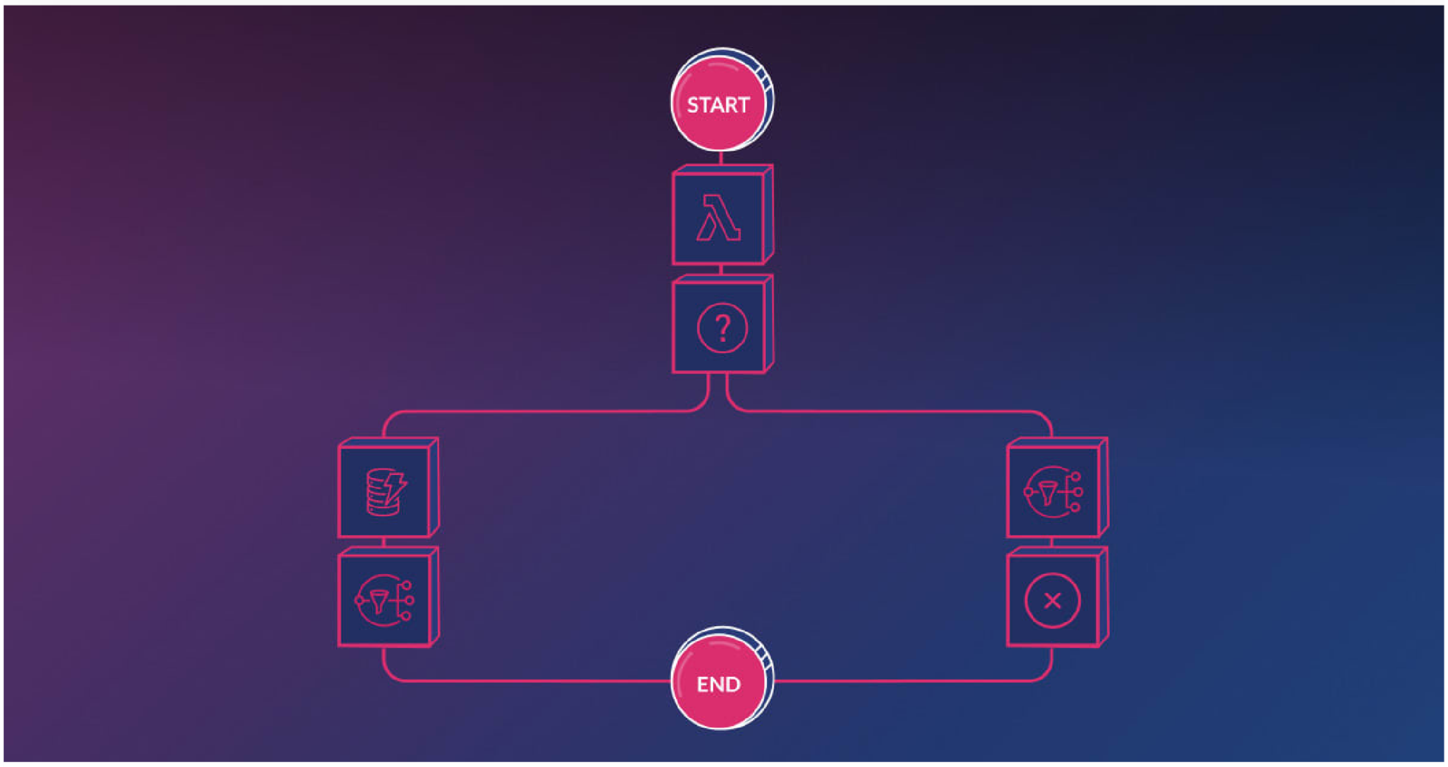
Testing strategies for Step Functions

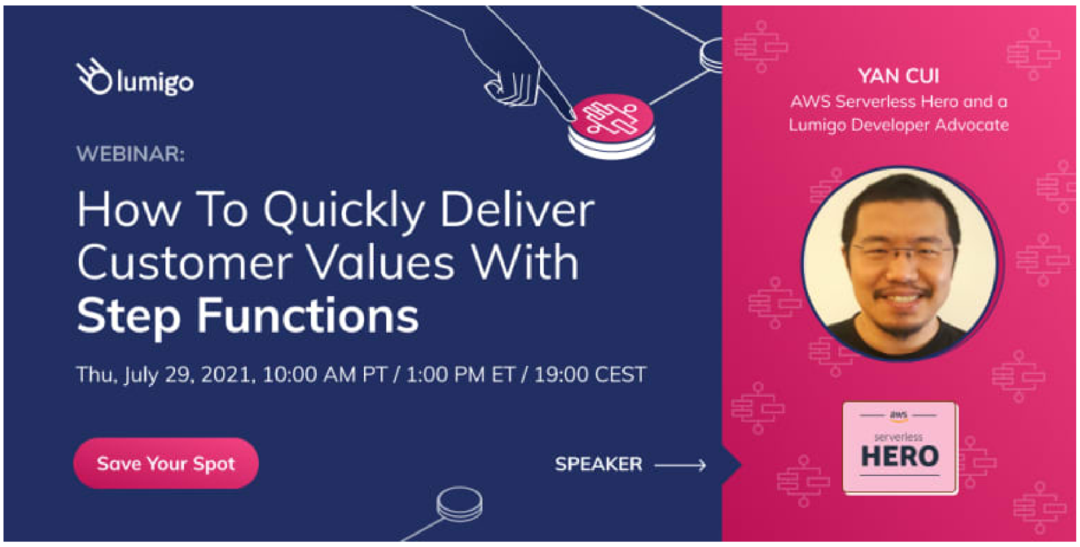


Yan Cui , Jul 14 2021

AWS Step Functions is a powerful orchestration service that lets you model even the most complex business workflows. It packs a great visualization tool (which you can also use to design your workflows visually now!) and can integrate with many AWS services directly, including Lambda, DynamoDB, and API Gateway.

It's one of my favorite AWS services and I often use it to model complex or business-critical workflows. However, a common challenge newcomers struggle with Step Functions is how do you go about testing them?

