

University of Hertfordshire
School of Computer Science

BSc Honours in Computer Science

6COM1053-
Computer Science Software Engineering
Project

Final Report
June 2020

Exploring Microservices In A World Gone
Cloud Mad

Jason Morsley
university@jasonmorsley.dev
SRN: 16048901

Supervised By: Joe Williams

1 Abstract

The purpose of this project is to create an AWS environment using IaC specifically Terraform to provision infrastructure and configure that infrastructure for Kubernetes and Docker, provision Kubernetes clusters for Rancher (a Kubernetes management suite), Concourse (a CI/CD pipeline) and our Walking Skeleton API.

The world has gone cloud mad. Companies are going to the cloud more and more as the pains of physical infrastructure and the constraints that are put on developers are more and more realised.

I carried out a Literature Review on Monoliths vs Microservices, this review revealed that for the majority of web workloads work better on the cloud and that tools like Docker and Kubernetes are so powerful that not taking advantage of them puts you at a disadvantage.

I've gained an understanding of microservices and can see why they may be the future of all web development and content distribution systems like Netflix and Amazon.

My findings in this project was that microservices are the future of all large-scale web applications like Netflix, the software used in this project make creating and managing this infrastructure much easier.

The system I created is functional, apart from a few security issues that are addressed in section 15.2, and an issue with getting certificates signed.

Table of Contents

1	Abstract.....	2
2	Introduction	5
2.1	Motivation.....	5
2.2	Aims & Objectives	6
2.3	Goals by the End of Project.....	6
3	Literature Review and Research	6
3.1	Abstract.....	8
3.2	Review of Literature.....	8
3.3	Limitations of Monoliths and Microservices.....	8
3.4	Tooling.....	9
3.5	Advantages and Disadvantages of Microservices vs Monoliths	11
3.6	Summary	13
4	Design.....	14
4.1	Literature Review and Research Conclusion.....	14
4.2	How Will This Be Implemented?	14
5	Docker	15
5.1	Installation Process	15
5.2	Why Docker?	16
5.3	What We've Done With Docker.....	17
6	Kubernetes.....	17
6.1	What Are Containers?	17
6.2	Brief Overview of Kubernetes Components	17
6.3	Clustering and Nodes	18
6.4	What Are Pods?	18
7	Concourse	19
7.1	Redundancy	20
7.2	Pipelines	21
7.3	CI	21
7.4	CI vs CD (Continuous Delivery) vs CD (Continuous Deployment)	21
7.5	Why Concourse?	21
8	Rancher	22
8.1	RKE (Rancher Kubernetes Engine)	22

9	AWS.....	23
10	API.....	23
10.1	Walking Skeleton	23
10.2	Restful API.....	23
11	Artifact Store.....	24
12	Source Control	24
13	Vault.....	24
14	Terraform.....	25
14.1	Terraform Modules.....	26
14.2	Terraforming a Kubernetes Cluster	26
14.3	Terraforming an AWS environment.....	27
14.4	Terraforming Concourse	30
14.5	Terraforming Rancher	31
14.6	Terraforming Walking Skeleton	33
15	Discussion And Evaluation	33
15.1	Strengths of the Project	34
15.2	Weaknesses of the Project.....	34
15.3	Improvements for Future Work.....	35
15.4	What was Learnt	36
16	Bibliography	37
17	Appendices.....	38
17.1	AWS Research:	39
17.2	Kubernetes Research:	39
17.3	Terraform Research:	55
17.4	DevOps Research:	62

2 Introduction

The object of this project was to use a wide variety of cloud-based technologies.

So to begin with, I have been provisioning infrastructure in AWS via Terraform. This has been reasonably complicated and has involved many hours of Googling. Throughout this process, I have been exposed to not only Terraform but also Ubuntu, Bash scripting, Kubernetes, Concourse, Helm, AWS, IaC (Infrastructure as Code) and Rancher.

During my research, I have been introduced to the concept of a walking skeleton.

Dr Alistair Cockburn [2008] stated that a Walking Skeleton is a tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components. The architecture and the functionality can then evolve in parallel. [Dr Alistair Cockburn, 2008, Walking Skeleton, Internet publication, <https://wiki.c2.com/?WalkingSkeleton>].

Before we can deploy our walking skeleton we need infrastructure to deploy it into. We also needed a Concourse implementation. During my research, I have come across a great tool, called Rancher. This facilitates the provisioning of Kubernetes clusters. To achieve this, I have used Terraform in combination with Helm to provision Rancher. Next, I will utilise Rancher to provision a Kubernetes cluster for the walking skeleton.

Our walking skeleton currently returns a simple greeting, however, the implementation of this involves a complete end-to-end CI/CD pipeline, using Concourse. The CI part listens to our GitHub repository and is triggered if a commit is made to the master branch. The code is then compiled, tested and containerized into a Docker image and uploaded to Docker Hub. What I will be working on next is the CD part, which will take this image and deploy into the above Kubernetes cluster.

In summary, I have learnt how to use/understand Terraform, Concourse, Bash, Helm, AWS, Rancher, C#, ASP.Net Core and Kubernetes.

From a research perspective, I have used Pluralsight, A Cloud Guru, YouTube and a plethora of articles detailing how to use the above technologies from the companies themselves and a multitude of people developing similar implementations.

Much of the work undertaken so far has been the IaC component. This has involved writing Terraform code along with Bash scripts to provision our infrastructure.

2.1 Motivation

If you are involved in web application development, it cannot have escaped your attention, but the whole world has gone microservices mad! Microservice architecture is now an accepted way to break up huge monolith applications into a significantly more manageable and robust system.

Microservices are so prolific and popular on the web, companies like Netflix use microservices in their infrastructure and Amazon uses something similar but it is distributed rather than microservices, the base principles are the same.

Microservice architecture is now becoming the de facto standard when it comes to re-platforming existing systems and moving them to the cloud. Microservices can be then be scaled far more efficiently and this helps to reduce costs and increase performance.

It has been my pleasure recently to meet some world-renowned industry professionals and my interactions have led me down the path of microservices and cloud-based infrastructure.

These personalities include but are not limited to:

- Julie Lerman
- Jon Skeet
- Scott Hanselman
- Damian Edwards
- Employees at VW Finance
- MK.NET Meetup members

2.2 Aims & Objectives

This project aims to be able to create an entire AWS environment complete with EC2 instances running Kubernetes and Docker, these will run Rancher, Concourse and our Walking Skeleton.

This will be entirely controlled by IaC, so the entire project can be built and destroyed in a very short amount of time, and each time it is built the environment will be identical architecturally.

2.3 Goals by the End of Project

To have a complete working CI/CD pipeline that will monitor a GitHub repository for code changes to the Walking Skeleton, it will then run through the various checks and tests to be run on the source code, then it will be compiled into a Docker image which will be sent to Docker Hub, this will then be picked up by the CD portion of our pipeline, this will

3 Literature Review and Research

A short preface to the Literature Review:

Throughout my research, I am discovering new and improved tools and technologies with regards to cloud-related deployments.

This research has exposed me to technologies such as Rancher and Terraform, which I probably would not have used, but are now becoming industry standards. Among these has been the following resource: <https://landscape.cncf.io/>

With regards to infrastructure, we needed to choose a cloud provider. I already knew the biggest players out there were, AWS (Amazon Web Services), GCP (Google Cloud Platform) and Microsoft Azure. However, I came across the following: <https://www.gartner.com/doc/reprints?id=1-1CMAPXNO&ct=190709&st=sb>

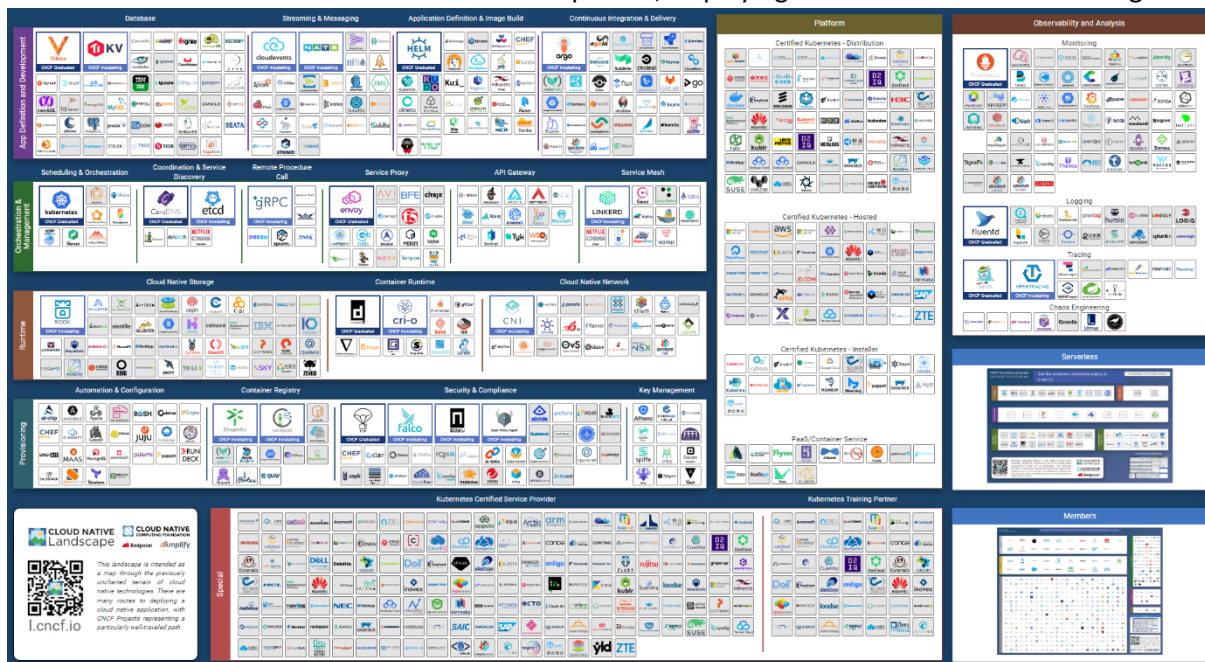
For this project, the vast majority of the research has been conducted as a member of MK.NET as they run talks on all things .NET and development, I have also used Pluralsite for a portion of my research as they have courses on the applications and tools I plan on implementing, recently they have been running seminars and talks on cloud-based infrastructure.

The resources I have taken away from these talks are numerous and I will attempt to collate them all in this section:

- [AWS](#)
- <https://cidr.xyz/> For the creation of CIDR blocks in AWS
- <https://cloudcraft.co/>
- <https://github.com/jason-morsley/>

- <https://github.com/kubernauts>
- <https://www.datadoghq.com/>
- <https://www.pluralsight.com/courses/concourse-getting-started>
- <https://www.pluralsight.com/courses/getting-started-terraform>
- <https://www.pluralsight.com/courses/deep-dive-terraform>
- <https://www.pluralsight.com/courses/patterns-library>
- <https://www.pluralsight.com/courses/encapsulation-solid>
- <https://landscape.cncf.io/>

The last resource CNCF landscape has been a tremendous help in choosing which providers and which software to use in this project as it lays out the current landscape for use in the cloud. Here is a screenshot from the CNCF Interactive landscape tool, displaying different tools and technologies:



As you can see it's a little hard to pick out the technologies we are proposing to use, simply because there are so many pieces of innovative cloud technology that we could have used, and as much as I would like to try and use as many as possible it would be very impractical.

All these resources are not instantly applicable to my project as they all work around generic scenarios for cloud development most of the work was adapting the knowledge from these courses into code that ran and provisioned correct infrastructure both automatically and with reasonable speed.

These courses were taken alongside the literature review but specifically in the literature review a great deal of research went into Tooling, and most of those tools are still being used and in the same ways researched, however, there have been additions. These tools are to be applied in the same way however they will be orchestrated by Rancher and Rancher will be run on infrastructure provisioned by Terraform.

Sarita and S. Sabastian stated that "each microservice is deployed on a container these containers are linked for a cohesive application, this application can be scaled easily by deploying one or more of these containers". This has remained the philosophy of the project, to create a scalable application which will run in containers, however, the scope of my project has crept. Now I am creating a system to provide the infrastructure automatically it will: create a CI/CD pipeline which

will detect changes from GitHub, compile into a docker image, run tests, run checks and depending on the type of CI pipeline either push to live or wait for human approval, create the Kubernetes cluster and load the image into a container.

3.1 Abstract

The purpose for this literature review is to explore the problems that come with creating a monolithic application [1], the solution to a lot of these problems will be explored in the form of microservice architecture [2].

There are many topics covered in this review and due to how niche this is there is limited peer-review studies and excerpts on the subject, 4 of such studies shall be used it should be noted that one such study is an amalgamation of all relevant grey literature on the subject.

3.2 Review of Literature

Here we will review the main premise of the project.

3.2.1 Monoliths vs Microservices

These two architecture types serve as two ways to provide a service in many forms, these could be to serve content like Netflix which happens to be a microservice or Amazon which is run as an SOA (service-oriented architecture) [3] SOA is similar to a microservice where it breaks down a monolith, but it doesn't go as far as a microservice in most cases.

In a radio excerpt from Software Engineering Radio, they talk about microservices compared monoliths [9]. In this, they describe what they believe to be a microservice, in their opinion a microservice is a small application, deployed independently, scaled independently, tested independently and has a single responsibility.

This quote matches with my view on what a microservice is and what a microservice does, it should be decoupled from its counterparts in the service so that if and when it is needed it can be replaced or updated easily, this also allows scalability which a microservice is so loved and famous for if an application is being stressed it can be easily load balanced by creating another of the same application to take a portion of the traffic.

They then go on to talk about the huge disadvantage that comes from maintenance and updating an old application built in a monolithic architecture;

"We've got this big application. It's been growing for two-and-a-half years or five years or 10 years, but we can't maintain it anymore. It's just too difficult to make any functional changes to it. We need to deploy this application into the cloud. We need software as a service, but at the moment that is impossible."

Again a point in microservices favour, breaking up an application of this scale would be a task of some magnitude. If this application was done in a microservice oriented way updating would be much more manageable as the application you need to update would be decoupled so instead of having to change the whole application and rebuild it, you would need to update or change a single module in the whole service. Applications designed without modularity will overtime be too coupled to change and test eventually collapsing under its weight. (Thönes, 2015)

This is also backed up in [11] and elaborated in and [10].

3.3 Limitations of Monoliths and Microservices

Microservices are very good at what they do, however not every application warrants the increased work to make it such. As seen in the paper by Sarita and S. Sebastian [10] and will be examined here. There are times when a good monolith will do the job just as well as a microservice, this is usually for

smaller applications or services within a business that do not need to be as polished as something released to the public, these services are usually in-house tools or logic that needs to be performed many times a day. Not all applications need to be microservices.

However, for larger applications or web applications, the tight coupling among modules is a real problem, sometimes stopping an application being SaaS (software as a service) [4] as updating and maintaining a SaaS compliant application requires fast addition of features and possibly constant changes to existing features to keep it competitive with other SaaS solutions. These applications would need to be updated eventually which would result in having to redeploy the whole monolith instead of updating a module in a microservice which would take far less time.

Scalability. This almost warranted its own section due to how impactful it is in the rise of microservices as a whole, this refers to how a microservice can be adaptive, whether this is vertical or horizontal scaling, vertical scaling refers to the power, memory, etc –usually limited to the power of a single machine - of the module whereas horizontal scaling refers to raw instances of the application. This can all be done in tooling such as Kubernetes if your application runs in Docker containers, these are not the only tools in the industry for microservice management, but they are the most widely used.

How can you scale a monolith?

If you wanted to scale a monolith even the modules which do not require additional resources will be getting them due to how monoliths lump all modules together, they scale together.

Adaptability, a monolith has an intrinsic difficulty in adapting to new languages and frameworks as it is all in one solution, built and deployed together limiting how each module can be created.

However, microservices due to being separate applications ran in containers are language-neutral, as they go on to state later in the article corroborated by [11], monoliths use language-level methods or function calls whereas microservices each service will be a process and communicate using IPC (inter-process communication) [5] for example REST API, HTTP based resource API for manipulating resources, which is what this literature review is an antecedent for.

3.4 Tooling

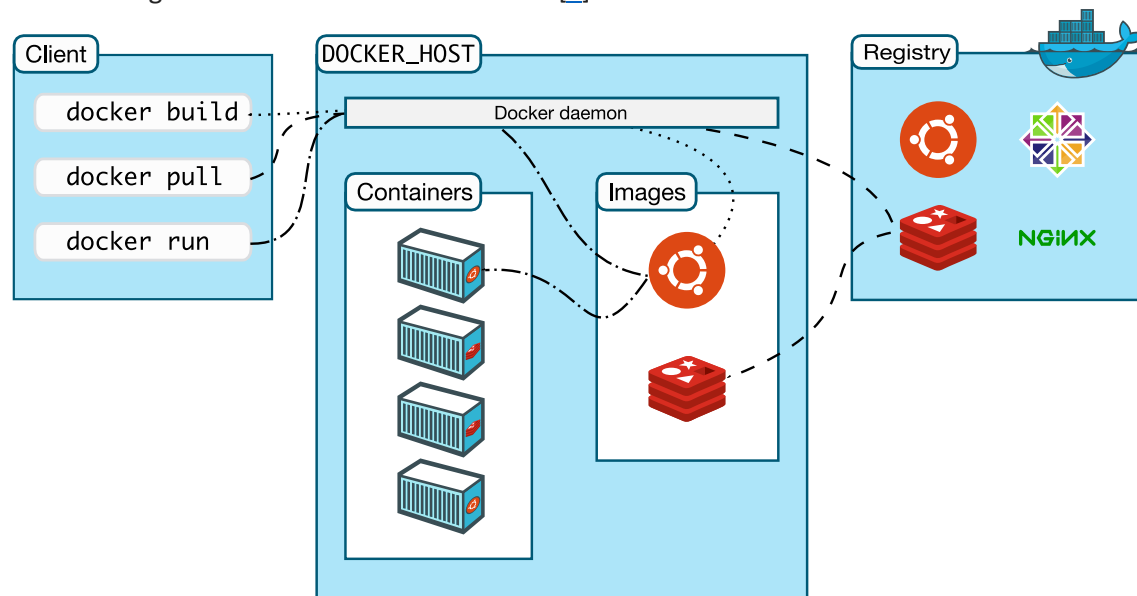
Here I will talk about the tooling and applications that help you run and manage your microservices, as outlined by Sarita and S. Sabastian [10] above, such as:

- Containers (Docker):
 - Docker Client
 - Docker Engine
 - Docker Registry
- Container orchestration (Kubernetes)
- IPC (Restful APIs)

Sarita and S. Sabastian cover Docker extensively in their paper on transforming monoliths into microservices using Docker, tooling is important in the context of this literature review as my proposal involves extensive use of Docker and other tools outlined by Sarita and S. Sabastian, I believe these tools to be a vital part in any expedient use of microservices as without them you would need to write a management system and a containerization tool for your services.

J. Thönes agrees with them on using Docker and other similar container systems as without them you would have turned a large project into a huge project for little to no reason, as these tools are open source although Kubernetes a tool I plan on using for container orchestration was first developed by Google but handed to the open-source Cloud Native Computing Foundation [6].

Sarita and S. Sabastian describe how Docker functions, they go in-depth on how Docker Engine is based on Client-Server architecture that includes: A daemon process, REST API and a CLI Client. This is a diagram taken from Dockers website [7].

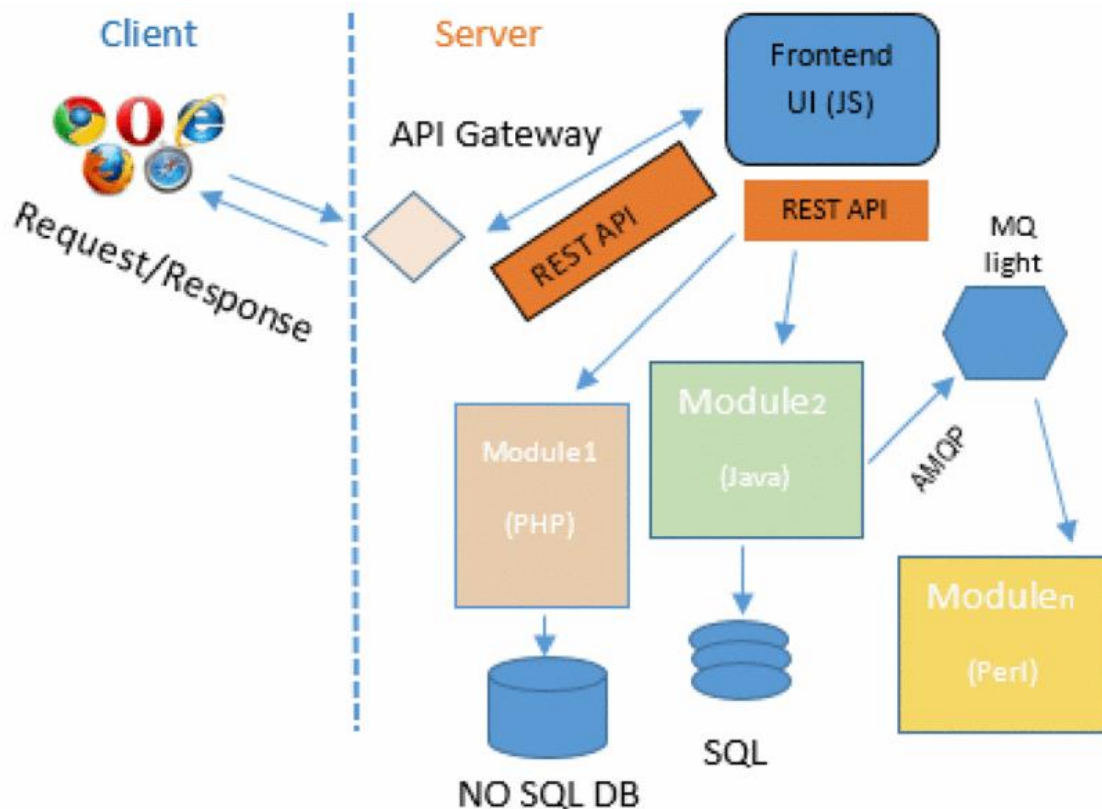


I'm going to use this diagram to explain their love of Docker for microservices, the client talks to the daemon, which builds, runs and distributes your Docker containers. The images can be compiled yourself or can be pulled from the registry, for example, a container for a MySQL database can be pulled and ran in minutes from the 'Docker Hub'.

