AWS Compute Blog

Building well-architected serverless applications: Understanding application health – part 2

by Julian Wood | on 16 APR 2020 | in Amazon CloudWatch, AWS CloudFormation, AWS Lambda, AWS Serverless Application Model, AWS Well-Architected Tool, Best Practices, Serverless, Technical How-to | Permalink | Share

This series of blog posts uses the AWS Well-Architected Tool with the Serverless Lens to help customers build and operate applications using best practices. In each post, I address the nine serverless-specific questions identified by the Serverless Lens along with the recommended best practices. See the Introduction post for a table of contents and explaining the example application.

Question OPS1: How do you evaluate your serverless application's health?

This post continues part 1 of this Operational Excellence question. Previously, I covered using Amazon CloudWatch out-of-the-box standard metrics and alerts, and structured and centralized logging with the Embedded Metrics Format.

Best practice: Use application, business, and operations metrics

Identifying key performance indicators (KPIs), including business, customer, and operations outcomes, in additional to application metrics, helps to show a higher-level view of how the application and business is performing.

Business KPIs measure your application performance against business goals. For example, if fewer flight reservations are flowing through the system, the business would like to know.

Customer experience KPIs highlight the overall effectiveness of how customers use the application over time. Examples are perceived latency, time to find a booking, make a payment, etc.

Operational metrics help to see how operationally stable the application is over time. Examples are continuous integration/delivery/deployment feedback time, mean-time-between-failure/recovery, number of on-call pages and time to resolution, etc.

Custom Metrics

Embedded Metrics Format can also emit custom metrics to help understand your workload health's impact on business.

The airline booking service uses AWS Step Functions, AWS Lambda, Amazon SNS, and Amazon DynamoDB.

In the confirm booking module function handler, I add a new namespace and dimension to associate this set of logs with this application and service.

Python
```python
metrics.set_namespace("ServerlessAirlineEMF")
```

Python
```python
metrics.put_dimensions({"service":"confirm_booking"})
```

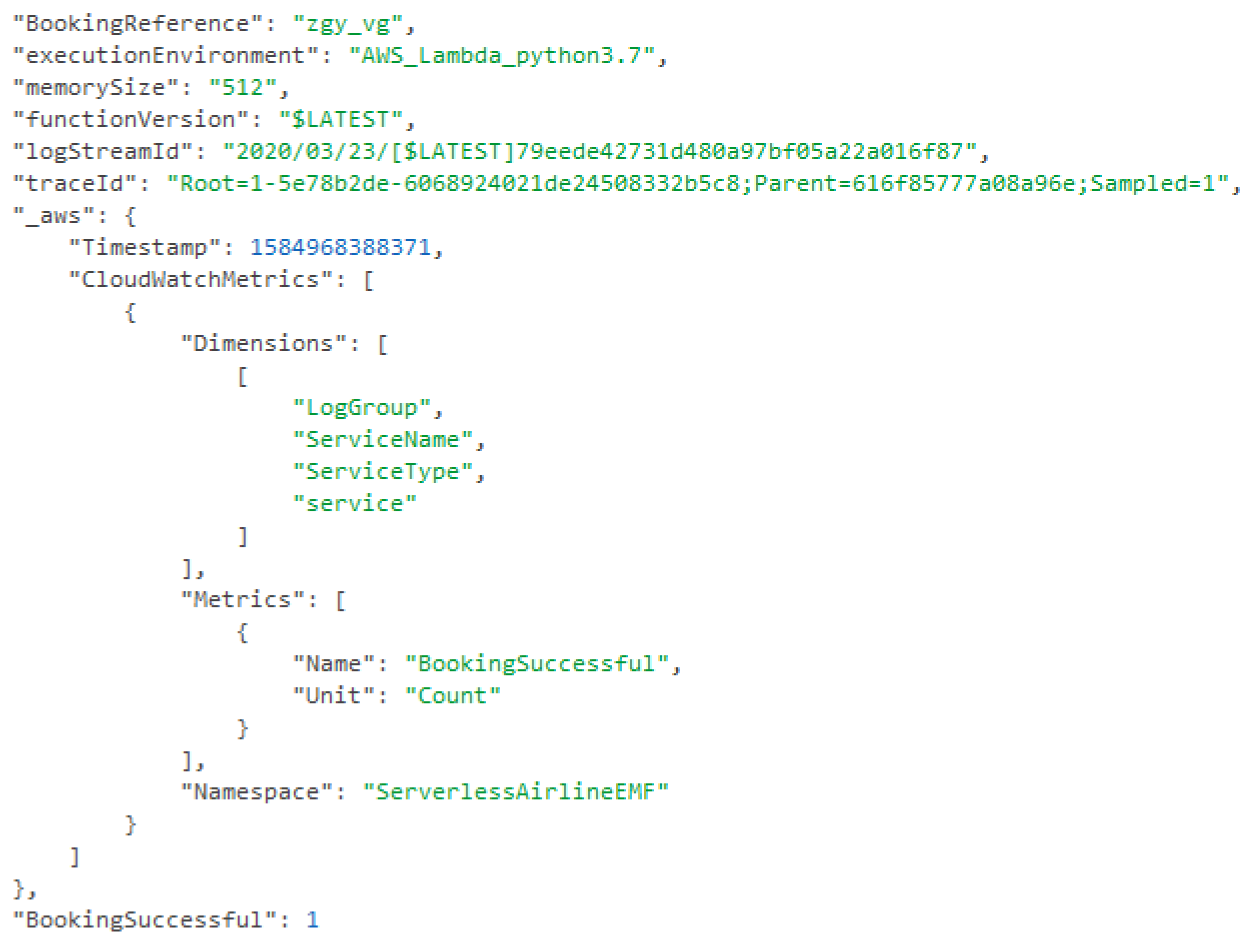
Within the *try* statement within the try/except block, I emit a metric for a successful booking:

Python
```python
metrics.put_metric("BookingSuccessful", 1, "Count")
```

And within the *except* statement within the try/except block, I emit a metric for a failed booking:

Python
```python
metrics.put_metric("BookingFailed", 1, "Count")
```

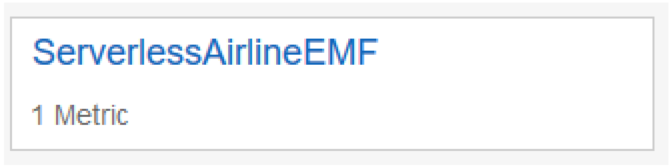
Once I make a booking, within the CloudWatch console, I navigate to **Logs | Log groups** and select the *Airline-ConfirmBooking-develop* group. I select a log stream and find the custom metric as part of the structured log entry.



```
"BookingReference": "zgy_vg",
"executionEnvironment": "AWS_Lambda_python3.7",
"memorySize": "512",
"functionVersion": "$LATEST",
"logStreamId": "2020/03/23/[$LATEST]79eede42731d480a97bf05a22a016f87",
"traceId": "Root=1-5e78b2de-6068924021de24508332b5c8;Parent=616f85777a08a96e;Sampled=1",
"_aws": {
    "Timestamp": 1584968388371,
    "CloudWatchMetrics": [
        {
            "Dimensions": [
                [
                    "LogGroup",
                    "ServiceName",
                    "ServiceType",
                    "service"
                ]
            ],
            "Metrics": [
                {
                    "Name": "BookingSuccessful",
                    "Unit": "Count"
                }
            ],
            "Namespace": "ServerlessAirlineEMF"
        }
    ]
},
"BookingSuccessful": 1
```

Custom metric structured log entry

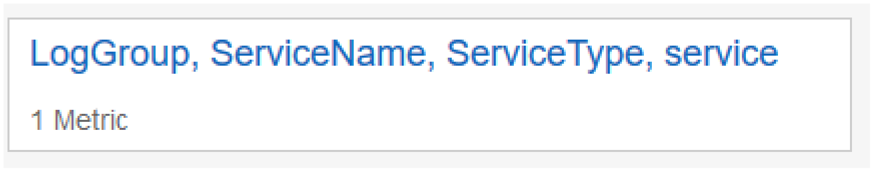
I can also create a custom metric graph. Within the CloudWatch console, I navigate to **Metrics**. I see the ServerlessAirlineEMF Custom Namespace is available.


ServerlessAirlineEMF

1 Metric

Custom metric namespace

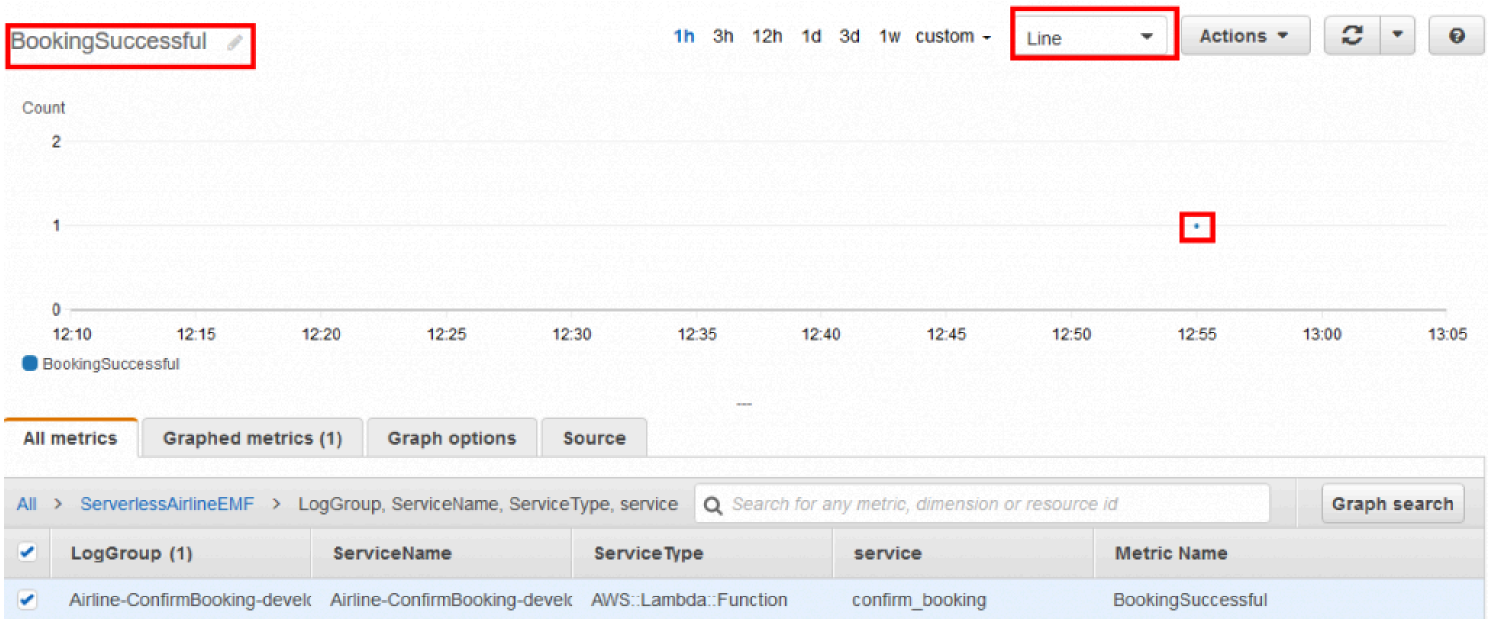
I select the Namespace and the available metric.


LogGroup, ServiceName, ServiceType, service

1 Metric

Available metric

I select a *Line* graph, add a name, and see the successful booking now plotted on the graph.