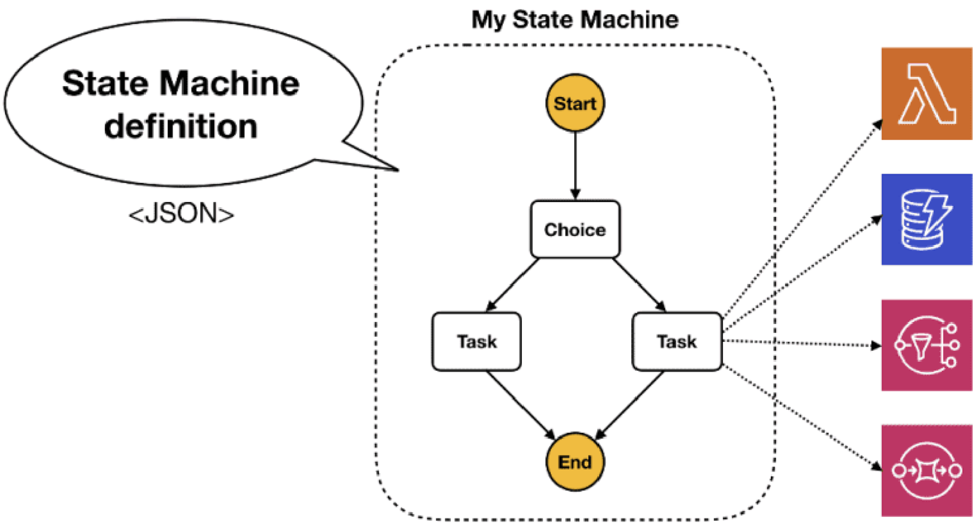
In this post, let's talk about the challenges with testing Step Functions, the failure modes that you should test for, and my strategy for testing Step Functions.


lumigo
WEBINAR:
How To Quickly Deliver Customer Values With Step Functions
Thu, July 29, 2021, 10:00 AM PT / 1:00 PM ET / 19:00 CEST
Save Your Spot
SPEAKER
YAN CUI
AWS Serverless Hero and a Lumigo Developer Advocate
aws serverless HERO

In Step Functions, you model workflows as state machines, i.e. a series of steps of tasks that need to be performed. You can include branching logic, retry failed tasks, perform tasks in parallel, and even call out to another state machine and wait for its response.

