployments



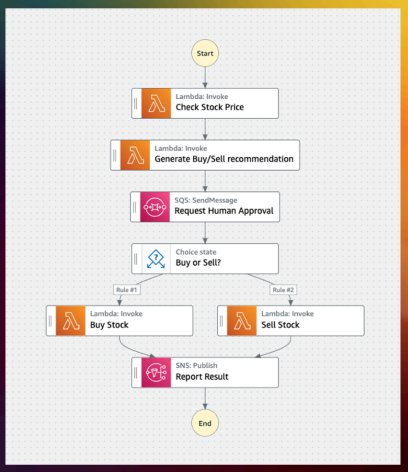


Introducing AWS Step Functions

- Low-code visual workflow service
- Provides orchestration for other AWS services
- Supports common workflow patterns
- Simplifies implementation of common tasks so developers can focus on higher-value business logic

Step Functions Workflows

- Are written using Amazon States Language (ASL)
- Are defined as state machines
- Are composed of steps called states
- Can be used to orchestrate multiple AWS services



Step Functions states

Choice
Adds if-then-else logic.

Parallel
Adds parallel branches.

Map
Adds a for-each loop.

Wait
Delays for a specified time.

Pass
Transforms data or acts as placeholder.

Success
Stops and marks as success.

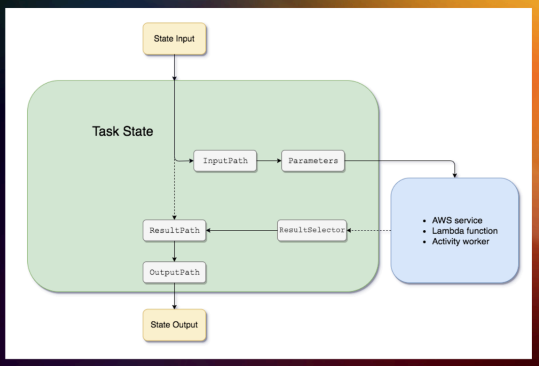
Fail
Stops and marks as failure.

Task

- AWS Lambda: Invoke
- Amazon SNS: Publish
- Amazon ECS: RunTask
- AWS Step Functions: StartExecution
- AWS Glue: StartJobRun

Input and Output Processing

- A Step Functions execution receives a JSON text as input and passes that input to the first state in the workflow.
- Individual states receive JSON as input and usually pass JSON as output to the next state.
- ASL provides tools to filter, manipulate and transform input and output between states.

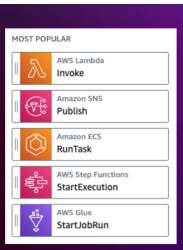


Standard vs. Express workflows

	Standard	Express
Use Cases	Ideal for **long-running**, durable, and auditable workflows.	Ideal for **high-volume** event-processing workloads.
Maximum duration	365 days	5 minutes
State Transitions	5,000 initial bucket with 1,500 per second refill rate (in limited regions)	Unlimited
Patterns	Supports all patterns	Does not support Job-run (.sync) or Callback (.wait For Callback)
Invocation	Asynchronous invocation only	Asynchronous and synchronous invocation
Pricing	$0.025 per 1,000 state transitions	$1.00 per 1M requests + $0.06 per GB-hour (tiered)

AWS SDK Integrations

- Step Functions integrates with 220+ supported AWS Services
- Step Functions supports 10,000+ API Actions



Integration syntax:

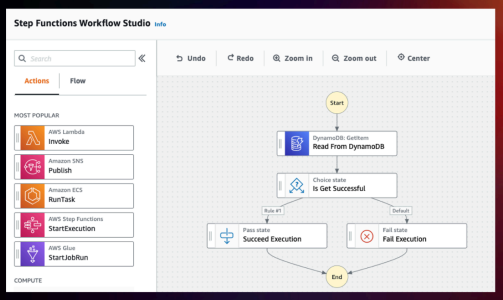
```
arn:aws:states:::aws-sdk:serviceName:apiAction.[serviceIntegrationPattern]
```

Example integration:

```
arn:aws:states:::aws-sdk:ec2:describeInstances
```

Workflow Studio

A low code visual editor for designing workflows



Service Integration Patterns

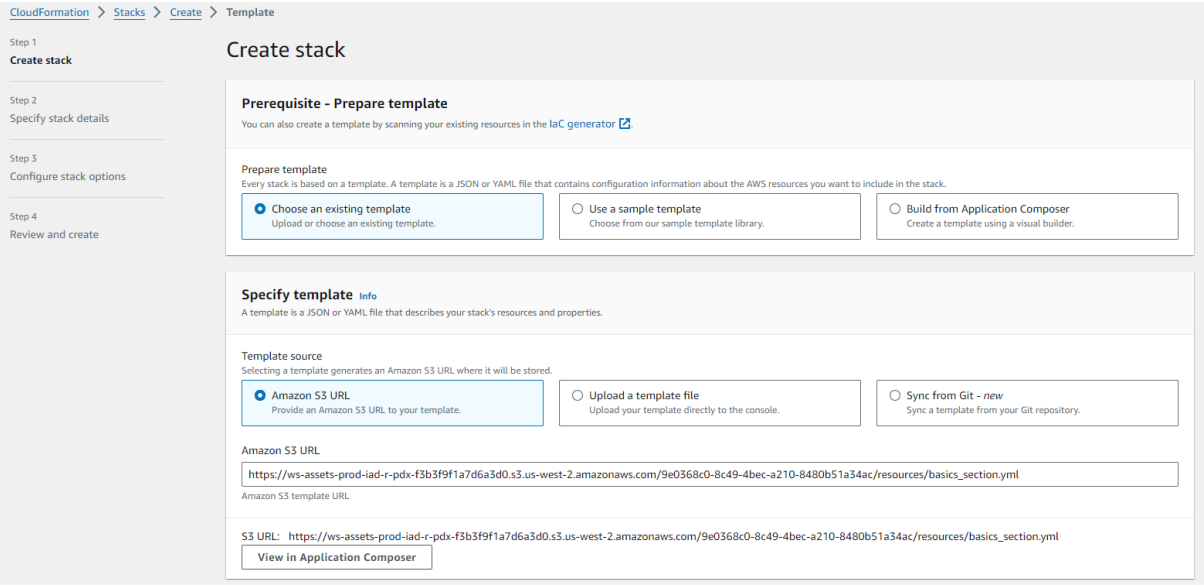
Request Response	Step Functions will call a service integration and proceed immediately without waiting for the service to complete its job.
Sync Polling	Step Functions will call a service integration and wait until its job is complete.
Callback	Step functions will call a service with a task token and proceed after the token is returned.

Example Use Cases

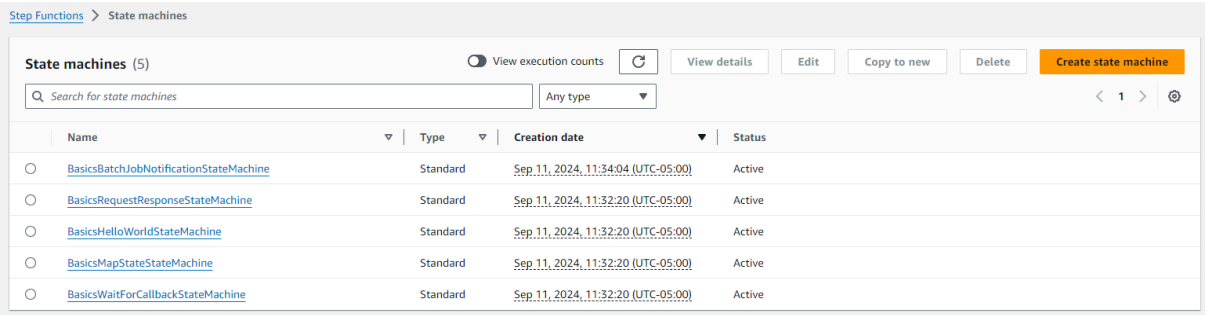
- Data processing (file, video, image; ETL)
- Machine learning (fraud detection, recommendations, data enrichment)
- Microservice orchestration (synchronous workflows, container orchestration)
- IT and security automation (incident management, auto-remediation, human approval scenarios)
- High performance computing (HPC)

Basics

Se toma el template de CloudFormation proporcionado por AWS (region us-west-2).

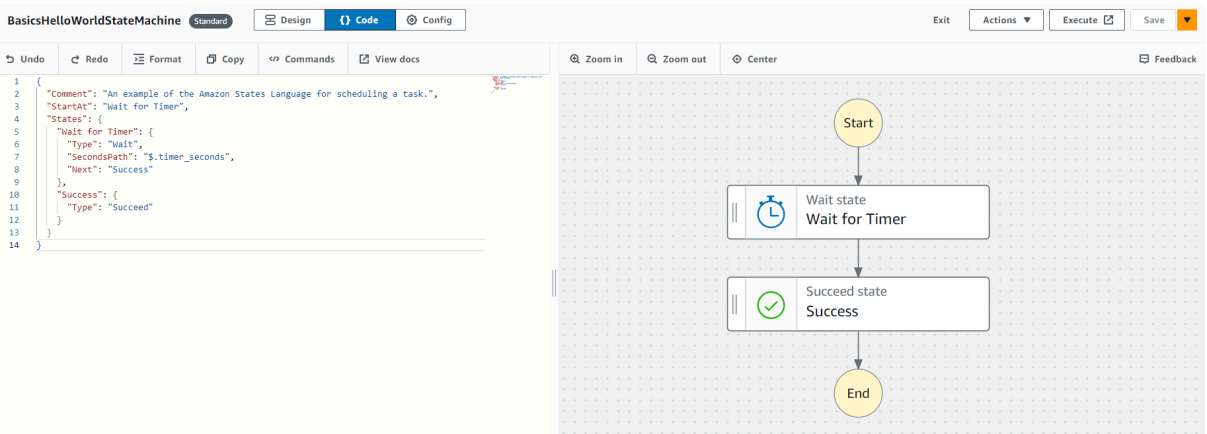


La configuración restante se deja por defecto. El template crea las siguientes state machines de tipo standard, como también otros servicios de AWS.

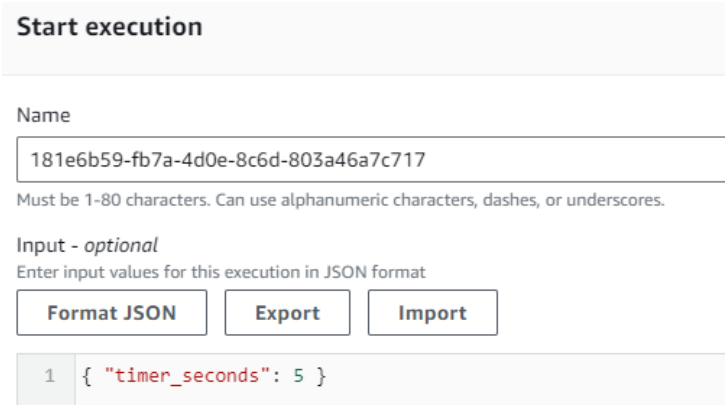


Exercise 1

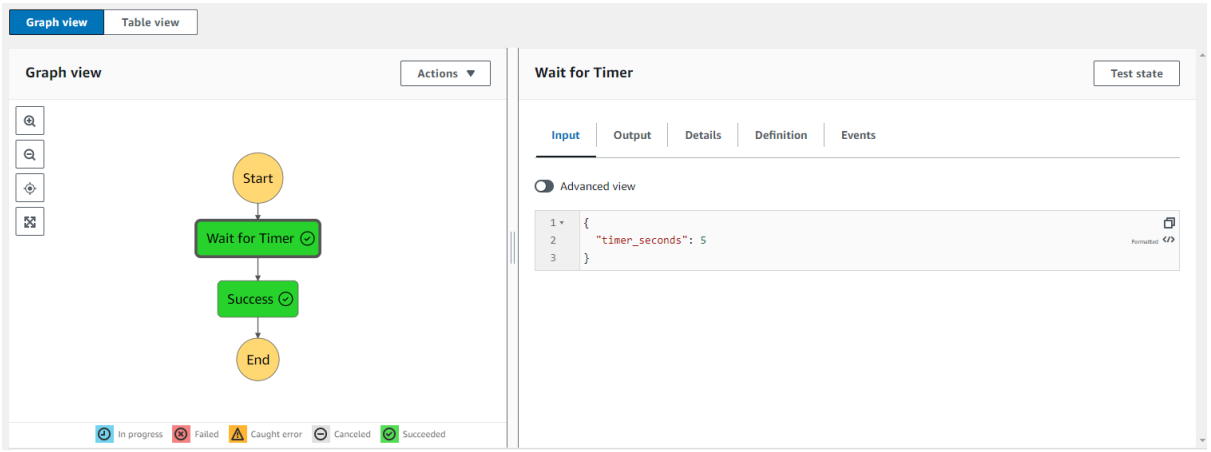
Se ingresa al state llamado BasicsHelloWorldStateMachine. Al dar click en la opción de Editar y luego en Code, se puede ver la definición de la state en Amazon States Language (ASL).



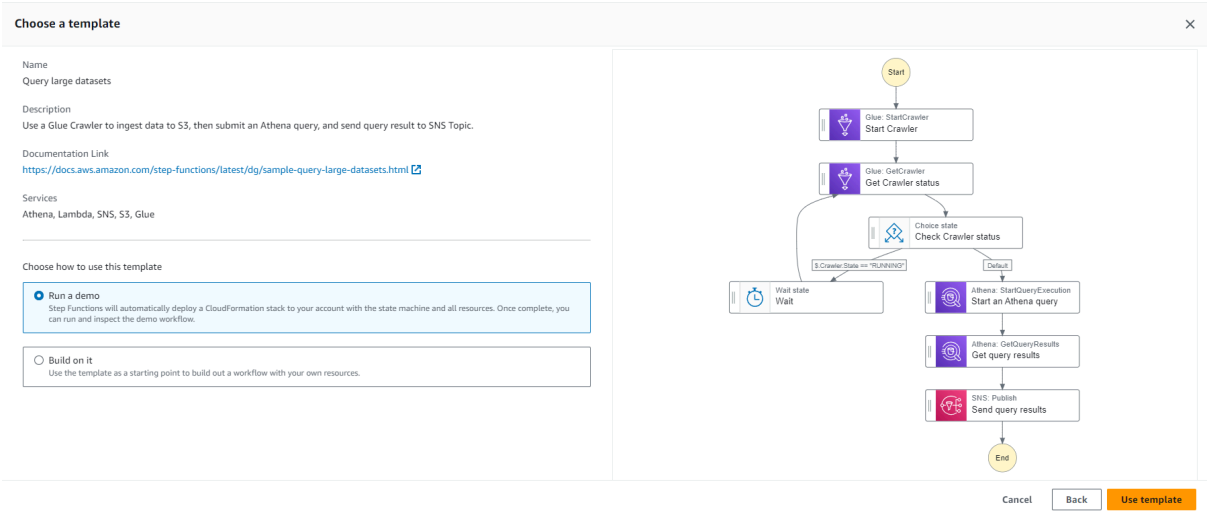
Para ejecutar el state, se debe enviar el valor del atributo time_seconds declarado en la definition. En este caso se establecen 5 segundos.



Al ejecutarse, los states toman un color verde indicando que todo concluyó satisfactoriamente.



Este servicio también ofrece la opción de elegir algún template que se adapte al escenario correspondiente.



Task State

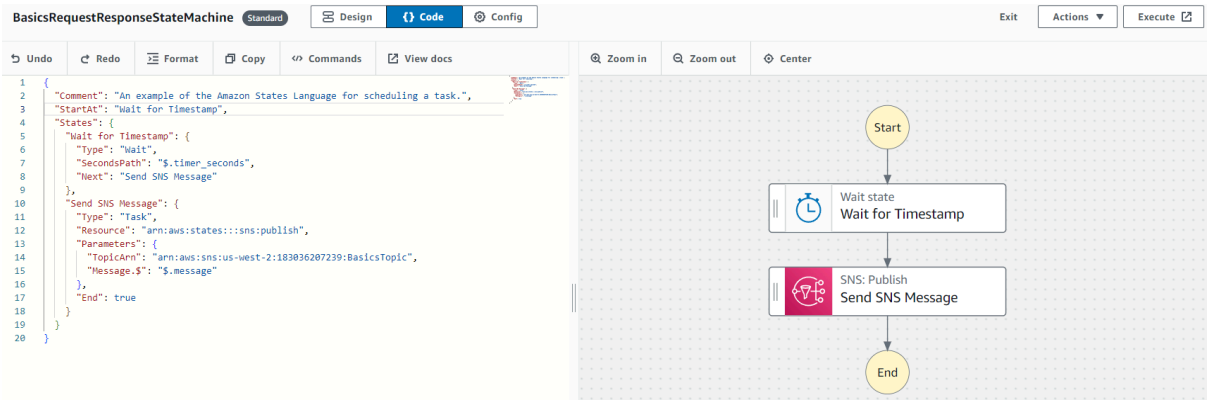
A Task state represents a **single unit of work** performed by a state machine. All work in your state machine is done by **tasks**. A task typically performs work by passing parameters to the API actions of other services which then perform their own activities. Step Functions supports API actions for over 200 AWS services. You can specify how a Task performs using a number of fields including Credentials, Retry,Catch, TimeoutSeconds, and others.

Exercise 2

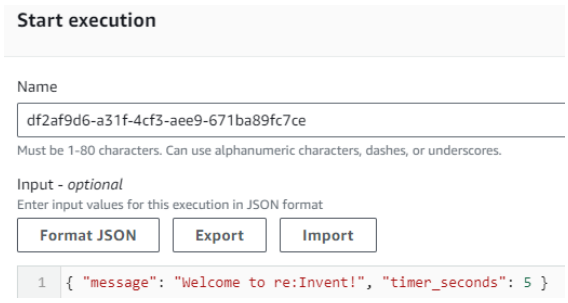
When Step Functions calls another service using the Task state, the default pattern is Request Response. With this service integration pattern, Step Functions will call the service and then immediately proceed to the next state. The Task state will not wait for the underlying job to complete.

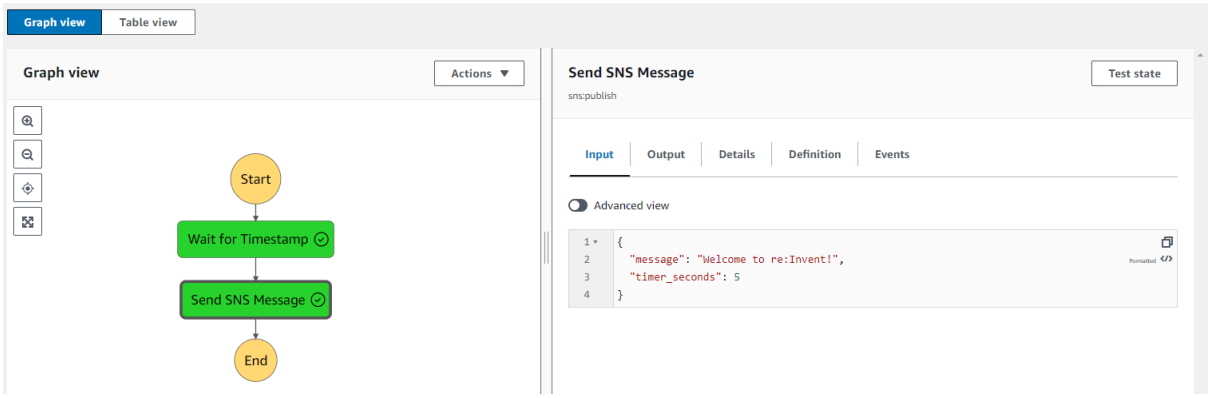
In this module you will run a Task using the **Request Response pattern**.

Se ingresa al state llamado BasicsRequestResponseStateMachine. Al dar click en la opción de Editar y luego en Code, se puede ver la definición de la state en Amazon States Language (ASL).



Se ejecuta con el input requerido.



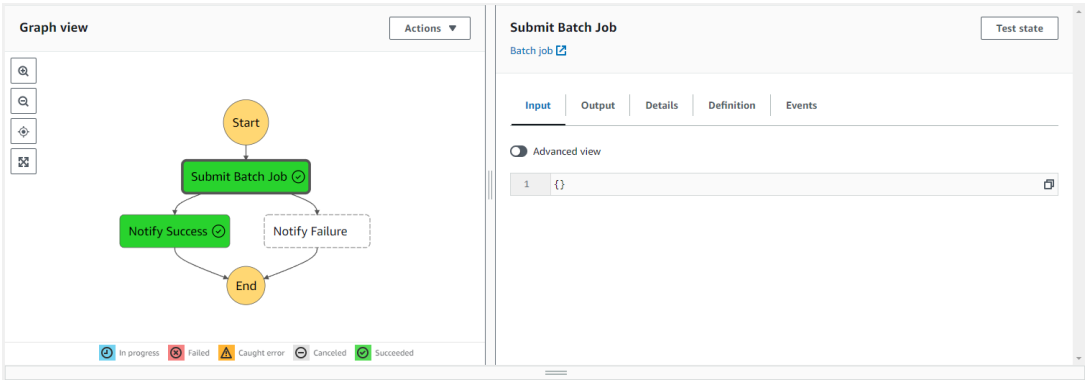


Exercise 3

For integrated services such as AWS Batch and Amazon ECS, Step Functions can wait for a task to complete before progressing to the next state. This task orchestration pattern is called Run a Job (.sync).

This workflow submits an AWS Batch job. In its initial state the job is **asynchronous**. The state machine submits the job to the AWS Batch service but does not wait for the job to complete. Instead, it immediately moves to the next state and sends a Notify Success message to an Amazon SNS topic.

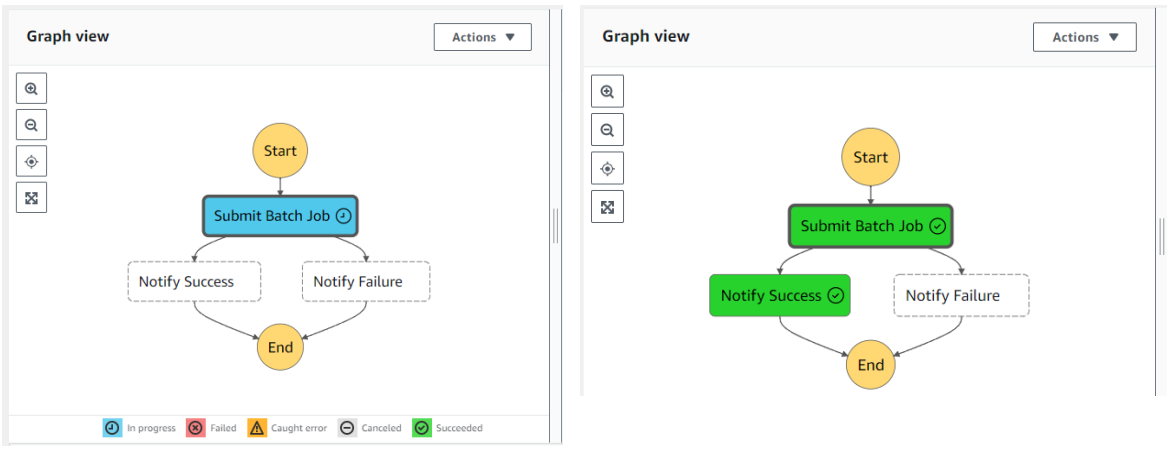
Se ejecuta el state.



Now you'll modify the ASL definition to make the task synchronous. Click the Edit state machine button and navigate to the {} code editor. Add the .sync method to the end of the Resource definition.

```
"States": {
  "Submit Batch Job": {
    "Type": "Task",
    "Resource": "arn:aws:states:::batch:submitJob.sync",
```

Ahora, al ser síncrono (.sync), primero se debe esperar a que finalice la ejecución del job de batch para luego continuar con los demás state steps.

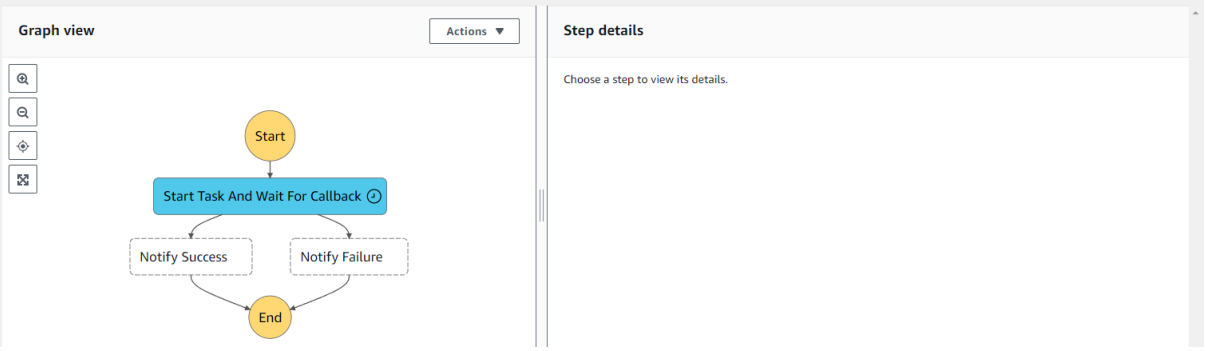


Exercise 4

The **Wait for Callback** feature provides a way to pause a workflow until a task token is returned.

For example, a task might require human approval, integrate with a third party, or call a legacy system. For workflows like these, a task can pass a unique token to the service integration and pause. The task will only resume when it receives the task token back with a SendTaskSuccess or SendTaskFailure call.

En primera instancia, se ejecuta el step y se ve que queda en espera en el 1er state step.



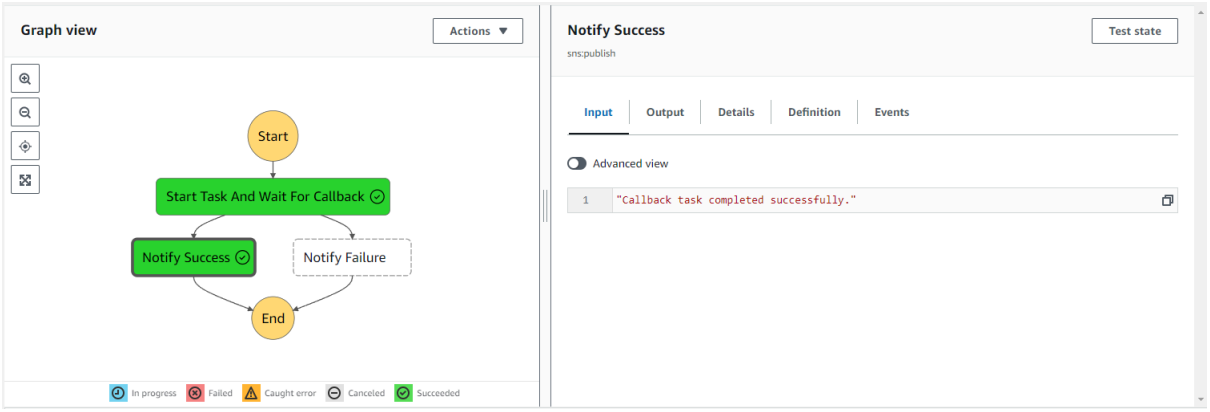
Review the ASL definition for this state machine. Although the .waitForTaskToken callback logic is implemented in the ASL definition, the callback is not yet being executed in the Lambda function that processes the SQS messages.

```
{
  "Comment": "An example of the Amazon States Language for starting a task and waiting for a callback.",
  "StartAt": "Start Task And Wait For Callback",
  "States": {
    "Start Task And Wait For Callback": {
      "Type": "Task",
      "Resource": "arn:aws:states:::sqs:sendMessage.waitForTaskToken",
      "Parameters": {
        "QueueUrl": "https://sqs.us-east-1.amazonaws.com/[redacted]/StepFunctionsSample-WaitForCallbacke1be56db-0f4d-4131-95a-SQSQueue-1HKE9M95ZHFY0",
        "MessageBody": {
          "MessageTitle": "Task started by Step Functions. Waiting for callback with task token.",
          "TaskToken.$": "$$.Task.Token"
        }
      },
```

Se debe ir a la función Lambda llamada BasicsCallbackWithTaskToken. Esta función es responsable de procesar los mensajes añadidos a la SQS queue (usando un trigger). La función recibe el TaskToken desde SQS y puede retornarlo al state machine como parámetro usando el método .sendTaskSuccess. Para ello, se quitan los comentarios en las siguientes dos lineas de código.

```
26  exports.lambda_handler = async(event, context, callback) => {
27
28      for (const record of event.Records) {
29          const messageBody = JSON.parse(record.body);
30          const taskToken = messageBody.TaskToken;
31
32          const params = {
33              output: "\"Callback task completed successfully.\"",
34              taskToken: taskToken
35          };
36
37          /**
38          * uncomment the lines below and redeploy the Lambda function
39          */
40          // console.log(`Calling Step Functions to complete callback task with params ${JSON.stringify(params)}`);
41          // let response = await stepfunctions.sendTaskSuccess(params).promise();
42      }
43  };
44
```

Al ejecutar nuevamente, se puede ver que todo concluyó satisfactoriamente.



Exercise 5

AWS SDK service integrations

AWS Step Functions integrates with many other AWS services. You can use Step Functions AWS SDK service integrations to call over 220 AWS services directly from your state machine, giving you access to over 10,000 API actions.

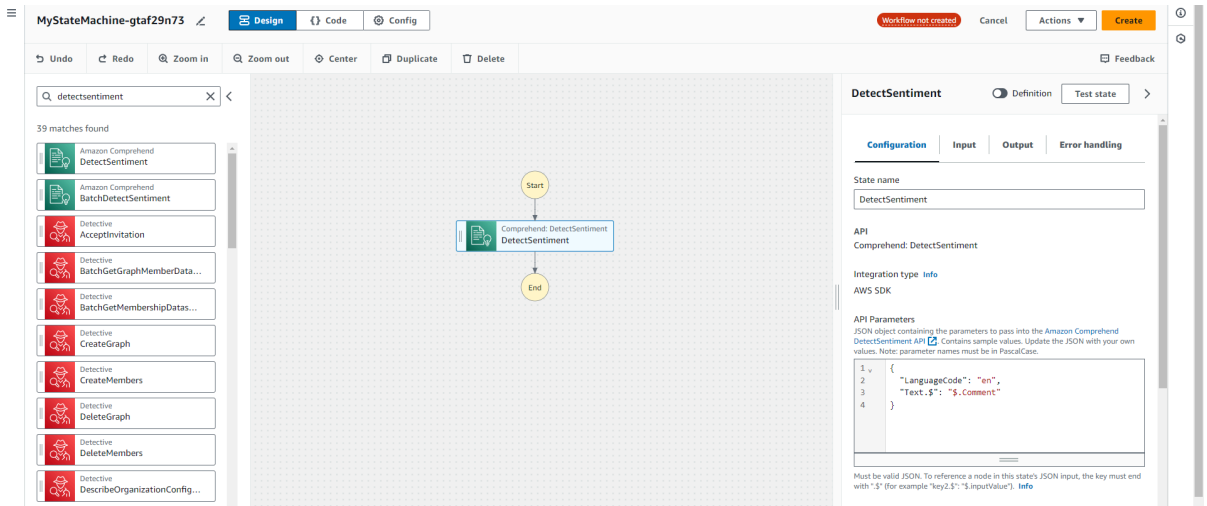
To use AWS SDK integrations, you specify the service name and API call. Some integrations require parameters and you may optionally specify a service integration pattern. Note that the API action will be camel case, and parameter names will be Pascal case. You can use Amazon States Language to specify an AWS API action in the Resource field of a task state. To do this, use the following syntax:

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

Examples

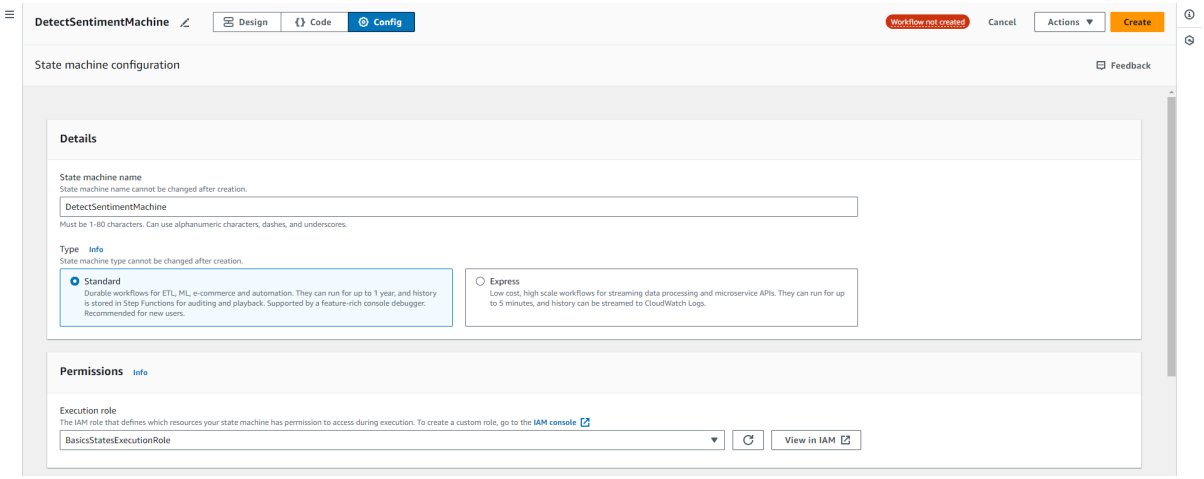
- To describe Amazon EC2 instances, use the syntax: `arn:aws:states:::aws-sdk:ec2:describeInstances`. This will return as output the return value of the Amazon EC2 describeInstances API call.
- To list buckets in Amazon S3, use `arn:aws:states:::aws-sdk:s3:listBuckets`. This will return as output the return value of the Amazon S3 listBuckets API call.
- To start a nested execution in Step Functions use the syntax: `arn:aws:states:::aws-sdk:sfn:startExecution`. You will then add StateMachineArn as a parameter. This will return as output the return value of the Step Functions nested workflow.

Para este ejercicio se crea una state machine en blanco.

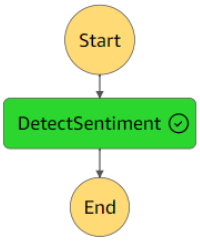


The state machine will pass the execution input's Comment value to Comprehend for sentiment analysis. arn:aws:states:::aws-sdk:comprehend:detectSentiment

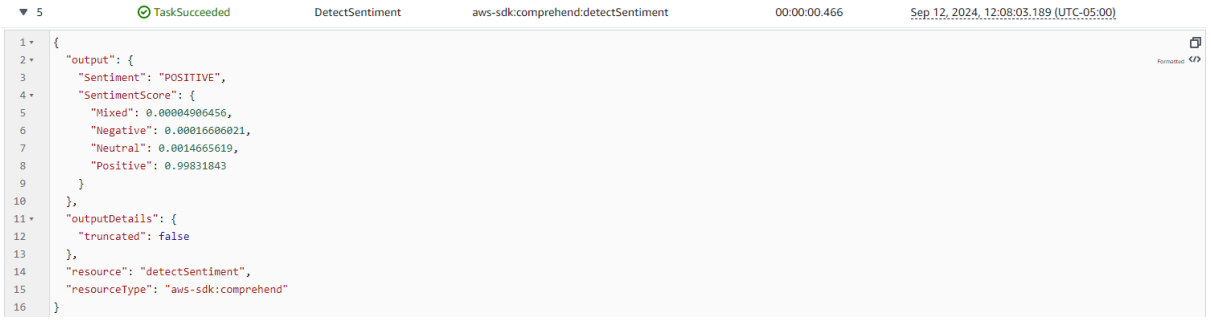
Se deja la siguiente configuración.



Se ejecuta el state machine enviando un comentario positivo.



En la sección de events se puede ver la tarea de Comprehend ejecutada con éxito y su resultado correspondiente (sentiment POSITIVE).



Exercise 6

In addition to the Task state, Step Functions provides a variety of other state elements that enable you to perform flow control options for your workflow.

These states include:

- ➢ Choice
- ➢ Parallel
- ➢ Map
- ➢ Fail or Success
- ➢ Pass
- ➢ Wait

Choice state

A Choice state adds branching logic to a state machine. In addition to most of the common state fields, `Choice` states contain the following additional fields:

- Choices (Required) - an array of Choice Rules that determines which state the state machine transitions to next.
- Default (Optional, Recommended) - the name of the state to transition to if none of the transitions in Choices is taken.

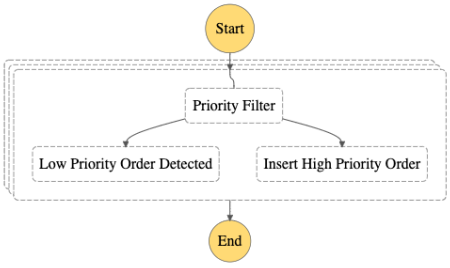
Map state

The Map state can run a set of parallel steps for each element of an input array. To configure a `Map` state, you define an `Iterator`, which is a complete child workflow execution. When a Step Functions execution enters a `Map` state, it will iterate over a JSON array in the state input. For each item, the `Map` state will execute one child workflow execution, potentially in parallel. When all child workflow executions complete, the `Map` state will return an array containing the output for each item processed by the Iterator.

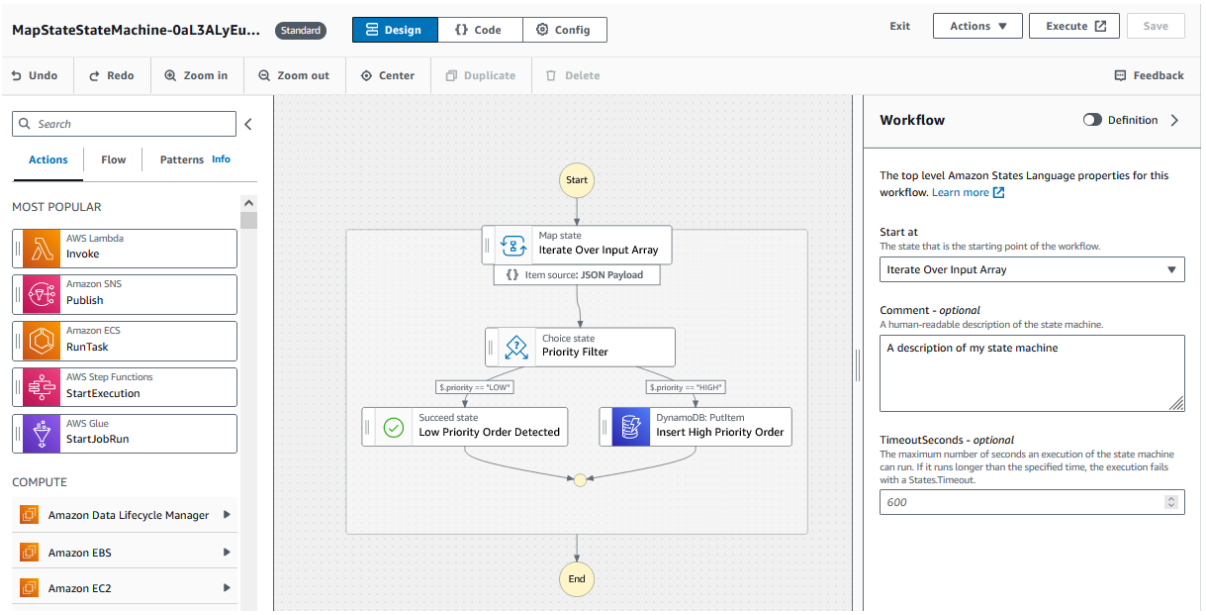
This module demonstrates dynamic parallelism using the Map and Choice states. It contains the following resources:

- ❖ One Amazon DynamoDB table
- ❖ One AWS Step Functions state machine

In this module, you'll pass in a JSON array of orders to be processed by your state machine. A Map state is used to iterate over the input array, similar to a loop in programming languages. In each iteration, the map state dynamically creates separate workflow branches. A choice state is used to determine what action to take for that item in the JSON array, similar to an if statement in programming languages. If the current item's priority field has a value "HIGH", Step Functions writes the order details into DynamoDB. If the current item's priority field is "LOW", no action is taken.



