When we discuss how to build applications with customers, we often align to the [Well Architected Framework](https://d0.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf) pillars of security, reliability, performance efficiency, cost optimization, and operational excellence. Designing for failure is an essential component to developing well architected applications that are resilient to spurious errors that may occur.

There are many ways you can use AWS services to achieve high availability and resiliency of your applications. For example, you can couple Elastic Load Balancing with Auto Scaling of Amazon EC2 instances to build highly available applications or use Amazon API Gateway and AWS Lambda to rapidly scale out a microservices-based architecture. Many AWS services have built in solutions to help with the appropriate error handling, such [as Dead Letter Queues (DLQ)](http://docs.aws.amazon.com/lambda/latest/dg/dlq.html) for AWS Lambda or [retries in AWS Batch](http://docs.aws.amazon.com/batch/latest/userguide/job_retries.html).

AWS Step Functions is an AWS service that makes it easy for you to coordinate components of distributed applications and microservices. AWS Step Functions allows you to easily design for failure, by incorporating features such as [error retries](https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-errors.html#amazon-states-language-retrying-after-error) and [custom error handling](https://docs.aws.amazon.com/step-functions/latest/dg/tutorial-handling-error-conditions.html) from Lambda exceptions. These features allow you to programmatically handle many common error modes and build robust, reliable applications.

In some rare cases, however, your application may fail in an unexpected manner. In these situations, you might not want to duplicate in a repeat execution those portions of your state machine that have already run, whether orchestrating a long-running jobs or executing a complex state machine as part of a microservice. Here, you would need to know the last successful state in your state machine from which to resume, so that you don’t duplicate previous work. In this post, we present a solution to enable you to resume from any given state in your state machine in case of unexpected failure.

# Resuming from a given state

To resume a failed state machine execution from the state at which it failed, we will first run a script that dynamically creates a new state machine which, when executed, resumes the failed execution from the point of failure. Our script contains the following two primary steps:

1. Parses the execution history of our failed execution to find the name of the state at which it failed as well as the JSON input to that state.
2. Creates a new state machine, which adds an additional state to failed state machine, called the “GoToState”. The “GoToState” is a choice state at the beginning of the state machine that branches execution directly to the failed state, allowing you to skip states that had succeeded in the previous execution.

The full script along with a CloudFormation template that creates a demo of this is available here (Hyperlink To Github), so that you can try this out for yourself.

# Diving into the script

We will walk through the script and highlight the core components of its functionality.

Apart from these major components of the script, described in the following sections, it also contains a main function, which adds a command line parameter for the *failedExecutionArn*, so that we can easily call the script from the command line as below:

*python gotostate.py --failedExecutionArn '<Failed\_Execution\_Arn>'*

## Identifying the failed state in your execution

First, we extract the name of the failed state given the Amazon Resource Name (ARN) of the failed state machine execution history, along with the input to that state. The failed state is marked in the execution history, along with the input to that state, and the script is able to parse these values from the log.

We loop through the execution history of the failed state machine, and trace it backwards until we find the failed state. If the state machine failed in a parallel state, we must restart from the beginning of the parallel state. Our script is able to capture the name of the parallel state that failed, rather than any sub-state within the parallel state that may have caused the failure. Below is the python function we use to do this.

def parseFailureHistory(failedExecutionArn):

'''

Parses the execution history of a failed state machine to get the name of failed state and

the input to the failed state

Input failedExecutionArn - a string containing the execution Arn of a failed state machine

Output - a list with two elements: [name of failed state, input to failed state]

'''

failedAtParallelState = False

try:

#Get the execution history

response = client.get\_execution\_history(

executionArn=failedExecutionArn,

reverseOrder=True

)

failedEvents = response['events']

except Exception as ex:

raise ex

#Confrim that the execution actually failed, raise exception if it didn't fail

try:

failedEvents[0]['executionFailedEventDetails']

except:

raise('Execution did not fail')

'''

If we have a 'States.Runtime' error (for example if a task state in our state

machine attempts to execute a lambda function in a different region than the

state machine, get the id of the failed state, use id of the failed state to

determine failed state name and input

'''

if failedEvents[0]['executionFailedEventDetails']['error'] == 'States.Runtime':

failedId = int(filter(str.isdigit, str(failedEvents[0]['executionFailedEventDetails']['cause'].split()[13])))

failedState = failedEvents[-1 \* failedId]['stateEnteredEventDetails']['name']

failedInput = failedEvents[-1 \* failedId]['stateEnteredEventDetails']['input']

return (failedState, failedInput)

'''

We need to loop through the execution history, tracing back the executed steps

The first state we encounter will be the failed state

If we failed on a parallel state, we need the name of the parallel state rather than the

name of a state within a parallel state it failed on. This is because we can only attach

the goToState to the parallel state, but not a sub-state within the parallel state.

