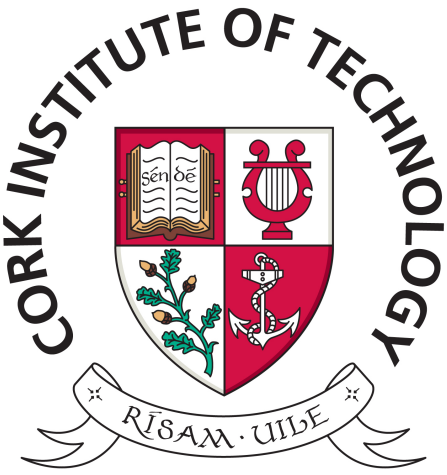

Automated CI/CD Creation Platform

by

Evan Day

This thesis has been submitted in partial fulfillment for the
degree of Bachelor of Science in Computer Systems

in the
Faculty of Engineering and Science
Department of Computer Science

May 2019

Declaration of Authorship

I, Evan Day , declare that this thesis titled, 'Automated CI/CD Creation Platform' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for an undergraduate degree at Cork Institute of Technology.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Cork Institute of Technology or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

CORK INSTITUTE OF TECHNOLOGY

Abstract

Faculty of Engineering and Science
Department of Computer Science

Bachelor of Science

by Evan Day

Continuous integration/continuous delivery (CI/CD) is an important aspect of the software development life cycle which enables the automation of manual steps throughout the development process. This automation minimises the time and cost associated with testing, reduces the time it takes to identify problems and accelerates the quality and delivery of production-ready code.

In this work, we explore why the state of automation is not enough in todays world. We investigate what has already been done and how we can drive this area forward. We take an in depth look at containerisation versus virtualisation and explore why Kubernetes has become such a major force in the open source world. Following this, we establish guidelines for a prototype implementation. With these guidelines, we deliver on a collection of four independent services that work together in order to deliver an application that automates the delivery of the core tools required.

Utilising technologies around orchestration and containerisation, we encapsulate the infrastructure provisioning and tool deployment into a singular offering. This offering enables the creation of a CI/CD infrastructure platform in a matter of minutes. As a result, we simplify the infrastructure provisioning process, we improve the quality of life for developers and enable the faster release of software by the enterprise.

Acknowledgements

Firstly, I would like to thank all my supervisors - David Murphy, John O'Brien and Diarmuid Grimes for all their help and guidance throughout the research and implementation phase of the project. It proved invaluable and the project would not be the same without it.

Secondly, I would like to thank everyone in the McKesson Cork office and in particular, my team. Everyone on my team helped me in immeasurable ways and I am forever grateful. In particular, I would like to thank Stephen OBrien for the initial project idea. I also want to thank Denis Canty for hiring me in the first place and James Hurley for being a great manager. I want to acknowledge Liam OBrien and Caoimhin King who are my brothers in arms from the very first days of working on our product internally. It is an absolute pleasure to work with you and I look forward to working with you guys in the near future. I also want to thank Shane Treacy for the help he gave in ensuring I survived the workload of final year. Finally, a huge shout out to Michael OHegarty, who is a fountain of knowledge and enjoyable stories, not to mention an excellent driver to and from Passage West and Kirbys.

I also want to acknowledge the Coder Dojo movement. While I did not exactly do a lot of coding in my time there and more played games, it gifted me two invaluable things. Firstly, it gave me a core group of best friends that I appreciate every single day. Secondly, it showed me what I wanted to do in life at a time where I had absolutely no clue. I am now living my best life and for that, I am forever grateful.

I want to thank my parents, Dan and Noreen who are forever supporting me with love, advice and monetary finance. My brothers, Steven, Philip and Jonathan and their partners, Jaque, Hannah and Juan Carlos. I love every single one of you and since I am the only one of us in Computer Science, I fully understand if you do not fully read this document. I also would like to acknowledge Denise and Conal Thomson, who put up with me throughout final year while being the loving parents that they are to Méabh Thomson. Méabh has brought me joy almost every day and has rarely woken me up any night which is a beautiful combination. I so look forward to seeing her grow and to see you guys continue to be the amazing parents that you are. I also want to thank my innumerable extended family, who all treated me like a son of their own. Finally, my dog Ted, who I do not get to see as much anymore, but is forever a good boy.

Contents

List of Figures

List of Tables

Abbreviations

AWS	**A**mazon **W**eb **S**ervices
CI/CD	**C**ontinuous **I**ntergration **C**ontinuous **D**elivery
YAML	**Y**et **A**nother **M**arkup **L**anguage
S3	**S**imple **S**torage **S**ervice
CLI	**C**ommand **L**ine **I**nterface
CIT	**C**ork **I**nstitute **T**echnology
PaaS	**P**latform **A**s **A** **S**ervice
SCM	**S**ource **C**ontrol **M**anagement
IaC	**I**nfrastructure **A**s **C**ode
DevOps	**D**eveloper **O**perations
GCP	**G**oogle **C**loud **P**latform
CSP	**C**loud **S**ervice **P**rovider
VPN	**V**irtual **P**rivate **N**etwork
VM	**V**irtual **M**achine
vCPU	**V**irtual **C**entral **P**rocessing **U**nit
RAM	**R**andom **A**ccess **M**emory
HA	**H**igh **A**vailability
LXC	**L**inu**X** **C**ontainers
CRUD	**C**reate **R**ead **U**pdate **D**elete
API	**A**pplication **P**rogramming **I**nterface
RBAC	**R**ole **B**ased **A**ccess **Control**
IAM	**I**dentity and **A**ccess **M**anagement
DNS	**D**omain **N**ame **S**ystem
TTL	**T**ime **T**o **L**ive

To my wonderful nephew, Samuel Jonathan Day...

Chapter 1

Introduction

1.1 Motivation

While I began my professional work with cloud computing technologies in McKesson working as a Software Engineer, my passion for the associated technologies began far before then. I self taught myself key Amazon Web Services technologies in early 2016 and found everything I worked with to be fascinating. I would find myself reading countless upon countless pages of documentation just out of sheer interest. My degree at the time, was called Software Development and Networking, which I found the Networking bit of the name to be the most interesting. While the first year networking fundamentals modules did not touch on cloud computing bar by name, I knew networking was where my initial passion was at.

