Going Serverless

Building Production Applications Without Managing Infrastructure

Objectives of this talk

- Outline what serverless means
- Discuss AWS Lambda and its considerations
- Delve into common application needs and how to address them in a serverless paradigm in AWS
- Detail my own experiences and thoughts in taking this approach

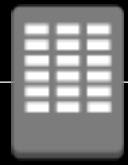
Who

Christopher Phillips

- Technical Backend Lead at Stanley Black and Decker
- 7+ years of development experience
- Delivered (and supported) production systems in Node, Erlang, Java
- Experienced with distributed, highly available, fault tolerant systems
- Serverless deployments since September 2015

What

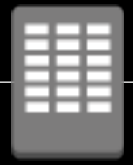
Pure hardware



Application

Bare metal servers

Virtualized hardware



Application

VMs

Hypervisor

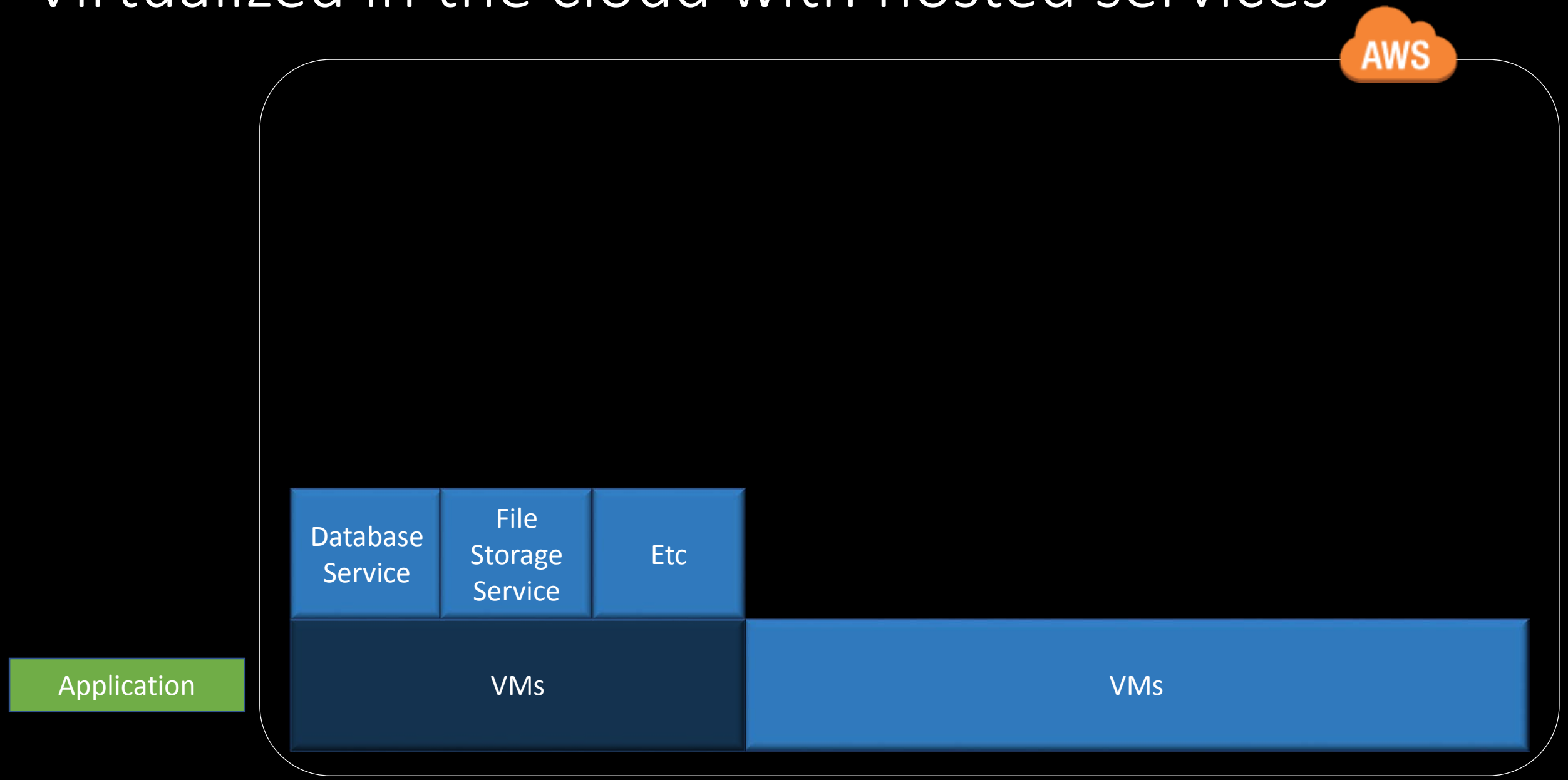
Bare metal servers

Virtualized in the cloud (IaaS)