This loop starts with the id of the latest event and uses the previous event id's to trace

back the execution to the beginning (id 0). However, it will return as soon it finds the name

of the failed state

'''

currentEventId = failedEvents[0]['id']

while currentEventId != 0:

#multiply event id by -1 for indexing because we're looking at the reversed history

currentEvent = failedEvents[-1 \* currentEventId]

'''

We can determine if the failed state was a parallel state because it an event

with 'type'='ParallelStateFailed' will appear in the execution history before

the name of the failed state

'''

if currentEvent['type'] == 'ParallelStateFailed':

failedAtParallelState = True

'''

If the failed state is not a parallel state, then the name of failed state to return

will be the name of the state in the first 'TaskStateEntered' event type we run into

when tracing back the execution history

'''

if currentEvent['type'] == 'TaskStateEntered' and failedAtParallelState == False:

failedState = currentEvent['stateEnteredEventDetails']['name']

failedInput = currentEvent['stateEnteredEventDetails']['input']

return (failedState, failedInput)

'''

If the failed state was a paralell state, then we need to trace execution back to

the first event with 'type'='ParallelStateEntered', and return the name of the state

'''

if currentEvent['type'] == 'ParallelStateEntered' and failedAtParallelState:

failedState = failedState = currentEvent['stateEnteredEventDetails']['name']

failedInput = currentEvent['stateEnteredEventDetails']['input']

return (failedState, failedInput)

#Update the id for the next execution of the loop

currentEventId = currentEvent['previousEventId']

## Create the new state machine

We then use the name of the failed state to create the new state machine, with the “GoToState” branching execution directly to the failed state.

In order to do this, we need the [Amazon States Language (ASL)](http://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html) definition of the failed state machine so that we can modify this to append the “GoToState” to the definition, and create a new state machine from it.

Our script gets the ARN of the failed state machine from the execution ARN of the failed state machine. This ARN allows us to get the ASL definition of our failed state machine by calling the [DesribeStateMachine](http://docs.aws.amazon.com/step-functions/latest/apireference/API_DescribeStateMachine.html) API and creates a new state machine with the “GoToState”. When our script creates the new state machine, it also adds an additional input variable called “resuming”. When we execute this new state machine, we’ll specify this resuming variable as *true* in the input JSON. This tells our “GoToState” to branch execution to the state that had previously failed. Let’s take a look at the function that does this:

def attachGoToState(failedStateName, stateMachineArn):

'''

Given a state machine arn and the name of a state in that state machine, create a new state machine

that starts at a new choice state called the 'GoToState'. The "GoToState" will branch to the named

state, and send the input of the state machine to that state, when a variable called "resuming" is

set to True

Input failedStateName - string with the name of the failed state

stateMachineArn - string with the Arn of the state machine

Output response from the create\_state\_machine call, which is the API call that creates a new state machine

'''

try:

response = client.describe\_state\_machine(

stateMachineArn=stateMachineArn

)

except:

raise('Could not get ASL definition of state machine')

roleArn = response['roleArn']

stateMachine = json.loads(response['definition'])

#Create a name for the new state machine

newName = response['name'] + '-with-GoToState'

#Get the StartAt state for the original state machine, because we will point the 'GoToState' to this state

originalStartAt = stateMachine['StartAt']

'''

Create the GoToState with the variable $.resuming

If new state machine is executed with $.resuming = True, then the state machine will skip to the failed state

Otherwise, it will execute the state machine from the original start state

'''

goToState = {'Type':'Choice', 'Choices':[{'Variable':'$.resuming', 'BooleanEquals':False, 'Next':originalStartAt}], 'Default':failedStateName}

#Add GoToState to the set of states in the new state machine

stateMachine['States']['GoToState'] = goToState

#Add StartAt

stateMachine['StartAt'] = 'GoToState'

#Create new state machine

try:

response = client.create\_state\_machine(

name=newName,

definition=json.dumps(stateMachine),

roleArn=roleArn

)

except:

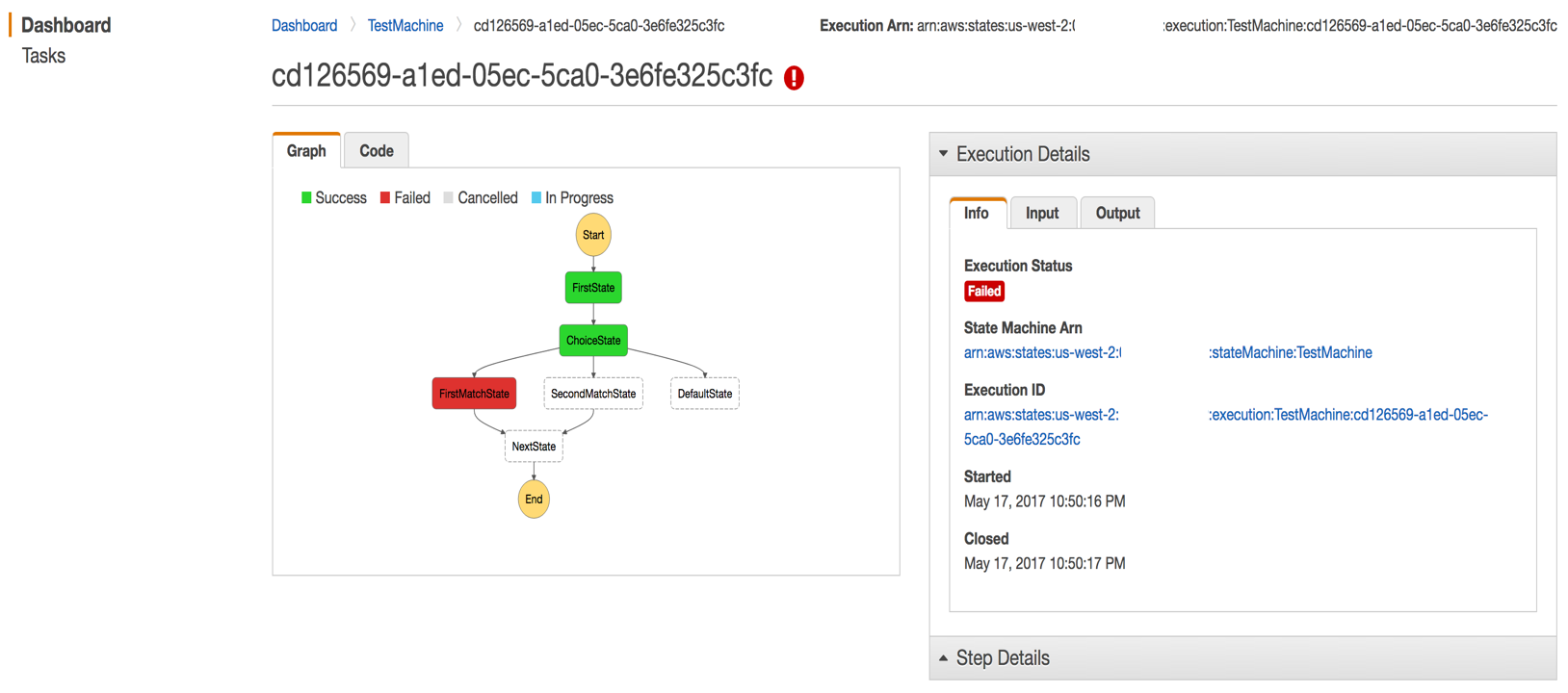
raise('Failed to create new state machine with GoToState')

return response

# Testing our script

Now that we understand how the script works, let’s test it out.

Below, is an example state machine that has failed, called “TestMachine”. This state machine successfully completed “FirstState” and “ChoiceState”, but when it branched to “FirstMatchState”, it failed.



