An Introduction to Cloud Development

Elliot Forbes

An Introduction to Cloud Development and Engineering

A full introduction to the world of cloud development.

Elliot Forbes and Alan Reid

This book is for sale at http://leanpub.com/an-introduction-to-cloud-development

This version was published on 2018-09-08

I'd like to say thanks to my Mum and Dad, my family, friends and my girlfriend for supporting me through all of my projects. These projects wouldn't be possible without you.

Contents

1. Introduction

The world is moving towards using the cloud. It's undeniable. We have thousands of companies, at all levels of development, starting to look at, and investigate the advantages of migrating their software to become cloud native and to leverage the services of providers such as AWS, Microsoft Azure and Google's own Cloud platform.

These companies are moving away from the burden of maintaining their own Physical infrastructure and by migrating their systems to the cloud, and subsequently, they can leverage hundreds, if not thousands of years of collective development expertise supporting the machines and systems they run on top of and focus more on delivering key functionality to their customers.

Cloud service providers allow small development teams or even single developers to build systems that are faster, more resilient and better than ever. These cloud service providers are enabling innovation at a rate that is previously unheard of and we are seeing a myriad of different startups emerging that rely purely on these cloud service providers.

The leader in this space, Amazon Web Services, has seen a monumental 40% Year-on-Year growth since its inception and it doesn't look like it's going to be slowing down any time soon. They are constantly releasing new services and new features and attempting to make their platform the most attractive out there for companies large and small.

1.1 Book Goals

So, what is this book all about? We'll, in this book, I'm hoping to demystify some of the key concepts you will need to know in order to design and build applications that leverage the full advantages of the cloud. We'll be covering how you can design your applications so that they follow all of the 12 factors that make your application truly cloud native.

We'll cover the fundamentals that you will need to know to be a competent cloud engineer and architect. This will include the likes of Load Balancers and how you can leverage them to improve the resiliency of your applications, containers and how this improves the lives of the operate people who handle deployment and monitoring of your platforms.

We'll also cover some of the newer branches of cloud computing, such as serverless offerings and Kubernetes and how these can help to revolutionize the development and deployment workflow.

By the end of this book, you should have a formidable grasp on everything it takes to ensure that all of your applications cloud native and have been designed to handle an incredible level of stress.

1.2 Preamble

About the Author

Over the past 2 years I have been working exclusively with a large number of development teams, helping them to re-architect and re-develop traditional monolithic and non-cloud native applications to move onto cloud platforms.

I've published work on a number of well-recognized cloud sites such as `acloud.guru`, `codeburst.io`, and `hackernoon`. These have been well received by the community and collectively have achieved well over half a million readers.

I've previously published `Learning Concurrency in Python` with the help of Packt Publishing in August of 2017 and have written and recorded hundreds of technical tutorials which are available on my website: `https://tutorialedge.net`.

Contacting the Author

If, for any reason, you should wish to get in touch with me for whatever reason, then feel free to do so through the following ways:

- Emailing me: elliot@elliotforbes.co.uk
- Tweeting me: @elliot_f[1].

I'd be happy to hear any questions, concerns of feedback you may have. The content in this book is constantly being updated with the latest techniques and concepts as time goes on.

Source Code

In some sections of this book, source code will be used to demonstrate concepts. Smaller code samples will be available as-is, larger examples and standalone projects can be found in the Github repo dedicated to this book: https://github.com/elliotforbes/an-intro-to-cloud-engineering[2]

Glossary

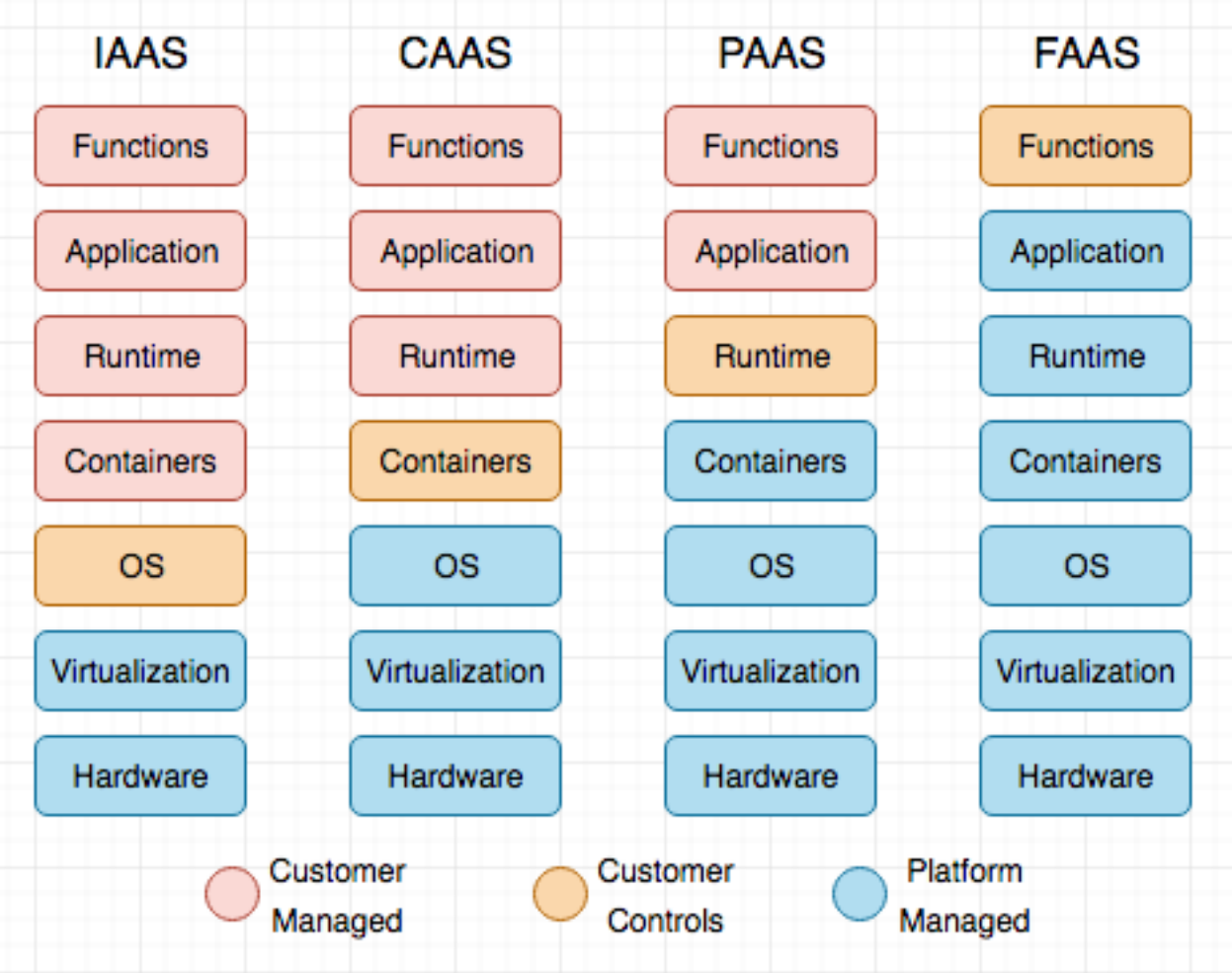
Term	Meaning
Load Balancer	A service that routes incoming requests over one or more instances of an application, effectively spreading the load so that one instance isn't getting hammered

[1] https://twitter.com/elliot_f
[2] https://github.com/elliotforbes/an-intro-to-cloud-engineering

2. Cloud Platforms

When it comes to cloud platforms, we can typically describe some of their offerings as `IAAS`, `PAAS`, `FAAS` or `SAAS`. Each of these is essentially a different level of abstraction, `IAAS` being the lowest-level of abstraction, `SAAS` being the highest. It's important to know the distinction between these different service levels that are currently on offer.

You'll more often than not see a grid that looks like the one below when people discuss the differences between these key terms:



Iaas-paas-faas

This grid is excellent at showing how each level of service differs in terms of what the application developer manages compared to what the platform or service provider manages.

2.1 IAAS

When people talk about `IAAS`, they are referring to Infrastructure as a Service. This is where you effectively rent servers using services such as AWS' EC2. You would then be responsible for the management of these servers, and for the deployment of your applications to said servers.

Any servers requested would have to be configured to meet your exact system needs. This includes specifying things like the amount of memory each server has, the operating system it is running

on and whether or not it is optimized for the likes of memory-intensive applications or graphical applications.

It ultimately offers you far more fine-grained control when it comes to managing the infrastructure your applications run upon. However, in the world of designing applications for the cloud, this fine-grained control may not necessarily needed and could slow down the rate at which you develop new applications.

2.2 PAAS

PAAS or Platforms as a Service are offerings such as Pivotal's CloudFoundry or AWS' Elastic Beanstalk. These offerings effectively manage the underlying servers for you and allow you to focus purely on your application.

These abstract away the difficulties you typically encounter when manage your own fleet of servers and, more often than not, provide you with things such as load balancers, monitoring and logging with minimal added fuss to the application developer.

2.3 FAAS

FAAS, or Functions as a Service, are a newer style of offering where you focus purely on individual functions. Say, for instance, Gemma was looking to write an API Endpoint that she wanted to be scalable, resilient and available at a moments notice. She could leverage FAAS, write her function or a group of functions with a single entry point, and then deploy this as an AWS Lambda function.