They also go on to state how Docker can resolve challenges that you come into contact with when working with microservices; complexity, fault tolerance, etc. So, Docker isn't just a container system it is so much more.

From what I understand from Sarita and S. Sabastian, each microservice is deployed on a container these containers are linked to for a cohesive application, this application can be scaled easily by deploying one or more of specific containers, for example, if your Database is being hit a lot more than usual, for example, it's Christmas time and you are a delivery service, you can simply deploy more containers managing the Database to load balance, or as I will go into a little later you can use a piece of software called Kubernetes which is a container orchestration service which can and will load balance for you, amongst many other features

They also showed us an example of a microservice as a whole I will leave this here as I find it very informative, it also displays what and why the literature review was created to survey the viability and nature of microservices and the review itself is a precursor for a bigger project creating a RESTful API [8] also known as representational state transfer which is a software architectural style.



3.5 Advantages and Disadvantages of Microservices vs Monoliths

In this section, we will go over the compilation of issues and benefits of such a service from Jacopo Soldani, Damian Andrew Tamburri, Willem-Jan Van Den Heuvel in their lengthy and very detailed review of microservices.

In a 2014 blog post by architectural geniuses James Lewis and Martin Fowler, they pioneered a new way to develop applications against the monolith model this new model would allow scaling on a whole new level without the monolith model applications that are being used more than others can be dynamically scaled due to their decoupling and decentralization, would focus on products rather than projects, decentralize data management, automate infrastructure, designed to react to failure and finally be the future for large scale applications.

My main excerpt from the grey literature review is their compilation of 'Pains' and 'Gains' from both architectural styles.

Stage	Concern	Pain
Design	Architecture	API Versioning, Communication heterogeneity, Service contracts, Service dimensioning, Size/complexity
	Security	Access control, Centralised support, CI/CD, Endpoint proliferation, Human errors, Size/complexity
Development	Microservices	Microservice separation, Overhead

Stage	Concern	Pain
Operation	Storage	Data consistency, Distributed transactions, Heterogeneity, Query complexity
	Testing	Integration testing, Performance testing, Size/complexity
	Management	Cascading failures, Operational complexity, Service coordination, Service location
	Monitoring	Logging, Problem location, Size/complexity
	Resource consumption	Compute, Network

I've yet to find a more comprehensive list of all the issues with microservices than this, I have had to wrestle with these in the past, reading my other sources has led me to these issues as well Sarita and S. Sabastian admit that without third party software like Docker and orchestration tools that managing a microservice would be a very large task for anyone to undertake.

Integration testing is something that caught my eye in this list as I've done extensive unit testing and would imagine that testing a decoupled system would be a hassle, something I will need to watch out for if this review proves to be fruitful.

If there are 'Pains', there are inevitable 'Gains':

Stage	Concern	Gain
Design	Architecture	Bounded contexts, Cloud-native, Decentralised governance, Fault tolerance, Flexibility
	Design patterns	API gateway, Circuit breaker, Database per service, Message broker, Service discovery
	Security	Automation, Fine-grained policies, Firewalling, Isolation, Layering
Development	Microservices	CI/CD, Loose coupling, Reusability, Service size, Technology freedom
	Storage	Data persistence, Data isolation, Microservice-orientation
	Testing	Automation, Rollback, Unit testing, Updates
Operation	Deployment	Containerisation, Independency, Reliability, Speed
	Management	Fault isolation, Scalability, Update ability

I've gone into some detail about the benefits of microservices over monoliths for larger systems and despite the 'Pains' outlined in this section, this has only strengthened my belief that microservices shall eventually be the future for all larger web-based products, companies like Netflix, Amazon and Google are all cashing in on the rise of microservices either by moving their systems to the architecture, setting up a huge infrastructure to rent out hosting for systems or creating and donating industry-leading tooling for microservices.

3.6 Summary

Throughout this review, I maintained a healthy scepticism about both monoliths and microservices but as I read more and more of the articles I have gained an understanding of microservices and can see why they may be the future of all web development and content distribution systems like Netflix and Amazon, I believe that my question of whether microservices are just an industry buzzword without much weight or an industry-leading architectural style have been answered. I strongly believe after much deliberation that microservices are the way forward and as such my solution has stayed very much the same, I shall be developing a RESTful API for use within a microservice, the information garnered from this review will be insightful in what tooling and in a systematic way I shall develop such an API.

3.6.1 Changes From the Literature Review

Although I am still writing a microservice and that microservice will be deployed to the cloud, I have discovered a great tool called Rancher. I will, therefore, be using Rancher to aid in the provisioning and administration of our Kubernetes clusters.

In addition to Rancher, I have also discovered Terraform (IaC). This has allowed me to write infrastructure as code. The beauty of using Terraform is your infrastructure is detailed in code. This allows anyone to look at the code and see how you are proposing to provision your infrastructure. Therefore your documentation is in the code.

It is beginning to become clear that a grounding in DevOps is not a bad thing at all. It seems that the boundaries between developers and operations are becoming blurred. This defiantly seems the case during my research and therefore I have decided to spend more time on the infrastructure and less time on the API, however, I do intend the API to be a fully functional RESTful API and if possible fully compliant with the Level 3 Richardson maturity model:

<https://martinfowler.com/articles/richardsonMaturityModel.html>

During my time spent among MK.NET, which is a meetup group based in Milton Keynes focused on all things developer-related, some of the above has been introduced to me and has therefore changed the direction of the project.

4 Design

In this section I will discuss how the project came together after the literature review above was carried out, what changes were made to the project as the scope crept.

The scope of this project started to creep as I discovered more and more relevant and impressive technologies to do with computing in the cloud, originally the project was simply about creating a microservice but as I learnt about AWS, Terraform, Docker, Kubernetes I found it increasingly hard to justify keeping the scope of my project so small, these technologies are game-changing and as such, I needed to explore and learn more about them.

4.1 Literature Review and Research Conclusion

From the literature review, I started to become more aware of the cloud environment, the tools, what purpose they served, and the general cloud ecosystem.

From this knowledge, I slowly realised I was only scratching the surface of microservices and they could do so much more than what I originally proposed, and as such my project grew instead of making a simple microservice the project now consists of using Terraform to provision infrastructure, facilitate the installation of other software like Rancher onto its RKE (Rancher Kubernetes Engine) cluster so that it can be used to manage our microservices using a web UI and more; monitoring them with logs, charts, notifications, automatic scaling, user profiles for different levels of management, using Helm charts to single-click install applications like WordPress, DataDog, etc.

4.2 How Will This Be Implemented?

The system consists of three primary Terraform modules and one docker image,

- [Concourse](#)
- [Rancher](#)
- Walking Skeleton

4.2.1 Concourse

Concourse is a widely used CI/CD pipeline.

The project to create a Concourse installation is infrastructure-aws-concourse, This project creates a Kubernetes cluster and then installs Concourse within it using Helm.

4.2.2 Rancher

Rancher is a complete software stack for teams adopting containers. It addresses the operational and security challenges of managing multiple Kubernetes clusters across any infrastructure while providing DevOps teams with integrated tools for running containerized workloads.

(from <https://rancher.com/why-rancher/>)

4.2.3 Walking Skeleton

"A Walking Skeleton is a tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components. The architecture and the functionality can then evolve in parallel." -- Alistair Cockburn

(from http://alistair.cockburn.us/index.php/Walking_skeleton)

5 Docker

Docker is an open-source tool designed to containerise applications making it easier to create, run and deploy applications. Docker allows developers to package software complete with all its dependencies in one simple container. When the application is in a container or a Docker image ready to be deployed to a container you can guarantee it will run on any Linux machine.

Docker has been described as a Virtual Machine, however, unlike a VM, docker containers do not need a whole operating system built-in for them to run, they run on the Linux Kernel, the same one the host machine they are running on, the benefits of this are smaller application size and huge performance boost.

This project uses Docker extensively, as all the components are inside containers.

Docker is the containerisation solution, but something needs to manage this solution, or you won't be able to take full advantage of the benefits of containers; scalability, low-performance overhead, cost reduction, simplicity etc. With a system like Kubernetes, which is a container orchestration system, you have access to health checking, which checks if any container is not running correctly and will disassemble and reassemble a container with the exact specifications without the system seeing any downtime as you'd have multiple containers load balancing applications so if one goes down the others take over, automatic scaling, which will add another node/cluster/container based on the current usage of your application, this can be set so that above a certain amount of users or a certain amount of system resource usage can spin up another container to take up the additional load, useful for holidays if you're Netflix, who also use microservices.

5.1 Installation Process

Docker is installed on all the EC2 instances we create **IF** the docker flag is set to true where they are instantiated in the module's files, for example:

```
29     docker = true
```

On line 29 of the Docker EC2 example located here: [terraform-aws-ec2/examples/Docker/Main.tf](https://github.com/terraform-aws-modules/terraform-aws-ec2/tree/master/examples/Docker/Main.tf) the docker flag is set to true, and as such the EC2 instance being created here will need docker installed, once this flag is set it will generate a null_resource in Terraform similar to a function called install-docker using the [count](#) parameter, [on line 12](#) located here: [terraform-aws-ec2/docker.tf](https://github.com/terraform-aws-modules/terraform-aws-ec2/blob/master/examples/Docker/Main.tf#L12)

```
12     count = var.docker ? 1 : 0
```

Count is a parameter in Terraform which lets you scale resources by incrementing a number or since we're using a Boolean create or not create a resource.

This resource will be created and as long as it's dependencies are created before, we use the depends_on tag in Terraform to mark resources that have dependencies to other resources, the install-docker resource has 3 main parts after the dependencies and whether it should be created or not with count, these parts are:

The SSH connection information for the instance that was just created so that we can use a remote execution provisioner to run scripts on the target machine.

```
connection {
  type      = "ssh"
  host      = aws_instance.this.public_ip
  user      = "ubuntu"
  private_key = module.keys.private_key
}
```

A file provisioner which relies on the dependency `null_resource.get-shared-scripts` whose job is to download the shared scripts from my GitHub and place them into a folder called `shared-scripts`. This file provisioner takes a specific script from the `shared-scripts` folder called [install_docker.sh](#) and moves it to the current directory.

```
provisioner "file" {
  source      = "${path.cwd}/${local.shared_scripts_folder}/docker/install_docker.sh"
  destination = "install_docker.sh"
}
```

Lastly for the installation process the `remote-exec` provisioner, this provisioner runs whatever is inside the 'inline' argument for us this will be giving the install script execute permissions and running it.

```
provisioner "remote-exec" {
  inline = ["chmod +x install_docker.sh && bash install_docker.sh"]
}
```

Now that Docker is installed it's time to check if Docker is ready, the script in question is [is_docker_ready.sh](#), this script checks if docker itself is running on the machine that it was just installed onto. Firstly a file provisioner copies the file from the shared scripts folder to its directory.

```
provisioner "file" {
  source      = "${path.cwd}/${local.shared_scripts_folder}/docker/is_docker_ready.sh"
  destination = "is_docker_ready.sh"
}
```

Lastly in this process is to run the file that was just copied, in the same fashion as the last script a `remote-exec` provisioner is created except the inline is now running `is_docker_ready.sh`

```
provisioner "remote-exec" {
  inline = ["chmod +x is_docker_ready.sh && bash is_docker_ready.sh"]
}
```

Now, Docker is installed on any instance marked with `docker = true`.

5.2 Why Docker?

The justification for using Docker hardly needs to be explained if you are familiar with the technology, however if not, Docker was used in this project as we needed a lightweight containerisation system for our applications, everything in our project runs inside a docker container, Rancher, Concourse, Walking Skeleton. These may be orchestrated by Kubernetes and Rancher, but they are all inside docker containers, running inside Docker Pods, running inside Docker Nodes being managed by Kubernetes Master Node.

5.3 What We've Done With Docker

Within this project you could say Docker containers are the workforce of this project, they contain and do all the work, they even run the applications that manage them in Docker containers, and as such is where the majority of our computation comes from.

Docker is used in this project to bundle up our applications with all the prerequisites they might need, for example, the Walking Skeleton has to be compiled into a Docker image before it can be used in a container, meaning all the prerequisites like the .Net runtime and various NuGet packages are stored so that the Walking Skeleton image can be used in any container on any Linux machine, as it's already bundled with all it needs to function.

6 Kubernetes

The official answer to the question, 'what is Kubernetes?' Is "Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available."

(from <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>)

A simpler answer would be, Kubernetes is a tool for automated management of containerized applications, also known as a container orchestration tool.

[Refer to the Appendices section for the research conducted into Kubernetes.](#)

You can find information and documentation on Kubernetes at the official Kubernetes site:

<https://kubernetes.io/>

To understand Kubernetes first you need an understanding of containers.

6.1 What Are Containers?

Containers wrap software in independent, portable packages, making it easy to quickly run the software in a variety of environments.

When you wrap your software in a container, you can quickly and easily run it (almost) anywhere, that makes containers great for automation!

With containers, you can run a variety of software components across a **cluster** of generic servers. This can help ensure **high availability** and make it easier to **scale resources**.

This raises some questions:

- How can I ensure that multiple instances of a piece of software are spread across multiple servers for **high availability**?
- How can I deploy new code changes and roll them out across the entire cluster?
- How can I create new containers to handle the additional load (scale-up)?

These tasks can all be done manually, however, the answer is to use an orchestration tool as these tasks would take a lot of work, orchestration tools allow us to automate these kinds of management tasks, this is what **Kubernetes** does!

6.2 Brief Overview of Kubernetes Components

Kubernetes includes multiple components that work together to provide the functionality of a Kubernetes cluster.

The control plane (node) components manage and control the cluster:

- Etcd
 - Provides distributed, synchronised data storage for the cluster state.
- Kube-apiserver
 - Serves the Kubernetes API, the primary interface for the clusters.
- Kube-control-manager
 - Bundles several components into one package
- Kube-scheduler
 - Schedules pods to run on individual nodes.

In addition to the control plane, each node also has:

- Kubelet
 - An agent that executes containers on each node.
- Kube-proxy
 - Handles network communication between nodes by adding firewall routing rules.

[With Kubeadm](#), many of these components are run as pods within the cluster itself.

6.3 Clustering and Nodes

Kubernetes implements a clustered architecture. In a typical production environment, you will have multiple servers that can run your workloads (containers). These servers which run the containers are called nodes.

A Kubernetes cluster has one or more control servers which manage and control the cluster and host the Kubernetes API. These control servers are usually separate from worker nodes, which run applications within the cluster.

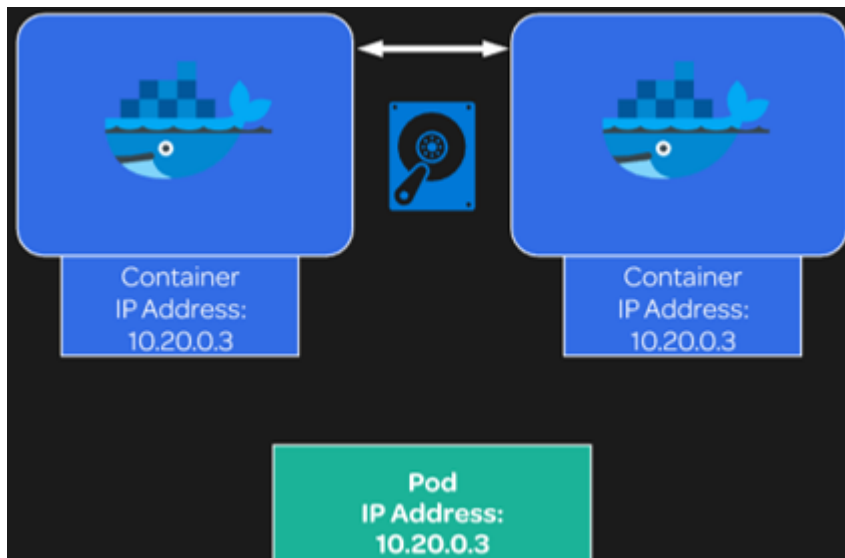
6.4 What Are Pods?

Pods are the smallest and most basic building block of the Kubernetes model.

A pod consists of one or more container(s), storage resources, and a unique IP address in the Kubernetes cluster network.

To run containers, Kubernetes schedules pods to run on servers in the cluster. When a pod is scheduled, the server will run the containers that are part of that pod.

Below is an example of what a simple Pod would look like:



As you can see this is a multi-container pod, you should notice that all addresses are the same, that is because if you want to address a pod you would use the pod IP address however if you wanted to address a specific container you would address it as localhost:port-number of the container you wish to hit.

It should be noted that all containers can communicate with all containers without NAT, and all nodes can communicate with all containers (and vice versa) without NAT.

7 Concourse

The official definition for Concourse is that it is an “open-source continuous thing-doer” and that it is built on “simple mechanics of [resources](#), [tasks](#), and [jobs](#)”.

Concourse is an automation tool that we have used as a CI/CD (Continuous Integration/Continuous Delivery) pipeline. In Concourse each job has a build plan, which is a sequence of steps to execute, these steps can be anything from running a task to configuring a pipeline.

If you have experience with CI you know that it is a valuable tool and aid to producing software inside organisations, the benefits of CI won’t just appear in the engineering/software development teams they will show themselves to the overall organisation, this includes faster code to build time, faster adding of new features, customers will receive their products faster and possibly with more features as this allows for considerably more time for developers to develop.

Concourse is a very powerful tool because of just how much you can do with it, here is an example,



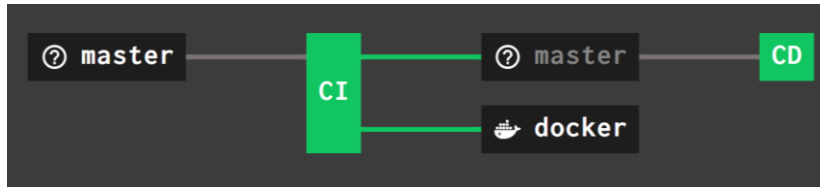
This pipeline shows a complete journey from source code to a product, using a chain of jobs and resources shown in a Concourse dependency graph, but configured with a pipeline.yaml which would look like this: [\(Link to Concourse example Booklit\)](#), in this graph in the web client you can set up any number of actions and monitor each step individually as they are happening as the dependency graph is very interactive.

A more advanced pipeline can be found here: <https://ci.concourse->

ci.org/teams/main/pipelines/concourse This shows just how far you can take concourse and how powerful of a tool it is for automation.

This pipeline shows a complete journey from source code to a product.

This is our pipeline:



A little simpler than the previous, this pipeline monitors GitHub, unfortunately my pipeline lost its icons, '? master' is our GitHub.

Once a change has been detected it will pass the source code to the CI portion of our pipeline, this will build our source code into a Docker image, this docker image once tested will be put on Docker Hub.

```
walking-skeleton / <> CI
CI #1
started 19m 36s ago
finished 17m 56s ago
duration 1m 40s
1
get: master
task: tree-source
task: build
task: tree-built
task: unit-tests
task: tree-unit-test-results
put: docker
  get: docker
```

The trees are simple commands that will print out everything in a directory, in human readable format.

This is purely here for the developer to see that it has fetched the correct source code and as an additional trouble shooting step.

```
walking-skeleton / <> CD
CD #1
started 18m 5s ago
finished 17m 40s ago
duration 25s
1
get: master
task: tree-source
```

The once that has been done the pipeline gets files from the same GitHub, these files are used in the CD portion of the pipeline, the main file will be the kube-config of the EKS cluster created before this pipeline is initiated, this EKS takes anywhere from 10-20 minutes to provision.

The files used in the pipeline can be located here: <https://github.com/jason-morsley/walking-skeleton/tree/master/Pipelines>

7.1 Redundancy

Concourse is similar to Terraform as they both use text files to store configuration information for Pipelines, Servers and Apps. If you were to lose a server or lose software you can simply destroy the environment and reload from the configuration files and you would have the same thing each time.

7.2 Pipelines

In Concourse, pipelines are all the steps that are required to go from source code from the source code repository to production, that may be a release or a production server.

As stated above Concourse pipelines consist of resources and tasks, Jobs compose resources and tasks.

The pipeline is described in a YAML file, Tasks may also be represented in the pipeline YAML file however as a good practice they are to be stored separately.

7.3 CI

Concourse is great for CI/CD automation, but what is meant by CI?

CI stands for Continuous Integration, which is the practice of integrating changes in code from multiple developers into a single repository or project. This is done automatically for most CI pipelines, they will most likely have a multitude of tests run on the merged code before they are given to the CD portion of the pipeline.

The importance of CI:

- Developers no longer have to manually coordinate when they are merging code to the source control solution.
- Replaces the communication overhead of inconvenient pre-CI environments, breaking away from the coordination and communication nightmare which only adds to the bureaucratic costs, this only grows exponentially as the size of development teams and codebase grow.
- Without a sturdy pipeline merging changes and new features can become an unknown risk, as there is no automated testing, so it may take hours or days to fully test a codebase manually, a pipeline can test, build to any form you require and send a program to be run well before any human could.

7.4 CI vs CD (Continuous Delivery) vs CD (Continuous Deployment)

These are three steps of an automated software release pipeline, firstly CI which we already know stands for Continuous Integration, this stage involves multiple developers and engineers merging their code changes to the code repository.

Secondly is CD Continuous Delivery, this step of release is responsible for packaging and delivering an artifact to end-users. This step relies on automated build tools to take source code from the CI portion and to compile it into an artifact that end-users want.

Lastly is CD Continuous Deployment, similar to the earlier step in that it delivers something to the end-user however in this step instead of delivering the artifact straight to the end users it instead automatically launches and distributes the software at the time of deployment, this means the artifact has successfully gone through the CI and previous CD stage, this then can be distributed through app stores or public repositories.

7.5 Why Concourse?

Concourse was picked over other CI tools because of just how adaptable it is, due to how Concourse abstracts nearly all of its application resources, Concourse can be adapted to almost any project, Concourse offers a wide variety of ways to implement each stage of a pipeline and as such don't force developers to change their practices and tools to suit Concourse. Making the implementation

cost negligible and if those practices and tools change because of how abstracted Concourse is, you can just adjust Concourse to suit.