You define the state machine using a JSON-based language, but you can also design the state machine visually using the new visual designer (announced on Jun 17th 2021).

The Task states are the meat of a state machine as they are the places where the state machine calls out to another resource to perform some computation or data operation. Task states can integrate directly with a number of AWS services such as Lambda functions, DynamoDB tables, SNS topics, and SQS queues.



From the testing perspective, we need to understand where things can go wrong and hope to cover as many of the possible failure modes as possible. Here are the failure modes that you should test for.

Incorrect state machine definitions

There can be mistakes in the state machine definition itself. For example, the "next" state is misconfigured and pointing to the wrong state, or perhaps you forgot to override the default 30s timeout for a Task state that often runs for a few minutes, or maybe you used the wrong input or output path for a Task state.

Whatever the case, your state machine definition might be syntactically correct but it would not perform as you'd expect.

Incorrect IAM permissions

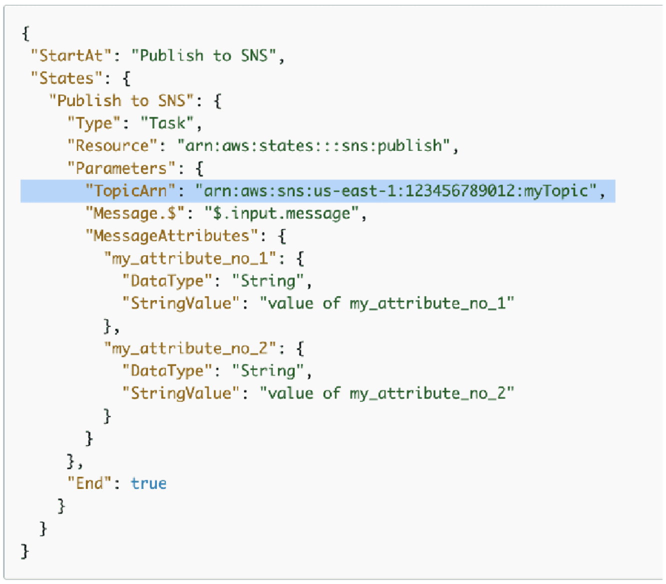
The state machine requires IAM permissions to interact with other AWS services. If you have a Task state that executes a Lambda function then the state machine must have the lambda:InvokeFunction permission for this function.

Similarly, if you have Task states that reads data from a DynamoDB then the state machine's IAM role must have the necessary DynamoDB permissions too.

Incorrect integration configurations

Step Functions supports a number of service integrations out-of-the-box, including Lambda, SNS, SQS and DynamoDB. When integrating with these services, sometimes you have to provide a Parameters object and include things like the ARN for a SNS topic or the URL for a SQS queue.

Maybe the topic ARN is not configured correctly, or maybe you're sending the wrong message, or perhaps you forgot to set the message attribute the subscriber needs to filter the incoming messages. Ultimately, the message is not delivered to the right place, and the app fails.