I run a small side business which namely deals with web design, but backed by AWS. My business is almost like a container for the client - I will manage everything for them in terms of the services the website requires. Very quickly I sought to automate key activities of the business for my own time saving. This eventually lead onto my introduction to CI/CD technologies. Initially with Gitlab, I was able to create a simple YAML file that just did a copy to AWS S3 with the CLI for AWS. However eventually, I was able to fully leverage AWS CodeCommit, CodeBuild and CodePipeline to run a CD pipeline which deployed to staging and production environments. It may seem over the top for small businesses and individuals to have a full CD system for their website, but to me that was not the motivating factor.

The motivation for me was that I can say I am able to architect development workflows that can function at scale, automatically. I saved a large amount of time by automating the deployment process, but the knowledge that I can make an impact by designing these

solutions for my clients lives was amazing. I am a firm believer in the idea of causality and the butterfly effect - how our actions can have a ripple effect in the universe. It is these ideas that quite literally guide how I behave in life and how I execute on my actions. I crave to make that positive impact on larger and larger scales.

Then came this project idea. In my time at McKesson, the bulk of my work was taken up with trying to create CI/CD based technologies for teams to leverage internally. Now on one hand, it was great that it was not automated since I always had work to do. But on the other, if it was automated, it would make lives a lot easier across the organisation. That is when the wheels started to turn and the scale of it all hit me with this project. Creating an automation platform which handles the provisioning, configuring and maintaining of an entire CI/CD platform for development teams would have a massive impact on not only the organisation, but on me too. To be able to architect and deliver such a platform would take my knowledge in the area to the next level and without a doubt, be able to improve the day to day working lives of countless developers within an organisation at the scale of McKesson.

1.2 Contribution

The main contributions to this project can be enumerated by several core factors. Primarily, the curriculum and module structure of the Computer Systems degree enabled a strong networking skill set which we believe ignited the interests the author has in cloud computing. It was evident that the lecturers delivering the modules were extremely passionate about the content they delivered. We believe this may have had an impact. The set of modules delivered in the third year of the degree, may have had the most impact. Agile Processes and Distributed Systems Programming delivered a strong conceptual understanding of continuous integration, continuous delivery and client server communications respectively. These both went hand in hand when it came to delivering a project which relied on several core cloud technologies and understanding of communications between web and mobile based applications.

The other core factor we believe that contributed to this project was the activities we undertook due to the knowledge the author received in college. As alluded to, our interests in cloud computing lead to the creation of a small business outfit that deals with the management of cloud technologies. By creating a layer of abstraction between the cloud technologies and the end customer, we are able to gain incredible knowledge of the core offerings of the primary cloud provider for the business (AWS). Via CIT, we was able to complete an internship with Cork Internet eXchange where we dealt with data centre level operations in the context of creating a public cloud offering. Through

this, we got great experience in seeing how cloud offerings are built from the ground up which furthered our overall interest in the field. Finally, the work placement that we completed with McKesson lead to spending over eight months working directly on cloud providers and with some of the newest technologies such as Kubernetes.

When these are all tied together, along with a personal workflow system the author adopted, this has lead to our ability to decisively execute on the core tasks of this project and any other project we choose to adopt. Following core agile principles leads to the creation of large work queues while also setting aside focused work slots to complete this work. Having the fundamental understandings of how networking is implemented leads to the creation of solutions that architecturally, are not simply the dropping of the old way of doing things into cloud providers. Rather, developing cloud first solutions that are designed for cloud and therefore can take full advantage of what the current generation of technology and power offers us.

1.3 Structure of This Document

Chapter One details an introduction to this project, our motivations for undertaking this project in particular and holds the structure of the document. Chapter Two seeks to determine what has already been done within Computer Science around this projects core areas. These areas primarily are based on cloud automation and continuous integration and continuous delivery technologies.

Chapter Three defines the problem that we are attempting to solve with this project along with setting out the functional and nonfunctional requirements for the project. Chapter Four delves into our implementation approach for the project. This includes the proposed architecture of the solution followed by a risk assessment and the methodology for completing the project. Finally this chapter shows an implementation plan schedule, an overall evaluation and prototype of the solution. Subsequently, we look back on this implementation with a retrospective which forms Chapter Five. Chapter Six is the conclusion of the paper, where we discuss the project along with any future work that may be undertaken.

Chapter 2

Background

2.1 Thematic Area within Computer Science

This projects core topic can be broken down into several core components that all have key bearings within Computer Science.

2.1.1 Cloud Automation

Cloud Automation is a relevant topic within the parent area of Cloud Computing and therefore Computer Science. With nearly 77% of enterprises having at least one application or a portion of their enterprise computing infrastructure in the cloud[1], new technologies are required to further enhance the core tenants that Cloud Computing gives end users and organisations. Namely, elasticity, scalability, redundancy and more. Cloud Automation is one area that is highly sought by organisations and users in order to improve their cloud posture.[2]Through Cloud Automation, organisations can create templates for their applications and infrastructure. If done right, these templates can be version controlled through the use of SCM and IaC such as GIT and Terraform. With these technologies, standard Software Development practices can be applied to an organisations cloud infrastructure which opens a whole realm of possibilities for the organisation. Giving Software Engineers a familiar system for managing infrastructure using methods they are already familiar with can be a driving force towards creating a DevOps culture within the organisation.

2.1.2 Platform as a Service

Cloud Computing can traditionally be divided into separate sections. By looking at

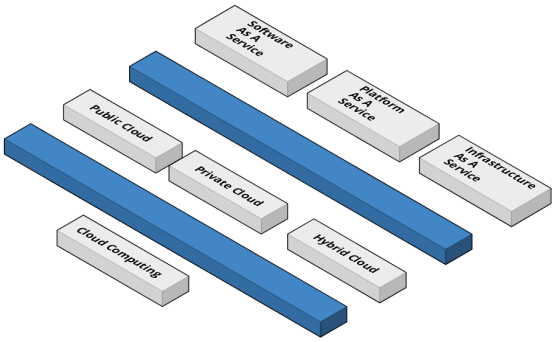

FIGURE 2.1: Cloud Computing Structure

figure 2.1, we can see a two tiered, triumvirate that sits on top of Cloud Computing. The first tier of As A Service models is typically what all end users interact with and the second tier of cloud types would power these models. Depending on the organisation that is involved, we can see all-in on either public or private cloud[3], or with current trends, a hybrid cloud strategy across public and private cloud. In the context of this project, a PaaS offering is the intended design goal such that the platform is completely agnostic of the underlying cloud provider. With this agnosticism, it enables the platform to be deployed anywhere in the context of public cloud (AWS, GCP, etc) or private cloud (OpenStack, VMWare, etc). By building it as a platform, best practices for infrastructure and the software being run on the infrastructure can be introduced.