Se ingresa al state llamado BasicsMapStateStateMachine.



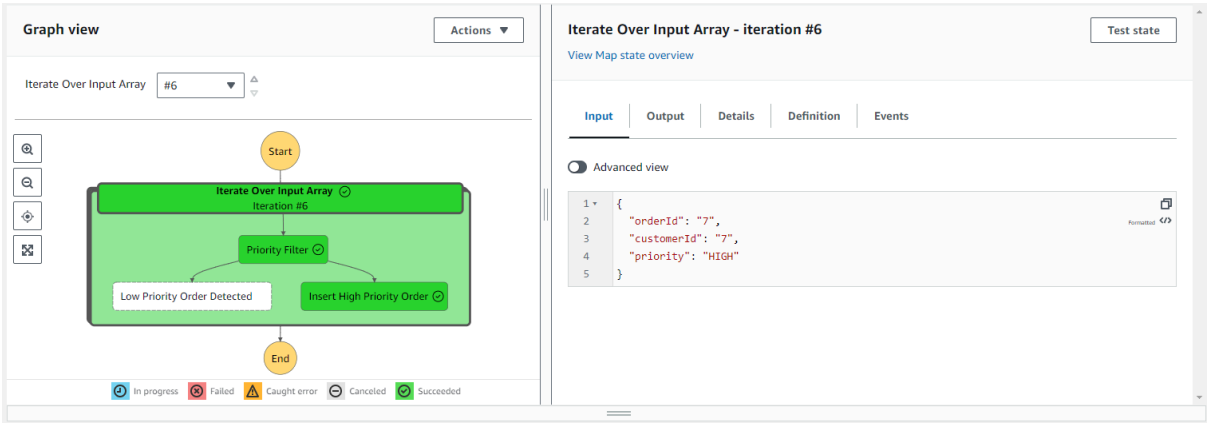
Se selecciona el Map state Iterate Over Input Array y se revisa la configuración.

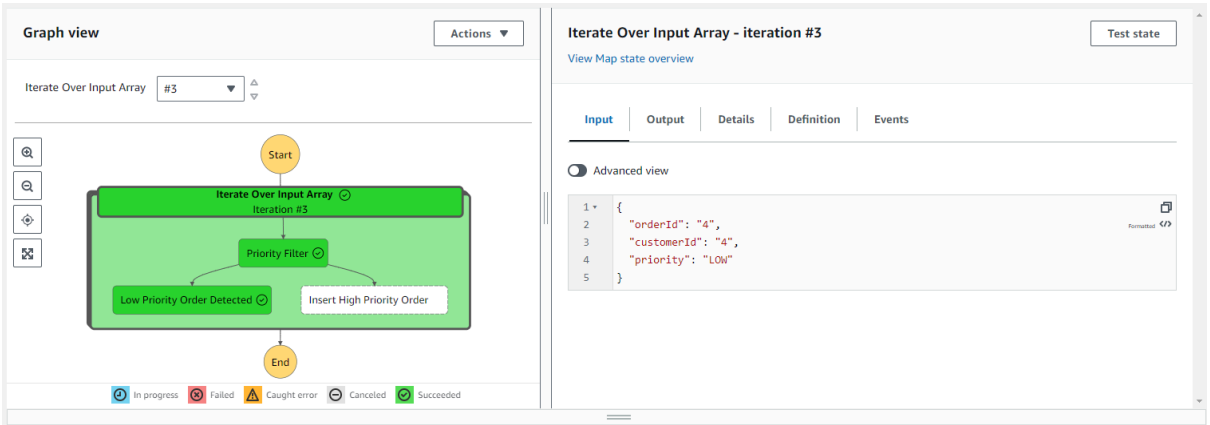
- ➔ Processing mode: Inline - This is the mode for processing relatively smaller inputs and data sources.
- ➔ Item Source - Provide a path to the items array - This setting is used to point the Map state to the specific array within the JSON input.
- ➔ Set concurrency limit - optional - This setting is used to define how many items you want to process in parallel. The default is 1, which means you'll process the data sequentially, one item at a time.

Se selecciona el Choice state Priority Filter y se visualizan las Choice Rules que se definieron.

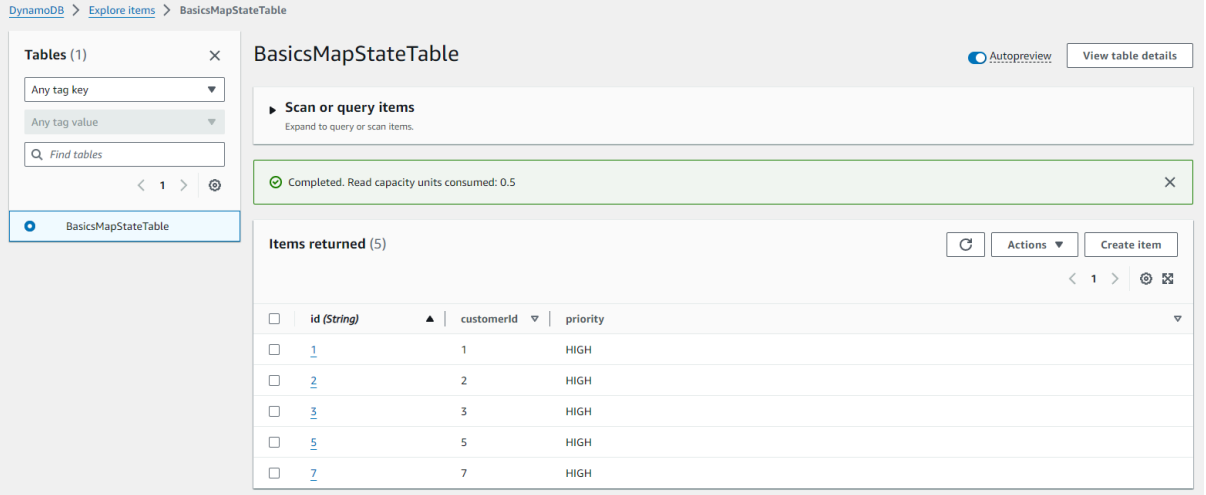
- ➔ Rule #1: $.priority == LOW. Within each item of the array, you'll check the priority value, and Step Functions will route the item to the defined state for that path. In this case, LOW priority items are routed to a success state Low Priority Order Detected, they are not written to the DynamoDB table.
- ➔ Rule #2: $.priority == HIGH. In this case, HIGH priority items go to Insert High Priority Order and are written to the DynamoDB table.

Se ejecuta el state machine. Por medio del graph view se puede ver el resultado de cada iteración. Cuando la orden tiene prioridad LOW no se realiza ninguna acción, y cuando la prioridad es HIGH se guarda el registro en la tabla BasicsMapStateTable de DynamoDB.

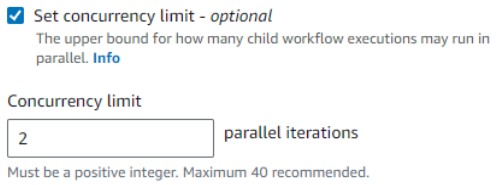




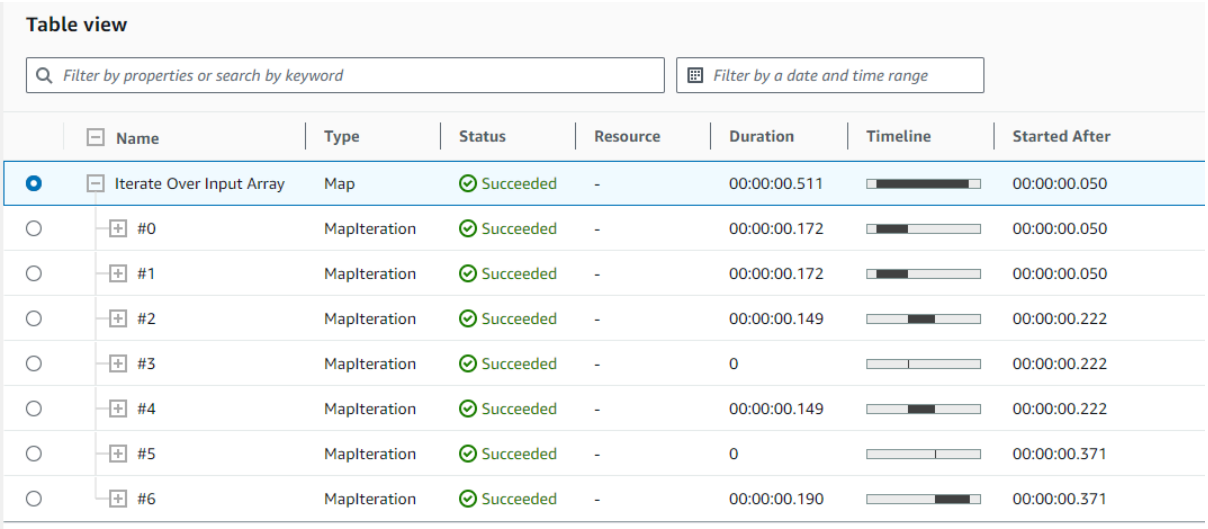
Al consultar la tabla se pueden ver los registros creados.



Ahora, para habilitar el procesamiento en paralelo, dentro de la configuración del Map state se incrementa el número de iteraciones en paralelo a 2.



Se guardan los cambios y se ejecuta nuevamente el state.



La ejecución dura menos y las iteraciones van iniciando al mismo tiempo unas con otras.

Exercise 7

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs. Step Functions integrates directly with API Gateway, enabling you to trigger the execution of a state machine with an HTTP request. API Gateway can be used with both Standard and Express workflows. Integrations may be synchronous or asynchronous.

The Parallel state can enable faster data processing by creating a fixed number of parallel branches of execution in your state machine.

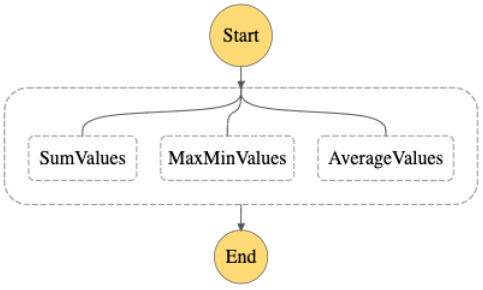
In this module you will learn about how API Gateway integrates with Express workflows via asynchronous and synchronous patterns. You will also learn how to use the Parallel state to create multiple concurrent branches of logic.

This module contains the following resources:

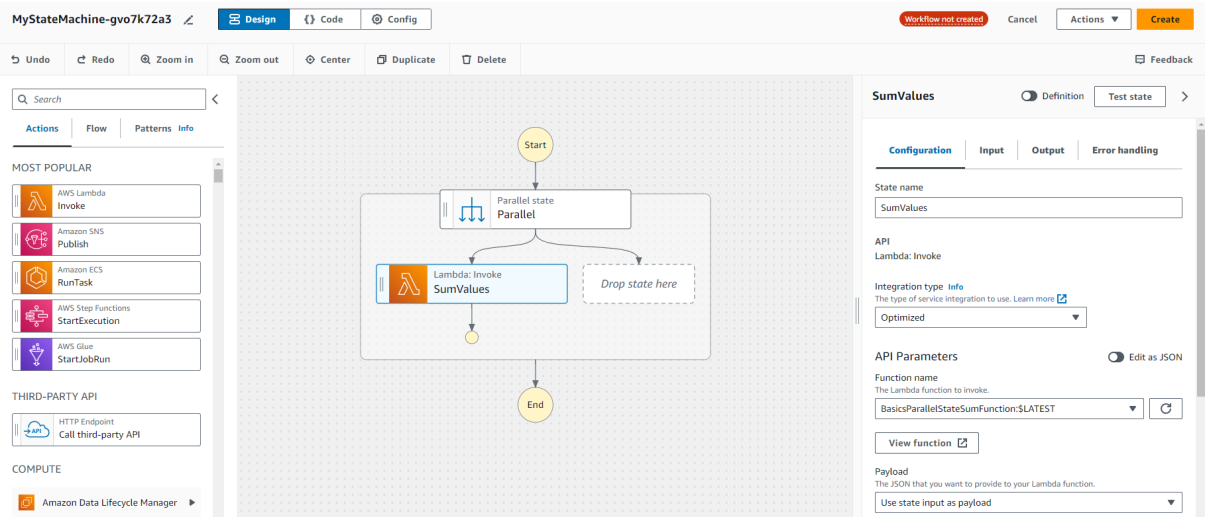
- Three AWS Lambda functions
- One API Gateway endpoint
- One AWS IAM Role used to integrate API Gateway with Step Functions

In this module, you will create an Express workflow that receives an array of integers as input, and in parallel calculates the sum, the average, and the maximum and minimum values. The state machine returns a JSON object with the responses from each of the parallel tasks.

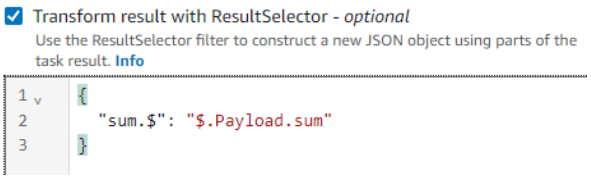
Next, you will set up an asynchronous API Gateway integration with your state machine. You will execute the state machine by sending a request to API Gateway. You will then modify the integration to make the response from Step Functions synchronous.



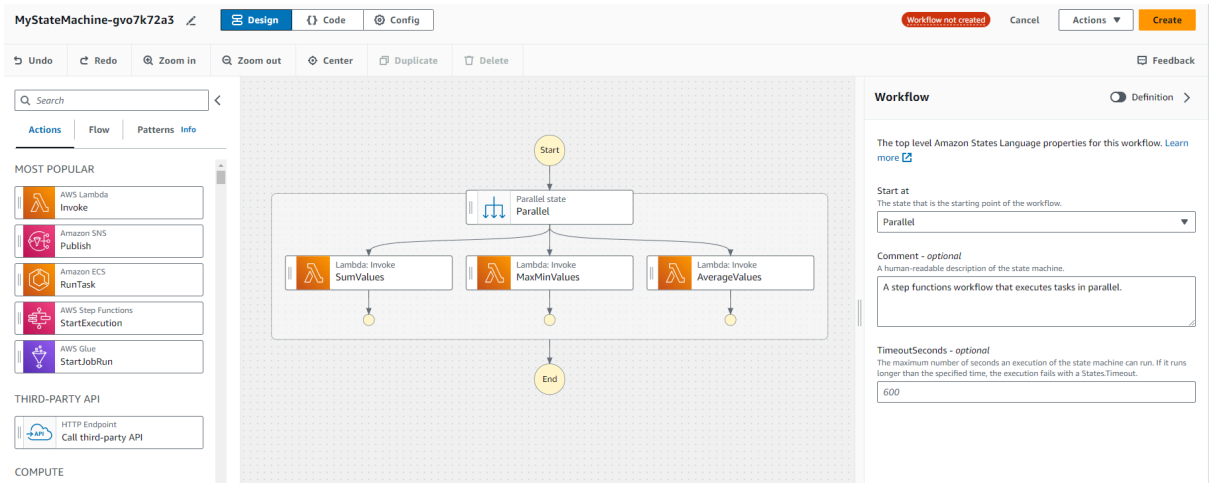
En principio se crea un state machine en blanco. Se agrega el Parallel state para poder correr las funciones Lambda en paralelo. Primero se agrega la función llamada BasicParallelStateSumFunction.



En la sección Output, solo se selecciona la opción Transform result with ResultSelector, lo demás no.

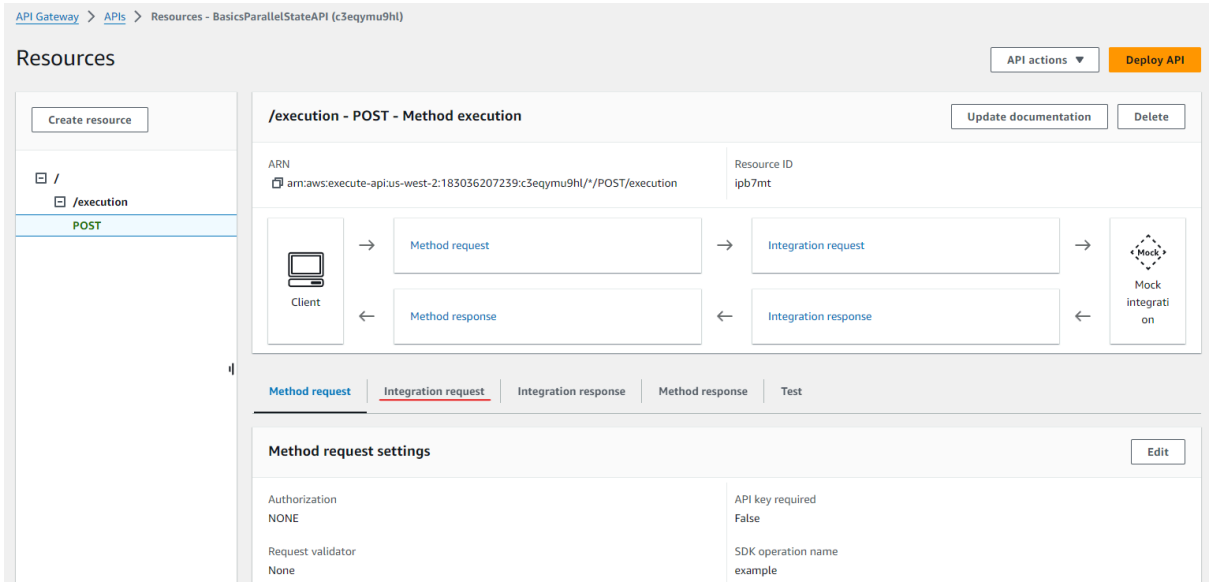


Se realiza el mismo proceso para las 2 funciones restantes: BasicsParallelStateAvgFunction y BasicsParallelStateMaxMinFunction. El resultado final es el siguiente.



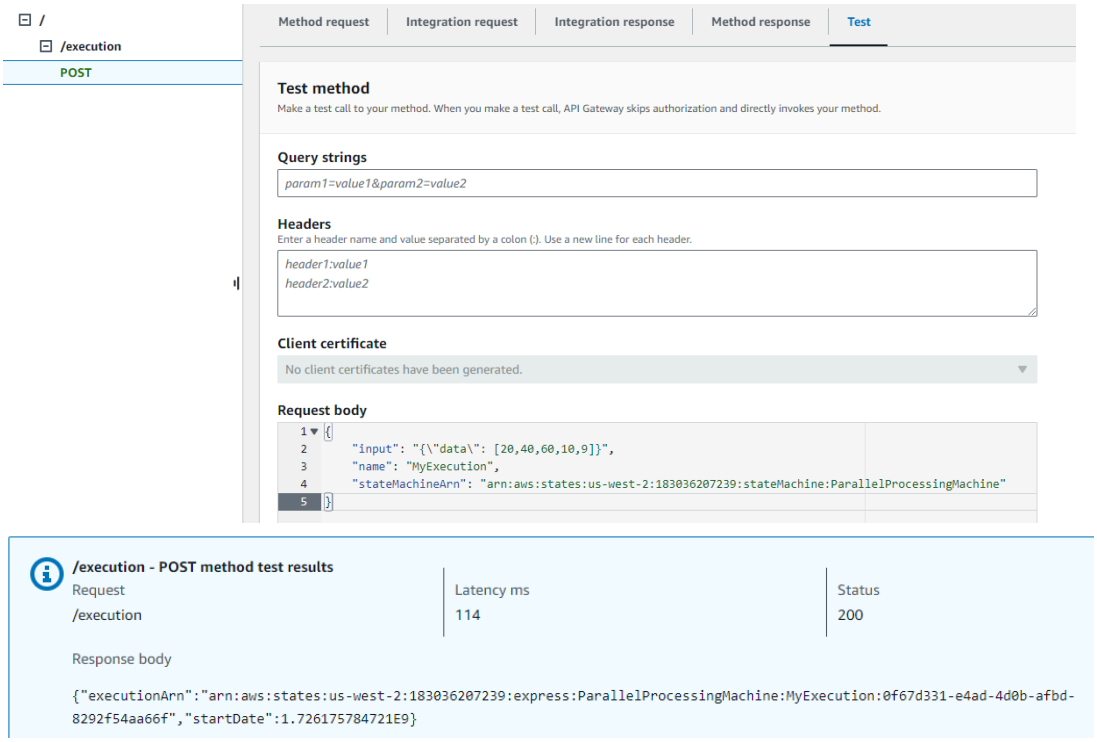
En la configuración de la state machine se asigna un nombre y el tipo Express. El rol sigue siendo el BasicsStatesExecutionRole y la config. restante se deja por defecto.

Ahora, en el servicio API Gateway, se abre el resource ya creado y se selecciona el método POST. Luego, en la sección Integration request, se selecciona la opción Edit.

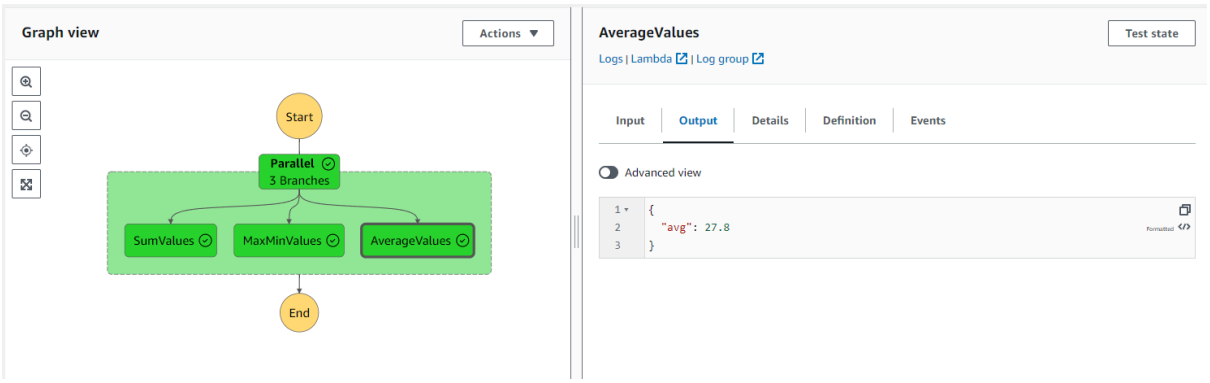


Como tipo de integración se elige AWS service. La región es la misma en donde se crean las state machines, es decir us-west-2. El servicio es Step Functions. El método HTTP requerido es POST. El tipo de acción es Action name y el nombre es StartExecution. Para el rol, se debe ir al servicio IAM y copiar el ARN del rol llamado BasicsApiGatewayExecutionRole. La config. restante se deja por defecto.

Ahora, se ejecuta un test usando data de prueba y el ARN de nuestra state machine.



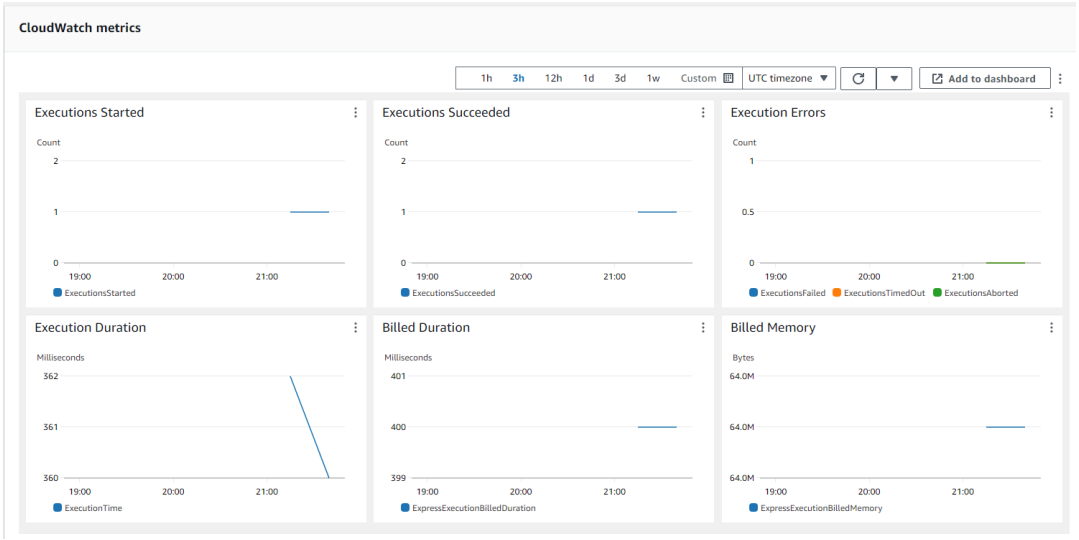
Al entrar a la ejecución se puede ver que todo concluyó correctamente.

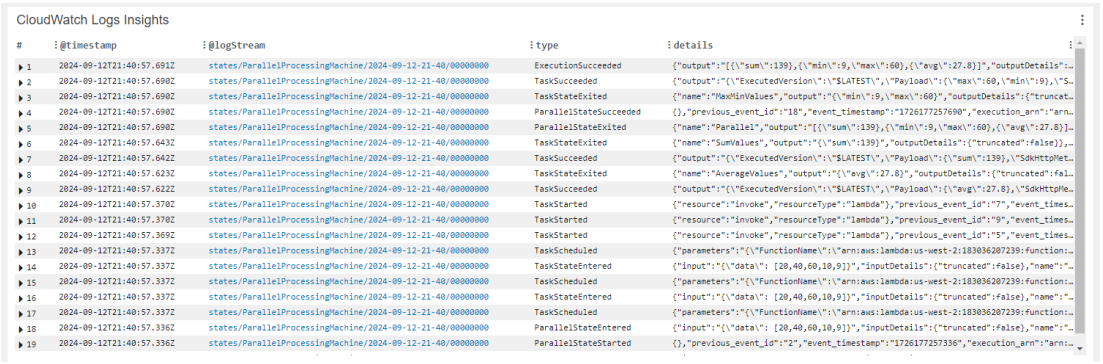


La anterior ejecución fue asíncrona. Para hacer una ejecución síncrona, se debe editar nuevamente la sección Integration request y cambiar el action name por StartSyncExecution.

Se ejecuta nuevamente el test. Ahora la respuesta tiene más detalles.

Al ser una state machine de tipo Express, se pueden ver las diferentes métricas de CloudWatch y los logs.





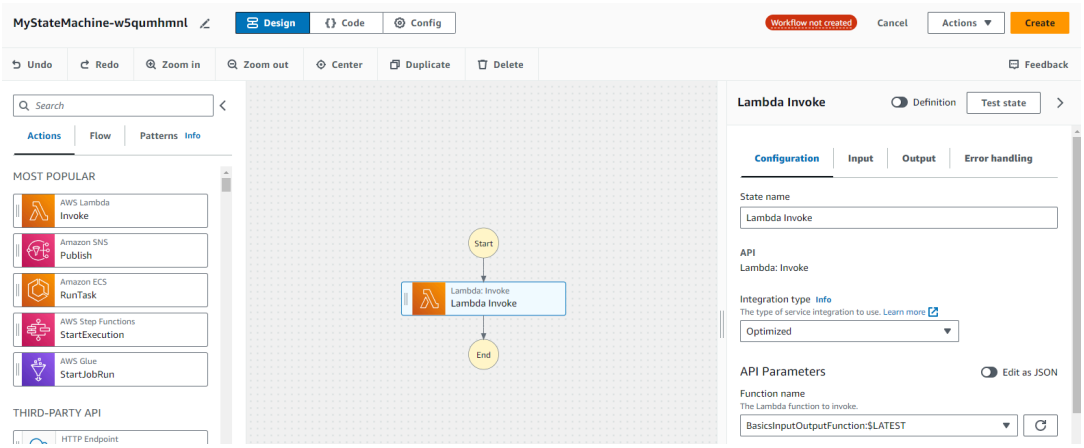
Exercise 8

A Step Functions execution receives JSON text as input and passes that input to the first state in the workflow. Each individual state receives JSON as input and usually passes JSON as output to the next state. Understanding how this information flows from state to state and learning how to filter and manipulate this data is critical to designing and implementing effective workflows in Step Functions.

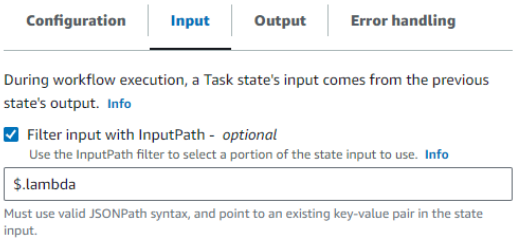
In the Amazon States Language, these fields filter and control the flow of JSON from state to state:

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

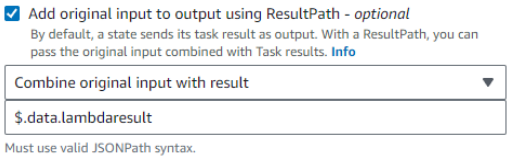
Se crea una state machine en blanco. En principio se elige una task para invocar la función lambda llamada BasicsInputOutputFunction.



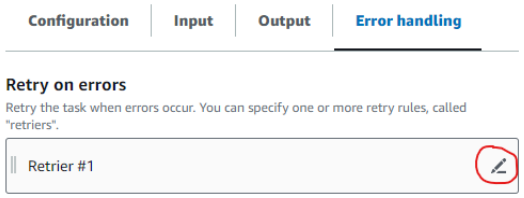
En la pestaña Input se debe seleccionar la opción Filter input with InputPath.



En la pestaña Output se debe seleccionar la opción Add original input to output using ResultPath.



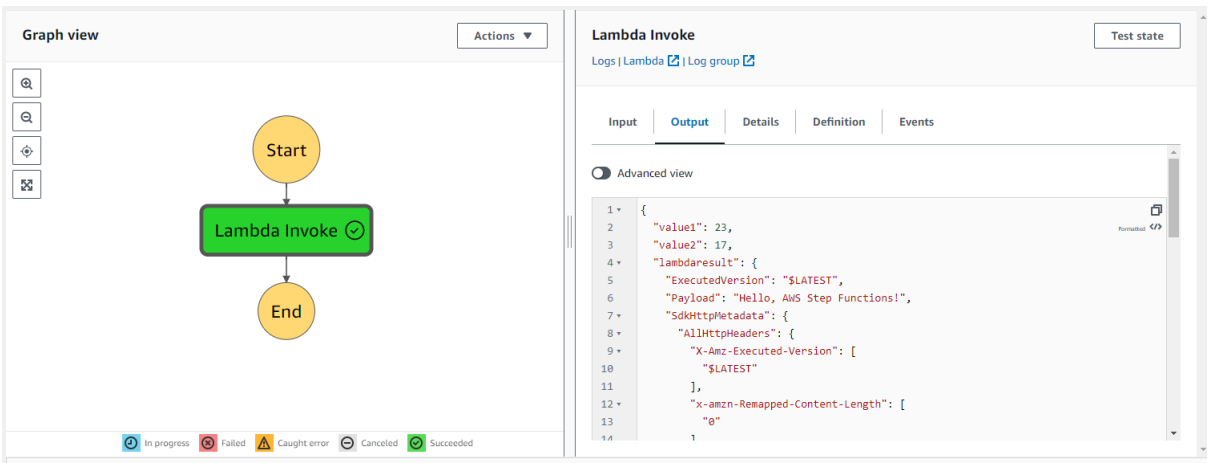
En la pestaña Error handling, se elimina el retrier que hay, usando la opción del lápiz (Editar) y luego seleccionando el botón Remove.



Luego de esto se establece la configuración del state machine. Se asigna un nombre, el tipo Standard y el rol básico. Al revisar el código ASL se pueden ver los campos InputPath y ResultPath configurados anteriormente.

```
{
  "Comment": "A step functions example showing input and output processing.",
  "StartAt": "Lambda Invoke",
  "States": {
    "Lambda Invoke": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "OutputPath": "$.Payload",
      "Parameters": {
        "Payload.$": "$",
        "FunctionName": "arn:aws:lambda:us-west-2:183036207239:function:BasicsInputOutputFunction:$LATEST"
      },
      "End": true,
      "InputPath": "$.lambda",
      "ResultPath": "$.data.lambdaresult"
    }
  }
}
```

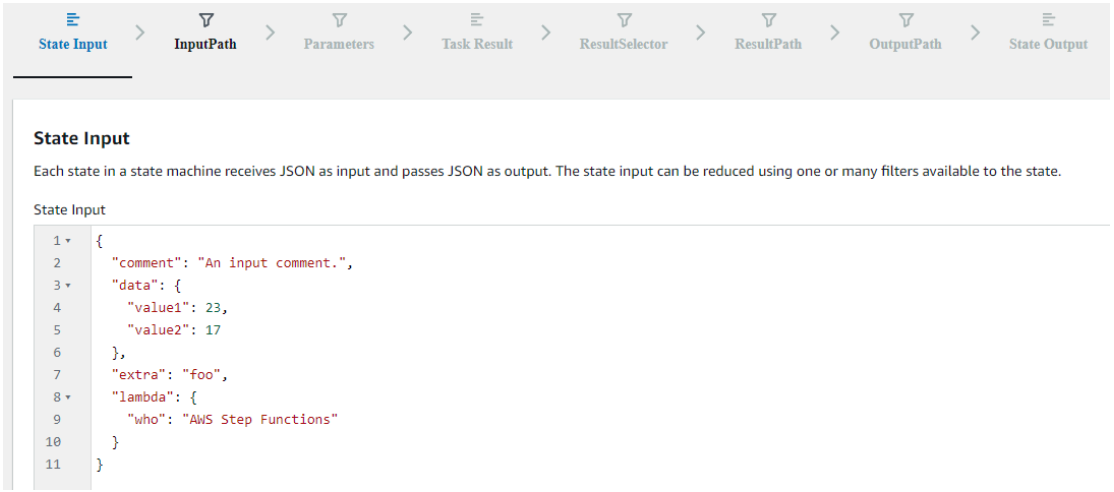
Todo se ejecuta correctamente.



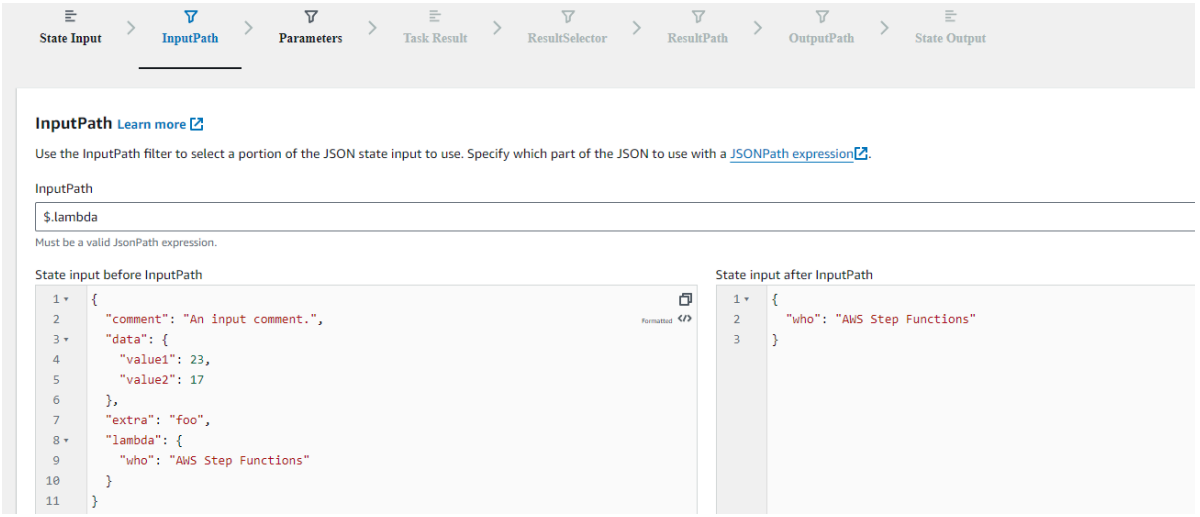
En el botón Actions se elige la opción *View dataflow simulator*.

The Data Flow Simulator allows developers to simulate the order of data processing that occurs in a single Task state during execution. This helps developers understand how to filter and manipulate data as it flows from state to state. Developers can specify a starting JSON input and evaluate it through each of the processing path stages.

The simulator starts at the State Input stage. The input field automatically validates the JSON object, and highlights any syntax errors.

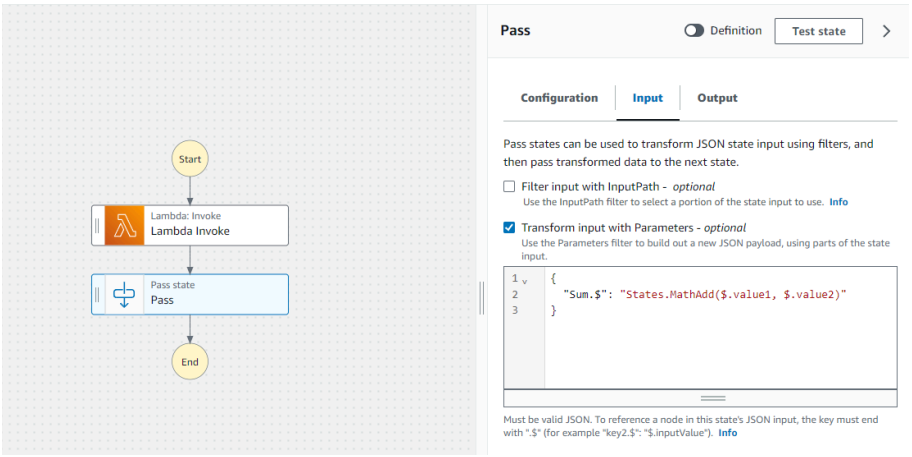


En la pestaña InputPath se cambia el valor por $.lambda.

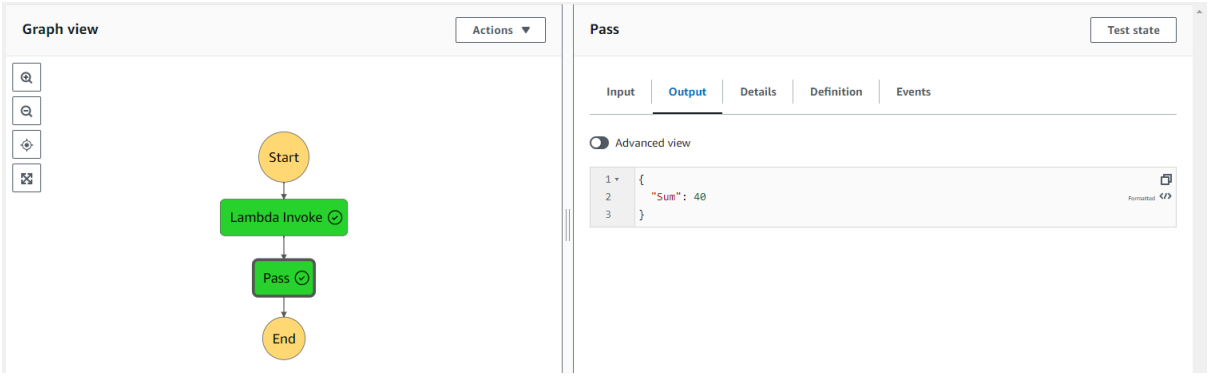


The Amazon States Language (ASL) provides several **intrinsic functions**, also known as intrinsics, that help you perform basic data processing operations without using a Task state. Examples of basic operations include math operations, creating unique ID's or merging JSON objects. You can use intrinsics like States.MathAdd, States.UUID, or States.JsonMerge in your workflow to avoid the overhead of a Lambda function to perform those operations.