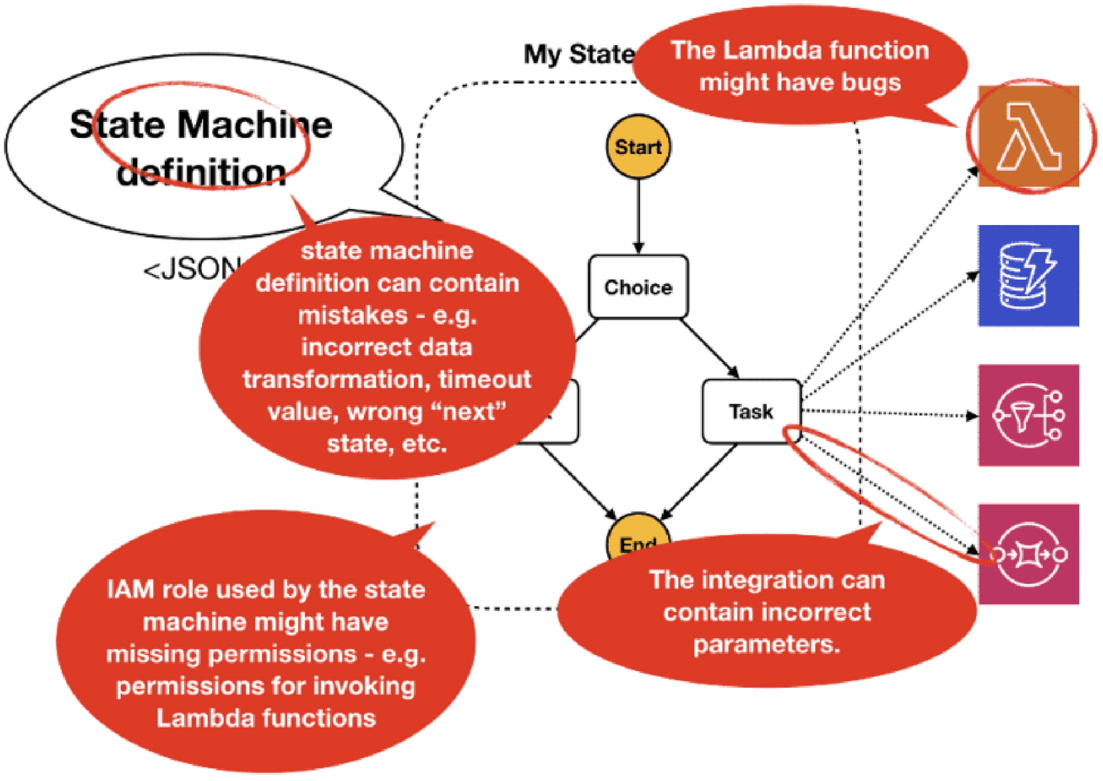


```
{
 "StartAt": "Publish to SNS",
 "States": {
   "Publish to SNS": {
     "Type": "Task",
     "Resource": "arn:aws:states:::sns:publish",
     "Parameters": {
       "TopicArn": "arn:aws:sns:us-east-1:123456789012:myTopic",
       "Message.$": "$.input.message",
       "MessageAttributes": {
         "my_attribute_no_1": {
           "DataType": "String",
           "StringValue": "value of my_attribute_no_1"
         },
         "my_attribute_no_2": {
           "DataType": "String",
           "StringValue": "value of my_attribute_no_2"
         }
       }
     },
     "End": true
   }
  }
}
```

Bugs in the Lambda functions

Many Task states would invoke a Lambda function to exercise some custom business logic. And you might have bugs in the Lambda function itself. This includes a whole class of potential failure modes including bugs in your business logic, misconfigured IAM permissions, or any number of program errors that can fail the Lambda invocation (null reference exception, missing dependencies, etc.)

In short, there are a lot of ways your state machines can fail and you should aim to catch as many of them with automated tests as possible.


State Machine definition
<JSON
My State
The Lambda function might have bugs
Start
Choice
Task
End
state machine definition can contain mistakes - e.g. incorrect data transformation, timeout value, wrong "next" state, etc.
IAM role used by the state machine might have missing permissions - e.g. permissions for invoking Lambda functions
The integration can contain incorrect parameters.

Many state machines in the wild can be pretty complex, take this one from CocaCola as an example!

Step Functions are often used to model complex workflows like this, and complex workflows naturally have complex failure modes. To have confidence that these complex state machines actually work, you need to have a good coverage of their execution paths and cover both happy and failure paths.

