

Building well-architected serverless applications: Understanding application health – part 1

by Julian Wood | on 02 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?

Evaluating your metrics, distributed tracing, and logging gives you insight into business and operational events, and helps you understand which services should be optimized to improve your customer's experience. By understanding the health of your Serverless Application, you will know whether it is functioning as expected, and can proactively react to any signals that indicate it is becoming unhealthy.

Required practice: Understand, analyze, and alert on metrics provided out of the box

It is important to understand metrics for every AWS service used in your application so you can decide how to measure its behavior. AWS services provide a number of out-of-the-box standard metrics to help monitor the operational health of your application.

As these metrics are generated automatically, it is a simple way to start monitoring your application and can also be augmented with custom metrics.

The first stage is to identify which services the application uses. The airline booking component uses [AWS Step Functions](#), [AWS Lambda](#), [Amazon SNS](#), and [Amazon DynamoDB](#).

When I make a booking, as shown in the [Introduction post](#), AWS services emit metrics to [Amazon CloudWatch](#). These are processed asynchronously without impacting the application's performance.

There are two default CloudWatch dashboards to visualize key metrics quickly: per service and cross service.

Per service

To view the per service metrics dashboard, I open the [CloudWatch console](#).

CloudWatch: Overview ▾

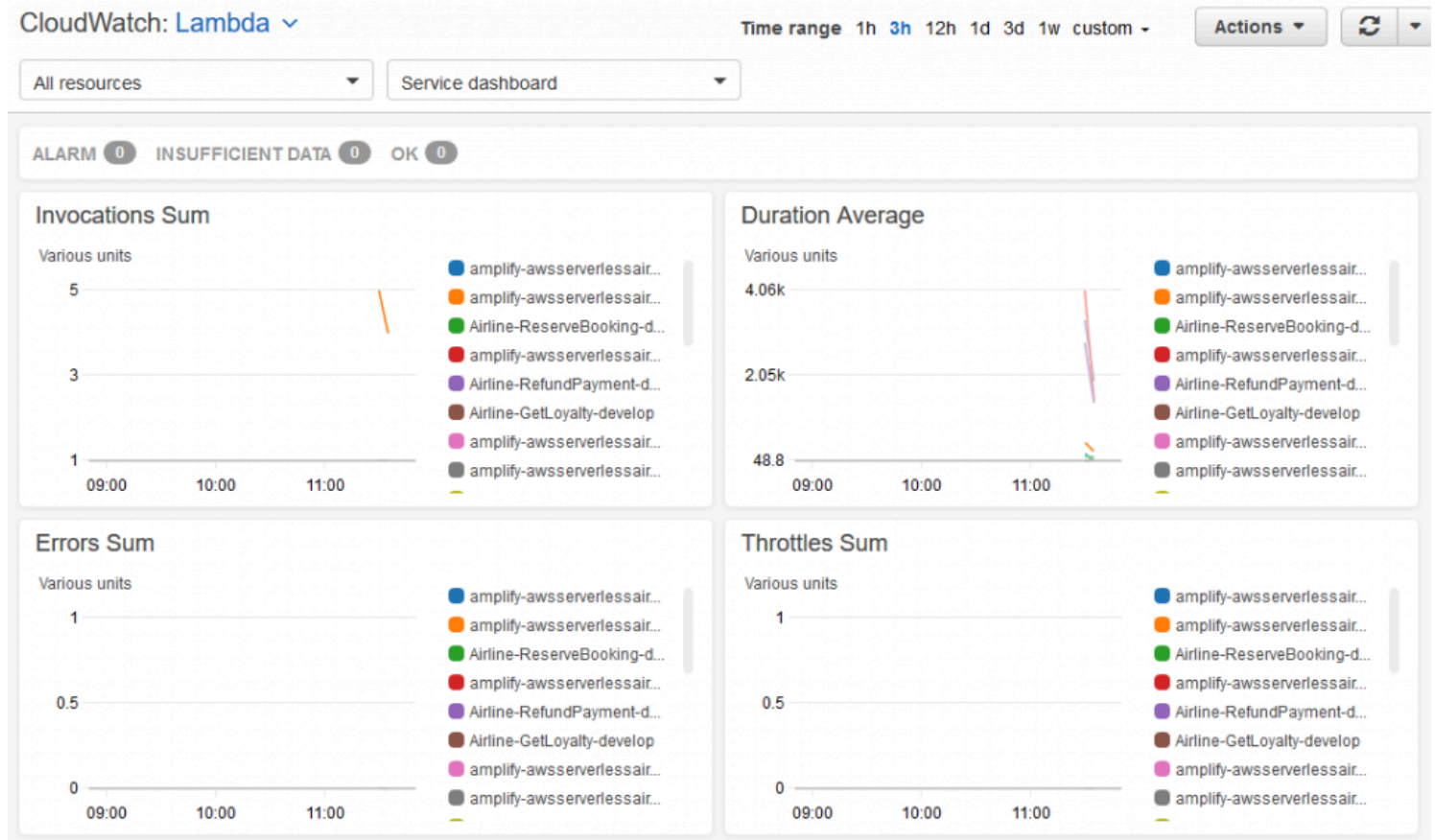
All resources ▾

Alarms by AWS service ⓘ

Services			
Status	Alarm	Insufficient	OK
✓ API Gateway	-	-	1
? AWS/X-Ray	-	-	-
? AppSync	-	-	-
? CloudWatch Events	-	-	-
? CloudWatch Logs	-	-	-
? Cognito	-	-	-
? DynamoDB	-	-	-
? EC2	-	-	-
? Elastic Block Store	-	-	-
? Kinesis Data Firehose	-	-	-

Per service metrics dashboard

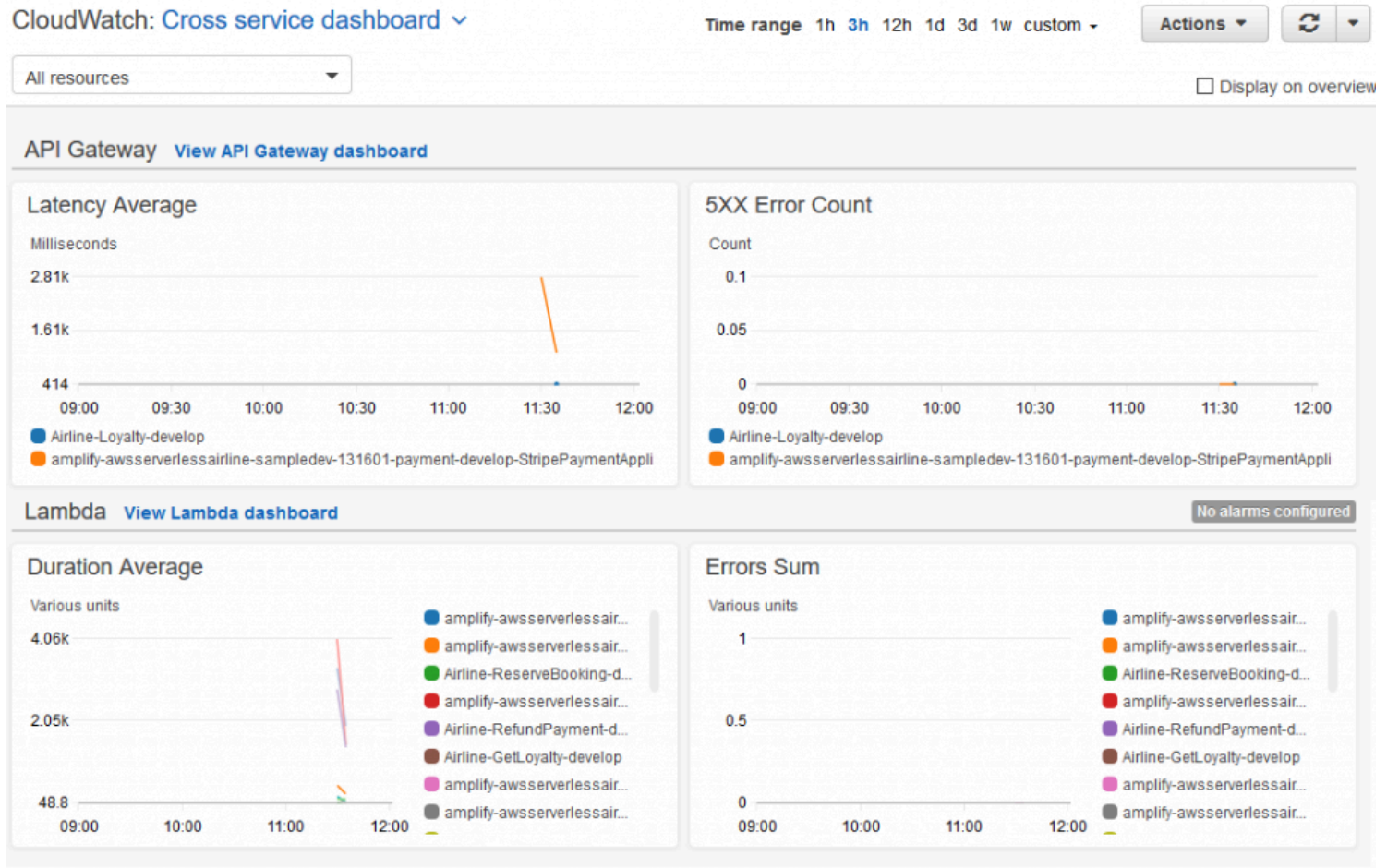
I select a service where **Overview** is shown, such as **Lambda**. Now I can view the metrics for all Lambda functions in the account.