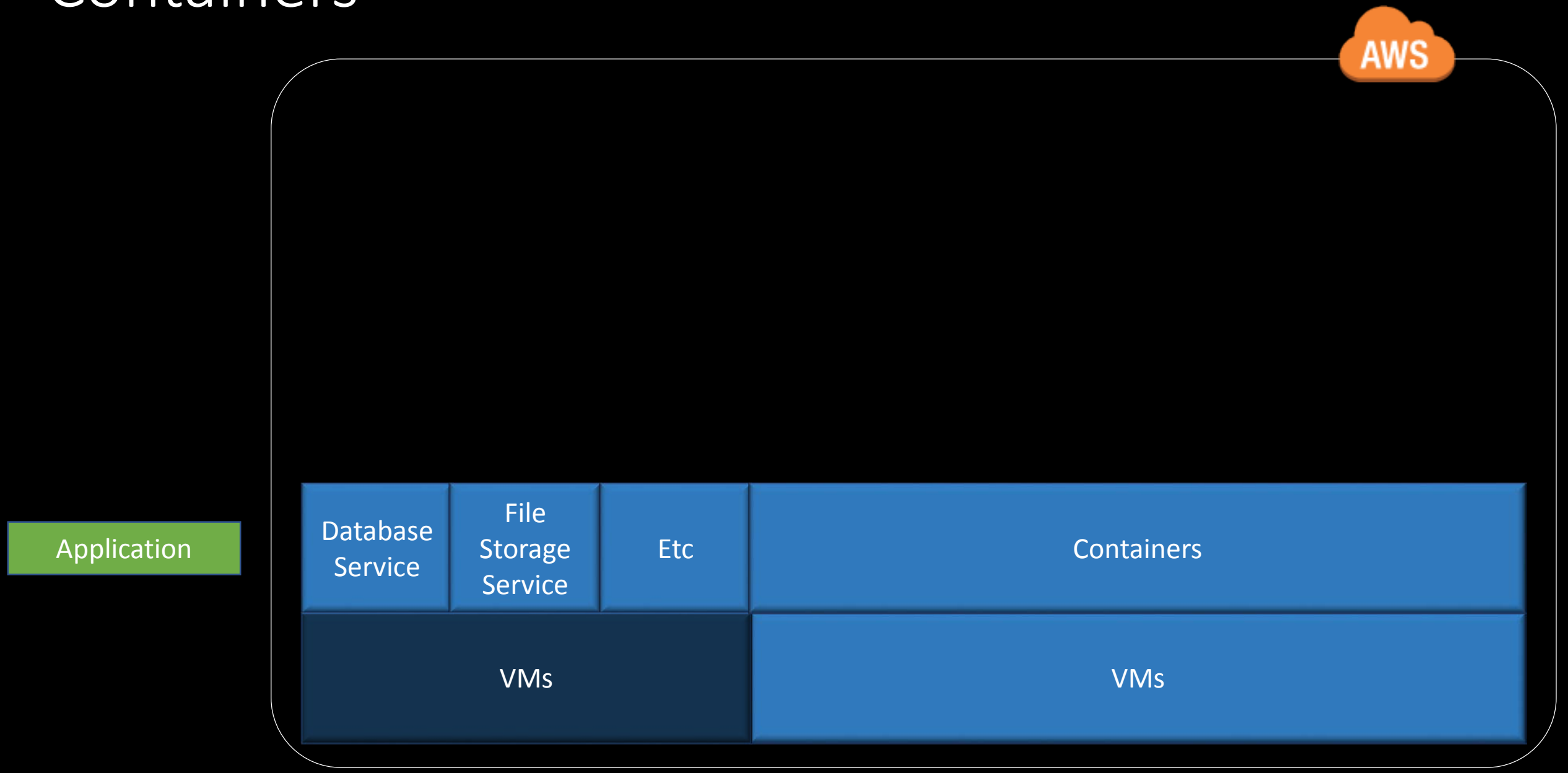
Application

VMs

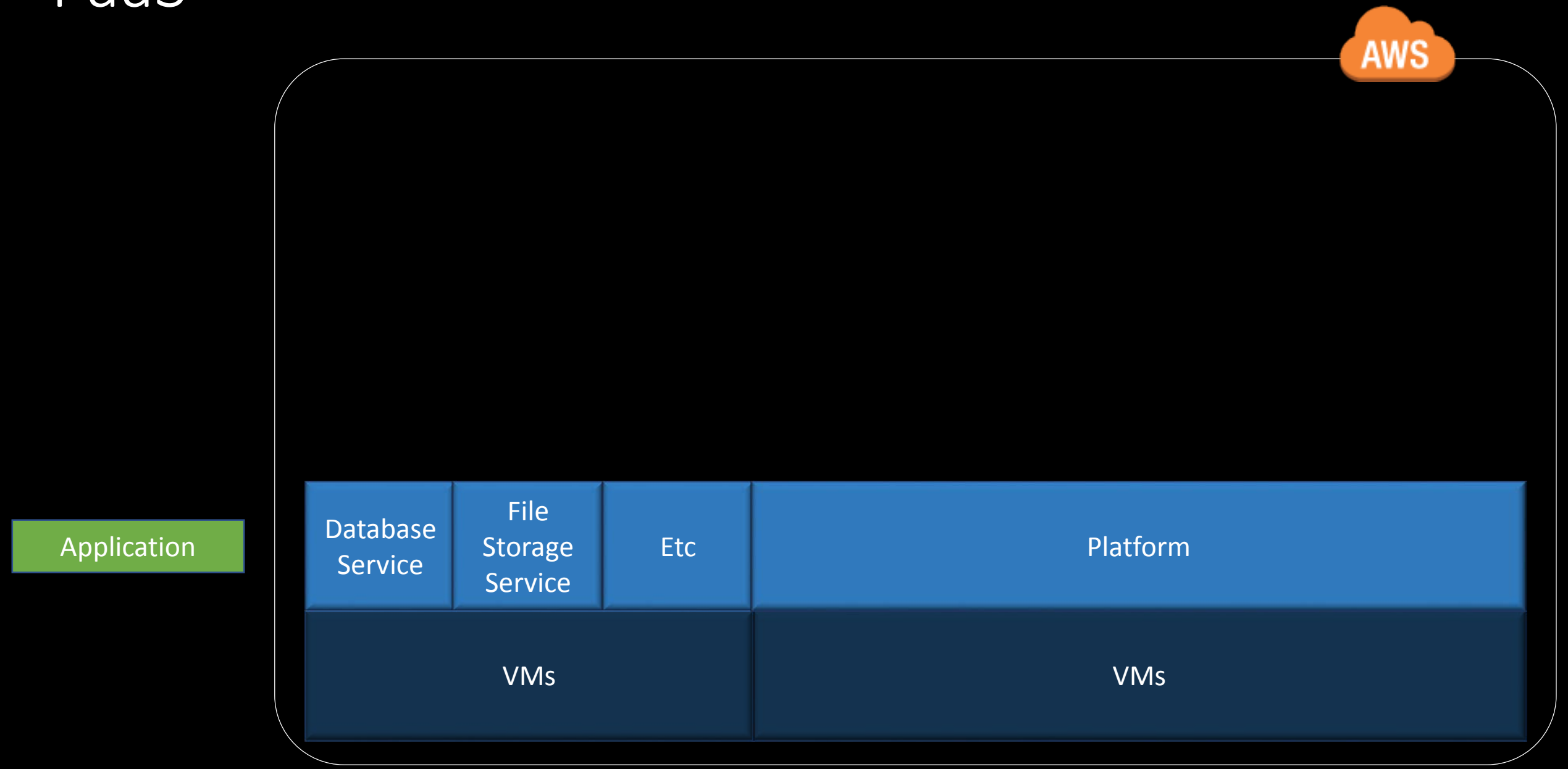
Virtualized in the cloud with hosted services