But to exercise every branch in your state machine is a huge challenge because you can't mock out the Task states and use stubbed outputs to drive the execution towards the path you want. Instead, you have to execute the state machine with a carefully chosen input so the execution would follow the path that you want to test. This requires a lot of thought and planning and is incredibly brittle as the slightest change in the state machine can render your test case invalid.

This is perhaps the biggest challenge with testing step functions and unfortunately it's also a challenge that we can't fully tackle ourselves without support from the service.

Although AWS published a local version of Step Functions that you can use to simulate the service, it doesn't have any mocking capabilities.

Also, the local version of Step Functions doesn't use the intended IAM role so you can't catch any IAM-related problems with it either. Why can't it assume the right role? Because the intended IAM role can only be assumed by the Step Functions service, not some Java application that simulates it.

So with these challenges in mind, how do you test Step Functions?

To cover all the possible failure modes for a state machine, I typically write two kinds of tests:

1. Tests that focus on the Lambda functions and runs locally.
2. End-to-end (e2e) tests that execute the state machine for real.

Testing the Lambda functions

The first kind of tests would exercise (locally) the individual Lambda functions in isolation and depending on what the function's job is, I would:

- Check the function's output conforms to expectation. A Lambda function's output is likely used as input to the subsequent state(s) in the state machine. When designing the state machine, I have an implicit contract in mind for each function – what input should it handle and what output should it return.
- Validate the function has created the expected side-effects. For example, a row is written to a DynamoDB table, or an event is published to EventBridge.

What would I call these tests? If they're executed locally, then are they unit tests or integration tests? Martin Fowler recently wrote about the problems with the whole unit vs integration tests debate, because everyone has a different definition for them!

Even if the functions are executed locally, I'd have the function code talk to the real AWS services as much as possible. As a general rule, I don't use mocks or stubs for AWS services, at least not for the happy path. In Martin Fowler's words, these are "sociable tests" that rely on other services to fulfill its behavior. If the Lambda function's business logic needs to perform a DynamoDB query, it would do so against the real DynamoDB table. Any mistakes in the configuration of the table or errors in the DynamoDB query request would be caught by these tests. As such, they give me a strong confidence that my code actually works, provided that they're given the right input.

As for mocks, they are still useful in some cases:

- To test error handling for failure modes that are hard to reproduce reliably with AWS services. For example, to simulate DynamoDB's ThroughputExceededExceptions.
- To prevent the test from executing customer-impacting actions, etc. For example, if you need to test integration with a payment service, but the service doesn't offer a sandbox environment.
- To test integration with internal services (i.e. services that are internal to your organization). Unlike AWS services or other SAAS applications, internal services can be unstable especially in the lower environments. Use mocks for these internal services to shield yourself from these instabilities.

You might be wondering "Wait if you use the real DynamoDB tables, aren't you going to fill those tables with test data?"

That's correct, and it can be an undesirable side-effect of this approach. But you can easily mitigate this by using temporary CloudFormation stacks when working on a feature or when you need to run these tests. For more information about this, have a look at this post.

Testing the state machine

For the second kind of tests, I would execute the deployed state machine with different inputs and see:

- Did it return the right output? (if the state machine output is important)
- Did it create the right side-effects? (e.g. is data written to DynamoDB tables?)
- Did it finish in a timely fashion?