This Lambda function will then scale dynamically to handle any incoming demands and she and her company will only be charged for whenever this Lambda Function is executed. She will not have to pay for idle servers waiting for demand to ramp up, and, more importantly, she will not have to worry about maintaining those servers, supporting those servers or requesting more servers.

We'll be covering the advantages of FAAS in a later chapter and how Gemma used them to revolutionize things such as monitoring and batch processing.

2.4 SAAS

SAAS, or Software as a Service would be akin to the likes of Microsoft Outlook, Twitter, or Github. These services are essentially a black box in terms of underlying infrastructure and are merely something you would consume through the likes of a web portal, a mobile application, or a standalone client.

2.5 Public Cloud vs Private Cloud

For many large enterprises, the concept of throwing all of their confidential data straight into a public cloud offering can sometimes be a terrifying prospect.

For larger enterprises, the solution to this is to expose private cloud offerings to enable internal developers to take advantage of all the benefits that cloud technologies offer, whilst retaining the data within on their own physical servers.

This approach, however, requires a team to manage any internal cloud offerings that are made available. But having one team do this is often far preferable, in terms of cost savings, than having every team having to manage their own servers.

This is the reason that platforms such as Pivotal's CloudFoundry are being developed. These platforms enable companies to develop and host their own private cloud offering with minimal fuss.

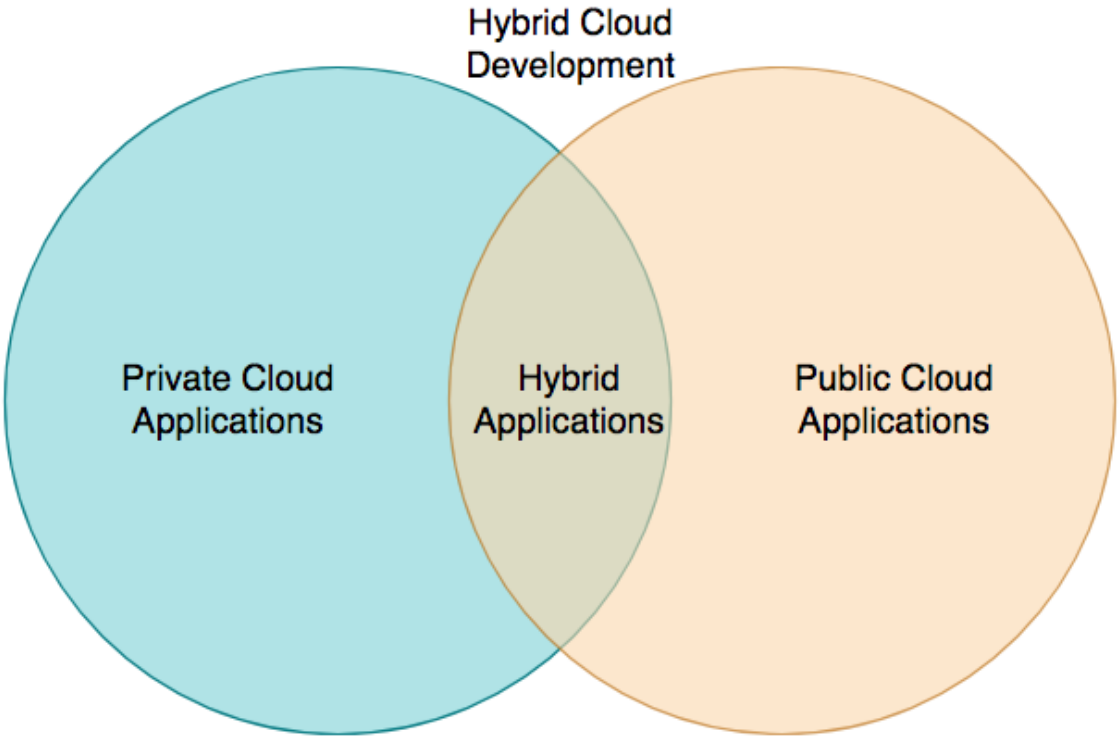
Hybrid Cloud Offerings

By offering an internal cloud platform, you by no means preclude yourself from using external public cloud applications. In fact, I've seen a number of successful examples of teams developing what is known as a hybrid cloud application.

These hybrid cloud applications are typically made up from a number of distinct services or microservices and deployed both on internal and external cloud offerings. The reason for doing

this is that it gives you the best of both worlds.

You can effectively architect your applications in such a way that it handles any confidential data internally, whilst leveraging public cloud platforms for your non-confidential data processing needs.



Hybrid Applications

This approach can also be extended further to include applications deployed across traditional infrastructure. Say, for instance, you have an application that leverages a infrastructure such as message queues which are deployed on traditional infrastructure within your organization.

A popular approach for moving to the cloud would be to pick apart pieces of your application and refactor those pieces to be cloud-native. You could then run this refactored part of your application at the same time as your traditional application until you are sure that everything is ready for a true production load.

By picking apart your application, and transitioning your applications in a phased approach, you save yourself from the potential scenario where everything fails at once and you experience

significant downtime. It gives you a safer route to onboarding to the cloud.

2.6 Conclusion

In this chapter we looked at the various levels of service offerings, from IAAS, all the way through to SAAS and we also looked at both Public and Private cloud offerings, how they differ and why they are needed.

3. Load Balancers and Auto Scalers

In this chapter, we'll be looking at how you can utilize Load Balancers and Auto Scalers in order to add resiliency to your application. We'll touch upon the different types of Load Balancers that are available and we'll look at how you can implement zero-downtime deployment pipelines using techniques like Blue-Green deployment and Canary Deployments.

3.1 Introduction

Now, many developers I've seen are content to run their software on a single server instance and hope and pray that their single instance does not crash for any reason. This may be fine for a simple blog site or something that isn't highly critical and earning any significant revenue, but for serious projects, it's absolutely unacceptable.

If you have applications that are highly critical and cannot, under any situation go down for any period of time, then deploying that application to a solitary server instance is on par with playing russian roulette.

That solitary server instance could fall over at a moments notice and knock out your entire infrastructure and effectively wipe out your application. This is why it's vital that we start architect our applications in such a way that they aren't reliant on a solitary instance so that they can handle server failures.

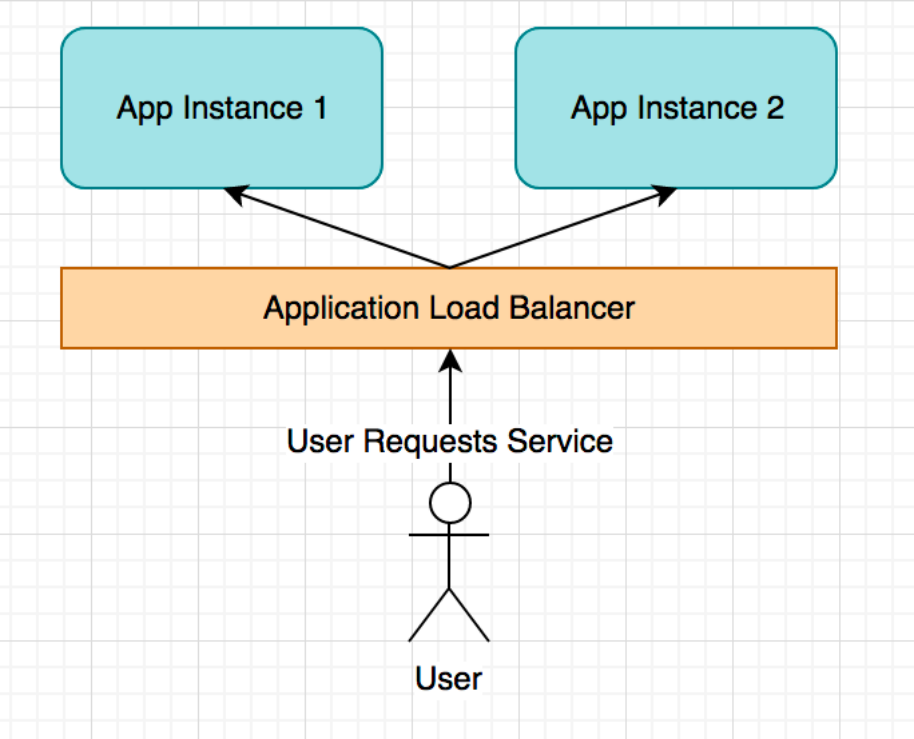
This is where load balancers come into play.

3.2 Load Balancers

So, load balancers are pretty cool in the sense that they can distribute every request coming into a load balanced service across multiple instances of an application. This effectively means that you

could have 4 instances of your application deployed and running on 4 different servers in 4 different regions and have a load balancer spread all of the requests across these instances of your application in various ways.

This is ideal for scenarios where you must never have any downtime. If one of your servers falls over for any reason, the load balancer will just route any incoming requests to any of the other instances that are still currently running.



Basic Load Balancer

Types of Load Balancer

When it comes to configuring a load balancer, you can choose from a number of different types of load balancer. You could opt for a classic round-robin style load balancer, or go for a load balancer that features some underlying logic in how it handles requests.

