


Building Serverless Observability Tools With Custom Metrics and Dashboards in CDK



Luke Yianni · Follow

Published in Serverless Transformation · 7 min read · Nov 2, 2022

👍 523

💬

🔖

🕒

📄

⋮



Meaningful logging is a must for Serverless architectures, and of course, there are a lot of third-party tools that can help with observability. However, there is a lot of untapped potential in using Cloudwatch Dashboards and Alarms. This article will show you how to have powerful observability, natively, across your project.

Current observability solutions in Serverless

Serverless (and almost all cloud) projects comprise multiple services carrying out different functions to form your product. This modularity is incredibly powerful, but by its very nature makes it hard to track any issues — there's a tonne of moving parts and it's often difficult to tell exactly where something went wrong.

This lack of observability is **one of the largest gripes** people have when adopting serverless, and companies have relied on a huge host of pre-built tools to help better understand their products. Until now, setting up robust monitoring using AWS services has been a painful process that takes up a lot of upfront work before getting any real benefit.

Now, The AWS Cloud Development Kit (AWS CDK) changes all of that.

CDK is an open-source framework to create Infrastructure as Code (IaC) and provision it through AWS CloudFormation. It supports the most common Object Oriented Programming Languages, and for Serverless that's most importantly Typescript.

CDK utilises this Object Oriented paradigm, and as a result, is instinctually modular and extendable. This allows you to abstract dashboards to use across services and teams, being incredibly easy to create custom constructs compared to other Cloud Formation-creating IaC that use YAML syntax. This has changed the creation of observability tools from arduous to automatic, and now makes **native the solution for project observability**.

Existing tools still have their place and for off-the-shelf there is no better, but for highly customizable observation in environments where you can't leverage external tools, CloudWatch IaC provides a powerful and flexible option if you're willing to dedicate the time.

Fair Warning

CDK is a maturing product, and at times, the documentation can be both sparse and complicated — it's easy to get lost down a rabbit hole, which I spent an embarrassingly long amount of time doing.

At the end of that process, however, I've now worked with the tool enough to be able to build **incredibly flexible** monitoring tools, alongside setting automated alarms should any issues arise. This article should help demystify Cloudwatch in CDK, and set you in the right direction to monitor your Serverless applications.

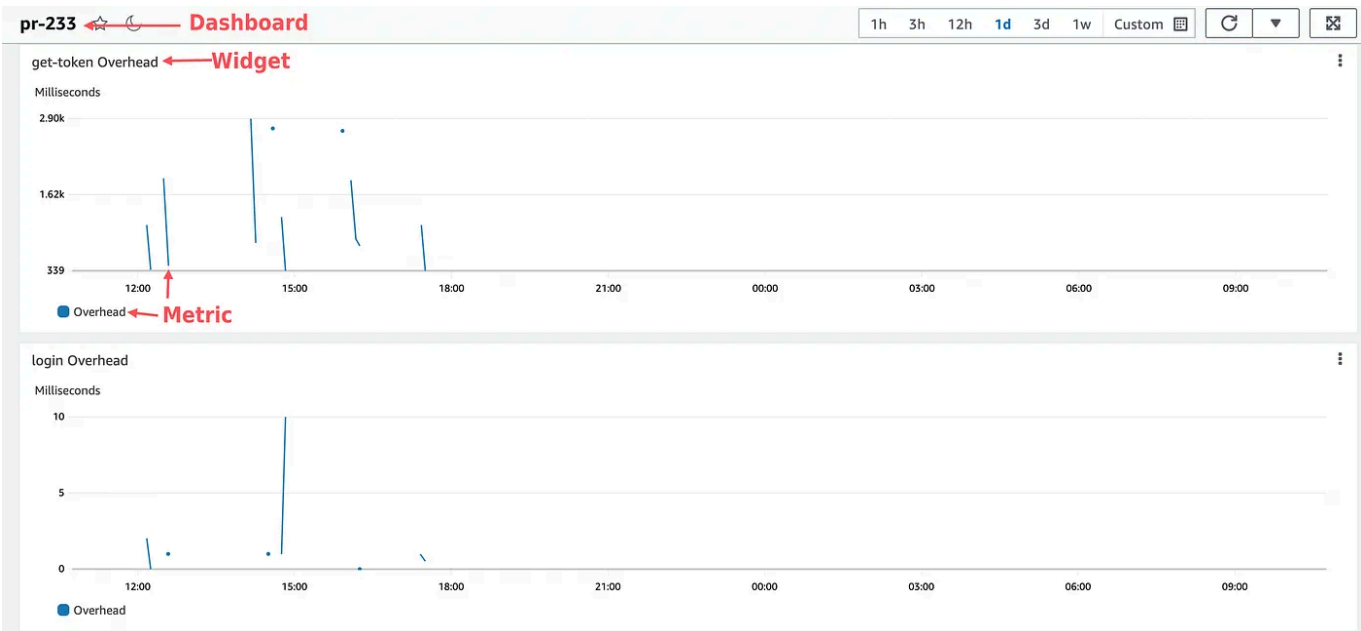
We will cover:

