Introduction to Serverless Computing with AWS Lambda

CONTINO | Serverless Computing

Serverless computing, in this context also known as the Function-as-a-Service (FaaS) model, has dramatically risen in popularity since the introduction of the Lambda[1] platform by AWS in 2014. Although not the first cloud services provider to offer such a platform, the scale and scope of the AWS offering meant that, for many, this was when the serverless paradigm hit the big-time.

The "serverless" name clearly does not suggest that no compute resources underpin the services, but instead that, to the consumer of the service, there are no virtual machines exposed. This means: no provisioning of infrastructure and no ongoing management of compute resources. The execution of users' functions is abstracted away from the machines on which it takes place, and as such it can be seen to offer a "pure" execution platform.

Although many cloud services providers now offer serverless execution platforms, this paper focuses on AWS Lambda. As the first major player in the market it has garnered significant traction, and along with its flexibility, scalability and comprehensive integration with other AWS services, this means that it has quickly become the leading platform in the field.

Scope

This paper focuses primarily on the various use cases for AWS Lambda, as well as other tightly integrated AWS services, most notably AWS API gateway and selected AWS database services such as DynamoDB. Although AWS publicise some of the technical underpinnings of their Lambda service, and some internal implementation details have been discovered by users, many reports are speculative, and because of this, Lambda internals are outside the scope of this document.

An upfront high-level understanding of AWS platform concepts may be helpful, but short explanations or links will be given where possible.

Advantages of the Serverless Model

Abstraction of Compute Resources

Clearly the first advantage of the serverless model is hinted at in the name – with a serverless approach, there is no infrastructure to manage. Because the underlying machine (physical and virtual) is abstracted away, traditional tasks such as infrastructure provisioning, configuration management and patching cycles cease to be relevant. Because much of a typical IT department's budget goes on these non-differentiating activities, moving to a serverless model can allow more resources to be dedicated to honing application-level code, allowing organizations to focus on these true differentiating factors.

Before the advent of the serverless model, many attempts were made to provide these kinds of benefits by way of Platform-as-a-Service (PaaS) offerings – while providing some of the above selling points, these platforms did not quite go far enough, with users still being required to manage aspects of the underlying servers and the configuration of the PaaS itself.

True "On-Demand"

With serverless, users only pay for compute resources that are usefully consumed, reducing waste and increasing efficiency. Unlike most previous virtualised compute offerings – which typically were billed in hourly (or larger) quanta – serverless platforms can be billed at a very granular level, with AWS Lambda being down to the 100-millisecond level. This means that users do not waste money on resources that are idle or underutilised. On top of this, consider an EC2 instance that executes a short batch task each hour and then terminates. In EC2 this would incur 24/7-equivalent usage charges, whereas if implemented in Lambda, only the task's execution duration would be billable.

With the shift to a "zero-use = zero-cost" model, many more advantages are evident:

- Growth in cost is linear, which provides for predictable and straightforward forecasting
- Pace of innovation can be higher, as experiments can be performed at extremely low cost
- Realistic application load-testing can be performed without the need to maintain production-sized environments at a high cost
- Production and non-production environments can be configured and sized identically without dramatically impacting spend, which aids representative testing and thus reduces the likelihood of production issues

Near-Limitless Compute

Lambda users are not constrained by individually provisioned compute resources - this means that workloads can grow to cover vast datasets or serve extremely high loads. Inherent to the platform is the facility to execute multiple invocations of functions concurrently, which means that users have enormous compute power at their fingertips. A side benefit of this is that, because the infrastructure provider can more effectively manage multi-tenancy on the underlying hardware, prices are pushed down and savings passed on to the consumer.

Rapidly Scalable

Because of the lack of infrastructure spin-up time - other than a very slight delay for initial cold-start, detailed later in this document - the ability to rapidly grow and shrink a platform to meet demand is greatly improved versus creating new, dedicated compute resources.

Consider the example of a web back-end that sees huge traffic spikes immediately following promotional TV advertisements. With traditional IaaS platforms typically offering new instance time-to-service durations measured in minutes, a coordinated pre-emptive scaling strategy would be required to satisfactorily meet these traffic bursts, with the need to scale up in advance of the broadcast of each advertisement.

With a platform implemented using a serverless model able to scale instantly from tens to many thousands of concurrent invocations, this pre-emptive scaling becomes unnecessary, and additionally allows the platform to scale to meet traffic that is spiky and truly unpredictable, such as that generated by breaking news events, for example.

No Long-Term OS State Management

Typically, with traditional long-lived compute resources, ongoing management of state can be a challenge. Techniques such as configuration management using Chef, Puppet or similar allow automation to be used to perform some of these tasks, but over time entropy can creep in and cause platform configuration to drift from a known-good state.

The move towards immutable infrastructure – in which no changes are made to compute resources once deployed – has come about to address some of these challenges, but serverless platforms take this one step further, with the user having no access to the underlying machine[2] or ability to make manual modifications that will persist.

This means that no easily-forgotten or error-prone manual "fixes" are possible, with all instances of a given function executing in identical, known-good environments.

Security

Often, in self-managed environments, operator error is the underlying reason for a platform being compromised. At the network level, a mismanaged firewall or overly lax ACL can open up an environment to malicious traffic, and unpatched operating systems have long since been easy routes in for attackers; this is not to mention zero-day or undisclosed vulnerabilities that can be exploited by criminals or agents of industrial or national espionage.