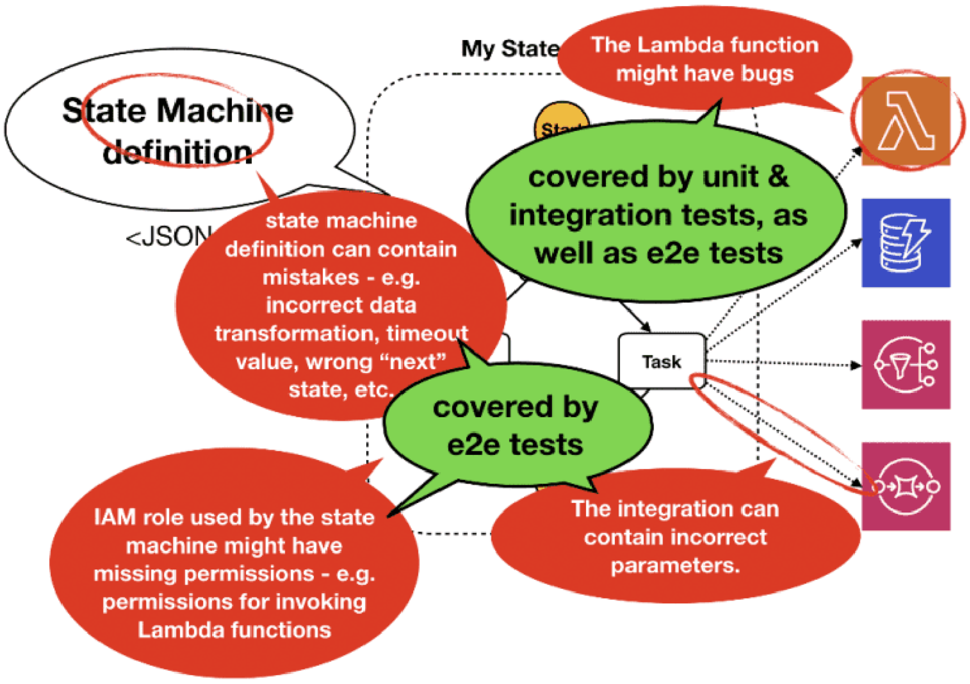
These tests are all encompassing and ensure that everything works together – the IAM permissions, the Lambda functions, the state machine definition, the whole shebang!

Although they're powerful, as mentioned before, it can be challenging to cover every execution path for complex state machines. Because of this, I leave the more exhaustive testing (that is, tests that cover a wide range of different inputs) to the sociable tests that target individual Lambda functions and focus the end-to-end tests on executing the happy paths through the state machine.

It's also worth noting that these end-to-end tests tend to take longer to complete so be sure to allow for extra time for them to complete. This is especially true right after a deployment, because all the Lambda functions executed by the state machine would incur cold starts during your end-to-end tests!

With these two types of tests, I'm able to cover all the aforementioned types of failure modes.

- Bugs in the Lambda function can be caught by the sociable tests that target individual functions. These tests can catch problems with your business logic as well as how your function integrates with other AWS or 3rd party services. And best of all, you can run these tests before deploying your changes and give you a faster feedback loop!
- All other problems (including IAM permission issues) will be caught by the end-to-end tests. These tests are slower and more difficult to orchestrate, so it's best to focus on the happy paths to ensure a high return-on-investment from the tests.


State Machine definition
<JSON
My State
The Lambda function might have bugs
covered by unit & integration tests, as well as e2e tests
state machine definition can contain mistakes - e.g. incorrect data transformation, timeout value, wrong "next" state, etc.
Task
covered by e2e tests
IAM role used by the state machine might have missing permissions - e.g. permissions for invoking Lambda functions
The integration can contain incorrect parameters.

However, testing doesn't stop here! Even applications with 100% test coverage can experience problems and outages in the wild. After all, test coverage only tells you how much of the code you have written is executed by tests. They don't tell you what scenarios you have missed in your code – for example, missing error handlers or branch conditions.

There is a school of thought around testing in production, which incorporates many practices such as canarying, feature flagging and chaos engineering. Regardless of whether you test in production, these practices are useful in their own rights. You also need to have good observability in the application to be able safely test in production, and this is where tools like Lumigo can be so valuable.

Within a few minutes I can have set up Lumigo for my application and have access to a wealth of information about my application.

- Distributed trace for every transaction, which supports both synchronous and asynchronous Lambda invocations. API Gateway, SNS, SQS, EventBridge, Kinesis and DynamoDB streams are all supported.
- The invocation event for every Lambda invocation.
- The request, response and duration for every HTTP(s) request my Lambda function makes to other services, including to other AWS services such as DynamoDB.