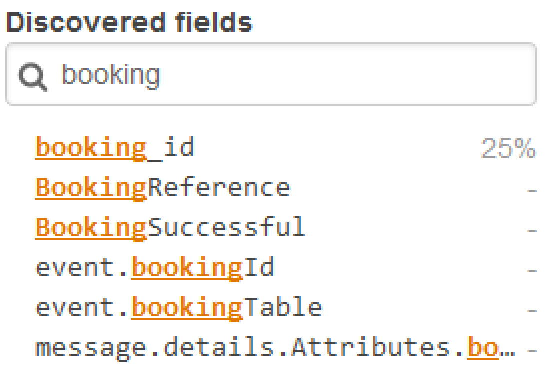
✓	LogGroup (1)	ServiceName	ServiceType	service	Metric Name
✓	Airline-ConfirmBooking-develc	Airline-ConfirmBooking-develc	AWS::Lambda::Function	confirm_booking	BookingSuccessful

Plotted CloudWatch metric

I can also visualize and analyze this metric using [CloudWatch Insights](#).

Once a booking is made, within the [CloudWatch console](#), I navigate to **Logs | Insights.** I select the */aws/lambda/Airline-ConfirmBooking-develop* log group. I choose **Run query** which shows a list of discovered fields on the right of the console.

I can search for discovered *booking* related fields.

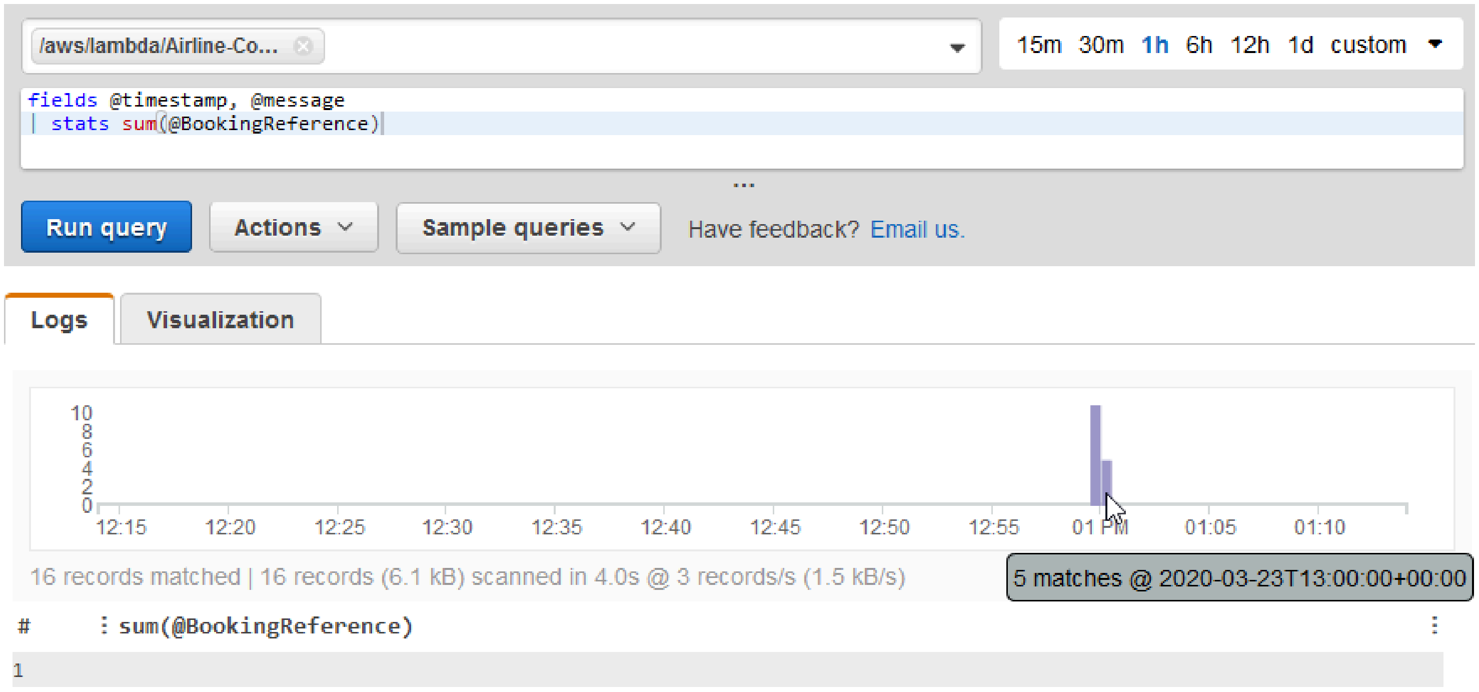

Discovered fields

booking

booking_id	25%
BookingReference	-
BookingSuccessful	-
event.bookingId	-
event.bookingTable	-
message.details.Attributes.bo…	-

I then enter the following in the query pane to search the logs and plot the sum of *BookingReference* and choose **Run Query**:

```
fields @timestamp, @message
| stats sum(@BookingReference)
```



```
fields @timestamp, @message
| stats sum(@BookingReference)
```

16 records matched | 16 records (6.1 kB) scanned in 4.0s @ 3 records/s (1.5 kB/s)

5 matches @ 2020-03-23T13:00:00+00:00

#	sum(@BookingReference)
1	

CloudWatch Insights query

There are a number of other component metrics that are important to measure. Track interactions between upstream and downstream components such as *message queue length*, *integration latency*, and *throttling*.

Improvement plan summary

1. Identify user journeys and metrics from each customer transaction.
2. Create custom metrics asynchronously instead of synchronously for improved performance, cost, and reliability outcomes.
3. Emit business metrics from within your workload to measure application performance against business goals.
4. Create and analyze component metrics to measure interactions with upstream and downstream components.

5. Create and analyze operational metrics to assess the health of your continuous delivery pipeline and operational processes.

Good practice: Use distributed tracing and code is instrumented with additional context

Logging provides information on the individual point in time events the application generates. Tracing provides a wider continuous view of an application. Tracing helps to follow a user journey or transaction through the application.