As per the Shared Responsibility Model[3], because AWS are responsible for the security of the underlying Lambda execution environment and are able to devote significant resources and expertise to this, developers can benefit from the managed security and are free to concentrate on application-level security. This is not to say that security is no longer a concern – clearly an application vulnerability or cross-site-scripting security hole will still be a problem – but more that the attack surface is reduced.

Serverless Model Watchpoints

The following sections detail some of the Lambda watchpoints that may surprise users new to the platform. Please note that, while the information below is correct at the time of writing, future service enhancements by AWS could mean that some of the limits described below may change over time. We recommend checking the AWS Lambda limits[4] page for the most up-to-date information.

Statelessness

Lambda functions are inherently stateless - while it is possible that a particular execution sandbox may continue to execute on the same underlying compute resource, this is by no means guaranteed and must not be relied upon.

Typical serverless architectures make use of in-memory stores such as Redis, SQL/NoSQL databases or message queues to hold state, with data to be processed read from or written to S3. Concurrency can introduce significant complexity into a platform and care must be taken to manage this with appropriate controls, such as transactional logic, locking and use of platforms that support atomic operations.

Limited Execution Duration

Lambda functions are limited in terms of their execution time to a maximum of 300 seconds, i.e. five minutes. This means that traditional "server" applications may not be suitable for execution in Lambda without re-architecture.

Limited Native Runtime Support

At the time of writing, Lambda supports the following run times:

- Node.js – v4.3.2
- Java – Java 8
- Python – Python 2.7
- .NET Core – .NET Core 1.0.1 (C#)

Other run times - such as functions written in Go, for example, or any other statically linked binaries - can be used by first executing a "shim" in a native runtime that subsequently executes the target function, but this is not officially supported by AWS.

Artefact Size

The maximum size of a compressed artefact that can be uploaded for Lambda execution is 50 MB. This artefact must uncompress to a maximum size of 250 MB, including dependencies - note that depending on the application runtime used it can be possible to download dependencies dynamically at runtime, though this increases external dependencies, and will increase the application start up time and thus the cost per execution.

Cold-start Time

Functions that are executed only rarely will generally see an extended cold-start time during their first periodic invocation, usually up to single-digit seconds. Periodically "pinging" a function, or scheduling an execution, is a technique to ensure that extended cold-start delays do not adversely affect user experience if a real-time response is required.

Concurrent Execution Limits

By default, the number of concurrent invocations of each function is limited to 100 - AWS state that this is done to protect users from potentially excessive Lambda execution costs caused by functions invoked in error by software misconfiguration. This soft limit may adversely affect the compute resources available to a serverless application. The limit on concurrent executions can be raised by AWS Support via the usual channels, and is definitely a recommended step before go-live of a platform of any scale.

Multitenancy

AWS Lambda is inherently multi-tenant - a large compute pool is shared by all users, which is key to the scalability that Lambda can provide. This means that there is potential for crosstalk between Lambda users. Obviously AWS seeks to make this impossible though technological sand-boxing, but this should always be a consideration when this degree of multi-tenancy is in use. Inter-user interference may be a security risk, a performance risk, or both.

Currently there is no way to request or provision a "private" Lambda back-end, which additionally would defeat one of the key points of the serverless execution model: near-limitless compute resource shared between individually billed users. Note that, while Lambda can run "inside" a VPC, the underlying compute pool is still shared.

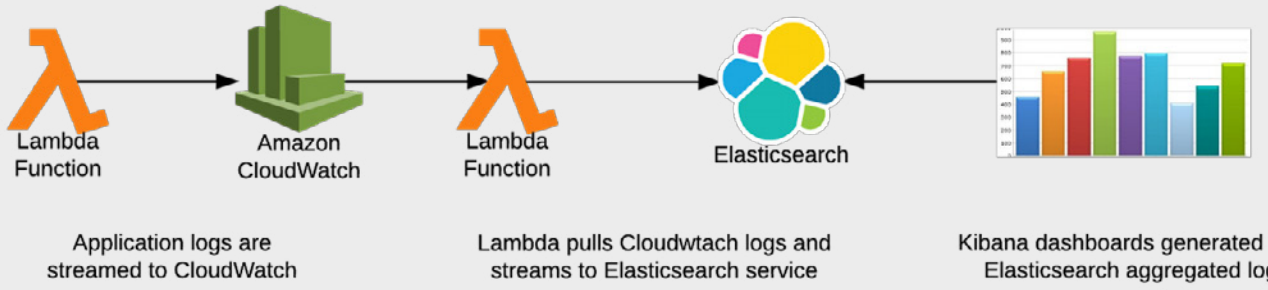
Compliance

For organizations bound by regulatory concerns, Lambda is not currently PCI or HIPAA compliant. This does not forbid its use within environments subject to PCI or HIPAA compliance, however, and means only that Lambda must not be used to process any data (including meta-data) that is subject to this compliance. It may still be used to trigger and/or chain together AWS events throughout a platform, for example, provided that no sensitive data is touched or processed by code running in Lambda.

Use of certain AWS technologies in PCI/HIPAA compliant environments is subject to the encryption of the data at rest within the platform, which at present is not built in to Lambda. The use of EC2 in HIPAA-compliant environments is subject to the instances being created as "dedicated" to the AWS account owner, i.e. the hypervisor must be single-tenant, which does not bode well for future Lambda compliance.

Limited Inbuilt Visibility

Out of the box, Lambda logs are streamed to AWS CloudWatch Logs. Some basic inbuilt log viewing and dashboarding facilities are provided, but for maximum visibility it is recommended to ingest these logs into an ELK or similar stack. This is a platform component that has to exist outside Lambda, although AWS do offer a hosted Elasticsearch service that is dramatically easier to run than the alternative approach of self-managing such a cluster on EC2. A typical simple pipeline may be as follows:



More information on approaches to logging and monitoring appears later in this document.