2.1.3 CI/CD

While the practice and principles of CI/CD are rooted in Agile methodology[4],there are numerous technologies that have been created to support these methodologies[5]across a variety of technology stacks. There are known performance differences in organisations that have fully adopted CI/CD principles and tooling in comparison to organisations that do not. In the context of this project, the PaaS that is intended to be created would automate the provisioning and configuration of a small set of CI/CD technologies that have been researched as the most suitable for the platform, in this paper.

Following the exploration of the core topics for the project, the research will then look into several broader areas of computer science that include the following.

2.1.4 Virtualisation versus Containerisation

A critical question that needs to be answered by this paper is the concept of virtualisation versus containerisation. Containers have taken the developer world by storm. Technologies such as Docker and Kubernetes have seen massive uptake across various organisations. In an article from TechCrunch, the author quotes a research report that states usage of these technologies went from 10% in 2015 to nearly 71% in 2017 [6] However, there may be cases where containerisation is not the best decision for an application. Couple this with perhaps a generic understanding that containerisation is just lightweight virtualisation, there may be a case for just deploying software on regular, cloud based virtual machines. The answer to this question will be reflected in the overall architecture of the platform that this project aims to create.

2.1.5 Cloud Service Providers CI/CD Services versus Project PaaS Offering

There are a variety of build technologies, many of which could be considered the old reliables of build technologies such as Jenkins for example with its high workload count [7]. With the shift to CSPs, it may be desirable to the organisation to leverage the CSP specific CI/CD technologies for the same reasons an organisation may move to the cloud in the first place (cost savings, scalability, etc). However, these CSP based services can lead to vendor lock in and potentially issues with what the organisations CSP based CI/CD pipeline can achieve. Beyond cost savings and vendor lock in, this section will aim to show qualitative reasons as to why the PaaS offered by this project would be more desirable over the CSP based technologies.

2.1.6 Build Technologies

Development teams that stand up their own CI/CD pipelines will tend to use familiar or old reliable build technologies such as Jenkins and other key step technologies in their pipeline. In particular, it was reported that there was roughly 100,000 known installations of Jenkins in 2015.[7]These technologies may not be the best choice, which can eventually lead to technical debt in the form of scaling the pipeline as the development work increases. The project research aims to identify the best build technologies for the platform, given the underlying architecture (Virtualized workload or containerized workload) that will enable the platform to scale out based on the overall workload the platform has.

2.2 A Review of Cloud Automation and Virtualisation versus Containerisation

Through this review, the author hopes to gain a thorough understanding of what has been done in the area of cloud automation. Cloud automation can be delivered in several ways. Dependent on how far the automation must go will typically determine the workload involved. For the purposes of this review, we will be taking a look at automation reaching as far as the CSP infrastructure layer and going as high as the software configuration layer.

Industries such as health care can typically be associated with archaic practices when it comes to software development and delivery of said software. One must be conscientious of the fact that the health care industry most certainly has stricter regulations than others. As a result of these restrictions, innovation in these areas will either be slower or non existent. Another industry that may fall under this category would be the insurance industry. During the research process of this paper, the author came across a paper which talked about creating end to end automation for a cloud build pipeline in the context of the insurance industry.

In this paper, the author sets out the stages for application development as coding, building, integrating, testing, troubleshooting, infrastructure provisioning, configuration management, setting up run time environment, and deploying applications in different environments. [8]. These are all reasonable stages, in the context of our platform we may simplify these application development stages to the following.

- Source
- Build
- Test
- Deploy

Several of the stages in application development, may be combined into singular stages for the context of the pipeline created by the platform. A note to make is that source is not equivalent to coding. When making a reference to the source stage, we are referring to the actual source control management system the platform uses to access the code developed.

Further, the author of [8] talks about the classic situation before the culture of DevOps entered the world. Traditionally, Developers and Operations were two independent

entities with a metaphorical wall between them. With a DevOps culture, we seek to break this wall down and enable the rapid delivery of software to the end user. For the health care industry, if we can create a system whereby we are able to rapidly deliver applications that follow the strict requirements of the industry, we can become game changers. By creating a platform in the context of these requirements and enabling the platform to enforce these requirements, it becomes even easier to reach this goal.

Similarly, as referenced by the author of [8], the insurance industry faces similar challenges as the health care industry but arguably, several of these challenges are faced by other industries too.

- Slow process to integrate customer feedback
- Manual build process, which requires regular fixes, is a big show stopper in deploying code more frequently
- Manual intervention for rollback
- An urgent need to introduce new processes and tools covering build integration, test, deployment, and end to end deployment orchestration
- To develop and deploy efficient applications in cost effective manner[8]

Through a platform that is at its core, cloud automation, a large portion of these challenges can be resolved. We should adopt a practice of not trying to become attached to particular environments or troubleshooting deployments that have gone wrong when in production. Instead, we should bring on board a ruthless mentality of infrastructure destruction and reconstruction where by failed builds (if they even reach the stage of infrastructure deployment) running in cloud instances are immediately taken down and replaced with the most recent, successful build. With a heavy emphasis on continuous building and testing of our code, we can be deploying to production multiple times a day with zero fear. Through automation, we can handle the provisioning and configuring of our environments and ensure that our developers are always working on environments that are as close to production as possible, while not being the actual production environment.

When we look at the proposed solutions towards an end to end automation cloud pipeline by [8], we see a solution that holds the continuous integration, continuous testing, continuous delivery and the provisioning and configuration management all within Jenkins [9]. While this solution does appear to achieve the desired outcome, in the opinion of the author there is a reasonable disadvantage. The proposed solution will be entirely bound by Jenkins as a system. Additional considerations will have to be taken as regards

what kind of infrastructure is required to run Jenkins with such a system. Jenkins is by nature, a plugin based system. How many of these plugins will be required to run the pipeline? Will they cause additional load on the system? If so, how do we scale out Jenkins to handle this demand? How many applications will this particular Jenkins system service? There is no doubt, additional questions to answer. But if we are to attempt to answer these questions in the context of what this project seeks to create.