En este caso, se añade un Pass state y en la pestaña Input se utiliza el state MathAdd con los valores que llegan del output de la función Lambda.

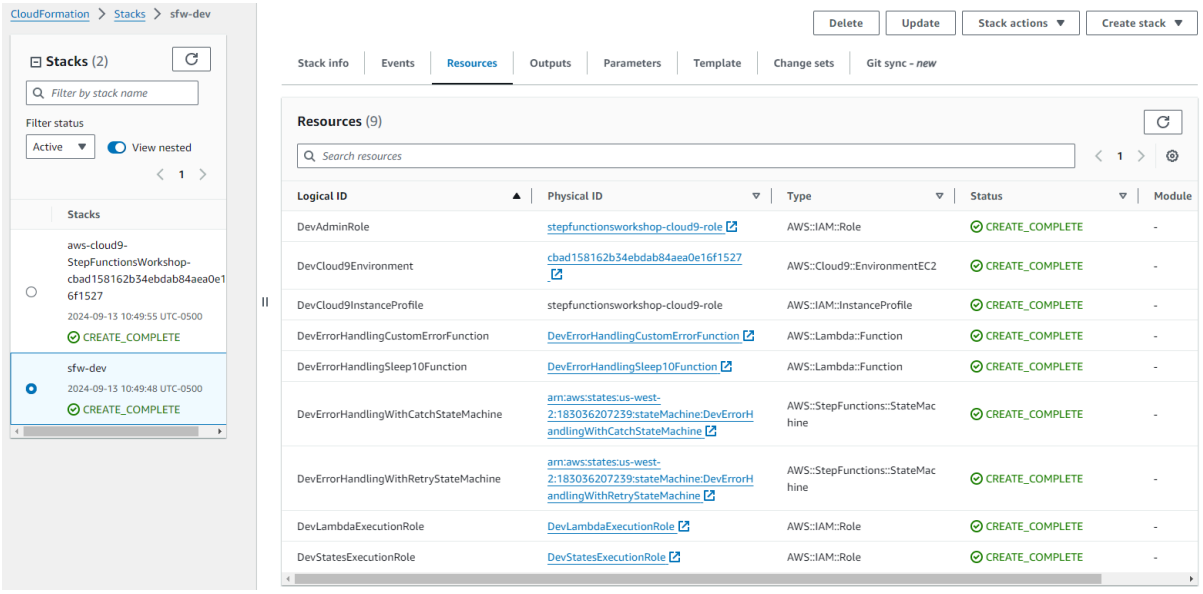


Al ejecutar todo, el output final es la suma de los valores (value1 y value2).



Development

Se toma el template de CloudFormation proporcionado por AWS (region us-west-2).



Exercise 1

Error Handling

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

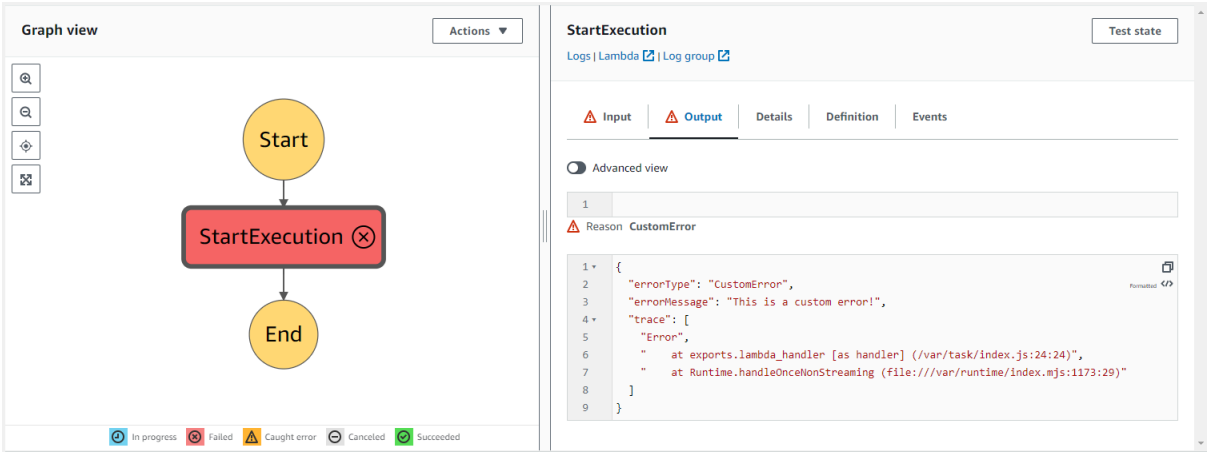
- State machine definition issues (for example, no matching rule in a Choice state)
- Task failures (for example, an exception in a Lambda function)
- Transient issues (for example, network partition events)

By default, when a state reports an error, Step Functions causes the execution to fail entirely. However, Step Functions has error handling features that enable you to retry or catch states that fail. Error handling features can define retry protocols and also catch and handle a variety of error conditions.

This module demonstrates **error handling** by using Lambda functions to simulate errors that are handled using the `Retry` and `Catch` fields.

En primer lugar, se ingresa a la función Lambda llamada DevErrorHandlingCustomErrorFunction. El código de esta función se encarga de arrojar un error. Se toma el ARN y se coloca en la state machine llamada DevErrorHandlingWithRetryStateMachine.

Al ejecutarse arroja el error correspondiente.



Ahora, para usar la funcionalidad **Retry**, se agregan los siguientes valores al ASL de la state machine.

```
{
  "Comment": "A state machine calling an AWS Lambda function with Retry",
  "StartAt": "StartExecution",
  "States": {
    "StartExecution": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-west-2:183036207239:function:DevErrorHandlingCustomErrorFunction",
      "Retry": [
        {
          "ErrorEquals": [
            "CustomError"
          ],
          "IntervalSeconds": 1,
          "MaxAttempts": 2,
          "BackoffRate": 2
        }
      ],
      "End": true
    }
  }
}
```

A continuación se explica para qué se usa cada valor.

5. Review the Error Handling Parameters. These parameters define the behavior of the `Retry`.

- ErrorEquals (Required)

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

- IntervalSeconds (Optional)

  > *An integer that represents the number of seconds before the first retry attempt (1 by default). IntervalSeconds has a maximum value of 99999999.*

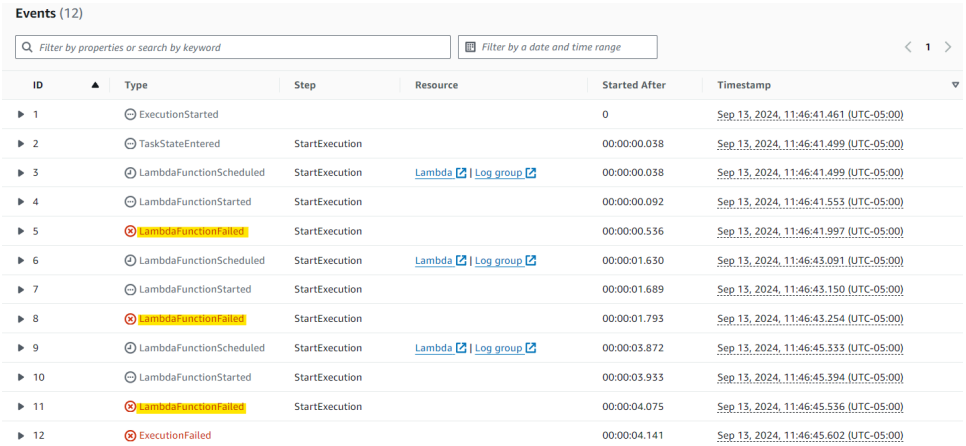
- MaxAttempts (Optional)

  > *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 or errors are never retried. MaxAttempts has a maximum value of 99999999.*

- BackoffRate (Optional)

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

Al ejecutarse nuevamente, en los Events se pueden ver los 2 reintentos.



Events (12)

Filter by properties or search by keyword | Filter by a date and time range | 1

ID	Type	Step	Resource	Started After	Timestamp
1	ExecutionStarted			0	Sep 13, 2024, 11:46:41.461 (UTC-05:00)
2	TaskStateEntered	StartExecution		00:00:00.038	Sep 13, 2024, 11:46:41.499 (UTC-05:00)
3	LambdaFunctionScheduled	StartExecution	Lambda | Log group	00:00:00.038	Sep 13, 2024, 11:46:41.499 (UTC-05:00)
4	LambdaFunctionStarted	StartExecution		00:00:00.092	Sep 13, 2024, 11:46:41.553 (UTC-05:00)
5	LambdaFunctionFailed	StartExecution		00:00:00.536	Sep 13, 2024, 11:46:41.997 (UTC-05:00)
6	LambdaFunctionScheduled	StartExecution	Lambda | Log group	00:00:01.630	Sep 13, 2024, 11:46:43.091 (UTC-05:00)
7	LambdaFunctionStarted	StartExecution		00:00:01.689	Sep 13, 2024, 11:46:43.150 (UTC-05:00)
8	LambdaFunctionFailed	StartExecution		00:00:01.793	Sep 13, 2024, 11:46:43.254 (UTC-05:00)
9	LambdaFunctionScheduled	StartExecution	Lambda | Log group	00:00:03.872	Sep 13, 2024, 11:46:45.333 (UTC-05:00)
10	LambdaFunctionStarted	StartExecution		00:00:03.933	Sep 13, 2024, 11:46:45.394 (UTC-05:00)
11	LambdaFunctionFailed	StartExecution		00:00:04.075	Sep 13, 2024, 11:46:45.536 (UTC-05:00)
12	ExecutionFailed			00:00:04.141	Sep 13, 2024, 11:46:45.602 (UTC-05:00)

Handle a failure using Catch

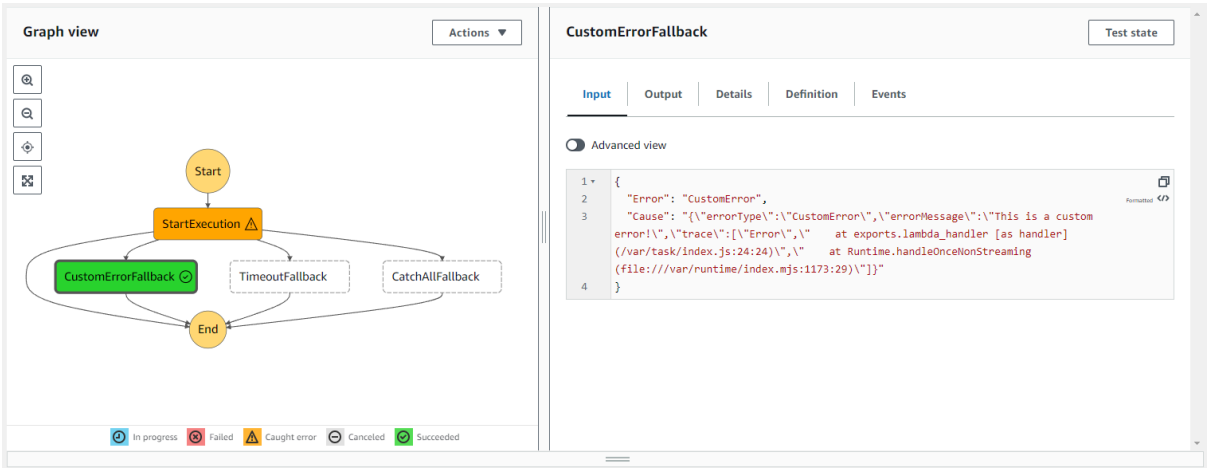
`Task`, `Map`, and `Parallel` states may contain a field named `Catch`. This field's value must be an array of objects, known as catchers. Each catcher can be configured to catch a specific type of error. ASL defines a set of built-in strings that name well-known errors, all beginning with the `States.` prefix. Catchers may also catch custom errors. Each catcher may be configured to forward to a specific **fallback** state. Each fallback state may implement error handling logic. Built-in error types include:

- `States.ALL` - a wildcard that matches any known error name
- `States.DataLimitExceeded` - an output exceeds quota
- `States.Runtime` - a runtime exception could not be processed
- `States.HeartbeatTimeout` - a Task state failed to send a heartbeat
- `States.Timeout` - a Task state timed out
- `States.TaskFailed` - a Task state failed during execution
- `States.Permissions` - a Task state had insufficient privileges

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

Se ingresa nuevamente a la función Lambda llamada DevErrorHandlingCustomErrorFunction. Se toma el ARN y se coloca en la state machine llamada DevErrorHandlingWithCatchStateMachine. Se puede ver en el ASL que se cuenta con 3 catch, los cuales se evidencian a continuación.

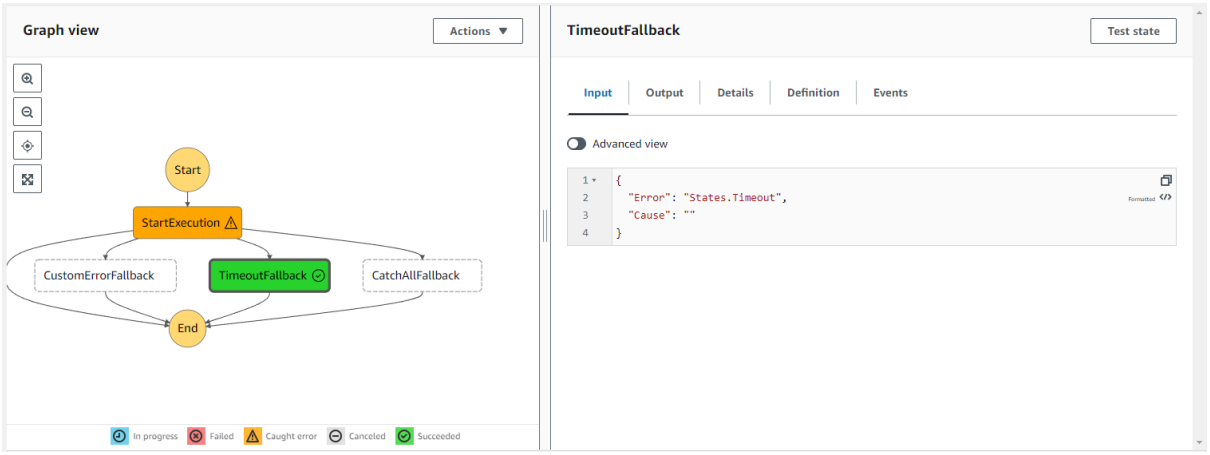
Primero está el CustomError, el cual se activa al ejecutar el state machine sin ninguna modificación.



Luego está el Timeout, para el cual se debe usar la función DevErrorHandlingSleep10Function, la cual usa la función setTimeout de JavaScript con 10 segundos. En este momento la state machine cuenta con un timeout de 5 segundos, por lo que el error se disparará.

```
"StartExecution": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-west-2:183036207239:function:DevErrorHandlingSleep10Function",
  "TimeoutSeconds": 5,
  "Catch": [
```

El resultado es el siguiente.



El CatchAll se activa en caso de que haya otro error a parte de los 2 ilustrados anteriormente.

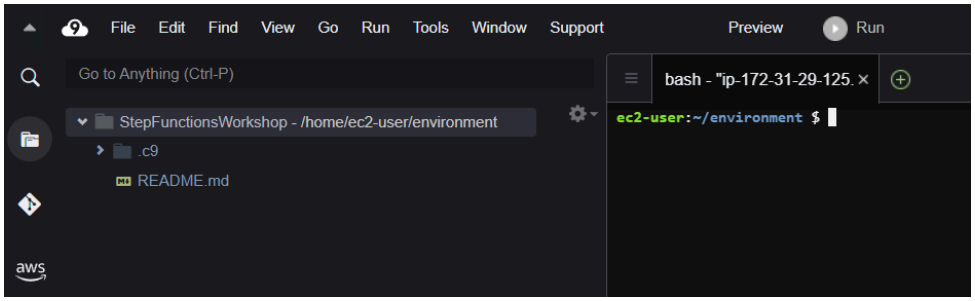
Managing State Machines with Infrastructure as Code

You can deploy your Step Functions state machine using a variety of Infrastructure as Code (IaC) options. In this workshop you're presented with three different deployment techniques:

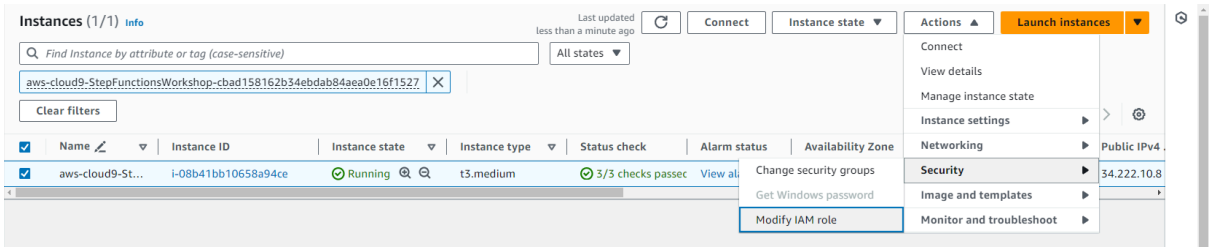
- AWS Cloud Development Kit (CDK)
- AWS Serverless Application Model (SAM)
- Terraform by HashiCorp

These three modules are somewhat similar to each other. You can choose one option to learn or you can learn all three.

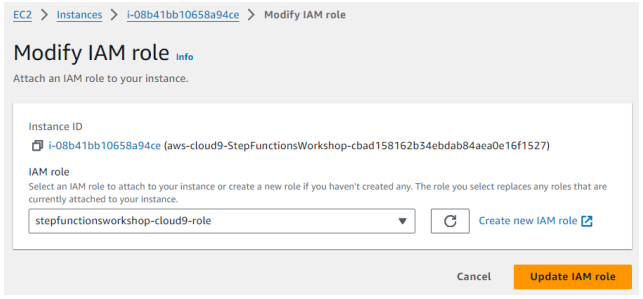
Para este ejercicio se accede al ambiente de Cloud9 creado.



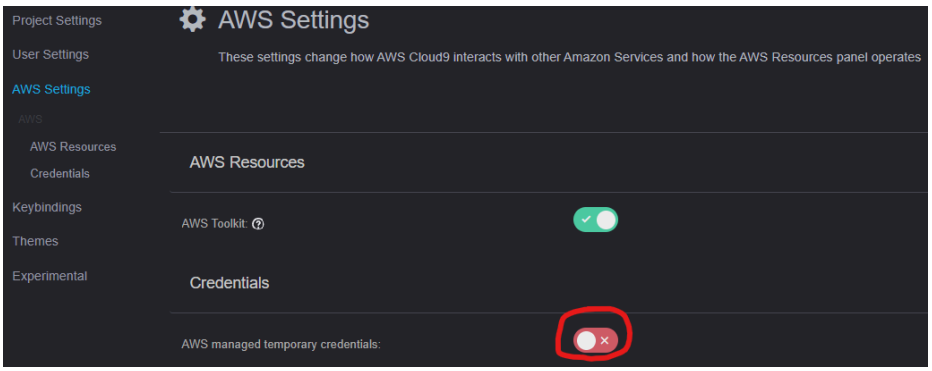
Se accede a la opción Manage EC2 Instance y dentro de la instancia se asigna el IAM role.



Se elige el rol llamado stepfunctionsworkshop-cloud9-role y se actualiza.



Ahora, se accede a AWS Settings del ambiente de Cloud9 y se deshabilita la opción de credenciales temporales manejadas por AWS.



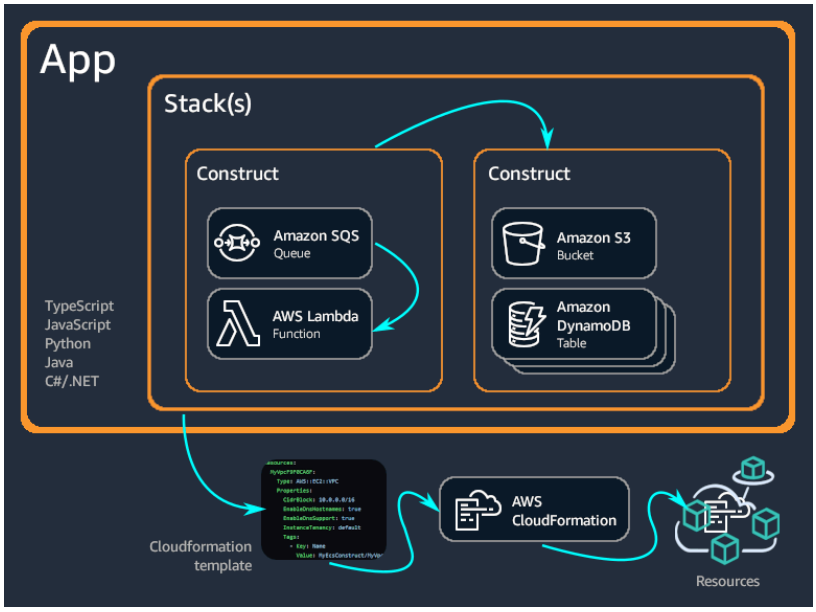
Luego por medio de comandos se realizan las siguientes acciones:

- ➔ Configure AWS CLI with the current region as default
- ➔ Save the AWS_ACCOUNT_ID and AWS_REGION to the bash_profile
- ➔ Verify the that the AWS Cloud9 IDE is configured to use the correct IAM role

```
ec2-user:~/environment $ aws sts get-caller-identity --query Arn | grep stepfunctionsworkshop-cloud9-role -q && echo "IAM role val
id" || echo "IAM role NOT valid"
IAM role valid
```

Exercise 2

The AWS Cloud Development Kit (CDK) lets you build applications in the cloud with the power of a programming language. The AWS CDK supports TypeScript, JavaScript, Python, Java, C#/.Net, and Go. Developers can use one of these supported programming languages to define reusable cloud components known as Constructs. You compose these together into Stacks and Apps.



Deploying AWS CDK apps into an AWS environment may require that you provision resources the AWS CDK needs to perform the deployment. These resources include an Amazon S3 bucket for storing files and IAM roles that grant permissions needed to perform deployments. The process of provisioning these initial resources is called **bootstrapping**. This typically needs to be done once per region in a given account.

```
ec2-user:~/environment $ cdk bootstrap aws://${AWS_ACCOUNT_ID}/${AWS_REGION}
  ⏳  Bootstrapping environment aws://183036207239/us-west-2...
Trusted accounts for deployment: (none)
Trusted accounts for lookup: (none)
Using default execution policy of 'arn:aws:iam::aws:policy/AdministratorAccess'. Pass '--cloudformation-execution-policies' to customize.
CDKToolkit: creating CloudFormation changeset...
  ✅  Environment aws://183036207239/us-west-2 bootstrapped.
************************************************
*** Newer version of CDK is available [2.158.0]  ***
*** Upgrade recommended (npm install -g aws-cdk) ***
************************************************
```

Se inicializa el proyecto CDK usando los siguientes comandos.

mkdir stepfunctions-rest-api
cd stepfunctions-rest-api
cdk init --language typescript

Use AWS CDK to create an API Gateway REST API with Synchronous Express State Machine backend integration.

En primer lugar se usa CDK para crear la state machine teniendo un Pass state.

```
const startState = new stepfunctions.Pass(this, 'PassState', {
    result: { value: 'Hello back to you!' },
})

const stateMachine = new stepfunctions.StateMachine(this, 'MyStateMachine', {
    definition: startState,
    stateMachineType: stepfunctions.StateMachineType.EXPRESS,
});
```

Luego el API Gateway correspondiente.

```
const api = new apigateway.StepFunctionsRestApi(this, 'StepFunctionsRestApi', { stateMachine: stateMachine });
```

Dentro de la carpeta creada en Cloud9 con el comando de typescript, se actualizan los archivos mencionados en el workshop. Con el comando `cdk deploy` se despliegan los servicios.

```
✅  CDKStepfunctionsRestApiStack

✨  Deployment time: 46.25s

Outputs:
CDKStepfunctionsRestApiStack.CDKStepFunctionsRestApiEndpointEBEA5F98 = https://vp151uqdz2.execute-api.us-west-2.amazonaws.com/prod/
Stack ARN:
arn:aws:cloudformation:us-west-2:183036207239:stack/CDKStepfunctionsRestApiStack/cc395fc0-7235-11ef-8b66-0239318b9859

✨  Total time: 56.24s
```

Ahora hay que ir al API Gateway creado y realizar un Test usando el payload proporcionado en el workshop. La respuesta obtenida es la siguiente, lo cual coincide con lo que se estableció con CDK.

Por último se utiliza el endpoint de API Gateway para obtener la respuesta de la state machine por consola, usando curl y enviando el mismo payload.

```
ec2-user:~/environment $ curl -X POST\
>  'https://vp151uqdz2.execute-api.us-west-2.amazonaws.com/prod' \
>  -d '{"key":"Hello Step Functions"}' \
>  -H 'Content-Type: application/json'
"Hello back to you!"
```

Se elimina el CDK project usando el comando `cdk destroy`.

AWS SAM

The AWS Serverless Application Model (SAM) is an open-source framework for building serverless applications. It provides shorthand syntax to express functions, APIs, databases, and event source mappings. With just a few lines per resource, you can define the application you want and model it using YAML. During deployment, SAM transforms and expands the SAM syntax into AWS CloudFormation syntax, enabling you to build serverless applications faster.

In this module you will use AWS SAM to deploy an application that configures API Gateway to trigger a synchronous Step Functions Express workflow.

Nuevamente se utiliza la consola de Cloud9. Aquí se crea una carpeta con tres archivos: template.yaml, api.yaml y hello_world.asl.json.

Cada archivo se actualiza con la información suministrada en el workshop. Se utilizan los siguientes comandos para hacer el despliegue de los servicios.

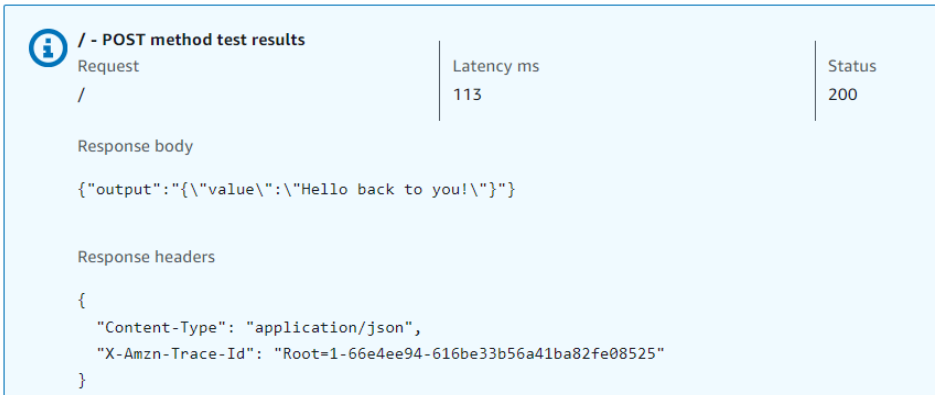
sam build
sam deploy --guided

```
CloudFormation outputs from deployed stack
-----------------------------------------------------------------------------------------------------------------------------
Outputs
-----------------------------------------------------------------------------------------------------------------------------
Key                 HelloWorldStateMachineArn
Description         Hello World State Machine ARN
Value               arn:aws:states:us-west-2:183036207239:stateMachine:HelloWorldStateMachine-hTUtnQ5T7TsN

Key                 HelloWorldStateMachineRole
Description         IAM Role created for Hello World State Machine based on the specified SAM Policy Templates
Value               arn:aws:iam::183036207239:role/api-HelloWorldStateMachineRole-5lUk9i8ZTfFW

Key                 HelloWorldApi
Description         API Gateway endpoint URL to call Hello World State Machine
Value               https://rptsajj288.execute-api.us-west-2.amazonaws.com/dev/
-----------------------------------------------------------------------------------------------------------------------------
```

Ahora hay que ir al API Gateway creado y realizar un Test usando el payload proporcionado en el workshop. La respuesta obtenida es la siguiente, lo cual coincide con lo que se estableció con SAM.



Por último se utiliza el endpoint de API Gateway para obtener la respuesta de la state machine por consola, usando curl y enviando el mismo payload.

```
ec2-user:~/environment $ curl -X POST\
>  'https://rptsajj288.execute-api.us-west-2.amazonaws.com/dev' \
>  -d '{"key":"Hello Step Functions"}' \
>  -H 'Content-Type: application/json'
{"output":"{\"value\":\"Hello back to you!\"}"}ec2-user:~/environment $
```

Se utiliza el comando sam delete para borrar todos los recursos creados.

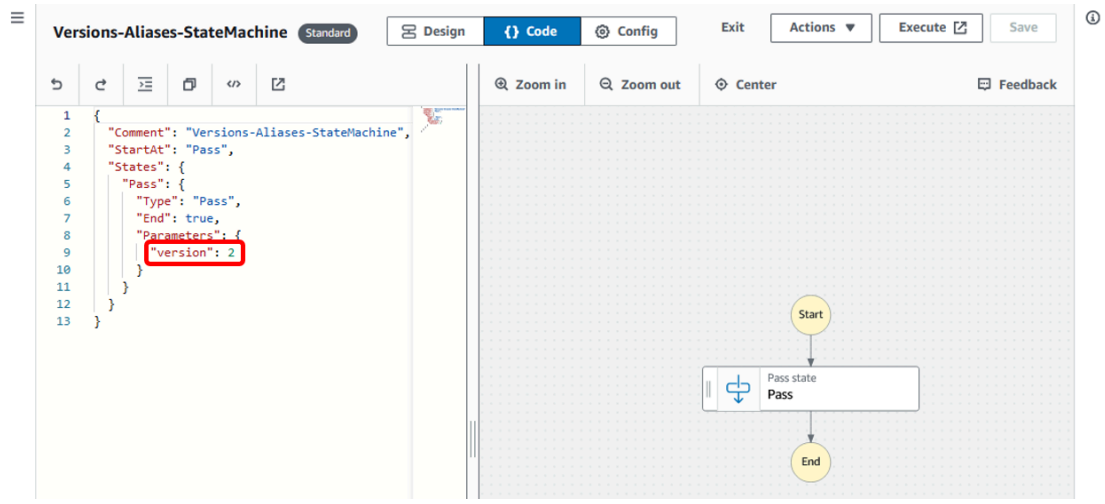
Versions and Aliases

You can use versions and aliases to manage continuous deployments of your Step Functions state machines.

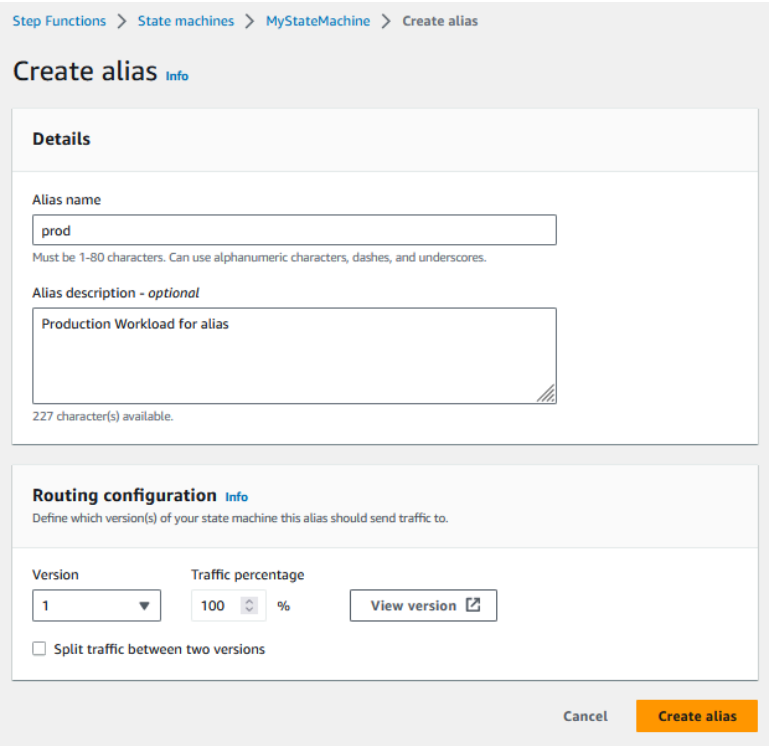
- A version is a numbered, immutable snapshot of a state machine that you can run.
- An alias is a pointer to up to two versions of a state machine.

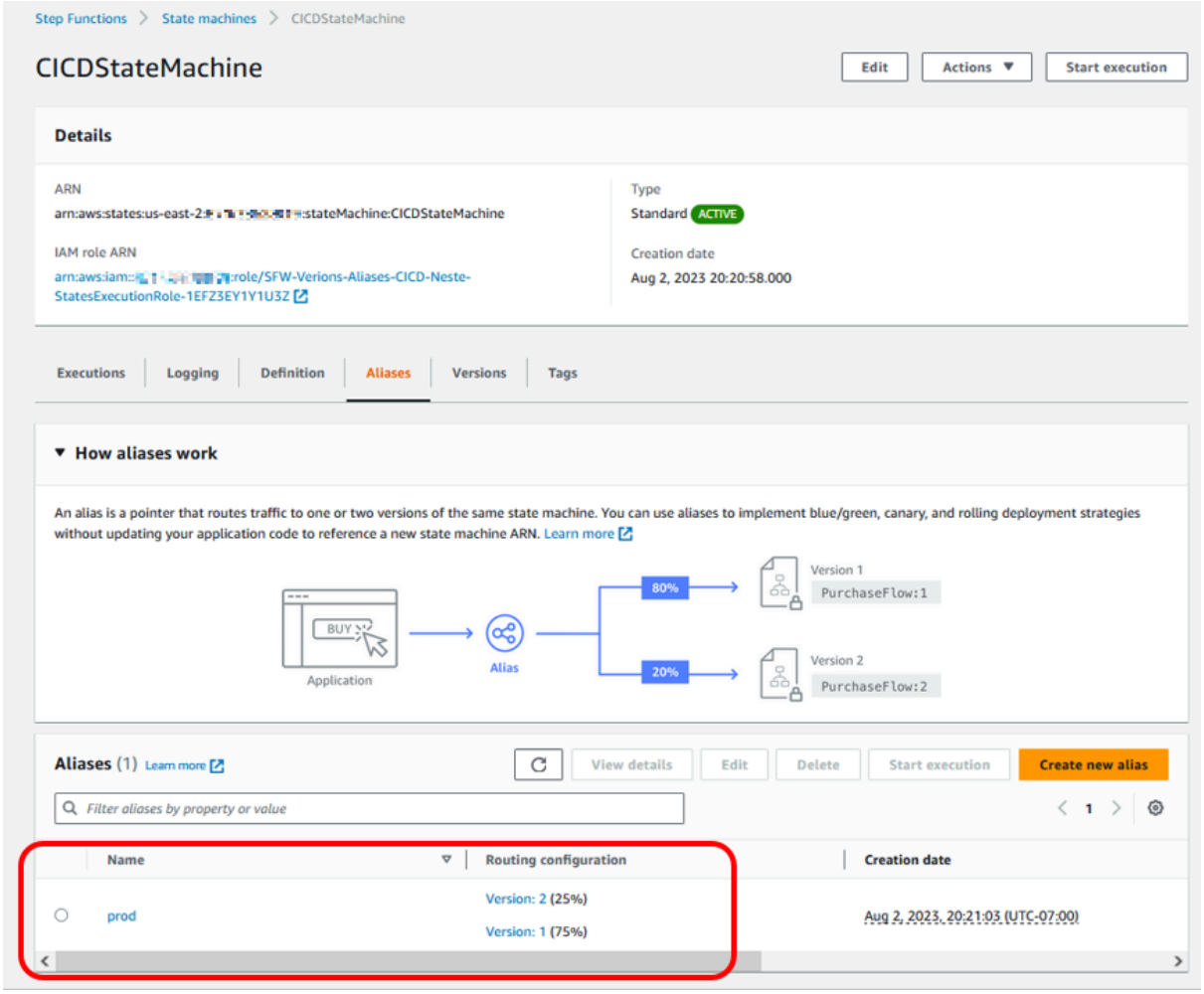
You can use gradual deployments to avoid “big bang” updates that affect all your workflow executions when you deploy new software. By using aliases, you can shift limited portions of your state machine executions to newly published versions, reducing the potential blast radius of new bugs in your code. Once you have confidence in the quality of your new version, you can shift all your executions it.

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.