Lambda Use Cases

The following sections detail some common Lambda use cases.

Web Services with API Gateway

With the release of AWS API Gateway, an elegant and scalable mechanism for making Lambda functions available as web services became available. Although initially somewhat lacking in features, the popularity of API Gateway in conjunction with Lambda has gone from strength to strength, and this is now a common deployment pattern for lightweight and web-scale services alike. A tutorial later in this document describes how to build a simple web application that runs in Lambda fronted by API Gateway.

Batch Data Processing

Increasingly Lambda is being used for batch data processing. The cost model of paying only for required compute is compelling; for batch jobs or workloads that run hourly or more frequently, implementation on EC2 would result in 24/7-equivalent costs. For an hourly batch job that requires significant computation, clearly this could be expensive. With the inbuilt ability to schedule the invocation of Lambda jobs, management overhead is further reduced.

Analytics

Following on from the above, but additionally incorporating streaming data processing, analytics workloads on Lambda are surging in popularity. Being a somewhat recent platform to be employed for analytics, frameworks and tooling are relatively immature when compared to traditional dedicated compute platforms, but the field is developing rapidly given the huge advantages of cost and scale. Dubbed “Serverless Map/Reduce”, several reference architectures and open-source frameworks are available - PyWren[5] is one such implementation, with which one benchmark achieved a peak of 40 TFLOPS, an impressive demonstration of scalable analytics compute capacity.

Event-Driven Processing

Throughout the AWS platform, events such as the arrival of objects in an S3 bucket or notifications from SNS can be used to trigger Lambda functions to process new data or otherwise react to the event. Consider the classic AWS example of an image upload site that needs to generate a thumbnail for each uploaded image - the S3 event generated by the arrival of each image invokes a Lambda function with an IAM role that allows it to retrieve the image, generate a thumbnail and write the thumbnail image to the correct location in S3 from where it can be retrieved.

AWS Environment Automation

In the past, often one or more “automation” instances were employed in a typical AWS account to perform tasks such as EC2 instance out-of-hours shut down, EBS volume snapshots and other housekeeping tasks. With the facility to schedule Lambda function invocations now available, this means that maintaining a running instance (that needs to be managed, patched, etc.) is no longer a requirement. Given an appropriate IAM role, Lambda functions can perform all these tasks at a fraction of the cost.

Log Ingest and Analysis

As described above, logs from Lambda functions are streamed to CloudWatch, but ingest and post-processing to improve searchability and visibility are recommended. As well as this, logs from other sources often need on-the-fly processing. Lambda can be used as part of this pipeline for log traffic in flight - the ability to scale up instantaneously should the number of log-lines/sec dramatically increase can be a lifesaver. During outages or incidents, when the volume of log traffic is often elevated, it’s important to ensure the ingest platform is able to deal with the load, and it’s also during these times that having a reliable logging platform is most important. Given the temporal nature of the data, and reliability of the platform, Lambda can be a good choice for these use cases.

Artefact Build and Test

A relatively recent, emerging use case for Lambda is within build systems - tests can be run with a huge degree of parallelisation, without needing to manage and run multiple build slaves. Obviously care must be taken that the 300-second invocation time limit is not exceeded, but with appropriate decomposition this issue can be avoided.

AWS Step Functions

Introduced at Re:invent 2016, AWS Step Functions[6] build on the underlying functionality of the Lambda platform and provides the user with a mechanism for coordinating and passing control between a number of small supporting Lambda services that jointly comprise a serverless application.

Frameworks to Support Serverless Applications with Lambda

Though serverless is a relatively new area, a number of open source frameworks have been developed to help put structure around how serverless web applications are built, deployed and managed.

Serverless Framework

The Serverless Framework[7] (originally JAWS) was one of the first to be released and as such is one of the more mature frameworks. Serverless is written in node.js and uses AWS CloudFormation to provision AWS resources - this means that the framework deploys the Lambda functions themselves, along with any other CloudFormation-provisionable resources that may be needed to support their execution. Serverless can additionally be used to deploy non-web-applications to Lambda.

Chalice

Produced by AWS Labs, Chalice[8] is a "microframework" to help deploy Python applications to Lambda and route requests via API Gateway. Chalice provides Python decorators that can be used to route requests, but does not offer management of any other AWS services outside of API Gateway.

Zappa

Zappa[9] is a framework for serverless applications written in Python. It works with any WSGI application, so can be used with common Python web application frameworks such as Flask, Bottle and even Django.