8 Rancher

Rancher is the service we will be using to manage all of our Kubernetes instances, as Rancher is simply packed with useful features, analytics, graphs, heatmaps, logs, telemetry and the ability to create various pieces of software from catalogues provided inside Ranchers web GUI, these can be ranchers own catalogues or a third-party but they allow for single-click deployment of resources using Rancher, we, however, will not be using this feature of Rancher as we need this project to stay as IaC (Infrastructure as code) as well want to be able to destroy all resources and apply them again for the same result this would be impossible if we used the catalogue feature, despite how powerful it is.

It is also written in Go, which is an open-source multi-paradigm programming language it supports functional, concurrent, imperative and object-oriented, and is packaged and distributed as a Docker container.

From Ranchers website: <https://rancher.com/>, they describe Rancher as “One Platform for Kubernetes Management” And “Rancher is a complete software stack for teams adopting containers. It addresses the operational and security challenges of managing multiple Kubernetes clusters while providing DevOps teams with integrated tools for running containerized workloads.”

And from using it Rancher I can say Rancher is all of these things.

Rancher can also:

- Create, upgrade and manage Kubernetes clusters.
 - Rancher can manage clusters created by Rancher itself or that have been built outside of rancher (custom nodes) and passed to Rancher for management.
- Import existing Kubernetes clusters.
- Set up secured access to Kubernetes clusters using an authentication provider so that only those who required use and are allowed can manage certain clusters.
- Clean user interface for easy management.
- API to automate anything you want, great for use with Concourse as Concourse can use the API to alter clusters for a new release.
- First party monitoring with alerts and a very nice implementation of logging that can be deployed in any cluster.
- Integrated Pipelines.

8.1 RKE (Rancher Kubernetes Engine)

RKE is both a CNCF-certified (Cloud Native Computing Foundation, <https://www.cncf.io/>) Kubernetes distribution that runs entirely within Docker containers. RKE solves the complexity problem when installing a Kubernetes cluster. With RKE the installation of a cluster is as simple as passing RKE a YAML file containing Kubernetes cluster configuration details.

RKE is also a command-line tool for creating and managing Kubernetes clusters, as stated above passing RKE a YAML file and running ‘rke up’ will create a Kubernetes cluster to the specification in the YAML file.

9 AWS

AWS is the cloud provider we decided to use for this project, this was decided by several factors mainly at the time of writing this (03/06/2020) they are the largest provider of cloud services and are the more widely used.

[Refer to the Appendices section for the research conducted into AWS architecture.](#)

10 API

The Wikipedia definition of an API is “An application programming interface is a computing interface which defines interactions between multiple software intermediaries.”

An API is an interface between two machines, these machines can be used by humans for messaging apps or they can both be microservices, our Kubernetes installation manages our Docker containers with an API located in the master node, the API server this is the front end for the Kubernetes control plane. All API calls are sent to this server, and the server sends commands to the other services.

There are two APIs that have been developed for this project, the original Restful API, which was compliant to tier 3 the Richardson Maturity model (which is HATEOS, Hypertext as the engine of application state) for measuring API maturity, it had a multitude of features from caching, concurrency, e-tagging, result filtering and data shaping.

10.1 Walking Skeleton

And then there is the Walking Skeleton, as defined above in the Design section the walking skeleton is the minimal amount of code to have a working artifact that does not have to use final architecture but must link all together with the main architectural components with a stripped-down functionality.

This will be a small application to later flesh out, the goal of this application was to make a Restful API and add as many features as possible, however, due to time constraints the Walking Skeleton was chosen to be our primary API as refactoring the Restful API to work with microservices would take far more time than is available to me, however, this project will continue as I have taken a personal interest in its completion.

This component is the smallest but still important as without it we would only have a management system managing and provisioning infrastructure for itself, this component is a simple API which will be picked up by Concourse CI/CD pipeline from GitHub directly and will have a test suite ran against it if these tests pass the application will be compiled into a docker image and will be sent to Docker Hub (where docker images are stored) to be deployed on the EKS cluster created by Infrastructure Walking Skeleton.

10.2 Restful API

This API was developed during my third year at The University of Hertfordshire, I started working on this as soon as I knew there would be a large project at the end of the year and I wanted to make sure I had sufficient technical skill to complete such a project, once the project was announced I decided to incorporate this project and the final year project, unfortunately, time constraints wouldn't allow me to fully implement this API as a microservice and run it on the infrastructure Terraform provisioned.

A link to the API: <https://github.com/jason-morsley/JasonMorsley.Dev.Users>

However, this is still a HATEOS compliant API with a multitude of features and Swagger an API documentation tool: <https://swagger.io/> This tool allows an API to become self-documenting as it is a tool to generate and maintain API documentation as your API evolves. Swagger can generate OpenAPI definitions during runtime using Swagger Inflector, meaning even an API without a definition can be documented.

This API is RESTful (Representational state transfer) which is an architectural style for creating web services, RESTful breaks down large tasks into small tasks giving developers maximum flexibility when designing their API.

It also has caching, concurrency, E-tagging, filtering and searching, data shaping, object mapping, fault handling, support for the HEAD API call, paging, sorting collections, fully HATEOS compliant, content negotiation, and the basic CRUD to a database of users.

11 Artifact Store

An artifact is a deployable component of your application, we use a CI pipeline created in Concourse to manage our artifact, this pipeline will attempt to pull if a change is detected from our GitHub, it will then run the tests against our artifact and compile it into a Docker image, this Docker image is then stored on Docker Hub.

12 Source Control

Source Control is a vital part of any large project as has been stated many times before, offerings from GitHub and various DevOps source control tools allow you to manage a large project with multiple developers working on multiple branches simultaneously without affecting each other until the code is finally merged to master, a system like this greatly decreases the bureaucratic costs of any project, there are many reasons for this:

- Mistakes which can be rolled back to previous commits
- Branches so that multiple developers can work without affecting the work of the other, instead of all working from the main branch.
- Various permissions can be set up so developers need to go through a code review process before their changes get merged into master, test runs can also be completed using branches.
- Almost all the CI tools have some kind of GitHub interaction as it is the de facto standard for source control and as such is widely supported for multiple workflows.

There are more reasons to use source control for large software projects, however, you already know that.

13 Vault

Vault is another offering from HashiCorp that has found its way into this project.

Vault is an API wrapped around a CLI, they also offer a GUI, for management of secrets and sensitive data, for us there would be the credentials for our IAM users like Concourse and Rancher these users give the corresponding software certain permissions, and as such the Terraform projects that provision these pieces of software would need access to the IAM users public key and private key,

these would be stored on Vault, along with the Docker Hub account used to store our compiled artifact from our CI/CD pipeline.

What should you use Vault for?

- Access to tokens
- Passwords
- Certificates
- Encryption keys for protecting secrets
- PEM & PUB files

Originally vault was to be used to store all of these for the project, however, due to external factors that affected the time I had to develop this project vault was not implemented.

Vault would have served as the key store for our pipeline, it would have stored the password to Docker Hub and the IAM user credentials.

Not having Vault means the system is not as secure as I would want it to be, but as it is more of a proof of concept than a production system this is disappointing but acceptable.

14 Terraform

Terraform is the backbone of this project, without an IaC (Infrastructure as code) provider managing our infrastructure would be very difficult and very time consuming with one of the modules that was created for this project I can create an entire AWS VPC with an EC2 instance running a Linux LTS build, with all the tools installed thanks to a script run during the provisioning that installed all prerequisite tools and creates a Kubernetes cluster to run one of our applications in, in under 10 minutes. This would take operations in a DevOps environment hours to create, and even then there's no guarantee it's what you wanted, but with IaC developers can code their physical infrastructure thanks to cloud providers like AWS, this is the power of IaC.

The core to using terraform is the concept of modules, these are the building blocks of infrastructure, so what I've done is broken down the essential parts of AWS I then use the blocks to build meaningful infrastructure.

The AWS provider is essentially a wrapper around the AWS CLI, this is used to interact with the resources supported by AWS, this is the way Terraform can create resources on the AWS account passed to the Terraform provider. More information can be found on their website:

<https://www.terraform.io/docs/providers/aws/>

This infrastructure will be:

- Concourse
- Rancher
- Vault (This is an essential part of this project for security. However, due to time constraints this has gone unfinished)
- Walking Skeleton

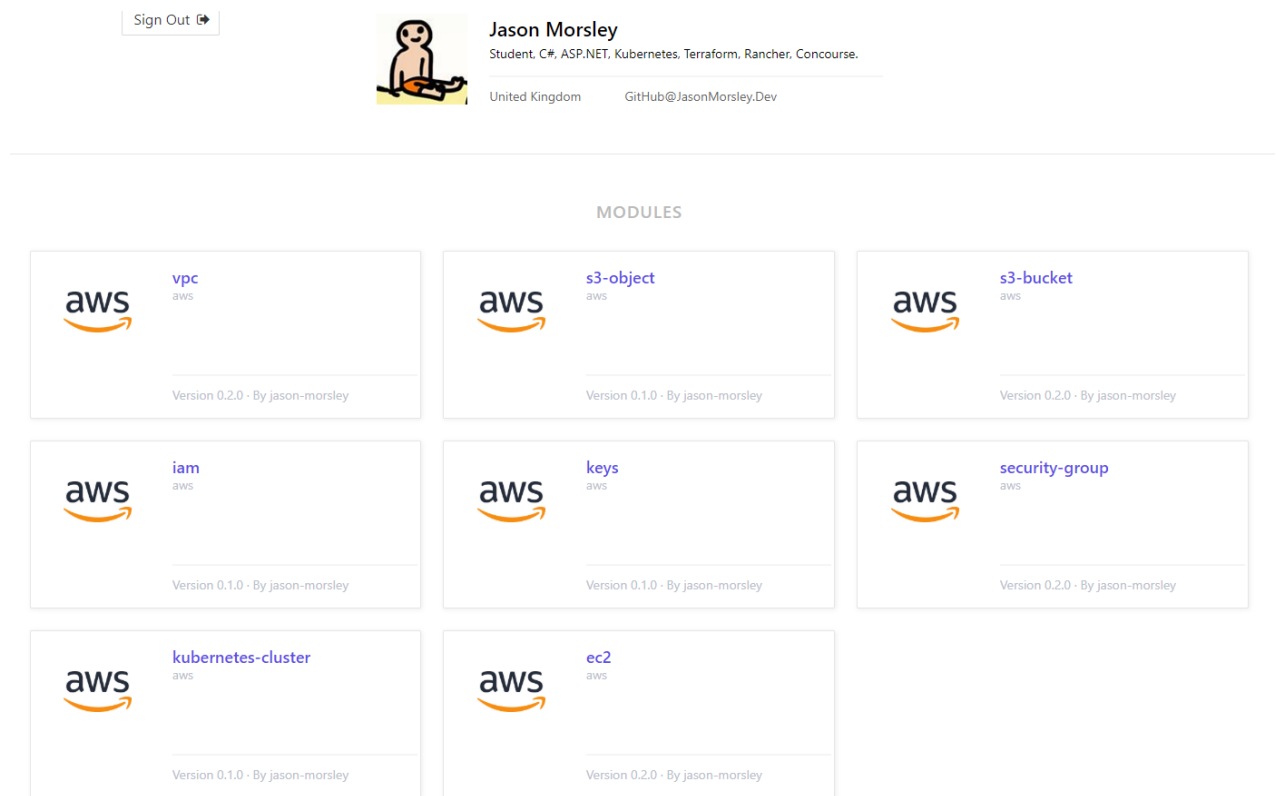
[Refer to the Appendices section for the research conducted into Terraform.](#)

[Refer to the Appendices section for the research conducted into DevOps.](#)

14.1 Terraform Modules

Terraforms official documentation defined a module as: “A module is a container for multiple resources that are used together” (From: <https://www.terraform.io/docs/modules/>)

A terraform module for us is going to be each part of the AWS infrastructure that I am planning on using, I have created 8 basic Terraform modules that are going to be heavily reused due to them being the basic building blocks of AWS:



These 8 modules hosted on my personal Terraform Registry profile can be accessed here: <https://registry.terraform.io/modules/jason-morsley> these modules are the building blocks for the entire project as to create more complex AWS components you need less complex AWS components and as such, each of these modules creates one of those AWS components.

We will use Terraform Modules as the alternatives are a lot messier and would require the more complex AWS objects to have the source code of the previous building blocks, and as such the module Kubernetes-cluster would need to have all 7 other modules packaged with it, which would make not only the file structure messy and hard to read, hard to update and hard to understand it would increase the size of these objects far beyond what they need to be and would make changes to code incredibly hard to propagate to all modules as each would need to be updated by hand, this is an example of code reuse for Terraform.

With these modules we have overcome one of many hurdles in this project, **creating the building blocks for all necessary AWS resources.**

14.2 Terraforming a Kubernetes Cluster

This function is used in later modules to print the full command to SSH into an instance, it is also used in the Kubernetes module to output the command to export the kubeconfig file which contains information about the cluster, to interact with the Kubernetes cluster you need two things, the

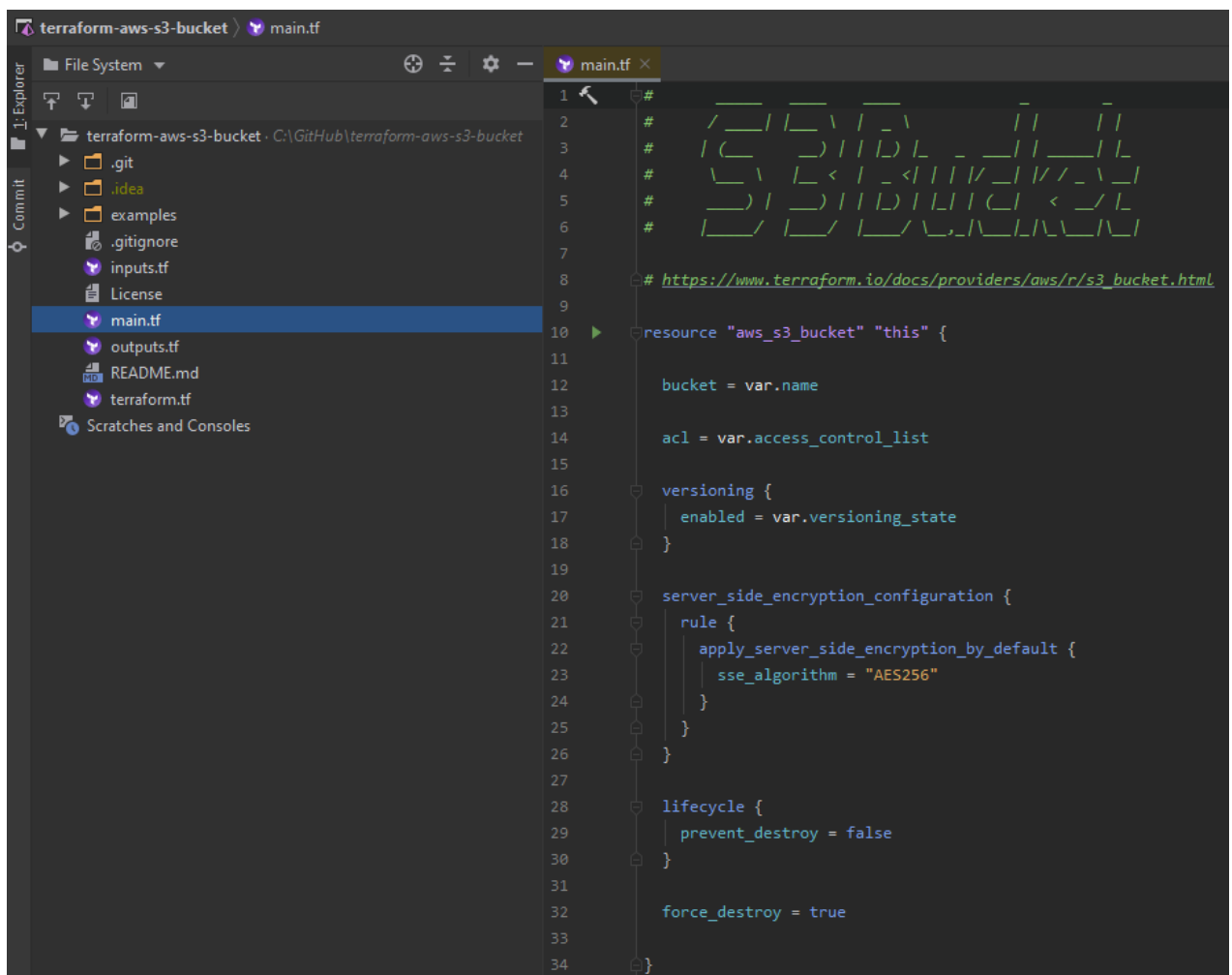
[kubectl](#) binary and the KUBECONFIG file, once you've SSH'd into the instance running Kubernetes you simply run:

```
1. kubectl --kubeconfig /path/to/kubeconfig/kube.config get pods
```

Replacing the path to the correct path of your kubeconfig and you have access to your nodes.

14.3 Terraforming an AWS environment

As an example of the Terraform modules, I will give an overview of the simplest AWS component represented by these modules, and that is an S3 Bucket, which if you refer to the [AWS Research](#) undertaken in the appendices is Amazon's Simple Storage Service hence S3, this is an object storage service.



Here you can see the basic file structure for a Terraform module, ignoring the .git and .idea folders are those are generated by Git and Rider the text editor used to create these modules. The file structure contains the root of the directory where the main and other supporting Terraform files are located.

It is important to note the naming convention of the directory, 'terraform-aws-s3-bucket' this is important as for a module to be compatible with Terraform Registry it needs to be named 'terraform-<PROVIDER>-<NAME>' AWS is our provider and we are creating an S3-Bucket, this stays

(Information from: <https://www.terraform.io/docs/registry/modules/publish.html>)

The supporting files like `inputs.tf` contain variables that are used throughout the configuration shown below:

These are variables like the name of the Bucket we are creating the ACL governing what can access the resource, it is set to private, so the owner of the resource gets full control, no one else has rights to this resource.

Lastly the tags of the resource, these are helpful as you can assign metadata to your resources to make it easier to manage, search and filter your resources, however as we are using an IaC solution I have left these tags mainly blank as we infrequently interact with the AWS Management Console.

This is the simplest module I have created for AWS, I will now show an example of a module dependant on the S3 Bucket module, this module will be the S3 Object module.

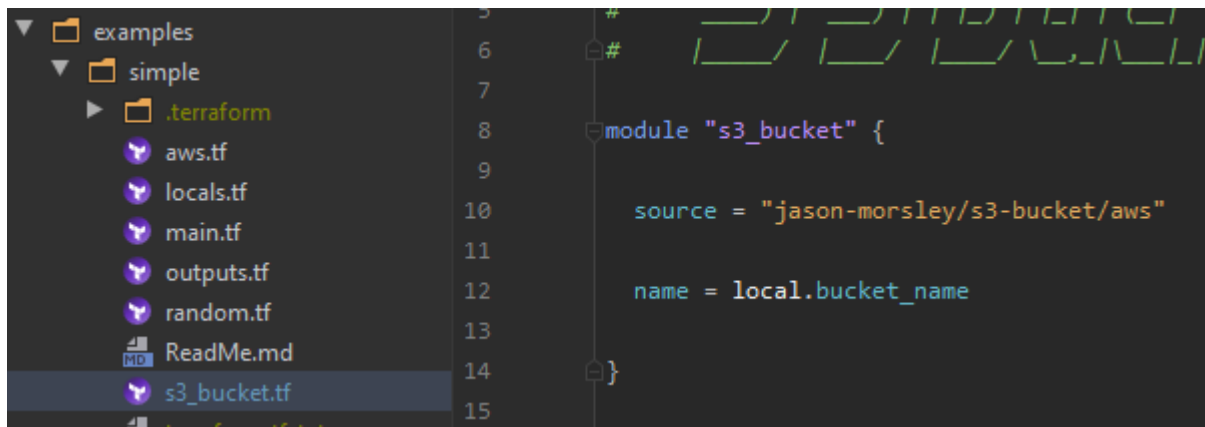


As you can see here we have another main.tf which contains the way to create an S3 object, these include the name of the bucket they will be created in, the object key and the content of the object.

A practical example of how these modules function is located in the examples folder, as you can see for this module we only have one example due to its simplicity.

Here we have a real-world example of how this object would be created, all fields are filled and there is now a new field, source this is where you specify the source code of the desired module, due to this module needing its source code to provision an object it needs to reference itself, it could

reference the Terraform Registry however that would mean if you made changes to the implementation of this module, only the code in Terraform Registry would be used.



As you can see here we are not referencing a local file due to not having the HCL './' local file identifier, we are instead referencing the Terraform Cloud user 'Jason-Morsley', modules under the cloud provider AWS, and finally the module itself, S3-Object. When you go to provision infrastructure with the Terraform CLI when you run the command 'terraform init' terraform will scan for the modules you have referenced and download them.

14.4 Terraforming Concourse

This is one of the most important sections along with the Terraforming of Rancher and the Walking Skeleton, this section is dedicated to the provisioning the infrastructure to create a Concourse instance.

Concourse is our CI/CD pipeline which will monitor GitHub for updates to our application if those are detected it will run whatever predetermined steps you wish to take, for this project it will run a test suite against the updated application and if all tests pass it will hand the artifact to the CD portion of the pipeline which will be responsible for accessing our Kubernetes node created during the Terraforming of our Walking Skeleton infrastructure, it will do this by accessing credentials (this is where Vault would be used to securely store our credentials but this was not feasible in the time frame) from our repository pushing our Docker Image to Docker Hub and then passing it to Kubernetes which will create our pod to run the artifact, our Walking Skeleton API.

The project that is responsible for this is stored in my GitHub account here:

<https://github.com/jason-morsley/infrastructure-aws-concourse> This project one of the most complicated we will be using it will be responsible for creating a Kubernetes cluster then using this cluster it will facilitate the installation of Concourse through various scripts run on both the parent machine and the child machine via a script which will install a helm chart of Concourse.

This project relies on all the building blocks in our Terraform Registry.

14.4.1 How does it work?

Terraform will create resources individually in the order they are dependent on each other, for example, an S3 object cannot be created without an S3 bucket to put the object in, and an EC2 instance cannot be created without a VPC to place it in.

Once these resources are created and the EC2 instance is running Terraform will download scripts from our source control (<https://github.com/jason-morsley/shared-scripts>) these scripts facilitate the installation of software like Docker, Kubernetes. It also has scripts to check the readiness of the

EC2 instance to see if it is ready, it also checks if Docker and Kubernetes are installed correctly and are ready to be used.