Most cloud service providers typically provide at least 3 different types of load balancer:

Type	Description
Classic Load Balancer	This is your standard round-robin style load balancer which distributes requests evenly across instances attached to that load balancer
Geographic Load Balancer	The geographic load balancer looks at where each particular request is coming from and tries to route the request to the closest geographic server instance
Network-based Load Balancer	The network based load balancer looks at latency and tries to route the request to the server that will provide the lowest latency

3.3 Deployment Practices

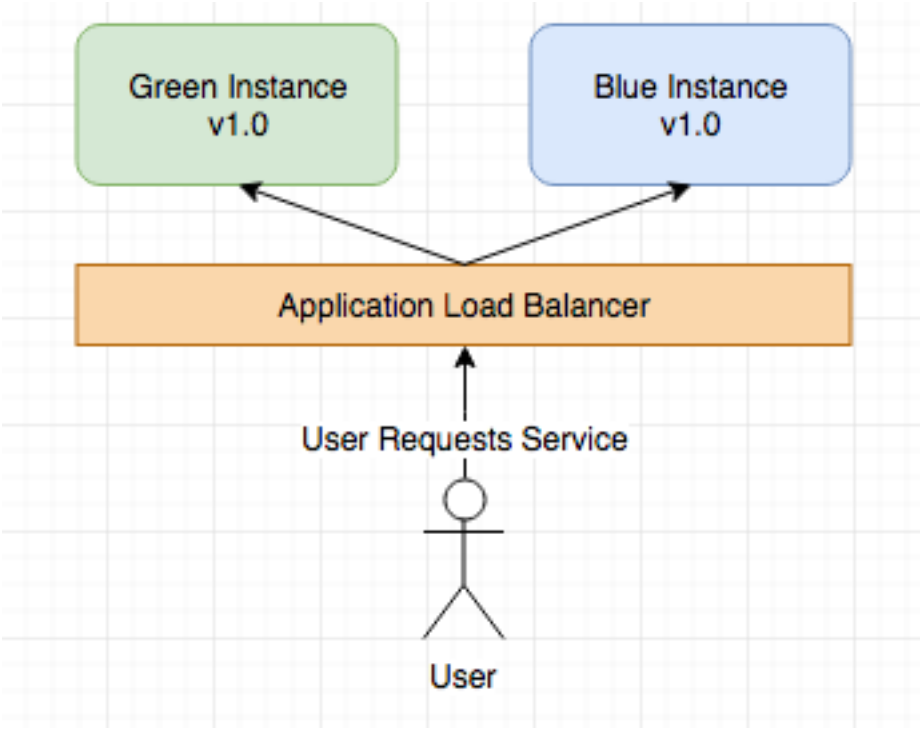
When it comes to deploying new versions of your systems to replace existing versions of your application, you typically have a variety of different approaches you could choose from. Each of these has certain advantages over the others and it's important to consider the benefits of each of them when planning how you wish to do deployments.

In this section, we'll look at some of the most popular deployment strategies you could use to ensure minimal-to-no downtime updates to your applications.

Green-Blue Deployment

The first deployment strategy, and quite possibly the most commonly used strategy, is your standard green-blue deployments. Say, for instance, we had 2 instances of our application running behind a load balancer.

In this particular scenario, we've called one instance our `green` instance and the other our `blue` instance. It's important to note that these are identical in all but name.



green blue deployments

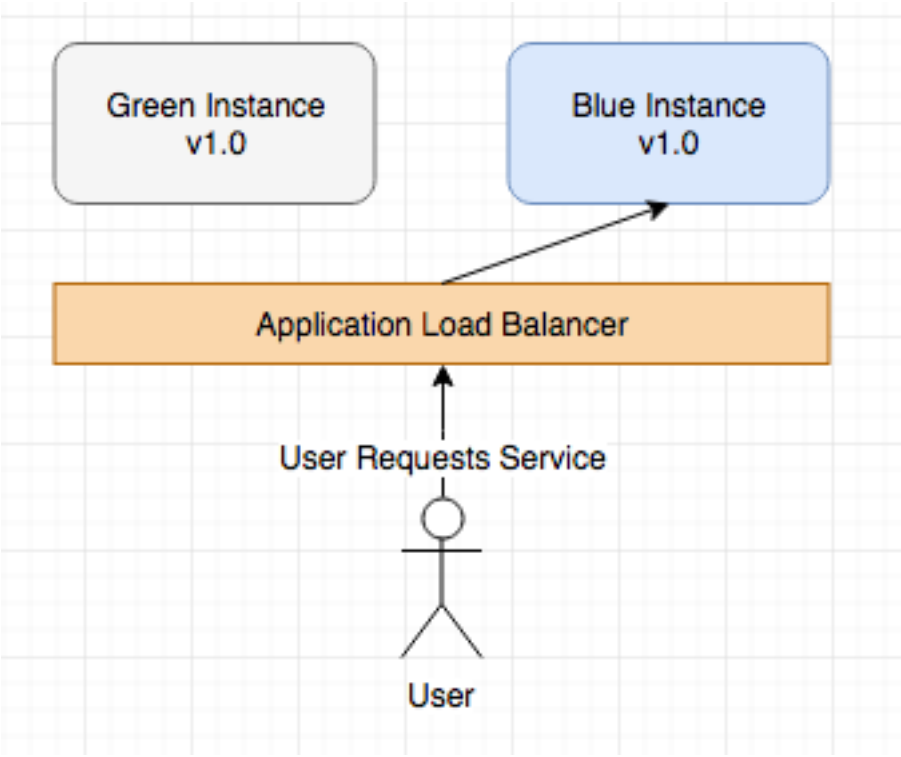
At this point, our application developers finish cutting the release for version 1.1 of our new application, and are now at the point that they wish to release this into production for clients to consume.

So, how do they do this without incurring downtime?

This is where this blue-green strategy comes into play. Let's have a look at how this works step-by-step.

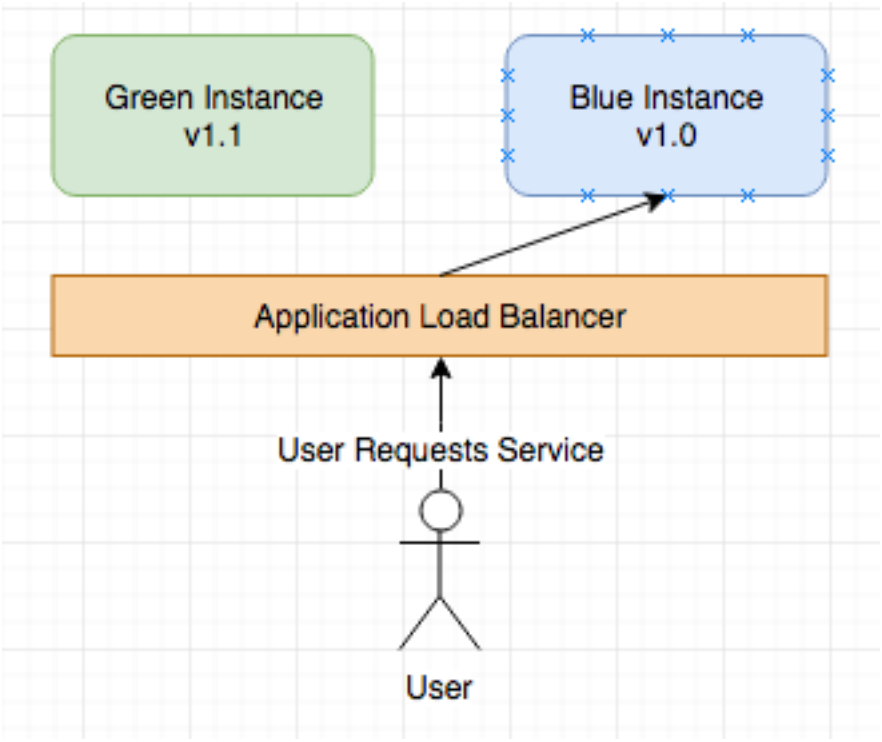
Step 1

Step 1 would be to remove our `green` instance from our load balancer. At this point in time we are down to only 1 server and instance of our application currently handling requests from our users.