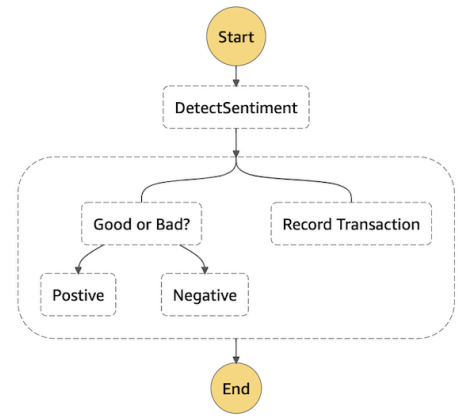



```
export STATEMACHINE_ARN="REPLACE_WITH_STATEMACHINE_ARN"
aws stepfunctions start-execution --state-machine-arn "$STATEMACHINE_ARN:1"
aws stepfunctions start-execution --state-machine-arn "$STATEMACHINE_ARN:2"
```





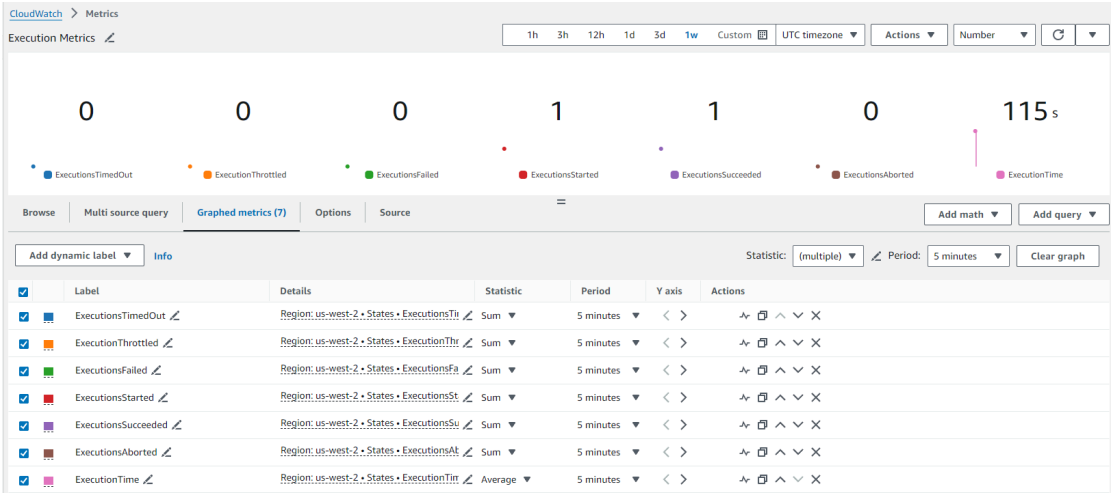
Operations



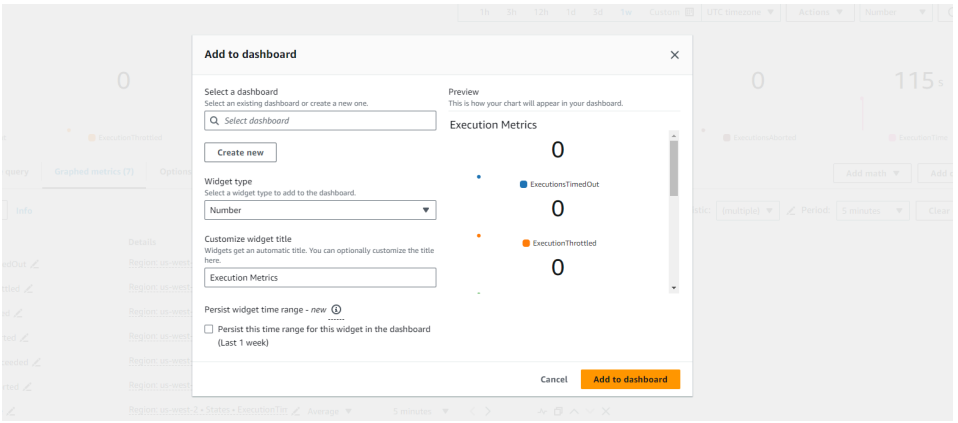
The following Step Functions Execution metrics are available in CloudWatch

- ExecutionTime
- ExecutionThrottled
- ExecutionsAborted
- ExecutionsFailed
- ExecutionsStarted
- ExecutionsSucceeded
- ExecutionsTimedOut

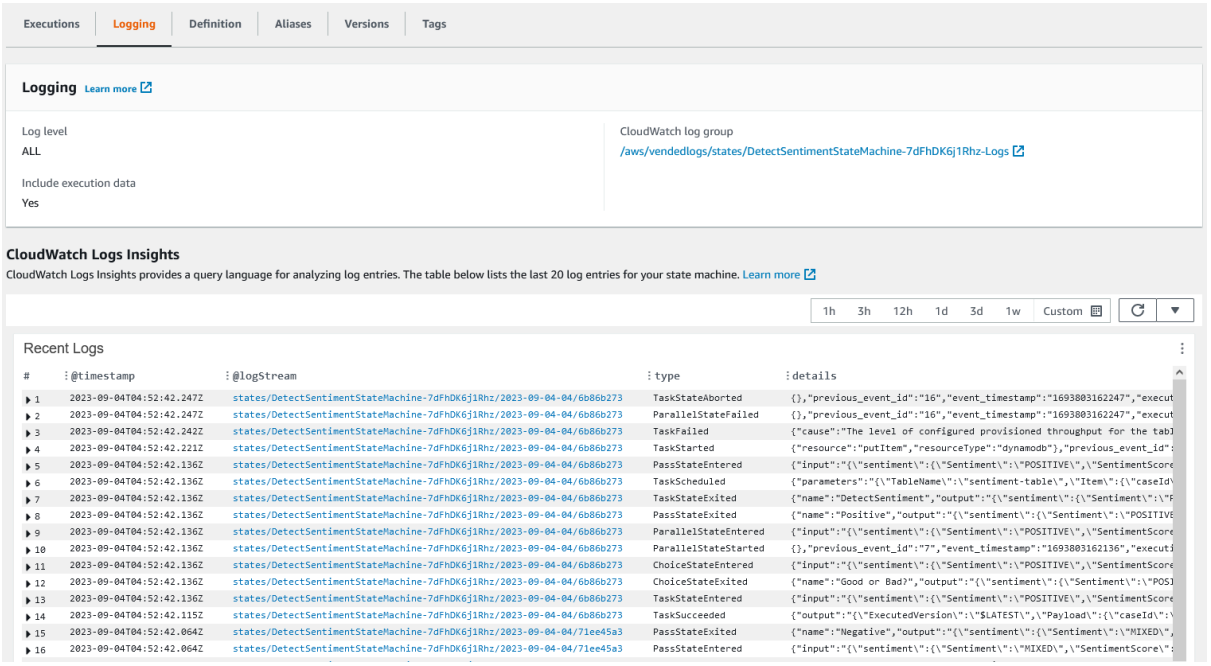
Desde la opción Metrics de CloudWatch es posible consultar las métricas de las ejecuciones de cada una de las state machines.



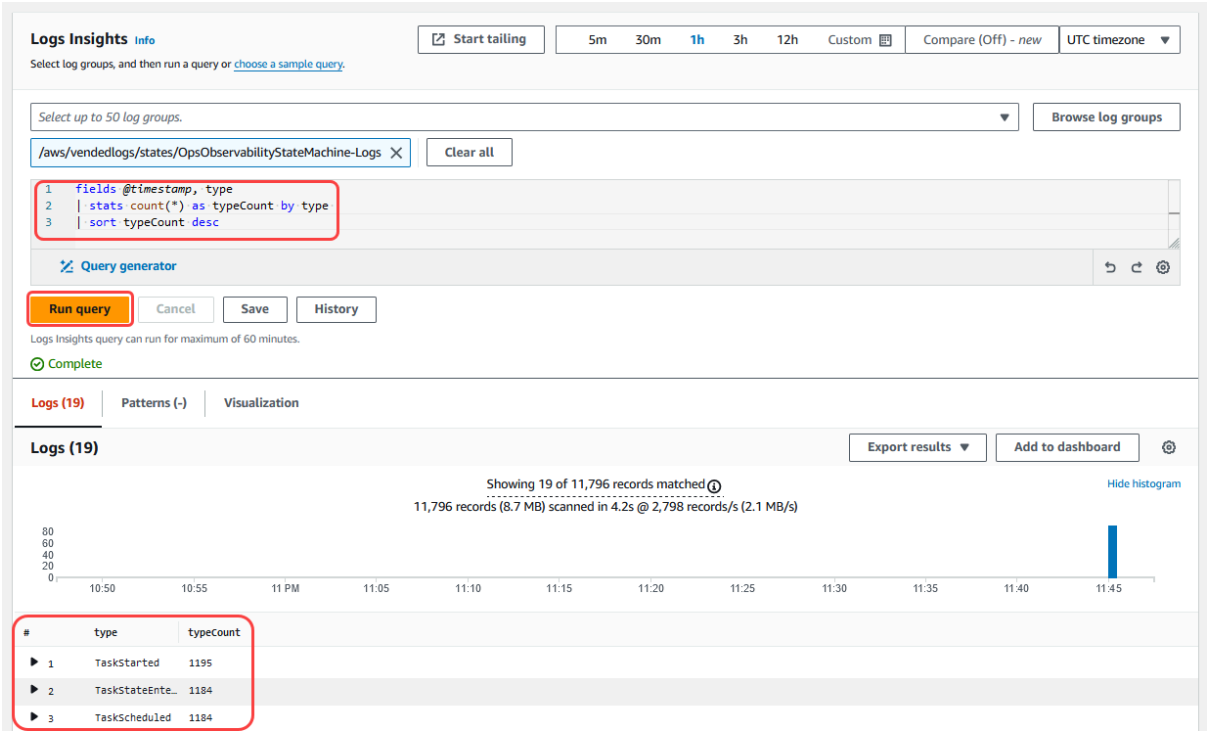
Se puede crear un dashboard a partir de estas métricas, en este caso apuntando a una sola state machine.

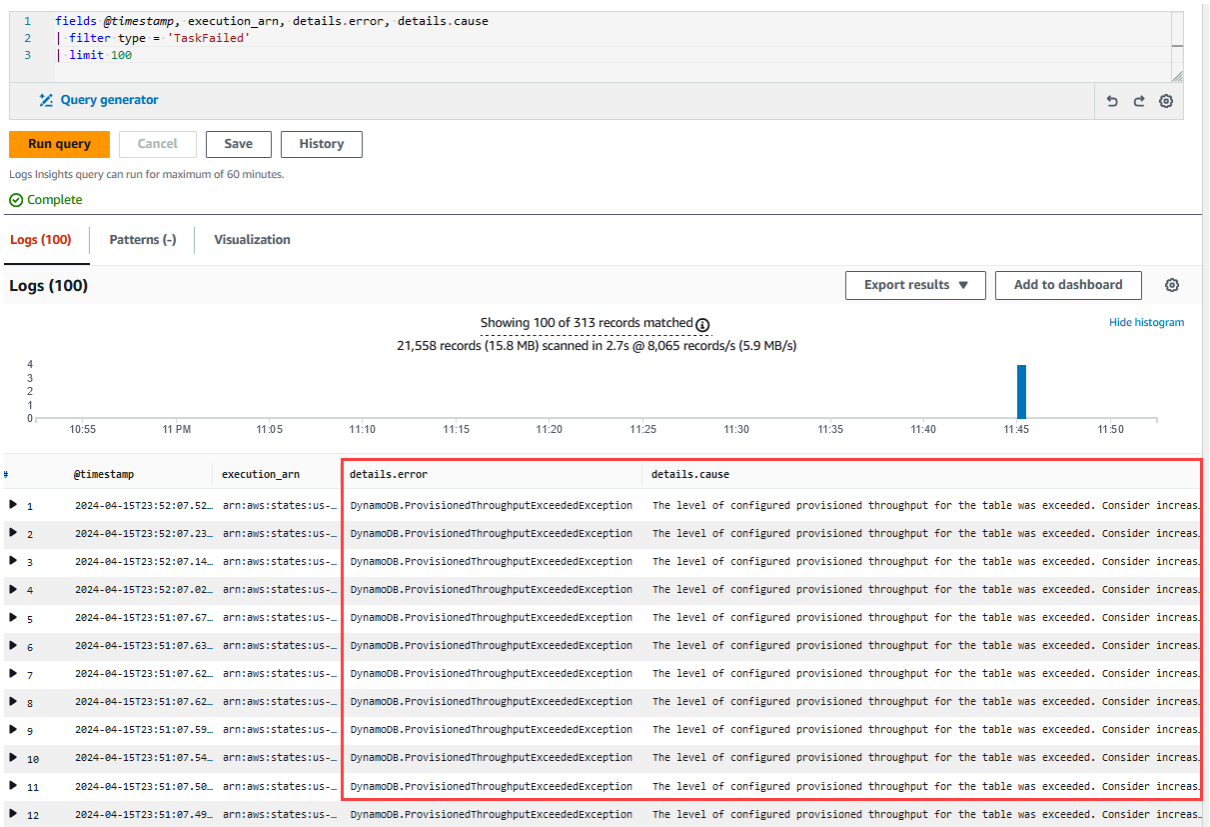


En cada state machine es posible activar el logging ingresando a la pestaña Logging.



Desde la opción Logs Insights es posible consultar los logs a través de queries.

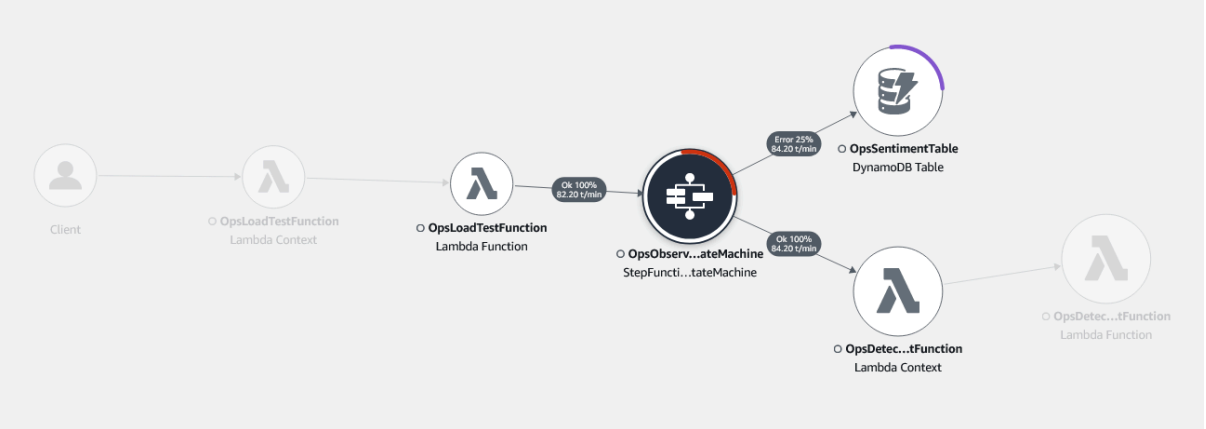




X-Ray Traces

X-Ray helps developers analyze and debug distributed applications, such as those built using a microservices architecture. With X-Ray, you can understand how your application and its underlying services are performing, and identify and troubleshoot the root cause of performance issues and errors.

This Trace Map shows that your state machine interacts with AWS Lambda and Amazon DynamoDB. The red legend on the OpsObservabilityStateMachine indicates Faults. The purple legend on the OpsSentimentTable node indicates throttling.



Traces (5)

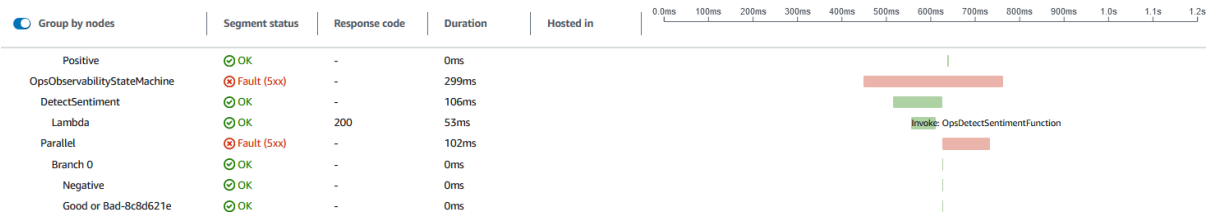
This table shows traces with updates within the last 5 Minutes, with an average response time of 0.01s. It shows as many as 1000 traces.

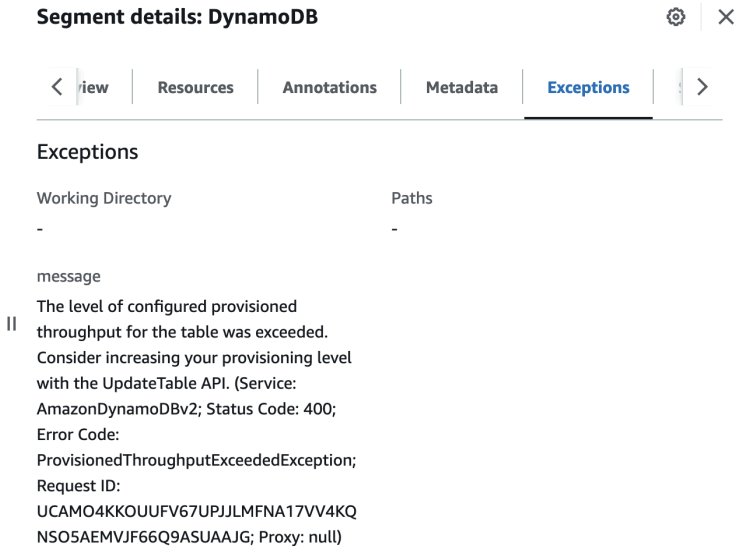
Add to dashboard

Start typing to filter trace list

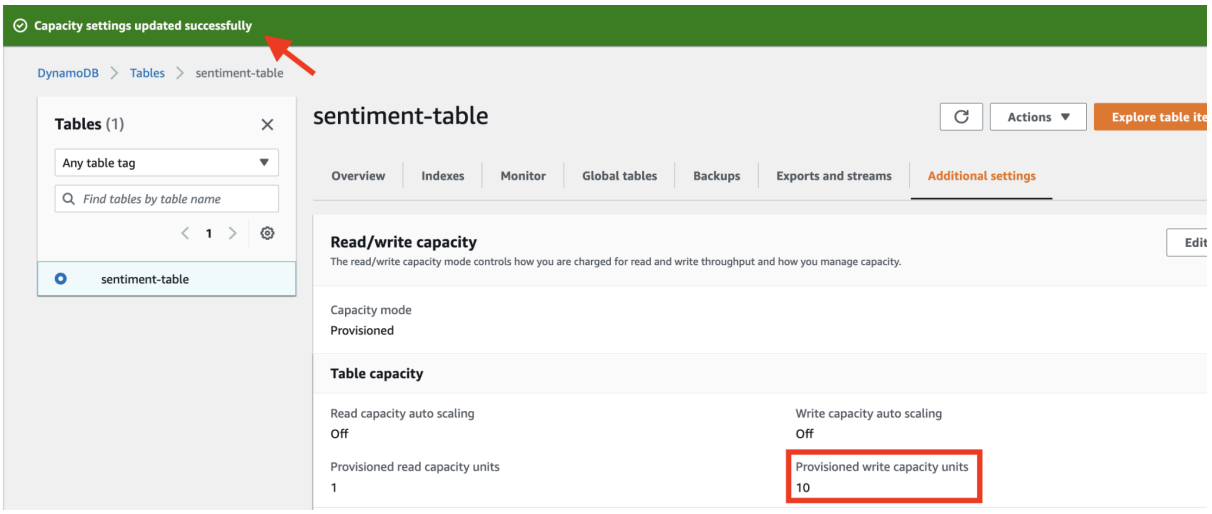
1

ID	Trace status	Timestamp	Response code	Response Time	Duration	HTTP Method	URL Address
...64f274d303789d00006756dd	OK	32.4s (2023-09-05 22:01:47)	202	0.009s	8.49s	-	-
...2f182b5c2d793e9c221e2631	OK	1.5min (2023-09-05 22:00:47)	202	0.011s	7.786s	-	-
...43721a711764be150603e516	OK	2.5min (2023-09-05 21:59:47)	202	0.013s	8.111s	-	-
...38fe0813070be9ec52d4a962	OK	3.5min (2023-09-05 21:58:47)	202	0.011s	8.11s	-	-
...572ad8050596d7e623f7a47e	OK	4.5min (2023-09-05 21:57:47)	202	0.018s	8.208s	-	-

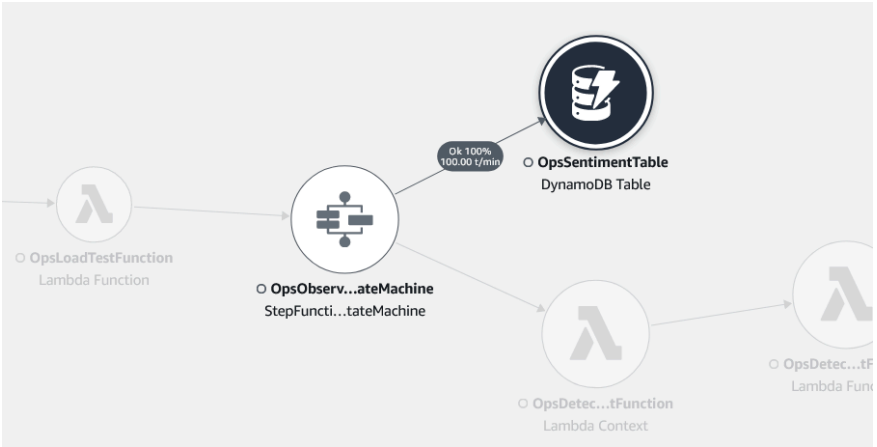




Para corregir el error en BD, se aumenta la cantidad de WCU provisionados a 10 unidades.



Ya no aparece el throttling en la table de DynamoDB.



Use Cases

Data Processing

Data processing involves discrete steps for ingesting, processing, storing the transformed data, and post-processing, such as visualizing or analyzing the transformed data. You can use Step Functions to orchestrate these steps and create scalable data processing pipelines as part of a larger business application.

Examples of Step Functions in Data Processing include:

- Orchestrate AWS Glue crawlers on an Amazon S3 bucket and execute Amazon Athena queries on the dataset.
- Quickly analyze gigabytes of data in Amazon S3 using thousands of concurrent workflows with Step Functions Distributed Map state
- Process streaming data off of Amazon Kinesis in real-time using Step Functions Express Workflows.

Se toma el template de CloudFormation proporcionado por AWS (region us-west-2).

In this module, the Step Functions workflow uses a Map state in Distributed mode to process a list of S3 objects in an S3 bucket. Step Functions iterates over the list of objects and then launches thousands of parallel workflows, running concurrently, to process the items.

The dataset used in this module is a small subset of the 37+ GB of NOAA Global Surface Summary of Day. The application code in this example finds the weather station that has the highest average temperature on the planet each month. This dataset is interesting for a few reasons:

- ➔ The data is organized by station and day. Each weather station will have a single record with averages per day.
- ➔ This example Step Function workflow will find the highest average temperature across all stations by month. That is, it answers the question: "What place on earth recorded the highest average daily temperature within a given month?"
- ➔ There are over 558,000 CSV files in the data set at over 37 GB. The average CSV file size is 66.5 KB. You'll look at a small subset of data for this workshop module.
- ➔ The CSV files are relatively simple to understand and parse.

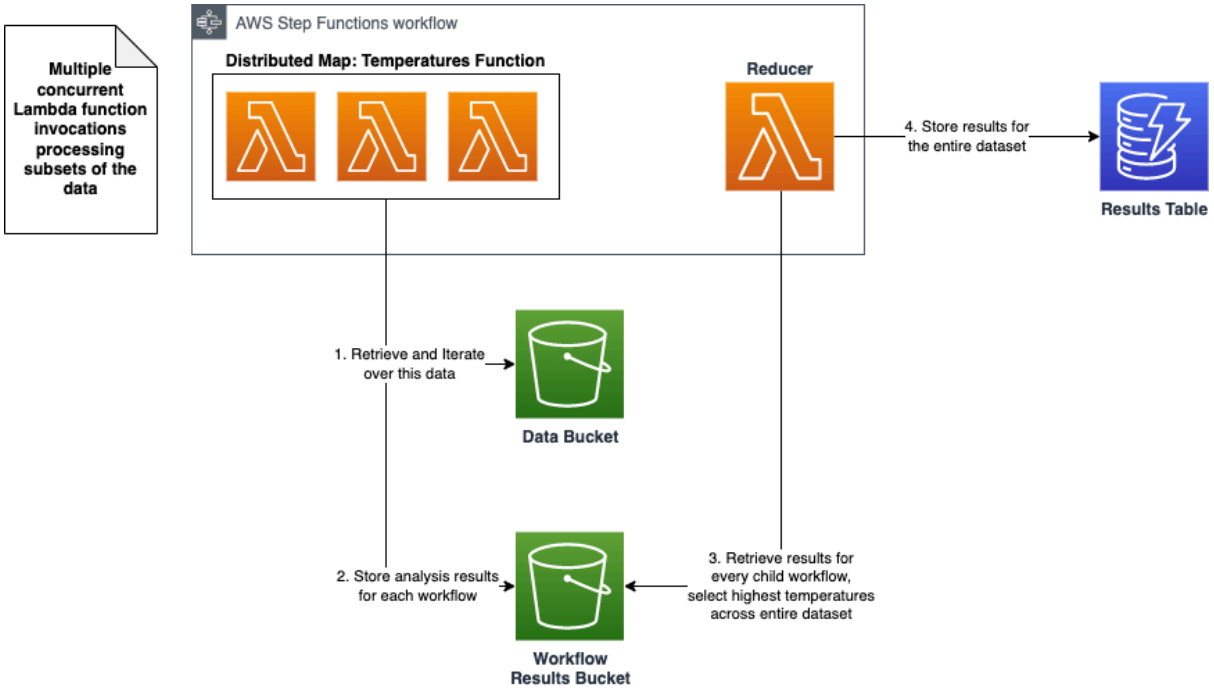
Data example

CSV File for Lerwick, UK in October 1929:

Station	Date	Name	Temp
3005099999	1929-10-02	LERWICK, UK	49.5
3005099999	1929-10-03	LERWICK, UK	49.0
3005099999	1929-10-04	LERWICK, UK	45.7

If you wanted to know the city with the highest average temperature during October 1929, you would go through each line of each city's CSV files and compare the averages for that month. Calculating averages of large datasets with gigabytes of temperature data across many locations would take hours in a single threaded process. The new distributed map state can launch up to ten thousand

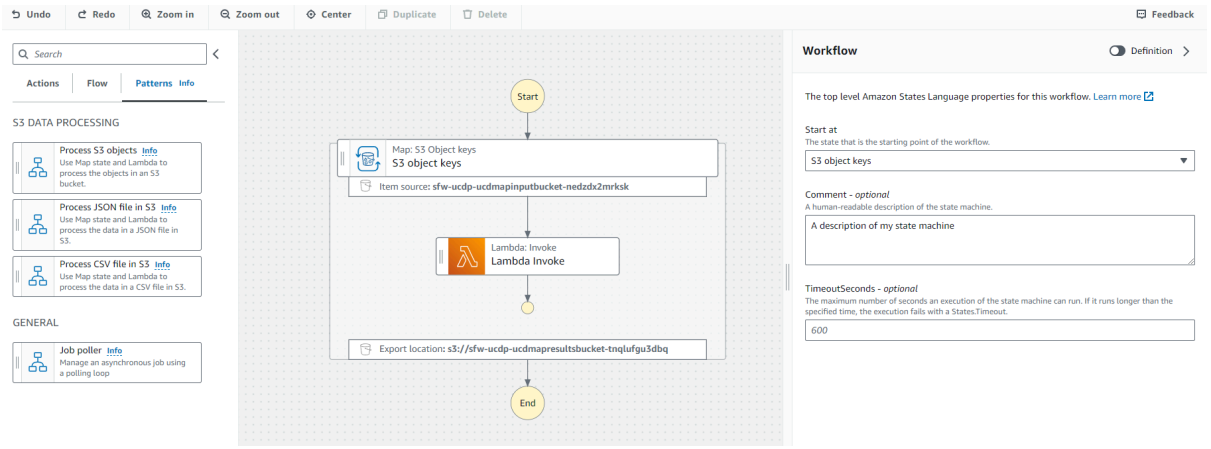
parallel workflows to iterate over millions of objects such as the CSV files in this module, logs or images stored in Amazon S3.



Data processing code has been provided in the following Lambda functions:

- TemperaturesFunction: This Lambda function is executed by each child workflow execution of the state and calculates the highest average temperatures by location and city for a subset of the data.
- ReducerFunction: This Lambda function will compare the results of each execution of the TemperaturesFunction. For example, child workflow execution 1 may find that Los Angeles, California, USA had the highest temperature on "1931-07" (July, 1931) while child workflow execution 2 finds that Cairo, Egypt had the highest temperature on "1931-07". The reducer function will take a final pass through the outputs from all of the child workflows to find the correct highest temperature for the entire dataset.

Se crea una nueva state machine en blanco y se elige el pattern llamado Process S3 objects.

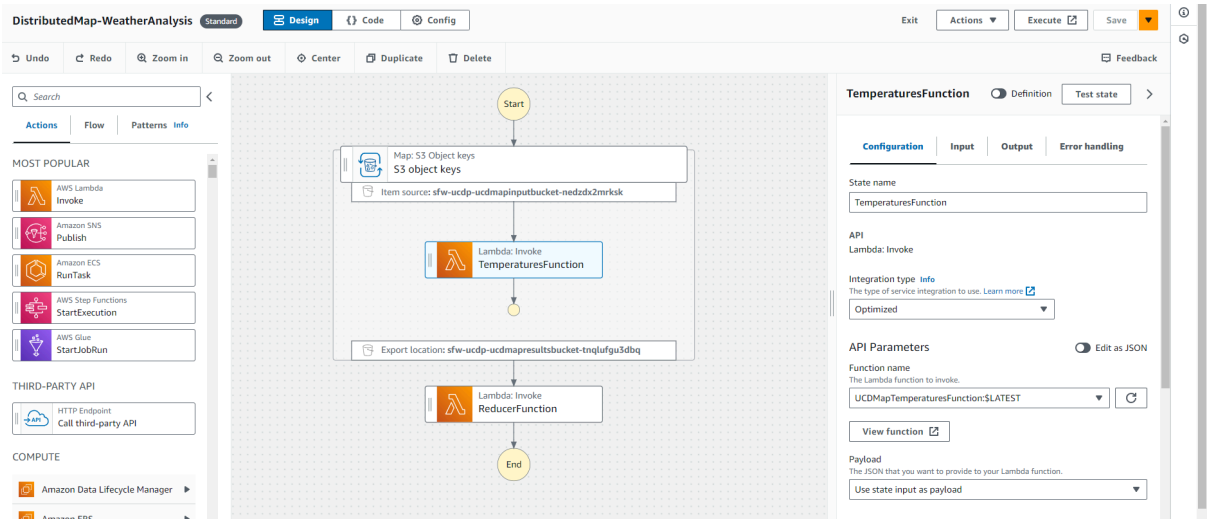


Se utiliza la siguiente configuración.

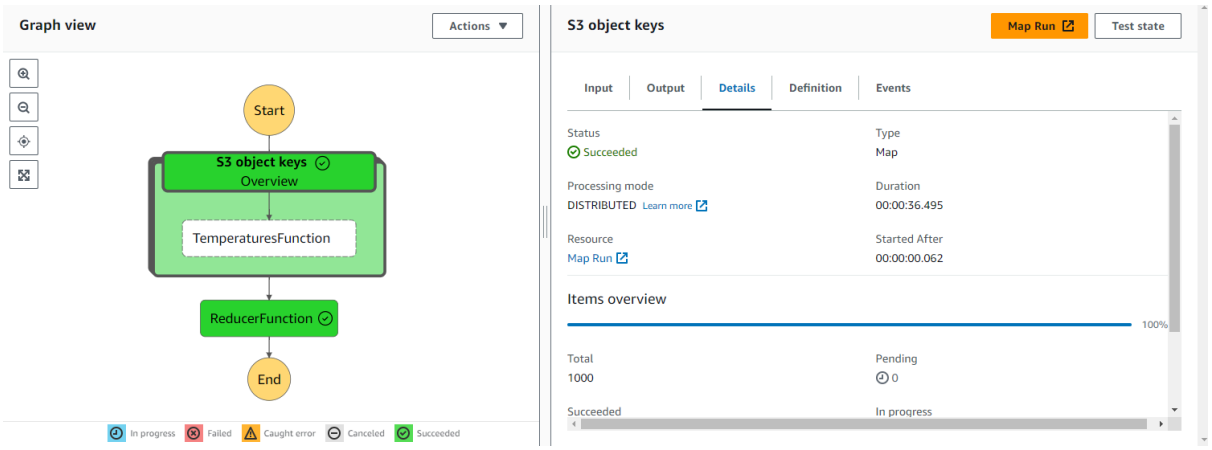
Setting	Value	Notes
Processing mode	Distributed	Map State in distributed mode runs child workflow executions for each map iteration to achieve up to 10K concurrent workflows
Item source	Amazon S3	
S3 item source	S3 object list	Since you have multiple CSV files you want to analyze, you will use this item source as it uses S3 ListObjects to get a list a paginated list of objects in the bucket.
S3 bucket	Use the *ucdmapinputbucket* bucket.	You can find the S3 bucket name in the CloudFormation stack resources tab for this module or use the *Browse S3 or enter S3 URI option* and look for *ucdmapinputbucket* with the Browse S3 button.
Enable batching	Check this box	
Max items per batch	500	Define the number of items to be processed by each child workflow execution
Set concurrency limit	500	The Lambda burst concurrency limit varies by region. You can modify this concurrency setting based on the capacity of your downstream systems.
Child execution type	Express	Given each of these child workflow executions only take a few seconds to run, you can use Express workflows.
Set a tolerated failure threshold	Expand Additional configuration to see this setting. Check this box	
Tolerated failure threshold	5%	Use this setting to consider a job *failed* if a minimum threshold of child workflow executions failed. This is useful if you have inconsistencies in your dataset.
Use state name as label in Map Run ARN	Check this box	
Export Map state results to Amazon S3	Check this box	
S3 Bucket	Use the *ucdmapresultsbucket* bucket.	Use *Enter bucket name and prefix option*. You can find the S3 bucket name in the CloudFormation stack resources tab for this module or look

		for the *ucdmapresultsbucket* bucket with the Browse S3 button.
Prefix	results	Append the "results" prefix at the end of the S3 URI i.e. s3://*ucdmapresultsbucket* bucket/results

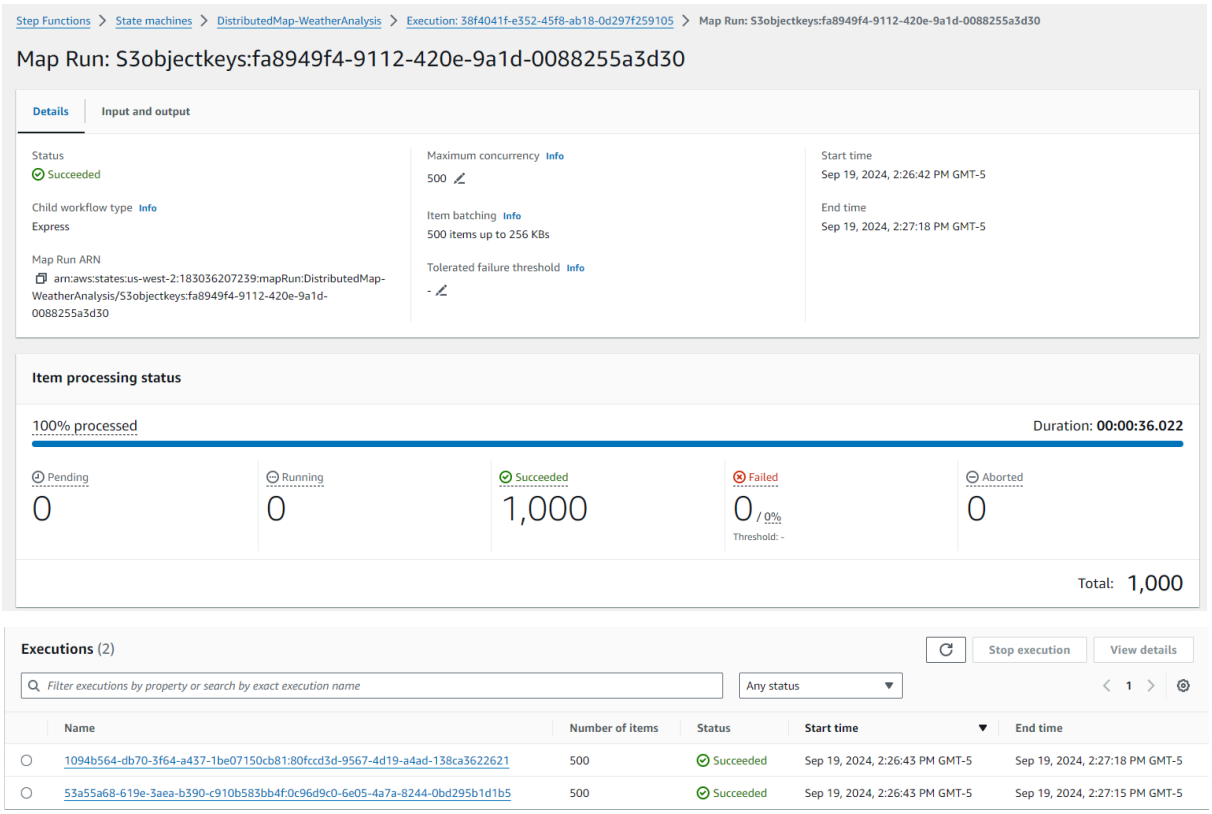
A su vez, se colocan las 2 funciones Lambda. En la configuración se asigna el rol UCDMapStateMachineRole.



Al ejecutarse se puede ver lo siguiente.

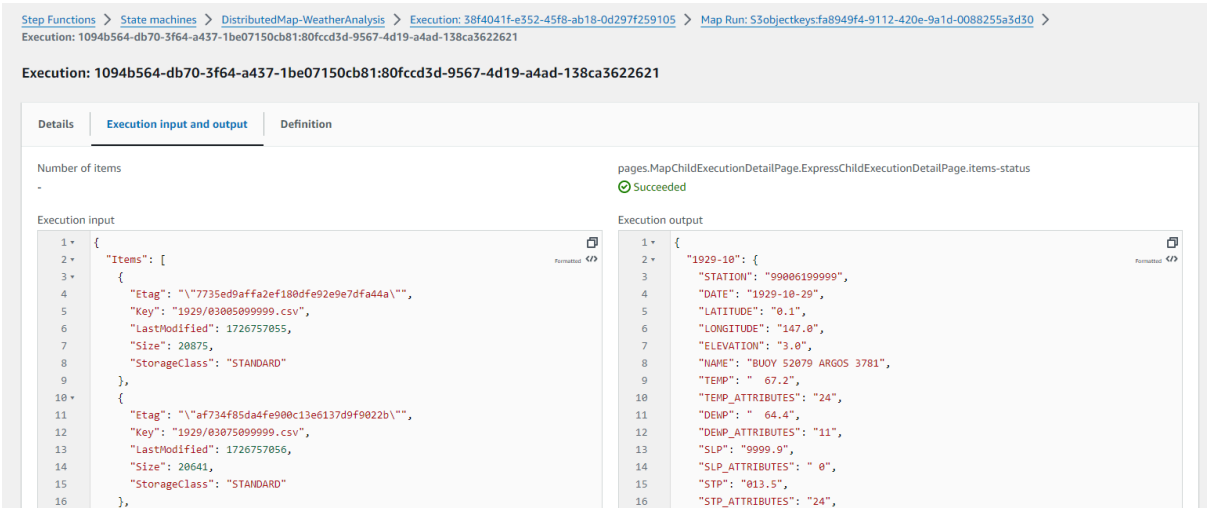


Se selecciona la opción Map Run en la pestaña Details.

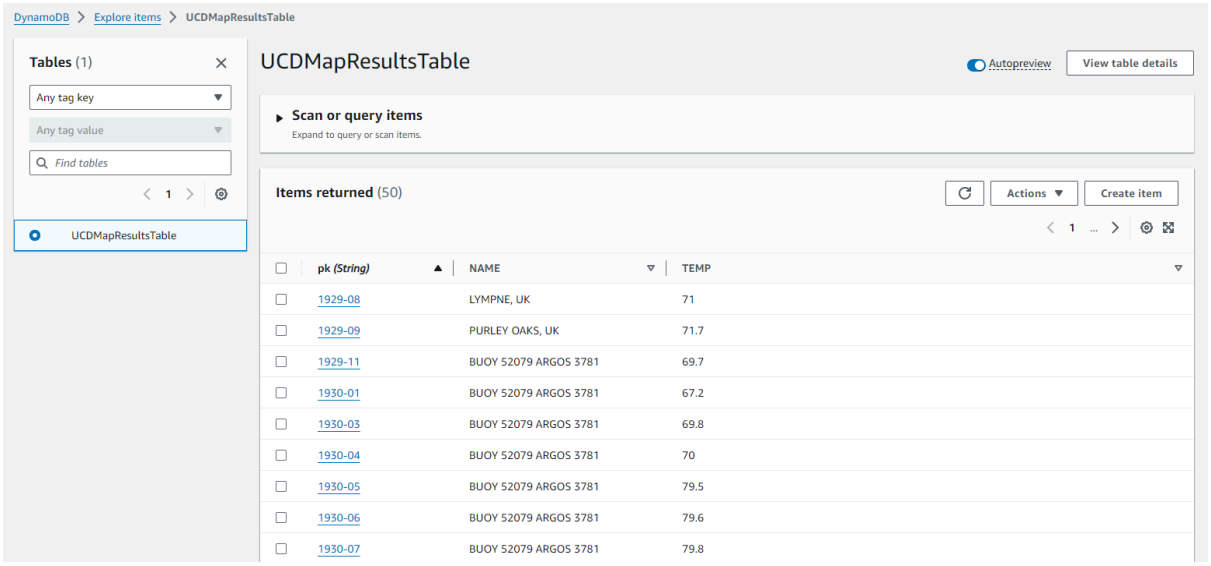


Se puede ver que se procesaron correctamente los 1000 items en 2 ejecuciones de 500 ítems cada una, tal y como se especificó en la configuración.