Per service metrics for lambda

Cross service

To see an overview of key metrics across all AWS services, open the [CloudWatch console](#) and choose **View cross service dashboard**.



Cross service metrics dashboard

I see a list of all services with one or two key metrics displayed. This provides a good overview of all services your application uses.

Alerting

The next stage is to identify the key metrics for comparison and set up alerts for under- and over-performing services. Here are some [recommended metrics to alarm on](#) for a number of AWS services.

Alerts can be configured manually or via infrastructure as code tools such as the [AWS Serverless Application Model](#), [AWS CloudFormation](#), or third-party tools.

To configure a manual alert for Lambda function errors using CloudWatch Alarms:

1. I open the [CloudWatch console](#) and select **Alarms** and select **Create Alarm**.
2. I choose **Select Metric** and from AWS Namespaces, select **Lambda, Across All Functions** and select **Errors** and select **Select metric**.

All metricsGraphed metrics (1)Graph optionsSource

All > Lambda > Across All Functions Graph search

☐ Metric Name (6)

☐ ConcurrentExecutions

☐ Duration

☒ Errors

☐ Invocations

☐ Throttles

Cancel

Select metric

Add metric to alert

3. I change the **Statistic** to *Sum* and the **Period** to *1 minute*.

MetricEdit

Graph

This alarm will trigger when the blue line goes above the red line for 1 datapoints within 1 minute.

Count

1

0.8

0.6

0.4

0.2

0

11:00

12:00

13:00

Errors

Namespace

AWS/Lambda

Metric name

Errors

Statistic

Period

1 minute

Metric values

4. Under **Conditions**, I select a **Static** threshold **Greater** than *1* and select Next.

Conditions

Threshold type

☒ **Static**
Use a value as a threshold

☐ **Anomaly detection**
Use a band as a threshold

Whenever Errors is...
Define the alarm condition.

☒ **Greater**
> threshold

☐ **Greater/Equal**
≥ threshold

☐ **Lower/Equal**
≤ threshold

☐ **Lower**
< threshold

than...
Define the threshold value.

Must be a number

► Additional configuration

Metric Conditions

Alarms can also be created using [anomaly detection](#) rather than static values if there is a discernible pattern or trend. Anomaly detection looks at past metric data and uses machine learning to create a model of expected values. Alerts can then be configured if they fall outside this band of “normal” values. I use a Static threshold for this alarm.

5. For the notification, I set the trigger to alarm to an existing SNS topic with my email address, then choose **Next**.

Notification

Alarm state trigger
Define the alarm state that will trigger this action. Remove

☒ **In alarm**
The metric or expression is outside of the defined threshold.

☐ **OK**
The metric or expression is within the defined threshold.

☐ **Insufficient data**
The alarm has just started or not enough data is available.

Select an SNS topic
Define the SNS (Simple Notification Service) topic that will receive the notification.

☒ **Select an existing SNS topic**


☐ Create new topic

☐ Use topic ARN

Send a notification to...

Only email lists for this account are available.

Email (endpoints)

 - [View in SNS Console](#)

Metric Notification

6. I enter a descriptive alarm name such as *serverlessairline-lambda-prod-errors > 1*, select **Next**, and choose **Create alarm**.

I have now manually set up an alarm.

Use [CloudWatch composite alarms](#) to combine multiple alarms to reduce noise and focus on critical issues. For example, a single alarm could trigger if there are both Lambda function errors as well as high Lambda concurrent executions.

It is simpler and more scalable to include alerting within infrastructure as code. Here is an example of [alerting programmatically using CloudFormation](#).

I view the out of the box standard metrics and in this example, manually create an alarm for Lambda function errors.

Improvement plan summary

1. Understand what metrics and dimensions each managed service used provides.
2. Configure alerts on relevant metrics for when services are unhealthy.

Good practice: Use structured and centralized logging

Central logging provides a single place to search and analyze logs. Structured logging means selecting a consistent log format and content structure to simplify querying across multiple components.

To identify a business transaction across components, such as a particular flight booking, log operational information from upstream and downstream services. Add information such as `customer_id` along with business outcomes such as `order=accepted` or `order=confirmed`. Make sure you are not logging any sensitive or personal identifying data in any logs.