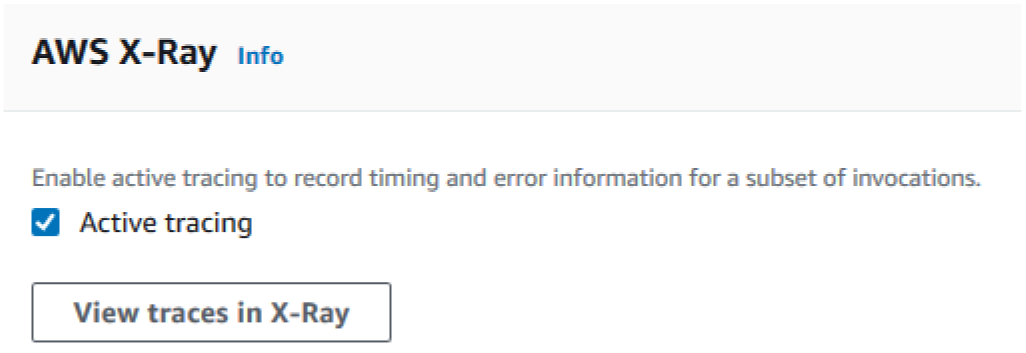
AWS X-Ray is an example of a distributed tracing solution, there are a number of third-party options as well. X-Ray collects data about the requests that your application serves and also builds a service graph, which shows all the components of an application. This provides visibility to understand how upstream/downstream services may affect workload health.

For the most comprehensive view, enable X-Ray across as many services as possible and include X-Ray tracing instrumentation in code. This is the list of AWS Services integrated with X-Ray.

X-Ray receives data from services as segments. Segments for a common request are then grouped into traces. Segments can be further broken down into more granular subsegments. Custom data key-value pairs are added to segments and subsegments with annotations and metadata. Traces can also be grouped which helps with filter expressions.

AWS Lambda instruments incoming requests for all supported languages. Lambda application code can be further instrumented to emit information about its status, correlation identifiers, and business outcomes to determine transaction flows across your workload.

X-Ray tracing for a Lambda function is enabled in the Lambda Console. Select a Lambda function. I select the *Airline-ReserveBooking-develop* function. In the **Configuration** pane, I select to enable **X-Ray Active tracing**.


AWS X-Ray Info

Enable active tracing to record timing and error information for a subset of invocations.

☑ Active tracing

X-Ray tracing enabled

X-Ray can also be enabled via CloudFormation with the following code:

YAML
```yaml
TracingConfig:
  Mode: Active
```

Lambda IAM permissions to write to X-Ray are added automatically when active tracing is enabled via the console. When using CloudFormation, Allow the Actions `xray:PutTraceSegments` and `xray:PutTelemetryRecords`

It is important to understand what invocations X-Ray does trace. X-Ray applies a sampling algorithm. If an upstream service, such as API Gateway with X-Ray tracing enabled, has already sampled a request, the Lambda function request is also sampled. Without an upstream request, X-Ray traces data for the first Lambda invocation each second, and then 5% of additional invocations.

For the airline application, X-Ray tracing is initiated within the shared library with the code:

Python
```python
from aws_xray_sdk.core import models, patch_all, xray_recorder
```

Segments, subsegments, annotations, and metadata are added to functions with the following example code:

Python
```python
segment = xray_recorder.begin_segment('segment_name')
# Start a subsegment
subsegment = xray_recorder.begin_subsegment('subsegment_name')
# Add metadata and annotations
segment.put_metadata('key', dict, 'namespace')
subsegment.put_annotation('key', 'value')
# Close the subsegment and segment
xray_recorder.end_subsegment()
xray_recorder.end_segment()
```

For example, within the collect payment module, an annotation is added for a successful payment with:

Python
```python
tracer.put_annotation("PaymentStatus", "SUCCESS")
```

CloudWatch ServiceLens

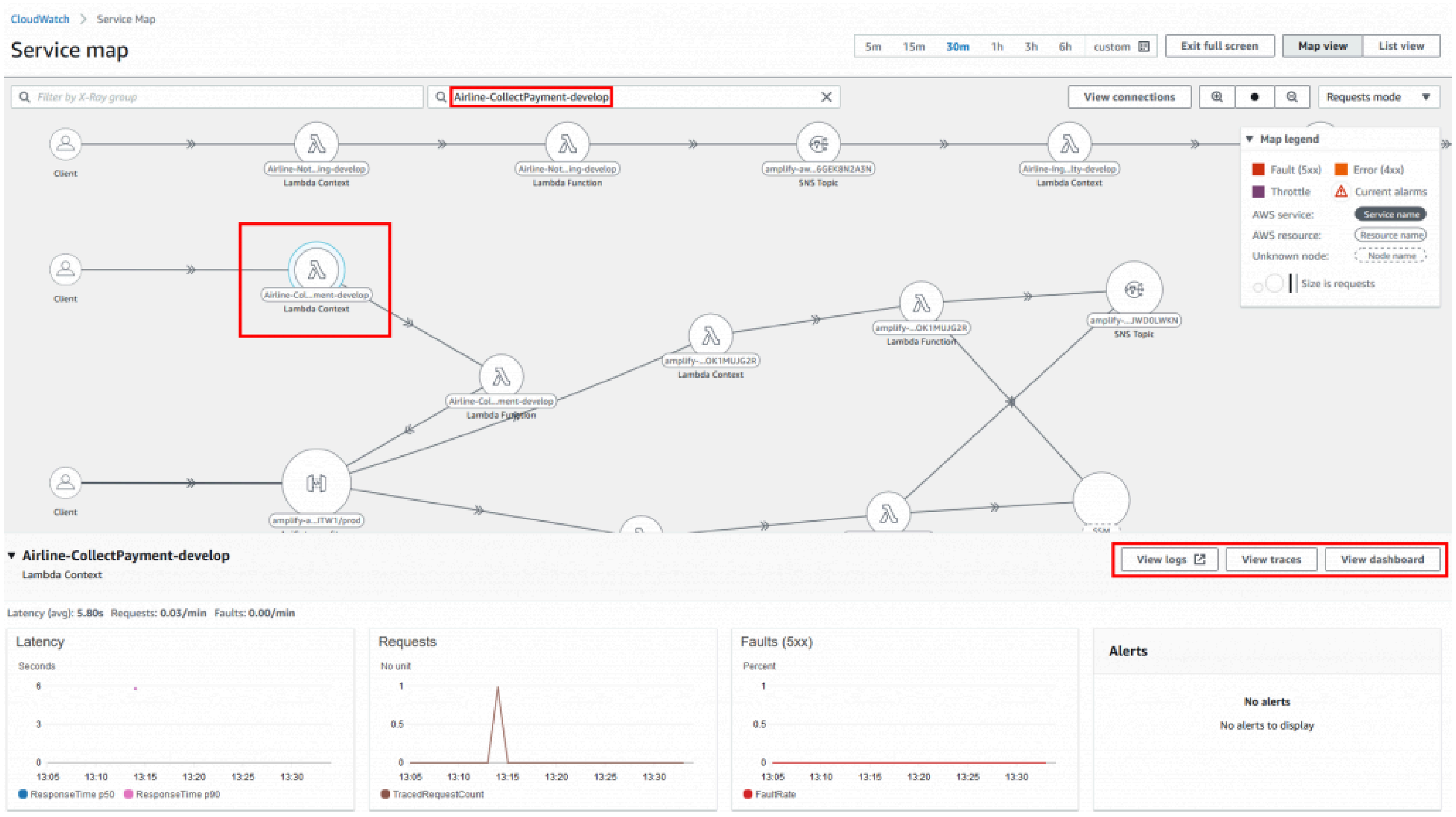
Once a booking is made and payment is successful, the tracing is available in the X-Ray console.