Al entrar a una de las ejecuciones se puede ver su respectivo input y output. Estos datos se guardan en S3 para posteriormente hallar las mayores temperaturas usando la función Lambda UCDMapReducerFunction y guardar dichos datos en la tabla de DynamoDB.



Al filtrar los resultados por columnas (fecha, nombre del lugar y temperatura) se obtiene el siguiente resultado.



Async Processing

Se toma el template de CloudFormation proporcionado por AWS (region us-west-2).

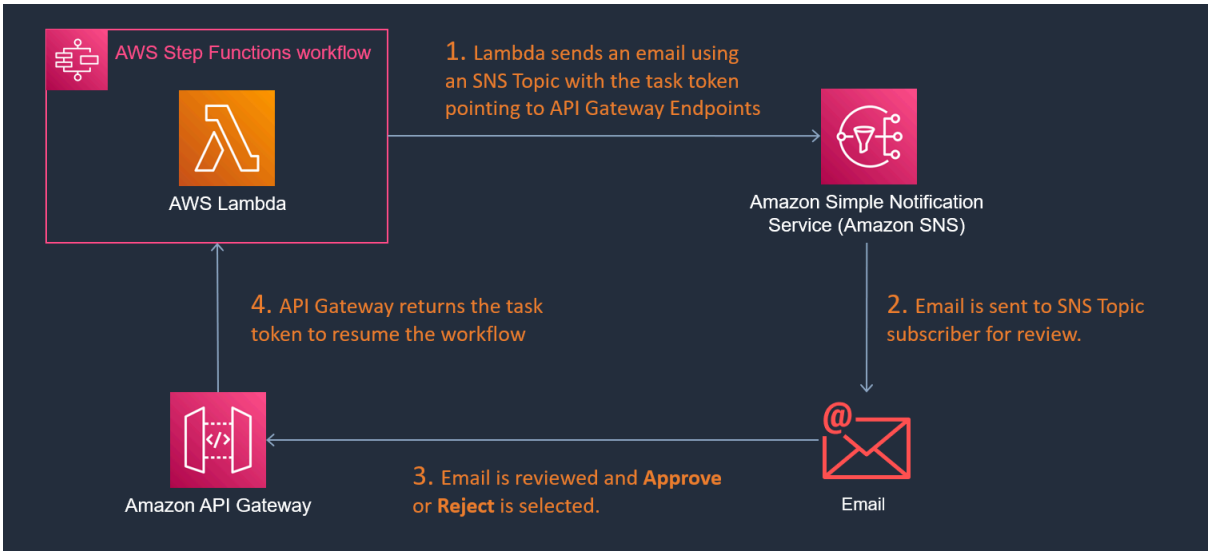
Human Approval

Workflows can require a human to review data and approve to proceed to the next step of a workflow.

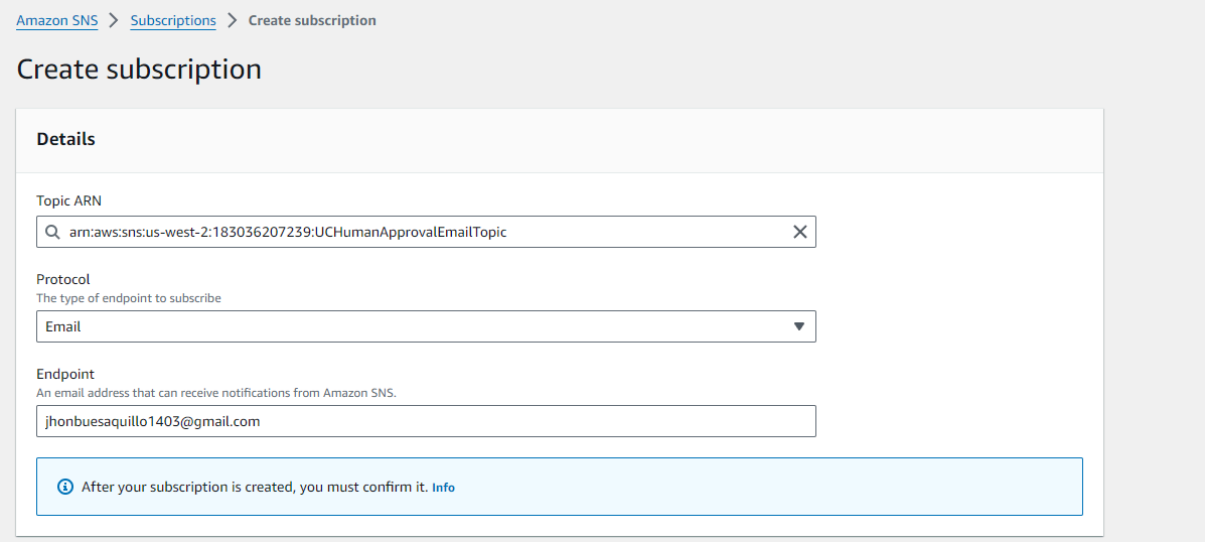
Common use-cases:

- Sending identity documents to a verification approver
- Leadership approval for business processes
- Automating third party communications as part of your workflow.

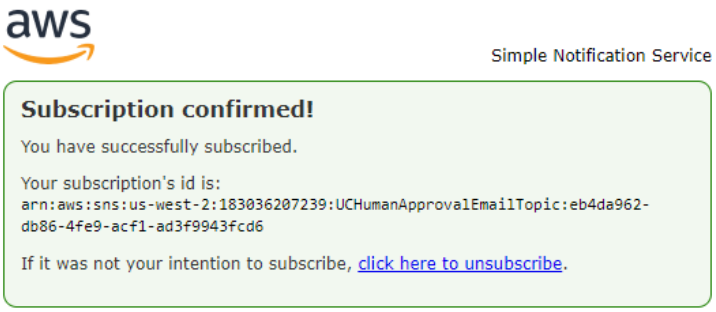
This tutorial shows you how to deploy a Wait for Callback service integration requiring human approval via email. The workflow progresses to the next state once the user has approved or rejected the request.



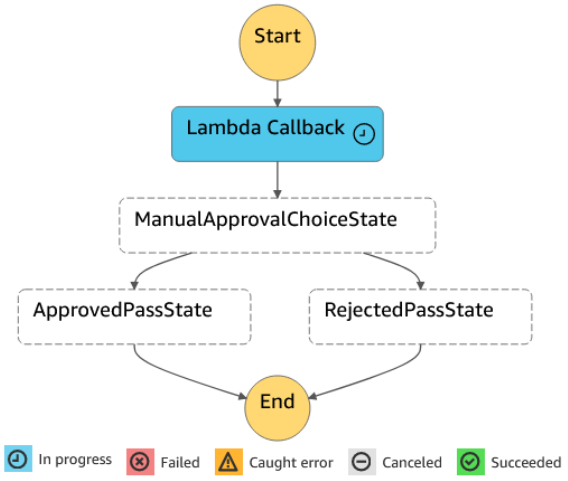
En principio se crea una suscripción por email en el topic UCHumanApprovalEmailTopic.



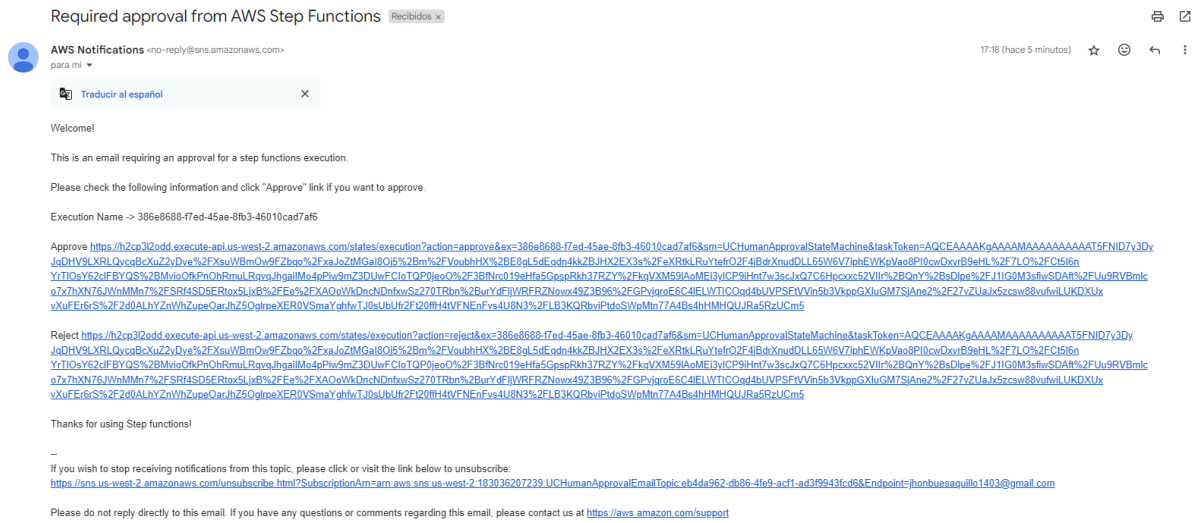
Se confirma la suscripción a través del correo proporcionado.



Se ejecuta la step machine llamada UCHumanApprovalStateMachine.



Como se puede ver, la función Lambda envía un email y espera un callback como respuesta. En este caso, se envía un enlace de aprobación y rechazo al correo proporcionado.



Required approval from AWS Step Functions Recibidos ×

AWS Notifications <no-reply@sns.amazonaws.com>
para mí

17:18 (hace 5 minutos)

Traducir al español

Welcome!

This is an email requiring an approval for a step functions execution.

Please check the following information and click "Approve" link if you want to approve.

Execution Name -> 386e8688-f7ed-45ae-8fb3-46010cad7af6

Approve https://h2cp3l2odd.execute-api.us-west-2.amazonaws.com/states/execution?action=approve&ex=386e8688-f7ed-45ae-8fb3-46010cad7af6&sm=UCHumanApprovalStateMachine&taskToken=AQCEAAAAKgAAAAMAAAAAAAAAAT5FNID7y3Dy JqDHV9LXRLQycqBcXuZ2yDye%2FXsuWBmOw9FZbqo%2FxaJoZtMGaI8Oj5%2Bm%2FVoubhHX%2BE8gL5dEqdn4kkZBJHX2EX3s%2FeXRtkLRuYtefrO2F4jBdrXnudDLL65W6V7iphEWKpVao8PI0cwDxyrB9eHL%2F7LO%2FCt5I6n YrTlOsY62clFBYQS%2BMvioOfkPnOhRmuLRqvgJhgailMo4pPiw9mZ3DUwFCloTQP0jeoO%2F3BfNrc019eHfa5GpspRkh37RZY%2FkgVXM59lAoMEl3ylCP9lHnt7w3scJxQ7C6Hpcxxc52VIlr%2BQnY%2BsDlpe%2FJ1IG0M3sfiwSDAft%2FUu9RVBmlc o7x7hXN76JWnMMn7%2FSRf4SD5ERtox5LjxB%2FEe%2FXAOoWkDncNDnfxwSz270TRbn%2BurYdFIjWRFRZNowx49Z3B96%2FGPvjqroE6C4lELWTICOqd4bUVPSFtVVin5b3VkppGXIuGM7SjAne2%2F27vZUaJx5zcsw88vufwiLUKDXUx vXuFEr6rS%2F2d0ALhYZnWhZupeOarJhZ5OglrpeXER0VSmaYghfwTJ0sUbUfr2Ft20ffH4tVFNEnFvs4U8N3%2FLB3KQRbviPtdoSWpMtn77A4Bs4hHMHQUJRa5RzUCm5

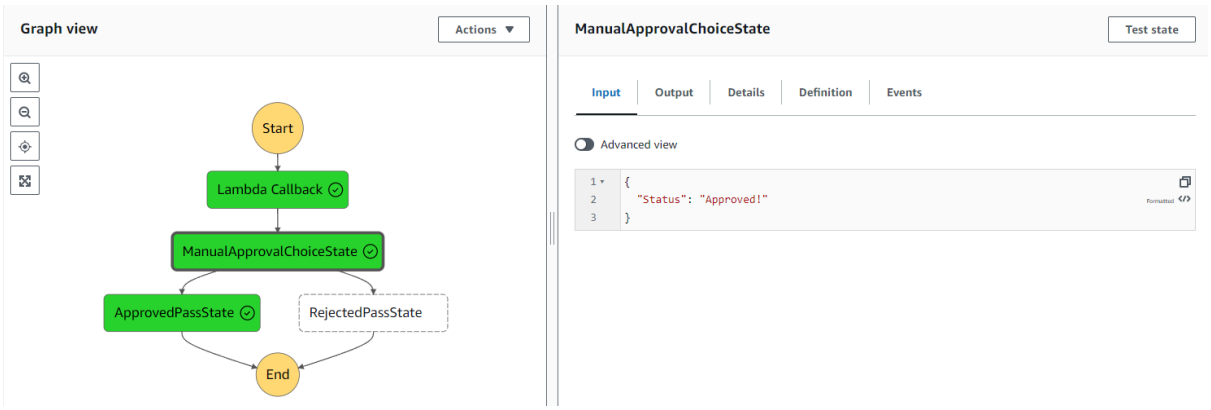
Reject https://h2cp3l2odd.execute-api.us-west-2.amazonaws.com/states/execution?action=reject&ex=386e8688-f7ed-45ae-8fb3-46010cad7af6&sm=UCHumanApprovalStateMachine&taskToken=AQCEAAAAKgAAAAMAAAAAAAAAAT5FNID7y3Dy JqDHV9LXRLQycqBcXuZ2yDye%2FXsuWBmOw9FZbqo%2FxaJoZtMGaI8Oj5%2Bm%2FVoubhHX%2BE8gL5dEqdn4kkZBJHX2EX3s%2FeXRtkLRuYtefrO2F4jBdrXnudDLL65W6V7iphEWKpVao8PI0cwDxyrB9eHL%2F7LO%2FCt5I6n YrTlOsY62clFBYQS%2BMvioOfkPnOhRmuLRqvgJhgailMo4pPiw9mZ3DUwFCloTQP0jeoO%2F3BfNrc019eHfa5GpspRkh37RZY%2FkgVXM59lAoMEl3ylCP9lHnt7w3scJxQ7C6Hpcxxc52VIlr%2BQnY%2BsDlpe%2FJ1IG0M3sfiwSDAft%2FUu9RVBmlc o7x7hXN76JWnMMn7%2FSRf4SD5ERtox5LjxB%2FEe%2FXAOoWkDncNDnfxwSz270TRbn%2BurYdFIjWRFRZNowx49Z3B96%2FGPvjqroE6C4lELWTICOqd4bUVPSFtVVin5b3VkppGXIuGM7SjAne2%2F27vZUaJx5zcsw88vufwiLUKDXUx vXuFEr6rS%2F2d0ALhYZnWhZupeOarJhZ5OglrpeXER0VSmaYghfwTJ0sUbUfr2Ft20ffH4tVFNEnFvs4U8N3%2FLB3KQRbviPtdoSWpMtn77A4Bs4hHMHQUJRa5RzUCm5

Thanks for using Step functions!

--
If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:
https://sns.us-west-2.amazonaws.com/unsubscribe.html?SubscriptionArn=arn:aws:sns:us-west-2:183036207239:UCHumanApprovalEmailTopic:eb4da962-db86-4fe9-acf1-ad3f9943fcd6&Endpoint=jhonbuesaquillo1403@gmail.com

Please do not reply directly to this email. If you have any questions or comments regarding this email, please contact us at https://aws.amazon.com/support

Al dar click en el enlace de Approve, este redirecciona a la finalización exitosa del state machine.



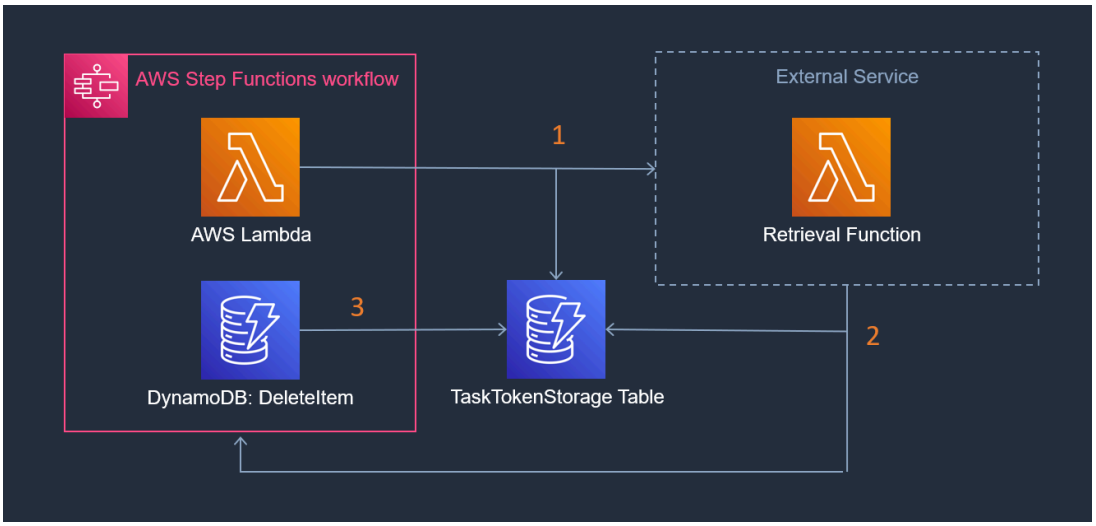
Waiting for External Systems

You can create a Step Functions workflow that contains a dependency on an external service. For example, you can design your state machine to contact an external service and pause, waiting for the service to complete before allowing the workflow to proceed to the next step.

Common use-cases:

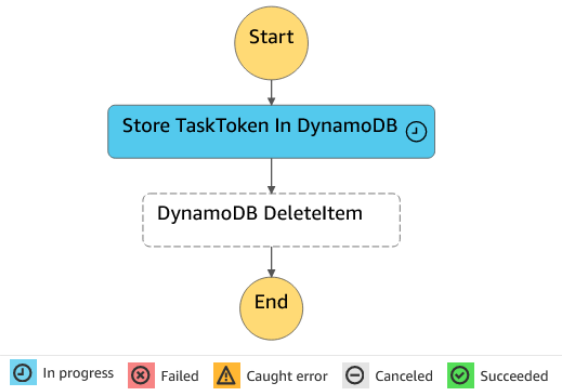
- Waiting on other AWS Service workflows
- Waiting on 3rd party services
- Waiting on legacy applications

In this tutorial, you will create a workflow that uses a Wait for Callback service integration. Your workflow will store a unique task token in Amazon DynamoDB. It will then send a request to an external system and pause. The external system will run a task. When the task is complete, the system will retrieve the task token and send it with a SendTaskSuccess API call to Step Functions. When Step Functions receives the task token, it will instruct your workflow to continue to the next step of the process.

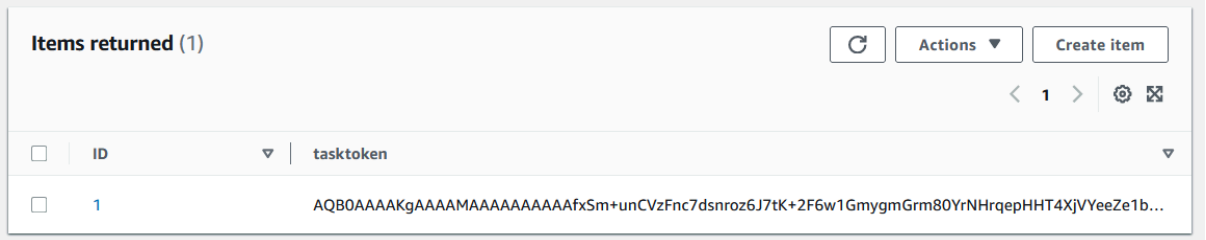


El proceso es el siguiente:

- ➔ Execute a state machine that makes a .waitForTaskToken call to invoke a Lambda function that stores a task token in DynamoDB.
- ➔ Verify that your workflow has paused. View the task token in DynamoDB.
- ➔ Invoke a second Lambda function that acts as an external service to:
  - ◆ Retrieve the task token from DynamoDB.
  - ◆ Make a SendTaskSuccess API call to Step Functions with the token to instruct your workflow to continue.
- ➔ Verify the state machine execution completed.
- ➔ Verify the task token was deleted from DynamoDB.



El token queda almacenado en la tabla de DynamoDB.

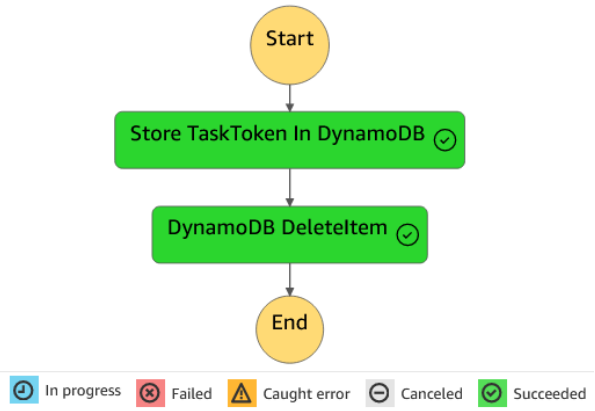


Ahora, usando la 2da función llamada UCExtSystemRetrieveTaskTokenFunction, se utiliza la opción Test para enviar el token a la state machine.

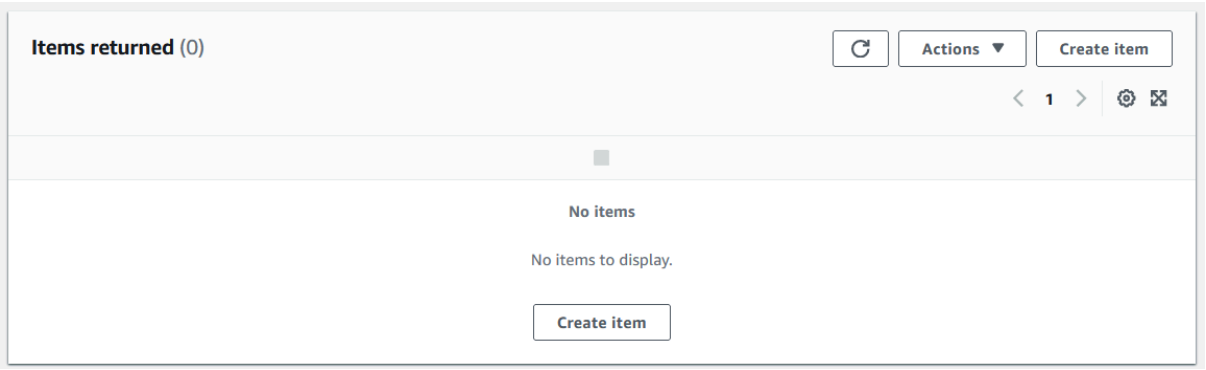
```python
import boto3
import os

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table(os.environ["DYNAMODB_TABLE_NAME"])
client = boto3.client('stepfunctions')

def lambda_handler(event, context):
    response = table.get_item(Key={'ID': event['ID']})
    client.send_task_success(
        taskToken=response['Item']['tasktoken'],
        output="{ \"Payload\":\"Success\"}"
    )
```



El token es eliminado de la tabla de DynamoDB.



You have retrieved a task token in DynamoDB and returned the task token to the state machine. You also have successfully deleted the task token from DynamoDB.

External HTTP Integration

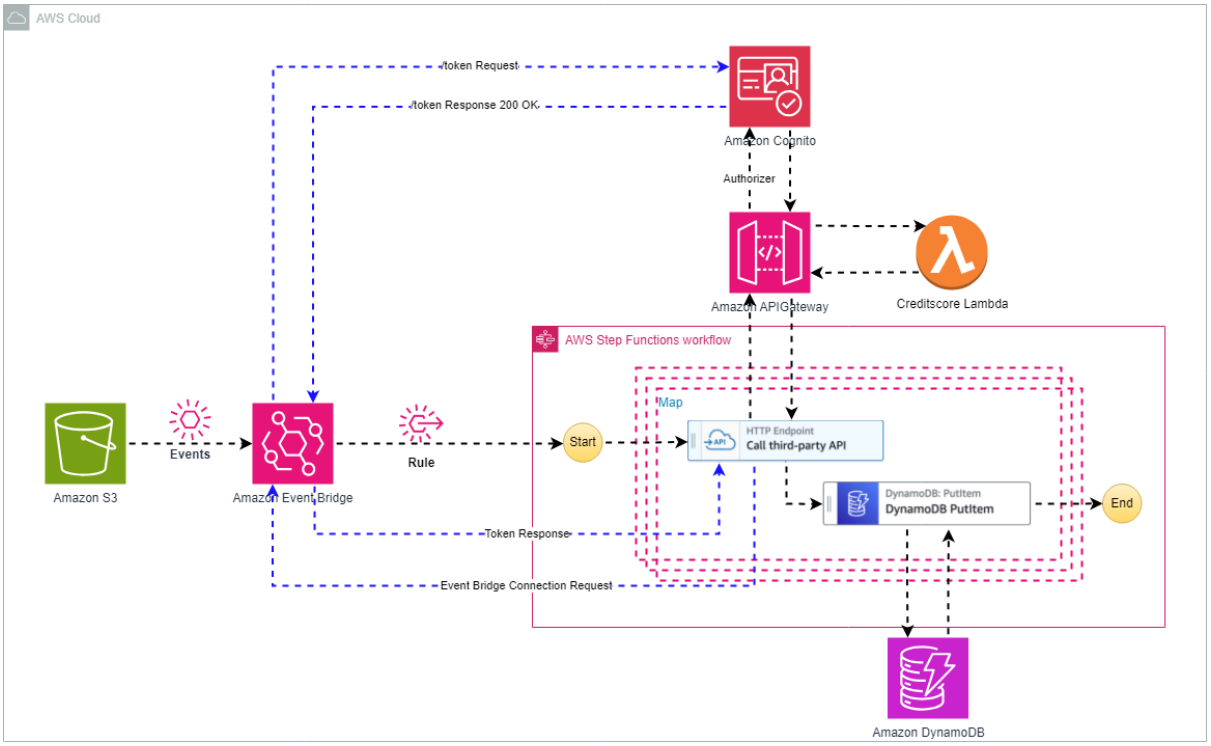
Workflows often require realtime integration with third party API's. This tutorial shows you how to deploy a step function statemachine to invoke a HTTP API endpoint.

Common use-cases:

- Query reference data from external API for processing
- Integration of payment gateway systems

- CRM integration with external marketing systems
- Realtime create or update of process status to external systems

This tutorial deploys the following architecture in the environment which provides a secured Mock Credit Score API to integrate using HTTP API Task.

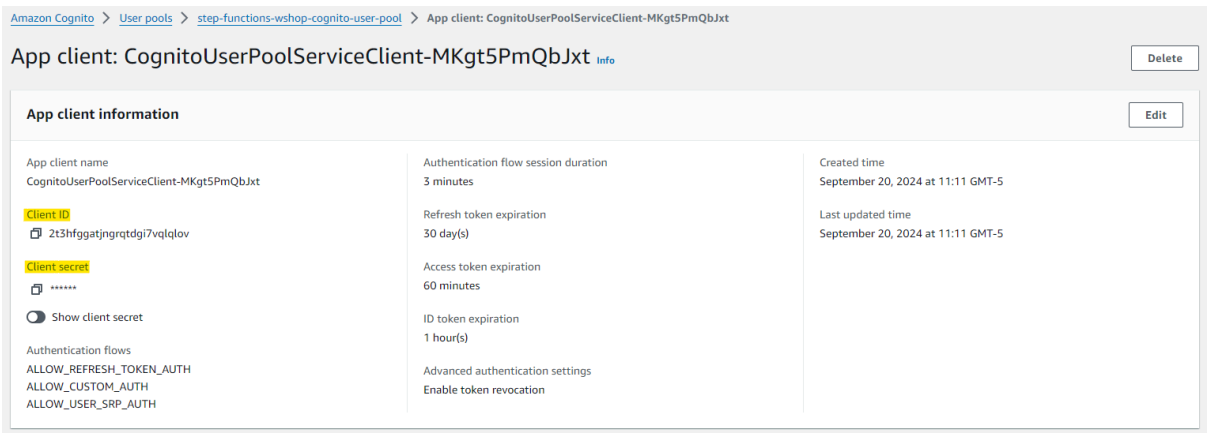


El proceso es el siguiente:

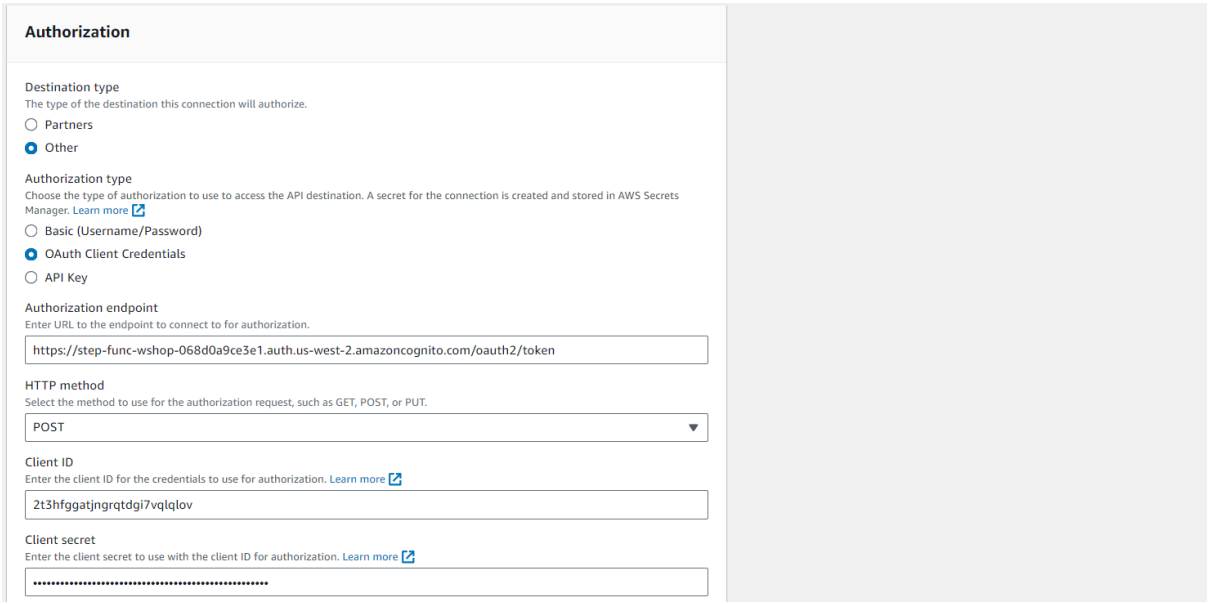
- Create EventBridge API destination connections using Cognito credentials.
- Update state machine Map task to include Call HTTP endpoint.
- Update DynamoDB PutItem task to include the response from HTTP API invocation.
- Test the changes by uploading the sample CSV file in the S3 bucket.

Se toma el template de CloudFormation proporcionado por AWS (region us-west-2).

Inicialmente, se ingresa al user pool creado en Amazon Cognito. En la pestaña App integration se copia el dominio para usarlo más adelante. A su vez, en la sección App clients and analytics se ingresa al app client creado y se toman los valores de Client ID y Client secret.

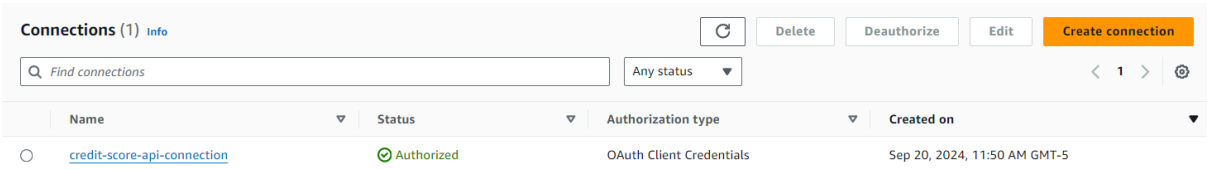


Ahora, en el servicio EventBridge se crea una connection con la siguiente configuración.

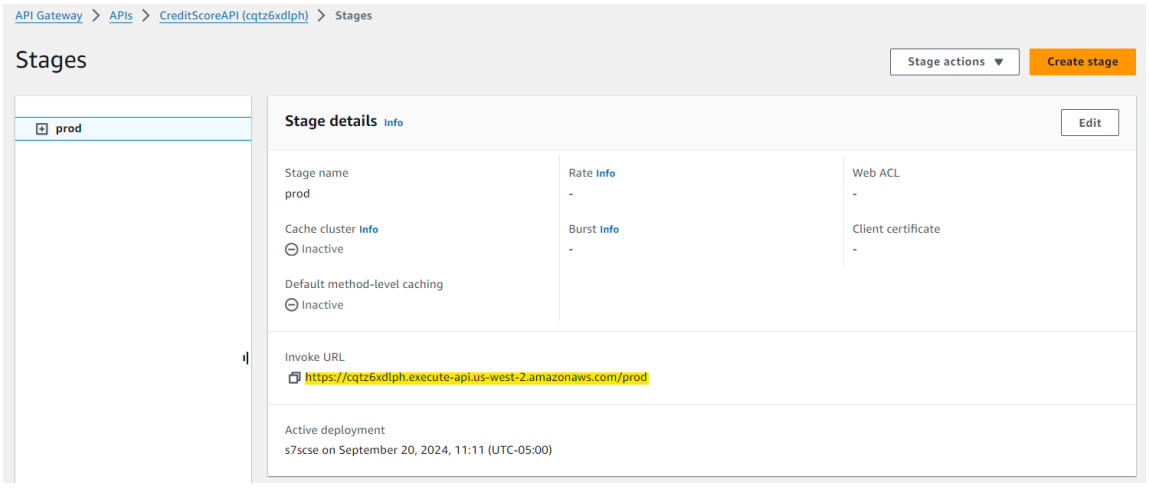


Como se puede ver, se apunta al user pool de Cognito utilizando el domain, client ID y client secret.

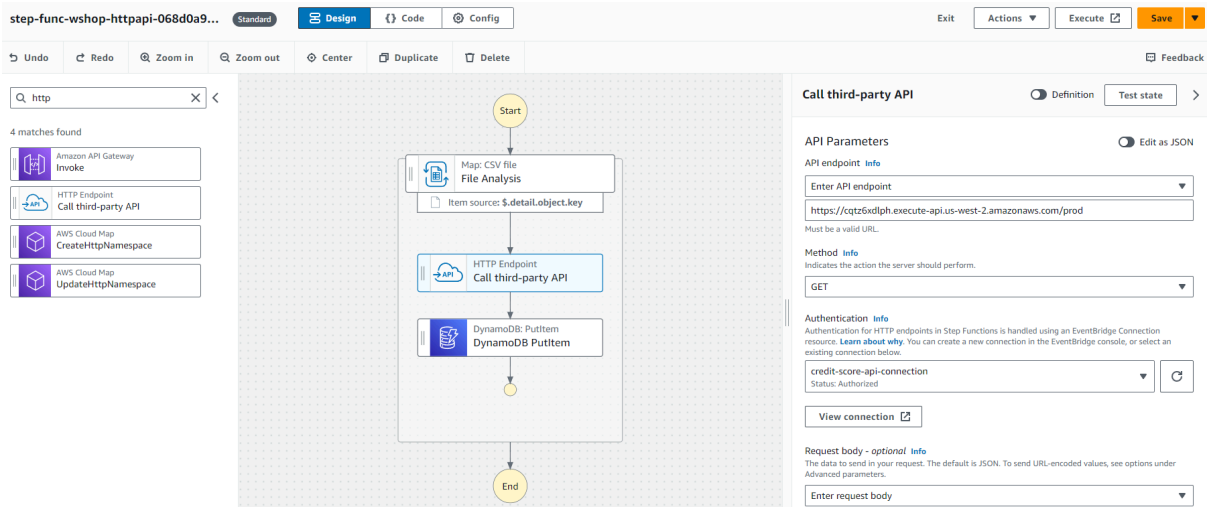
API Destination will create the **connection** by invoking the token endpoint with client id and secret, generating the **token** to be served for step functions API call and shows the green check mark as Authorized.



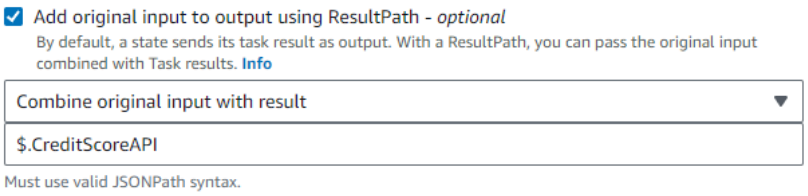
En el servicio API Gateway se ingresa a la API creada y en la sección Stages se toma el Invoke URL.



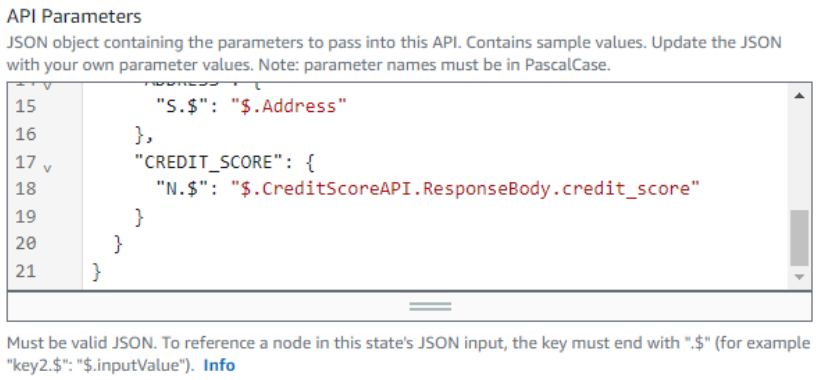
Ahora, en la state machine ya creada se agrega un nuevo step llamado HTTP Endpoint (Call third-party API). En su configuración, se agrega el endpoint y se elige el método GET. En la sección de Authentication, se elige la connection creada anteriormente en EventBridge.



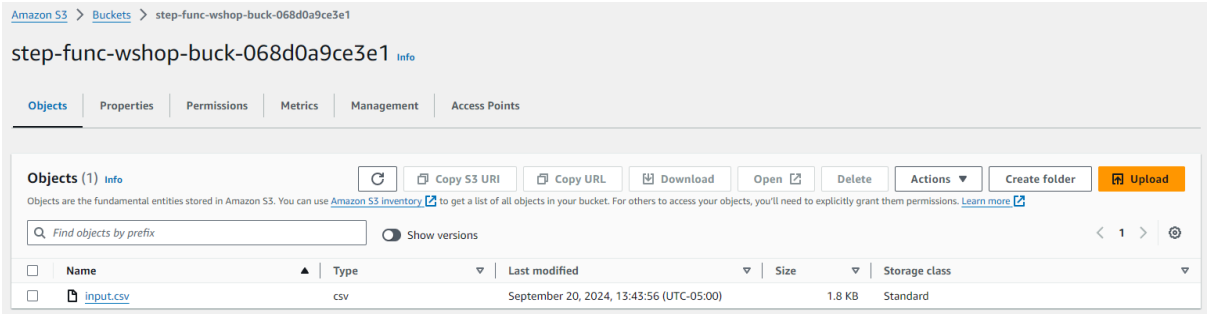
En la pestaña Output se selecciona la opción Add original input to output using ResultPath.



Por último se actualizan los parámetros del DynamoDB Putitem task, agregando el credit_score.



Se sube el archivo .csv al bucket ya creado con la información proporcionada en el workshop.