Sparta

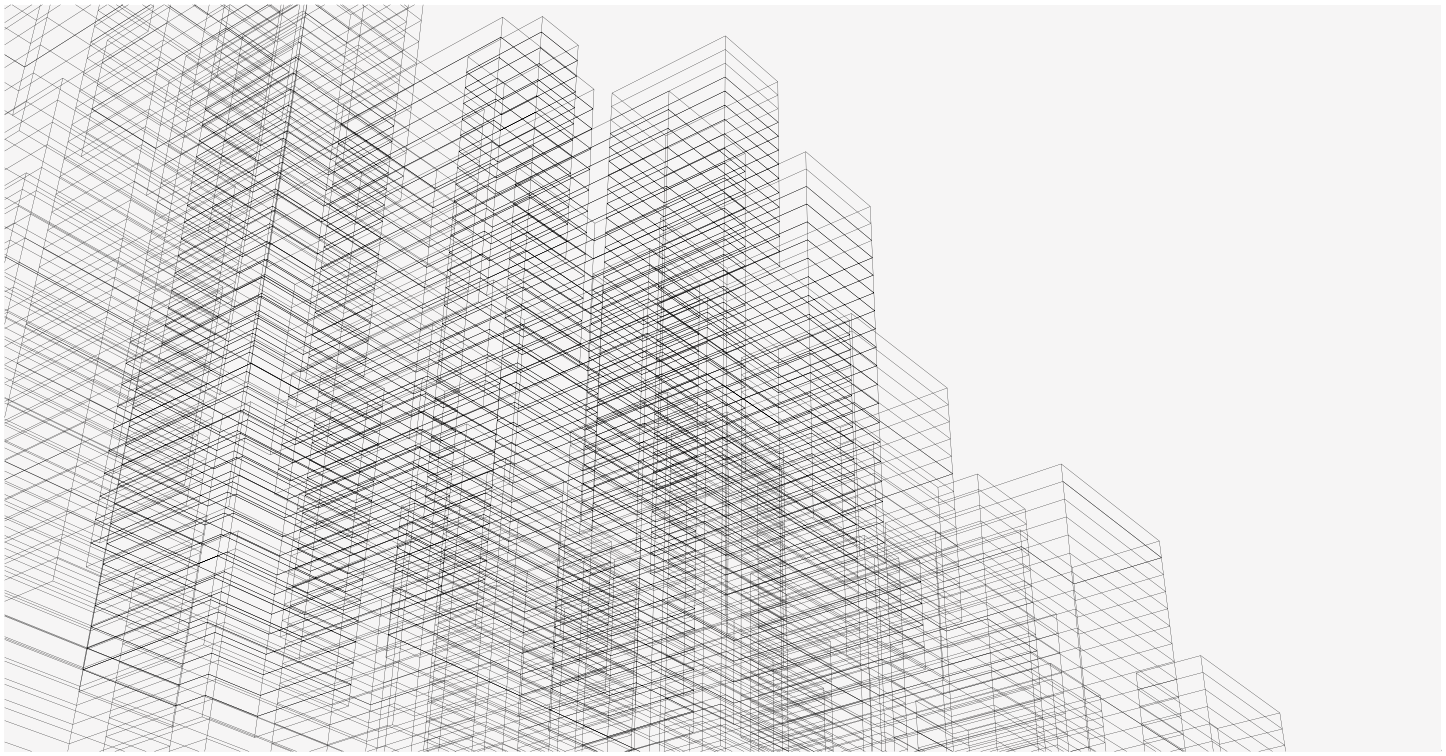
Written in Go, Sparta[10] is similar to Serverless (described above) in that it generates and uses CloudFormation to provision and manage AWS resources "behind the scenes".

Apex

The node.js Apex[11] framework allows for the build, provisioning and management of Lambda functions. Apex integrates with other tools including Hashicorp Terraform, and because of this does not attempt to manage any AWS resources other than Lambda, including API Gateway.

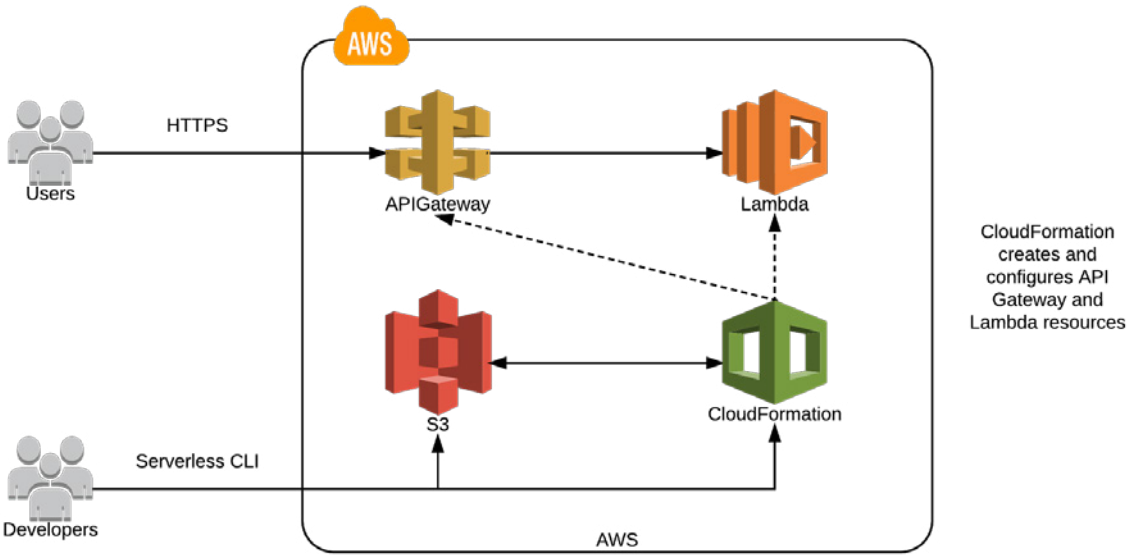
SAM

AWS' own Serverless Application Model[12] (SAM) is a set of extensions to CloudFormation designed to enable the deployment of serverless applications with API Gateway, Lambda, DynamoDB, and so on.



Creating Your First Serverless App: A Tutorial

The following sections describe how to create and deploy a simple serverless application using the Serverless Framework. This will be an HTTP endpoint that will return a simple string along with a standard HTTP 200 response to indicate success. The following diagram illustrates the application and supporting resources that will be created:



The following instructions assume a unix-like OS but should be easy to adapt for other operating systems.