Then each project will install the software it needs to these containers managed by Docker.

This is a very high-level overview of what goes on behind the scenes, the video presentation will show the scripts running from start to finish.

14.4.2 Desired Outcome

The desired outcome of this would be for the creation of a Kubernetes cluster designed to house Concourse when it is installed via a Helm chart, this will happen once all the prerequisites are provisioned these include:

- A VPC to house our infrastructure
- An EC2 instance to run our Kubernetes cluster
- S3 buckets for storage
- Altering Route 53 A records, this is so my domain `concourse.jasonmorsley.io` points to the public IP address of our EC2 instance.
- An IAM Role using `assume_role_policy` granting an entity to assume the role, the `role_policy` itself, this policy configures permissions for the Concourse role.
- A Security Group which currently is simply an allow all for both Ingress and Egress, this needs tightening for security.
- EBS (Elastic Block Storage) for our Concourse instance, these are then passed to our cluster for the worker nodes to use as persistent storage for objects and Concourse's PostgreSQL database.

That's all the 'physical' infrastructure this creates, however after it has created all the infrastructure it needs it can start executing scripts on the EC2 instance it just created.

The beauty of Terraform is that only the scripts that aren't covered in previous modules need to be included, so our Kubernetes cluster is already created without running any scripts from the terraform Concourse project as the previous module terraform Kubernetes would have already installed Docker and Kubernetes to the EC2.

The scripts included in this project preside over the installation of Concourse (our CI/CD pipeline), Cert Manager (which is a native Kubernetes certificate management controller, capable of issuing certificates from a variety of certificate authorities) and lastly Let's Encrypt (our chosen certificate authority, this authority is free and automated more information can be found here:

<https://letsencrypt.org/>).

14.5 Terraforming Rancher

This is the project that creates our Rancher instance that manages all of our Kubernetes clusters, each cluster will be handed off to Rancher for management.

Rancher is our one-stop-shop for all things Kubernetes management, the list of features offered by this piece of software is enormous, we will be using it mainly for the management of Kubernetes clusters and visualisation of our Kubernetes infrastructure. Rancher will be able to if needed scale up instances to match demand based on pre-programmed responses to load scenarios that could mean scaling up on holidays and scaling down on weekdays etc.

The project responsible for this is stored in my GitHub here:

<https://github.com/jason-morsley/infrastructure-aws-rancher> This project is responsible like all

previous projects to create Kubernetes cluster like previous projects this cluster will be using a specific Kubernetes distribution called RKE which stands for Rancher Kubernetes Engine for more information on this distribution you can visit the Rancher RKE documentation found here:

<https://rancher.com/docs/rke/latest/en/>

This RKE cluster is designed to run entirely inside Docker containers, RKE solves the problem of installation complexity, as it can be easily automated as shown in the Kubernetes module (<https://registry.terraform.io/modules/jason-morsley/kubernetes-cluster/aws/0.1.0> OR <https://github.com/jason-morsley/terraform-aws-kubernetes-cluster>) as we have already automated this process, RKE is also operating system and platform agnostic, due to how RKE uses the Linux kernel of the machine it is running on, thus can operate without an operating system, and how RKE is designed to be generic so it can be used on any cloud provider.

14.5.1 How does it work?

The general process is identical to the previous project [Terraforming Concourse](#), the only difference is that this will install Rancher.

14.5.2 Desired Outcome

The desired outcome is the same as the previous project, that we create a Kubernetes cluster to house Rancher, the prerequisites to be provisioned are:

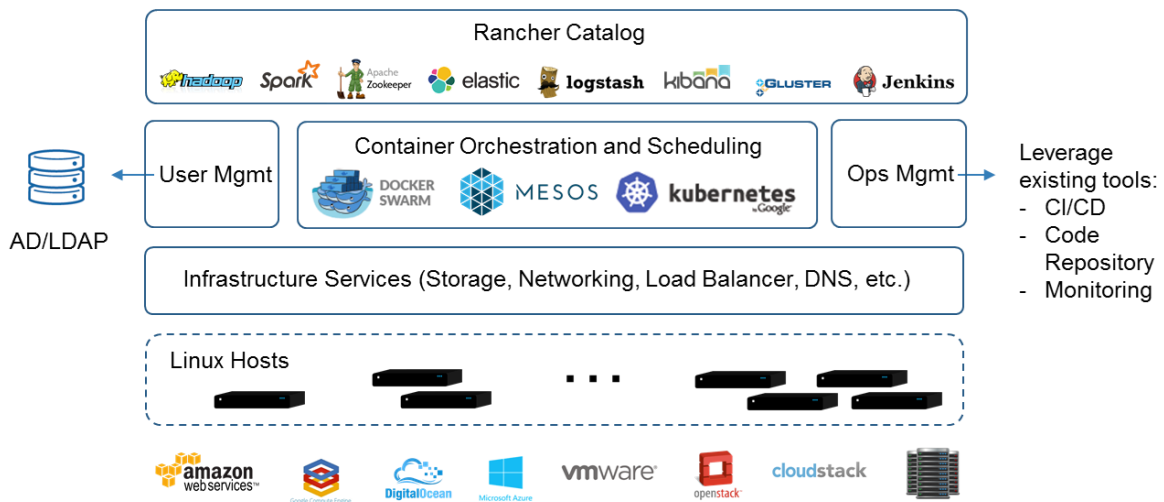
- A VPC to house our infrastructure
- The main difference between the Rancher install and the Concourse install is that Rancher is set up to run in HA (High Availability), this means that even if we lost 2 of our EC2 instances as long as we have 1 running it will route to the remaining instance of Rancher and will recreate the 2 instances that were lost.
- Network Load Balancer for our 3 EC2 instances running our Rancher HA nodes.
- The Route 53 records will need to be altered similarly, however this time the sub-domain will be rancher.jasonmorsley.io and the A record will be set to the IP of the [Network Load Balancer](#) which is one of the three types of Elastic Load Balancer.
- An IAM role for Rancher, same as before it uses the assume_role_policy to assign the policy to the Rancher user.
- A Security Group which currently is simply an allow all for both Ingress and Egress, this needs tightening for security.

After the creation of the AWS environment, the infrastructure will be used as a to create an RKE cluster this cluster is then used to install Rancher AND Cert-Manager via helm charts.

Rancher will then responsible for;

- Infrastructure Orchestration
- Container Orchestration
- [Application Catalogues](#)
- Enterprise-grade Control
- And far more.

Here is an example of what Rancher has to offer:



14.6 Terraforming Walking Skeleton

This project is by far the smallest out of the three Terraform projects, it will provision infrastructure for our Walking Skeleton to preside.

This project differs from the others due to its low complexity and provisioning of a different distribution of Kubernetes, this project will create an EKS (Amazon's Elastic Kubernetes Service) cluster which will house our Walking Skeleton artifact after it has finished running through our CI/CD pipeline.

This cluster was created using the Terraform provider Rancher2, this is similar to the AWS provider as it is a wrapper to interact with Rancher2 resources, so the management of the newly provisioned EKS cluster is in Rancher's hands, the reason we don't let Rancher create this using a Helm chart is that this is all supposed to be IaC (Infrastructure as Code) and as such if Rancher creates something and we then need to make a change to our infrastructure we would be unable to use Terraform to destroy the environment and put it up exactly as it was before, if instead of IaC we decided to use Helm it would complicate our environment and Terraform would no longer be able to manage our environment.

15 Discussion And Evaluation

The overall functionality of the project is a success it should be noted that I vastly underestimated the complexity of this project and as such, it is missing some features like credential security.

Due to how bleeding edge the technology I am using is mainly Terraform resources were tight, I had to learn these technologies enough to be able to apply them to my project, this took up hundreds of hours and due to how new they are documentation and examples were scarce and either completely wrong due to versioning or incomplete.

Before we get into the issues I had during this project, in true software fashion, things went wrong right at the end, if you've already watched my video presentation you may have noticed that the websites will be unsigned unless I can correct this before I create the video presentation, this is one of the major issues with this project I will be exploring below.

However, there are two major issues:

Firstly we do not use Vault for storing our secrets, as such I have had to store the credentials to an IAM user on GitHub this means that anyone could have access to my AWS account under that IAM

user, thus the user has been limited to only have access to S3 buckets. Vault was also going to be the store for various private keys so that the Terraform projects could be run from any machine as long as you had access to the Vault server.

Secondly, something similar can be said for our websites Certificate, I decided to go with Let's Encrypt for my Certificate signing authority, this was a mistake. Let's Encrypt is a free certificate signing authority however it imposes harsh rate limits on users and as such I have been without a valid certificate for quite a few stages in development and have had to use the website in an insecure state without a certificate if I had the chance to choose another certificate authority it would be a paid-for service as you can always rely on something you pay for.

Apart from these glaring faults the system works as intended, it takes source code from GitHub compiles it into a Docker image and eventually, that artifact is hosted inside a Kubernetes cluster that Terraform created for it, its management is also passed to Rancher. This was the primary goal for this project, for this to be a success it had to build on over 13 separate Terraform projects and a multitude of bash scripts for installing various resources and software on the instances we created in Terraform.

And as such, I feel this project has been a major success, not only does the project work from start to finish but it also showcases just how powerful IaC is, IaC is responsible for creating the entire AWS environment in a single command.

`'terraform apply --auto-approve'`

This single command (given to each project) would create the environment specified in the Terraform files, run all necessary scripts on the EC2 instances, setup Kubernetes, Docker, Concourse and Rancher, and link them all to a central domain 'jasonmorsley.io' using subdomains.

15.1 Strengths of the Project

Everything is free, apart from in specific circumstances AWS but AWS is considerably cheaper due to IaC as our infrastructure is built without fault every time, and as such, there is no playing around with instances to get them to cooperate, or human error misconfiguring the thousands of variables that go into creating such a large environment in AWS.

Free because all the major components are open source and free to use, Terraform, Rancher, Concourse, Docker, Kubernetes, all of these integral and incredibly powerful tools are widely accepted as exceptional cloud development tools, especially Docker and Kubernetes.

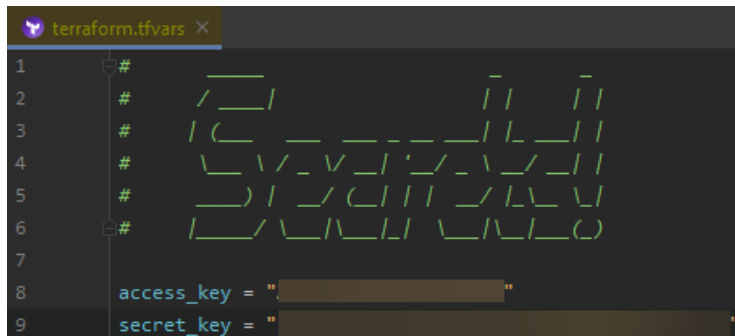
Due to IaC there will never be an error in your infrastructure, as long as it was configured correctly, as IaC will do **exactly** as you tell it to every single time, removing the human error from the equation, this makes projects cheaper, less time consuming, and most importantly less frustrating for the developers.

Rancher, this is a great tool for managing Kubernetes with a beautiful web interface and more features than I could even think to use, its analytics and monitoring are fantastic if set up correctly, the statistics and graphs it gathers and displays are very helpful, you can instantly spot if there is something wrong with one of your instances.

15.2 Weaknesses of the Project

There are a few weaknesses with this project, mainly security, without Vault we've had to pass keys as variables with our Terraform projects, this is fine as no secrets will leak as a simple addition to the

.gitignore removes all files corresponding to .tfvars which is the file type that stores secrets in Terraform.



```
1 #
2 #
3 #
4 #
5 #
6 #
7 #
8 access_key = ""
9 secret_key = ""
```

Here is an example, this file is used by Terraform on the initial Terraform apply, the secrets have been blurred out, but the format is the same.

This tfvars file holds our access and secret key for our AWS account, giving terraform the ability to Terraform infrastructure for us.

However later in the project, specifically Concourse, when the CI pipeline has finished compiling the artifact it then needs to store it in the artifact store, Docker Hub, this requires access to the Docker Hub account, and due to not having vault these details are publicly available in the GitHub for the artifact, also once this process has finished the artifact needs to be run in our Kubernetes cluster, and as such requires the kubeconfig of the cluster to be able to access the nodes and run our artifact, without Vault this kubeconfig would need to be stored somewhere where the CD pipeline can access however I have yet to find a place to store this artifact, I have tried an S3 Bucket but the CD pipeline could not access it despite me following the documentation and asking numerous questions in Slack and their Discord server, next I tried GitHub this worked but now our credentials and kubeconfig are exposed this doesn't matter for the kubeconfig but it does for the credentials.

However with Vault, our credentials would be a secure API call away, and our kubeconfig could be stored in Vault once the Kubernetes cluster was created and the kubeconfig was exported, then when the CD pipeline required the kubeconfig to setup the Walking Skeleton artifact in a cluster it could call the Vault API for the config.

15.3 Improvements for Future Work

Here I will discuss the future improvements that we're not time permitted in this particular project.

15.3.1 Vault

As stated many times before if our project had Vault it would be very secure as we would no longer need to store things in public GitHub repositories or unsecured S3 buckets it would be stored by a Vault server running on each of our Kubernetes cluster and managing our secrets. It would also make it so you could run the Terraform projects from any machine without the tfvars credential file as it would be stored in Vault.

15.3.2 RESTful API

The RESTful API would have been a very nice artifact to push through our CI/CD pipeline but due to complexity it was deemed unfit for time constraints, however, this API is packed with advanced features and is compliant to Tier 3 of the Richardson maturity model, this API will be packaged with this report as it was still created for the project and is still an impressive piece of technology.

15.3.3 Logging

We have basic logging of our Rancher managed clusters, however, these are only top-level we have not set up any logging inside the Docker containers and these logs are not exported to external storage, they are stored only inside the Rancher instance so if you lost access you would lose your logs, this is very bad practice as logs are extremely important for troubleshooting.

15.3.4 Monitoring

Rancher does include the monitoring of our Kubernetes clusters as it does with basic logging, however, this is only monitoring our clusters nothing deeper, the logging provided by Rancher is very good for managing our clusters as it generates fantastic charts and gives a good insight into what's happening with your various Kubernetes clusters, but only at surface level.

Monitoring should be applied to all instances created and run during this project, these can all be hooked up to Rancher with providers like DataDog who do very good logs too.

15.4 What was Learnt

This project taught me a great deal about the cloud, those who use it and what they use it for. During my background research, I found that companies like Netflix rely on microservices and software like EKS to manage and run their infrastructure so they can serve millions of customers high bitrate video at all times, this is because of auto-scaling and a great many other technologies that go into making the cloud so powerful.

I also learnt some industry-standard technology like Terraform and Concourse, Terraform has been great to learn as once you have a grasp on the HCL (HashiCorp Configuration Language) and how you use HCL to create IaC code, you can slowly piece together your AWS environment testing each piece of 'physical' (Most of AWS' infrastructure is virtualised, though you can rent bare-metal servers) infrastructure as you go.

Learning Concourse was equally as enlightening, I did extensive research on the DevOps culture and environment and CI/CD is a huge part of DevOps as it is the culmination of the two roles Developers and Operations coming together as one, the need for operations is becoming less and less as Developers are getting tools like Terraform to create infrastructure laid out in code, and Concourse an automation tool for almost anything, it can do a huge array of things similar to tools like Jenkins.

In Concourse you can:

- Increment versions after a successful build, so only working builds increment the version number.
- Apply release notes to builds if they reach a shipping stage.
- Take code from as many repositories as you can handle, compile the code, create a full system and release it straight to customers and or a production server without even looking at the pipeline.
- Send Slack notifications on a successful release.
- And much more...

Docker and Kubernetes are fantastic pieces of software and are both industry standard for containerising applications and managing those containers, there are many reasons to learn docker here are some that I felt were the most important:

- Dockerized apps don't need their operating system
 - They instead use the host OS, this means you only need to update one operating system to update your entire container library.
- Dockerized apps have their own set of dependencies
 - They are contained within the Docker Image, or they can be installed separately to each pod.
- Docker Hub images

- They are open source and as such are updated frequently by other community members, meaning a multitude of software packages can be installed at the click of a button or a single command.
- Automated control of Docker using software like Kubernetes.

Kubernetes is a tool to manage, deploy and scale applications, a piece of software like this can be useful in almost any web-based/cloud environment, here are some reasons I learnt and added it to this project:

- Automatic scaling based on metrics and various resource usage like CPU/RAM/GPU
- Works with any OCI (Open Container Initiative) conforming container technology so you don't need to use Docker if it is too heavy for your workload.
- Load balancing between nodes and containers
- Secrets management with software like Vault inside containers
- Can run almost any application no matter the size, and you can run as many as you want side by side they will not interfere with each other unless you told them to.
- DevOps, using the Kubernetes API you can generate any number of helpful resources like heatmaps, resource allocation, etc. All of these very useful in a DevOps environment as you can immediately see what is going wrong and where easily destroy a misbehaving cluster if it was not done automatically and rebuild it exactly as it was before from an image or Helm chart.

16 Bibliography

- [1] Wikipedia contributors. (2019, October 3). Monolithic application. In *Wikipedia, The Free Encyclopaedia*. Retrieved 21:38, November 16, 2019, from https://en.wikipedia.org/w/index.php?title=Monolithic_application&oldid=919362691
- [2] Wikipedia contributors. (2019, November 13). Microservices. In *Wikipedia, The Free Encyclopaedia*. Retrieved 21:41, November 16, 2019, from <https://en.wikipedia.org/w/index.php?title=Microservices&oldid=925946040>
- [3] Wikipedia contributors. (2019, September 24). Service-oriented architecture. In *Wikipedia, The Free Encyclopaedia*. Retrieved 21:51, November 16, 2019, from https://en.wikipedia.org/w/index.php?title=Service-oriented_architecture&oldid=917681137
- [4] Wikipedia contributors. (2019, November 15). Software as a service. In *Wikipedia, The Free Encyclopaedia*. Retrieved 22:19, November 16, 2019, from https://en.wikipedia.org/w/index.php?title=Software_as_a_service&oldid=926287816
- [5] Wikipedia contributors. (2019, November 12). Inter-process communication. In *Wikipedia, The Free Encyclopaedia*. Retrieved 23:29, November 16, 2019, from https://en.wikipedia.org/w/index.php?title=Inter-process_communication&oldid=925892338
- [6] Wikipedia contributors. (2019, November 16). Kubernetes. In *Wikipedia, The Free Encyclopaedia*. Retrieved 16:56, November 17, 2019, from <https://en.wikipedia.org/w/index.php?title=Kubernetes&oldid=926463487>
- [7] Docker. (2019). Docker Architecture. In Docker the open platform for developing, shipping and running applications. Retrieved 19:56, November 17, 2019, from <https://docs.docker.com/engine/docker-overview/>
- [8] Wikipedia contributors. (2019, November 13). Representational state transfer. In *Wikipedia, The Free Encyclopaedia*. Retrieved 17:28, November 17, 2019,

from https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=925915922

[9] J. Thönes, "Microservices," in *IEEE Software*, vol. 32, no. 1, pp. 116-116, Jan.-Feb. 2015.
DOI: 10.1109/MS.2015.11

URL:

<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7030212&isnumber=7030140>

[10] Sarita and S. Sebastian, "Transform Monolith into Microservices using Docker," 2017 *International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, Pune, 2017, pp. 1-5.

DOI: 10.1109/ICCUBEA.2017.8463820

URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8463820&isnumber=8463637>

Concourse, 2020, Concourse Main Page, viewed: 03/06/2020: <https://concourse-ci.org/>

Used extensively their documentation that was used can be accessed from this page.

Concourse, 2018, Booklit Example Pipeline, viewed: 03/06/2020:

<https://github.com/vito/booklit/blob/8741a4ca3116dcf24c30fedfa78e4aadcaff178a/ci/pipeline.yml>

Rancher, 2020, Rancher Main Page, viewed: 03/06/2020: <https://rancher.com/>

Used extensively their documentation that was used can be accessed from the company blog and under resources docs.

GitHub, Restful API: <https://github.com/jason-morsley/JasonMorsley.Dev.Users>

Pluralsight, 2020, A multitude of resources for learning various technologies this was used extensively a list of courses undertaken would be far too long so only the main website will be referenced: <https://www.pluralsight.com/>

A Cloud Guru, 2020, another website used to learn cloud-based technologies, used extensively for AWS research and IaC: <https://acloud.guru/>

Linux Academy, 2020, another website to learn cloud-based technologies, used to learn about AWS and Kubernetes: <https://linuxacademy.com/>

Terraform Registry, 2020, the location of all Terraform Modules these are publicly accessible to all who would want them: <https://registry.terraform.io/modules/jason-morsley>

17 Appendices

These are for various research tasks undertaken during this project.

The code for this entire project can be located on the GitHub links above or can be found in the Zip file included with this upload. They are included in a Zip file as there are multiple gigabytes of source code and their dependencies. The source code included in the Zip files should be disposed of after being analysed as in some cases it contains security key.

If the zip is unacceptable the full source code and all repositories can be located here:

<https://github.com/jason-morsley>

17.1 AWS Research:

17.2 MFA

MFA, Multi-factor Authentication, an additional layer of security on your root account provided by a third party, it provides a continually changing, random, six-digit code that you will need to input with your password when logging into your root account.

17.3 IAM

IAM, Identity and Access Management,

Is where you manage your AWS users and their access to AWS accounts and services.

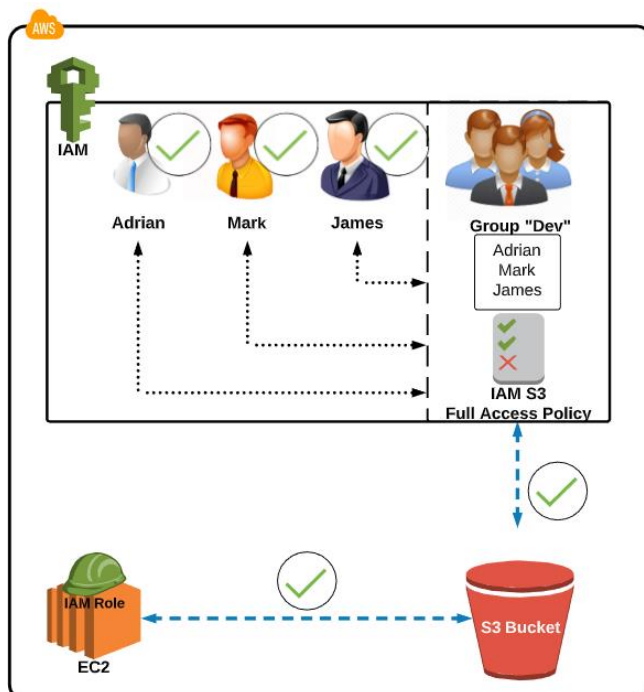
IAM is commonly used to manage:

- Users
- Groups
- IAM Access Policies
- Roles

The 'root' user is created when you create an AWS account, the root user has **FULL** administrative rights and can access every part of the account.