Para obtener el valor de credit_score se usa la función Lambda ya creada.

index.py | Environment Var

```python
import json
import random

def handler(event,context):
  credit_score = random.randint(600, 850)

  return {
        "statusCode": 200,
        "body": json.dumps({
          'credit_score': str(credit_score)
        }),
  }
```

Se realiza un test en el método GET del API Gateway.

/ - GET method test results

Request

/

Latency ms

347

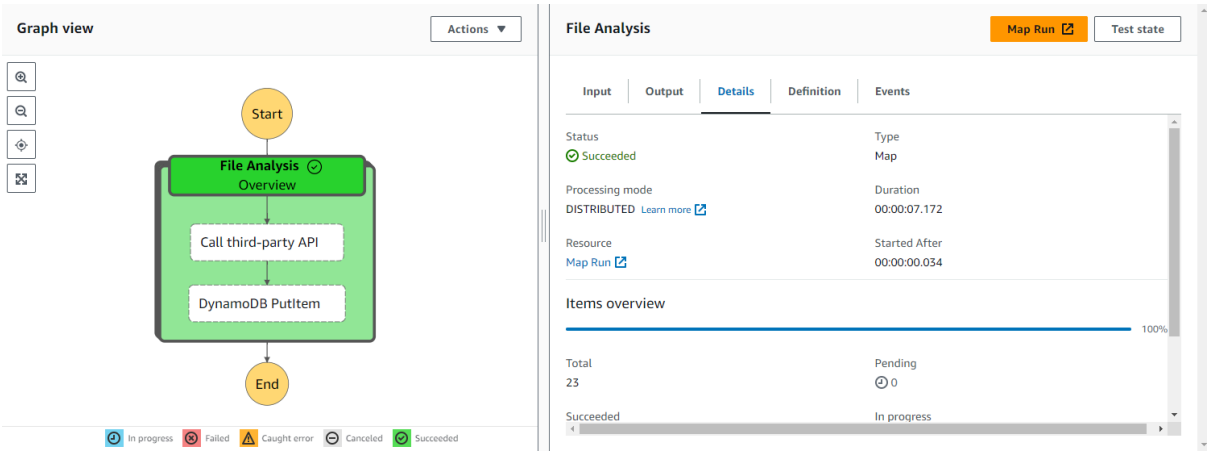
Status

200

Response body

```
{"credit_score": "779"}
```

Se ejecuta la state machine exitosamente.



Al consultar en la tabla de DynamoDB se puede ver que almacena los datos del archivo .csv junto con el credit_score para cada registro.

Items returned (46)

Actions | Create item

ID (String)	ADDRESS	CREDIT_SC...	FIRST_NA...	LAST_N...	PHONE_NUMBER
9c69447c-6139-4ef3-...	162 ypMIV...	715	Emily	Brown	863-462-1951
08cfeb31-27d5-4306...	105 ZnVEce...	832	Christopher	Davis	921-122-6187
776fdc8f-8c47-420e-...	188 pkAvh...	601	Christopher	Davis	775-266-1574
5524acbc-34cf-44fc-...	957 MNPCp...	623	Sarah	Brown	742-942-1425
cec25247-9c9f-4cef-a...	732 EUWXg...	652	Olivia	Johnson	990-733-6537
0b9a31a8-29a6-4cf9-...	361 pZfbKP...	653	Emily	Wilson	582-502-5183

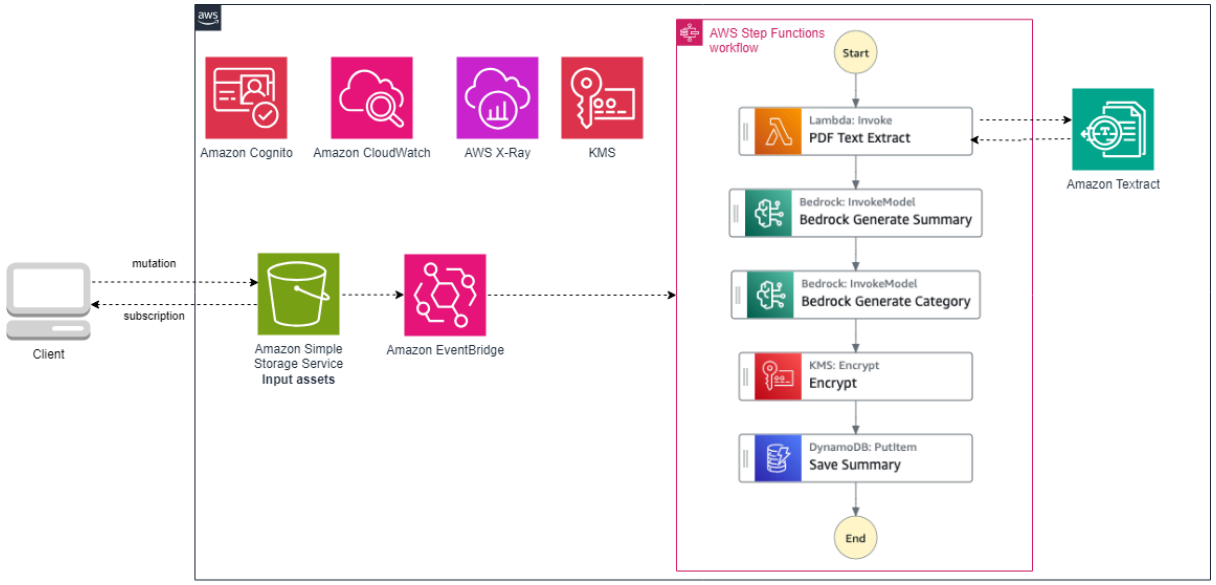
Document Summarization Using Amazon Bedrock

Amazon Bedrock and AWS Step Functions offer a powerful combination for creating elaborate intelligent document processing pipelines. Amazon Bedrock is a machine learning (ML) service that provides pre-trained models for various natural language processing (NLP) tasks, including text summarization. AWS Step Functions, on the other hand, is a serverless orchestration service that allows you to create and manage complex workflows involving multiple AWS services.

In this workshop module you will be modifying the state machine to invoke an Amazon Bedrock API (InvokeModel).

A pipeline has been pre-configured and deployed with the following steps:

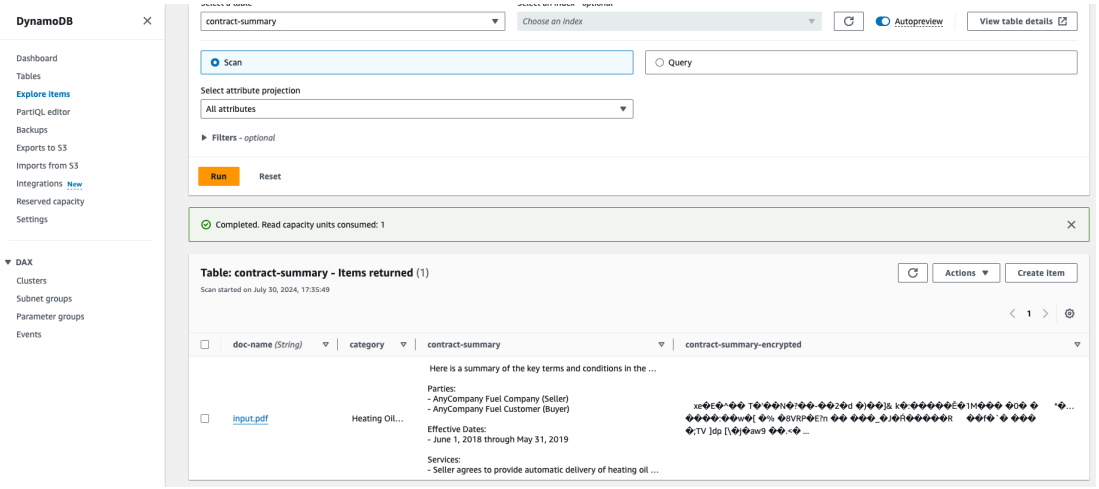
- S3 bucket is configured with Amazon EventBridge notification and Amazon Eventbridge rule configured to trigger the state machine
- In the state machine, AWS lambda invokes Amazon Textract to extract the content from a PDF document
- Uses AWS KMS to encrypt the document summary, ensuring data security and compliance
- Persists the category, summary and encrypted summary in DynamoDB



Tutorial:
https://catalog.workshops.aws/stepfunctions/en-US/use-cases/genai-document-processing/step-3

Nota: El template de CloudFormation arrojó error 3 veces por el KMS service.

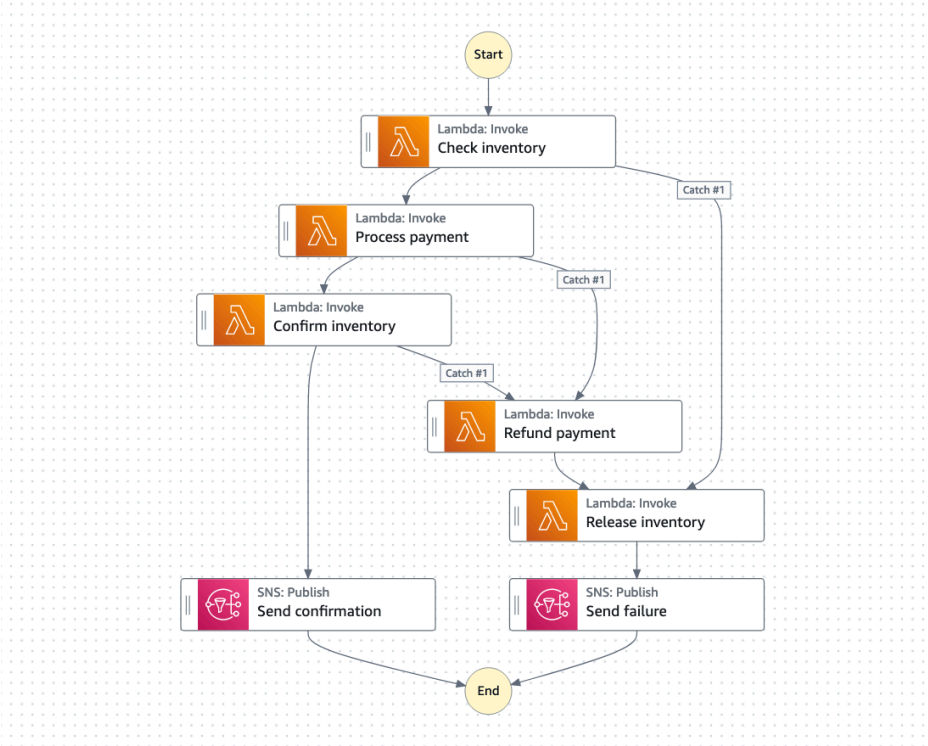
El resultado final en DynamoDB es un registro con la categoría encontrada y el resumen del documento pdf, como también este mismo resumen encriptado usando KMS.



Distributed Transactions or the "Saga Pattern"

The **saga pattern** is a failure management pattern that helps establish consistency in distributed applications and coordinates transactions between multiple microservices to maintain data consistency. Basically, every service that performs a part of the transaction publishes an event that triggers a subsequent step. This continues until the last part of the transaction in the chain is complete.

In case a part of the business transaction fails, you might end up with partial transactions. In this case, some control logic is needed to undo the transactions that have already been processed. You will need to orchestrate a series of compensating transactions that undo the changes that were made by the preceding transactions.

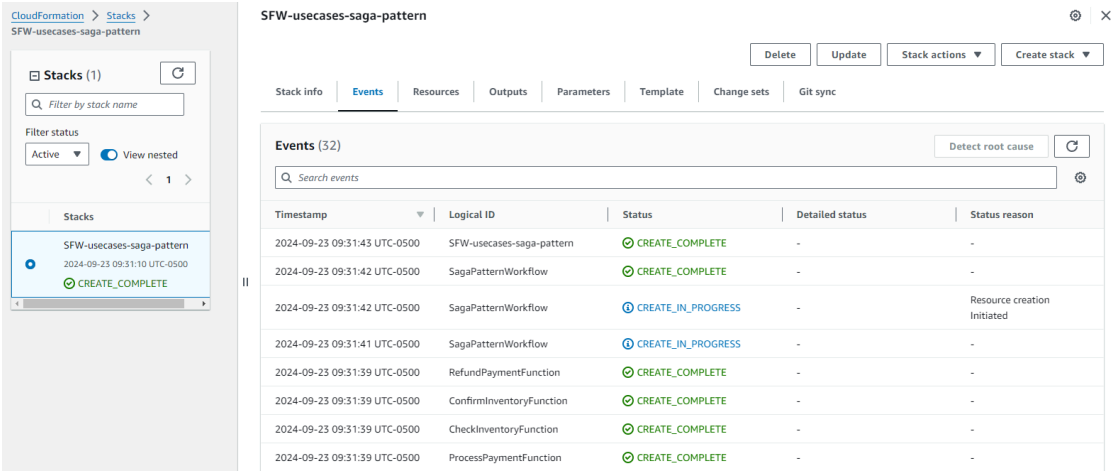


In this module, you will create a workflow that uses Retriers as well as Catchers to implement the saga pattern. Your workflow will orchestrate several Lambda functions that represent the business logic of the individual components. You will add retry and catch logic to enable the distributed transaction handling correctly.

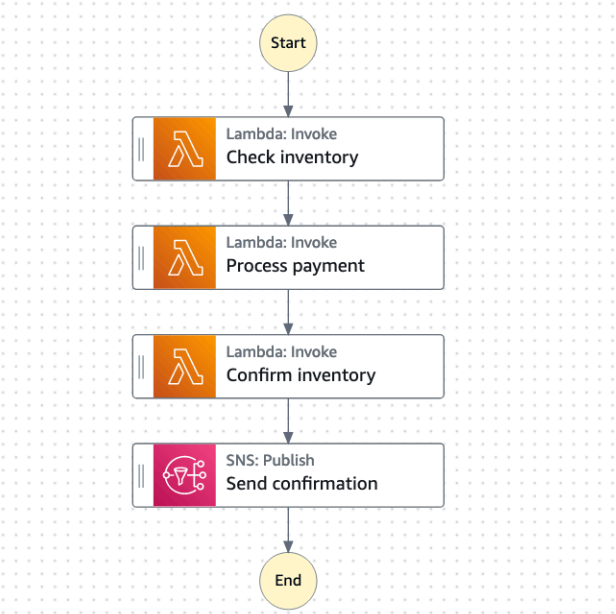
Currently, the workflow contains the business logic for a (simplified) ordering process, like on an e-commerce website. The business logic has been split into three separate parts that are each fulfilled by an AWS Lambda function.

- Check inventory - this function checks if the requested item is actually valid and in stock.
- Process payment - this function serves as the payment processor. It serves as a placeholder for a real payment processor and doesn't actually contain any real payment processing logic.
- Confirm inventory - the requested item isn't actually reserved for the user until after the payment has been successfully processed. Meaning that it could have been bought by another user in the meantime. This function confirms the item if it is still available.

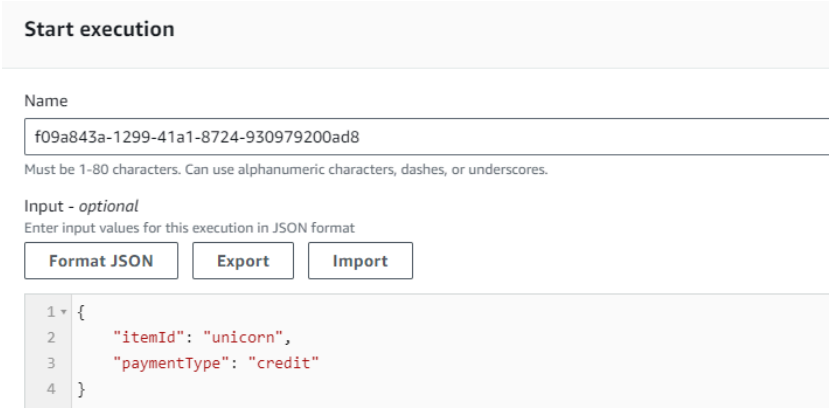
Se toma el template de CloudFormation proporcionado por AWS (region us-west-2).



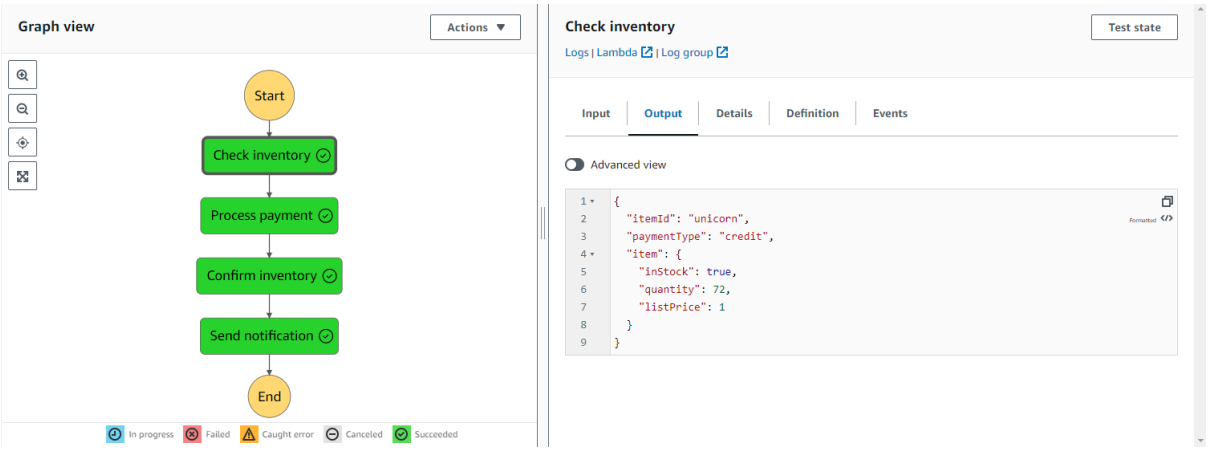
Se ingresa a la step machine llamada SagaPatternWorkflow y se validan los pasos que tiene.



Se ejecuta con el siguiente payload.



El flujo se ejecuta correctamente.

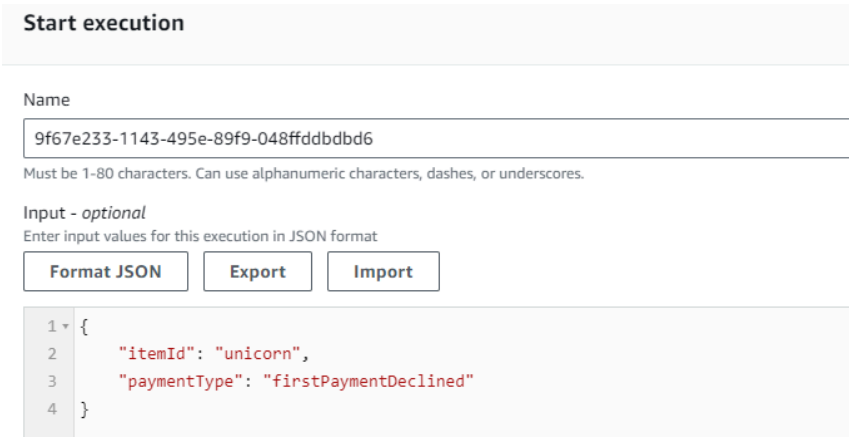


Simulating an error

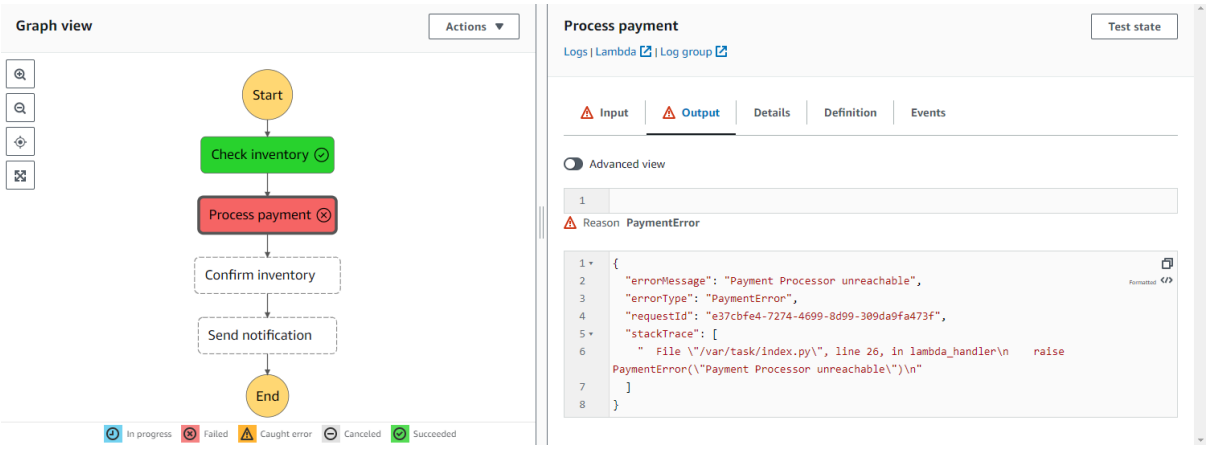
It is therefore important to build workflows that are resilient. That means the workflow should not fail unexpectedly when an error occurs. This is especially important when your workflow represents a business transaction that contains multiple steps across more than one service.

The code of the Lambda functions representing the business logic has been designed for this workshop so that different errors can be simulated. Let's start by simulating that the user provided an incorrect payment method and that the processing fails.

Se inicia una nueva ejecución con el siguiente payload.

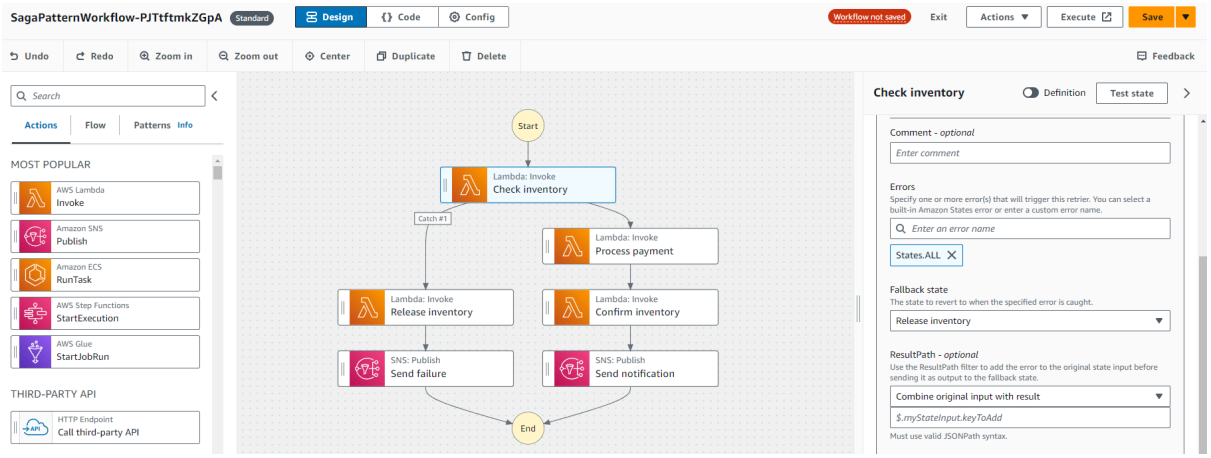


Se dispara el error que se encuentra en la función Lambda llamada ProcessPayment.



Se debe añadir la lógica de **error handling** para hacer más resiliente la state machine.

Para ello, se abre la pestaña Error handling de la función Lambda Check inventory y se crea un nuevo catch, teniendo en cuenta cualquier error (States.ALL) y apuntando a la función Lambda ReleaseInventory. Finalmente se añade un nuevo step de SNS para enviar el mensaje de fallo.



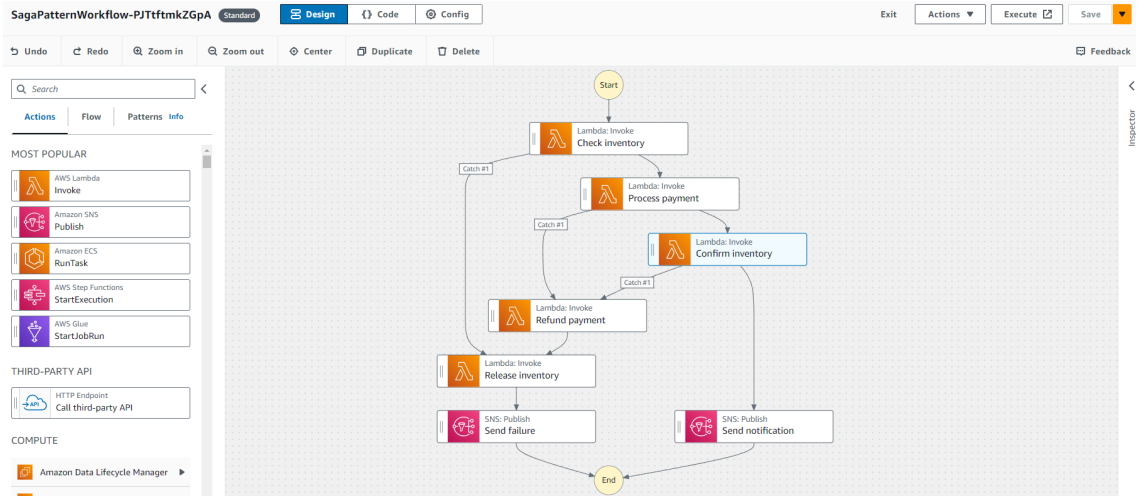
Basically, every service that performs a part of the transaction publishes an event that triggers subsequent steps. This continues until the last part of the transaction in the chain is complete.

In case a part of the business transaction fails, you might end up with partial transactions. In this case, some control logic is needed to undo the transactions that have already been processed. You will need to orchestrate a series of compensating transactions that undo the changes that were made by the preceding transactions.

To expand the error handling of your business process you need to process unresolved errors of the Process payment state. In the context of a business transaction, it is important that you not only handle the immediate effects - refunding payment - but also the effect of earlier states, reserving inventory in this case.

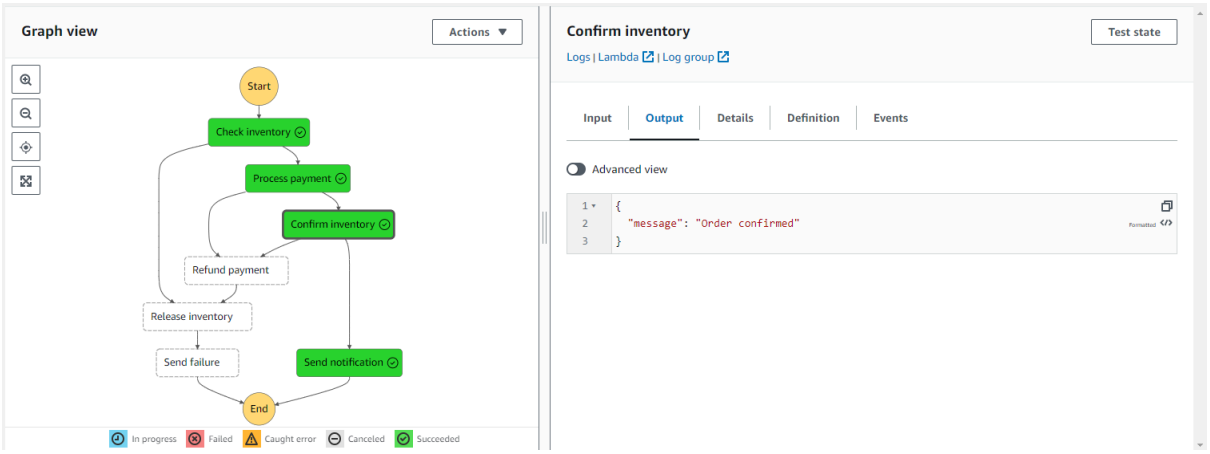
To complete the error handling, you need to add a Catcher to the Confirm inventory state. Note that not every step needs to have its own rollback logic. In this scenario for example, there are no compensating actions to perform specifically for the Confirm Inventory step as the desired item quantity wasn't available when necessary. Only the preceding steps need to be rolled back therefore.

El workflow queda finalmente de la siguiente manera.

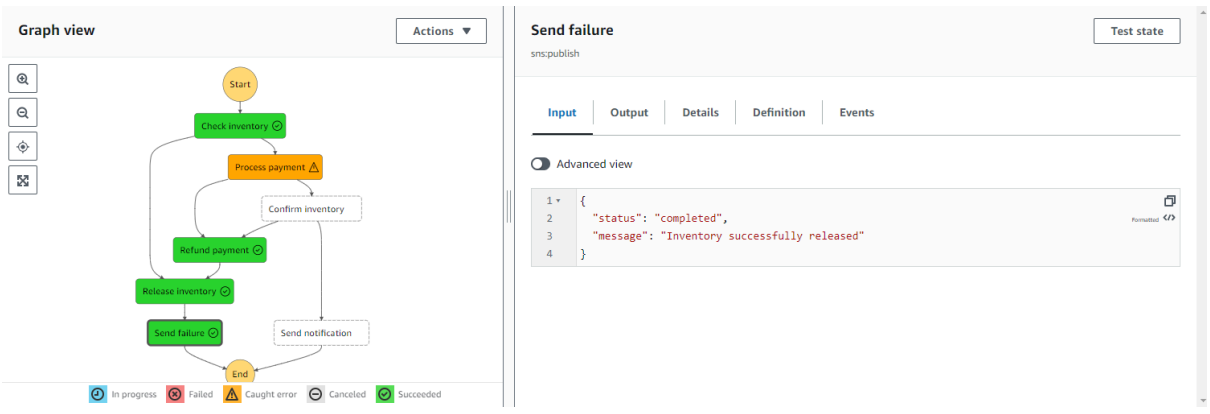


Básicamente, ante cualquier fallo en el proceso de pago o en los pasos posteriores a este, se hará un reembolso del dinero. Luego se actualizará el inventario con el objeto que finalmente no se compró y se notificará el fallo usando el servicio SNS.

Al ejecutar el state machine con el primer payload, todo concluye tal y como pasaba antes de la actualización del workflow.

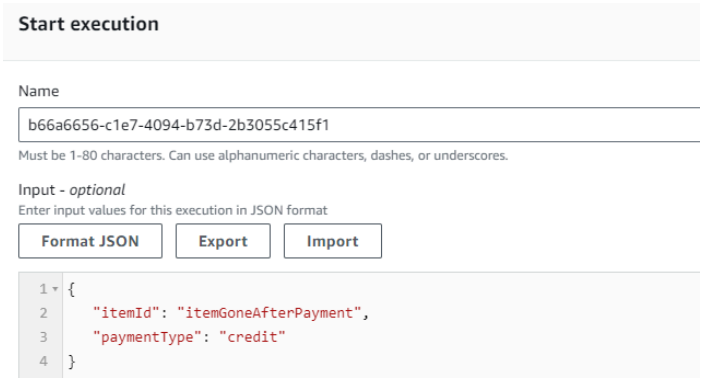


Ahora, se ejecuta con el segundo payload para disparar el error en el proceso de pago.

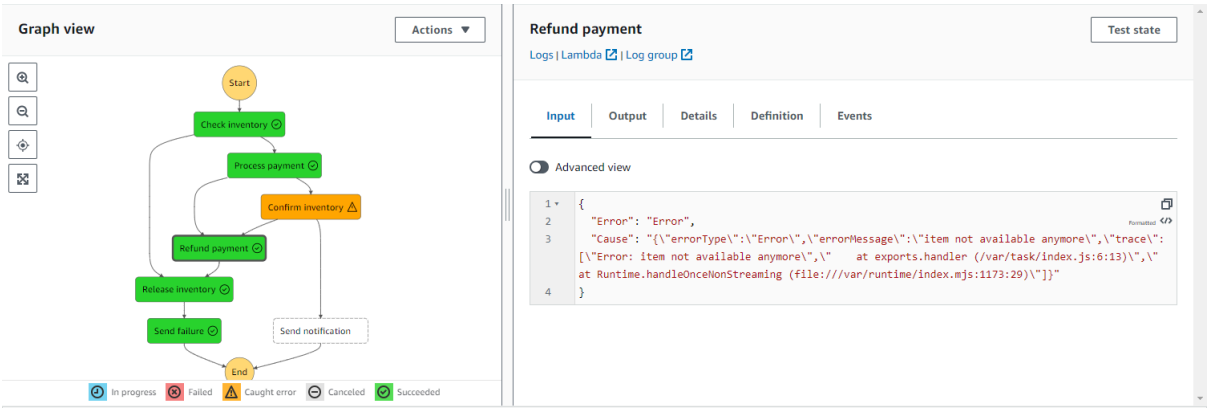


As expected, the Catcher transitioned the execution to the Refund payment state to undo any potential payments. Afterwards the inventory is released and the user would be notified.

Lastly, test what happens when the Confirm inventory step fails. For example because the item was sold to someone else in the meantime.



Al usar el payload que indica que el ítem ya se vendió, se dispara el error en la función Confirm inventory.

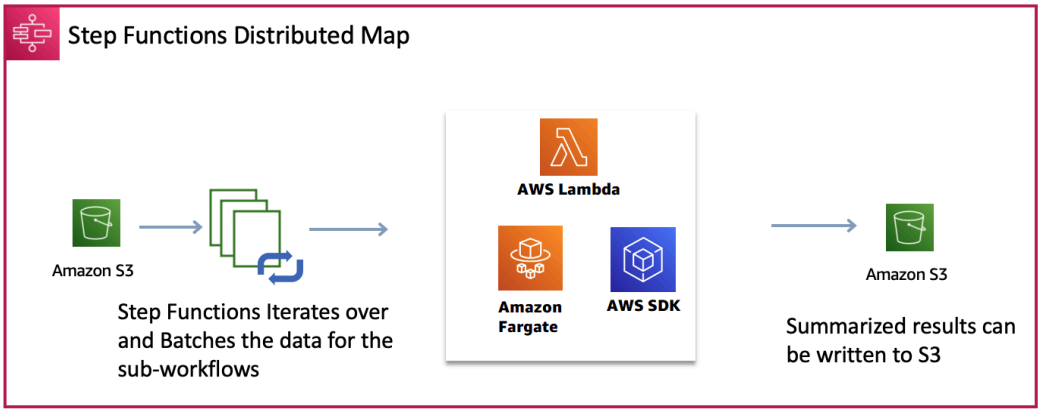


Large-scale Data Processing with Step Functions

Basics

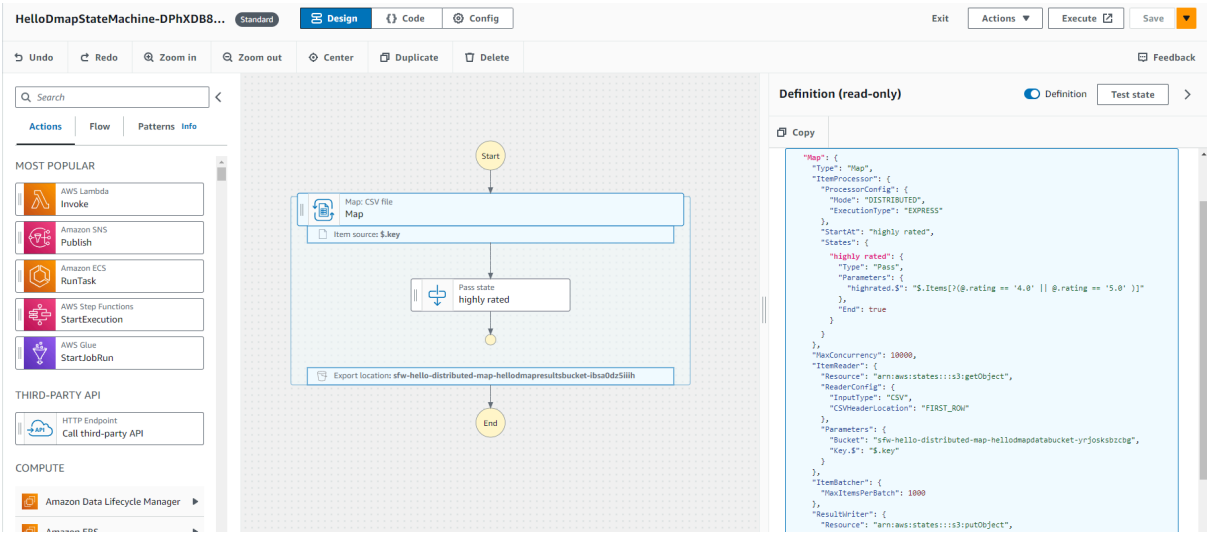
Introduction to Distributed Map

Distributed map is a map state that executes the same processing steps for multiple entries in a dataset at 10,000 concurrency. This means you can run a large-scale parallel data processing workload without worrying about how to parallelize the executions, workers, and data. Distributed map can iterate over millions of objects such as logs, images or records inside .csv or json files stored in Amazon S3. It can launch up to 10,000 parallel child workflows to process the data. You can include any combination of AWS services like AWS Lambda functions, Amazon ECS tasks, or AWS SDK calls in the child workflow.



Se toma el template de CloudFormation proporcionado por AWS (region us-east-1).

Se ingresa a la state machine llamada HelloDmapStateMachine y se visualiza la definición del workflow.



Firstly, you can do batching. Do you see MaxItemsPerBatch set as 1000?

You can not only run 10,000 (10K) workflows, you can also batch the data to each workflow which means, in a single iteration, you can process 10K * 1K = 10M records from the csv file!

You can set failure toleration. What does that mean? You don't want to run 100M records when half of them are bad data. It is a waste of time and money to process those records. By default, the failure toleration is set to 0. Any single child workflow failure will result in the failure of the workflow.