Requirements

Install the Serverless Framework (requires Node, instructions are here):

```
$ npm install -g serverless
```

Configuring your AWS credentials

Ensure that your AWS credentials are configured as described here.

Creating the Service

The next steps are to create the files needed by the Serverless Framework to deploy the application.

Handler

The handler contains the actual code that runs in response to an HTTP request, and in this case is a short Python snippet. Create a file with the following contents and save it as **handler.py**:

```
import json

def endpoint(event, context):
body = {
“message”: “Hello Contino World!”
        }
        response = {
“statusCode”: 200,
“body”: json.dumps(body)
        }
        return response
```

Service Configuration

The service configuration is used by the Serverless Framework to configure API Gateway and the underlying Lambda functions that will comprise the service.

Create a file with the following contents and save it as **serverless.yml**:

```
service: hello-contino-world

frameworkVersion: “>=1.2.0 <2.0.0”

provider:
 name: aws
 runtime: python2.7

functions:
 hello:
        handler: handler.endpoint
        events:
        - http:
        path: hello
        method: get
```

Further Reading

The Serverless Framework documentation is comprehensive and a good resource for getting started:

- Serverless Framework documentation

As well as the core documentation, an excellent set of example repositories is available:

- Serverless Framework example repositories

Deploying the Service

To deploy the service, type the following command:

```
$ sls deploy
```

This will cause the Serverless Framework to:

1. Generate and upload CloudFormation templates for the required AWS resources to S3
2. Signal CloudFormation to create API Gateway and Lambda resources
3. Package the application code and upload it to S3 as an artefact
4. Update the CloudFormation stack to reference the application artefact
5. Display some information about the deployed service

You can test the service by sending an HTTP request to the endpoint displayed in the output:

```
$ curl https://XXXXXXXX.execute-api.us-east-1.
amazonaws.com/dev/hello

{“message”: “Hello Contino World!”}
```

Additional commands can be used to query or interact with the service:

```
$ sls info
$ sls deploy list
$ sls logs -function hello
$ sls metrics
```

Notice that the default “environment” is in the URL path here as “dev”, and is implied in all of the above commands, done this way for safety. Additional endpoints can also be deployed:

```
$ sls deploy -stage test
$ sls deploy -stage prod
```

This shows that a very simple mechanism for progressing changes from development through to production can be implemented with ease - of course in a real-world deployment these different environments would likely be in separate AWS accounts. In this model the API Gateway stage is not used to indicate the environment.

To decommission all services, run the following commands:

```
$ sls remove
$ sls remove -stage test
$ sls remove -stage prod
```

Continuous Delivery With AWS Lambda

Even though serverless is a powerful and rapid model for developing applications, many of the traditional considerations around the software development lifecycle still exist. For instance, we still need to build, test and deploy our code - usually in a team setting. This section provides some background and guidance on how this can be done using continuous delivery.

Why Continuous Delivery?

The benefits of the practices of continuous delivery are widely documented and it is outside of the scope of this paper to describe the advantages in full. In short, the use of CI/CD:

- Encourages merges to be frequent and thus more manageable
- Drives comprehensive automated unit and integration testing
- Allows code issues to be found and fixed quickly
- Increases confidence in the built artefacts
- Allows for more frequent, more granular releases
- Decreases the number and severity of production defects

Lambda-Specifics

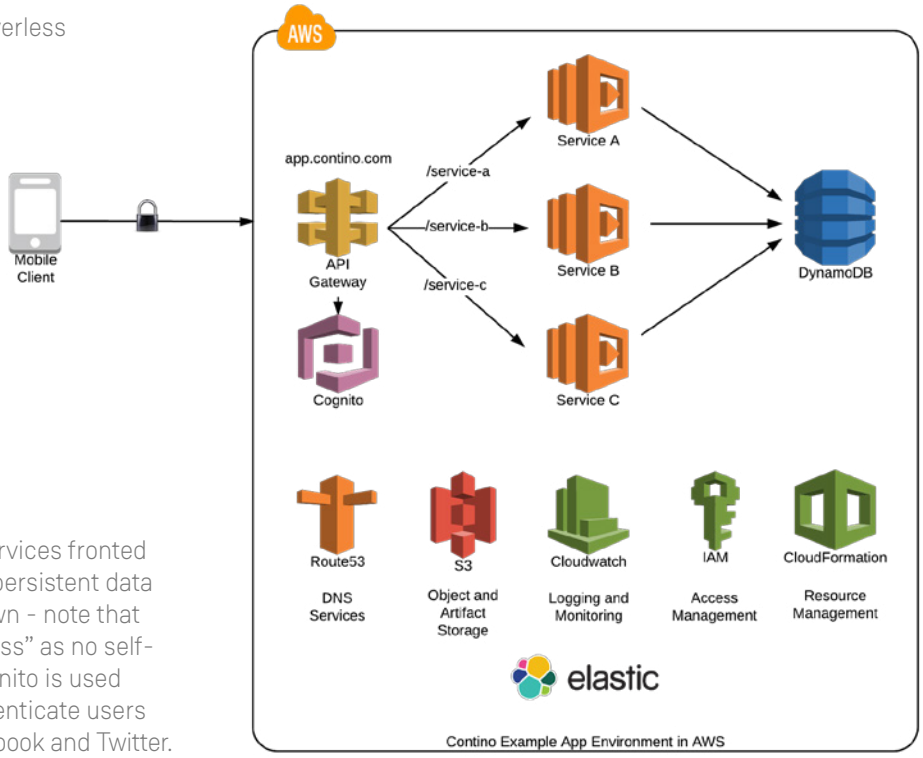
Given the distinct characteristics of serverless applications, build and deployment comes with some challenges - most of these are a result of the relatively new technology in play and the somewhat immature ecosystem and tooling that surrounds it. In contrast to these challenges, however, some aspects become easier - there is no long-term state to manage on the host OS or filesystem, and because each function invocation has to be able to start “from fresh”, this enforces a clean and consistent approach.