Let’s use our script to create a new state machine that allows us to rerun this state machine, but skip the “FirstState” and the “ChoiceState” that had already succeeded. We can do this by calling our script:

*python gotostate.py --failedExecutionArn 'arn:aws:states:us-west-2:<AWS\_ACCOUNT\_ID>:execution:TestMachine-with-GoToState:b2578403-f41d-a2c7-e70c-7500045288595*

This creates a new state machine called “TestMachine-with-GoToState”, and returns its ARN, along with the input that had been sent to “FirstMatchState”. We can then inspect the input to determine what caused the error. In this case, you’ll notice that the input to “FirstMachState” was:

{

“foo”: 1,

“Message”: true

}

However, this state machine expects the “Message” field of the JSON to be a string rather than a Boolean. Let’s execute the new “TestMachine-with-GoToState”, and change the input to be a string, and add the “resuming” variable that the “GoToState” requires:

{

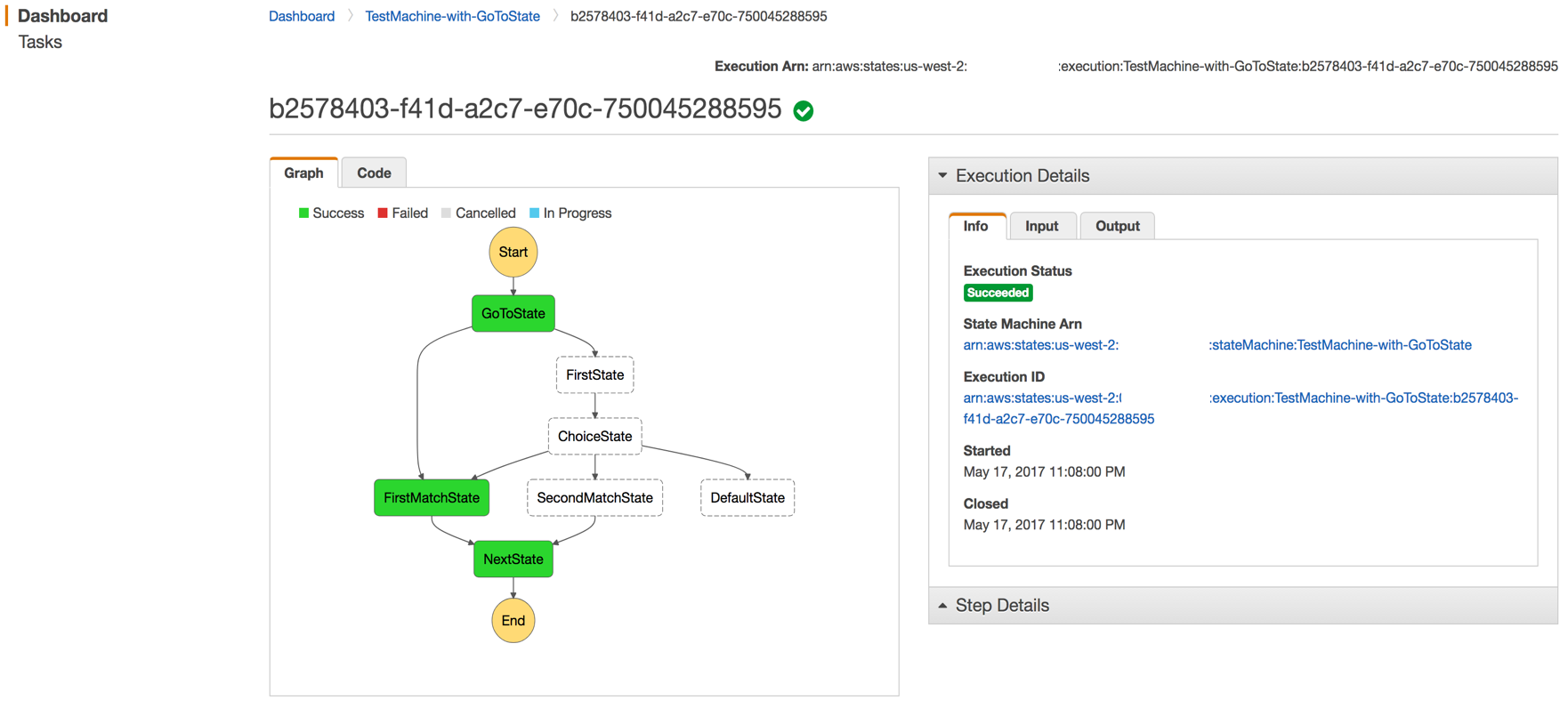
“foo”: 1,

“Message”: “Hello!”,

“resuming”:true

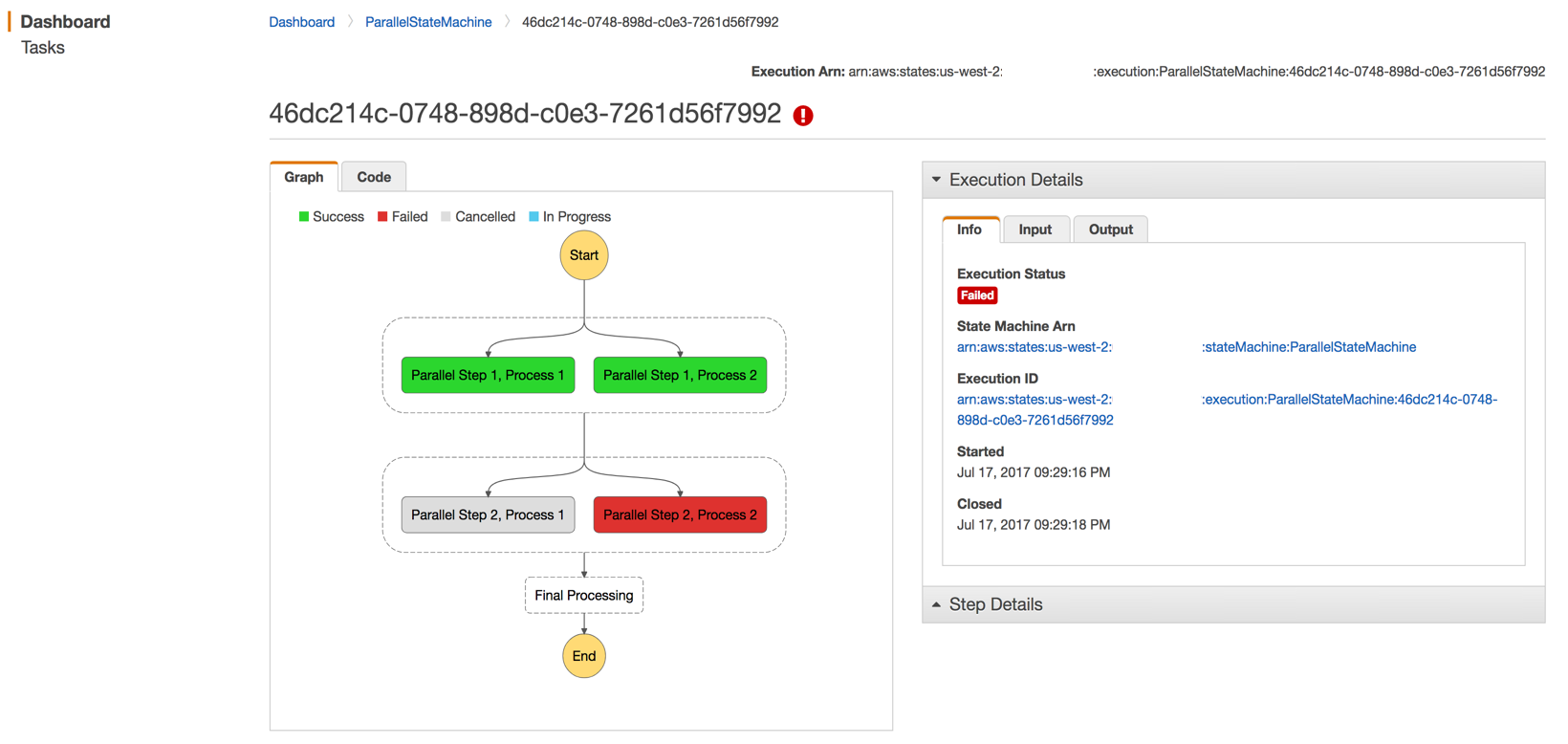
}

You’ll see that when we execute the new state machine, we skip “FirstState” and “ChoiceState”, and go directly to “FirstMatchState”, which was the state that had failed:



Let’s take a look at what happens when we have a state machine with multiple parallel steps. This example is incuded in the Github repository (INSERT LINK HERE) associated with this blog post with a CloudFormation template that sets up this state machine and provides instructions to replicate this solution.

Below is a state machine called “ParallelStateMachine” that takes an input through two subsequent parallel states before doing some final processing and exiting, along with the JSON with the ASL definition of the state machine.