- As versus relying on plugin based configuration for a single software solution, we should instead incorporate several software solutions. These software solutions can focus on executing their singular task while also retaining independence for scaling out for additional capacity.
- We should be creating a singular platform that is the point of contact for relevant stakeholders. The platform should be responsible for creating independent infrastructure that encompasses our previous stages such as build and test, while also ensuring that this infrastructure is separate from itself and other pipelines. As a result, each pipeline then scales itself up or down as needed

Later on in this review, we will decide if we should create this platform and as a consequence, pipelines in a container based deployment model or a virtual machine based deployment model. This will certainly raise additional questions as regards our platforms abilities. Nevertheless, core takeaways include that we should split out our individual services of our pipeline as versus encapsulating them all in the one software solution.

Irrespective of this over reliance on Jenkins, the remaining, standard components recommended for a build pipeline are there and are recommended. Quality gates at every step of the pipeline ensure that when failure occurs against our set metrics, the pipeline progresses no further. Triggers going from stage to stage ensure we can automate the progression of every step. If we take the step of configuring notifications, all the relevant stakeholders can be made aware of critical events in the pipeline.

After we have established the basics of the pipeline, we must now look at the provisioning of environments for our applications once a build has passed the required stages of the pipeline. The paper [8] mentions the usage of Chef[10] for infrastructure deployment and configuration. An important aspect to note about this is as follows. If the application we are deploying is not containerized, then it makes sense to use a tool such as Chef or Ansible [11] to handle the provisioning of the target instance and the configuration of the necessary tools. However, if we are deploying containerized applications, then there is a key difference. If we execute on our usage of containers correctly, then we have handled the configuration of our application already. We then only require an instance with a

container runtime, most likely Docker [12]. We should then reconsider our choice of tool to an option that is more suited towards infrastructure provisioning and orchestration. Software such as Terraform [13] for the CSP provisioning and Kubernetes [14] for our application runtime would be more desirable in this case.

Finally, [8] goes on to list several benefits that are gained from such a pipeline such as automated deployments, faster delivery of builds, etc [8]. Overall, this paper has done a great deal of work in the area of cloud automation. But therein lies improvements to the overall system which we touched on in the previous sections of this review.

Let us now take a look at another work within the area of cloud automation. In [15], we see that they too talk about a concept we already explored in 2.1. This concept being how cloud can be looked at in terms of hardware and software stacks is shown below.[15]

- Infrastructure as a Service
- Platform as a Service
- Software as a Service

Including, how ownership of cloud computing can be divided[15]

- Private Cloud
- Public Cloud
- Hybrid Cloud

These models of stacks and ownership are important to understand going forward and are good to know in the context of industry. Again, we look to create something similar to a Platform as a Service model, which sits on top of a layer of abstraction that can allow deployment across any ownership model of cloud.

When we read [15], we realize that the overall concept that this project is attempting to deliver is not something brand new. Large organisations are most likely not unfamiliar with the concept of a Service Catalog[16]. When combined with a self service portal, end users can visit this portal and request services from the catalog. Examples may include a work phone, VPN access and so forth. We intend to mimic a similar concept with a portal that the end user accesses to create their CI/CD pipeline.

[15] then goes on to mention the overall structure of their system. In the context of [15], they have an Adapter Framework section which looks to provision the resources

in the IaaS model. With our project, we would most likely have a two layered system. The first layer being the equivalent of the Service Catalog and Self-Service Portal for us. Then the second layer being equivalent to Global Orchestration but behaving slightly differently.

This layer is very much the heart of the system. It is a critical layer from what we can see in [15] that handles the automation and orchestration of the various elements at the IaaS level. A core concept that we will need to duplicate and modify is the concept of northbound and southbound communication. [15] mentions north bound communication as communication between the global orchestration module and the self-service portal module[15]. They go on to say that we can use a standardized protocol such as web services (SOAP and REST)[15]. When we develop our global orchestration module later on, along with our overall system we should incorporate REST. By doing so, we can ensure effective communication going down to orchestration (Create My Pipeline Please!) via a request, then leverage HTTP codes to determine if the action was a success or not through the response that goes up (Okay Pipeline Is Created!). Preferably, a microservices based approach with communication all being logged and monitored would be a great innovation on top of an excellent proposed solution in this paper.

We shall now reference [17] as part of our decision towards architecting our project for virtual machines or for containers. In [17], they do a high level comparison of virtual machines and containers. VMs have been around for a long time and typically will virtualize the entire components of a regular machine. This virtualized machine will then sit on top of a host machine. An easy example is how one can run a Ubuntu Linux [18] machine on top of a Windows [19] host. As you can imagine, no matter the optimization attempts made, this leads to a rather large overhead in terms of performance. An example would be how the CPU instructions of Ubuntu will have to be translated into CPU instructions for Windows.

Containers on the other hand, are far more lightweight than VMs. Containers will be running on the host operating system as versus virtualizing an entire operating system themselves. Container runtimes such as Docker allow us to create an image for our containers to use. With these images, we can complete all our configuration within the image and then load in our application. Coupled with the version controlling of our images via tags [20], we can be certain that our environments are near identical and would only require a Docker runtime in order to run. [17] goes on to list several other benefits that we may gain from containers. Even looking at [21], the author of that paper makes several excellent points around VMs and Containers. They mention how Containers are tools for delivering software while VMs are about Hardware allocation

and management [21]. They go on to mention how VMs have become a core component of cloud yet we still have a myriad of issues. Particularly around resource consumption and start up times. We are again talking about the facts that VMs will contain full guest operating systems plus binaries and applications needed for said operating system. We cannot simply disregard this inefficient usage of resources both in terms of the physical compute and in terms of the resources of organisations.[21].

[21] then reiterates what containers are and the benefits we can gain. When done right, our containers are fully packaged, ready to deploy, components of our application. Looking at figure one in [21] we see how the traditional hypervisor in a VM deployment is replaced by the container engine that then sits on the host OS. Through CSPs, it is getting easier and easier to run container based workloads. It is simply just a matter of doing it effectively. We should develop for containers as first class citizens. Not a simple lift and shift of our applications as we run them now.