I explore how Amazon CloudWatch ServiceLens connects metrics, logs and the X-Ray tracing. Within the CloudWatch console, I navigate to **ServiceLens | Service Map**.

I can visualize all application resources and dependencies where X-Ray is enabled. I can trace performance or availability issues. If there was an issue connecting to SNS for example, this would be shown.

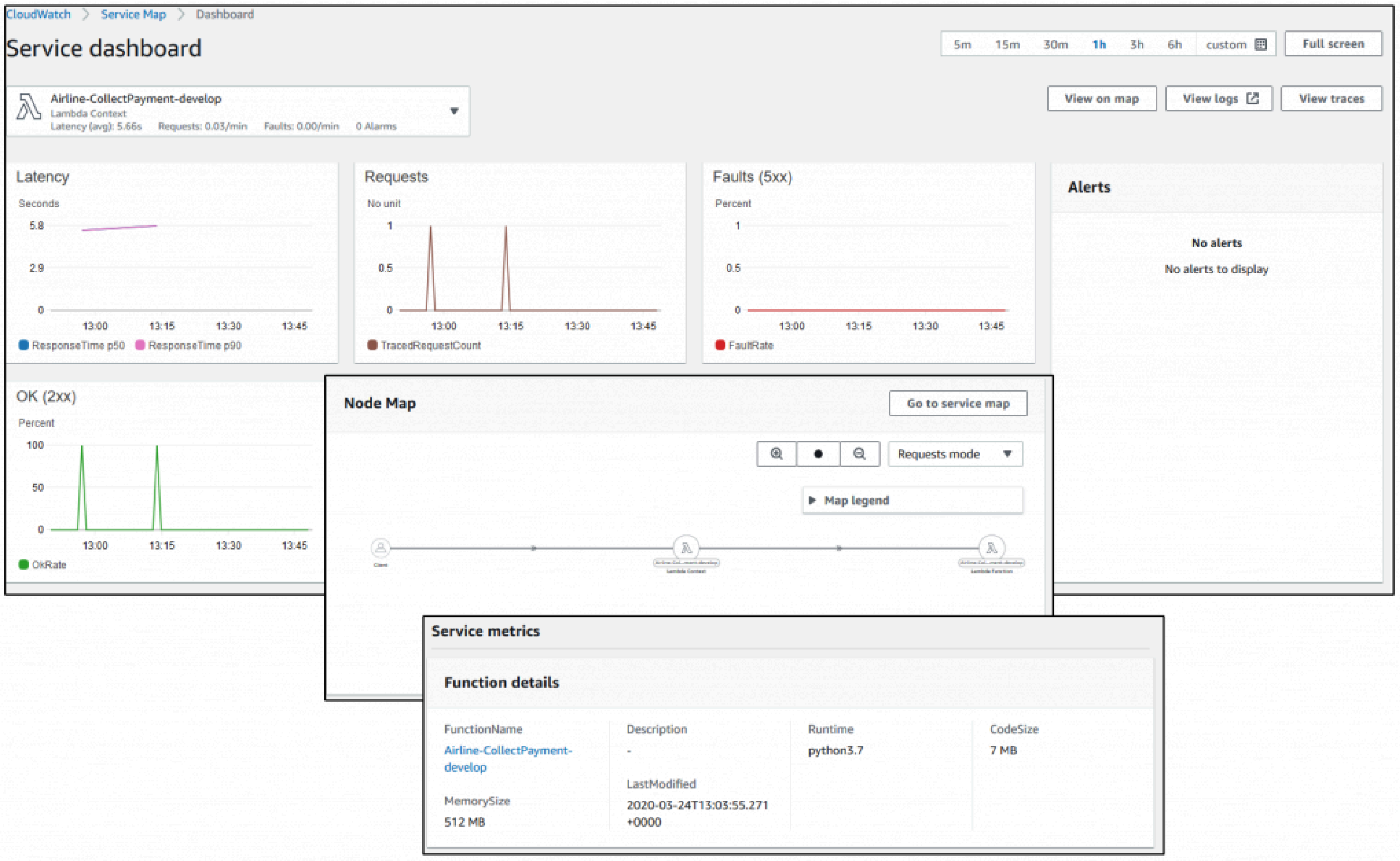
I select the *Airline-CollectPayment-develop* node and can view the out-of-the-box standard Lambda metrics.