Containers


AWS
Application
Database Service
File Storage Service
Etc
Containers
VMs
VMs

PaaS


AWS
Application
Database Service
File Storage Service
Etc
Platform
VMs
VMs

Serverless


AWS
Application
Database Service
File Storage Service
Etc
Functions
VMs
VMs

Functions as a Service

+

Backend as a Service

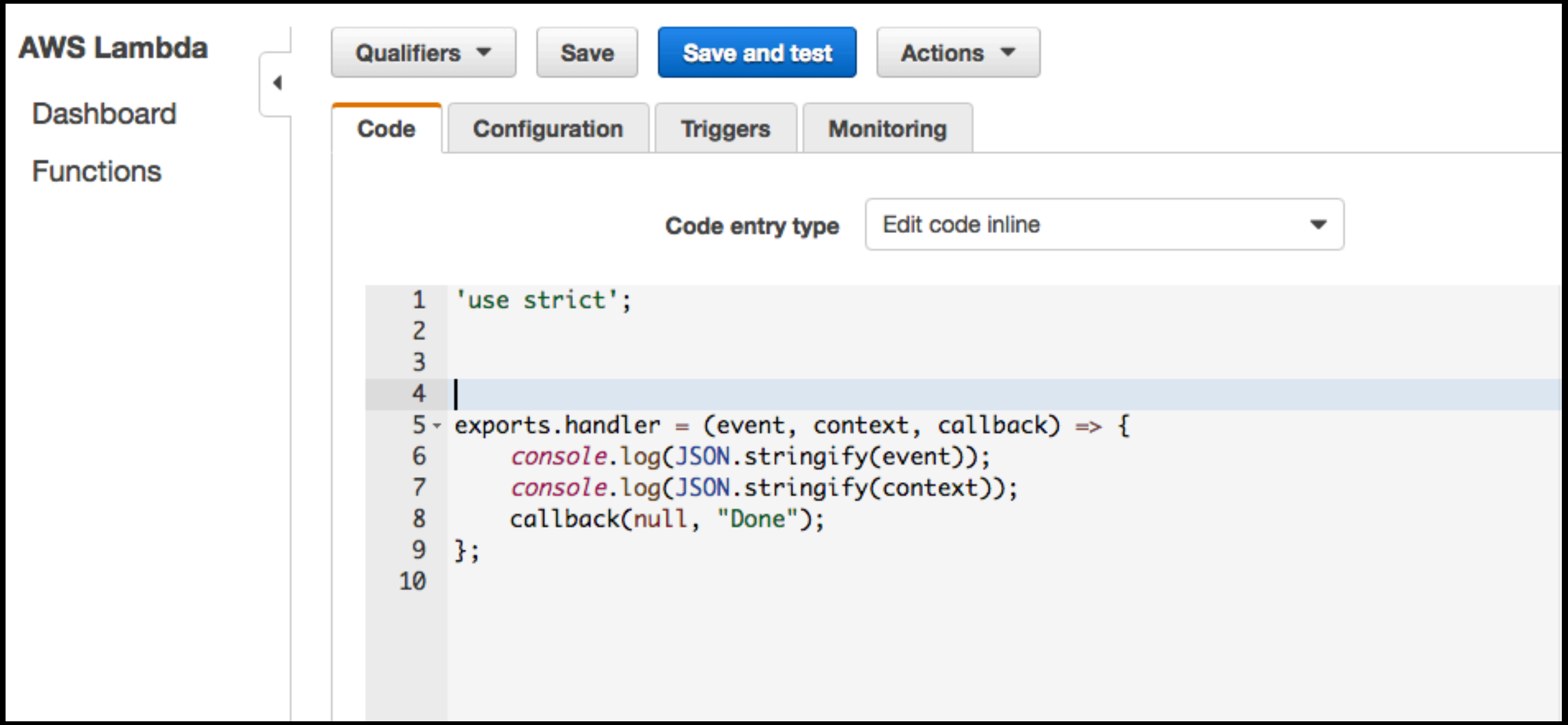
Functions as a Service

Functions as a Service

- User defined functions that run inside a container
- Shared nothing
- Can be spun up on demand
- Can be spun down when not in use
- Scales transparently
- Can be triggered based on various cloud events.
- Biggest disadvantage is startup latency for the container

Examples

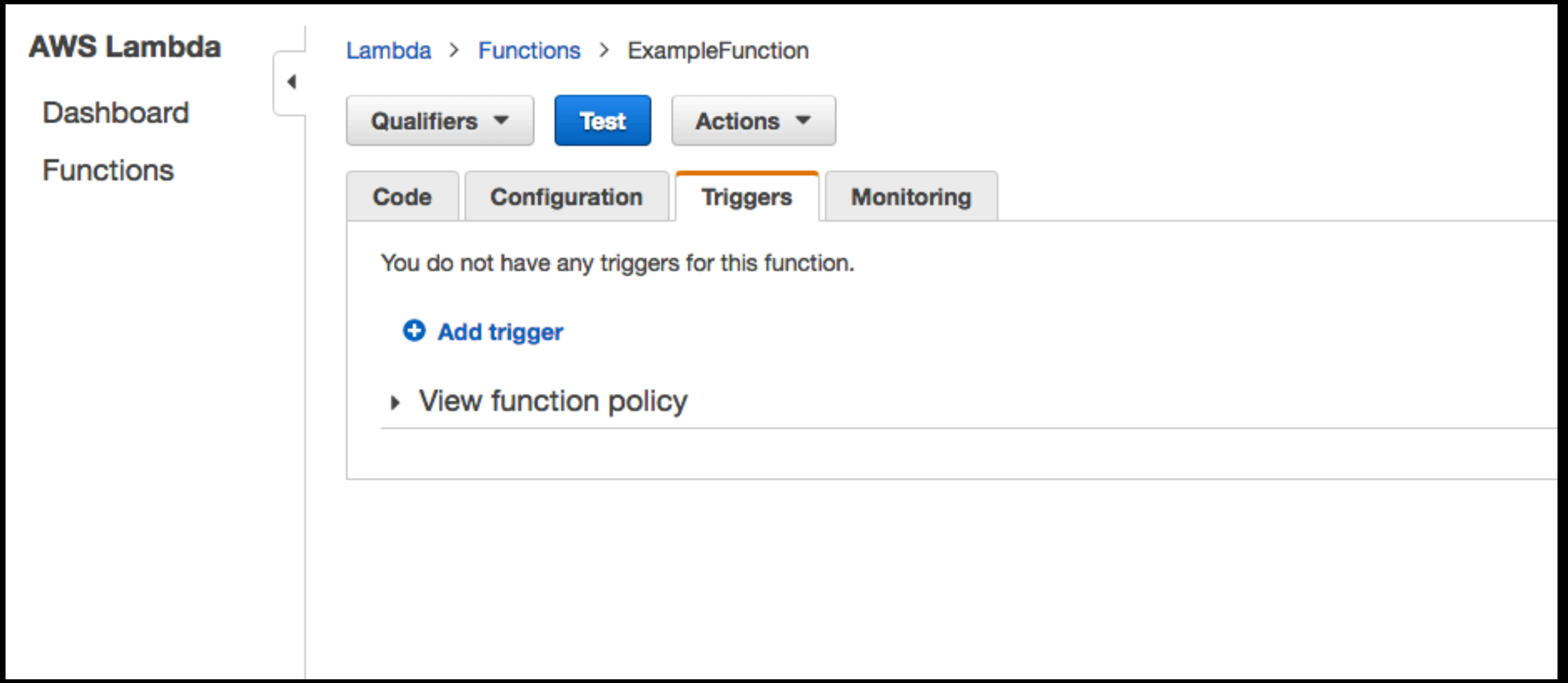
- AWS Lambda
- Google Cloud Functions
- Azure Functions
- IBM Openwhisk
- Others