After this VM and container comparison, [21] then looks at the requirements of a PaaS [21]. They then demonstrate an excellent use case of PaaS where by we have several options for running our applications. In these options, the second and third options are where we will argue how best to implement our own PaaS. These options are running an app in its own individual VM or running each app in a container which is shared. [21]. We shall explore these options in further detail below.

Let us say that for our intended PaaS, we have three required services. Let us explore the second option mentioned previously, running each service in its own VM. We surely already gain a benefit in knowing that all the resources of the VM are benefiting that application. Let us also assume we add a constraint that all services must be highly available with a second VM, bringing us to a total of six VMs. All VMs are running two vCPU[22] and 8 gigabytes of RAM. We now have a compute fleet with a total twelve vCPU and 48 gigabytes of RAM.

This may seem like a reasonable set up, but let us begin to take this a part. In this setup, we scale out by adding additional virtual machines. We have also adopted a traditional approach that harkens back to older days where we had one server for one application. This lack of cloud nativeness can potentially lead to higher costs from our CSP by wasting money on unnecessary infrastructure. We must also ensure we configure each VM with our required software and tooling, which may require the onboarding of a tool such as Chef that was previously mentioned.

Now, let us look at the third option, the container approach. When we take this approach, we should not look at how many VMs we have. Rather, we should consider our

PaaS to have a common pool of vCPU and RAM. With orchestration tools such as Kubernetes, this is made possible. We have the same three services as before, but now we specify that in reality these services only require one vCPU and four gigabytes of RAM to start with. With our previous virtual machine size and high availability constraint, we can in fact now run two containers per virtual machine. The orchestration layer (Kubernetes) will ensure these containers are distributed across the VMs. Therefore, we can fulfill our HA constraint, while reducing the overall VM footprint by 50%. Since we leverage Docker, we do all our configuration in the image, meaning we can rely on the orchestration tools for delivering our services.

Overall, [21] dives into the architecture of containers for the remainder of the paper. By studying the architecture of containers, we can better decide what applications are more ready for a container based deployment than others. For example, we would consider containers to be easily destroyed and redeployed. Therefore applications that are stateful (for example, rely on a database) should probably offload the database to a non container based source.

Let us now take a step to look at [23]. The author of [23] takes a whistle stop tour on containers by first defining what is it that cloud computing is. The author reaches a conclusion that were left with the working definition that cloud computing, at its core, has hypervisors or containers as a fundamental technology[23]. Containers are not new by any measure, with their origins routed in technologies such as LXC[24].

[23] then shows a similar figure to what we saw in [21] previously. The figure deals with a hypervisor deployment as versus a container deployment. A key difference in the layout of this figure is that for the hypervisor deployment, it is individual operating systems that sit on the hypervisor as versus VMs. A VM at its heart, most certainly holds an operating system. But this figure does demonstrate the fact that container based systems typically share an operating system. As a result, the container engine is the hypervisor and were able to strip away the operating system component in a traditional hypervisor system. Onwards from the introduction and the context of hypervisors and containers in [23], we now look at Docker containers. In [23], the author talks about how Docker is an extension of the previously mentioned Linux Containers while also mentioning that Docker also uses namespaces to completely isolate an application's view of the underlying operating environment, including process trees, network, user IDs, and file systems. [23]. The author of [23] gives a great high level overview of how Docker containers are formed. We ourselves, may describe the image creation process as the layers of a cake. We start with a base layer, typically a stripped down version of an operating system such as Ubuntu[18]. Following this, we may add application dependencies as another layer. Then finally, our own application on top of this. As you

can see, this layered approach allows us to create images that have all our configuration done already. Now all we need is something to orchestrate these containers, which [23] talks about shortly after this section.

Kubernetes [14] is a piece of orchestration technology for Docker containers which as [23] quotes Google, Kubernetes is the decoupling of application containers from the details of the systems on which they run...GCP provides a homogenous set of raw resources for Kubernetes to use and in turn, Kubernetes schedules containers to use those resources.[23]. Albeit, this text was originally published in 2014. At this stage, it is hard to argue against the fact that Kubernetes has become an industry standard for container based deployments. For our PaaS in a container based approach, Kubernetes is the perfect technology for us. Once it is architected correctly, we can offload the majority of our CSP based orchestration to Kubernetes and focus on running our PaaS services.

[23] then concludes by saying how Docker and Kubernetes seem to have sparked the hope of a universal cloud application and deployment technology[23] and we believe that the author is not that far off that statement.

Chapter 3

Automated CI/CD Creation Platform

3.1 Problem Definition

When an organisation looks to create a CI/CD pipeline, there can be numerous problems encountered which can lead to a spin up time of weeks. Ideally, pipeline spin up should be at least within a day. Coupled with the fact that this is currently an entirely manual process, trial and error is inevitable. We aim to solve these problems by looking at the following sections.

For starters, we must first eliminate the opportunity to deviate from a set of technologies. As we saw with [5], there is a very large amount of tooling options available for use. By creating a predefined set of technologies, we eliminate time spent searching and running test configurations on potential technologies. We also can save time by ensuring we mirror the same configurations across multiple instances of the technology that is being used.

Following this, we must determine if there are any organisational specific rules that may impact the deployment of the technologies involved. One example rule can be that all cloud based infrastructure must be launched on internal facing networks. A topology could involve an organisational network connected to the cloud service provider network via a physical connection. A challenge in this may involve the creation of infrastructure via automation in pre existing resources. Another rule may come from a compliance perspective and requiring the provisioning of resources on infrastructure that is a dedicated tenancy model. That is, the physical server in the CSP is reserved to that organisation only. The configuration of the platform would then have to shift

to an installation of technology purpose as versus a provisioning of infrastructure, then installation of technology.

Dependent on the size of the organisation, one may find themselves dealing with multiple teams which handle a variety of tasks. Overall this can be good in general as key areas are divided among multiple stakeholders and enable the spending of man hours on the relevant tasks of the team. However, a situation may arise where teams are spread geographically across the world, thus making it hard to contact each other during work hours. Another factor can occur in the case of a new team being created in the organisation that they are unaware of who to contact for the resources they need, thus increasing the time to getting a pipeline up and running.