AWS provides multiple building-block options that can be used for deploying serverless application code from a repository to a Lambda execution environment. Because these are somewhat low-level, and will require some scripting/coordination to be effective in a CI environment, most non-trivial CI pipelines will make use of a supporting framework to perform some of the more repetitive tasks and avoid having to reimplement common functionality. Factors that may influence framework choice are, in no particular order:

- Support for required languages/run times (or support via shims)
- Implementation language and ability to contribute to future enhancement/extension
- Integration with other provisioning mechanisms (CloudFormation, Terraform, etc.)
- Developer support and development community
- Quality of documentation
- Feature set

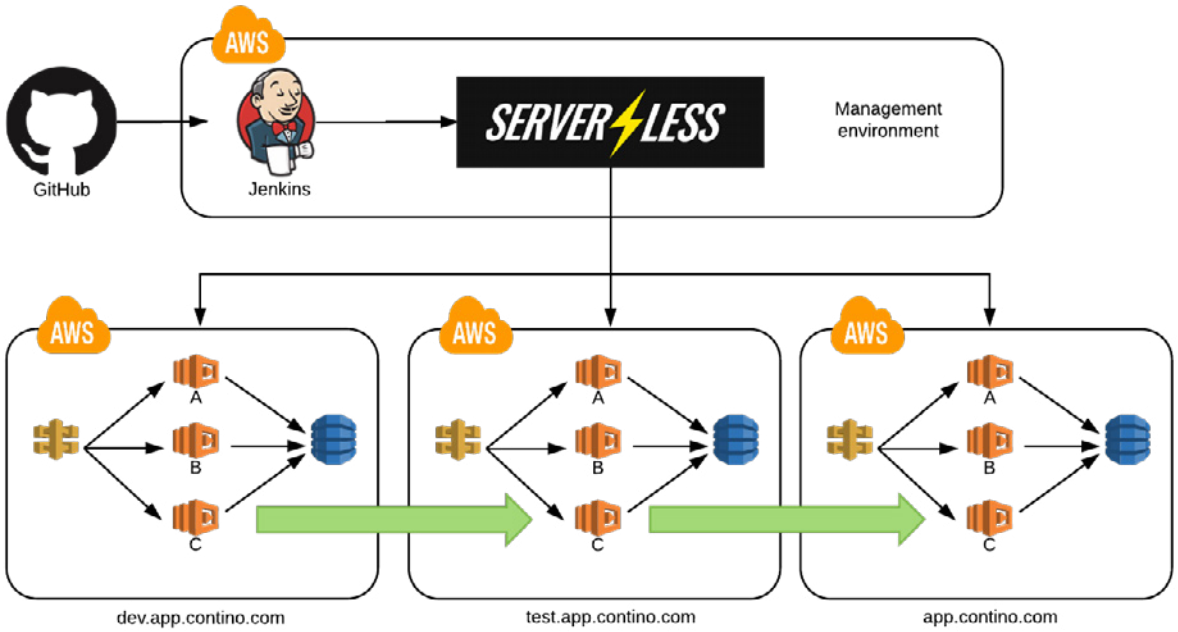
Example CI/CD Pipeline for Lambda Applications

The following diagram shows how a simple serverless application may be implemented in AWS:

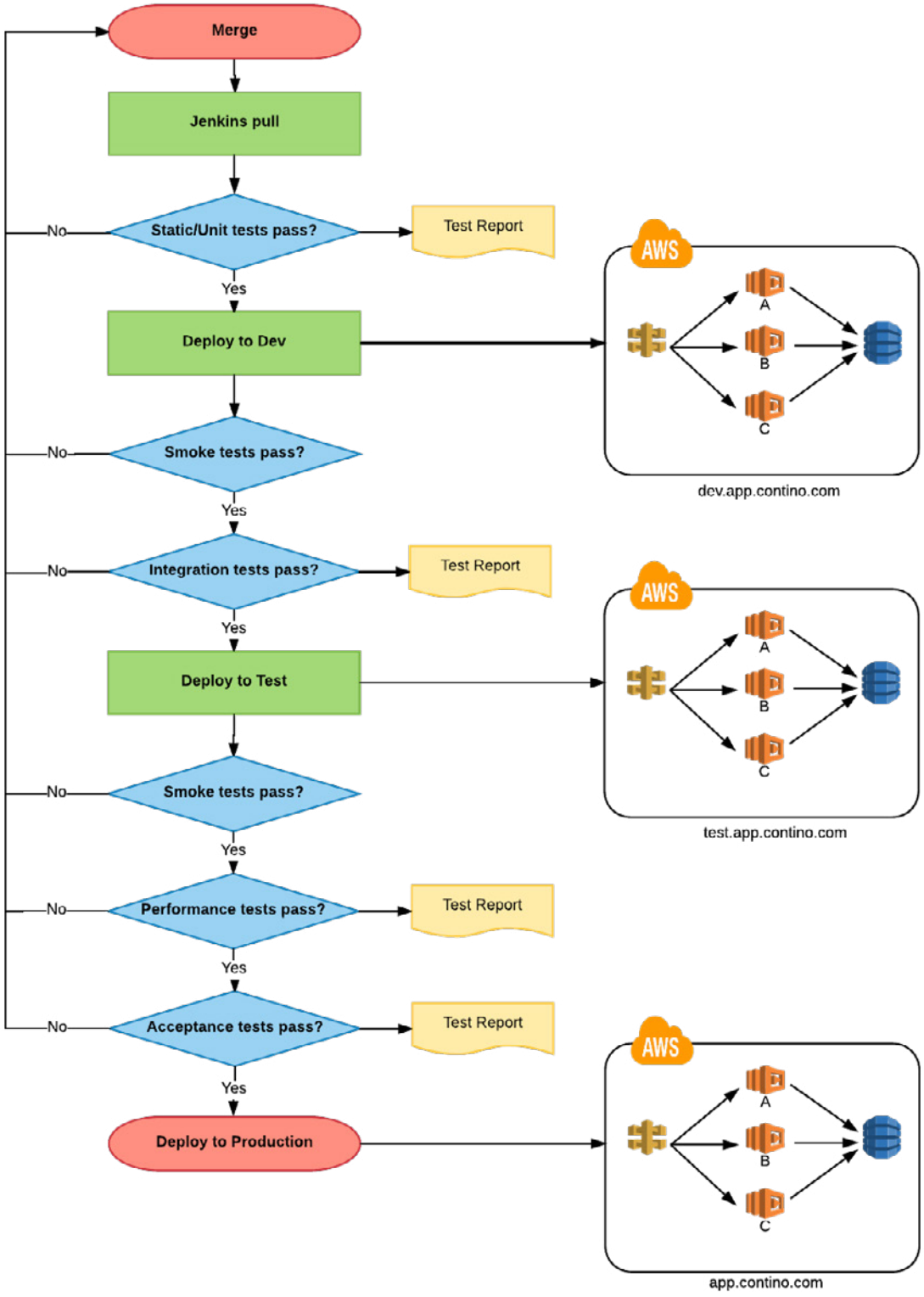


This shows a number of Lambda application services fronted by API Gateway and backed by DynamoDB for persistent data storage. Supporting AWS services are also shown - note that the application environments are truly "serverless" as no self-managed EC2 instances are required. AWS Cognito is used to allow user sign-up and sign-in, and can authenticate users against identity provider platforms such as Facebook and Twitter.

Multiple application environments - in this case development, test and production - are managed with a single central management environment as follows, with green arrows showing the promotion of application services from Dev to Prod:



For code merged by a developer to eventually be deployed to production, a number of stages or "gates" must be passed, in their entirety forming the build pipeline. Example stages in a pipeline for an application service may be as follows, with each stage causing the pipeline to exit if the stage fails (for example, if tests do not pass):



An example Jenkins 2.0 pipeline (slightly different from that described above) is as follows:

More sophisticated pipelines may incorporate blue/green and/or canary deployments so that new deployments can be validated good before taking existing deployments out of service, thus minimising risk. Additional Jenkins jobs may be used to perform database migrations between regions, perform backups, and so on.

Operational Considerations

Although the abstraction of the underlying infrastructure has led to the term "NoOps" being coined, implying reduced operational demands, running serverless applications in production does present some challenges. The following sections detail considerations for logging and monitoring.

Ingesting Serverless Applications Logs

As mentioned earlier, logs from Lambda functions are streamed into AWS CloudWatch Logs, from where they can be retrieved and viewed via the AWS console or CLI. For serverless applications that are comprised of multiple Lambda functions, however, this approach is not recommended:

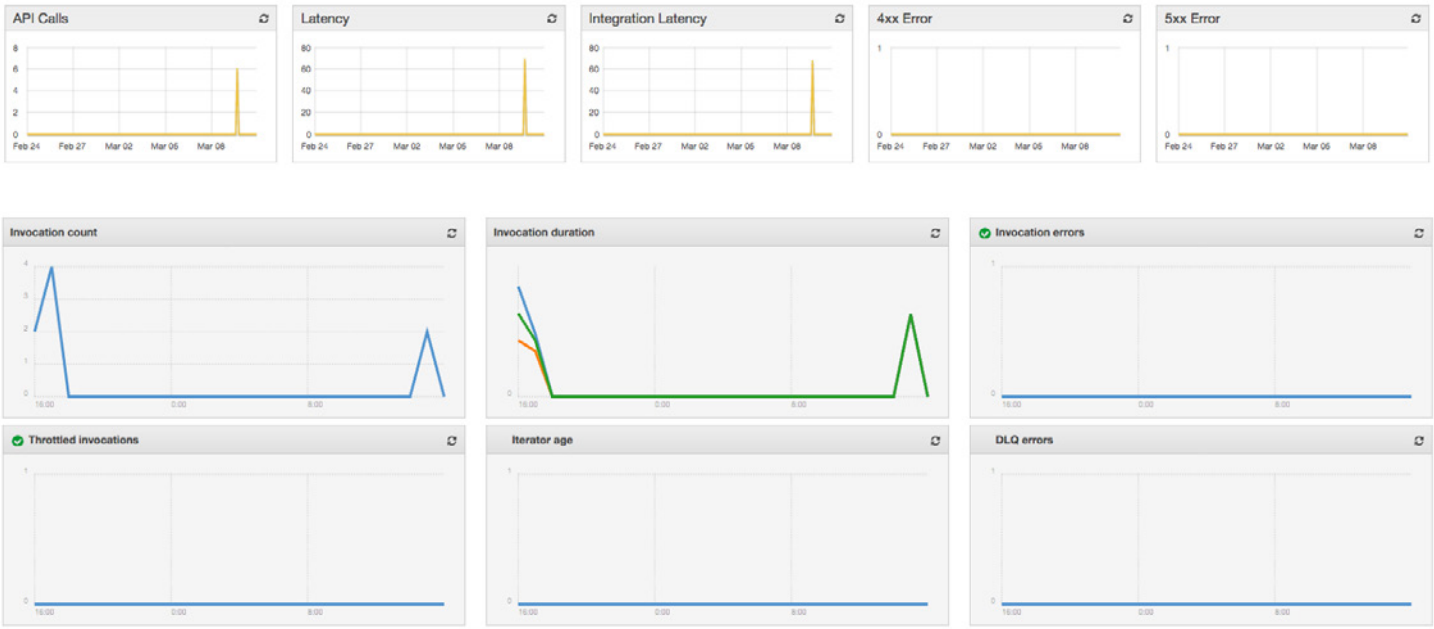
- Tracking requests or user journeys incorporating multiple services is impossible
- Native search and correlation functionality is very limited
- Complex searches/alerts can be difficult (e.g. percentage change, rate of change)
- At scale, the volume of log traffic becomes unwieldy to deal with