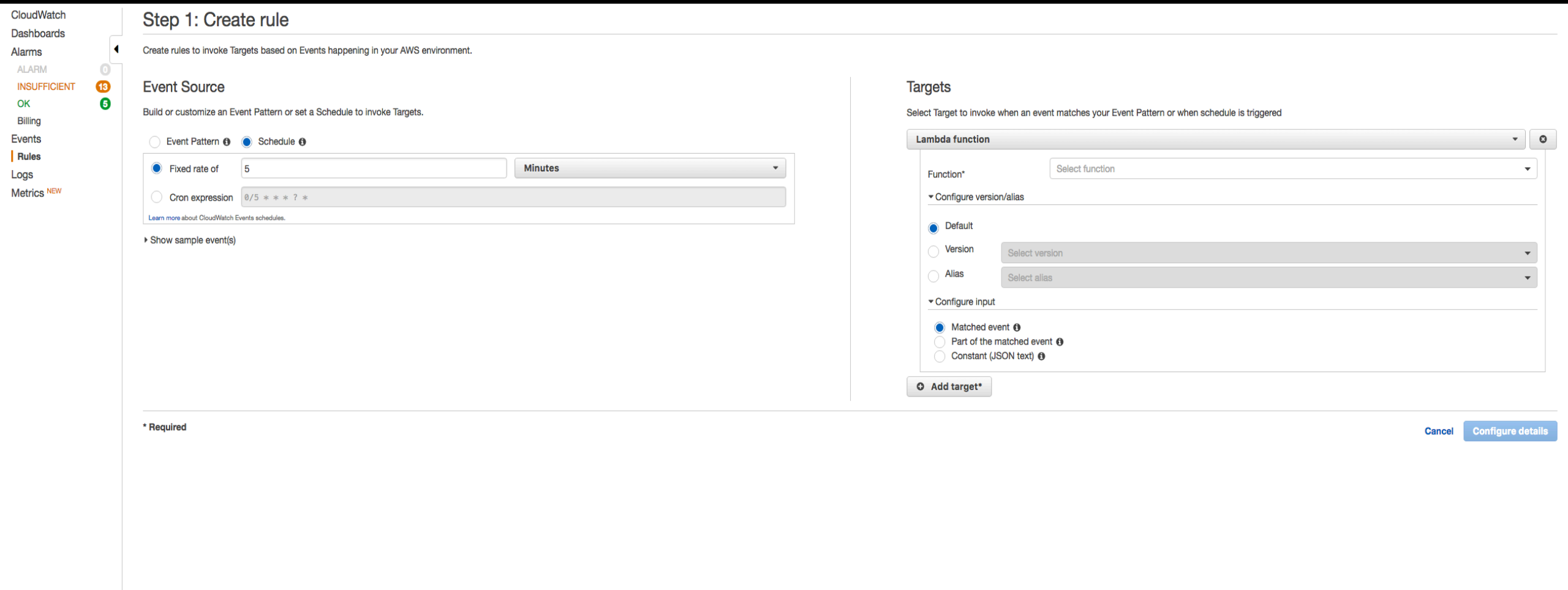

AWS Lambda
Dashboard
Functions
Qualifiers
Save
Save and test
Actions
Code
Configuration
Triggers
Monitoring
Code entry type
Edit code inline
'use strict';
exports.handler = (event, context, callback) => {
console.log(JSON.stringify(event));
console.log(JSON.stringify(context));
callback(null, "Done");
};

AWS Lambda

- RAM/CPU minimum can be set per function.
- Can take a few seconds to initially start, but containers can be reused (automatically if there is one available).
- Invisible limit on containers per account (can be raised)
- Logs are placed in Cloudwatch, according to the function name (log group), and the container (log stream).
- Have access to OS, with scratch HD space (do not rely on).
- Are given an IAM role to execute under.
- Billed based on hardware specified, and execution time.

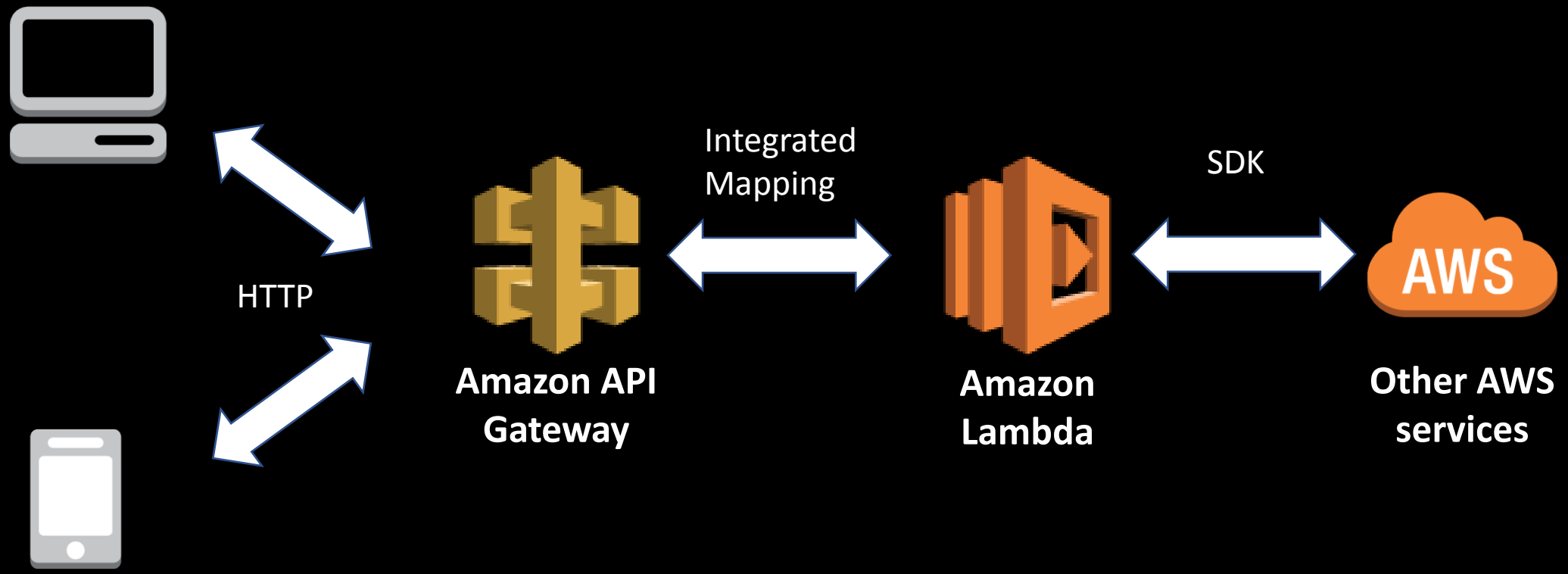
Lambdas are triggered in response to events. These events can be from other AWS services, or via a schedule set in Cloudwatch.