IAM Policies are a set of rules presiding over general use, these like S3FullAccess Policy give full access to S3 buckets, there are other permissions that give less access.

IAM Groups can be used to manage users under a blanket group, with specific policies so all members inside a group have identical policies, a user can be in 10 groups, below is how the dev group can access the S3 Bucket and how the EC2 instance can access the S3 bucket via AWS Roles.



AWS Roles, you can add a role to a service which gives it access to a different service, for example giving an EC2 instance S3FullAccess so that it can access the S3 bucket the same way the users can, shown above.

17.4 Network

AWS Regions, are a grouping of AWS resources located in a specific geographical location

Availability zones are geographically isolated zones within a region that houses AWS resources, Availability Zones (AZs) are where separate, physical AWS data centres are located, multiple AZs are in each region to provide redundancy for AWS resources in that region.

All these services and resources are running on equipment in AWS datacentres located in a availability zone which is in a region in the world.

17.4.1 VPCs

Or Virtual Private Cloud, this is a private subsection of AWS that you control, into which you can place AWS resources. You have **full** control over who has access to the AWS resources that you place inside your VPC.

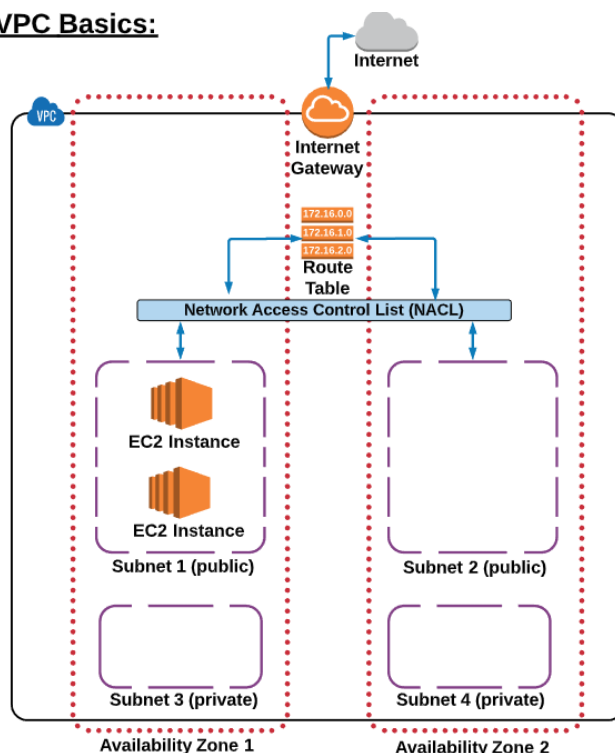
AWS Definition:

“Amazon Virtual Private Cloud (Amazon VPC) lets you provision a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that you define. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways. You can use both IPv4 and IPv6 in your VPC for secure and easy access to resources and applications.”

Standard components needed for a functional VPC:

- Internet Gateway (IGW)
- A route table (with predefined rules for access)
- A Network Access Control List (with predefined rules for access)
- Subnets to provision AWS resources (such as EC2 instances)

VPC Basics:



For example if the EC2 instance inside Subnet 1 needs to access the internet for example google.com, it first must go through our network access control list. A network access control list behaves like a virtual firewall. We must allow Port 80 traffic to communicate through our network access control lists. If the traffic is allowed, then it gets passed to our route table which determines where to send that traffic. If that traffic is destined for the internet, then it sends it to our internet gateway, and you would compare our internet gateway to the modem in your home network. So our internet gateway would then pass that traffic to the internet and off to google.com it goes. Google would then send the communication back through the internet, where it would go through our internet gateway and a route table would then

determine how to send that traffic back to our EC2 instance. Before it could go to our EC2 instance, it must go back through our network access control list. So our network access control list would decide whether it should block or allow that communication. Because we've allowed Port 80 web traffic our network access control list would then allow that communication back into our subnet where our EC2 instance would then receive that web traffic. Comparatively basic communication process is the same with a home network and a VPC, the flow of communication is very similar.

17.4.2 Internet Gateway (IGW)

A combination of hardware and software that provides your private network with a **route** to the world outside (internet) of the VPC.

AWS Definition:

“An internet gateway is a horizontally scaled, **redundant and highly available** VPC component that **allows communication between instances in your VPC and the internet**. It therefore imposes no availability risks or bandwidth constraints on your network traffic.”

The **default VPC** that was created when you registered an account with AWS has a IGW **attached** by default.

Route tables rules and details you need to know:

- Only 1 IGW can be attached to a VPC at any time.
- An IGW cannot be detached from a VPC while there are active AWS resources in the VPC such as an EC2 instance or a Database.

17.4.3 Route Table (RT)

AWS Definition:

“A route table contains a **set of rules**, called **routes**, that are used to **determine where network traffic is directed**.”

Route tables rules and details you need to know:

- Unlike an IGW, you can have multiple **active** route tables in a VPC
- You cannot delete a route table if it has **dependencies** (associated subnets)

17.4.4 Network Access Control List (NACL)

AWS Definition:

“A network access control list (NACL) is an optional layer of security for your VPC that acts as a firewall for controlling traffic in and out of one or more subnets.”

NACLs are stateless and as such require inbound and outbound rules, the default rules allow ALL traffic inbound and outbound.

Rules are processed from lower to higher number, for example rule 100 will be after rule 90.

Rule #	Type	Protocol	Port Range	Source	Allow / Deny
90	SSH (22)	TCP (6)	22	0.0.0.0/0	DENY
100	SSH (22)	TCP (6)	22	0.0.0.0/0	ALLOW

NACL rules and details you need to know:

- Rules are evaluated from lowest to highest based on rule #
- The first rule found that applies to the traffic type is immediately applied, regardless of any rules that come after it (have a higher rule #)
- The default NACL allows all traffic to the default subnets
- Any new NACLs you create DENY all traffic by default
- A subnet can only be associated with one NACL at a time
- An NACL allows or denies traffic from entering a subnet. Once inside the subnet, other AWS resources (i.e. EC2 instances) may have additional security layers (security groups)

17.4.5 Subnets

A subnet, shorthand for subnetwork, is a sub-section of a network. Generally, a subnet includes all the computers in a specific location. Circling back to the home network analogy if you think of your ISP being a network, then your home network can be considered a subnet of your ISP's network.

AWS Definition:

“When you create a VPC, it spans all of the Availability Zones in the region. After creating a VPC, you can add one or more subnets in each Availability Zone. Each subnet **must reside entirely within one Availability Zone** and **cannot span zones**.”

To have public and private subnets in AWS you will need two separate Route Tables (RTs), one RT for public which has access to the internet gateway (IGW) and another without access to the IGW.

Subnet rules and details you need to know:

- Subnets must be associated with a route table
- A public subnet has a route to the internet
- A Private subnet does not have a route to the internet
- A subnet is in one specific Availability Zone.

17.4.6 Availability Zones (VPC Specific)

Any AWS resource that you launch (like EC2) must be placed in a VPC subnet. Any given subnet must be in an Availability Zone. You can (and should) utilize multiple Availability Zones to create redundancy in your architecture. This is what allows for **High Availability** and **Fault Tolerant** systems.

AWS Definition:

“When you create a VPC, **it spans all of the Availability Zones in the region**. After creating a VPC, you can add **one or more subnets in each Availability Zone**. Each subnet must reside entirely within one Availability Zone and cannot span zones.”

Availability Zones are distinct locations that are engineered to be isolated from failures in other Availability Zones. By launching instances in separate Availability Zones, you can protect your applications from the failure of a single location.

High Availability:

Creating your architecture in such a way that your system is always available (or has the least amount of downtime as possible).

What High Availability sounds like:

- “I can always access my data in the cloud”
- “My website never crashes and is always available to my customers”

Fault Tolerant

The ability of your system to withstand failures in one (or more) of its components and remain available.

What Fault Tolerant sounds like:

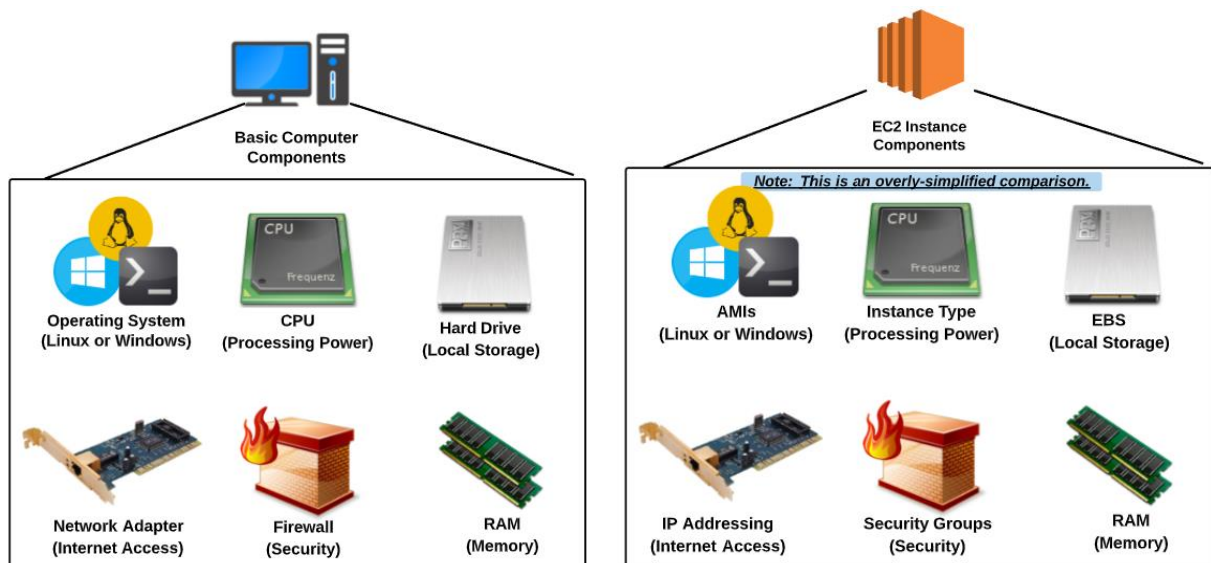
- “One of my web servers failed, but my backup server immediately took over”
- “If something in my system fails, it can repair itself.”

17.5 Elastic Compute Cloud (EC2)

EC2 is a virtual server that you can use to run applications in Amazon Web Services (AWS).

AWS Definition:

“Amazon Elastic Compute Cloud (Amazon EC2) provides **scalable computing capacity** in the AWS cloud. Using EC2 eliminates your need to invest in hardware up front, so you can develop and deploy applications faster. You can use EC2 to **launch as many or as few virtual servers as you need**, configure security and networking, and manage storage. Amazon EC2 enables you to scale up or down to handle changes in requirements or spikes in popularity, reducing your need to forecast traffic.”



You load an AMI onto an EC2 instance so you can have access to the operating system and EC2 itself, the instance type is the power of the machine for example t2.micro, EBS (Elastic block storage) is our storage, network access comes from the virtual network adapter inside our VPC which gives our EC2 instances internet access, security groups define who and what passes through to our instances and what comes back from those requests if requests are sent, lastly RAM is RAM, the amount of RAM and its speed is decided by Instance Type.

17.5.1 EC2 Instance Purchasing Options (Most Common):

17.5.2 On-demand

On-demand purchasing allows you to choose any **instance type** you like and provision/terminate it at any time (on-demand).

- is the **most expensive** purchasing option
- is the most flexible purchasing option
- You are only charged when the instance is running (and billed by the hour).
- You can provision/terminate an on-demand instance at anytime.

17.5.3 Reserved:

Reserved purchasing allows you to purchase an instance for a **set time period** of one (1) or three (3) years

- This allows for a **significant price discount** over using on-demand.
- You can select to pay upfront. partial upfront. no upfront.
- Once you buy a reserved instance you own it for the selected time period and are **responsible for the entire price** - regardless of how often you use it.

17.5.4 Spot:

Spot pricing is away for you to "**bid**" on an Instance type, then only pay for and use that instance when the spot price is **equal to or below your price**.

- This option allows Amazon to sell the use of **unused instances**, for short amounts of time at a **substantial discount**.
- **Spot prices fluctuate** based on supply and demand in the spot marketplace.
- You are **charged by the minute**.
- When you have an active bid, an instance is provisioned for you when the spot price is equal to or less than your bid price.
- A provisioned instance automatically terminates when the spot price is greater than your bid price.

Price can also vary based on **Instance Type** or processing capacity:

- General purpose
- Compute Optimized
- Accelerated Computing
- Memory optimised
- Storage optimised

AMI type:

- Linux (price varies depending on distro/software packages)
- Windows (price varies depending on version/software packages)

17.5.5 AMIs

A preconfigured package required to launch an EC2 instance including an **operating system**, software packages and other required settings.

AWS Definition:

“An Amazon Machine Image (AMI) **provides the information required to launch an instance**, which is a virtual server in the cloud. You specify an AMI when you launch an instance, and you can launch as **many instances from the AMI** as you need. You can also launch instances from as many different AMIs as you need.”

AMI Components:

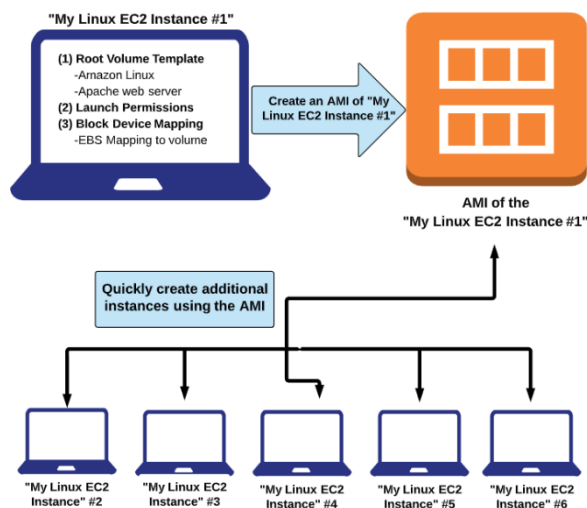
- Root Volume Template
 - Operating System
 - Application Software
- Launch Permissions
- Block Device Mappings
 - EBS (Hard drive mapping)

‘My’ Linux EC2 Instance:

- Root Volume Template
 - Amazon Linux
 - Apache Web Server
- Launch Permissions
- Block Mapping
 - EBS mapping to volume

When creating an AMI, you are essentially creating a template that you can use to launch another EC2 instance that has the exact same components as the original.

Conceptually Understanding AMIs:



Selecting an AMI:

AMIs come in three main categories:

- Community AMIs:
 - Free to use
 - Generally with these AMIs you are just selecting the OS you want.
- AWS Marketplace AMIs:
 - Pay to use
 - Generally comes packaged with additional, licensed software
- ‘My’ AMIs
 - AMIs that you create yourself

17.5.6 Instance Types

The **Instance Type** is the **CPU AND memory (RAM)**

AWS Definition:

“When you launch an instance, the **Instance Type** that you specify determines the **hardware of the host computer** used for your instance. Each instance type offers different **compute, memory and storage capabilities** and are grouped in instance families based on these capabilities. Select an instance type based on the requirements of the application or software that you plan to run on your instance.”

Instance components:

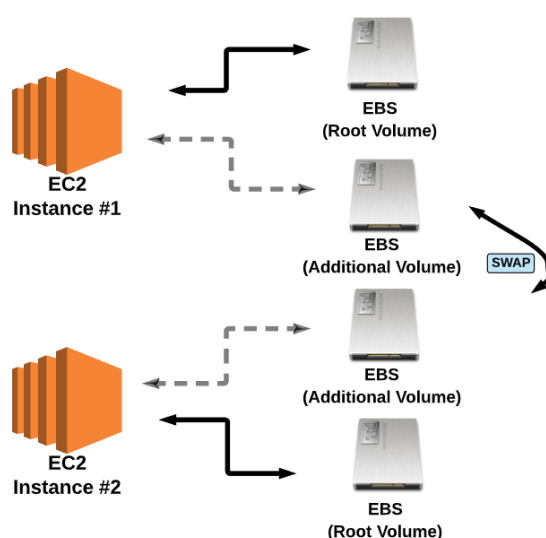
- Family
 - A way of categorising instance types based on what they are optimised to do
- Type
 - Subcategory for each family type
- vCPUs
 - The number of virtual CPUs the instance type uses
- Memory
 - The amount of RAM the instance type uses
- Instance Storage
 - The local instance storage volume
- EBS-Optimised Available
 - Indicates if EBS-Optimised is an option for the instance type
- Network Performance
 - Network performance rating based on its data transfer rate

17.5.7 Elastic Block Store (EBS)

EBS is a **storage volume** for an EC2 instance.

AWS Definition:

“Amazon EBS provides block level storage volumes for use with EC2 instances. EBS volumes are **highly available and reliable storage volumes that can be attached to any running instance that is in the same Availability Zone**. EBS volumes that are attached to an EC2 instance are exposed as **storage volumes that persist independently from the life of the instance**.”



EBS volumes can be swapped if you have more than 1 EBS per EC2, as shown in the diagram to the left.

This is done by detaching both EBS volumes and attaching them to the other EC2 instance as an additional volume.

EBS volumes can also be encrypted.

EBS volumes can be created to be independent of the EC2 instance itself, so if one of your EC2s need to be taken down you don't lose the EBS volume as it's persistent.

17.5.8 EBS Snapshots

- A snapshot is an image of an EBS volume that can be **stored as a backup** of the volume or used to **create a duplicate**.
- A snapshot is **not an active EBS volume**. You **cannot attach or detach** a snapshot to an EC2 instance.
- To **restore** a snapshot, you need to **create a new EBS volume using the snapshot as its template**.

17.5.9 IOPS

Input/output Operations Per Second

The amount of data that can be written to or retrieved from EBS per second.

AWS Definition:

“IOPS are a **unit of measure** representing input/output operations per second. The operations are measured in KiB, and the underlying drive technology determines the maximum amount of data that a volume type counts as a single I/O. I/O size is capped at 256 KiB for SSD volumes and 1,024 KiB for HDD volumes because SSD volumes handle small or random I/O much more efficiently than HDD volumes.”

What does this mean and what determines the amount of IOPS?

- More IOPS means better volume performance (faster read/writes)
- EBS volume size. The larger the storage size (in Gigabytes), the more IOPS the volume has.

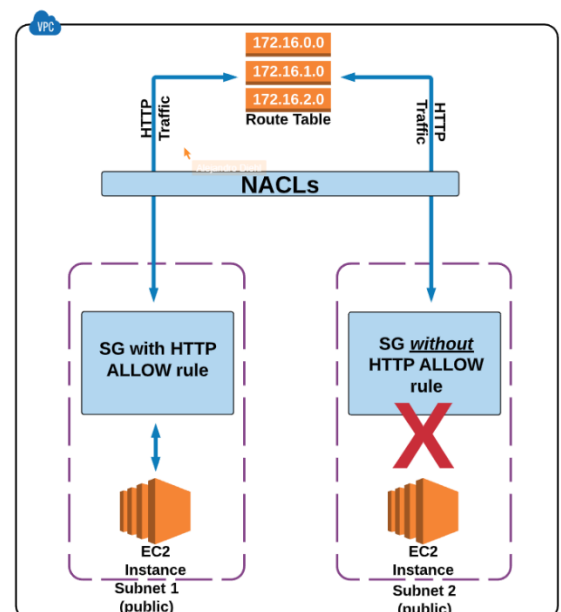
17.5.10 Security Groups

Security Groups are very similar to NACLs; they **allow or deny traffic**. However, security groups are found on the instance level (as opposed to the subnet Level). In addition, the way **allow/deny rules work are different from NACLs**

AWS Definition:

“A security group acts as a **virtual firewall that controls the traffic for one or more instances**. When you **launch an instance, you associate one or more security groups with the instance**. You add rules to each security group that allow traffic to or from its associated instances. You can modify the rules for a security group at any time; the new rules are automatically applied to all instances that are associated with the security group. When we decide whether to allow traffic to reach an instance, we evaluate all the rules from all the security groups that are associated with the instance.”

All traffic is denied unless there is an explicit ALLOW rule for it, there are no DENY rules only ALLOW rules.



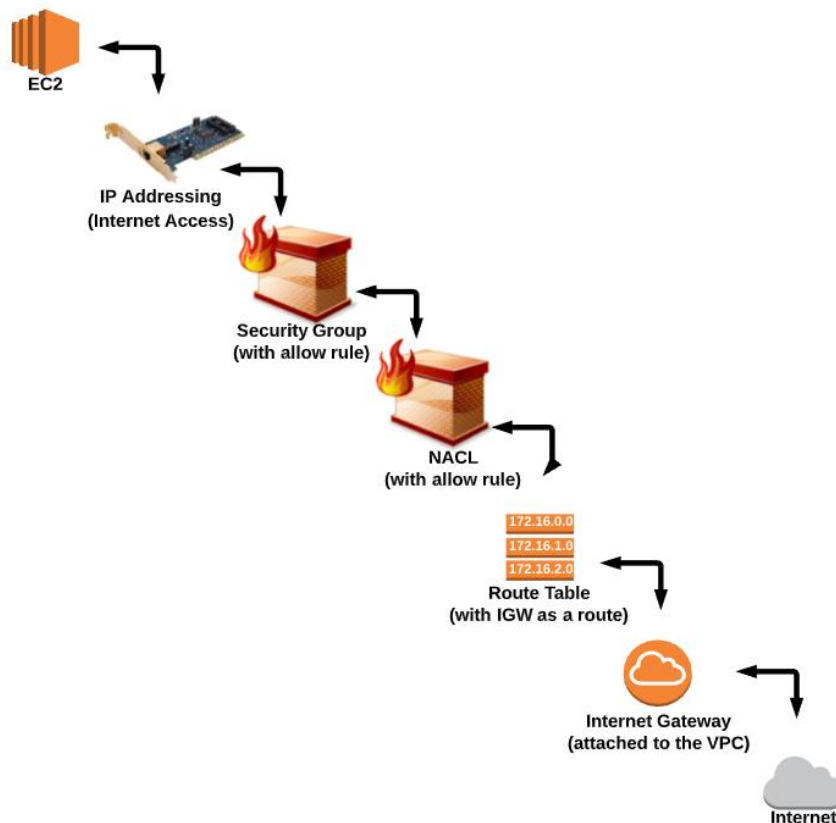
17.5.11 IP Addressing

Providing an EC2 instance with **public** IP address.