Scalability is critical once development teams begin to gain velocity. The pipeline that is created by the development team should be capable of scaling up and down with the demands of the team. High availability is mandatory. If a single component of the pipeline goes down, it can impact developer productivity for the entire duration of the outage.

Once the development team has completed the creation of the pipeline, they need to document it. Once it is documented, a process can be established for the creation of additional pipelines as required. Ideally this process should be entirely automated by the team. A manual process increases the probability of error during creation while also taking away man hours from the individual(s) creating the new pipeline.

Finally, managing the various pipelines that are created by teams is an activity that must be undertaken. Developer teams should be able to gain valuable insights from the performance of the application pipelines they operate. Health checking should be available too so that data is available for monitoring what components of the pipeline are healthy while being able to identify the individual components that are unhealthy.

3.2 Objectives

This project aims to achieve several core objectives to solve the aforementioned problem. One may consider the project broken into three phases of infrastructure, deployment and configuration. These phases are enumerated into the following objectives.

We wish to create software that is responsible for encapsulating CSP Kubernetes cluster creation technologies. Our software will behave as the gateway to the creation of these clusters for the end user. Typically, high level infrastructure within CSPs (such as a Kubernetes cluster) will always have some form of prerequisite service required. The

software we create should handle the creation of these prerequisite services, thus forming the full stack of our required infrastructure for the end user.

Once this stage is complete, we will want to create the software which provisions the required technologies onto the Kubernetes cluster created in the previous stage. Required technologies will be predetermined and will help to answer the question of this project surrounding the automating of a Java pipeline based on Gradle build technology.

Following this, we will need to ensure the necessary steps are taken to connect each provisioned technology. Configurations for each tool should be generated and implemented so that the platform is configured to the end users desire. Once this is complete, the core automation of provisioning infrastructure and installing technologies should be ready.

All previously created software for the project should then be encapsulated into a singular offering. This can be an intermediary service between the end user and the previous software. Positioned as an intermediary, this will enable the creation of automated CI/CD platforms via a variation of a CRUD API.

Finally, we will want to create a front end platform which is the primary point of entry for end users. This platform is where users will initiate the requests to create their CI/CD platforms. This platform will also handle the accessing of these platforms through URLs such as platform.com/build, platform.com/artifactory, etc.

3.3 Functional Requirements

We have defined the following as the functional requirements for this project.

- Create an API for installing specific technologies onto infrastructure provisioned by a separate API.
- Create an API responsible for generating configurations for specific technologies.
- Create an application which centralises the previously created APIs.
- Create an authentication service responsible for user management.
- Create a platform which enables a CRUD with the technologies created by the previous application.

3.4 Non-Functional Requirements

From defining our functional requirements, we shall now define our non-functional requirements

- All services created should be agnostic of any specific CSP.
- All services should use standard HTTP codes to determine operation success or failure.
- The platform should continuously poll created technologies to determine health status.

Chapter 4

Implementation Approach

4.1 Architecture

When we are looking at the architecture of this project, we can look at the phases that are talked about in the previous chapter. As a reminder, these phases are infrastructure, deployment and configuration. For the overall architecture of the project, we will change these phases slightly to instead reflect the following.

- Infrastructure
- Technologies
- Platform

In this chapter, we will elaborate on what each of these phases entails by giving a high level overview of each sub component.

4.1.1 Infrastructure

The anchor component around the infrastructure phase of this project is Terraform. Terraform is tooling that allows us to write code which enables Infrastructure as Code for us. By defining our infrastructure as code, it allows us to maximise the automation of our platform along with giving us key benefits. The reasoning behind the usage of Terraform is as follows. Terraform is an orchestration based IaC provider as versus a configuration based IaC provider. We do not require configuration for our platform resources as we are using Kubernetes.

Kubernetes therefore, only needs raw resources, which can be automated with Terraform. Terraform will be used extensively in the provisioning of our CSP specific Kubernetes clusters and the prerequisite technologies. These will include the likes of Identity and Access Management resources, enabling CSP APIs and creating the connection details for the platform to access the CSP resources.

Once we have our Kubernetes cluster, we have to undertake several prerequisite steps. Kubernetes employs a form of Role Based Access Control when it comes to resources within the cluster. In order for the tooling we wish to use to function, we need to create these RBAC based permissions. Once this is in place, our Kubernetes cluster will be ready for configuration by the platform.

4.1.2 Technologies

Now that the underlying infrastructure is ready, we will now look at deploying the key technologies of our platform. As part of the pipeline that our platform will be creating, several key technologies will be used. These technologies include the following.

- Webhook for our SCM for the application being developed on the pipeline.
- Concourse, for building our application files.
- Credhub, responsible for managing credentials for Concourse and other technologies.
- jFrog Artifactory, stores the artifacts that are generated as part of our application build.
- Sonarqube, executes code quality tests on our code against a set of best practices.
- Docker registry, which stores the Docker images created by the pipeline for deployment onto our Kubernetes cluster.
- Nginx, enables the proxying of our services towards the platform friendly name that we have as a goal for the project.

As we know, Kubernetes is an orchestration platform for Docker containers. We give Kubernetes the raw resources for it to deploy on to. Docker uses images and we will typically then create Kubernetes specific resources such as deployments and services to run our applications. Helm is a piece of software that behaves like a package manager for Kubernetes. It will encapsulate all the required items for our application into what

is known as a chart. We then deploy this chart against our Kubernetes cluster and let Helm do all the work.

With Kubernetes and Helm as a base, it means we can create Helm charts for each component of our pipeline. Since these charts are code files, we can then version control the individual components of our pipeline as needed. This compartmentalisation is an amazing benefit as we will achieve independent scaling of each component as it is required. Helm relies on value files for application specific configuration. If we can dynamically generate these files, we can ensure a far stronger configuration capability for the end user.

4.1.3 Platform

Finally, we have the platform component. The platform is the end user component of the entire project. The platform has several sub components that play critical roles in the execution of the entire creation of pipelines for the end user. We will now talk about these individual components in greater detail.