AWS Lambda
Dashboard
Functions
Lambda > Functions > ExampleFunction
Qualifiers
Test
Actions
Code
Configuration
Triggers
Monitoring
You do not have any triggers for this function.
Add trigger
View function policy


CloudWatch
Dashboards
Alarms
ALARM 0
INSUFFICIENT 13
OK 5
Billing
Events
Rules
Logs
Metrics NEW
Step 1: Create rule
Create rules to invoke Targets based on Events happening in your AWS environment.
Event Source
Build or customize an Event Pattern or set a Schedule to invoke Targets.
Event Pattern
Schedule
Fixed rate of
5
Minutes
Cron expression
0/5 * * * ? *
Learn more about CloudWatch Events schedules.
Show sample event(s)
Targets
Select Target to invoke when an event matches your Event Pattern or when schedule is triggered
Lambda function
Function*
Select function
Configure version/alias
Default
Version
Select version
Alias
Select alias
Configure input
Matched event
Part of the matched event
Constant (JSON text)
Add target*
* Required
Cancel
Configure details

HTTP termination in AWS is handled by the API Gateway; this can also trigger a lambda.

In AWS


HTTP
Integrated
Mapping
SDK
AWS
Amazon API
Gateway
Amazon
Lambda
Other AWS
services

You can also invoke lambdas directly with the SDK (from another lambda), as well as execute SWF workflows or Lambda Step Functions.

What happens when Lambdas fail?

Lambdas behave differently depending on whether they're called synchronously, asynchronously, or from a streaming service.

Lambdas called **synchronously** will simply return a failure.

Lambdas called **asynchronously** (from other AWS services) will automatically be queued up and retried a couple times and can be sent to a dead letter queue if retries are used up.

Lambdas called from a **streaming service** will be retried, in order, until success or expiration.

So...

With functions as a service, you have the ability to execute an arbitrary blob of code in response to an event, in your cloud environment.

A key use case of this is executing them in response to a REST API call.

But nothing is persisted, and the containers can only do work between being called and responding (i.e., during the invocation).

Backend as a Service

First up: State Management

Persisting state means distributed state

S3

Pros

- Extremely Cheap
- No provisioning required
- Trivial to replicate across regions
- Version controlled
- Very durable, with SLA
- Can trigger off of updates

Cons

- File system-like (no queries)
- Eventually consistent

DynamoDB

Pros

- Infinitely scalable
- Queryable
- Can trigger off of updates.

Cons

- Expensive
- Requires explicit throughput provisioning
- NoSQL model dictates denormalization and all that entails
- Eventually consistent

Dynamo also has some interesting caveats in how it shards and distributes load.

RDS

Pros

- ACID transactions
- Queryable
- Familiar

Cons

- Provisions according to hardware; downtime to scale up.
- Limited ways to scale out (read replicas).
- Questionable distribution story.
- Requires you managing the DB
- Should be in a VPC, which has some limitations you need to be aware of.
- Complexity when it comes to connection handling.

RDS should be scaled to have the same number of connections as lambda concurrent executions, and ENI’s.

Lambdas should have one connection per container.

Note that startup latency for lambdas inside of a VPC is increased.

Be wary of disk and memory persistence between function invocations.

So now we can build CRUD-like REST APIs. But what about processing outside of a request/response?

We already talked about triggering events; that’s one way.

SQS

Default SQS queues don’t preserve message order. But they’re probably what you want if you need a queue for later processing.

SQS also provides FIFO queues, but extremely limited availability.

Kinesis

Kinesis requires provisioning shards, but allows for massive ingests of data that you can specify be sent to lambda in batches.

So we understand a few ways to process outside of the request/response. How do we alert users?

- SES can be used to easily send emails.
- SNS allows for pushes via SMS or HTTP.
- The AWS IoT service allows for pushes to browsers (https://github.com/lostcolony/examples/blob/master/deviceless_ws.js)
- But for browsers, consider polling

So build our REST API's, have some background processing, alert people accordingly… what about access control?

IAM

Amazon provides Cognito for Identity management. Out of the box it supports Facebook, Amazon, and Google accounts, as well as a generic user signup flow, but you can use a lambda for custom authentication.

Authorization happens through AWS policies. These are simply statements of what resources are allowed vs denied, that are applied to an identity. You need to understand this.

Amazon | Facebook | Google

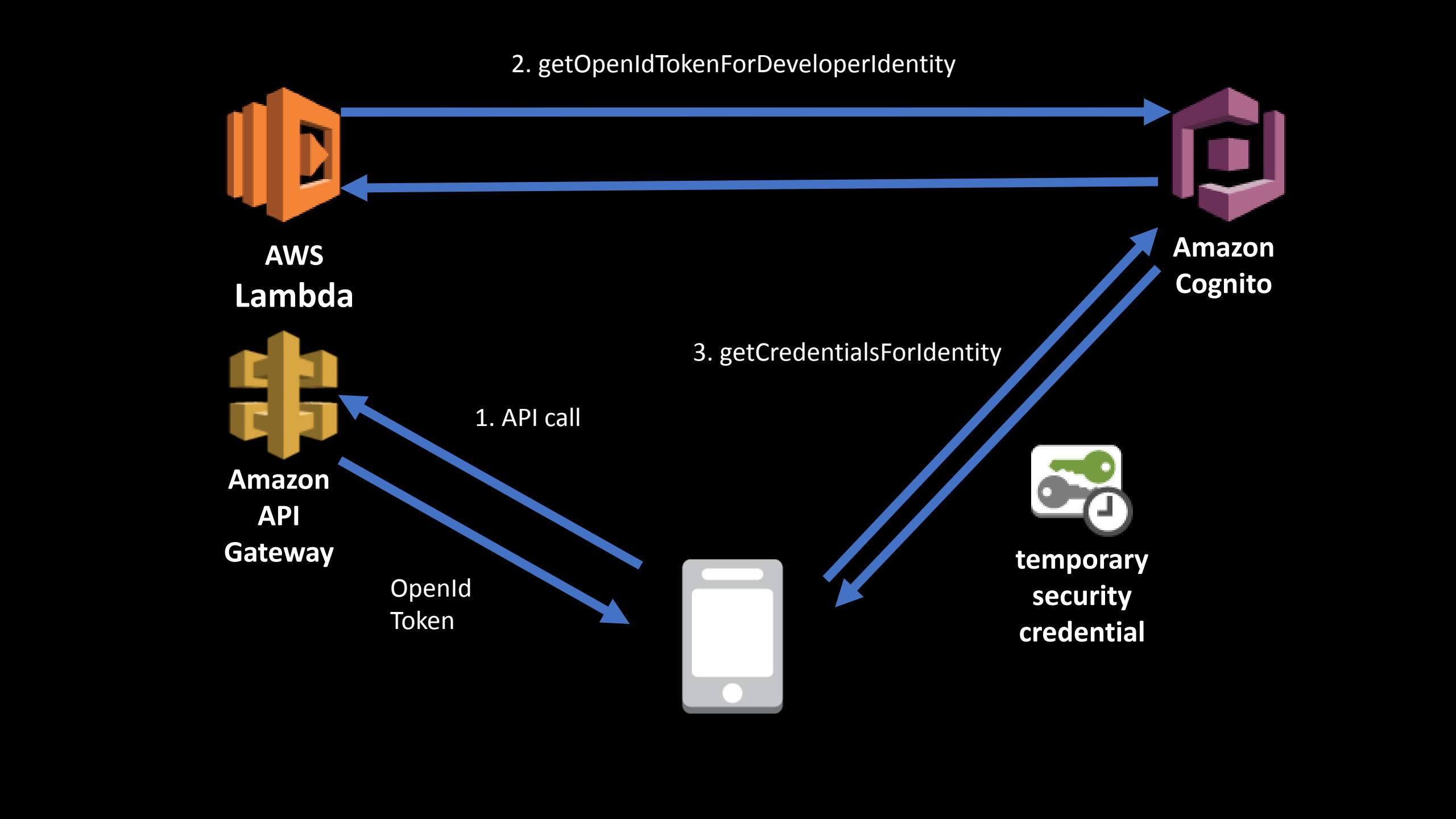
1. Create a federated ID bucket that allows for the appropriate service to log into it. Assign the appropriate role.
2. Log the user in to the service using the relevant service's API, to get an openID token.
3. Call getCredentialsForIdentity specifying the appropriate login provider.