Use JSON as your logging output format. Log multiple fields in a single object or dictionary rather than many one line messages for simpler searching.

Here is an [example of a structured logging format](#).

The airline booking component, which is written in Python, currently uses a [shared library](#) with a separate log processing stack.

[Embedded Metrics Format](#) is a simpler mechanism to replace the shared library and use structured logging. CloudWatch Embedded Metrics adds environmental metadata such as Lambda Function version and also automatically extracts custom metrics so you can visualize and alarm on them. There are open-source client libraries available for [Node.js and Python](#).

I then add embedded metrics to the individual [confirm booking module](#) with the following steps:

1. I install the [aws-embedded-metrics library using the instructions](#).
2. In the function init code, I import the module and create a `metric_scope` with the following code

Python

```
from aws_embedded_metrics import metric_scope
@metric_scope
```

3. In the function handler, I log the generated `bookingReference` with the following code.

Python

```
metrics.set_property("BookingReference", ret["bookingReference"])
```

In this example I also log the entire incoming event details.

Python

```
metrics.set_property("event", event)
```


It is best practice to only log what is required to avoid unnecessary costs. Ensure the event does not have any sensitive or personal identifying data which is available to anyone who has access to the logs.

To avoid the duplicate logging in this example airline application which adds cost, I remove the existing shared library `logger.*()` lines.

When I make a booking, the CloudWatch log message is in structured JSON format. It contains the properties I set `event, BookingReference`, as well as function metadata.

```
{
  "LogGroup": "Airline-ConfirmBooking-develop",
  "ServiceName": "Airline-ConfirmBooking-develop",
  "ServiceType": "AWS::Lambda::Function",
  "event": {
    "outboundFlightId": "27fb5ef0-c5c8-4287-9d1f-ae3fa4b66376",
    "customerId": "d98de550-f500-4281-bc96-dfaa7be2346c",
    "chargeId": "ch_1GPoWlAIFQKk1KMybGDU4oRa",
    "bookingTable": "Booking-vgxe6n4chbf5bobhm6ac2576mm-sampledev",
    "flightTable": "Flight-vgxe6n4chbf5bobhm6ac2576mm-sampledev",
    "name": "235cc76a-6a60-4ee2-8225-f6415c95d8bc",
    "createdAt": "2020-03-23T11:33:08.433Z",
    "bookingId": "63026eb3-7229-4834-a0ef-d743c9eece06",
    "payment": {
      "receiptUrl": "https://pay.stripe.com/receipts/acct_1FX8D8AIFQKk1KMy/ch_1GPoWlAIFQKk1KMybGDU4oRa/rcpt_Gxk041TYqTqB1OW6H1HUkgtJuDO3xG0",
      "price": 200
    }
  },
  "BookingReference": "A2_hvA",
  "executionEnvironment": "AWS_Lambda_python3.7",
  "memorySize": "512",
  "functionVersion": "$LATEST",
  "logStreamId": "2020/03/23/[$LATEST]c90d3832ac6041c2bcdca85938be128c",
  "traceId": "Root=1-5e789e7b-e4aede6c4c4c05e0d2ef955c;Parent=b2e6e7304ac28abf;Sampled=1",
  "_aws": {
    "Timestamp": 1584963197050,
    "CloudWatchMetrics": [
      {
        "Dimensions": [
          [
            "LogGroup",
            "ServiceName",
            "ServiceType"
          ]
        ],
        "Metrics": [],
        "Namespace": "aws-embedded-metrics"
      }
    ]
  }
}
```

Structure log example

I can then search for all log activity related to a specific booking across multiple functions with `booking_id`. I can track customer activity across multiple bookings using `customer_id`.

Logging is often created as a shared library resource which all functions reference. Another option is using [Lambda Layers](#), which lets functions import additional code such as external libraries. Multiple functions can share this code.

Improvement plan summary

1. Log request identifiers from downstream services, component name, component runtime information, unique correlation identifiers, and information that helps identify a business transaction.

2. Use JSON as the logging output. Prefer logging entire objects/dictionaries rather than many one line messages. Mask or remove sensitive data when logging.
3. Minimize logging debugging information to a minimum as they can incur both costs and increase noise to signal ratio

Conclusion

Evaluating serverless application health helps understand which services should be optimized to improve your customer's experience. I cover out of the box metrics and alerts, as well as structured and centralized logging.

This well-architected question continues in [part 2](#) where I look at custom metrics and distributed tracing.

TAGS: [Amazon CloudWatch](#), [AWS Lambda](#), [CloudWatch Logs](#), [lambda](#), [Python](#), [serverless](#), [well-architected](#)