Health checking is the process of continuously asking a service if it is still available and healthy. This is typically accomplished by sending pings to a service and expecting a HTTP status code of 200 to indicate a healthy status. Any other code would be considered unhealthy. In the context of the platform, health checking should be performed on the infrastructure and every tool that the infrastructure supports. The architecture of the platform will have to support in depth health checking, so that services do not report false positives for example.

Another critical technology for the entire platform is Nginx. Nginx supports several functions, for our use case, we will rely on the web serving and reverse proxy capabilities. We want to ensure that the usage of the platform is as simple as possible. So with the reverse proxy capabilities, we will be able to have all technologies accessible from one, root IP address or domain name. This architectural design will enable far easier management of the individual services.

When we remind ourselves of the overall objective for the project, we see that the highest priority objective for the platform, is to have a pipeline CRUD in place. A CRUD as we talked about previously, is Create Read Update and Delete operations. We typically find this in database applications, but the acronym works for us too. Our platform needs to be able to do the following.

- Create pipelines for the end user.

- Read pipelines for the end user, for display and general information purposes.
- Update the overall state of specific pipelines.
- Delete the entirety of specific pipelines.

With this type of architecture, we can cater for virtually all forms of potential operations for the pipeline aspect of the project in a very deliverable manner.

Next, we will want to supplement the previously aforementioned health checking with a system for log processing, event monitoring and generally be able to see at a glance, how the platform and its associated resources are performing. For this, we will be deploying ElasticSearch, Logstash and Kibana as part of the platform to perform the level of monitoring that we will require.

To enable end users to use the platform, we will have to ensure we have a robust authentication service to allow access to the platform. From an architectural position, this may sit outside of the overall platform, potentially leveraging a third party service for simplicity reasons.

Finally, we will wish to ensure that we accommodate DNS in some shape or form. This will include ensuring all services within the platform are resolvable to each other. We may use Kubernetes Ingress objects to accomplish this, with the IP address of the Ingress being associated with an alias record in a specific DNS service provider.

4.2 Risk Assessment

Table 4.1: Project risk matrix

Frequency/ Consequence	**1-Rare**	**2-Remote**	**3-Occasional**	**4-Probable**	**5-Frequent**
4-Fatal		10, 11, 14	6	5	
3-Critical				2, 8	
2-Major				4	
1-Minor	12	7, 9		1, 13	3

4.2.1 Risk One - Terraform IAM Creation

- Consequence - Minor.
- Chance of occurring - Probable.

- Explanation - Best practice tells us that we should create IAM credentials with the lowest level of permissions possible. We should be able to automate this creation with Terraform. However, there is potential that this creation is not physically possible with Terraform.
- Possible mitigation - We can mitigate this risk by pre creating the required credentials in the CSP and embedding these in the project.

4.2.2 Risk Two - Acquiring Kubernetes credentials from cluster created by Terraform

- Consequence - Critical.
- Chance of occurring - Probable.
- Explanation - When we create a Kubernetes cluster, we can get the credentials of the cluster using Google Cloud Platforms CLI tool. This requires the correct information for the cluster. We need to be able to acquire this information from the output of Terraform, which at this stage in the project, we do not know if this is possible.
- Possible mitigation - Building a separate API for polling GCP Kubernetes clusters and then using the returned information to get the cluster connection information would suffice.

4.2.3 Risk Three - Kubernetes Role Based Access Control

- Consequence - Minor.
- Chance of occurring - Frequent.
- Explanation - Kubernetes uses a Role Based Access Control (RBAC) system for determining what actions can occur within a cluster. Using Helm, it requires specific RBAC privileges. Often, the applications that are deployed with Helm will require their own privileges too. As a result, we may find ourselves running into deployment failures due to RBAC issues. Or, more troubling, deployment success but applications that do not function correctly.
- Possible mitigation - We will have to either spend time learning the best practices for RBAC within Kubernetes, or give administrative rights to each application in use.

4.2.4 Risk Four - Kubernetes Nginx Ingress

- Consequence - Major.
- Chance of occurring - Probable.
- Explanation - To accomplish the project objective of having the created pipelines reside at example.com/serviceOne, example.com/serviceTwo etc, we will be using the Nginx Ingress in Kubernetes. As we have never configured this technology before and that its critical to the project, there is a risk that we will not be able to successfully configure this service. Therefore, there is a strong chance that this particular project objective will not be delivered on.
- Possible mitigation - An alternative service will have to be found, or a more traditional deployment of an Nginx reverse proxy will be required.

4.2.5 Risk Five - Helm Chart Installation

- Consequence - Fatal.
- Chance of occurring - Probable.
- Explanation - Helm, one of the primary technologies in use in this project, uses Charts to install services on a Kubernetes cluster. Through hands on experience, often a chart may be misconfigured, leading to issues with deployment. Or, the chart may succeed in deployment but the custom value file we rely on may be very minimal, or nonexistent. As a result, issues or delays caused by Helm charts failing to install may lead to fatal failures within the project.
- Possible mitigation - We may not rely on the publicly available charts and instead, create our own charts.

4.2.6 Risk Six - Helm Value File Generation

- Consequence - Fatal.
- Chance of occurring - Occasional.
- Explanation - Helm charts by default will use preset values for configuration of the deployed technology. For our project, we want to auto generate these files based on user choices and security. If we find that we cannot auto generate these files, then we will have to rely on predefined value files that will impact the overall functionality of the project.

- Possible mitigation - If we know that our end users will be software engineers and we assume they have a basic understanding of Helm value files, we may allow file upload for these value files.

4.2.7 Risk Seven - Service Health Checking

- Consequence - Minor.
- Chance of occurring - Remote.
- Explanation - In order to ensure that the resources the project creates are healthy, we need to have health checks in place. While we can do a basic health check via HTTP codes, this may not reflect the true status of the service(s) involved. As a result, we may have false positives being reported to us when it comes to determining if the platform is healthy.
- Possible mitigation - We can create custom health checking where we test to see if the service can perform the tasks it needs to execute. If these tests pass, we can then confidently confirm the services are healthy.

4.2.8 Risk Eight - DNS Record Creation and Propagation Times

- Consequence - Critical.
- Chance of occurring - Probable.
- Explanation - We will need to be able to create DNS records programmatically on the specific CSP we are deploying to. Dependent on propagation times, it can delay the platform from reaching a complete state of readiness.
- Possible mitigation - We can avoid creating DNS records entirely and rely on using the IP address of the Ingress or Load Balancer for the created platform.