Generic User Pool

1. Create a user pool and app id.
2. Create a federated ID bucket that allows for the generated pool and app id to log into it. Assign the appropriate role.
3. Log the user in via authenticateUser, to get back a JWT.
4. Use the JWT in the aws-sdk to create a new CognitoIdentityCredentials object (or call getCredentialsForIdentity)

Custom Identity Provider

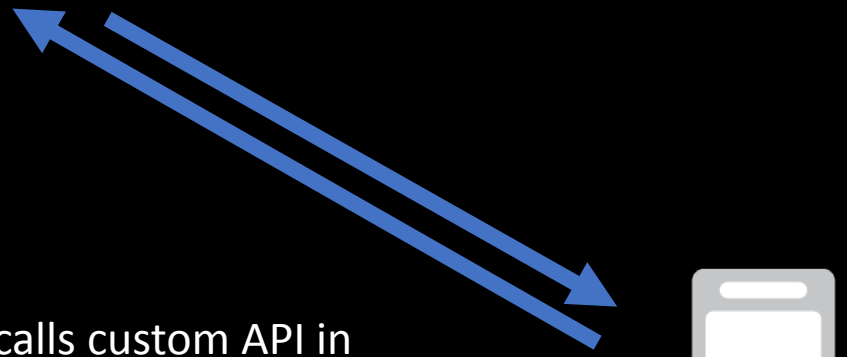
1. Create an endpoint that validates user credentials appropriately.
2. On successful validation, create a token with getOpenIdTokenForDeveloperIdentity. Return this to the client.
3. On the client, call getCredentialsForIdentity. Use the login provider “cognito-identity.amazonaws.com”

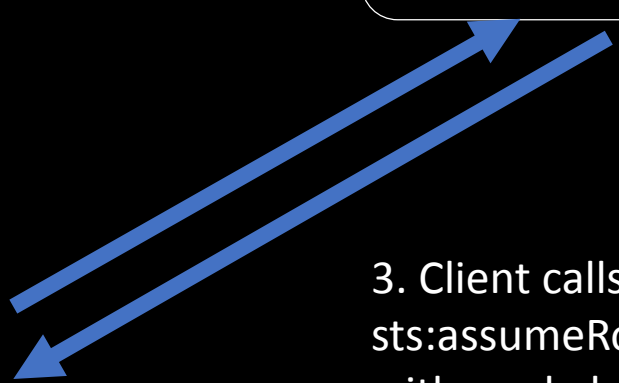

2. getOpenIdTokenForDeveloperIdentity
AWS
Lambda
Amazon
Cognito
3. getCredentialsForIdentity
1. API call
Amazon
API
Gateway
OpenId
Token
temporary
security
credential

You can also call sts:assumeRole* variants to generate temporary credentials, and to further restrict policies (though this requires being done on the backend)

You can set up identity pools, federated identity pools, and roles, in separate accounts, and still use them in a single app.

1. Set up a role in your application account that has a trusted principal of the Cognito bucket in your authentication account.





2. Client calls custom API in authentication account for OpenID token

3. Client calls sts:assumeRoleWithWebIdentity with a role belonging in a second account. Get credentials for that account.

Every AWS service can be accessed with these credentials using AWS Signature Version 4

You can restrict access to your APIs this way, too, but not via the SDK. Use a third party signing library rather than reimplement Sig4

AWS does not currently have user controlled rate limiting to prevent malicious users from DDoSing you.

However, there is a Web Application Firewall.

There is also AWS Shield. It’s their solution in this space; basic functionality is automatically enabled.

Most usage billed services AWS provides include access limits. Production workloads may need these raised.

Some can be automatically raised via code. Most can’t.

For the ones that can, you can actually trigger lambdas from Cloudwatch alarms (which I'll get to). Use this to raise a service's limits

For those that can’t be automatically addressed…you need to be able to monitor your application.

Monitoring and Alerting

Unsurprisingly, with Serverless, both logging and alerting change.

Lambda logs go directly into Cloudwatch, but they are not easily followable, as they’re per function container.

Happily, Cloudwatch has pretty decent search and metrics tools.

You can set up alarms for these metrics, to post to SNS (and then send out SMS messages, or emails to alert people).