I can select **View Logs** to jump to the CloudWatch Logs Insights console.


CloudWatch > Service Map
Service map
5m 15m 30m 1h 3h 6h custom
Exit full screen
Map view
List view
Filter by X-Ray group
Airline-CollectPayment-develop
View connections
Requests mode
Map legend
Fault (5xx)
Error (4xx)
Throttle
Current alarms
AWS service:
AWS resource:
Unknown node:
Size is requests
Client
Airline-Not...ing-develop Lambda Context
Airline-Not...ing-develop Lambda Function
amplify-aw...6GEK8N2A3N SNS Topic
Airline-Ing...lty-develop Lambda Context
Airline-Col...ment-develop Lambda Context
Airline-Col...ment-develop Lambda Function
amplify-...OK1MUJG2R Lambda Context
amplify-...OK1MUJG2R Lambda Function
amplify-...JWD0LWKN SNS Topic
amplify-a...ITW1/prod
Airline-CollectPayment-develop
Lambda Context
View logs
View traces
View dashboard
Latency (avg): 5.80s Requests: 0.03/min Faults: 0.00/min
Latency
Seconds
ResponseTime p50
ResponseTime p90
Requests
No unit
TracedRequestCount
Faults (5xx)
Percent
FaultRate
Alerts
No alerts
No alerts to display

CloudWatch Insights Service map

I select **View dashboard** to see the function metrics, node map, and function details.

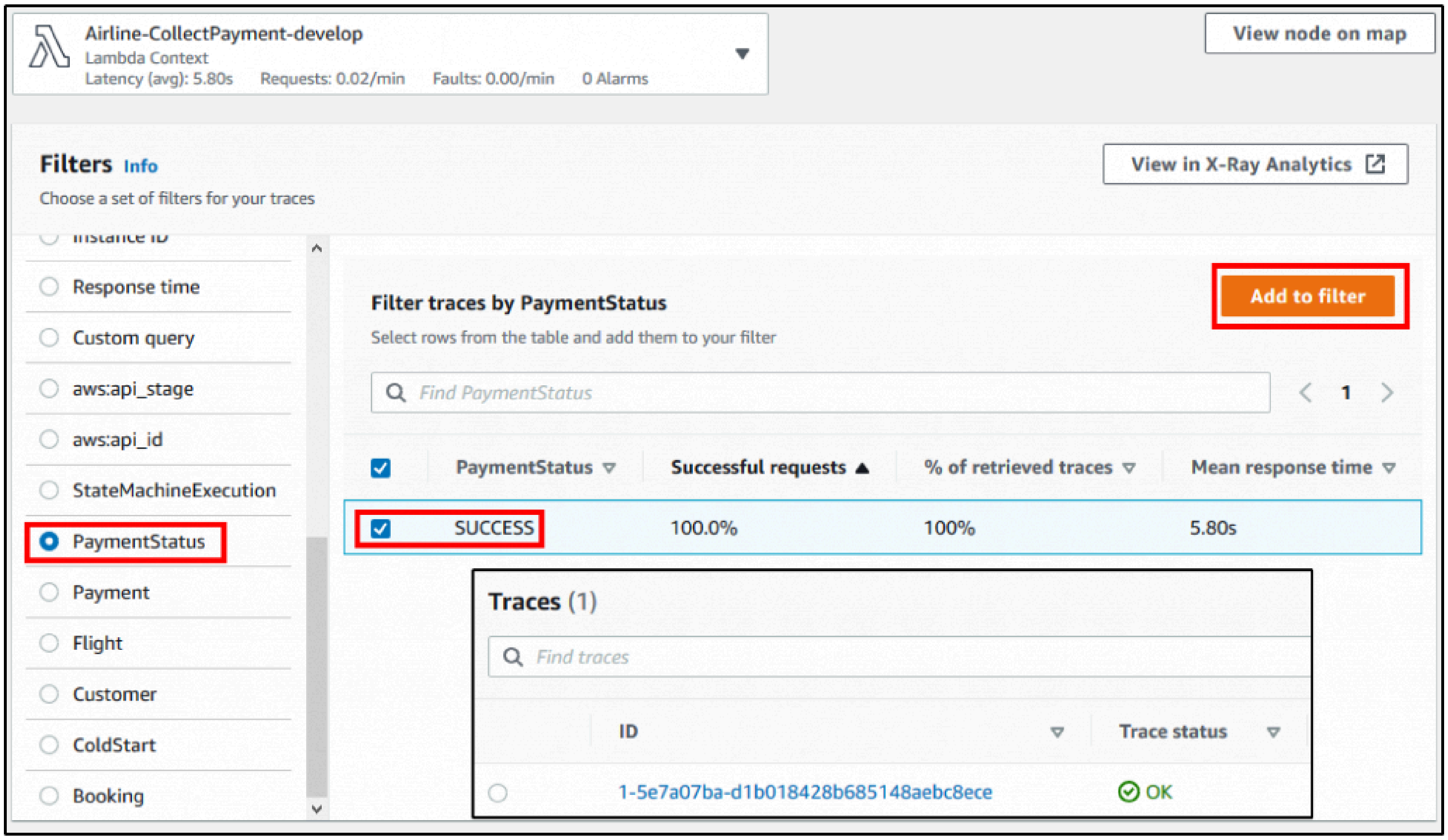


Function details

FunctionName	Description	Runtime	CodeSize
Airline-CollectPayment-develop	-	python3.7	7 MB
MemorySize	LastModified		
512 MB	2020-03-24T13:03:55.271 +0000		

CloudWatch Insights Service Map dashboard

I select **View traces** and can filter by the custom metric *PaymentStatus*. I select SUCCESS and chose **Add to filter**. I then select a trace.