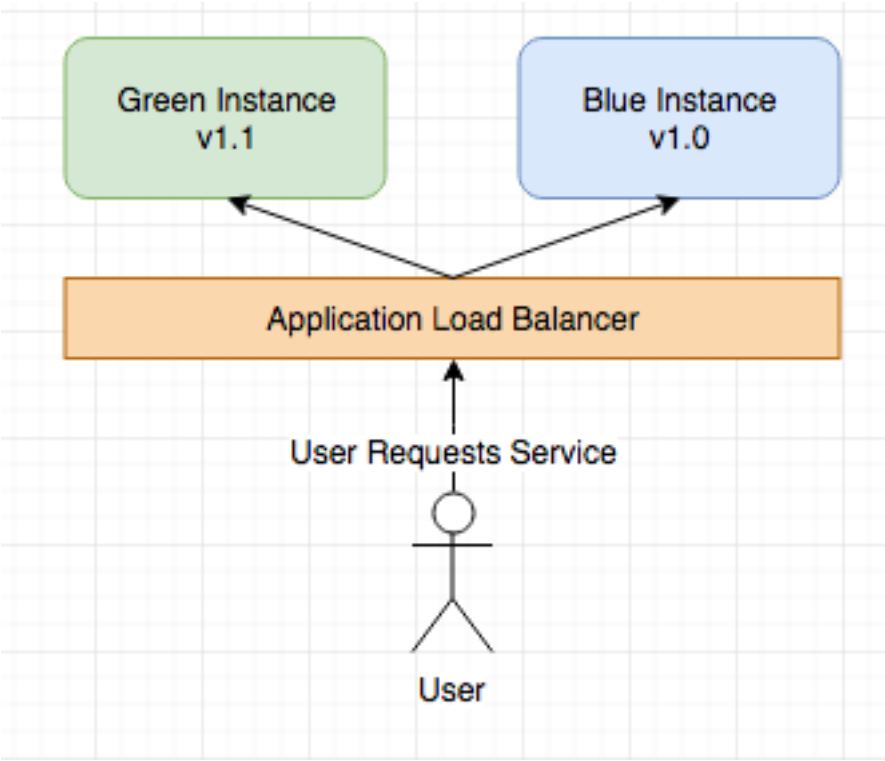
step 1

We would then swap out our `green` v1.0 instance for our newly cut v1.1 instance and perform any last minute checks or integration tests to ensure that when our instance does start accepting production requests, it doesn't start throwing up any errors:



step 2

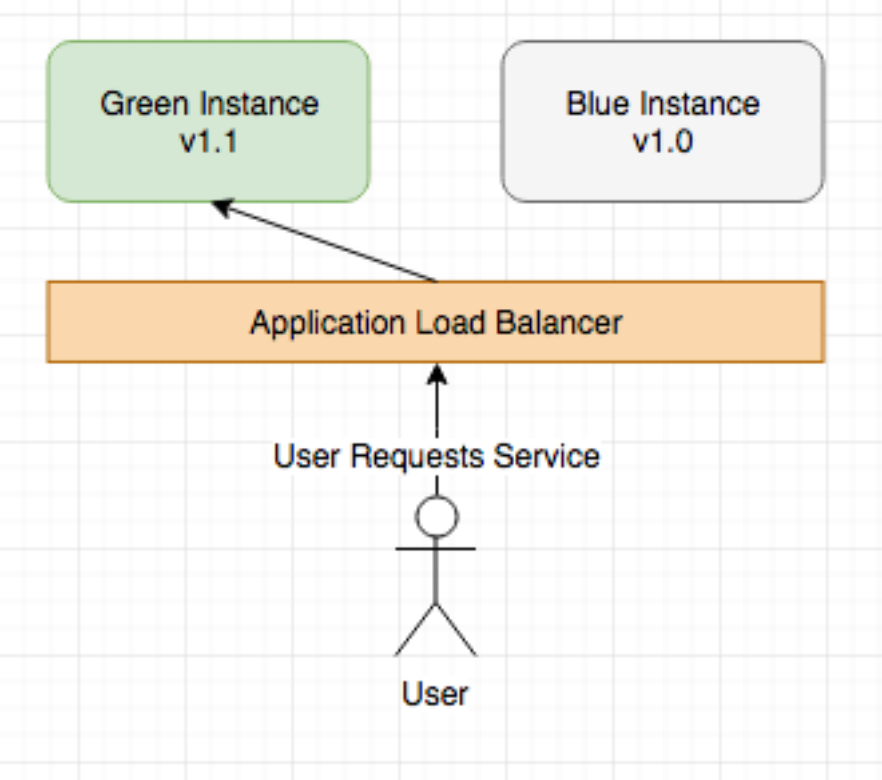
And once we were happy v1.1 of our application is up and running as we expected, we would subsequently re-add it back to our load balancer setup:



step 3

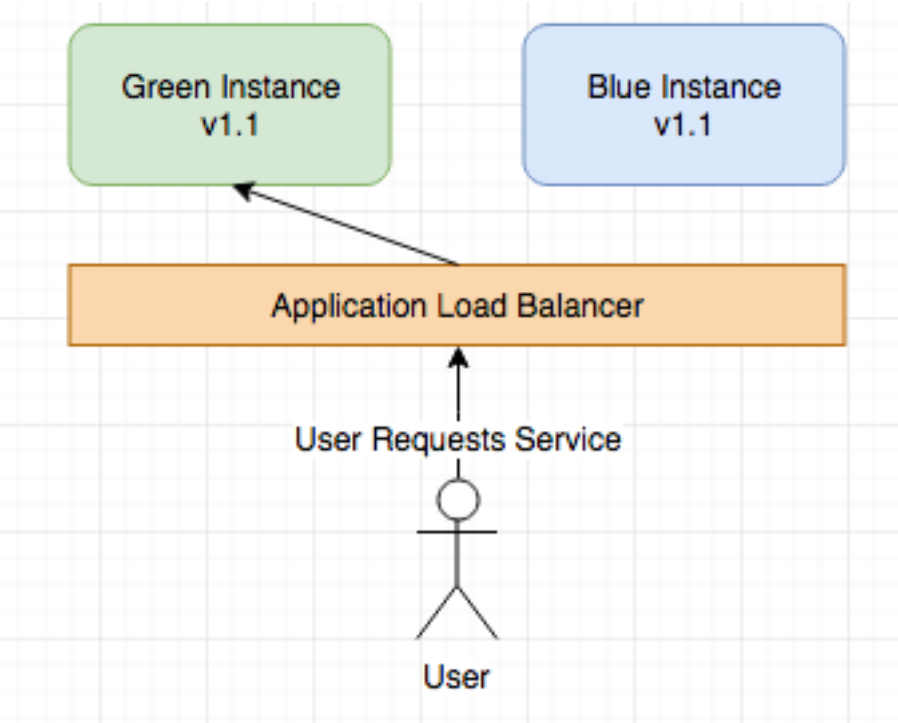
At this point in time, our application will be sending requests to both v1.1 and v1.0 of our application.

We now need to update our `blue` instance in an identical fashion to how we updated our `green`. We first remove the `blue` instance from our load balancer setup:



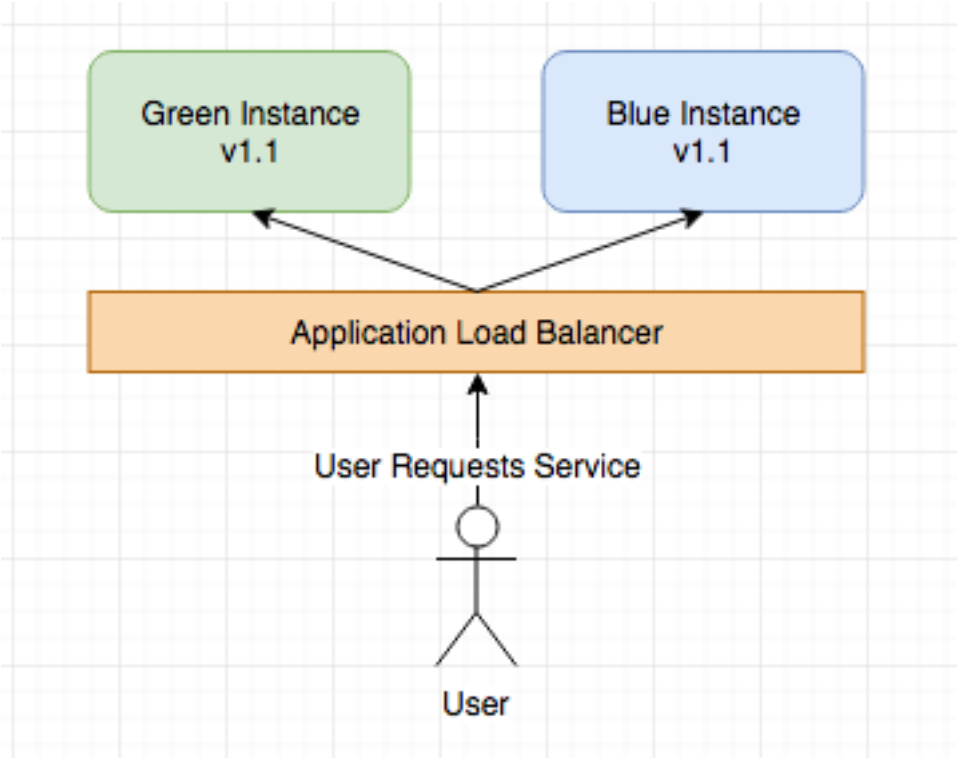
step 4

And then subsequently deploy our new v1.1 of our `blue` application over our old version:



step 5

Before finally re-adding our `blue` instance back into our load balancer configuration:



step 6

And ta-da! We've managed to update our application from version 1.0 to version 1.1. You should also notice that, at no point throughout this deployment process, was our application not accepting requests from our users.

We've effectively managed to deploy a new version of our hypothetical application with no downtime and no stress. We were able to verify that our new v1.1 of our application was good to go before going live, should there have been any issues with v1.1 when we performed out tests, we could have simply redeployed version 1.0 and re-added that back to our load balancer configuration.

Canary Testing

Another method of deployment that Gemma could leverage is Canary Testing. The name of this comes from the olden days of people working in mines, there would often be scenarios where Carbon Monoxide levels would rise dangerously and the miners had no way of determining this.

In order to combat this, miners would place canary birds in cages along the sections of the mine

that they inhabited. When carbon monoxide levels rose, this meant bad things for the poor canaries. The miners however would see their demise and quickly get out of the mine until levels of Carbon Monoxide dropped.