By default, all EC2 instances have a private IP address, private IP addresses allow for instances to communicate with each other, if they are in the same VPC.

EC2 instances can be launched with or without a public IP address (by default), depending on VPC/Subnet settings. Public IP addresses are required for the instance to communicate with the internet.

RECAP: Everything an EC2 instance needs to communicate with the Internet.



17.6 Auto Scaling

Auto Scaling automates the process of adding (**scaling up**) or removing (**scaling down**) EC2 instances **based on traffic demand** for your application.

AWS Definition

"Auto Scaling helps you ensure that you have the correct number of Amazon EC2 instances available to **handle the load for your application**. You create collections of EC2 instances, called **Auto Scaling groups**. You can specify the minimum number of instances in each Auto Scaling group, and Auto Scaling ensures that your group never goes below this size. You can specify the maximum number of instances in each Auto Scaling group, and Auto Scaling ensures that your group never goes above this size. If you specify the desired capacity, either when you create the group or at any time thereafter, Auto Scaling ensures that your group has this many instances. If you specify scaling

policies, then Auto Scaling can launch or terminate instances as demand on your application increases or decreases."

Components of Auto Scaling:

- **Launch Configuration** is a template used when auto scaling needs to add an additional server to your **Auto Scaling Group**.
- **Auto Scaling Group** governs when an EC2 server is automatically added or removed.

Prices:

- Auto Scaling is free to use, however
- You will be charged for the resources that Auto Scaling provisions (Any EC2 instances AWS provisions that goes beyond the free tier)

Creating an Auto Scaling Launch Configuration

(1) Select an AMI

(2) Select an Instance Type

(3) Create Launch Configuration:

- Give the Launch Configuration a name.
- Make sure that a public IP address will be assigned.
- (optional) For this demonstration we are going to include the **bash script** to install the Apache web server software:

```
#!/bin/bash
yum update -y
yum install -y httpd
service httpd start
```

(4) IP Address Type: Assign a public IP address to every instance (required if you are not using the default VPC.

(5) Select/Add storage type

(6) Configure Security Group:

- Make sure you select an SG group that has the ports open that you need.

(7) Review and Create

Creating an Auto Scaling Group:

Basic Steps:

(1) Create an Auto Scaling Group using this launch configuration:

- Select the launch configuration you want to use.

(2) Configure Auto Scaling group details:

- Give the group a name.
- Select the number of instances with which the group should start.
- Select the VPC and subnets in which you want Auto Scaling to provision instances.
- Open **Advanced Details**:
 - Check the box to include Load Balancing.
 - Select the ELB you want to use.
 - Configure the health checks.

(3) Configure scaling policies:

- Select "Use scaling policies to adjust the capacity of this group."
- Choose the MIN and MAX number of instances that Auto Scaling can scale between.
- Create an "Execute Policy" for both the "Increase" and "Decrease Group Size" sections. These are the metric thresholds that govern when Auto Scaling adds or removes instances. CPU utilization is the most commonly used metric.

(4) Configure notifications:

- Select an SNS topic to send notifications to whenever Auto Scaling launches/terminates **or** fails to launch/terminate an instance.

(5) Configure add

- Use if you like, but not required)

(6) Review and Create

17.7 Storage

17.7.1 Amazon S3 (Simple Storage Service)

An online, bulk storage service that you can access from any device.

AWS Definition:

“Amazon S3 has a simple web services interface that you can use to **store and retrieve any amount of data, at any time, from anywhere on the web**. It gives any user access to the same highly scalable, reliable, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximise benefits of scale and to pass those benefits on to the user.”

Creation of a bucket requires a name that is:

- unique across **ALL** of AWS
- 3 to 63 characters in length
- Only contain lowercase letters, numbers and hyphens
- Must be formatted as an IP address (e.g., 192.168.1.1)

Basics:

- AWS's primary storage service
- You can store any type of file in S3

Buckets:

- Root level “Folders” you create in S3 are referred to as **buckets**.
- Any “Subfolder” you create in a bucket is referred to as a **folder**.

Objects:

- Files stored in a bucket are referred to as **objects**.

Regions:

- When you create a bucket, you must select a specific region for it to exist. This means that **any data you upload to the S3 bucket will be physically located in a data centre in that region**.
- **Best practice** is to select the region that is physically **closest to you**, to **reduce transfer latency**.
- If you are serving files to a **customer** based in a certain area of the world, **create the bucket in a region closest to them**.

Storage Cost:

- Applies to data at rest in S3
- Charged per GB used
- Price per GB varies based on region and storage class

Make sure to review pricing model before going for any major use.

Storage classes:

- Standard

- Designed for general, all-purpose storage
- 99.999999999% Object durability (11 nines)
- 99.99% Object availability
- Most expensive
- Intelligent-Tiering
 - Designed for objects with changing or unknown access patterns. S3 monitors access patterns of the objects and moves the ones that have not been accessed to the infrequent tier.
 - 99.999999999% Object durability (11 nines)
 - 99.90% Object availability
 - Less expensive than standard
- Standard Infrequent Access
 - Designed for objects you do not access frequently but must be immediately available when accessed
 - 99.999999999% Object durability (11 nines)
 - 99.90% Object availability
 - Less expensive than standard
- One Zone-Infrequent Access (One Zone-IA)
 - Designed for non-critical, reproducible objects.
 - 99.999999999% Object durability (11 nines)
 - 99.90% Object availability
 - Less expensive than standard
- Glacier
 - Designed for long-term archival storage
 - May take several hours for objects stored in Glacier to be retrieved
 - 99.999999999% Object durability (11 nines)
 - Low-cost S3 storage class (very low cost)
- Glacier Deep Archive
 - Designed for long-term archival storage
 - May take several hours for objects stored in Glacier to be retrieved
 - 99.999999999% Object durability (11 nines)
 - Cheapest S3 storage class (very low cost)
- Reduced Redundancy

17.7.2 Object Durability

Is the percentage over a one-year time period that a file stored in S3 **will not be lost**.

For object durability of 99.999999999% (11 nines), that means there is a 0.000000001% change of a file being lost in a year, for example if you have 10,000 files stored in S3 (If it is 11 nines of durability) then you can expect to lose 1 file every 10 million years.

17.7.3 Object Availability

Is the percentage over a one-year period that a file stored in S3 will be accessible.

With an object availability of 99.99%, there is a 0.01% chance you won't be able to access a file stored in S3 in a year, for example for every 10,000 hours, you can expect a total of one hour for which a file may not be available to access.

17.7.4 S3 Permissions

S3 permissions are what allow you to have **granular control** over who can view, access, and use specific buckets and objects.

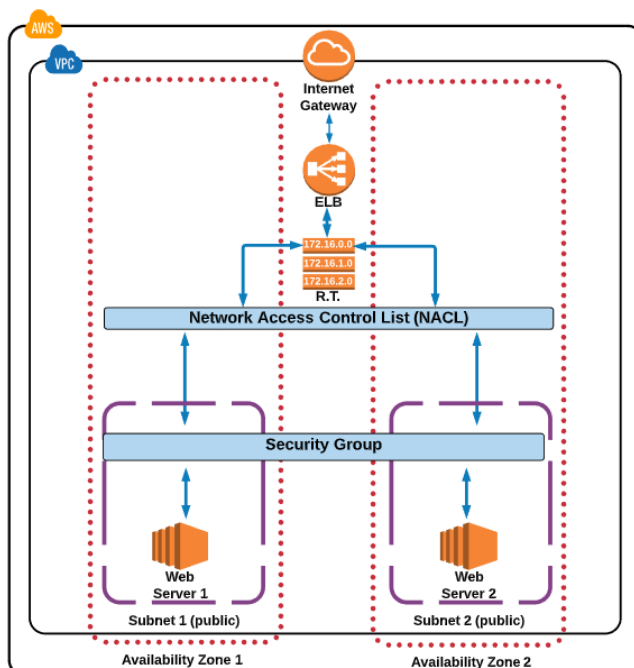
- Permissions functionality can be found on the bucket **and** object level
- On the bucket level you can control (for each bucket individually):
 - **List:** Who can see the bucket name and what objects are in the bucket
 - **Upload/Delete:** Upload and delete from the bucket
 - **Permissions:** Add/edit/delete/view
- On the object level, you can control (for each object individually):
 - **Open/Download**
 - **View Permissions**
 - **Edit Permissions**

17.8 Elastic Load Balancer (ELB)

An ELB evenly distributes traffic between EC2 instances that are associated with it.

AWS Definition:

“A load balancer **distributes application traffic across multiple EC2 instances in multiple Availability Zones**. This **increases the fault tolerance** of your applications. Elastic Load Balancing **detects unhealthy instances and routes traffic only to healthy instances.**”



The diagram shows how an ELB sorts incoming data using the route table through our NACL through the security group and to the EC2 instance, then the opposite for outgoing connections.

ELBs help servers from overloading or becoming slow and bogged down with so many connections, the ELB ensures even load balancing between two web servers for example.

17.8.1 Pricing

Free tier is NOT available for ELB

Charged for:

- Each hour or partial hour the load balancer is running.
- Each GB of data transferred through the load balancer.

Creating a Application ELB:

Basic Steps:

- (1) *In EC2 dashboard navigate to Load Balancers.*
- (2) *Create Load Balancer -> choose Create on Application Load Balancer.*
- (3) **Basic Configuration:**
 - Give the ELB a name.
 - Select the scheme **internet-facing**.
 - Leave ipv4 as the **IP address type**.
 - Leave the Listener **Load Balancer Protocol** set to HTTP
 - Leave the **Load Balancer Port** set to 80.
 - Select a minimum of two availability zones (max one subnet per availability zone)

Note: If the ELB is for serving web traffic to EC2 instances, make sure ELB/instance protocol is set to HTTP AND ELB port/instance port is set to 80. If you also want to add support for HTTPS, add another protocol for HTTPS traffic (port 443).
- (4) **Configure Security Settings:**
 - Verify that the security group has the appropriate rules set up to allow traffic based on the protocols selected in the previous step.

Note: *HTTPS requires additional settings (this is outside the scope of this course).*
- (5) **Configure Routing:**
 - Assuming a name, set **Protocol** to HTTP, **Port** to 80 and **Target type** to instance.
- (6) **Configure Health Checks:**
 - Select the **HTTP Protocol** and enter a HTTP path for the health check..
 - Alter **Advanced Details** to increase/decrease health check thresholds.
- (7) **Register Targets:**
 - Select the EC2 instances that you want the ELB to serve traffic to.

Note: *This is not required during creation; you can add instances later.*
- (8) **Add Tags:** Add tags if you wish, but it is not required.
- (9) **Review and Create!** Done!

The basic guide for creating an ELB.

17.9 Route 53

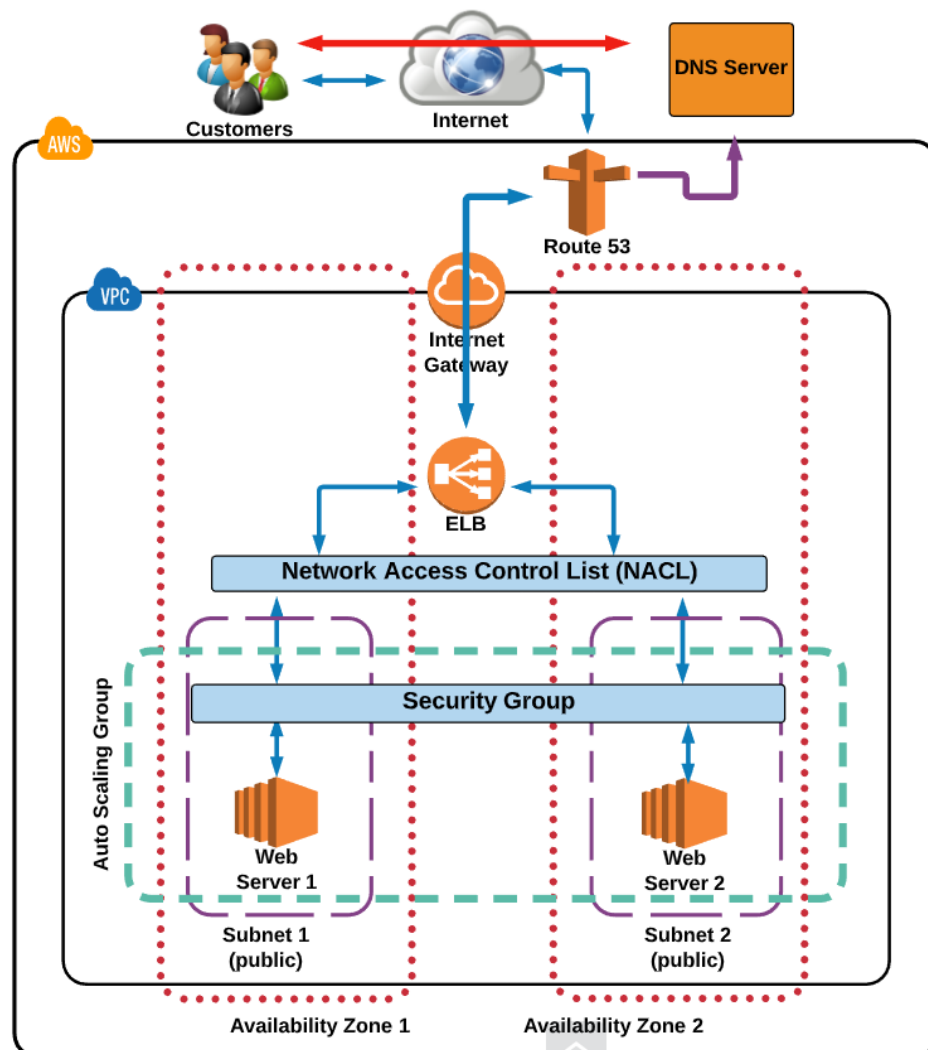
Route 53 is where you **configure and manage web domains for websites or applications you host on AWS**.

AWS Definition:

"Amazon Route 53 performs three main functions:

- **Domain registration** — Amazon Route 53 lets you register domain names such as example.com.
- **Domain Name System (DNS) service** — Amazon Route 53 translates friendly domain names like www.example.com into IP addresses like 192.0.2.1. Amazon Route 53 responds to DNS queries using a global network of authoritative DNS servers, which reduces latency.
- **Health checking** — Amazon Route 53 sends automated requests over the Internet to your application, to verify that it's reachable, available, and functional.

You can use any combination of these functions. For example, you can use Amazon Route 53 as both your registrar and your DNS service, or you can use Amazon Route 53 as the DNS service for a domain that you registered with another domain registrar."



Above is an example of how a user would hit our DNS server provided by Route 53 if you chose to use AWS for your DNS server.

17.10 Kubernetes Research:

18 Introduction

Kubernetes is a tool for automated management of containerized applications, also known as a container orchestration tool.

You can find information and documentation on Kubernetes at the official Kubernetes site:

<https://kubernetes.io/>

18.1 What are containers?

Containers wrap software in independent, portable packages, making it easy to quickly run software in a variety of environments.

When you wrap your software in a container, you can quickly and easily run it (almost) anywhere, that makes containers great for automation!

Containers are a bit like a virtual machine, but they are smaller and start up faster, this means that automation can move quickly and efficiently with containers.

With containers, you can run a variety of software components across a **cluster** of generic servers. This can help ensure **high availability** and make it easier to **scale resources**.

This raises some questions:

- How can I ensure that multiple instances of a piece of software are spread across multiple servers for **high availability**.
- How can I deploy new code changes and roll them out across the entire cluster?
- How can I create new containers to handle additional load (scale up)?

These tasks can all be done manually, however the answer is to use an orchestration tool as these tasks would take a lot of work, orchestration tools allow us to automate these kinds of management tasks, this is what **Kubernetes** does!

19 Clustering and Nodes

Kubernetes implements a clustered architecture. In a typical production environment, you will have multiple servers that are able to run your workloads (containers). These servers which run the containers are called nodes.

A Kubernetes cluster has one or more control servers which manage and control the cluster and host the Kubernetes API. These control servers are usually separate from worker nodes, which run applications within the cluster.

20 Brief Overview of Kubernetes Components

Kubernetes includes multiple components that work together to provide the functionality of a Kubernetes cluster.

The control plane (node) components manage and control the cluster:

- Etcd
 - Provides distributed, synchronised data storage for the cluster state.
- Kube-apiserver

- Serves the Kubernetes API, the primary interface for the clusters.
- Kube-control-manager
 - Bundles several components into one package
- Kube-scheduler
 - Schedules pods to run on individual nodes.

In addition to the control plane, each node also has:

- Kubelet
 - Agent that executes containers on each node.
- Kube-proxy
 - Handles network communication between nodes by adding firewall routing rules.

[With Kubeadm](#), many of these components are run as pods within the cluster itself.

21 Basic Installation Guide

The guide I am watching first disabled SELinux however they stated that this is not recommended and was just done for the sake of the quick start guide so the first 2 steps should not be followed.

how to install Kubernetes on a CentOS 7 server in our Cloud Playground. Below, you will find a list of the commands used in this lesson. ****Note Commands 1-10 need to be run on all nodes. *Note in this lesson we are using 3-unit servers as this meets the minimum requirements for the Kubernetes installation. Use of a smaller size server (less than 2 CPUs) will result in errors during installation.**

1. The first thing that we are going to do is use SSH to log in to all machines. Once we have logged in, we need to elevate privileges using `sudo`. `` sudo su ``
2. Disable SELinux. `` setenforce 0 sed -i --follow-symlinks 's/SELINUX=enforcing/SELINUX=disabled/g' /etc/sysconfig/selinux ``
3. Enable the `br_netfilter` module for cluster communication. `` modprobe br_netfilter echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables ``
4. Disable swap to prevent memory allocation issues. `` swapoff -a vim /etc/fstab.orig -> Comment out the swap line ``
5. Install the Docker prerequisites. `` yum install -y yum-utils device-mapper-persistent-data lvm2 ``
6. Add the Docker repo and install Docker. `` yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo yum install -y docker-ce ``
7. Configure the Docker Cgroup Driver to systemd, enable and start Docker `` sed -i '/^ExecStart/ s/\$/ --exec-opt native.cgroupdriver=systemd/' /usr/lib/systemd/system/docker.service systemctl daemon-reload systemctl enable docker -now systemctl status docker docker info | grep -i cgroup ``
8. Add the Kubernetes repo. `` cat < /etc/yum.repos.d/kubernetes.repo [kubernetes] name=Kubernetes baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64 enabled=1 gpgcheck=0 repo_gpgcheck=0 gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg EOF ``
9. Install Kubernetes. `` yum install -y kubelet kubeadm kubectl ``
10. Enable Kubernetes. The kubelet service will not start until you run kubeadm init. `` systemctl enable kubelet `` ****Note: Complete the following section on the MASTER ONLY!***
11. Initialize the cluster using the IP range for Flannel. `` kubeadm init --pod-network-cidr=10.244.0.0/16 ``

12. Copy the `kubeadmin join` command.
13. Exit `sudo` and run the following:

```
mkdir -p $HOME/.kube sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config sudo chown $(id -u):$(id -g) $HOME/.kube/config
```
14. Deploy Flannel.

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```
15. Check the cluster state.

```
kubectl get pods --all-namespaces
```

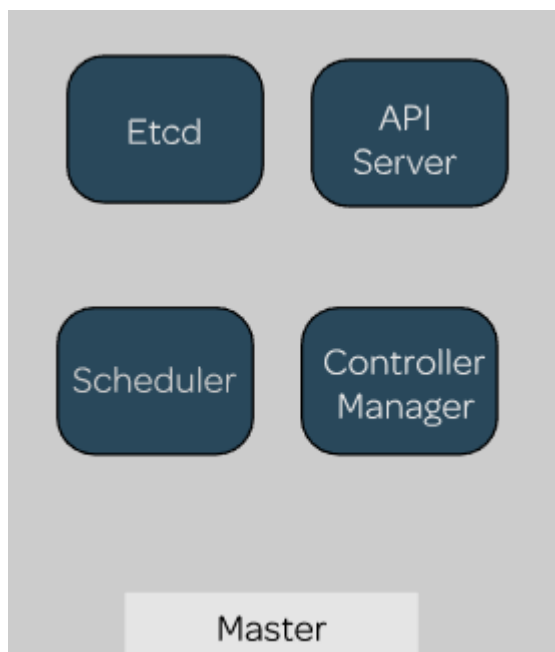
****Note: Complete the following steps on the NODES ONLY!***
16. Run the `join` command that you copied earlier (this command needs to be run as sudo), then check your nodes from the master.

```
kubectl get nodes
```

22 Masters & Nodes

The master node will look something like this:

22.1 Master Node



22.1.1 Etcd

Etcd is the key value store for the cluster. When an object is created, that object's state is stored here.

Etcd acts as the reference for the cluster state. If the cluster differs from what is indicated here, the cluster is changed to match.

22.1.2 API Server

This is what we interface with when we run the KubeCTL command.

This is the front end for the Kubernetes control plane. All API calls are sent to this server, and the server sends commands to the other services.

22.1.3 Scheduler

The scheduler watches the nodes and determines when a pod needs to be deployed and determines which node is best capable of serving that pod based on resource usage.

When a new pod is created, the scheduler determines which node the pod will be run on. This decision is based on many factors, including hardware, workloads, affinity, etc.