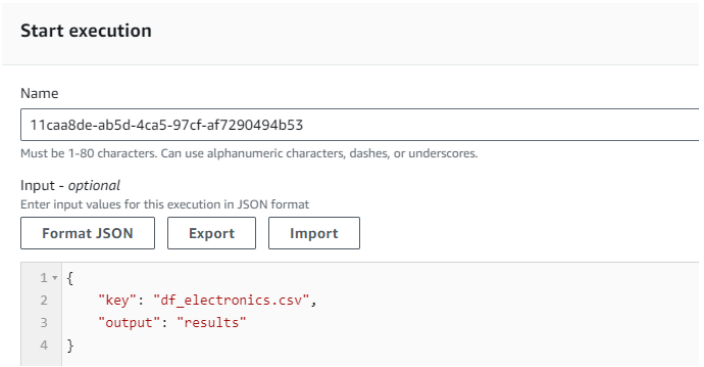
Data quality is a big challenge with large data processing. So, you can set a percentage or number of items that can be tolerated as failures. When failures exceed that tolerance, the Step Functions workflow fails, saving you time and money.

```
"MaxConcurrency": 10000,
"ItemReader": {
  "Resource": "arn:aws:states:::s3:getObject",
  "ReaderConfig": {
    "InputType": "CSV",
    "CSVHeaderLocation": "FIRST_ROW"
  },
  "Parameters": {
    "Bucket": "sfw-hello-distributed-map-hellodmapdatabucket-yrjosksbzcbg",
    "Key.$": "$.key"
  }
},
"ItemBatcher": {
  "MaxItemsPerBatch": 1000
},
"ResultWriter": {
  "Resource": "arn:aws:states:::s3:putObject",
  "Parameters": {
    "Bucket": "sfw-hello-distributed-map-hellodmapresultsbucket-ibsa0dz5iiih",
    "Prefix.$": "$.output"
  }
},
"ToleratedFailurePercentage": 1,
"End": true
```

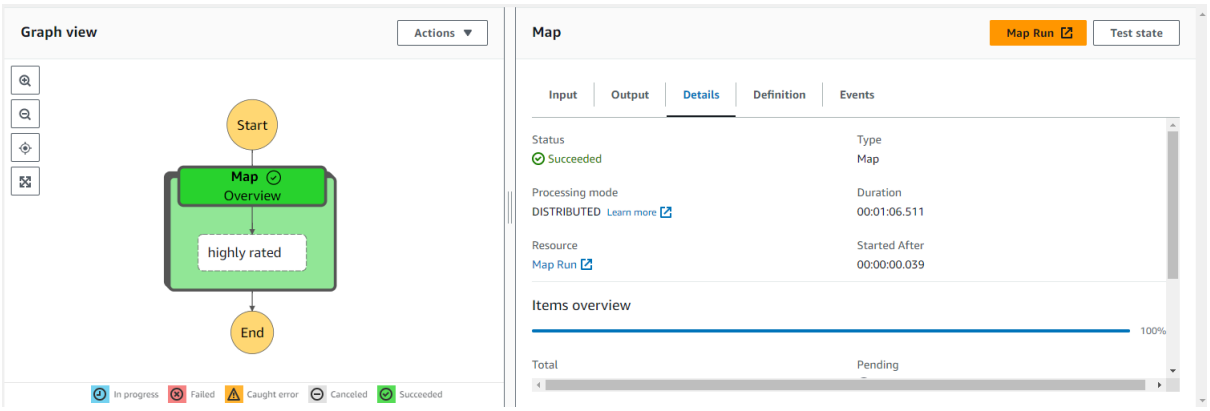
The states inside the distributed map are run as separate **child workflows**. The number of child workflows is dependent on the concurrency setting and the volume of the records to process. For example, you might set the concurrency to 1000 and batch size to 100, but if the total number of records in the file is just 20K, Step Functions only needs 200 child workflows (20,000 / 100 = 200). On the other hand, if the file has 200K records, Step Functions will spin up 1000 child workflows to reach the max concurrency and as child workflows complete, Step Functions will spin up child workflows until all 2000 (200,000 /100 = 2000) child workflows are completed.

```
"States": {
  "highly rated": {
    "Type": "Pass",
    "Parameters": {
      "highrated.$": "$.Items[?(@.rating == '4.0' || @.rating == '5.0' )]"
    },
    "End": true
  }
```

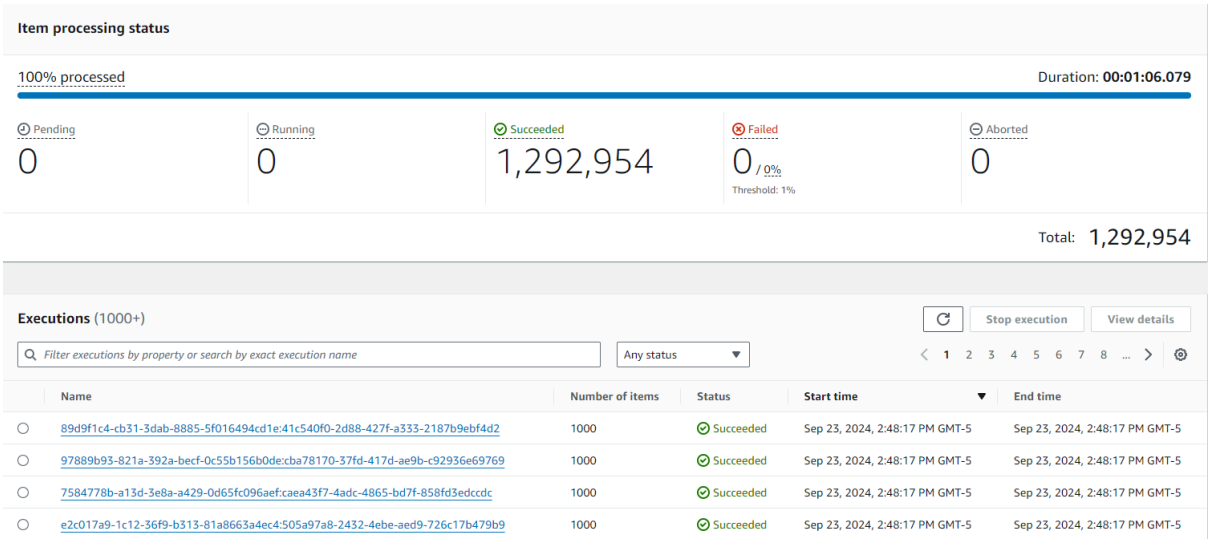
Ahora, se carga el archivo df_electronics.csv proporcionado en el workshop y se ejecuta el state machine usando el siguiente payload.



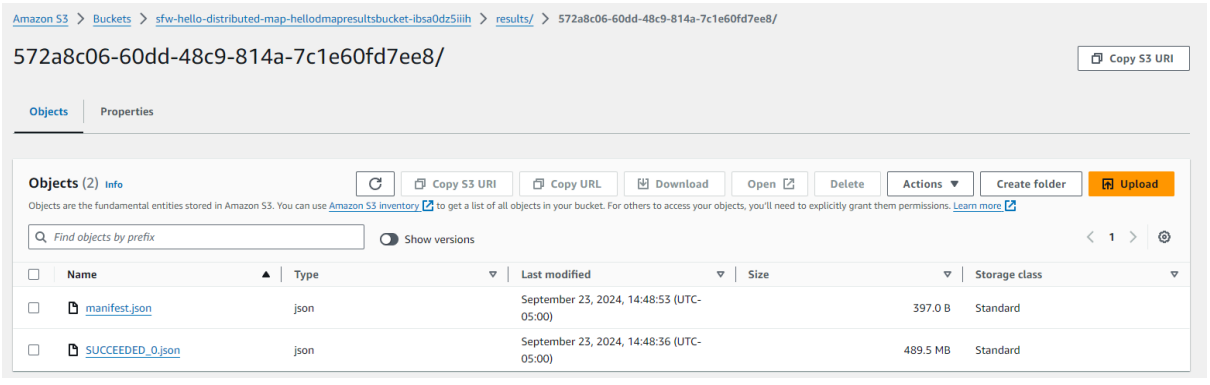
Se ejecuta el workflow correctamente.



Al revisar el Map Run se puede ver que todos los registros (+1M) se procesaron sin problema.



En el bucket de salida se puede ver la información almacenada. En este caso los registros con valor de rating mayor o igual a 4.

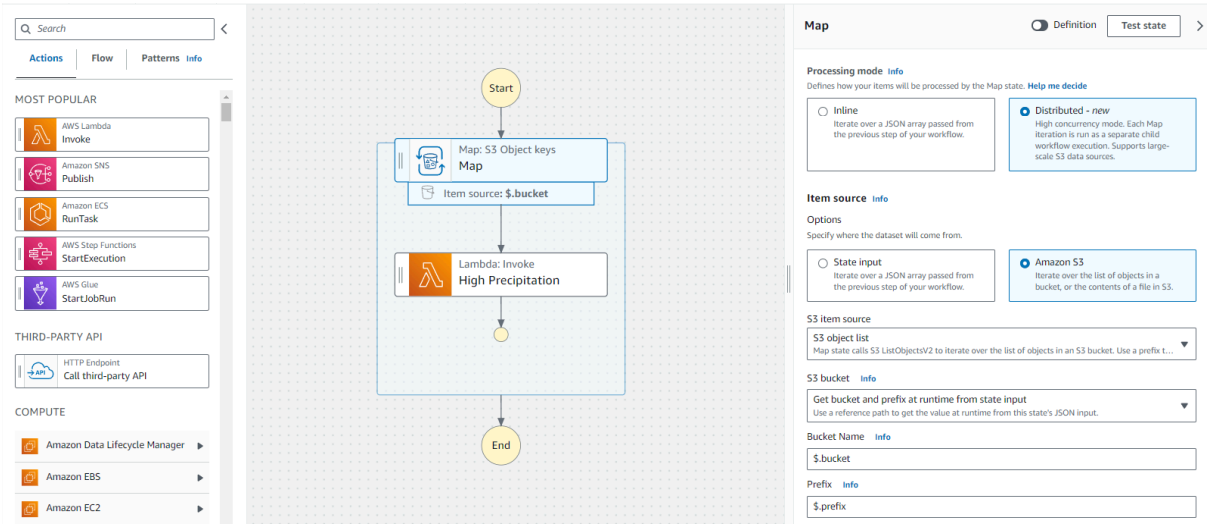


Fantastic! You processed 1.3M records in less than 90 seconds without using servers and running complex code. Distributed map is exciting, right!?!

Building a Distributed Map Workflow

Se toma el template de CloudFormation proporcionado por AWS (region us-east-1).

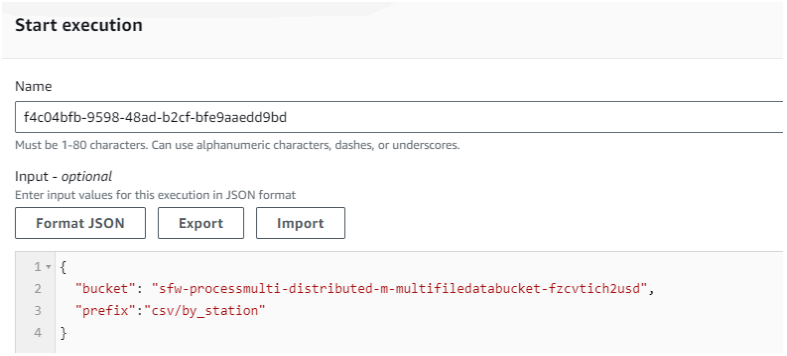
Se crea una state machine en blanco. Se agrega un Map state con la siguiente configuración. A su vez se agrega una función Lambda que va a manipular cada archivo del bucket origen.



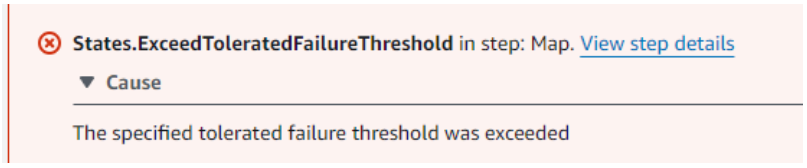
Notice that the ItemReader object uses listobjectV2. In sub-module 1, you saw GetObject in ItemReader. The reason is that you processed a single S3 object in sub-module 1 and you are processing multiple S3 objects here in this sub-module. You can nest both patterns to read multiple csv/json files in a highly parallel fashion.

```
"ItemReader": {
  "Resource": "arn:aws:states:::s3:listObjectsV2",
  "Parameters": {
    "Bucket.$": "$.bucket",
    "Prefix.$": "$.prefix"
  }
},
```

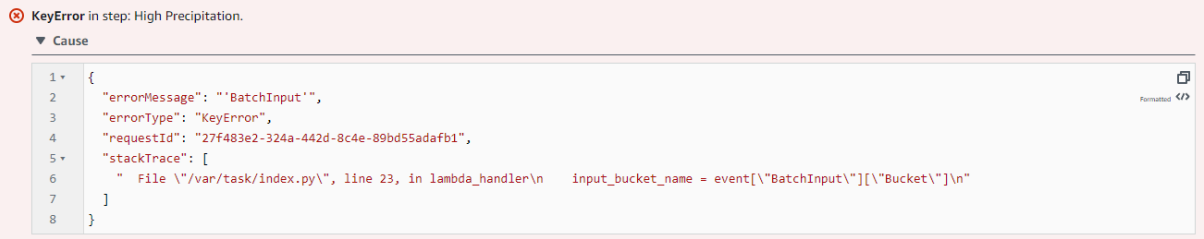
Se crea finalmente el state machine usando el rol StatesHighPrecipitation. Ahora se ejecuta usando el nombre del bucket que contiene la data y el prefijo correspondiente.



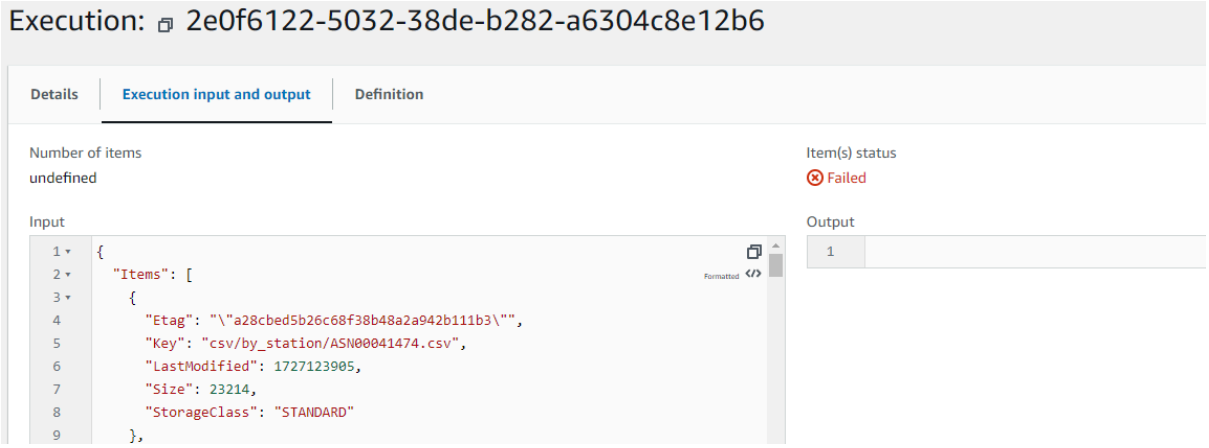
La ejecución falla y se obtiene el siguiente error.



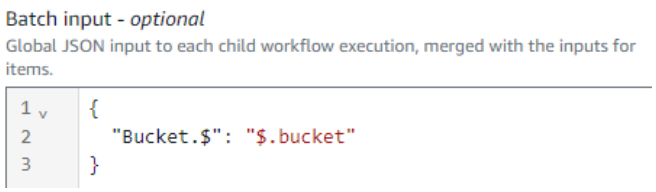
Al consultar el Map Run, se valida una de las ejecuciones y el error indica que se estaba esperando el valor event[BatchInput][Bucket] pero no se encontró.



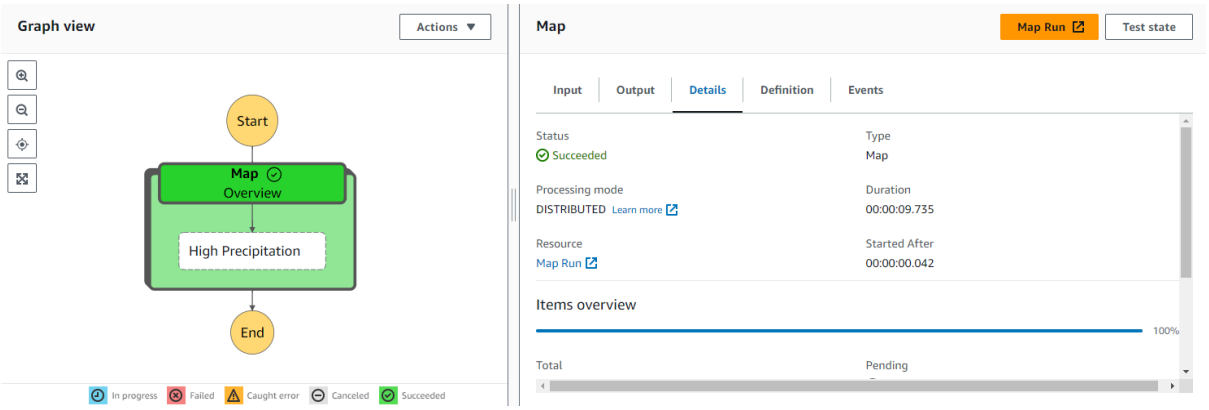
The input only contains the S3 key. It is missing the bucket name.



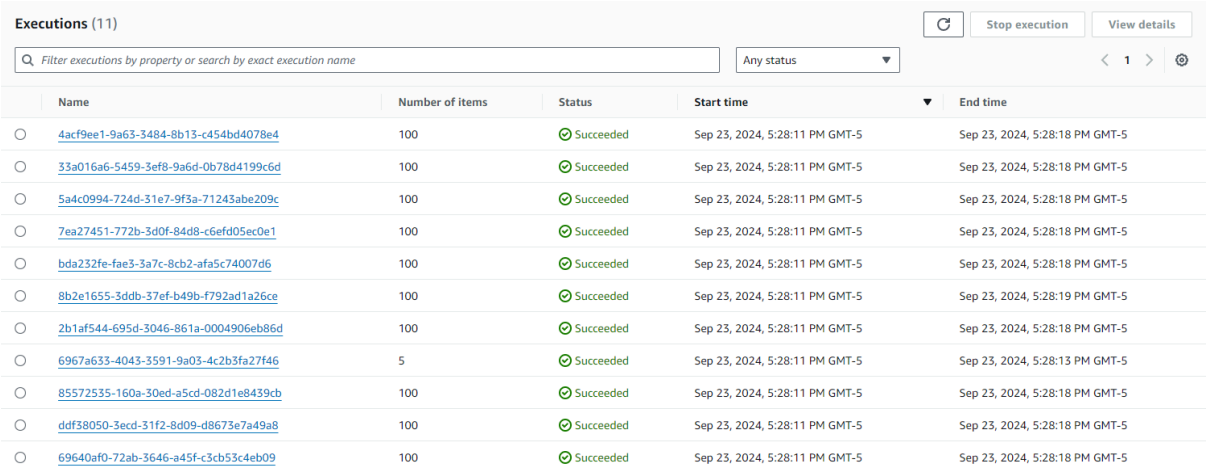
Se edita el Map state en la sección de Batch input, agregando el nombre del bucket a cada ejecución de cada child workflow.



Se ejecuta nuevamente el state machine, el cual concluye satisfactoriamente.



Al tener un batch size de 1000 y un tamaño de 1005 registros, se realizan 11 ejecuciones (child workflows).



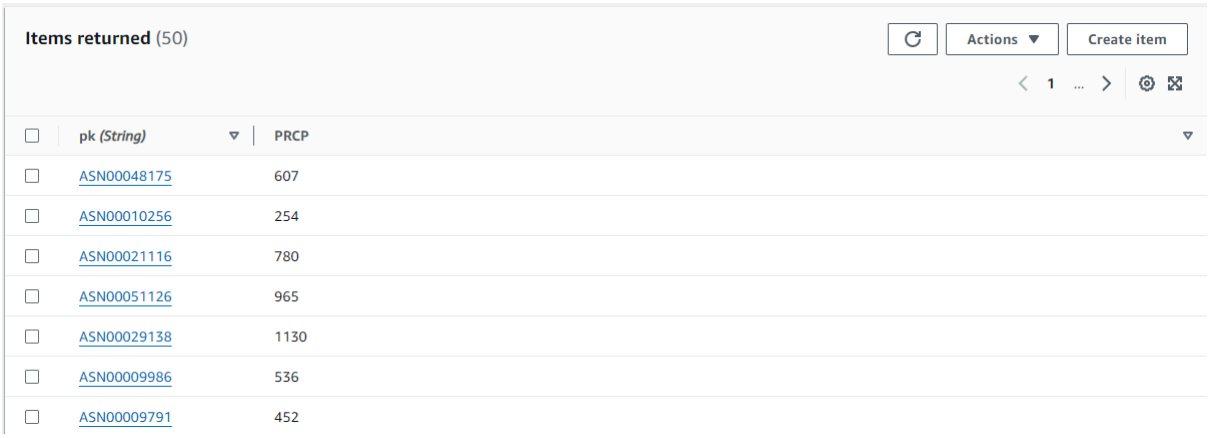
Executions (11)

Stop execution | View details

Filter executions by property or search by exact execution name | Any status | 1

Name	Number of items	Status	Start time	End time
4acf9ee1-9a63-3484-8b13-c454bd4078e4	100	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:18 PM GMT-5
33a016a6-5459-3ef8-9a6d-0b78d4199c6d	100	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:18 PM GMT-5
5a4c0994-724d-31e7-9f3a-71243abe209c	100	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:18 PM GMT-5
7ea27451-772b-3d0f-84d8-c6efd05ec0e1	100	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:18 PM GMT-5
bda232fe-fae3-3a7c-8cb2-afa5c74007d6	100	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:18 PM GMT-5
8b2e1655-3ddb-37ef-b49b-f792ad1a26ce	100	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:19 PM GMT-5
2b1af544-695d-3046-861a-0004906eb86d	100	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:18 PM GMT-5
6967a633-4043-3591-9a03-4c2b3fa27f46	5	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:13 PM GMT-5
85572535-160a-30ed-a5cd-082d1e8439cb	100	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:18 PM GMT-5
ddf38050-3ecd-31f2-8d09-d8673e7a49a8	100	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:18 PM GMT-5
69640af0-72ab-3646-a45f-c3cb53c4eb09	100	Succeeded	Sep 23, 2024, 5:28:11 PM GMT-5	Sep 23, 2024, 5:28:18 PM GMT-5

Ahora se consulta la tabla en DynamoDB. You can now view the calculated highest precipitation across stations.



Items returned (50)

Actions | Create item

1 ...

pk *(String)*	PRCP
ASN00048175	607
ASN00010256	254
ASN00021116	780
ASN00051126	965
ASN00029138	1130
ASN00009986	536
ASN00009791	452

Important

When processing large numbers of objects in S3 with Distributed Map, you have a couple of different options for listing those objects: S3 *listObjectsV2* and S3 Inventory List. With S3 *listObjectsV2*, Step Functions is making S3 *listObjectsV2* API calls on your behalf to retrieve all of the items needed to run the Distributed Map. Each call to *listObjectsV2* can only return a maximum of 1000 S3 objects. This means that if you have 2,000,000 objects to process, Step Functions has to make at least 2000 API calls. This API is fast and it won't take too long, but if you have an S3 Inventory file that has all the objects listed in it that you need to process, you can use that as the input.

Using an S3 Inventory file as the input for a Distributed Map when processing large numbers of files is faster than S3 *listObjectsV2*. This is because, for S3 Inventory ItemReaders, there is a single S3 *getObject* call to get the manifest file and then one call for each Inventory file. If you know that your Distributed Map is going to run on a set schedule you can schedule the S3 Inventory to be created ahead of time.

Advanced

Optimization

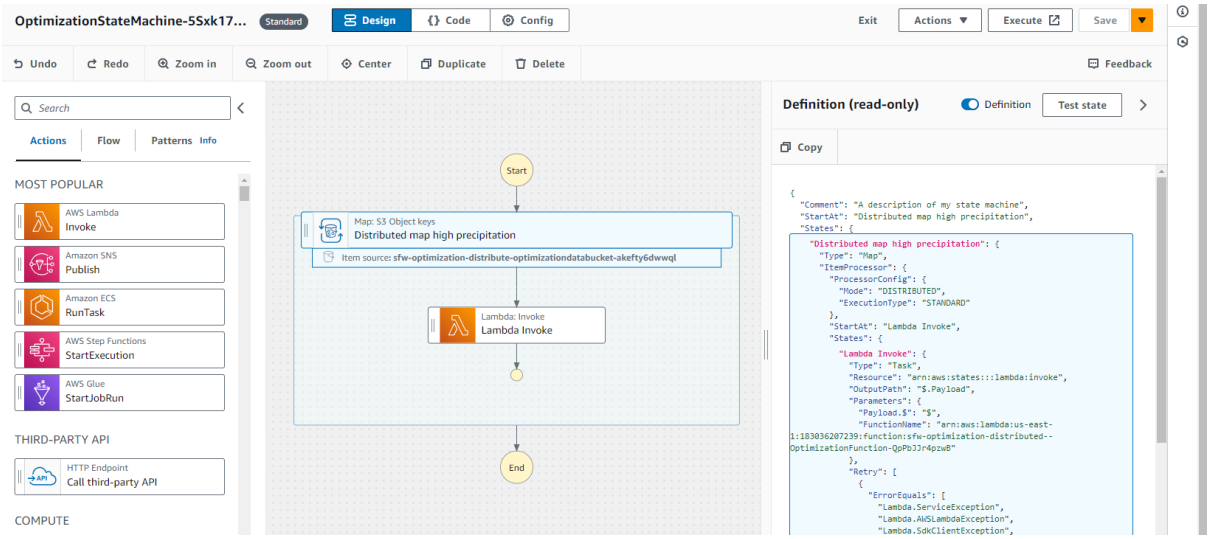
Workflow types

Step Functions offers two types of workflows - **Standard** and **Express**. Standard workflows are ideal for a long running workflow; it can run for 365 days whereas express workflows can run only for 5 minutes. Another important distinction is how pricing works. Standard workflows are priced by state transition while express workflows are priced by number of requests and the duration.

When you use distributed map, Step Functions spins up child workflows to run the states inside the distributed map. The number of child workflows to spin up is dependent on the number of objects or records to process, batch size and concurrency. You can define the child workflow to run as either standard or express based on your use case.

Se toma el template de CloudFormation proporcionado por AWS (region us-east-1).

Se ingresa a la state machine llamada OptimizationStateMachine. Como se puede ver, tiene un Map state en modo Distributed, utilizando un batch de 100 objetos.



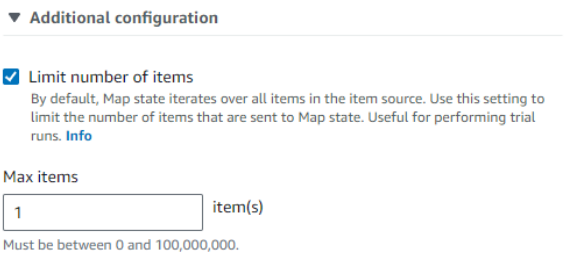
Identify if workflow can be Express

Child workflow can be run either STANDARD or EXPRESS. Express workflows are generally less expensive and run faster than Standard workflows.

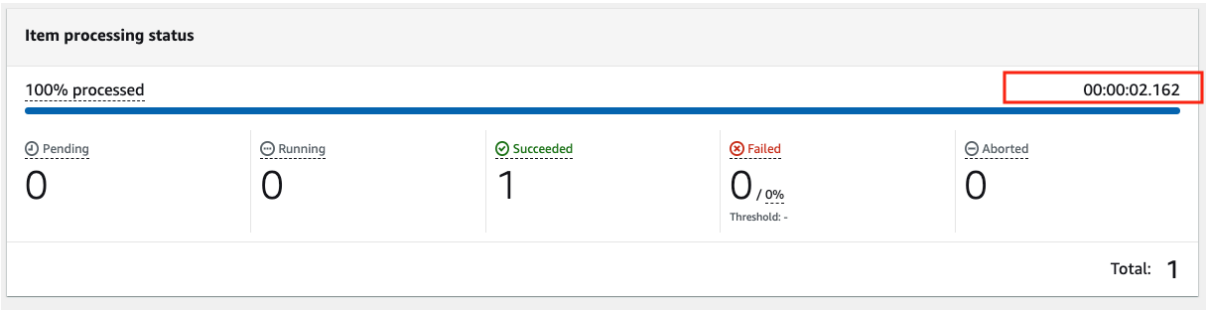
Sometimes, you may not be sure if your workflow runs within 5 minutes. In this section, you are going to use a feature of distributed map that allows you to test your data with a small number of items. This technique is helpful in couple of ways

- To determine the duration of the child workflow
- To gain confidence that the child workflow logic will run fine when running with a full data set.

En la configuración del Map state se establece un número límite de ítems para hacer la prueba. En este caso 1 ítem.



Se ejecuta el state machine y se puede ver que tarda un poco más de 2 segundos.



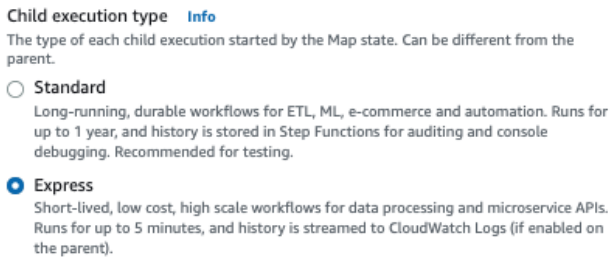
Explore the Map Run page to find the duration for 100 items. It is less than 30 seconds.

Now you know it takes 3 seconds to run 1 item and 26 seconds for 100 items, you can even run all the 1000 items in a single child workflow with 1 concurrency. But, you don't utilize any parallelism to speed up the process.

This simple test is really handy in finding the right batch size and choosing the workflow type without running the entire dataset!!!

Changing the workflow type to Express

If you now run this workflow as a Standard workflow type and compare the durations, you will notice the Express workflow execution was faster.



Review Cost impact

Consider you are processing 500K objects and set the batch size to 500.

500k objects / 500 objects per workflow = **1000** child workflows

Distributed map runs a total of 1000 child workflows to process 500K objects.

Standard child workflow execution cost

Total cost = (number of transitions per execution x number of executions) x $0.000025

Tener cuidado con dejar la misma cantidad de workflows que registros. Puede haber un sobrecosto inesperado.

Let's take another scenario. You have 2 steps inside your child workflow, the number of state transitions per child workflow is 3. Let's assume you can not utilize batching, the number of child workflows to complete the work = 500K

Total cost = (number of transitions per execution x number of executions) x $0.000025

Total cost = (3 * 500K) x $0.000025 = **$37.4**

Express child workflow execution cost

With Express workflows, you pay for the number of requests for your workflow and the duration. With scenario outlined earlier under Review cost impact, we need additional dimension of how long the workflow runs to calculate the express workflow cost. Let's assume express child workflow runs for an average of 100 sec to process 500 objects using 64-MB memory.

Duration cost = (Avg billed duration ms / 100) * 0.0000001042

Duration cost = (100,000 MS /100) * $ 0.0000001042 = $0.0001042

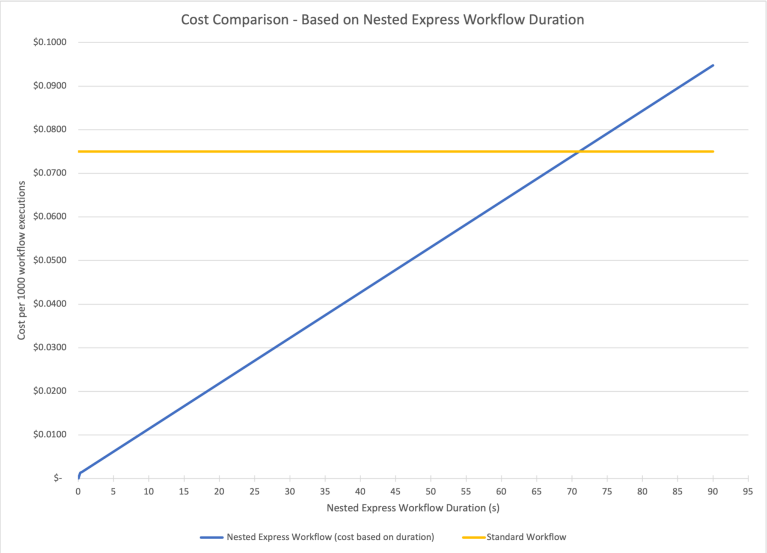
Express request cost = $0.000001 per request ($1.00 per 1M requests)

workflow cost = (Express request cost + Duration cost) x Number of Requests

workflow cost = ($0.000001 + $0.0001042) x 1000 = **$0.10**

If we repeat the calculation for an express workflow that runs for 1 second to process 1 object because you cannot utilize batching, the total cost = $13.42.

Express workflows are cheaper when the duration is lesser. They are also cost effective if there are more steps in the child workflow or your distributed map cannot make use of batching. Remember, Standard workflows are priced by state transitions meaning when number of steps and number of child workflow executions increase, cost increases.

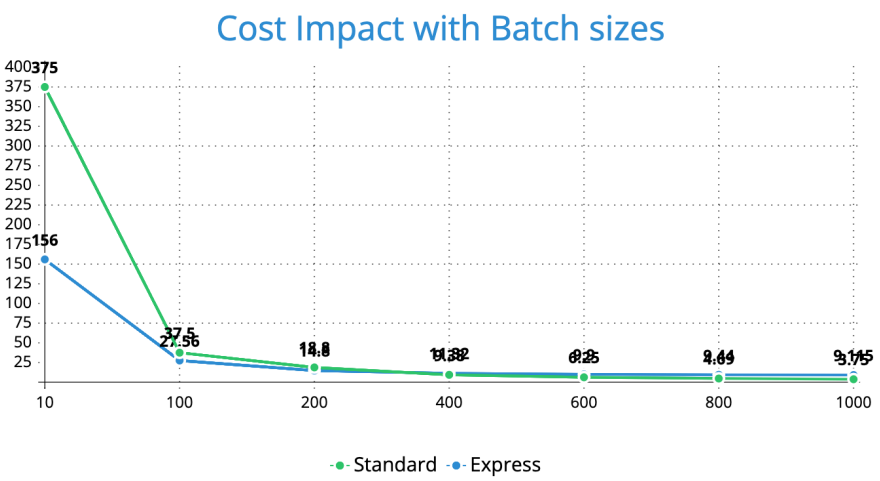


Higher parallelism causes scaling bottlenecks for downstream services inside the child workflow. You can use distributed map concurrency control to control the number of parallel workflows. If you have multiple workflows and need to manage downstream scaling then you can use techniques such as **queueing** and **activities**.

Express vs Standard vs Batch size

A continuación se puede ver la reducción significativa de costos al elegir un batch size adecuado.

Batchsize	child workflows	Child workflow Duration(ms)	Standard cost	Express cost
100	500,000	28,000	$37.5	$27.59
200	250,000	32,000	$18.75	$14.84
400	125,000	62,000	$9.38	$11.33
500	100,000	74,000	$7.5	$10.31
600	83,333	90,000	$6.25	$9.98
800	62,500	120,000	$4.69	$9.44
1000	50,000	150,000	$3.75	$9.12



What did you observe?

1. More parallelism (less batching) speeds up the process.
2. Increasing batch size reduces cost.
3. Generally, if you have to go for lower batch sizes due to data size or other reasons and the child workflow can be run within 5 minutes, you should go for Express.

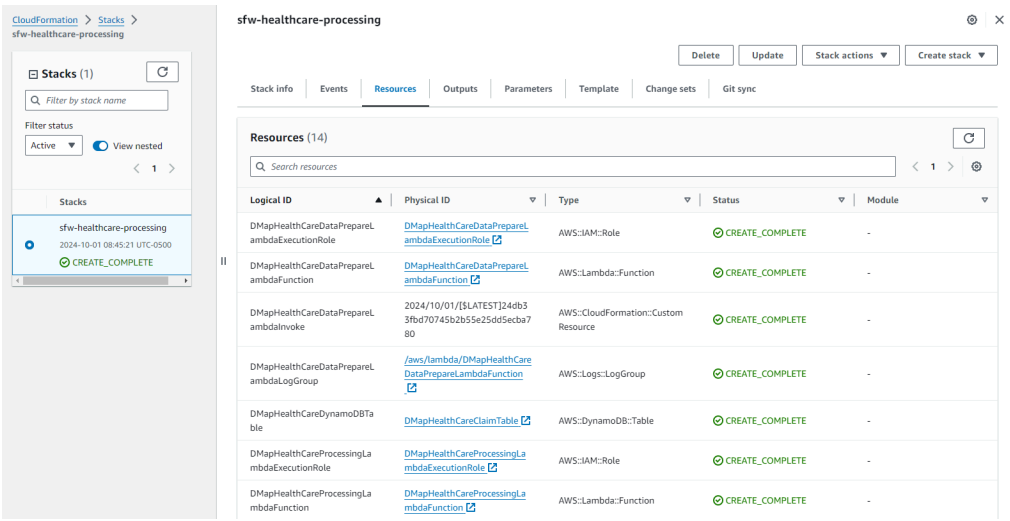
Important

The degree of parallelism is determined by the Concurrency setting in Distributed Map, which determines the maximum number of parallel child workflows you want to execute at once. A key consideration here is the service quotas of the AWS services called in your child workflow. For example AWS Lambda in most large regions has a default concurrency quota of 1500 and a default burst limit of 3000, other services such as AWS Rekognition or AWS Textract have much lower default quotas.

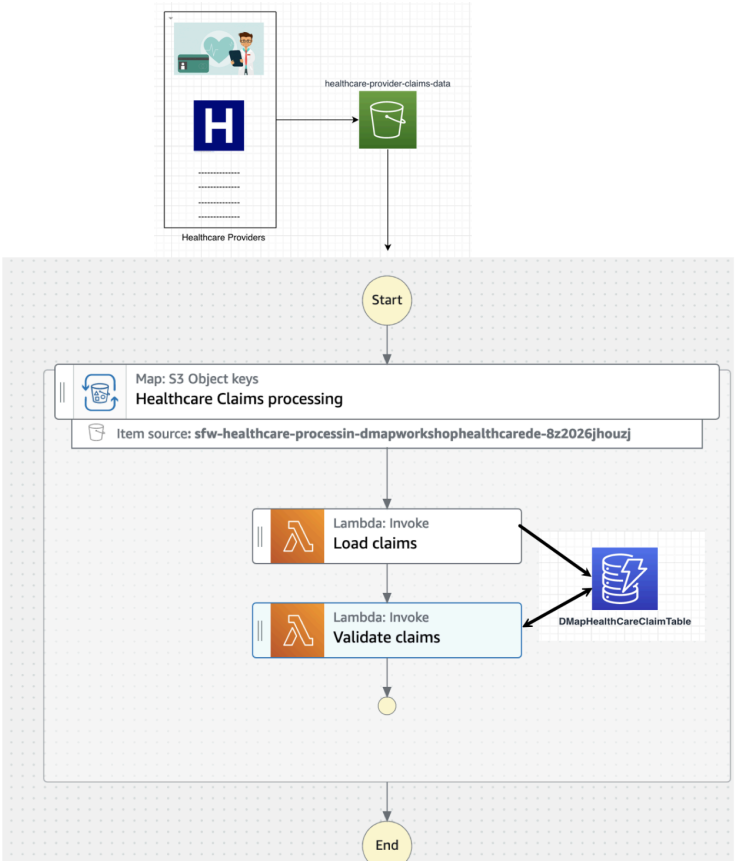
The other thing to keep in mind is any performance limitations of other systems that your child workflow interacts with. An example here would be an on-prem relational database that Lambda within a child workflow connects to. This database might have a limit to the number of connections it can support, so you would need to limit your concurrency accordingly. Once you identify all of the AWS Service quotas and any additional concurrency limitation you'll want to test various combinations of batch size and concurrency to find the best performance within your concurrency constraints

Use Case - Healthcare Claims Processing

Se toma el template de CloudFormation proporcionado por AWS (region us-east-1).



You will build a Step Functions workflow that processes healthcare claims data in a highly parallel fashion. The workflow uses the Distributed Map state that runs multiple child workflows, each processing a batch of the overall claims data. Each child workflow picks a set of individual claims files and processes them using AWS Lambda functions that load the data to an Amazon DynamoDB table and then apply rules to determine validity of the claims. Upon processing the claims, the function returns the output back to the workflow.