This means that a more scalable and fully featured solution is required. As briefly mentioned earlier, a simple Lambda function can be used to stream CloudWatch logs to AWS Elasticsearch, from where it can be indexed and presented via Kibana dashboards to different audiences with different needs; for example, there may be distinct dashboards for BAU, troubleshooting and management use, for example. An alerting mechanism can be incorporated to notify interested or responsible parties when particular conditions occur.

A more sophisticated logging infrastructure may make use of AWS Kinesis Firehose or similar technology to stream CloudWatch logs at scale into a warehousing and analytics platform, or alternatively a third party provider such as DataDog may be used.

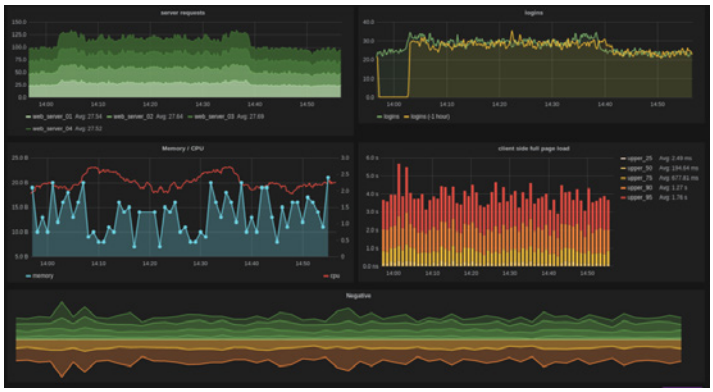
Monitoring Serverless Applications

AWS platform metrics are surfaced by CloudWatch - this includes API Gateway, Lambda and DynamoDB:



Alerts via email or SNS can be configured to alert teams or individuals to particular conditions, such as the number of non-200 HTTP responses per minute being elevated beyond an acceptable number, for example.

An alternative approach to the above is to pull CloudWatch metrics and ingest into a time-series database such as Graphite or InfluxDB. From there, a front-end such as Grafana can be used for dashboarding.



Conclusion

As we've seen, AWS Lambda - and the serverless computing model in general - is a game-changer. This is in terms of both efficiency and scale, both of which eventually equate to cost: the ability to provision near-infinitely scalable services, on-demand and with sub-second billing granularity provides the best possible value to public cloud consumers.

The differences between "traditionally" provisioned EC2 instances and the Lambda execution environment obviously mean that architectural thought must be given to Lambda-specifics, such as fixed maximum execution duration, statelessness and other factors discussed earlier in this document. Of course, not all workloads are well-suited to Lambda, which means that an upfront evaluatory step is required for each new application or application component; generally speaking, it makes sense to run in Lambda when possible according to these criteria.

The open-source serverless supporting framework space is an interesting one - currently it is the Serverless Framework that is emerging as a market leader, but there is significant scope for innovation and development in this area and around tooling for serverless computing in general.

It remains to be seen what will be on AWS' future Lambda roadmap, but the development of such an interesting and disruptive technology is worth watching with keen interest, and we will see over the upcoming months and years how the serverless paradigm pans out.

References

1. https://aws.amazon.com/lambda
2. Other than via unofficial mechanisms such as https://github.com/alestic/lambdash
3. https://aws.amazon.com/compliance/shared-responsibility-model
4. http://docs.aws.amazon.com/lambda/latest/dg/limits.html
5. https://github.com/pywren/pywren
6. https://aws.amazon.com/step-functions
7. https://serverless.com
8. https://github.com/awslabs/chalice
9. https://www.zappa.io
10. http://gosparta.io
11. http://apex.run
12. https://aws.amazon.com/about-aws/whats-new/2016/11/introducing-the-aws-serverless-application-model

Contino is a global consultancy that enables enterprise organizations to accelerate innovation through the adoption of Enterprise DevOps and cloud-native computing.

Our dual delivery and upskilling approach supports organizations to modernize their IT processes and technologies whilst helping them develop their own innovation engine. From strategy and operations, to culture and technology, we support business leaders on their digital transformation journey, helping them maximize opportunities for growth and profitability.

Backed by Columbia Capital and with a global presence, Contino is ideally positioned to scale to meet the demands of the world's largest enterprises.

We are a global Amazon Web Services APN Advanced Consulting Partner, a Docker Premier Consulting and Training Partner, and a HashiCorp System Integration Partner.

Learn more at **contino.io**