Imagine you were now trying to test how good a new feature of your application was but didn't want to impact everyone in the world using your service. With Canary testing you would deploy changes only to a very limited geographic spread and monitor to see how well these changes are picked up. If they are successful, you would then start to roll these changes out to other regions in an incremental fashion. If they failed then you would only have impacted a certain geographic location.

Large companies such as Facebook utilize this method and will commonly deploy things to the country of New Zealand before any other country in order to test any big changes.

Feature Flags

Another cool way to deploy new changes is to utilize something called feature flags. Essentially, within your application, you would have a conditional statement that determines whether or not to use a new feature depending on what state a particular flag is in.

If the flag is enabled then the new feature is utilized. If they face issues with this new feature and it starts to impact users, they can simply toggle the feature off until such time as they can debug what has went wrong and how they can fix it.

These flags can be flipped at runtime and thus, if they do need to turn it off, it presents minimal effort on behalf of the developers and it presents minimal impact to the users.

3.4 Auto Scalers

One of the key issues that Gemma and her team faced was that of a variable amount of traffic coming through their services. At times this would be just a couple of users per minute which was

easily handled, however when some of the comics went viral, this number could go up to hundreds of people using their site, per second.

In order to ensure their services never went down due to too many people being on their site, they had to provision servers that were large and more than capable of handling any level of the traffic that hit them.

This however was not ideal as it meant that in times of incredibly low usage these servers were left under-utilized and were costing more money than they were making. There were also times where even the largest of servers would start to struggle when certain pages went viral online.

This is where autoscalers came in to save the day.

Gemma and her team could couple their services up with autoscalers that would be able to dynamically observe and react to varying levels of traffic being placed on their systems. This meant that in times of low traffic, the autoscalers would minimize the number of servers currently active and only increase this number when demands on the system increased.

This not only saved the company a lot of money as they were no longer paying for incredibly powerful servers to be up at all times, it also saved them from constantly worrying how their site was going to handle periods where their content went viral.

Stateless vs Stateful Applications

When it came to designing new applications one thing Gemma had to bear in mind was the statefulness of said applications. When designing cloud-native systems it's best to keep the applications stateless so that they are not only easier to test, but they are also easier to scale horizontally.

Horizontal vs Vertical

When it comes to scalability, you can either scale your apps horizontally or vertically.

When someone says that they wish to scale their app horizontally it means that they wish to deploy more instances of their application across more servers in order to handle the load.

If they wished to scale their application vertically then it would be a case of increasing the size of the servers their application was currently running upon.

Each of these ways has their merits but the easiest one to perform is without a doubt horizontal scaling. With vertical scaling you have to get a snapshot of your current EC2 instance, provision a larger instance and then deploy that instance with your snapshot. This requires stopping and killing the old instance.

With horizontal scaling, you simply clone an existing t2 instance and start it. Once it's started you add it to your load balancer and, if you have designed your application to be stateless, it should now be able to handle more traffic with minimal fuss.

3.5 Summary

In this chapter we looked at the various ways that Gemma and her team could improve the resiliency of their applications running in the cloud through the use of various different types of load balancers.

We also looked at the various ways that they could safely release new changes through the utilization of different deployment practices such as Green-Blue deployment, canary testing and feature flags.

4. The Cloud 12 Factors

In this chapter, we'll be looking at all of the factors that go into making your application truly cloud-ready. These factors are language-agnostic and when developing your own applications, you should be checking to ensure that each design decision you take adheres to these 12 factors.

By adhering to these 12 factors, you will ensure that all of your applications are able to leverage the full benefits of the cloud. The official 12-Factor App website can be found here: https://12factor.net/, I highly recommend checking this out!

> For the purpose of demonstration, we'll be using Python to demonstrate how you can adhere to some of the following 12 factors.

4.1 Codebase

> One Codebase Tracked in revision control, many deploys

In order to comply to this factor, you will have to store your project's code within a code repository and use a software version control system such as `git`, `mercurial`, or `subversion`.

Say for instance Gemma, our senior engineer was developing a new cloud-native application using NodeJS. She would start by creating a new code repository and saving her initial code within this.

She would then be able to create multiple branches within this repository which would then be used to deploy to the likes of her development, testing, or production environments.

Typical Flow:

When developing a cloud native application, the standard process it to create `feature branches` that correspond to the likes of a `JIRA` ticket or other ticketing system ticket.

A developer like Gemma would then be able to do the work detailed in the ticket she was working on and commit any changes she makes to that branch in her codebase. Once she is happy with her changes and has tested them locally, she can then merge these changes into a `develop` branch.

At this point, a CI/CD pipeline would pick up these changes and deploy them automatically to a `development` environment in which a series of integration tests could be run and teams could evaluate whether these changes work as expected.

Once they are happy that the development tests are passing, these changes would be merged to a `UAT` branch. They would again be deployed automatically, this time to a higher testing environment that closely resembles production. Further tests would be run and once these pass, the changes Gemma was working on would then be merged into a `master` or `production` branch.

Upon merging into a `master` or `production` branch, the changes would automatically deployed to Production and users would be able to see those changes.

4.2 Dependencies

> Explicitly declare and isolate dependencies

Traditionally, when working on systems, you would install a series of packages and dependencies on the server prior to deploying your application there. This means your application would be implicitly expecting underlying dependencies to be there on the box already.

When it comes to managing a fleet of servers, this starts to become a problem and keeping dependencies across all servers inline becomes difficult.

In order to combat this, Gemma could explicitly declare all of the underlying dependencies that her applications need to run. If you were using Python or NodeJS, you would typically do this by specifying a `requirements.txt` or `package.json` file that explicitly list all of the dependencies your application needs in order to run.

This is where containerization technologies such as `Docker` start to shine. If you were running your

application within a Docker container, you would essentially enforce this kind of behavior as your docker application would fail to run should dependencies be missing.

Benefits

The key benefit of explicitly declaring your dependencies is that it simplifies the process of deploying and running your applications. New developers can come into the team and get the application up and running fairly quickly as they know exactly what is needed in order for their application to run successfully.

4.3 Config

> Store config in the environment

When developing applications for multiple environments, you will typically want to change things like the servers you wish to interact with, or the database you connect to when performing tests. You don't typically want your `development` or `testing` environment running on top of a `Production` database and potentially corrupting your important data.

> If you are developing Open Source code, storing usernames and passwords within your code is generally quite dangerous. You could find yourself exposing credentials to paid services quite easily and as such rack up a huge bill when hackers scrape these credentials and use them to their own gain.

These variables are typically known as your application's configuration and this configuration should be stored outwith your applications codebase at all times. Having any hard-coded URLs or credentials is a direct violation of this factor.

In order to be truly cloud-native, the best practice for working with configuration variables within your application is to inject them directly into your environment. These environment variables, typically shortened to `env vars` or `env`, can be changed easily and prevents you from potentially checking your usernames and passwords into public repositories.

Environment Variables in Practice

Cloud service providers such as AWS allow you to specify things like the environment variables you wish to set when you are instantiating new servers.

4.4 Backing Services

> Treat backing services as attached resources

Say Gemma wished to expand her application and start interfacing with services such as a database or a message queue. These would be considered your backing services for your application.

Ideally, if a database server in Production started acting weird or breaking, then you would be able to dynamically swap this with another database without restarting your application. Frameworks such as Python's Django support this dynamic runtime configuration and it's possible to add this functionality to your own applications should you wish.

You could periodically have your Database service within your application poll for environment variable changes and then update when it sees any of these changes to use the new database specified.

4.5 Build, release, run

> Strictly separate build and run stages

Your systems should have three distinct phases: `Build`, `Release`, and `Run` that are entirely separate from one another.

- `The Build Stage` - This stage is where our source code is compiled into an executable bundle which we refer to as the `build`. This is typically built from a specific `commit` or `tag` within your code repository and should be repeatable and automatic.

- `The Release Stage` - This stage is where we combine both our executable `build` and our applications configuration. At the end of this stage, your application should be immediately executable.
- `The Run Stage` - In this stage we start the execution of our application.

4.6 Processes

> Execute the app as one or more stateless processes

When designing your applications, you should try to architect them in such a way that they are a stateless by design.

By following this practice, developers like Gemma would be able to dynamically scale the number of applications running in their production environments to suit the demands placed on their application.

Say for instance, Gemma was writing a RESTful web service that would serve up all of the comics to TheComicCo's website. This web service may typically only see a couple hundred requests per minute during a standard work day. However, when Saturday comes rolling around and a new incredibly popular Comic is released, the number of requests per minute may skyrocket and the servers running the REST service may start to struggle to cope.

By designing this service in such a way that it's stateless, Gemma could spin up new servers running her application and add them to a Load Balancer Group. The load balancer would then distribute the number of requests coming in across the available servers within the group and be able to handle the requests without worrying of one particular server becoming overwhelmed and crashing.

These instances should share nothing and any state should be stored within a backing service such as a database or messaging queue. Should you need a temporary cache, you can resort to using the filesystem if it's for things such as downloading/uploading files but you should typically avoid this if at all possible.

4.7 Port binding

> Export services via port binding

Twelve factor applications need to be completely self contained. If Gemma was developing an application locally, she may be able to access that application by navigating to the likes of `http://localhost:3000`. This is an example of port binding and when deploying your application to a production environment, you would typically rely on a routing layer on top of your application to forward any requests to your underlying service running on a distinct port.