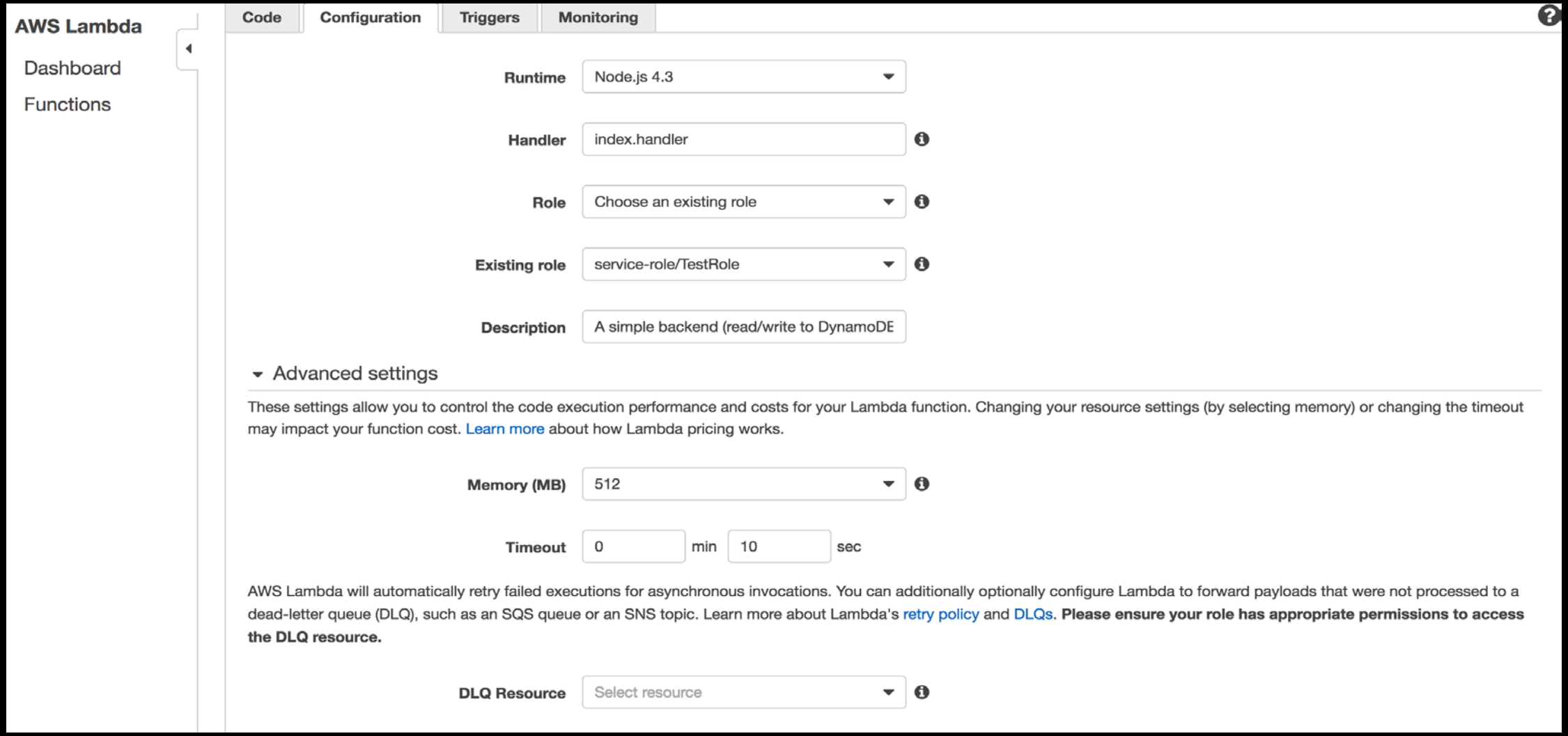
Do not use Cloudwatch for application metrics, however. Only the provided, technical ones.

Now that we've talked about the backend, what happens if an asynchronous lambda fails too many times?

Well, they’re retried (as mentioned before), but otherwise nothing by default.

But you can set up dead letter queues.

Asynchronous lambdas can drop a failing event onto an SNS topic or SQS queue.


AWS Lambda
Dashboard
Functions
Code
Configuration
Triggers
Monitoring
Runtime
Node.js 4.3
Handler
index.handler
Role
Choose an existing role
Existing role
service-role/TestRole
Description
A simple backend (read/write to DynamoDE
Advanced settings
These settings allow you to control the code execution performance and costs for your Lambda function. Changing your resource settings (by selecting memory) or changing the timeout may impact your function cost. Learn more about how Lambda pricing works.
Memory (MB)
512
Timeout
0
min
10
sec
AWS Lambda will automatically retry failed executions for asynchronous invocations. You can additionally optionally configure Lambda to forward payloads that were not processed to a dead-letter queue (DLQ), such as an SQS queue or an SNS topic. Learn more about Lambda's retry policy and DLQs. Please ensure your role has appropriate permissions to access the DLQ resource.
DLQ Resource
Select resource

Cloudwatch logs can also be copied to S3 buckets, or pipe events into a Kinesis stream for processing in other tools, code, etc.

So we can build an app. We can monitor it. But what about the actual lifecycle of integration and deployment?

Let’s start with our lambdas.

Serverless - https://serverless.com/

Apex - http://apex.run/

Sparta - http://gosparta.io/

Zappa - https://github.com/Miserlou/Zappa

Chalice - https://github.com/awslabs/chalice

Because these are CLI tools, you can leverage existing CI/CD tools fairly easily, but direct plugins are few.

Serverless (the framework), also bundles in Cloudformations. I can’t speak to the others.

If you maintain config in separate files, consider uploading them to an encrypted S3 bucket.

You can, instead, use environment variables as part of your lambdas. Serverless (the framework) also includes support for this.

Console Home

Dashboard

Functions

Qualifiers | Test | Actions

Code | Configuration | Triggers | Monitoring

Code entry type: Edit code inline

```
1  exports.handler = (event, context, callback) => {
2      // TODO implement
3      console.log(process.env['Ab'])
4      callback(null, 'Hello from Lambda');
5  };
```

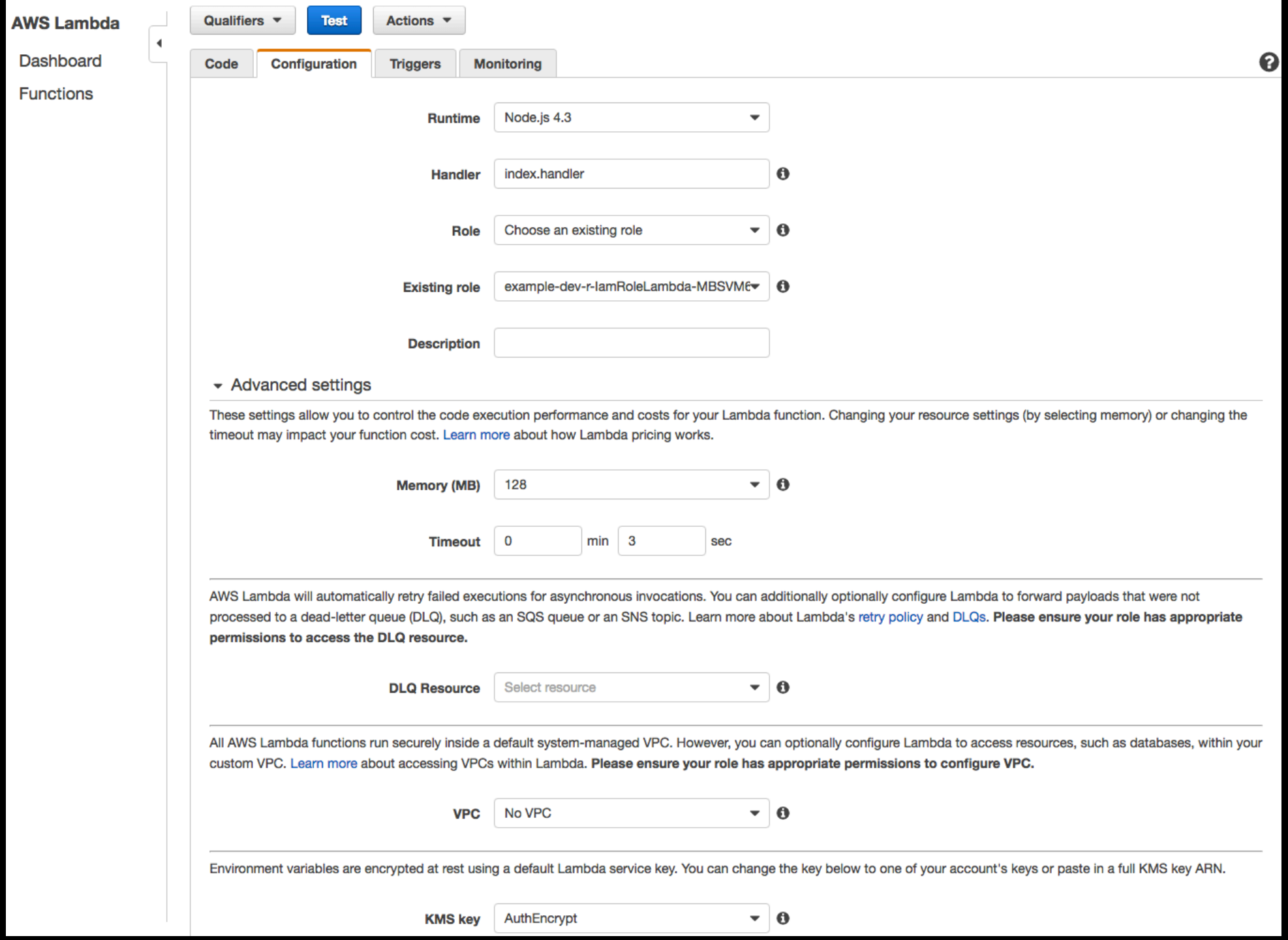
You can define Environment Variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the need to change function code. Learn more. For storing sensitive information, we recommend encrypting values using KMS and the console's encryption helpers.