22.1.4 Control Manager

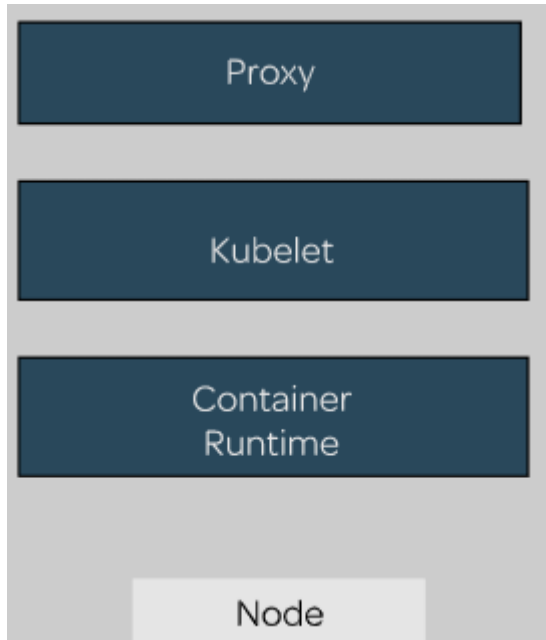
Operates the cluster controllers.

- **Node Controller:** Responsible for noticing and responding when nodes go down.
- **Replication Controller:** Responsible for maintaining the correct number of pods for every replication controller object in the system.
- **Endpoints Controller:** Populates the Endpoints object (i.e. joins services and pods).

- **Service Account & Token Controllers:** Creates default accounts and API access tokens for new namespaces.

The Node will look something like this:

22.2 Node



22.2.1 Proxy

This runs on the nodes and provides network connectivity for services on the nodes that connect to the pods. Services must be defined via the API in order to configure the proxy.

22.2.2 Kubelet - Daemon

This is the primary node agent that runs on each node. It uses a PodSpec, a provided object that describes a pod, to monitor the pods on its node. The kubelet checks the state of its pods and ensures that they match the spec.

22.2.3 Container Runtime - Daemon

This is the container manager. It can be any container runtime that is compliant with the Open Container Initiative (such as Docker). When Kubernetes needs to instantiate a container inside of

a pod, it interfaces with the container runtime to build the correct type of container.

Daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user.

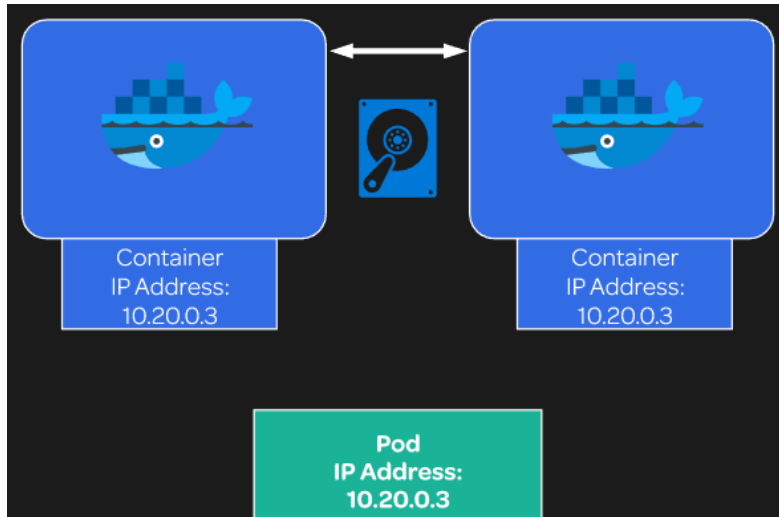
23 Pods

Pods are the smallest and most basic building block of the Kubernetes model.

A pod consists of one or more container(s), storage resources, and a unique IP address in the Kubernetes cluster network.

In order to run containers, Kubernetes schedules pods to run on servers in the cluster. When a pod is scheduled, the server will run the containers that are part of that pod.

Below is an example of what a simple Pod would look like:



The containers can communicate with themselves on localhost, it would address it as localhost:portnumber of the container.

Containers share the port space of the pod

Applications in a pod have access to shared volumes.

All containers can communicate with all other containers without NAT.

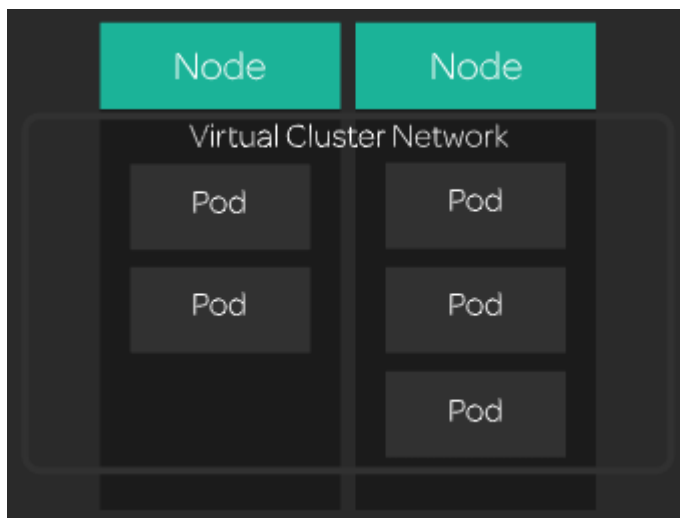
All nodes can communicate with all containers (and vice versa) without NAT.

The IP that a container sees itself as is the IP that others see it as.

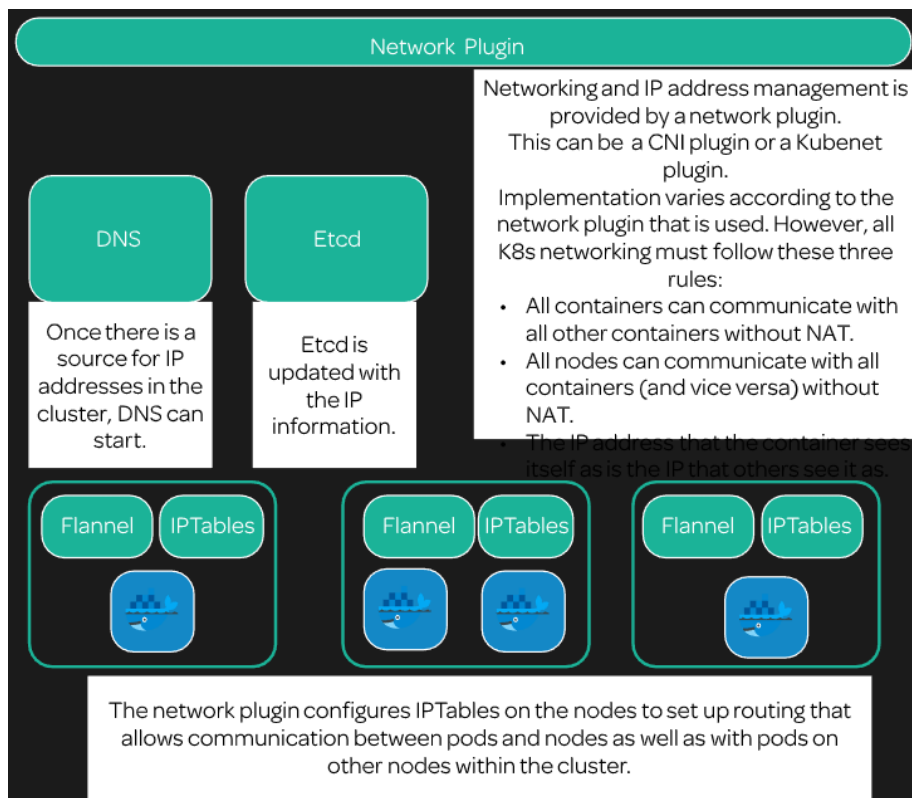
24 Networking

When using Kubernetes, is important to understand how Kubernetes implements networking between pods (and services) in the cluster.

The Kubernetes networking model involves creating a **virtual network** across the whole cluster. This means that every pod on the cluster has a unique IP address, and can communicate with any other pod in the cluster, even if that other pod is running on a different node.

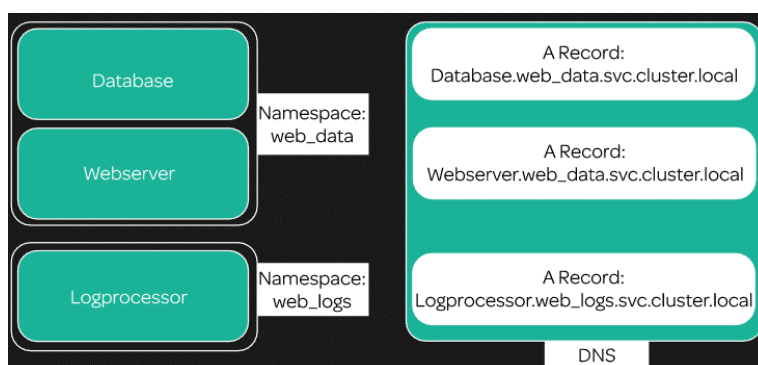


Kubernetes supports a variety of networking plugins that implement this model in various ways, from my research a nice starting network framework would be [Flannel](#).

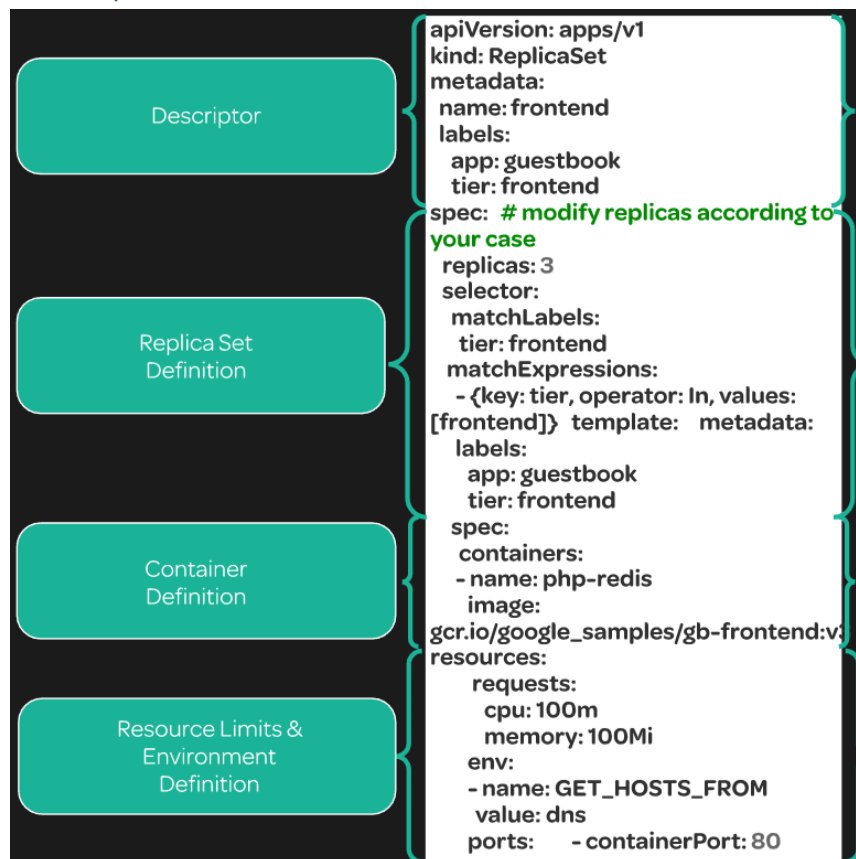


25 DNS

- All services that are defined in the cluster get a DNS record, this is true for the DNS service itself as well.
- Pods search DNS relative to their own namespace.
- The DNS server schedules a DNS pod on the cluster to configure the kubelets to set the containers to use the cluster's DNS service.
- PodSpec DNS policies determine the way that a container uses DNS. Options include Default, ClusterFirst, or None.



26 ReplicaSets



Descriptor outlines the labels so that when we invoke this ReplicaSet those pods with those labels will be controlled by the definition we give it.

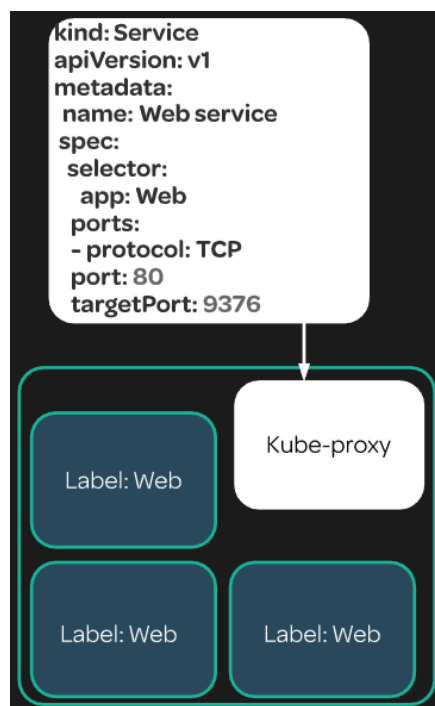
Definition how many pods we should set up for this set, this allows us to declare we need 3 pods and it will match that declaration.

Container Definition we tell it what containers should be in those pods.

Resource Limits & Environment we can limit the amount of resources these pods get and expose container ports.

<https://linuxacademy.com/cp/courses/lesson/course/3238/lesson/4>

26.1 Services



This service definition Selector matches pods with the “Web” label. It exposes port 80 and targets port 9376 on the pods.

Kube-Proxy implements a virtual IP address for the service and performs load balancing of requests, it also maps the service’s IP to the backend pod.

27 Deployment Manifest

```
Name:      nginx-deployment
Namespace:  default
CreationTimestamp:  Thu, 30 Nov 2017 10:56:25 +0000
Labels:     app=nginx
Annotations: deployment.kubernetes.io/revision=2
Selector:   app=nginx
Replicas:   3 desired | 3 updated | 3 total | 3 available | 0
             unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
    nginx:
      Image:  nginx:1.9.1
      Port:   80/TCP
      Environment: <none>
      Mounts:  <none>
      Volumes: <none>
```

Here we have a **deployment manifest**, this deployment can be used to:

- Roll out replica sets
- Declare a new state for the pods in a replica set
- Roll back to an earlier deployment
- Scale up deployments for load
- Clean up replica sets that are no longer needed

Deployments manage replica sets

Deployments can be used to **gradually remove and replace the pods** defined in a replica set that is managed by the

deployment.

When an update is required, a new replica set is created, and pods are brought up and down by the deployment controller.

Deployments define an update **strategy** and allow for rolling updates, in this case, the pods will be replaced in increments of 25% of the total number of pods. Shown under **RollingUpdateStrategy**.

Lastly is a **PodSpec** this is defined after **Pod Template** in this example. PodSpec can be updated to increment the container version or the structure of the pods that are deployed, if the PodSpec is updated and differs from the current spec, this triggers the rollout process.

27.1 Terraform Research:

Learning Terraform

“Terraform is the infrastructure as code offering from HashiCorp. It is a tool for building, changing, and managing infrastructure in a safe, repeatable way. Operators and Infrastructure teams can use Terraform to manage environments with a configuration language called the HashiCorp Configuration Language (HCL) for human-readable, automated deployments.”

<https://learn.hashicorp.com/terraform/getting-started/intro>

28 Infrastructure as Code

If you are new to infrastructure as code as a concept, it is the process of managing infrastructure in a file or files rather than manually configuring resources in a user interface. A resource in this instance is any piece of infrastructure in each environment, such as a virtual machine, security group, network interface, etc.

At a high level, Terraform allows operators to use HCL (HashiCorp Configuration Language) to author files containing definitions of their desired resources on almost any provider (AWS, GCP, GitHub, Docker, etc) and automates the creation of those resources at the time of apply.

28.1 Declarative or Imperative

Declarative programming is when you say what you want, and imperative language is when you say how to get what you want.

An example, using Tacos:

Imperative:

```
#Make me a taco  
get shell  
get beans  
get cheese  
get lettuce  
get salsa  
  
put beans in shell  
put cheese on beans  
put lettuce on cheese  
put salsa on lettuce
```



Declarative:

```
#Make me a taco  
food taco "bean-taco" {  
  ingredients = [  
    "beans", "cheese", "lettuce", "salsa"  
  ]  
}
```

Terraform is an example of a declarative approach to deploying IaaS.

28.2 Idempotence and Consistency

For example if someone asks you to make them a taco, and you were going about it Idempotently you would make them a taco, however if they ask for another one, you would tell them that they still have a taco and would not remake the taco.

Terraform is Idempotent and will not make a resource if it is already created, so if you haven't changed anything about your configuration and you apply the configuration to the environment nothing will happen to the environment because your defined configuration matches the reality of the infrastructure that exists.

29 Terraform

29.1 Terraform Components

- Terraform Executable
 - Self-contained written in Go and available for every operating system you could want to run Terraform on.
- Terraform Files

- Typically have the file extension .tf, when terraform sees multiple terraform files, terraform will take all those files and stitch a configuration together based on the contents of these files.
- Terraform Plugins
 - Used to interact with providers like AWS.
- Terraform State
 - State file contains current state of the configuration, when you want to do an update terraform will compare your current configuration with what is in the state file and then makes the necessary changes, so the state matches your desired configuration.

Terraform Variables

```
variable "aws_access_key" {}
variable "aws_secret_key" {}

variable "aws_region" {
  default = "us-east-1"
}

provider "aws" {
  access_key = "var.access_key"
  secret_key = "var.secret_key"
  region = "var.aws_region"
}

data "aws_ami" "alx" {
  most_recent = true
  owners = ["amazon"]
  filters {}
}
```

Variables:

Terraform provides you with ways of storing your information as well as many other types of information you may need in your configuration.

Providers:

In Terraform terminology AWS is a provider and it will need credentials to be defined and the region to provision those resources.

Data:

An example of how to get all the Linux AMIs on AWS so that we can select an Ami to create an EC2 Instance, this is done by pulling a data source.

Once we've set our provider we can ask that source about that provider.

```
resource "aws_instance" "ex" {
  ami = "data.aws_ami.alx.id"
  instance_type = "t2.micro"
}
```

Resources:

To create a server to host the database and web components that is called a resource, as you can see a resource takes several different arguments these can either be hard coded or

passed as a data source, we give it the AMI we got from the data source and hard coding the instance type to T2.Micro.

```
output "aws_public_ip" {
  value =
    "aws_instance.ex.public_dns"
}
```

Outputs:

If you wanted to get information outside of your deployment for example a public IP address you can use outputs.

29.2 Terraform State

Terraform must store state about your managed infrastructure and configuration. This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.

This state is stored by default in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.

Terraform uses this local state to create plans and make changes to your infrastructure. Prior to any operation, Terraform does a refresh to update the state with the real infrastructure.

The state file will be in JSON format, **do not touch!** As the state file contains the exact infrastructure that you have provisioned any changes in the state file will break terraform functionality as you will not be able to make modifications to existing infrastructure without an exact map to extrapolate needed deletions, creations and modifications of resources.

It contains resource mappings and meta data, what it is, version, serial number, other meta data and resource mappings all the information on each of those objects.

The state file may also support locking depending on where it is located.

The state file doesn't need to be saved to the host machine that ran the terraform commands, it can be stored remote in AWS, Azure, NFS, and Terraform Cloud by HashiCorp.

Terraform workspaces, each workspace within terraform has its own separate state file, so when you switch contexts between workspaces it changes which state file it is reviewing.

In this example state file there is a version, a version of the last instance of Terraform that

```
{
  "version": 4,
  "terraform_version": "0.12.5",
  "serial": 30,
  "lineage": "",
  "outputs": {},
  "resources": []
}
```

manipulated this file, this is important because if someone else goes to manipulate the state file with an old version it could damage the configuration, so you will not be allowed to make any changes if your Terraform version is lower than what is written to the state file.

Next is the serial which is incremented each time a change is made to the state file, this is useful because when you make changes to the infrastructure through Terraform it will increment this serial stopping TF plans being used to modify the state file if the

serials are incorrect, as this means something has changed in the configuration of the state file and it may be harmful to run the plan.

Lineage, unknown what this is for as HashiCorp has not outlined this clearly where I can find.

Outputs are simply output information.

Finally resources, this is a list of resources that are either generated by the configuration or data sources that pull information during the configuration.

As far as Terraform is concerned the state file is the TRUTH. And as such the first rule of using Terraform is **to make all changes in Terraform** once a configuration is deployed if you still want to use Terraform to manage that configuration, DO NOT go in manually and make changes.

29.3 Terraform Syntax

Terraform is written in Go, but the language you use to manipulate Terraform is **HasiCorp Configuration Language**, this language is Human readable, and has many of the important features of other programming languages like expressions, conditionals, functions and templates.

```
#Basic block
block_type label_one label_two {
  key = value
  embedded_block {
    key = value
  }
}
```

The first fundamental Terraform Syntax, The **Block**.

Block type, what the block is will be written here in the form of what it is and its name.

Inside the block itself will be numerous key value pairs these are based on attributes on that resource and what value you want to assign to that attribute.

You can also have embedded blocks

inside the block, these are used for additional configuration where there are multiple key value pairs that are part of a larger component, within this block you will have multiple key value pairs and you could even have another embedded block.

Below is an example of a block used to create an AWS Route Table:

```
resource "aws_route_table" "route-table" {
  vpc_id = "id928310928"
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = "id128073987"
  }
}
```

First the block type we let Terraform know we are creating a resource, of type `aws_route_table` then we give it a label to refer to, so when you want to associate this route table with a subnet you have away to refer to it.

Inside the block you have a key value pair the `vpc_id`

and the ID of the VPC this route table should be associated with.

Then we define one or more routes for this route table within an embedded block, within this embedded block we define the CIDR block that is associated with this route in this example it is the default route, then what gateway should it send the traffic out of. The route block can be embedded multiple times as a route table can have multiple routes associated with it.

29.3.1 Terraform Value Types

- string = "I am a string"
- number = 5
- bool = true
- list = ["Hello", "Hej"]
- map {name = "Jason", age = 21, loves_games = true}

29.3.2 Keyword References

- var.pizza_day
 - Terraform knows this is a variable and you don't need to let it know that this is a special type of value, it knows it's a variable based off the fact it starts with var.
- aws_instance.pizza_truck.name
 - if your string starts with a resource type like aws_instance then Terraform knows you are talking about a resource of aws_instance with the label pizza_truck if you want to get a property out of that resource you can refer to it such as name, what is the name of the aws_instance pizza_truck.
- local.pizza_toppings.vegetables
 - Local, are locally defined values within a configuration but they are defined within the context of the configuration, assuming pizza_toppings is a map type variable and you wanted to get the type of vegetables out you would use the syntax above.
- module.pizza_hut.locations
 - If you want to refer to something that is in a module that you have instantiated then you would simply type module as the keyword, name of the resource in that module and finally the property in that resource you want to access.