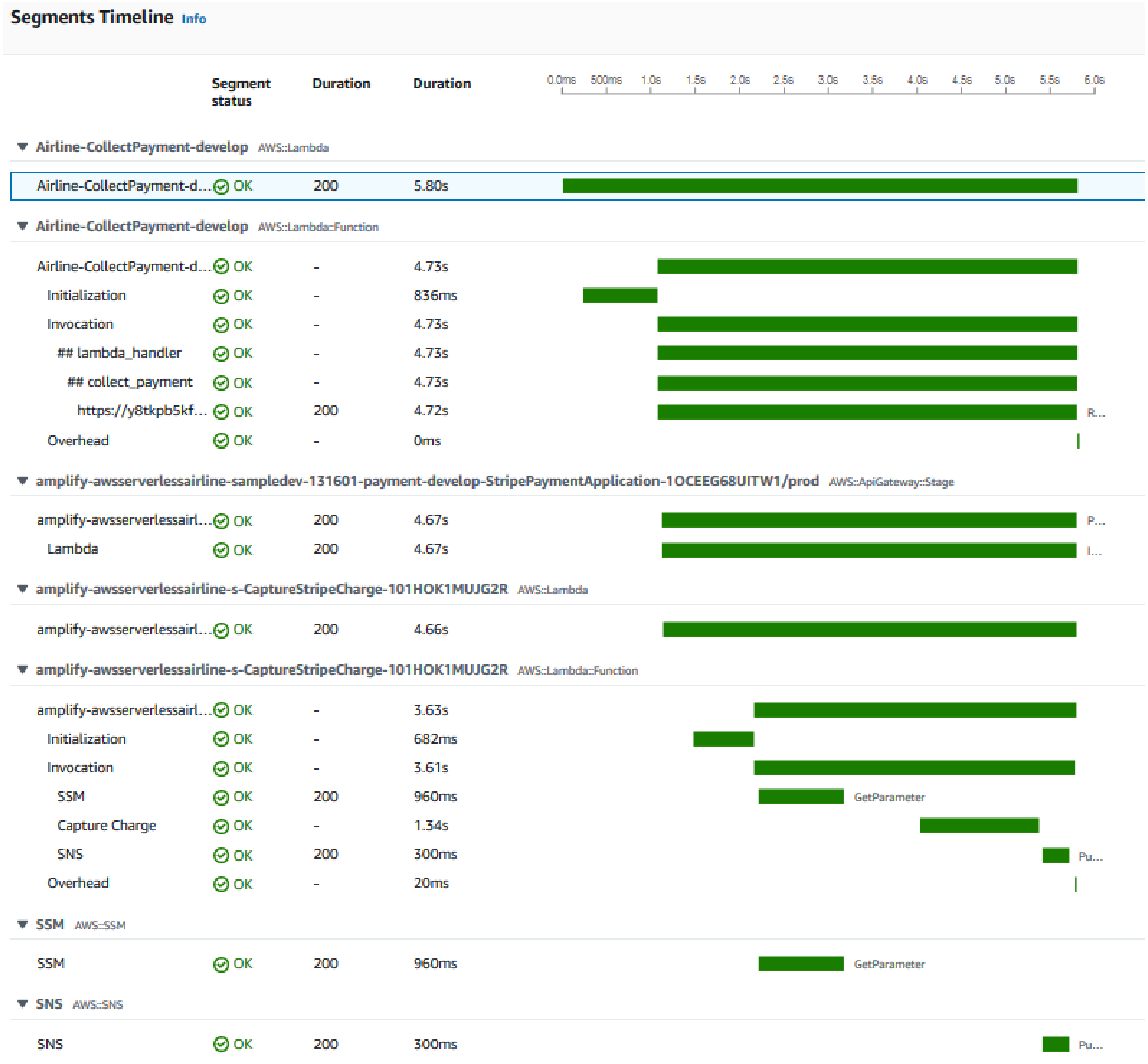


	PaymentStatus	Successful requests	% of retrieved traces	Mean response time
✓	SUCCESS	100.0%	100%	5.80s

	ID	Trace status
○	1-5e7a07ba-d1b018428b685148aebc8ece	OK

CloudWatch Insights Filtered traces

I see the full trace details, which show the full application transaction of a payment collection.