4.8 Concurrency

> Scale out via the process model

You should ideally design your system to be a set of smaller processes that act independently and in a accordance to `VI - Processes`. You could have a number of processes within your application dealing with new requests coming in, a number of processes processing any more complex requests, and possibly a number of processes handling sending email or pushing notifications to clients.

4.9 Disposability

> Maximize robustness with fast startup and graceful shutdown

When designing applications that are cloud-native, you should try to ensure that they can be started up and killed incredibly quickly. This again ties into another of the twelve factors: `VIII - Concurrency`.

When scaling out your application to meet surges in demand, you typically want the start up times of any newly created instances to be as low as possible. This ensures your application can scale to

suit demand in a timely demand and the likelihood of your application crashing due to surges goes down.

Any applications that are killed gracefully should be able to be started up in a clean manner. There should be no side-effects of a previous crash impacting a new instance of your application.

4.10 Dev/prod parity

> Keep development, staging, and production as similar as possible

In an ideal world, your test environment should be identical to your produciton environment in order to ensure that what works in test, will continue to work in production after it has been released.

This however, isn't always possible due to factors such as cost. You may not be able to afford to replicate the fleet of production servers you have serving your clients in lower environments.

4.11 Logs

> Treat logs as event streams

When it comes to implementing logging in your cloud-native application, the best practice is to have your application log everything to standard out and have your underlying infrastructure handle it.

In the past, Gemma may have simply written log statements to a series of time-ordered log files that would stay on the server either indefinitely or in such a manner that older log files would be over-written after a certain period of time.

When dealing with hundreds of instances of your application, debugging any issues may be troublesome. You may find yourself logging into quite a number of individual boxes in order to find the associated log files for any errors that may have occurred.

Piping to Off Platform Logging Systems

The best way to handle logs within your distributed systems would be to pipe all log messages to the stdout using the appropriate print statements for whatever language you happen to be working in:

```
# Python
print("my super important Log message")
# Node.JS
console.log("my log message")
# Golang
fmt.Println("My Log message")
# Java
System.out.println("My log message")
```

You would then have a system in place on your servers that would pipe all of these log messages to an off-platform logging system such as Splunk, or Elk.

By implementing logging in this way, we save ourselves the trouble of having to log into the vast fleets of servers we may have running our applications and we centralize where we have to go in order to debug issues within our system.

4.12 Admin Processes

> Run admin/management tasks as one-off processes

There are times where Gemma and members of her team may wish to perform one-off tasks such as database migrations and so on. These are tasks that typically reside outwith your applications normal function.

When it comes to these one-off processes, it's typically best to write a script that performs this one-off task for you and then commit that into your application's code base. This means it's easy to run this against a parallel version of your code running in a testing environment and it also means that you have an audit trail which you can reference further down the line.

5. Containerization

In this chapter, we'll be looking at containerization in depth. We'll be touching upon the history of containerization and then looking at the current industry leader, Docker, and how you can then use Docker for a simple Python 3.6 application.

5.1 Introduction

Docker is a containerization technology that allows you to define a container that will feature everything your application needs to run.

Say for instance you are writing a small Python application using a very specific version of Python. You can easily create a docker container that is based upon that python version and run it very easily using the docker run command.

This is immensely powerful as Docker enables you to specify only what's needed for your application to run. You don't need to build every container off incredibly heavy Ubuntu images unless you absolutely need to. This allows me the ability to save memory and only have run containers that have the absolute minimum requirements needed to run my app.

5.2 Containerization

What is containerization? What does it really do?

Well, the concept of containerization initially came from the shipping industry. In days gone by, ships had no standards as to how they should package produce that they were to transport across the globe. This lead to a lot of space wasted, and it would be difficult at the other dock when it came to unloading the different shapes and sizes.

In 1955, a mr Malcom p. McLean, a trucking entrepreneur from North Carolina bought a steamship company with the idea of transporting entire truck trailers with their cargo still inside. This proved to be vastly more efficient and improved the way containers moved between ships to trucks and trains.

Docker essentially allows us to benefit from these same advantages. Regardless of the place you are running them, you handle the docker container in the same manner. This means that the way you run your docker based application on your local machine is exactly the same way that you would handle it in on a production server, regardless of operating system underlying it.

This means that windows developers can develop an application on top of their preferential windows machine the exact same way a MacOS or a Linux developer would develop on top of their machine.

The Benefits of Containerization

The benefits of using containers, even within teams that aren't looking to leverage cloud based platforms, is huge. Within Gemma's team there were times where newer developers would come in and getting them up and running with the appropriate environment configuration for all of the services would take up to a week of a senior developers time.

This was just to get their services running locally, which was required before the new developer could dive in and try to understand how the system worked.

By adopting docker as the basis for all of their applications, Gemma and her team were able to reduce the time taken to get a service up and running on a new persons local machine down to a single command. This hugely improved the speed at which new people could be brought up to speed and made them far more effective far sooner.

Another inherit advantage that a technology like docker brings is the fact it is so lightweight. Traditionally teams may have used full blown Virtual Machines in which they would run their applications. These virtual machines are most definitely not well known for how lightweight they are and startup times left a lot to be desired. This is due to the fact they are starting up a complete operating system in order to run.

Docker containers, on the other hand, are far more lightweight and can be started in a couple of seconds.

5.3 Images

I tend to think of images as blueprints for any containers I wish to run. Images in docker are inert, immutable files that are essentially snapshots of a container. You can instantiate a container from an image.

In order to list the available images you have, you can use the `docker images` command which will display a table similar to this one below:

```
REPOSITORY                      TAG                 IMAGE ID            CREATED    \
         SIZE
red                             latest              6484161e895a        3 weeks ago\
         234MB
chrome                          latest              4836d476232e        3 weeks ago\
         624MB
firefox-vnc                     latest              4836d476232e        3 weeks ago\
         624MB
<none>                          <none>              173e558c3f0f        3 weeks ago\
         743MB
```

Images are incredibly useful for getting things up and running really quickly. For example, you could pull down an image from the official Docker registry: registry.hub.docker.com which could contain a RabbitMQ broker all ready to go, all you would have to do is call `docker run` specifying that image name and things like the ports you would wish to expose and you could have RabbitMQ running on your machine in a couple of minutes!

If we consider Gemma and the Comic Co, if she, for instance, wanted to run a static website and wasn't concerned about the underlying Operating System. She could pull down an `nginx` base image and then configure her application to run within this nginx container. She could then run this container on whatever hardware she may have available, be it windows machines, linux machines or her mac and she wouldn't have to perform any additional configuration.

As long as each of the machines she wanted to run this site on had docker installed, she could start her application with the exact same command and her `nginx` based container would start up and begin serving her static site.

5.4 Container Lifecycle + Persistance

- You can persist data!
- docker is really good for stateless apps

5.5 How Containers were used at The Comic Co.

During the transition to a microservice based approach Gemma oversaw a couple of new faces joining her team. There's always a learning curve when joining a new team and a transition period which reduces the overall effectiveness of the team as they look to get everything set up on their work machine.

No newcomer should feel stranded and as such Gemma put some of her senior engineers with the newcomers in an attempt to get them up and running with the codebase as quickly as possible.

5.6 Dependency and Environment Hell

Originally there were 4 people within Gemma's team, they all worked as a tightly knit unit and were able to very quickly and easily fix any dependency issues or environment issues on each others machine with not too much time wasted.

However, as the team expanded, the number of differing environments became an issue. One team member may have been running on CentOS, another on MacOS and a third on Windows 10. With 3 respective services composing their comic-book viewer this meant they had to ensure every machine had the correct dependencies and environment variables set in order to run.

One team member may have had Python 3.1 installed on their machine whilst the account service actually required Asyncio in order to work. Asyncio only became available in version 3.4 of the language and thus the service would fail to start on their machine.

Now whilst this may have been somewhat trivial to solve, it did represent time spent trying to synchronize everyone to be on the right version. As the codebase grew in complexity, so too did the environment config needed in order for it to run.

III. Config — 12 Factor Applications

If you have worked on making applications cloud friendly then you may have encountered the 12 factors. Gemma and her team were trying to ensure that every service within their application adhered to all of these factors in order to easily scale and deploy their services with very little to no manual effort.

The third factor is that you store all config items within your environment. This typically means using python commands such as:

```
1 import os
2 db_pass = os.environ["DB_PASS"]
```

In order to retrieve database passwords etc. This however makes running it on your local machine rather difficult and you would have to set all of these variables first in order for your program to run. You could do this with the likes of a powershell script or a bash script but making this cross-platform compatible takes time and effort and any changes become difficult.

5.7 The Issue

The combination of setting appropriate dependencies and environment variables for all newcomers was starting to become a bit of a burden and there were times when tests were run against production databases. This was due to developers forgetting to set environment variables back to development values.

With plans for further team expansion in the future this issue needed to be resolved.

5.8 The Solution — Docker

This is when Gemma discovered the wonderful world of containerization and Docker. With containerization Gemma and her team were able to define everything they needed for their application to run within a Dockerfile and then run their newly coined docker app with one simple command.

They started by converting their account service which was written in Python to use docker and came up with something that looked like this to start with:

```
FROM python:3.6.3
WORKDIR /app
ADD . /app
RUN pip install -r requirements.txt
EXPOSE 80
ENV DB_PASS dolphins
CMD ["python", "app.py"]
```

They specified the precise Python version they wanted their app to run in, they specified the working directory of their application and they ran a pip install -r requirements.txt in order to install all the necessary Python dependencies. Finally they exposed port 80 which was the port that the underlying Python application was running on.