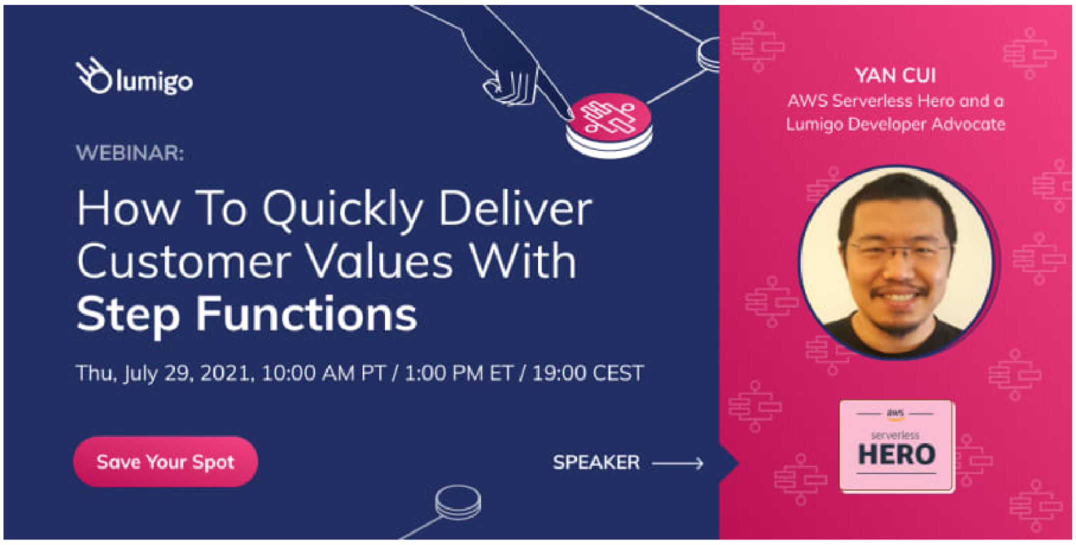
- The request, response and duration for some TCP requests, such as those to Redis and MySQL databases.
- Error message and stack trace for erroneous Lambda invocations.

With all these information ready at hand, I'm able to easily infer the internal state of my application and quickly troubleshoot problems, even ones that I hadn't encountered before.

And in the context of Step Functions, Lumigo also supports Step Functions!

To start using Lumigo for free and develop serverless applications like a pro, go to https://platform.lumigo.io/signup to sign up.

Join Yan on Thursday, July 29, to learn all about Step Functions, model business workflows as state machines, cover real-world case studies and design patterns. Click the link to save your spot! https://info.lumigo.io/webinar-stepfunctions-2021-07-29


lumigo
WEBINAR:
How To Quickly Deliver Customer Values With Step Functions
Thu, July 29, 2021, 10:00 AM PT / 1:00 PM ET / 19:00 CEST
Save Your Spot
SPEAKER
YAN CUI
AWS Serverless Hero and a Lumigo Developer Advocate
aws serverless HERO

Yan Cui, Aug 27 2019

Serverless Framework Plugin: Get Instant Visibility

Observability is a common challenge for serverless applications and we believe Lumigo is perfectly positioned to help you tackle this...

Read More >

Saar Tochner, May 17 2022

The Hidden Magic of Extensions

AWS Lambda execution lifecycle has 3 main phases: initialization, invocation, and shutdown. In the

Danny Lev, Nov 13 2019

Migrating from IOpipe to Lumigo

You've no doubt heard that IOpipe has been acquired by New Relic (congratulations to both). As part of the acquisition,...

initialization phase, a Lambda creates the...

Read More >

Read More >

lumigo

Product

Features

Docs

Pricing

Company

About

Careers

Terms of Service

Privacy Policy

Privacy Shield Notice

Support

Contact Us

Join Us