{

"Comment": "An example of the Amazon States Language using a parallel state to execute two branches at the same time.",

"StartAt": "Parallel",

"States": {

"Parallel": {

"Type": "Parallel",

"ResultPath":"$.output",

"Next": "Parallel 2",

"Branches": [

{

"StartAt": "Parallel Step 1, Process 1",

"States": {

"Parallel Step 1, Process 1": {

"Type": "Task",

"Resource": "arn:aws:lambda:us-west-2:XXXXXXXXXXXX:function:LambdaA",

"End": true

}

}

},

{

"StartAt": "Parallel Step 1, Process 2",

"States": {

"Parallel Step 1, Process 2": {

"Type": "Task",

"Resource": "arn:aws:lambda:us-west-2:XXXXXXXXXXXX:function:LambdaA",

"End": true

}

}

}

]

},

"Parallel 2": {

"Type": "Parallel",

"Next": "Final Processing",

"Branches": [

{

"StartAt": "Parallel Step 2, Process 1",

"States": {

"Parallel Step 2, Process 1": {

"Type": "Task",

"Resource": "arn:aws:lambda:us-west-2:XXXXXXXXXXXXX:function:LambdaB",

"End": true

}

}

},

{

"StartAt": "Parallel Step 2, Process 2",

"States": {

"Parallel Step 2, Process 2": {

"Type": "Task",

"Resource": "arn:aws:lambda:us-west-2:XXXXXXXXXXXX:function:LambdaB",

"End": true

}

}

}

]

},

"Final Processing": {

"Type": "Task",

"Resource": "arn:aws:lambda:us-west-2:XXXXXXXXXXXX:function:LambdaC",

"End": true

}

}

}

First, we’ll use an input that initially fails:

{

"Message": "Hello!"

}

This fails because the state machine expects us to have a variable in our input Json called “foo” in the second parallel state to run “Parallel Step 2, Process 1” and “Parallel Step 2, Process 2”. Instead, our original input gets processed by our first parallel state and produces the following output to pass to the second parallel state:

{

"output": [

{

"Message": "Hello!"

},

{

"Message": "Hello!"

}

],

}

Let’s run our script on the failed state machine to create a new state machine that allows us to resume directly at the second parallel state instead of having to redo the first parallel state. This creates a new state machine called “ParallelStateMachine-with-GoToState”. Below is the JSON that was created by our script to define the new state machine in ASL. Notice it contains the “GoToState” that was attached by the script.

{

"Comment":"An example of the Amazon States Language using a parallel state to execute two branches at the same time.",

"States":{

"Final Processing":{

"Resource":"arn:aws:lambda:us-west-2:XXXXXXXXXXXX:function:LambdaC",

"End":true,

"Type":"Task"

},

"GoToState":{

"Default":"Parallel 2",

"Type":"Choice",

"Choices":[

{

"Variable":"$.resuming",

"BooleanEquals":false,

"Next":"Parallel"

}

]

},

"Parallel":{

"Branches":[

{

"States":{

"Parallel Step 1, Process 1":{

"Resource":"arn:aws:lambda:us-west-2:XXXXXXXXXXXX:function:LambdaA",

"End":true,

"Type":"Task"

}

},

"StartAt":"Parallel Step 1, Process 1"

},

{

"States":{

"Parallel Step 1, Process 2":{

"Resource":"arn:aws:lambda:us-west-2:XXXXXXXXXXXX:LambdaA",

"End":true,

"Type":"Task"

}

},

"StartAt":"Parallel Step 1, Process 2"

}

],

"ResultPath":"$.output",

"Type":"Parallel",

"Next":"Parallel 2"

},

"Parallel 2":{

"Branches":[

{

"States":{

"Parallel Step 2, Process 1":{

"Resource":"arn:aws:lambda:us-west-2:XXXXXXXXXXXX:function:LambdaB",

"End":true,

"Type":"Task"

}

},

"StartAt":"Parallel Step 2, Process 1"

},

{

"States":{

"Parallel Step 2, Process 2":{

"Resource":"arn:aws:lambda:us-west-2:XXXXXXXXXXXX:function:LambdaB",

"End":true,

"Type":"Task"

}

},

"StartAt":"Parallel Step 2, Process 2"

}

],

"Type":"Parallel",

"Next":"Final Processing"

}

},

"StartAt":"GoToState"

}

We can then execute this state machine with the correct input by adding the “foo” and “resuming” variables:

{

"foo": 1,

"output": [

{

"Message": "Hello!"

},

{

"Message": "Hello!"