- The infrastructure for a Cloudwatch Dashboard,
- How to publish your custom metrics
- How to create widgets and a dashboard to represent them
- Setting up an Alarm for a Metric
- Sending an email with SNS when an Alarm is triggered
- Storing an email list in SSM

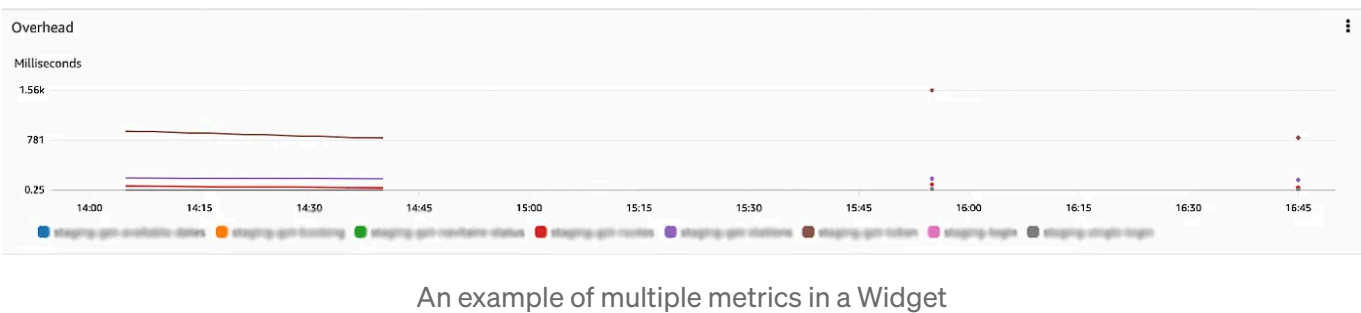
The pieces that make up a Cloudwatch Dashboard

A Cloudwatch Dashboard is made up of three parts:

- Metrics — the actual data you want to show in the dashboard
- Widgets — effectively a 'graph' in the dashboard
- Dashboards — the collection of widgets



A Dashboard can contain multiple Widgets, and a Widget can visualise multiple Metrics.



An example of multiple metrics in a Widget

Creating a Metric

The first step in creating a dashboard is to create a metric. There are two kinds, default and custom metrics.

Default Metrics

By default, lambdas already come with some metrics you can record, such as Errors and the Number of Invocations. The whole list can be found here.

It's straightforward to reference these metrics in CDK, and the official documentation for doing so is quite good!

Here's how you reference a metric that shows the average number of errors per minute.

```
declare const fn: lambda.Function;

const minuteErrorRate = fn.metricErrors({ statistic: 'avg',
period: Duration.minutes(1), label: 'Lambda failure rate' });
```

You could then reference this metric in your widgets (covered below). It's more difficult if you want to measure something not covered by these default metrics. With more complicated projects, I have found that you will often have to rely on creating your own metrics.

Publishing a Custom Metric

To instantiate a custom metric, you have to publish a data point. As publishing data during run-time isn't IaC, you have to use the [SDK](#), using the `putMetricData` function. This creates an instance of a datapoint, which you can then aggregate to create a metric.

Here's an example call that would facilitate a similar Error Count metric, initially creating the datapoint and then creating a metric from that.