You'll see on the second last line that they were also able to specify their DB_PASS environment variable which would subsequently allow them to connect to their development database.

5.9 Running Their App

The real beauty of defining this Dockerfile was that everyone in the team would be able to build and start the application after having installed Docker on their local machine by using the following 2 commands:

```
docker build -t py36-account-service .
docker run --name "py36-account-service" -d -p 9000:80 py36-account-service
```

This would subsequently go away, download all of requirements needed and build our Docker image file which we could subsequently run.

A screenshot of Gemma running this locally After this point she was able to run the application by calling the second Docker command specified above.

5.10 Running The Account Service

By migrating the app to this new format it meant that anyone new to the team could be told to simply install Docker if they hadn't already, and then run these two commands and they'd have a working account-service on their machine. This would ultimately save a lot of time in the long run.

This worked across all of the developers Operating Systems with minimal fuss.

5.11 Dealing With Multiple Environments

So whilst the above structure allowed developers to get up and running with one environment of their application, it doesn't solve the problem of developers accidentally running application tests against a production environment.

To combat this we can do things like specifying what –env-file we want to pass in to our application. Within these environment files we could specify our dev, test and production credentials separately. We could then specify that we wanted our application to start and use our development database like so:

```
docker run my_app --env-file .dev
```

5.12 Conclusion

Through the utilization of docker, Gemma and her team were able to minimize the barrier to entry for new developers coming into the team and greatly reduce the time taken to get someone up and running with a working development environment.

Docker-izing their application also made deployments simpler as they were able to leverage AWS services such as the Elastic Container Service (ECS) and not have to worry so much about the underlying operating systems etc. This ultimately saved them time and effort and the costs were the same as if they were to run it across a normal EC2 instance.z

6. Container Management with Kubernetes

In this chapter, we'll be looking briefly and how you can utilize container management technologies such as Kubernetes to create a really simple, resilient application. We'll also be covering some of the advantages of Kubernetes and how you can implement Kubernetes today on AWS.

For the purpose of this chapter, we'll be creating a really simple Kubernetes application

6.1 Introduction

When you start to break up your monolithic applications into a series of smaller microservices, you'll start to encounter more pain when it comes to managing your fleet of microservices.

Deployments start to become trickier and monitoring and keeping individual instances of your app running becomes painful. Kubernetes helps to mitigate this pain by monitoring each instance of your app and ensuring that it's still alive, if it sees an instance go down for any reason, it immediately tries to provision a new instance and ensure that our application is available at all times!

6.2 A Basic Example

Let's imagine a scenario where your company has a number of servers on which they deploy their systems. If you wanted to ensure full utilization of these systems you could employ the likes of Kubernetes to schedule your container based applications on these servers in such a way that it optimizes hardware utilization.

A Simple Docker Based Application

Let's create a really simple NodeJS 9 application that resides within a Docker container. We need to first create a new directory on our local machine and add a `Dockerfile` which features the following setup:

```
FROM node:9-slim

EXPOSE 3000
WORKDIR /app
COPY package.json index.js ./
RUN npm install

CMD ["npm", "start"]
```

Our `index.js` will look like so:

```

```

6.3 Pods

`Pods` within a Kubernetes context, typically represent either a single Docker container, or multiple grouped Docker containers. This could, for example, be an nginx container serving the frontend of your application, coupled with a Node.JS docker container which serves your applications RESTful API.

6.4 Services

6.5 Ingress

6.6 Readiness + Liveliness Endpoints

When designing your distributed systems, monitoring is an important aspect to consider. By implementing both `Readiness` and `liveliness` health-check endpoints within your container based applications, you are able to leverage the self-healing functionality of Kubernetes and improve the way your system handles failure.

The Liveliness Endpoint

The liveliness endpoint is the `HTTP` endpoint that Kubernetes will pool at set intervals to check whether or not your container based application is healthy and hasn't crashed. If the Kubernetes daemon hits this endpoint and receives a `500` response, it will terminate this `pod` and schedule a new one within the Kube cluster.

Readiness Endpoint

The `readiness` endpoint differs slightly to the `liveliness` endpoint in the sense that it signals whether the application is ready to receive new requests.

You could, for instance, be running a larger application that takes several minutes to start up and set up things like database connections or perform tasks such as populating any in-built caches. At this stage, the application is still considered healthy but we don't want to start routing requests to this particular `pod` as it will not be able to handle these requests.

Instead, we specify a `readiness` endpoint that will return a `200` response when it is ready to start receiving responses.

6.7 Conclusion

In this chapter, we briefly covered some of the basics concepts that you will have to become familiar with in order to successfully start your journey developing Kubernetes-based systems.

7. Serverless

Lambda's are an incredibly powerful and fairly new concept that allow you to forget about the hassles of dealing with massive frameworks and instead focus on writing simple small functions that can be triggered through either different events taking place or through an API.

When Gemma and her team were looking at introducing newer features that didn't necessarily fit within the scope of any of their defined services they looked towards AWS Lambda.

By going with Lambda they could leverage the languages that best suited these incredibly niche jobs.

7.1 Functions as a Service

When it comes to categorizing AWS Lambdas, they essentially fall into the FaaS category or Function as a Service. These so called FaaS offerings are excellent in the sense that they do a lot of the heavy lifting for us.

They are known as *Serverless* as the developer does not have to worry at all over what servers the functions are being run on or where these servers are. The underlying platform is essentially a black box to the developer.

When you utilize a service such as AWS Lambda it automatically handles quite complex problems such as resiliency and scaling for you. The service essentially allows you, as a developer, to focus on building more valuable systems instead of worrying about the operations side of things.

7.2 Intro to Lambdas

When it came to choosing a language runtime that Gemma could utilize for her new Lambda based service she had a variety of different options. Currently AWS Lambda supports Java, Node.js, C#,

and Python and the support for other languages is certainly coming. The ability to write Go based functions is already there but requires a Node-shim to transpile the Go code into executable Node.js code.

Thankfully, Gemma and her team were quite the Pythonista's and were happy going with the latest version of Python as a base for their new Serverless offering.

Anatomy of a Lambda Function

All Lambda functions feature an entry point function or **handler**. This handler is comprised of the name of the file and the name of the function when working in Python. Therefore if I defined a Lambda function called `lambda_handler` in my `lambda_function.py` file, I would expect the handler to look like: `lambda_function.lambda_handler`.

When defining our Lambda function it should look something like so:

```
1 def lambda_handler(event, context):
2     # body of our function
3     return "My Response!"
```

You'll notice this `lambda_handler` function takes in two parameters, an `event` parameter and a `context` parameter. The `event` passed in is the actual event that has triggered this function. `context` simply provides more information that is passed to the function at runtime.

Triggering Lambda Functions

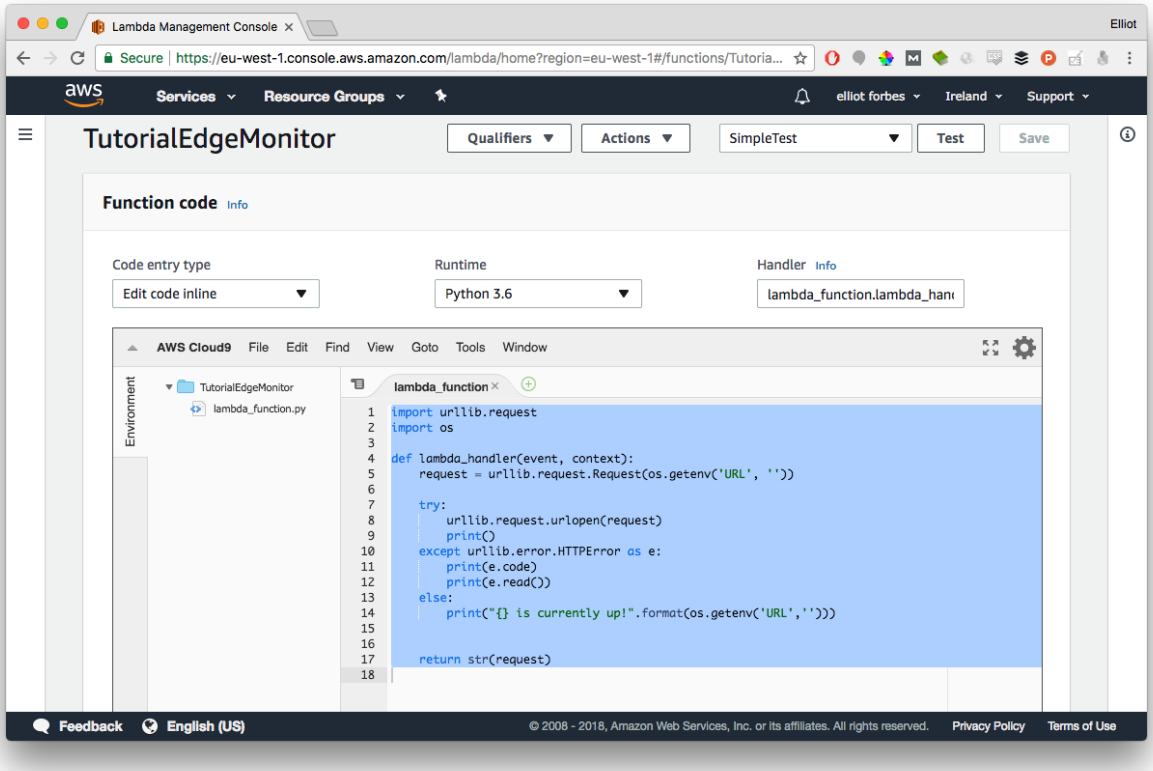
These Lambda functions can be triggered in a massive number of different ways. You could set these lambda functions to trigger at a given time interval or you could watch for changes to AWS resources.