29.3.3 Interpolation

Pizza_name = "jasons-\${var.pizza_type}"

If you needed to create a key value pair and the value needed to be a variable of some type, the only way to refer to that variable is with the dollar sign (interpolation symbol) and a curly brace, the variable nomenclature and then a closed curly brace.

This syntax still works in Terraform but is only used for string concatenation.

29.3.4 Useful Functions

```
variable "long_key" {  
  type = "string"  
  default = <<EOF  
This is a long key.  
Running over several lines.  
EOF  
}
```

EOF – or "Here Doc" string.

Multiline strings can use shell-style "here doc" syntax, with the string starting with a marker like <<EOF and then the string ending with on a line of its own. The lines of the string and the end marker must not be indented

29.4 DevOps Research:

30 Learning DevOps

DevOps stands for Dev (Development) + Ops (Operations)

This definition from Wikipedia is a good starting point:

“DevOps is a software engineering **culture** and **practice** that aims at unifying software development (Dev) and software operation (Ops)...

DevOps aims at shorter development cycles, increased deployment frequency, more dependable releases, in close alignment with business objectives.” - Wikipedia (Feb. 2018)

DevOps **is**:

- DevOps is first a Culture of collaboration between developers and operations people
- This culture has given rise to a set of Practices
- DevOps is a grassroots movement, by practitioners, for practitioners

DevOps is **not**:

- DevOps is NOT tools, but Tools are essential to success in DevOps
- DevOps is NOT a standard
- DevOps is NOT a product
- DevOps is NOT a job title

Agile Software Development:

- DevOps grew out of the Agile software development movement
- Agile seeks to develop software in small, frequent cycles in order to deliver functionality to customers quickly and quickly respond to changing business goals
- DevOps and Agile often go together

Brief History of DevOps

- 2007: Agile software development was gaining popularity, but it was also suffering from a growing divide between development and operations.
- 2007: Patrick Debois, an engineer with experience doing both dev and ops, was doing testing on a project and became frustrated by the huge divide between dev and ops.
- 2008: Patrick Debois and Andrew Shafer met at the Agile2008 Conference in Toronto, Canada. They began to start conversations and seek others interested in bridging the divide between dev and ops.
- June 23, 2009: John Allspaw and Paul Hammond gave a talk at Velocity Conference: “10+ Deploys Per Day: Dev and Ops Cooperation at Flickr.” Patrick was watching via livestream. People began discussing it via twitter.
- October 30-31, 2009: Patrick hosted the first DevOpsDays in Ghent, Belgium; a conference for both devs and ops engineers. The conversation continued on Twitter: #devops.

DevOps grew into an organic, grassroots movement all over the world and spawned many tools to support the practices valued by DevOps.

The DevOps movement has not stopped growing since 2009, and is no longer a small, niche movement. It has since:

- Become mainstream.
- Spawned a large variety of tools.

- Completely changed the IT industry forever.

DevOps Culture

DevOps culture is about **collaboration** between Dev and Ops.

Under the traditional separation between Dev and Ops, Dev and Ops have **different** and **opposing** goals.

Development is traditionally speed and changes, providing features and changes to the customers, and Operations is traditionally stability, this is how they were viewed as opposites.

However within a DevOps culture, Dev and Ops work together and share the **same goals**.

These goals include:

- Fast time-to-market (TTM)
- Few production failures
- Immediate recovery from failures

In a DevOps culture, devs care about stability as well as speed, and ops care about speed as well as stability.

The traditional roles of developers and operational engineers can even become blurred under DevOps.

Instead of “throwing code over the wall,” dev and ops work together to create and use tools and processes that support both speed and stability.

DevOps recognizes that dev and ops are more powerful when they are together!

30.1 Traditional Silo Model

The silo model is where Developers are separated from Operations and QA Team, they are all separated by “walls”.

One of the worst parts of this model is that all three sections are a “black box” to the other groups this leads to scenarios like:

- “Our systems are fine; it’s your code!”
- “But the code works on my machine!”

Dev and Ops are black boxes to each other, which leads to finger pointing:

- Because Ops is a black box, Devs don’t really trust them
- And Ops doesn’t really trust Dev

Dev and Ops have different priorities, which pits them against each other:

- Ops views Devs as breaking stability
- Devs see ops as an obstacle to delivering their code

Even if they WANT to work together:

- Dev is measured by delivering features, which means deploying changes
- Ops is measured by uptime, but changes are bad for stability

As you can see this Silo system is inherently bad as it pits developers against operations and QA against both.

Points Against Traditional Silos:

- “Black boxes” lead to finger pointing
- Lengthy process means slow time-to-market
- Lack of automation means things like builds and deployments are inconsistent
- It takes a long time to identify and fix problems

30.2 Code Journey in DevOps:

- Devs write code
- Code commit triggers automated build, integration, and tests
- QA can get their hands on it almost immediately
- Once it is ready, kick off an automated deployment to production
- Since everything is automated, it is much easier to deploy while keeping things stable
- Deployments can occur much more frequently, getting features into the hands of customers faster
-

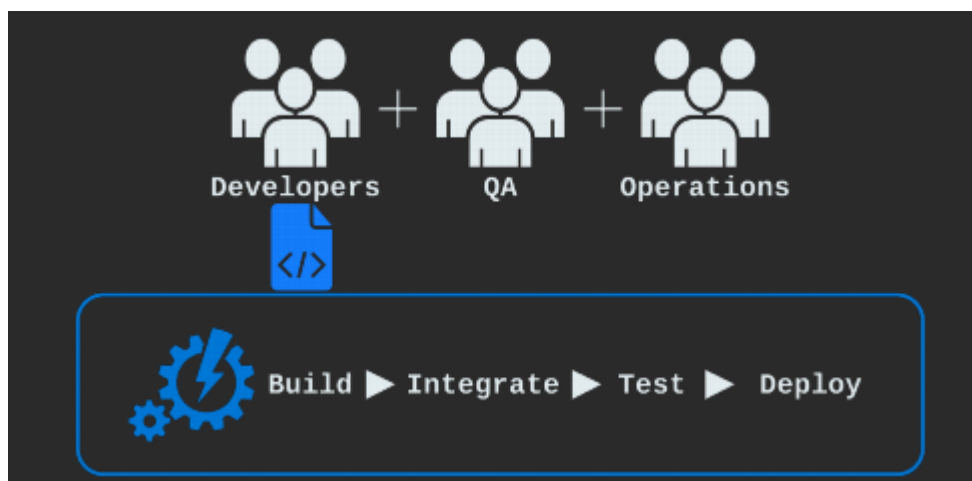


Figure 1- Image of DevOps team and automated build, integration, testing and deployment

What if something goes wrong?

- The latest deployment broke something in production!
- Fortunately, automated monitoring notified the team immediately
- The team does a rollback by deploying the previous working version, fixing the immediate problem quickly.
- An hour later, the dev team was able to deploy a fixed version of the new code

An ideal scenario for a DevOps team if something goes wrong, this will be the solution to most problems that happen within a DevOps environment.

Why DevOps:

- Automation led to consistency:
 - Building, testing, and deploying happened the same way every time
 - Building, testing, and deploying happened much more quickly and more often

- Good monitoring, plus the swift deployment process, ensured problems could be fixed even before users noticed them:
 - Dev and Ops worked together up front to build good processes
 - Even though a code change caused a problem, users experienced little or no downtime
- Dev and Ops worked together to build a robust way of changing code quickly and reliably:
 - Both Dev and Ops worked together to prioritize both speed of delivery and stability
- Happier teams:
 - Tech employees tend to be happier doing DevOps than under traditional silos
 - More time innovating and less time putting out fires
 - Devs don't feel like they must fight to get their work out there
 - Operations people don't have to fight Dev to keep the system stable
- Happier customers:
 - DevOps lets you give customers the features they want quickly
 - And you don't have to sacrifice stability to do it!

30.3 Build Automation:

automation of the process of preparing code for deployment to a live environment; Depending on what languages are used, code needs to be compiled, linted, minified, transformed, unit tested, etc.

Build automation means taking these steps and doing them in a consistent, automated way using a script or tool, the tools of build automation often differ depending on what programming languages and frameworks are used, but they have one thing in common: automation!

What does Build Automation look like?

- Usually, build automation looks like running a command-line tool that builds code using configuration files and/or scripts that are treated as part of the source code
- Build automation is independent of an IDE
- Even if you can build within the IDE, it should be able to work the same way outside of the IDE
 - This is important as it allows us to automate builds on a continuous integration server.
- As much as possible, build automation should be agnostic of the configuration of the machine that it is built on
 - This frequently has happened during the API project for I have been undertaking for my final project with the University of Hertfordshire, as Windows would consistently throw errors when scripts were running, however Linux would cause no such issues.
- Your code should be able to build on someone else's machine the same way it builds on yours

Why Automate?

- **Speed** - Build automation is fast- Automation handles tasks that would otherwise need to be done manually.
- **Consistency** - The build happens the same way every time, removing problems and confusion that can happen with manual builds.
- **Repeatable** - The build can be done multiple times with the same result. Any version of the source code can always be transformed into deployable code in a consistent way.
- **Portable** - The build can be done the same way on any machine. Anyone on the team can build on their machine, as well as on a shared build server. Building code doesn't depend on specific people or machines.
- **Reliable** - There will be fewer problems caused by bad builds.

30.4 Continuous Integration (CI):

The practice of frequently merging code changes done by developers.

Traditionally, developers would work separately, perhaps for weeks at a time, and then merge all their work together at the end in one large effort, continuous integration means merging constantly throughout the day, usually with the execution of automated tests to detect any problems caused by the merge.

Merging all the time could be a lot of work, so to avoid that it should be automated!

This will be done with the help of a CI Server, when a developer commits a code change, the CI server sees the change and automatically performs a build, also executing automated tests, this occurs multiple time a day, If anyone commits code that "breaks the build" they are responsible for

fixing the problem or rolling back their changes immediately so that other developers can continue working.

30.4.1 Why use Continuous Integration?

- **Early detection** of certain types of bugs – If code doesn't compile or an automated test fails, the developers are notified and can fix it immediately. The sooner these bugs are detected, the easier they are to fix!
- **Eliminate the scramble** to integrate just before a big release – The code is constantly merged, so there is no need to do a big merge at the end.
- Makes **frequent releases** possible - Code is always in a state that can be deployed to production.
- Makes **continuous testing** possible – Since the code can always be run, QA testers can get their hands on it all throughout the development process, not just at the end.
- Encourages **good coding practices** – Frequent commits encourages simple, modular code

30.5 Continuous Delivery (CD) And Continuous Deployment (CD):

30.5.1 Continuous Delivery (CD)

The practice of continuously maintaining code in a deployable state.

Regardless of whether the decision is made to deploy, the code is always in a state that can be deployed, some use the terms continuous delivery and continuous deployment interchangeable, or simply use the abbreviation CD.

30.5.2 Continuous Deployment (CD)

The practice of frequently deploying small code changes to production.

Continuous delivery is keeping the code in a deployable state. Continuous deployment is doing the deployment frequently, Some companies that do continuous deployment deploy to production multiple times a day, There is no standard for how often you should deploy, but in general the more often you deploy the better!

With continuous deployment, deployments to production are routine and commonplace. They are not a big, scary event.

30.5.3 What does Continuous Delivery and Continuous Deployment look like?

Each version of the code goes through a series of stages such as automated build, automated testing, and manual acceptance testing. The result of this process is an artefact or package that can be deployed.

When the decision is made to deploy, the deployment is automated. What the automated deployment looks like depends on the architecture, but no matter what the architecture is, the deployment is automated.

If a deployment causes a problem, it is quickly and reliably rolled back using an automated process (hopefully before a customer even notices the problem!)

Rollbacks aren't a big deal because the developers can redeploy a fixed version as soon as they have one available.

Finally, no one grips their desk in fear during a deployment, even if the deployment does cause a problem.

30.5.4 Why use CD and CD?

- **Faster time-to-market** – Get features into the hands of customers more quickly rather than waiting for a lengthy deployment process that doesn't happen often.
- **Fewer problems caused by the deployment process** – Since the deployment process is frequently used, any problems with the process are more easily discovered.
- **Lower risk** – The more changes are deployed at once, the higher the risk. Frequent deployments of only a few changes are less risky.
- **Reliable rollbacks** – Robust automation means rollbacks are a reliable way to ensure stability for customers, and rollbacks don't hurt developers because they can roll forward with a fix as soon as they have one.
- **Fearless deployments** – Robust automation plus the ability to roll back quickly means deployments are commonplace, everyday events rather than big, scary events.

30.6 Infrastructure as Code (IaC)

Manage and provision infrastructure through code and automation.

With IaC, instead of doing tasks manually, you can use automation to create and change:

- Servers
- Instances
- Environments
- Containers
- Other infrastructure

To make changes to instances you might need to SSH into a host machine, issue various commands to perform changes, however in IaC you would use automated tools to perform changes, you would change the code or configuration files that can be used with the automation tool, commit them to source control then use the tool to enact these changes defined in code.

30.6.1 Why IaC?

- **Consistency** in creation and management of resources – The same automation will run the same way every time.
- **Reusability** – Code can be used to make the same change consistently across multiple hosts and can be used again in the future.
- **Scalability** – Need a new instance? You can have one configured the same way as the existing instances in minutes (or seconds).
- **Self-documenting** – With IaC, changes to infrastructure document themselves to a degree. The way a server is configured can be viewed in source control, rather than being a matter of who logged in to the server and did something.
- **Simplify the complexity** – Complex infrastructures can be stood up quickly

30.7 Configuration Management

Maintaining and changing the state of pieces of infrastructure in a consistent, maintainable, and stable way. Configuration Management goes very hand in hand with IaC.

What is Configuration Management?

- Changes always need to happen – configuration management is about doing them in a maintainable way

- Configuration management allows you to minimize **configuration drift** – the small changes that accumulate over time and make systems different from one another and harder to manage
- Infrastructure as Code is very beneficial for configuration management

An example of why Configuration Management is so important:

- **Without good configuration management**, you log into each server and perform the upgrade. However, this can lead to a lot of problems. Perhaps one server was missed due to poor documentation, or perhaps something doesn't work while the versions are temporarily mismatched between servers, causing a lot of downtime while you do the upgrade.
- With **good configuration management**, you define the new version of the software package in a configuration file or tool and automatically roll out the change to all of the servers.

Configuration management is about managing your configuration somewhere outside of the servers themselves

Why use Configuration Management?

- **Save time** – It takes less time to change the configuration.
- **Insight** – With good configuration management, you can know about the state of all pieces of a large and complex infrastructure.
- **Maintainability** - A more maintainable infrastructure is easier to change in a stable way.
- **Less configuration drift** – It is easier to keep a standard configuration across a multitude of hosts.

30.8 Orchestration

Automation that supports processes and workflows, such as provisioning resources, this is how Netflix handles their resource provisioning, as load and usage change during the day.

With orchestration, managing a complex infrastructure is less **like being a builder** and more like **conducting an orchestra**, instead of going out and creating a piece of infrastructure, the conductor simply signals what needs to be done and the orchestra performs it:

- The conductor does not need to control every detail
- The musicians (automation) can perform their piece with only a little bit of guidance

Here is an example without an Orchestration tool:

- A customer requests more resources for a web service that is about to see a heavy increase in usage due to a planned marketing effort
- Instead of manually standing up new nodes, operations engineers use an orchestration tool to request five more nodes to support the service
- A few minutes later, the tool has five new nodes are up and running

With an Orchestration tool:

- A monitoring tool detects an increased load on the service
- An orchestration tool responds to this by spinning up additional resources to handle the load
- When the load decreases again, the tool spins the additional resources back down, freeing them up to be used by something else
- All of this happens while the engineer is getting coffee

Why do Orchestration?

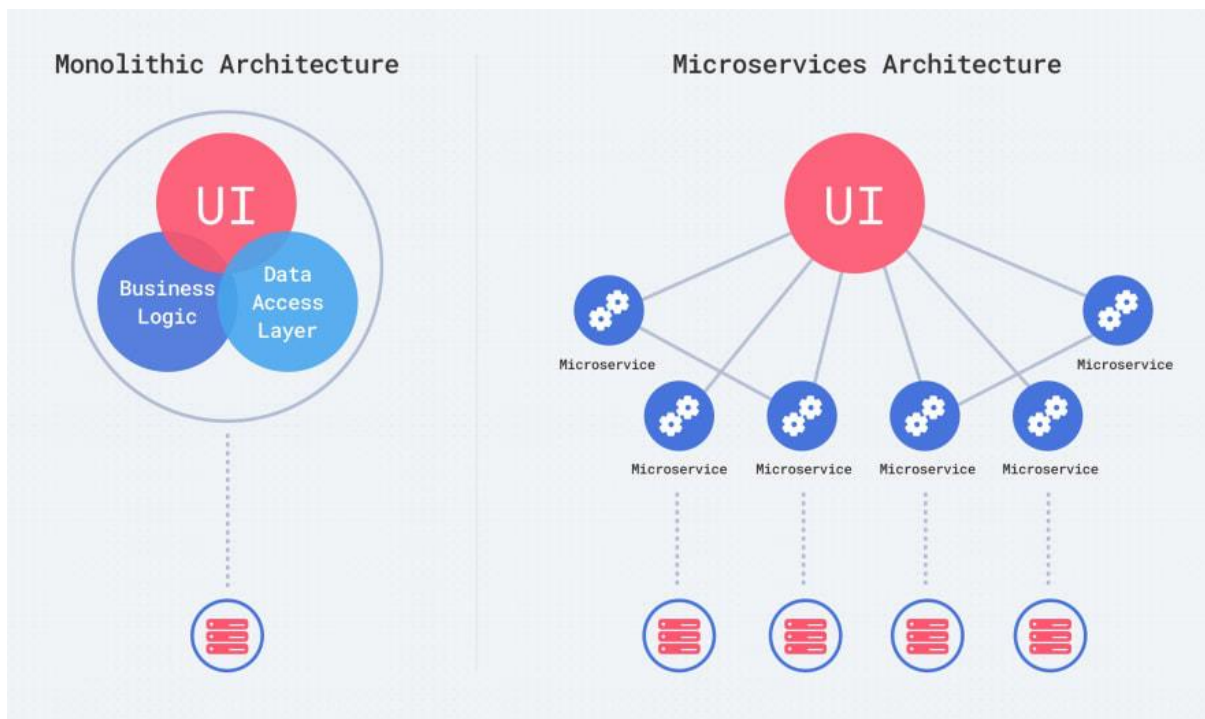
If it wasn't obvious enough with the above examples here are some more reasons to use Orchestration tools like Kubernetes if you are using containers:

- **Scalability** – Resources can be quickly increased or decreased to meet changing needs.
- **Stability** – Automation tools can automatically respond to fix problems before users see them.
- **Save time** – Certain tasks and workflows can be automated, freeing up engineers' time.
- **Self-service** – Orchestration can be used to offer resources to customers in a self-service fashion.
- **Granular insight into resource usage** – Orchestration tools give greater insight into how many resources are being used by what software, services, or customers.

30.9 Microservices

A microservice architecture breaks an application up into a collection of small, **loosely coupled** services.

Traditionally, apps used a monolithic architecture. In a monolithic architecture, all features and services are part of one large application. Microservices are small: each microservice implements only a small piece of an application's overall functionality. Microservices are **loosely coupled**: Different microservices interact with each other using stable and well-defined APIs. This means that they are independent of one another



30.9.1 What do Microservices look like?

There are many ways to structure and organize a microservice architecture.

For example, a pet shop application might have:

- A pet inventory service
- A customer details service

- An authentication service
- A pet adoption request service
- A payment processing service

Each of these is its own codebase and a separate running process (or processes). They can all be built, deployed, and scaled separately, these are all useful features if a service is being hit more than others for example it is unlikely all of your users get to your payment processors and as such they can be considerably weaker than the other nodes and still perform excellently.

30.9.2 Why use Microservices?

- **Modularity** – Microservices encourage modularity. In monolithic apps, individual pieces become tightly coupled, and complexity grows. Eventually, it's very hard to change anything without breaking something.
- **Technological flexibility** – You don't need to use the same languages and technologies for every part of the app. You can use the best tool for each job.
- **Optimized scalability** – You can scale individual parts of the app based upon resource usage and load. With a monolith, you must scale up the entire application, even if only one aspect of the service needs to be scaled.

Microservices aren't always the best choice. For smaller, simpler apps a monolith might be easier to manage.

30.10 DevOps and AWS

30.10.1 Amazon EC2 (Elastic Compute Cloud):

- IaaS – Infrastructure as a Service
 - As an IaaS platform you can control every single piece of the overall cloud infrastructure.
- Easily Scalable
- Full control over your cloud infrastructure
- Integrates with tons of tools, both AWS and 3rd party

30.10.2 AWS Elastic Beanstalk:

AWS Definition:

"AWS Elastic Beanstalk is an easy-to-use service for deploying and scaling web applications and services...

You can simply upload your code and Elastic Beanstalk automatically handles the deployment, from capacity provisioning, load balancing, auto-scaling to application health monitoring. At the same time, you retain full control over the AWS resources powering your application and can access the underlying resources at any time."

- PaaS – Platform as a Service
- Out of the box load balancing and autoscaling
- Can still access underlying AWS resources.

30.10.3 Continuous Integration, Delivery, and Deployment:

- AWS CodeBuild – Continuous integration
- AWS CodeDeploy – Continuous deployment
- AWS CodePipeline – Full code pipeline from build to deploy

- AWS CodeStar – Integrates all parts of the process with project management tools and JIRA issue tracker

30.10.4 IaC:

- CloudFormation – Stack templating engine, YAML or JSON-based
- OpsWorks – IaC with Chef

30.10.5 Serverless / FaaS – Function as a Service:

- AWS Lambda – run serverless functions on AWS

30.10.6 Monitoring:

- Amazon Cloudwatch – track metrics and logs, set alarms, and automate responses to monitoring data

Also useful for automate responses to monitoring data, automation run to fix problems when they occur. And of course, you can also use that to automate responses to your monitoring data. So you can have automation run to fix problems when they occur, or to really do anything you want. Any automation that you want execute in response to something that you're monitoring data is reflecting, you can create that using Amazon CloudWatch.