Creating the Data Point:

```
const params = {
  MetricData: [
    {
      MetricName: 'Errors Thrown',
      Dimensions: [
        {
          Name: 'Error Code',
          Value: request?.error?.code.toString() ?? '500',
        },
        {
          Name: 'Function Name',
          Value: request.context?.functionName ?? 'Unknown Function',
        },
      ],
      Unit: 'None',
      Value: 1,
      Timestamp: new Date(),
    },
  ],
  Namespace: `${process.env['STAGE']}`, //STAGE
};
await cloudwatch.putMetricData(params).promise();
```

Creating the Metric:

To then reference the Metric you are publishing in IaC (for use in Dashboards and Alarms), you could then create the following!

```
const errorRate = new Metric({
  namespace: stage,
  metricName: 'Errors Thrown',
  region: 'eu-west-1',
  dimensionsMap: {
    'Error Code': 500,
    'Function Name': functionName,
  },
  period: Duration.minutes(1),
  statistic: 'Average',
});
```

The key part to note is the metricName, dimensionsMap, region and namespace all have to be the same as the ones that you *put*.

Creating a Dashboard & Widget

Creating the metric is the difficult part, and the CDK documentation for creating a Widget and Dashboard is fairly good.

Firstly, to create a dashboard you could write the following:

```
const lambdaDashboard = new Dashboard(this, 'MyDashboardId', {
  dashboardName: 'MyDashboardName',
});
```

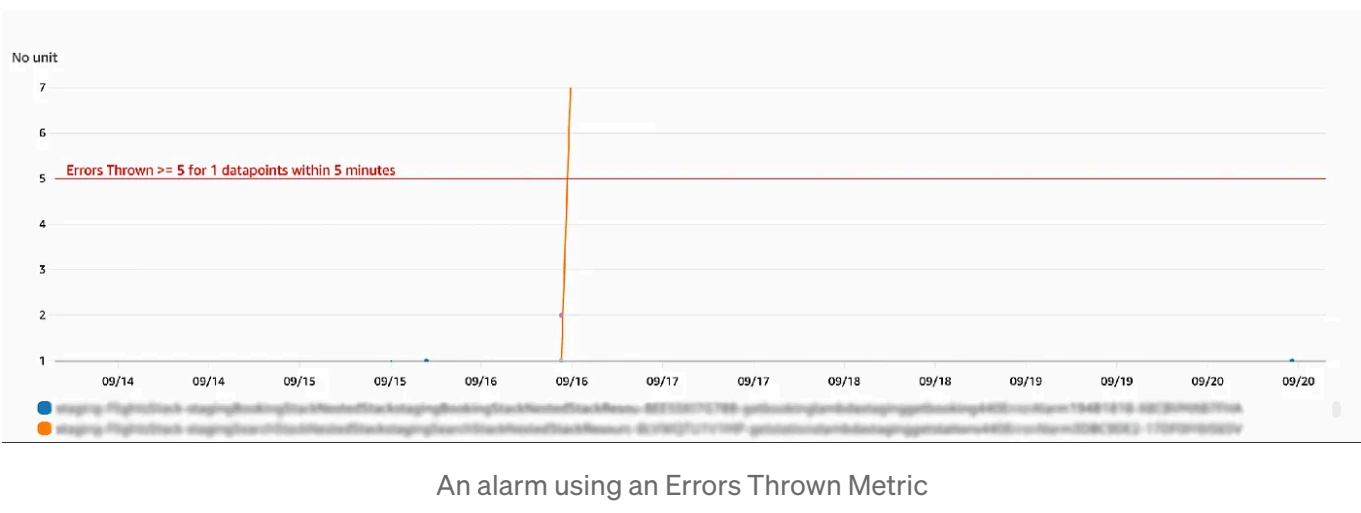
And the following to add a Widget:

```
lambdaDashboard.addWidgets(
  new GraphWidget({
    title: `${name} Errors`,
    width: 24,
    left: [
      myMetric
    ],
  }),
);
```

Piecing all of this together, you would have a working dashboard! One of the great things about CDK is your [ability to create constructs create multiple AWS constructs](#), which I show an example of at the end of the article.

However, now that you have a metric to measure, perhaps you want to be notified when a metric is a certain value...

Creating an Alarm



Cloudwatch Alarms are events that are triggered when a Metric value passes a threshold. Common examples would be an SNS topic to notify people about the alarm (e.g. via email) or triggering an autoscaling policy.