Enable encryption helpers

Encryption key: AuthEncrypt

Environment variables

Ab	bafad	Encrypt	Code
Key	Value	Encrypt	Code


AWS Lambda
Dashboard
Functions
Qualifiers
Test
Actions
Code
Configuration
Triggers
Monitoring
Runtime
Node.js 4.3
Handler
index.handler
Role
Choose an existing role
Existing role
example-dev-r-lamRoleLambda-MBSVM6
Description
Advanced settings
These settings allow you to control the code execution performance and costs for your Lambda function. Changing your resource settings (by selecting memory) or changing the timeout may impact your function cost. Learn more about how Lambda pricing works.
Memory (MB)
128
Timeout
0
min
3
sec
AWS Lambda will automatically retry failed executions for asynchronous invocations. You can additionally optionally configure Lambda to forward payloads that were not processed to a dead-letter queue (DLQ), such as an SQS queue or an SNS topic. Learn more about Lambda's retry policy and DLQs. Please ensure your role has appropriate permissions to access the DLQ resource.
DLQ Resource
Select resource
All AWS Lambda functions run securely inside a default system-managed VPC. However, you can optionally configure Lambda to access resources, such as databases, within your custom VPC. Learn more about accessing VPCs within Lambda. Please ensure your role has appropriate permissions to configure VPC.
VPC
No VPC
Environment variables are encrypted at rest using a default Lambda service key. You can change the key below to one of your account's keys or paste in a full KMS key ARN.
KMS key
AuthEncrypt

Maintain separate environments for dev/stage/prod/etc.

Consider separate environments for separate applications as well.

We’ve had success allowing devs access to a dev AWS account, which has AWS Config enabled.

We’ve also found integration tests, in an actual, deployed environment, to be the most useful.

So, we can design a solution, everything can run securely, we'll know when things fail…what else?

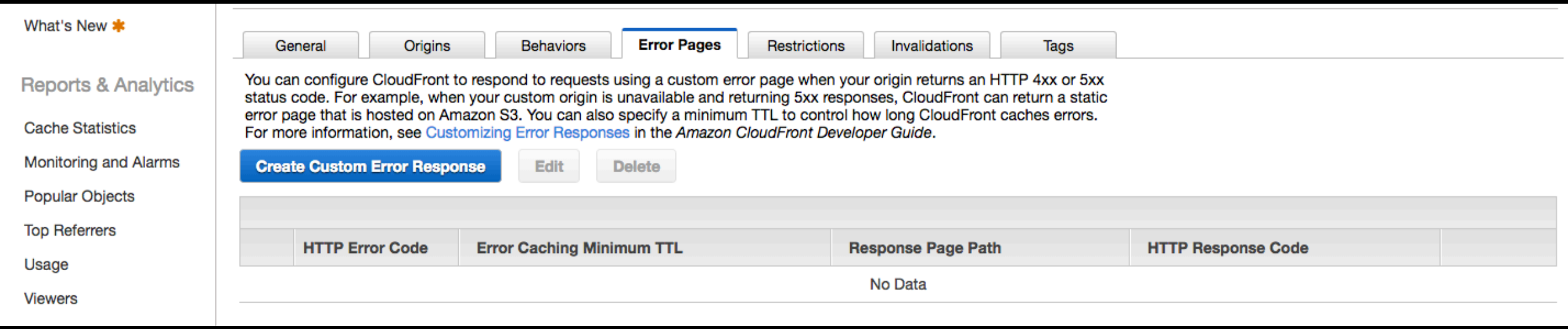
Latency can be a problem…but not always. User experience matters, not latency numbers.

You can pre-warm functions by creating a scheduled event to invoke them periodically.

Use Cloudfront for caching.

And more TLS options.
And a layer of indirection.
And for AWS Shield.

Be careful of caching error codes.


What's New
Reports & Analytics
Cache Statistics
Monitoring and Alarms
Popular Objects
Top Referrers
Usage
Viewers
General
Origins
Behaviors
Error Pages
Restrictions
Invalidations
Tags
You can configure CloudFront to respond to requests using a custom error page when your origin returns an HTTP 4xx or 5xx status code. For example, when your custom origin is unavailable and returning 5xx responses, CloudFront can return a static error page that is hosted on Amazon S3. You can also specify a minimum TTL to control how long CloudFront caches errors. For more information, see Customizing Error Responses in the Amazon CloudFront Developer Guide.
Create Custom Error Response
Edit
Delete
HTTP Error Code
Error Caching Minimum TTL
Response Page Path
HTTP Response Code
No Data

The API Gateway also has a separate caching mechanism. It’s not really worth it.

Rethink your API design.

Use Route53 + Certificate Manager liberally.

The API Gateway allows HTTP passthrough. If you want to move to Serverless, this is how to start.

Write your lambdas as libraries, and isolate the handler code.

```
1   const my_lib = require('./src/my_lib');
2
3   module.exports.handler = function(event, context, cb) {
4       if(event.isPrewarm) {
5           return cb(); //This is to handle pre-warming
6       } else {
7           //Do whatever here to grab context and event values before passing it into
8           //your library, to create a layer of abstraction between the two things.
9           return my_lib(event, cb);
10      }
11
12  }
```

For testing, you can easily unit test your code this way. It’s just a library.

For integration tests, test it in an AWS environment itself. If you're doing things 'properly', this is easy.

Pros

- Costs are more visible
- Complexity is hidden
- Stateless
- Scales trivially
- Secure

Cons

- Costs are more visible
- Complexity is hidden.
- Stateless can seem harder to reason about
- Latency issues (cold start)
- Immature tooling
- Vendor lock-in (to some degree)

Conclusion

For a hands on demo, see David Aktary’s talk at 1:00

Questions