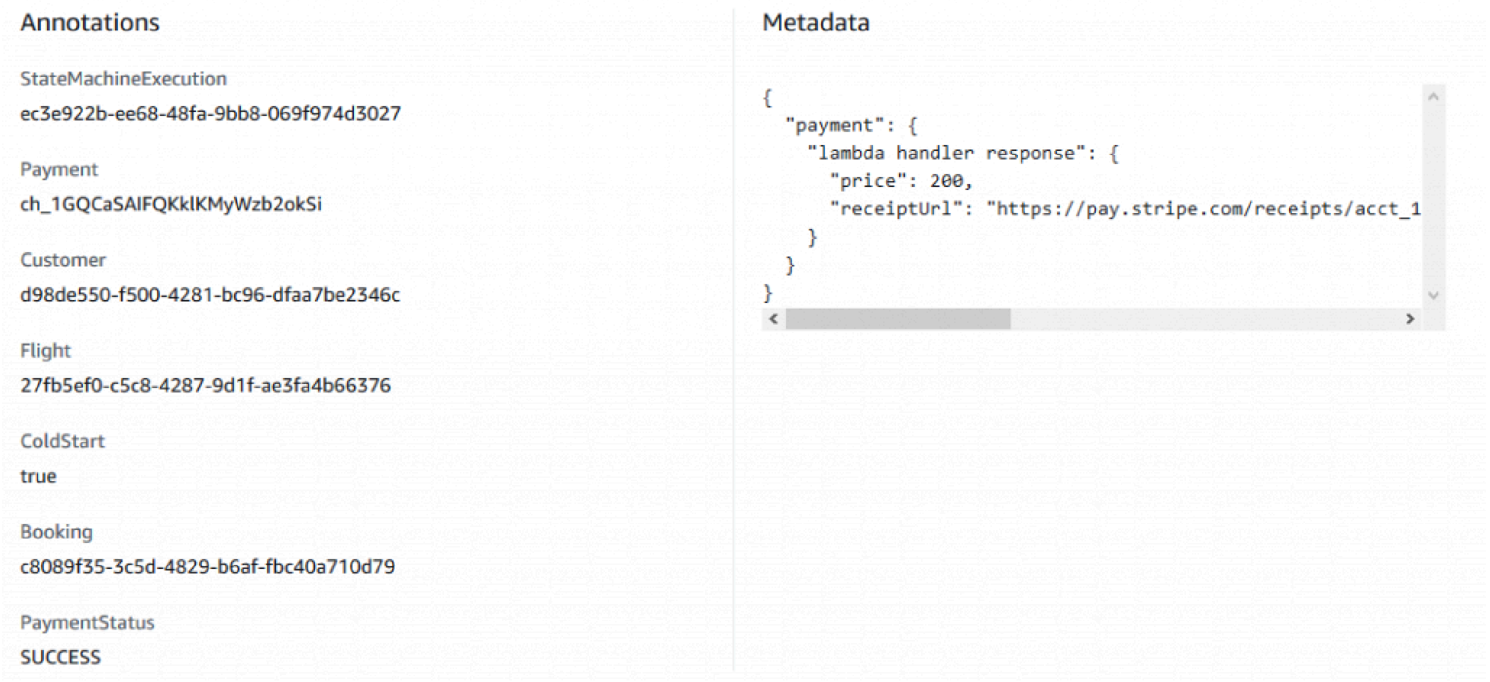
Segments Timeline Info

	Segment status	Duration	Duration	
▼ **Airline-CollectPayment-develop** AWS::Lambda				
Airline-CollectPayment-d…	OK	200	5.80s	
▼ **Airline-CollectPayment-develop** AWS::Lambda::Function				
Airline-CollectPayment-d…	OK	-	4.73s	
Initialization	OK	-	836ms	
Invocation	OK	-	4.73s	
## lambda_handler	OK	-	4.73s	
## collect_payment	OK	-	4.73s	
https://y8tkpb5kf…	OK	200	4.72s	R…
Overhead	OK	-	0ms	
▼ **amplify-awsserverlessairline-sampledev-131601-payment-develop-StripePaymentApplication-1OCEEG68UITW1/prod** AWS::ApiGateway::Stage				
amplify-awsserverlessairl…	OK	200	4.67s	P…
Lambda	OK	200	4.67s	I…
▼ **amplify-awsserverlessairline-s-CaptureStripeCharge-101HOK1MUJG2R** AWS::Lambda				
amplify-awsserverlessairl…	OK	200	4.66s	
▼ **amplify-awsserverlessairline-s-CaptureStripeCharge-101HOK1MUJG2R** AWS::Lambda::Function				
amplify-awsserverlessairl…	OK	-	3.63s	
Initialization	OK	-	682ms	
Invocation	OK	-	3.61s	
SSM	OK	200	960ms	GetParameter
Capture Charge	OK	-	1.34s	
SNS	OK	200	300ms	Pu…
Overhead	OK	-	20ms	
▼ **SSM** AWS::SSM				
SSM	OK	200	960ms	GetParameter
▼ **SNS** AWS::SNS				
SNS	OK	200	300ms	Pu…

0.0ms 500ms 1.0s 1.5s 2.0s 2.5s 3.0s 3.5s 4.0s 4.5s 5.0s 5.5s 6.0s

Segments timeline

Selecting the Lambda handler subsegment – *## lambda_handler,* I can view the trace Annotations and Metadata, which include the business transaction details such as *Customer* and *PaymentStatus*.


Annotations

StateMachineExecution
ec3e922b-ee68-48fa-9bb8-069f974d3027

Payment
ch_1GQCaSAIFQKklKMyWzb2okSi

Customer
d98de550-f500-4281-bc96-dfaa7be2346c

Flight
27fb5ef0-c5c8-4287-9d1f-ae3fa4b66376

ColdStart
true

Booking
c8089f35-3c5d-4829-b6af-fbc40a710d79

PaymentStatus
SUCCESS

Metadata

```
{
  "payment": {
    "lambda handler response": {
      "price": 200,
      "receiptUrl": "https://pay.stripe.com/receipts/acct_1
    }
  }
}
```

X-Ray annotations

Trace groups are another feature of X-Ray and ServiceLens. Trace groups use filter expressions such as `Annotation.PaymentStatus = "FAILED"` which are used to view traces that match the particular group. Service graphs can also be viewed, and CloudWatch alarms created based on the group.

CloudWatch ServiceLens provides powerful capabilities to understand application performance bottlenecks and issues, helping determine how users are impacted.

Improvement plan summary

1. Identify common business context and system data that are commonly present across multiple transactions.
2. Instrument SDKs and requests to upstream/downstream services to understand the flow of a transaction across system components.

Recent announcements

There have been a number of recent announcements for X-Ray and CloudWatch to improve how to evaluate serverless application health.

- CloudWatch ServiceLens – a “single pane of glass” monitoring tool
- CloudWatch Synthetics – Generate “canary” tests for your APIs and other services
- Cloudwatch composite alarms – Combine multiple alarms to reduce alerting noise.
- Embedded Metric Format for CloudWatch Logs
- CloudWatch Contributor Insights – Identify top talkers, common patterns in logs
- X-Ray Trace Maps – Map a single event though your distributed application

- X-Ray integration with CloudWatch Synthetics

Conclusion

Evaluating application health helps you identify which services should be optimized to improve your customer's experience. In part 1, I cover out-of-the-box standard metrics and alerts, as well as structured and centralized logging. In this post, I explore custom metrics and distributed tracing and show how to use ServiceLens to view logs, metrics, and traces together.

In the next post in the series, I cover the second Operational Excellence question from the Well-Architected Serverless Lens – Approaching application lifecycle management – part 1.

TAGS: Amazon CloudWatch, AWS Lambda, CloudWatch Log, lambda, Python, serverless, well-architected