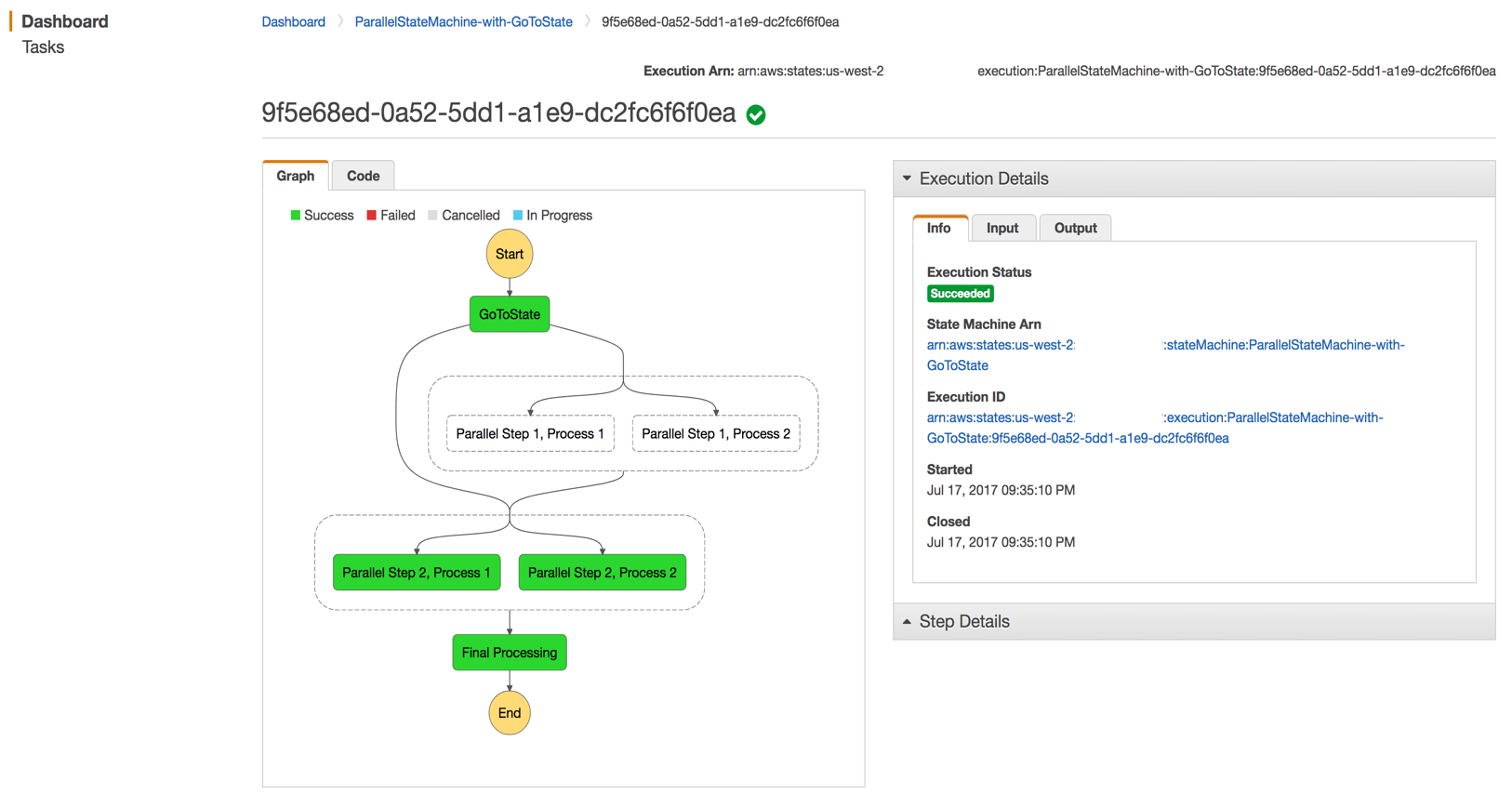
}

],

"resuming": true

}

This yields the result shown below.

****

# Conclusion

When building you’re building out complex workflows, it’s important to be prepared for failure. You can do this by taking advantage of features such as automatic error retries in Step Functions and custom error handling from Lambda exceptions. Nevertheless, state machines can still have the possibility of failing. With the methodology and script presented in the blog post, we are able to resume a failed state machine from it’s point of failure. This allows us to skip the execution of steps in our workflow that had already succeeded, and recover the process from the point of failure.