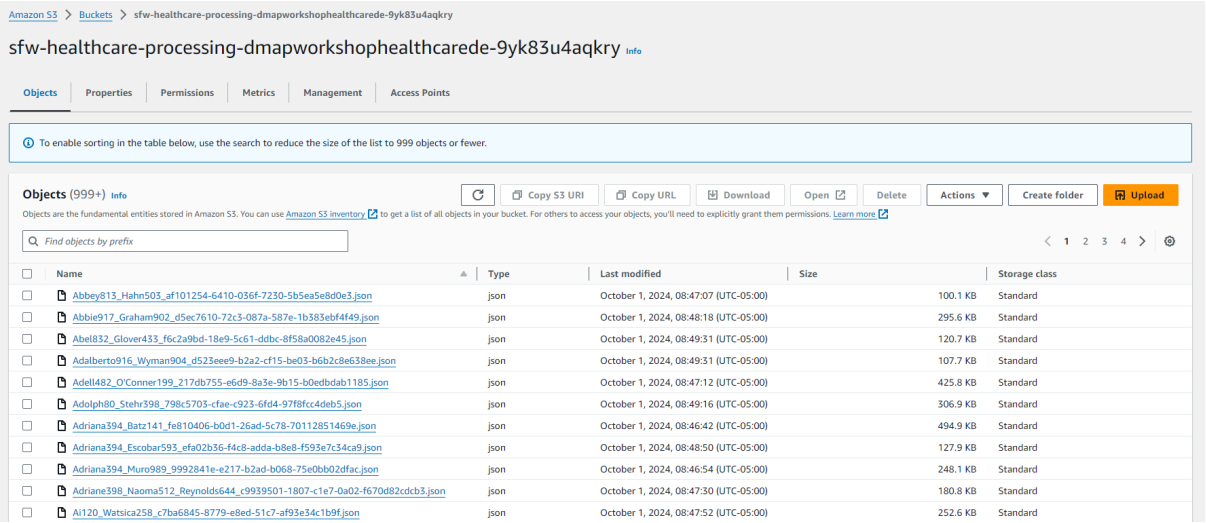


Data processing code in the following AWS Lambda functions:

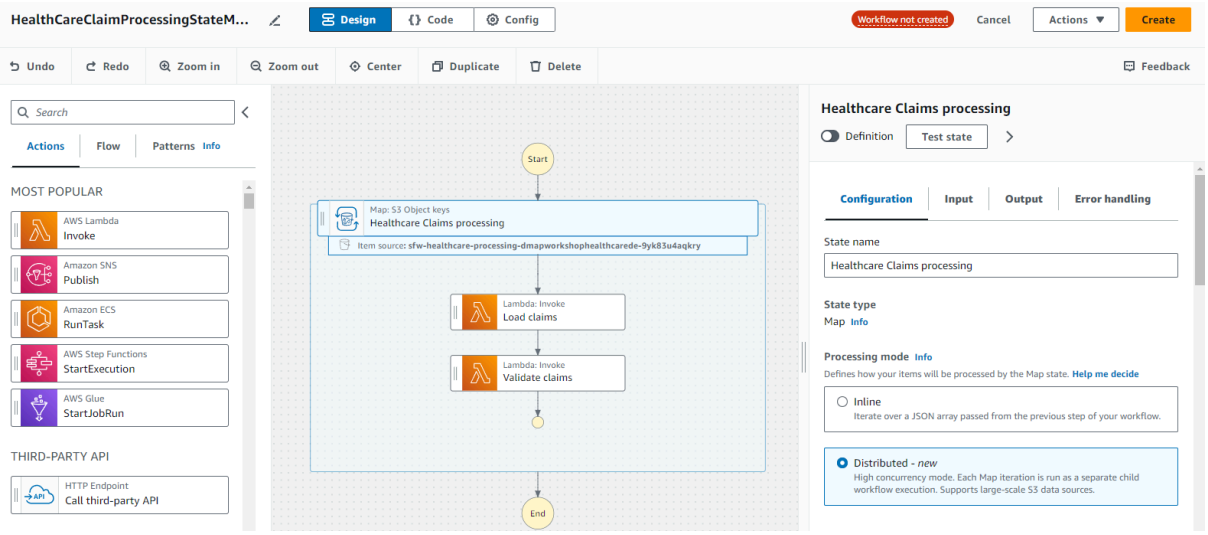
- **DMapHealthCareClaimProcessingFunction**: This function reads a claims file and stores data in an Amazon DynamoDB table (DMapHealthCareClaimTable).

- **DMapHealthCareRuleEngineLambdaFunction**: This function reads data from the Amazon DynamoDB table and applies rules to determine whether the claims need to be accepted or rejected and returns the output. If it is rejected, it will return the reason as well.

Navigate to the S3 Bucket. Search for **dmapworkshophealthcare** bucket. This bucket contains 1026 JSON files with around 60,000 records, with a total size of 270MB. These JSON files are generated using Synthea Health Library to simulate patient claims data in FHIR format.

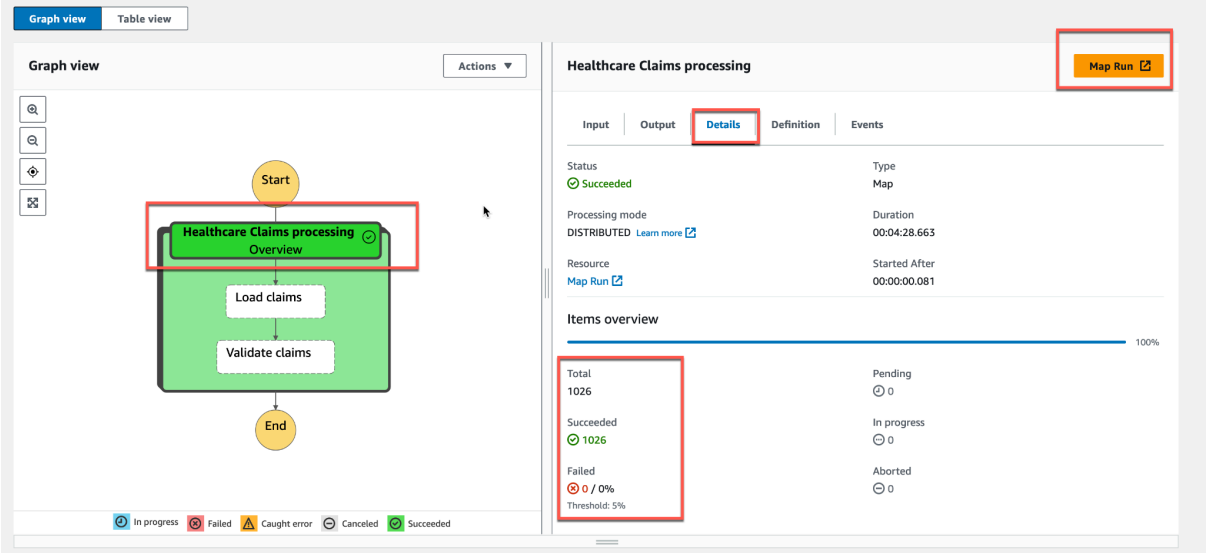


Se ingresa al servicio de Step Functions y crea una state machine en blanco. Inicialmente se coloca un Map state y se le asigna la configuración proporcionada en el workshop. A su vez, se modifica la función Lambda que viene por defecto y se agrega una nueva, apuntando a las funciones que se mencionan en el workshop. El diseño debería quedar así:

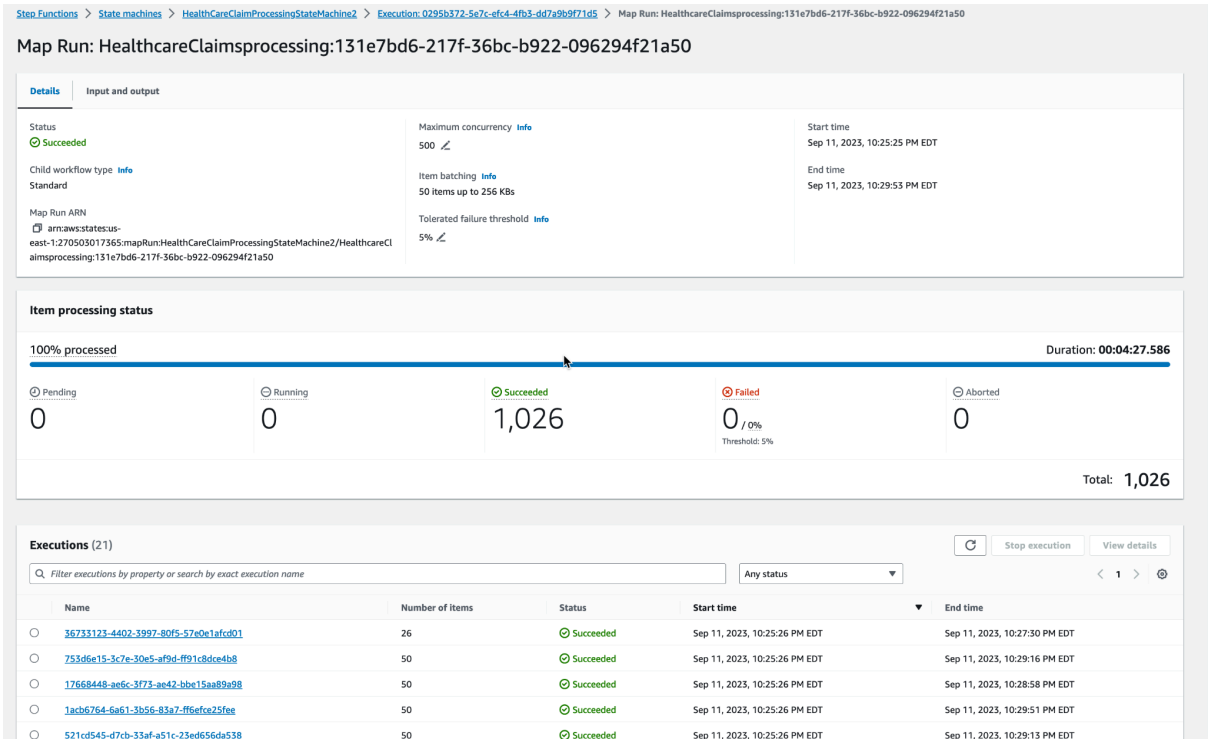


En cuanto a la configuración, se deja el nombre HealthCareClaimProcessingStateMachine y el rol sfw-healthcare-processing-HealthcareClaimProcessing-W0LRuaKKCfZF. Lo demás no se modifica.

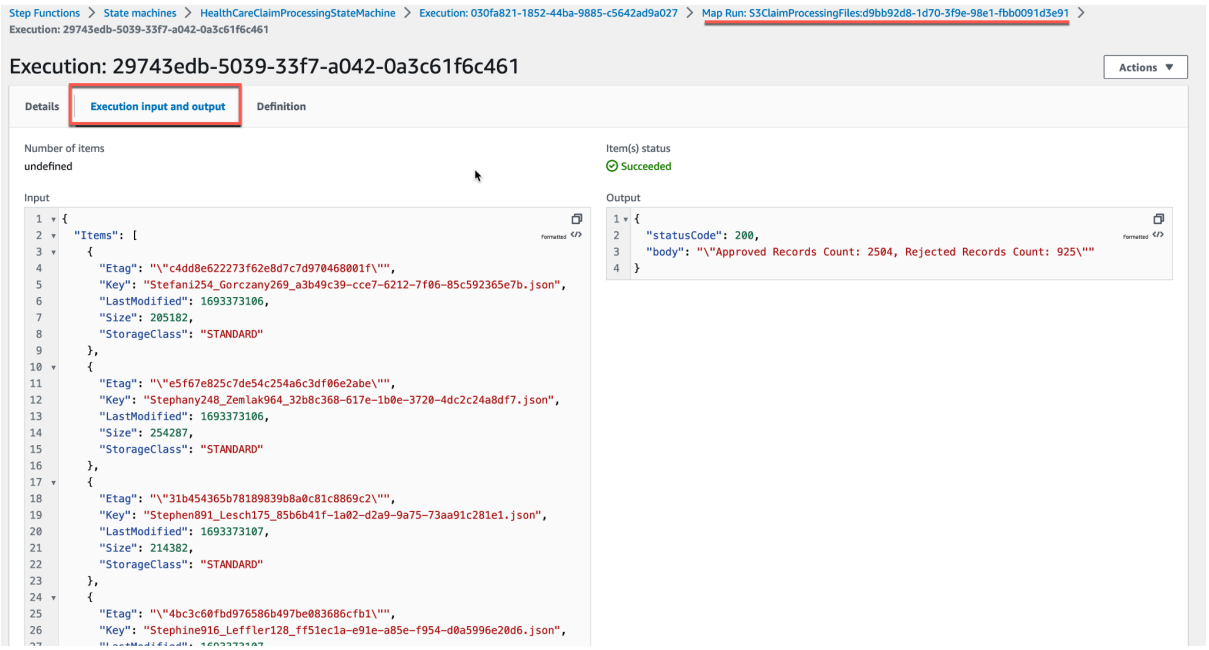
El proceso se ejecuta correctamente. Ahora se abre el Map Run para tener más detalle de cómo se ejecutó cada child workflow.



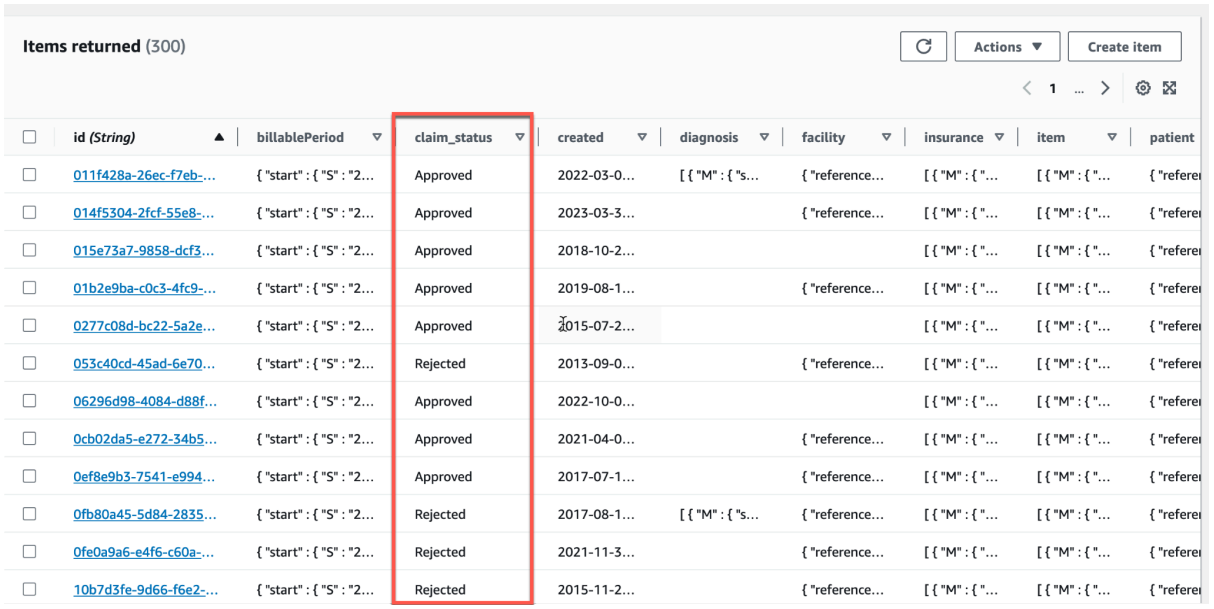
We can see that 21 child workflow executions completed successfully with 0 failures. Each child workflow processed 50 files. We can view the duration of each child workflow execution. You can see overlapping timestamps for the start and end times, indicating that the data was processed in parallel.



If you select the execution name, you can use the Execution Input and output tab to view the input files for a child workflow execution and the execution output with details.



Se verifica el resultado en la tabla de DynamoDB creada con el stack de CloudFormation.



You used Distributed Map state to quickly process a large dataset using parallel processing.

- Increase the concurrency limit to 1000 and execute it again. Does it change the duration of the execution?
- What happens if you decrease the Item Batching size to 25 and execute the workflow? What is the impact on duration as well as cost?
- What combination of concurrency limit and batching size would be optimal?
- What happens if you change the type of the workflow to 'Express' and execute it? What is the impact on cost? Would this workflow type work for any batching size of the provided data set?

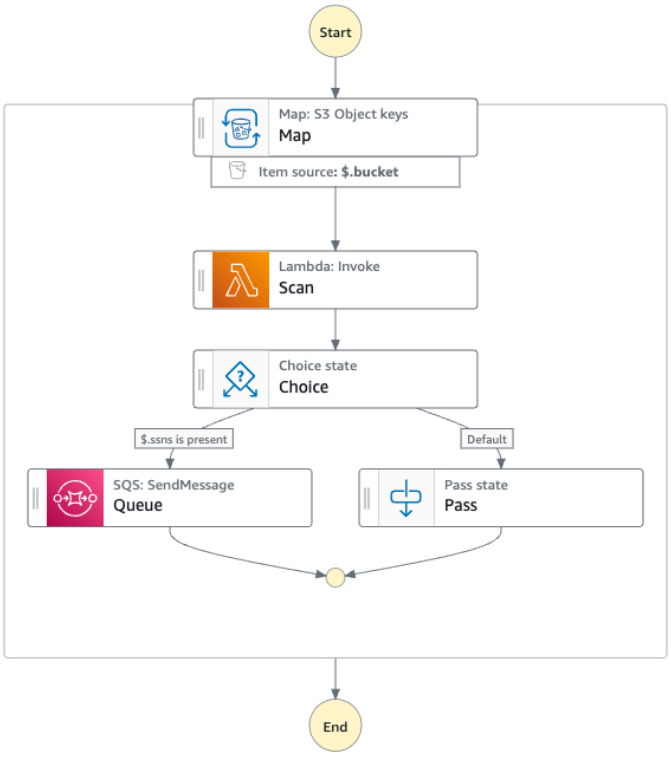
Use Case - Security Vulnerability Scanning

Se prueba desplegando el stack de CloudFormation proporcionado en el workshop pero el tamaño de la memoria (MemorySize: 10240) en la función Lambda excede las 3GB que tiene una cuenta común de AWS.

2024-10-01 13:59:24 UTC-0500	VulnerabilityScanningFunction	CREATE_FAILED	-	Resource handler returned message: "'MemorySize' value failed to satisfy constraint: Member must have value less than or equal to 3008 (Service: Lambda, Status Code: 400, Request ID: 6bea8ba0-7863-4003-bca8-335887364a35)" (RequestToken: 037cc46f-88c7-437e-9004-2956c0f9fdb4, HandlerErrorCode: InvalidRequest)

Se sigue el lab con la información del workshop.

You are developing a security vulnerability scanning application that alerts you of sensitive information in plain text files. The application you have built executes a workflow that scans a single file for exposed social security numbers (SSNs) and, if one is detected, sends a message to a queue for further downstream processing.



Para esta primera ejecución se toma el nombre del bucket y una key de uno de los objetos.

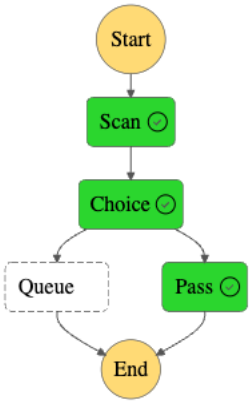
Input - *optional*
Enter input values for this execution in JSON format

```
{
    "detail": {
        "bucket": {
            "name": "vulnerability-scanning-mo-vulnerabilitydatabucket-1r4mp2l1c5lu6"
        },
        "object": {
            "key": "00000404-1368-45e3-b060-0ba6039eed48.txt"
        }
    }
}
```

Open in a new browser tab

Cancel | Start execution

En este caso el proceso se completó sin reconocer algún SSN, por lo que el Choice state elige el Pass state para finalizar.



Limiting the number of items that are sent to the Map state is useful for performing trial executions.

Se modifica la configuración del Map state estableciendo un máximo de items de 1000.

Modify items before they are passed on to child executions, selecting only the relevant items and adding input where needed with ItemSelector.

12. Select **Modify Items with ItemSelector** then enter the following JSON:

```
{
    "detail": {
        "bucket": {
            "name.$": "$.bucket"
        },
        "object": {
            "key.$": "$$.Map.Item.Value.Key"
        }
    }
}
```

Express Workflows are ideal for quick, high-volume workloads like this. They can run for up to five minutes.

Se debe cambiar Child execution type, pasará de Standard a Express.

Ahora, se hace una nueva ejecución con el siguiente payload.

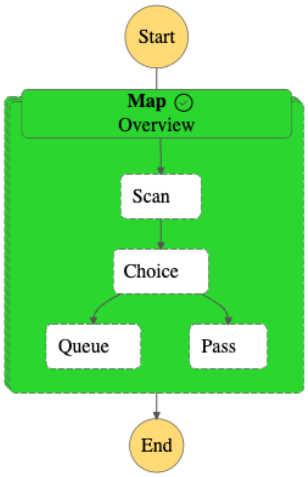
Input - *optional*
Enter input values for this execution in JSON format

```
{
    "bucket": "vulnerability-scanning-mo-vulnerabilitydatabucket-1r4mp2l1c5lu6",
    "prefix": ""
}
```

Open in a new browser tab Cancel Start execution

Because you added a limitation on the number of items processed, under Executions, there are only 1000 workflows running despite there being more files in S3.

El resultado final debería ser el siguiente.



Al dirigirse a la queue creada con CloudFormation, se puede ver los mensajes con los SNSs identificados.

The body of the message contains the location and serial number of the SSN(s).

Luego se da click en Purge para eliminar todos los mensajes que llegaron (depurar).

Ahora, se utiliza la herramienta de batching en el Map state, habilitando la opción **Max MBs per batch** a 50 KBs.

With batch input, you can pass a global JSON input to each child execution, merged with the inputs for items. In the last section, the bucket name was included in every single item, increasing the total input size. Instead of including the bucket name repeatedly with ItemSelector, you will include it once in the batch input.

6. Enter the following **Batch input**:

```
{
  "bucket.$": "$.bucket"
}
```

A su vez, se cambia la siguiente configuración.

7. Under **Item source**, expand **Additional configuration**.
8. Under **Modify items with ItemSelector**, overwrite the JSON with the following, removing details of the bucket:

```
{
  "key.$": "$$.Map.Item.Value.Key"
}
```

Express Workflows only run for up to five minutes and batch processing increases the number of files processed per child execution. If you configure a sufficiently large batch size, you may need to use a Standard Workflow.

Se cambia el tipo de child workflow a Standard.

Se actualiza el código de la función Lambda.

```python
import json
import boto3
import re

def handler(event, context):

  bucket = event["BatchInput"]["bucket"]

  ssns = []
  for item in event["Items"]:

    key = item["key"]

    obj = boto3.client('s3').get_object(
      Bucket=bucket,
      Key=key
    )
    body = obj['Body'].read().decode()

    searches = re.findall("ssn=[^\s]+", body)
    if searches:
      ssns.extend([{"key": key, "serial": number[-4:]}
        for ssn, number in (search.split("=") for search in searches)
      ])

  if ssns:
    return {"ssns": ssns}
  else:
    return {}
```

Finalmente se ejecuta con el siguiente payload.

Input - *optional*

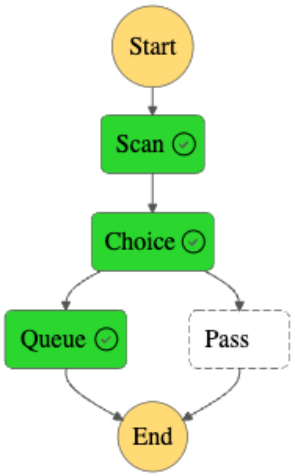
Enter input values for this execution in JSON format

```
{
    "bucket": "vulnerability-scanning-mo-vulnerabilitydatabucket-1r4mp2l1c5lu6",
    "prefix": ""
}
```

Open in a new browser tab

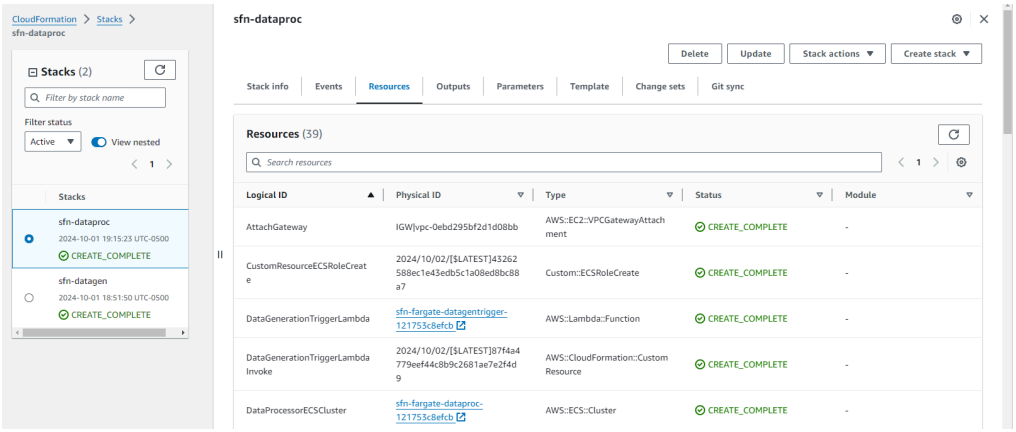
Cancel | Start execution

Al detallar uno de los ítems, se puede ver cómo termina en el state de SQS.



Use Case - Monte Carlo Simulation

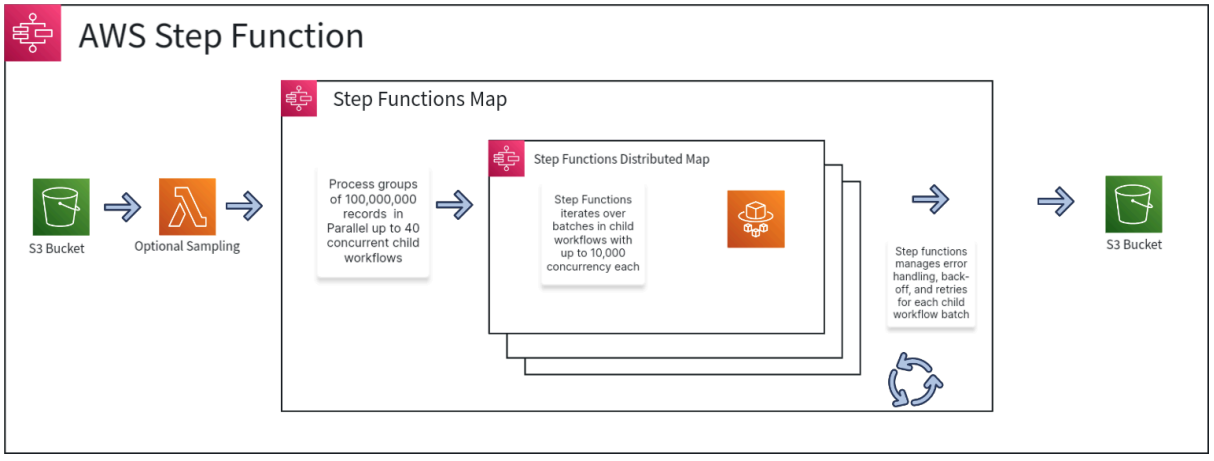
Se despliegan los 2 stacks que corresponden a la región us-east-1: Data Generation Stack y Data Processing Stack.



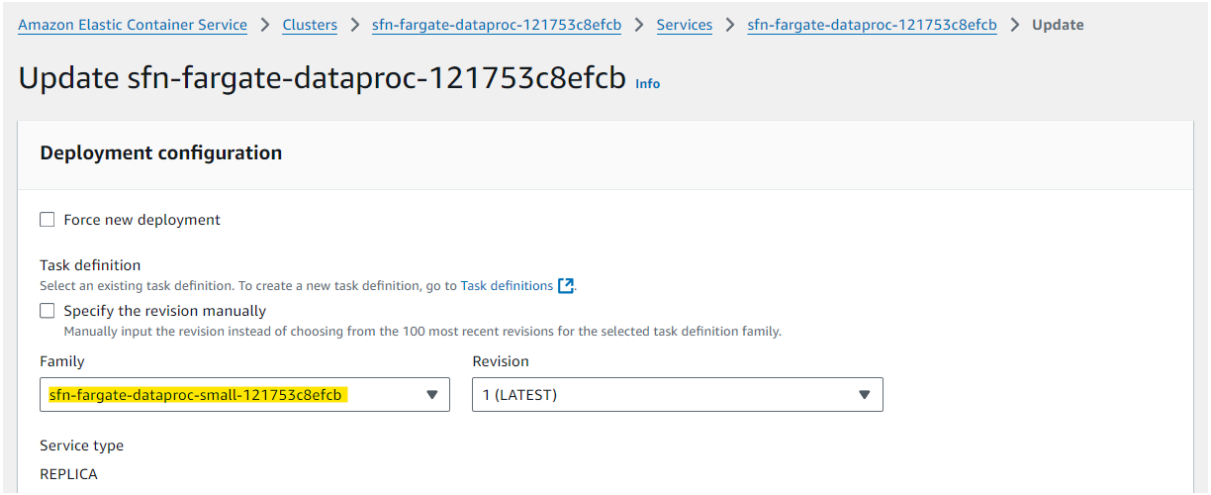
For this fictitious use case we will be working with a portfolio of personal and commercial loans owned by our company. Each loan is represented by a subset of data housed in individual S3 objects. Our company has tasked us with trying to predict which loans will default in the event of a Federal Reserve rate increase. Loan defaults occur when the borrower fails to repay the loan. Predicting which loans in a portfolio would default in various scenarios helps companies understand their risk and plan for future events.

In this solution we are distributing the data using a **Step Functions Activity**. Activities are an AWS Step Functions feature that enables you to have a task in your state machine where the work is performed by a worker that can be hosted on Amazon Elastic Compute Cloud (Amazon EC2), Amazon Elastic Container Service (Amazon ECS), mobile devices—basically anywhere. Think of activity as a Step Functions managed internal queue. You use an activity state to send data to the queue, one or more workers will consume the data from the queue. For this solution we utilize Amazon ECS on Amazon Fargate to run our Activity Workers (Worker).

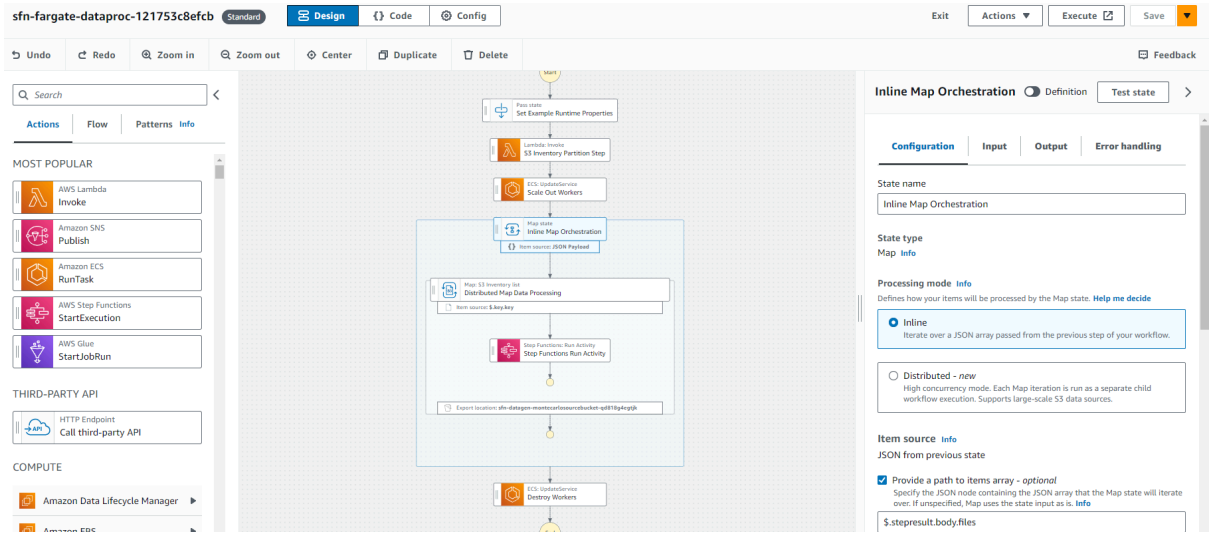
The solution uses AWS Step Functions to provide end to end orchestration for processing billions of records with your simulation or transformation logic using AWS Step Functions Distributed Map and Activity features. At the start of the workflow, Step Functions will scale the number of workers to a (configurable) predefined number. It then reads in the dataset and distributes metadata about the dataset in batches to the Activity. The workers are polling the Activity looking for data to process. Upon receiving a batch, the worker will process the data and report back to Step Functions that the batch has been completed. This cycle continues until all records from the dataset have been processed. Upon completion, Step Functions will scale the workers back to zero.



Se debe actualizar el service del cluster creado por el stack de CloudFormation. La idea es apuntar a la task definition llamada sfn-fargate-small.



La state machine a ejecutar se puede ver a continuación.



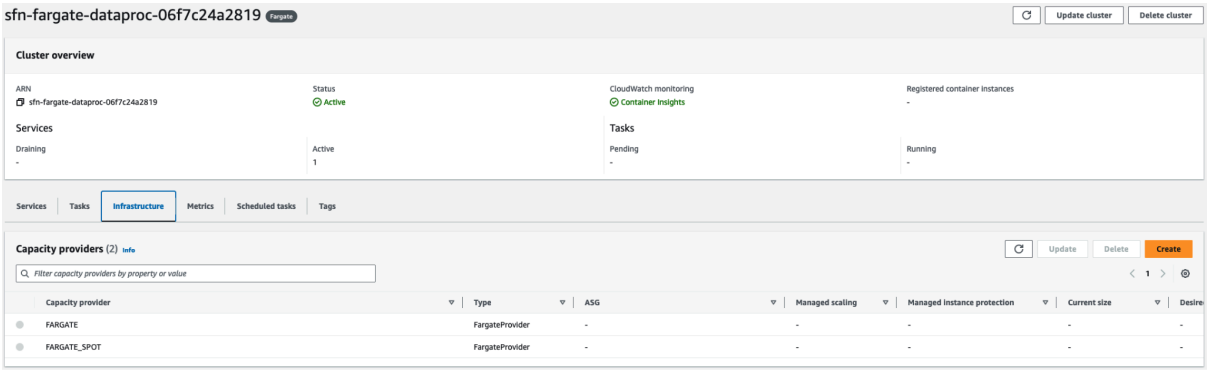
Ahora se explica más a detalle para que se usa cada componente o state step en la state machine.

Distributed Map

The Processing DMap step is an AWS Step Functions Distributed Map step that reads the S3 Inventory manifest provided by the Parent Map and processes the referenced S3 Inventory files. For this use case each line of the S3 Inventory file contains metadata referencing an object is S3 containing a single customer loan. The Distributed Map feature creates batches of 400 loan files per our configuration for concurrent distribution to the Step Functions Activity step. Each Distributed Map step supports up to ten thousand concurrent workers. For this example, the Runtime Concurrency is set to 1000. As Step Functions adds messages to the Activity, the workers are polling to pull batches for processing. Once complete, the worker reports back to Step Functions to acknowledge the batch has been completed. Step Functions removes that message from the Activity and adds a new batch, it will repeat this process until all batches have been completed.

Amazon ECS Workers

The Amazon Elastic Container Service (ECS) Cluster is using a Fargate Spot Capacity Provider to reduce costs and eliminate maintaining EC2 instances. Fargate provides us with an AWS managed compute for scheduling our containers.

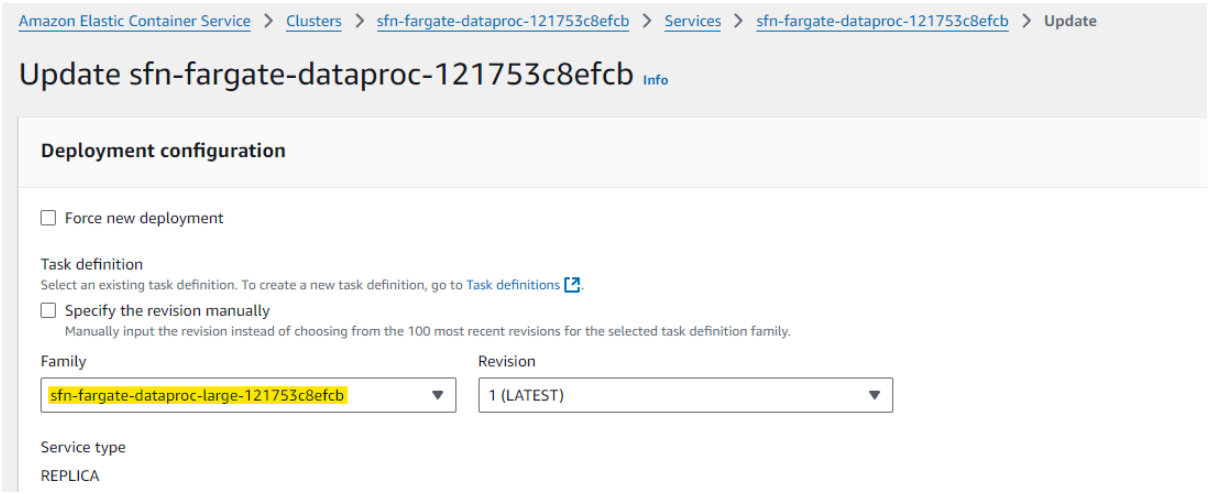


Data Processing

The Run Activity step consists of a single Step Functions Activity which accepts a JSON payload from Distributed Map containing a batch of Loan objects stored in S3. The Run Activity step will continuously add/remove batches of loans by reading the contents of the S3 object. These batches are picked up by our workers and processed. After the worker reports that a batch was successfully processed, Step Functions will remove the batch from the Activity and add a new one in its place, maintaining our concurrency limit of batches until all batches are processed. In this example the output files contain batched loans to facilitate more efficient reads for analytics and ML workloads.

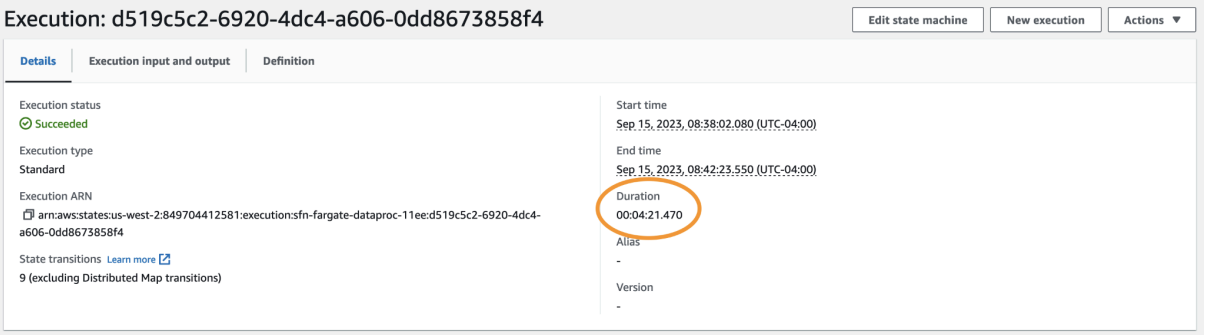
Step Functions error handling features removes the need to implement catch and retry logic within the python code. If a batch fails processing or a worker fails to report the batch as complete, Step Functions will wait the allotted time and re-add the batch for another worker to pick up and process.

Se vuelve a actualizar el service del cluster. La idea es apuntar a la task definition llamada sfn-fargate-large, la cual cuenta con el doble de recursos que la small.



Se ejecuta nuevamente la state machine.

In the first execution you used a Task Definition that allocated .25 vCPU's and .5GB of RAM to each container. In the second execution you used a Task Definition that allocated .5 vCPU's and 1GB of RAM to each container. Let's check out the results and see how they differ.



Do they differ? For this simulated dataset, which would be considered small for a Monte Carlo Simulation, the difference is typically less than a minute but overall the increased vCPU/RAM will lead to ~25% time savings. However, to get this ~25% savings you had to double the resources for each container, effectively doubling your Fargate costs. If time is your most critical factor, this may be worth it. If cost is your most critical factor, perhaps not. The variation for both cost and time are minimal with a dataset this small, however if you extrapolate this to a dataset consisting of millions or even billions of objects, both time and cost variations are considerable. You will want to experiment with your actual workloads on what settings work best.