Creating an Alarm is mercifully fairly easy in CDK, but you will likely have to involve other services as the resultant action of that alarm being triggered.

For my project, I was working on a wrapper around a complicated third-party API, and we wanted to email our devs if the overhead time (the additional time our lambda takes above the time spent calling the third-party endpoint) is above one second. Without going into too much detail, you would use SNS to create a topic that the Alarm publishes to; the topic would have Email Subscribers that would receive the email the Alarm creates.

Customising the SNS email was out of the scope of this work, and the default one worked just as well.

To create the SNS Topic, you would write the following:

```
const overheadTopic = new Topic(this,
  buildResourceName('OverheadSNSTopic'));
overheadTopic.addSubscription(new EmailSubscription(email));
snsAction = new SnsAction(overheadTopic);
```

Then you could instantiate the Alarm, and add an action to it to send to that Topic.

```
const overheadAlarm = new Alarm(
  this,
  `${name} Overhead Alarm`,
  {
    metric: overheadMetric,
    threshold: 1000,
    comparisonOperator:
      ComparisonOperator.GREATER_THAN_OR_EQUAL_TO_THRESHOLD,
    evaluationPeriods: 1,
    alarmDescription: 'Alarm if the overhead for a lambda is above 1 second',
  },
);
overheadAlarm.addAlarmAction(snsAction);
```

You could opt not to have an action with your alarm, and in CloudWatch it would still indicate the alarm has been triggered — you just wouldn't have any notification to check it.

Putting everything Together

Accumulating all the constructs discussed in this article, we could have a construct like the one below. The one in my project is incredibly similar,

with the only difference being that I create a REST endpoint as well as the constructs below — the Object Oriented nature of CDK helps emphasise the power of creating custom constructs!

Depending on the granularity of your project, you can flexibly change when and how you are instantiating different constructs. For example, you might want to have a new dashboard for each Lambda, and can easily do so by creating a Dashboard in the Lambda Construct below, rather than passing it in as a parameter. For our use case, we only wanted one Dashboard and SNS Topic per stack, so created them at the top level of our project and passed it down as a parameter.

I hope this has been a helpful insight into Cloudwatch in CDK, and shown you how powerful of a tool it can be!

```
type MyLambdaProps = {overheadMetric
vpc: IVpc;
securityGroups: [ISecurityGroup];
role: Role;
name: string;
entry: string;
dashboard: Dashboard;
timeout?: Duration;
snsAction?: SnsAction;
};

export class MyLambda extends Construct {
functionArn: string;
constructor(scope: Construct, id: string, props: MyLambdaProps) {

super(scope, id);

const stage = process.env.STAGE ?? 'Unknown';

const {
vpc,
securityGroups,
role,
name,
entry,
apiResources,
httpMethod,
dashboard,
timeout,
snsAction,
} = props;

const overheadMetric = new Metric({
namespace: stage,
metricName: 'Overhead',
region: 'eu-west-1',
dimensionsMap: {
'Function Name': name,
},
period: Duration.minutes(5),
statistic: 'Average',
});
if (snsAction) {
const overheadAlarm = new Alarm(
this,
`${name} Overhead Alarm`,
{
metric: overheadMetric,
threshold: 1800,
comparisonOperator: ComparisonOperator.GREATER_THAN_OR_EQUAL_TO_THRESHOLD,
evaluationPeriods: 1,
alarmDescription: 'Alarm if the overhead for a lambda is above 1 second',
},
);
overheadAlarm.addAlarmAction(snsAction);
}

dashboard.addWidget(
new GraphWidget({
title: `${name} Overhead`,
width: 24,
left: overheadMetric,
}),
);
);

const lambda = new NodejsFunction(this, `${name}-lambda`, {
vpc: vpc,
handler: 'handler',
securityGroups: securityGroups,
runtime: Runtime.NODEJS_16_X,
entry: path.join(__dirname, `../${entry}`),
functionName: name,
environment: {
STAGE: stage,
},
role,
timeout: timeout ?? Duration.seconds(3),
});
};
```

Thank you for reading.