For instance, say you were uploading a new comic book to an S3 bucket, you could have a Lambda function watch for uploads to said bucket and trigger when the upload is finished. This means that you could potentially write a lambda function that automates the notification to your massive user group that a new comic is now available on the website.

Writing Lambda Function

When it comes to writing Lambda functions you currently have two options. You can either use your normal development environment or you can use the inline AWS editor to edit your functions.

This editor tends to look something like this:



Writing in the editor

Now, when writing quick and dirty lambda functions, the inline editor is a nice and simple option and it's one that Gemma and co. used to get to grips with the numerous ins and outs of the Lambda service.

7.3 Developing and Deploying Lambda Functions

Thankfully with the AWS SAM hitting it's beta testing stage at the time of writing this, the ability to test any Lambda functions locally is now a possibility. The team can write their lambda functions in the code editor of their choice and then run the appropriate AWS SAM commands to test them quickly and effectively.

Deploying Lambda Functions

When it comes to deploying the teams lambda functions there are a number of different options. You could deploy using the AWS CLI every time you make a change to the function.

You could alternatively leverage something like CodePipeline and CodeDeploy atop your favorite source code management system.

- CodePipeline + CodeDeploy
- A Deployment Lambda?

7.4 API Gateway

So whilst a lambda function may be useful within the AWS ecosystem, what happens if Gemma and her team wanted to call a Lambda function from one of their websites?

Say for instance they had a lambda function that handled new user registrations, they could write the lambda function that would insert the user into their database or their Cognito user pool. In order to provide a way of accessing this Lambda function, they could expose it through use of the API Gateway service.

This would allow them to define an API endpoint that was accessible from anywhere in the world through a simple `HTTP` request and it would scale to cope with any surges in traffic that the site may see.

They would no longer have to run relatively expensive EC2 instances for their registration service, they could simply define their registration service as a series of Lambda functions and expose these Lambda functions through the API gateway. This is ideal as it takes away concerns about resiliency and monitoring as the API Gateway comes with all this included by default.

In terms of cost, it would also help to reduce the companies infrastructure bill as the API Gateway is incredibly low cost and charges $3.50 per million API calls received as well as the cost of data out of the service. This pricing is tiered and varies depending on what region you are in. The cost for the EU (Ireland) service is:

Amazon API Gateway Data-Transfer-Out Rates

Cost	Capacity
$0.09/GB	The first 10 TB
$0.085/GB	The next 40 TB
$0.07/GB	The next 100 TB
$0.05/GB	The next 350 TB

Deployments

When it comes to deploying your API, Gemma could make changes to various endpoints and then choose to deploy these changes to a particular stage. This means that she could in theory make some changes, deploy the test stage of her API and then run a full battery of integration tests.

Once she is happy with the results she could then deploy her changes to her Production environment. These changes would be deployed almost instantly and users of the site would be able to use any new features that may have been deployed. Again the important thing to highlight here is that these changes will be reflected across the world almost instantly, and will be fully resilient.

7.5 Monitoring and Alerting

One of the major pain points of writing an application on legacy infrastructure is the management of logs. Traditionally, Gemma and her team would persist log files to the disk of the server that they were running their applications on top of. The major challenge that this presents is, how do we backup these logs so that we don't lose them in the case of a total loss of the server?

They would then have to start configuring incredibly complex logging services that features redundancy across multiple servers in different locations in order to be truly resilient.

Debugging Nightmares

When it came to debugging applications as well, trying to find out which server handled a certain request before failing was tricky and measures had to be implemented to improve this process.

In order to debug an issue, one of the developers would first have to find the box in question, log into that box through ssh and then navigate it's file system in order to find the appropriate log files.

More often than not the log statements they were looking for would be spread over multiple partitioned log files. These would be most often partitioned based on the size of the files generated or the date at which they were recorded. This meant that some pretty crazy looking `grep` commands had to be constructed in order to grep all log files in a given directory for a particular set of log statements.

Storing Log Files

When it came to storing the log files of the various applications they had deployed across their old servers, it was a nightmare. They had to manage the persistance of these log files to a particular folder, they would then compress these into a file and push them to another backup server that featured more backing storage. In the event of that backup server going down, they were screwed and thus had to put in place measure to backup their disks to other disks that could be swapped in should one disk fail.

Overall, this process took time and money to implement. Moving to use AWS' proprietary monitoring and logging system meant that they could refocus their efforts away from ensuring their logs were safe and instead work on improving their own services. This was a huge win as far as both the developers and upper management were concerned.

Limitations of Lambda

Now before the team could fully migrate some of their existing services to AWS Lambda, they had to realise some of the limitations of the service. There are always going to be limitations with anything and cloud is no exception.

One of the first limitations is that the maximum file size of a lambda is 250MB. This means that the entire zip file that you upload must not be larger than this. As this is on a per-function basis, the likelihood of hitting this is incredibly low, unless you are using an incredibly complex array of heavy libraries.

Another key point to note is that functions will timeout if they hit the 300 second mark. You should bear this in mind when doing the initial architecture of your application. For longer running processes you may have to leverage the likes of an EC2 instance or utilize something like ECS.

- file size < 250MB to deploy to Lambda
- total function packages in a region < 75 GB
- Compressed function package < 50MB
- Ephemeral File Limitation < 512MB
- Maximum execution duration == 300 seconds
- – functions will time out if they take longer
- 100 concurrent lambda functions at a time

Workarounds

- Ask AWS increase Limitations
- Chain functions together
- Load and store files in S3

7.6 Social Media promotion with SQS

One of the things that Gemma was tasked with automating was the companies social media promotion. Every week they would send out an email and publish posts across all their social media platforms that covered everything they had released in the previous week.

These social media promotions always followed the same structure and it would typically take an hour or two for someone to craft this post every week.

Gemma decided that she would leverage a lambda function in order to automate the sending of this email. Within the lambda function she could connect to the Database powering the site and retrieve the last 5 comic book entries and their descriptions and concatenate this into an email template.

She could then schedule this lambda function to run every Saturday at 2pm using a CloudWatch trigger. Writing this simple Lambda function took a couple of hours to figure out, but once it was done it was effectively saving the company 1-2 hours a week from now on and that time could be better devoted to another task.

7.7 Summary

In this chapter we covered quite a number of different topics focused around the AWS Lambda service. We touched upon some of the things that could be achieved with Lambda's as well as their limitations.

We also looked at how we could expose these Lambda functions using the API Gateway.

8. Building Cross Cloud Applications

One of the key issues people are concerned about when developing cloud native applications is the fact that everything seems to be geared towards one main cloud service provider, namely, Amazon Web Services. They have achieved an unprecedented level of growth over the past few years and they are showing no signs of slowing down.

They have essentially monopolized cloud and rival service providers such as Azure and GCP have a fair bit of catching up to do if they wish to compete at the same level.

A large number of people are realising that, whilst the services AWS provide are excellent, what happens when if these services were to disappear overnight? Businesses that have built their entire product around a few of these offerings may struggle to find a new, similar service or will spend months developing their own proprietary platform in order to continue doing business.

This problem is generally known as `vendor lock-in`. Large enterprises are already looking for solutions to this particular problem and are exploring tools such as Terraform which abstracts away the underlying cloud service provider and allows you to consume generic services that are available across multiple service providers.

In order to remain truly vendor-agnostic, you will have to ensure that any services that you consume from the likes of AWS has a counterpart that offers similar, if not identical functionality.

8.1 An Introduction to Terraform

8.2 Conclusion

9. Conclusion

Throughout Gemma and her team's transition into using cloud technologies they faced numerous challenges. These included learning new paradigms such as containerization, learning how to rearchitect their application from a traditional monolith to a series of microservices and then how they can effectively manage this large number of microservices.

Whilst they may have faced a number of challenges they also reaped the rewards of these new technologies and they were able to innovate faster once the groundwork had been put in place. They improved the resiliency of their systems to an incredibly high degree and were able to move to implement a bullet proof CI/CD pipeline that allows them to deploy changes to their systems at a moments notice.

Overall, these improvements trickled down to the users as they were no longer impacted by frequent outages of the site or issues with certain features. As the stability of the core platform improved, the development team were able to trial a whole suite of new features with their users and not focus so much of their time fighting fires.

9.1 What the Future Brings

In terms of what the future brings for the team, that is yet to be determined. Technologies like kubernetes are relatively new in the grand scheme of things and could be replaced by faster/better options in the near future. The key point of this book though was to showcase exactly how you can leverage some of the newer technologies to improve the workflow of your own team and move away from legacy infrastructure and reap the benefits of the cloud.

9.2 Conclusion

I hope you found this an entertaining read! I'd love to hear your feedback on everything within the book or any comments on some of the topics covered, feel free to email me: elliot@elliotforbes.co.uk