- Serverless
- AWS
- Cloudwatch
- Cdk
- Typescript

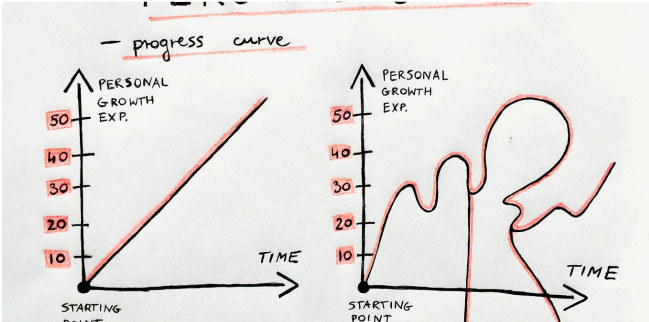


Written by Luke Yianni

17 Followers · Writer for Serverless Transformation

Follow

More from Luke Yianni and Serverless Transformation



Luke Yianni in Grounded

Slowing Everything Down

Low Expectations

5 min read · Jan 29, 2021

55

...



Xavier Lefevre in Serverless Transformation

Asynchronous client interaction in AWS Serverless: Polling...

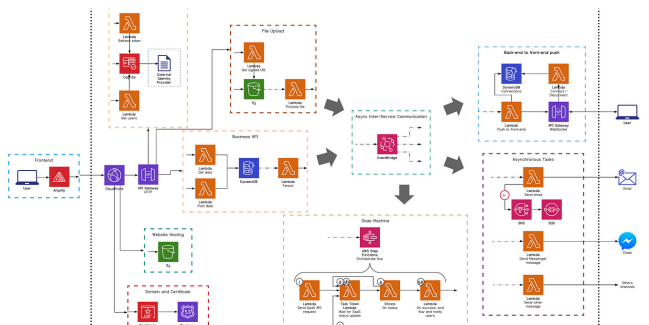
Event-driven Serverless often requires async update to the client. There are several ways ...

6 min read · Apr 18, 2020

730

7

...



Xavier Lefevre in Serverless Transformation

What a typical 100% Serverless Architecture looks like in AWS!

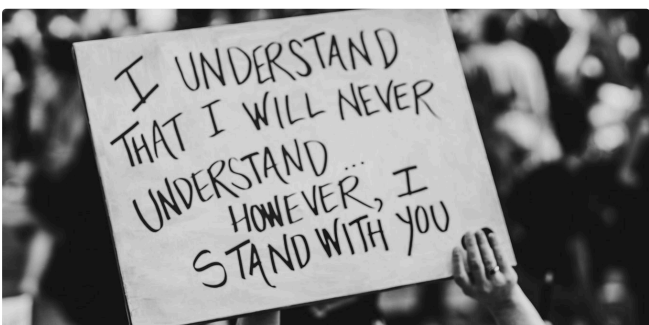
If you are new to serverless and looking for a high level web architecture guide, you've...

11 min read · May 19, 2020

2.8K

19

...



Luke Yianni in Grounded

How we can do better

I quite often go for late-night walks around the area I live in during lockdown, either...

7 min read · Mar 16, 2021

6

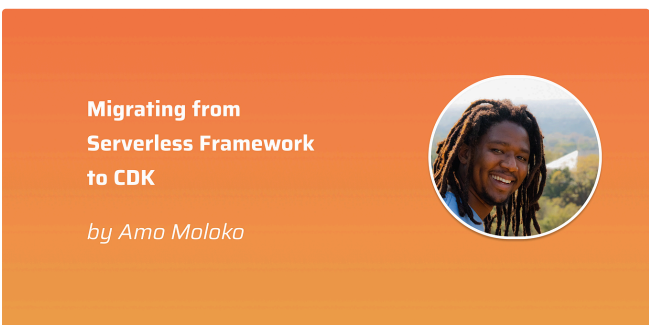
Q

...

See all from Luke Yianni

See all from Serverless Transformation

Recommended from Medium



Amo Moloko

Migrating from the Serverless Framework to AWS CDK

Serverless as a technology is no longer a fad as we venture into 2024. Aleios who have...

12 min read · 4 days ago



Vladdkens

Effortless REST API Deployment on AWS Lambda with Terraform...

A single workflow for creation and updating, using FastAPI & Docker.

5 min read · Oct 12, 2023