4.2.9 Risk Nine - Authentication Service

- Consequence - Minor.
- Chance of occurring - Remote.
- Explanation - We have to decide between creating our own authentication service or leveraging a third party authentication service for user sign up, sign in and determining what users can and cannot do. There is a risk that using a specific

third party we limit the options of the platform. For example, using Amazon Cognito when the project operates on Google Cloud Platform.

- Possible mitigation - We can search for a third party that is agnostic of cloud providers. This risk may be completely removed however, since where the authentication service is located is out of scope for the project.

4.2.10 Risk Ten - Inter Service Communication

- Consequence - Fatal.
- Chance of occurring - Remote.
- Explanation - As we deploy the individual core services of our pipeline, we must ensure that said services can communicate with each other. Given that we are deploying to Kubernetes, we have to use standard Kubernetes networking resources. There is the risk that our core services will not work with these networking resources and this could cripple our project.
- Possible mitigation - We may expose individual service deployments as node ports or give them individual external IP addresses so that the core services of the pipeline can rely on traditional means.

4.2.11 Risk Eleven - Service Configuration

- Consequence - Fatal.
- Chance of occurring - Remote.
- Explanation - Each service will require configuration to function, much like any regular deployment of these tools. If we cannot configure these services with these deployments automatically, then the core goal of this project will be compromised.
- Possible mitigation - Helm value files should successfully mitigate this risk. Failing this, we have a fall back option. We can opt to configure services in Docker images and then deployments use these images.

4.2.12 Risk Twelve - Docker Registry Location

- Consequence - Minor.
- Chance of occurring - Rare.

- Explanation - Docker registries are remote storage for our Docker images. We will need to pick a registry that will be accessible to whenever the Kubernetes cluster is located. Preferably, an independent registry choice will be used.
- Possible mitigation - We can always use a cloud service provider registry, or we can self deploy a registry.

4.2.13 Risk Thirteen - Webhook Configuration

- Consequence - Minor.
- Chance of occurring - Probable.
- Explanation - As we have never configured webhooks before, we may experience teething issues in ensuring the source control portion of the end to end pipeline is configured so that the platform can clone the source code.
- Possible mitigation - Investing the necessary time to configure a webhook correctly should mitigate this low level risk.

4.2.14 Risk Fourteen - Cloud Platform Networking

- Consequence - Fatal.
- Chance of occurring - Remote.
- Explanation - Deploying our resources into existing cloud networking infrastructure may prove to be challenging. Terraform normally will create new networking infrastructure with any deployment that requires it. As a result, it may be hard to deploy the platform into any pre existing networking infrastructure that Terraform is not aware of.
- Possible mitigation - Acknowledging the low level of this risk occurring and prioritising the scenario of a green field deployment with no existing infrastructure is the current plan.

4.3 Methodology

The methodology that has been in use throughout this project relies on three core techniques that we use in the majority of our workloads. These techniques include -

- Dedicated information gathering.
- Systemist[25], a productivity workflow system.
- Kanban boards.
- Feature driven development.

4.3.1 Dedicated Information Gathering

Firstly, we take the time to gather as much information as possible around core areas of the project. For example, we may take the time to learn more about webhook configuration. We would begin by learning what a webhook really is and then begin to follow a trail of information. We have an exceptional reading speed which enables large quantities of information to be consumed easily. We use this to our advantage in as many areas as we can to ensure we can effectively deliver on our goals.

4.3.2 Systemist - Productivity Workflow System

We adopted the Systemist system roughly six months ago. One could comment that there are many ways of getting things done, thus you can get hung up on what method you use. We chose this method as we found it suited our work methodologies the best, we use the tool it was designed for (Todoist[26]) and it closely aligns with several Kanban concepts. While it is out of scope to fully explain the system, the core tenets of Systemist are as follows.

- Take it everywhere, so we can capture any potential ideas or thoughts related to our project.
- Capture everything, no matter how small. This increases freedom since we will not forget critical tasks and gives us the best overview of what to do.
- Break it down. If we can divide a task down, we will do so. While this may lead to massive task lists that may prove daunting it has an amazing benefit. With a large amount of smaller tasks, we build velocity and thus gain motivation to continue working.
- Prioritisation. We tend to prioritise by the date that a task is due and the impact that task has on the bigger picture. We may complete two high impact tasks in one day as versus five, low impact tasks.

- Task list zero. Completing an entire days worth of tasks is good for making progress and ensuring the system stays efficient.
- Get feedback. If it turns out that we are getting less done than we think we are, then we need to hear this feedback and take it onboard.

4.3.3 Kanban Boards

Paired with Systemist, we find Kanban boards can give the perfect high level overview of the project. While our productivity system does not align with pure Kanban, we can still use a board to demonstrate overall feature progress. The board would be updated at the end of every day, post a Systemist review. Duplicating tasks from Todoist to a Kanban board would virtually double the required time for task management, so we will avoid this practice.

4.3.4 Feature Driven Development

For our Kanban board, we will break down the entire project into Features. For a fixed time duration, we will work on that particular feature until it is delivered or we reach a blocker. If the feature takes longer than the fixed time duration, then we may have not broken the feature down far enough. This focus on individual features is another tactic that should reduce the chance of being overwhelmed. It will also ensure that we can construct a more feasible implementation plan schedule where we can see what features will be delivered on what weeks.

4.4 Implementation Plan Schedule

As we are using feature driven development, we plan to divide the project into several features. We also invite you to look at the prototype section where you can see a more visual overview of the implementation for the project. We plan to align our project to the semester calendar of twelve weeks. We have broken the project down into four core features, or epics.

- Terraform Controller
- Kubernetes Controller
- Infrastructure Controller

- Platform Controller

Each controller will receive an estimated three week development cycle. The core feature will be divided down into smaller, more actionable features that enable a fast development cycle. This will also help avoid any form of intimidation in terms of the large amount of work required. The general approach we are taking is to work from the cloud provider up towards the end user. This approach also shows the overall priority of each independent controller. There is little reason to work on the platform controller if our infrastructure controller is not fully operational, etc. If we find that our workload is too great, we can strip away the platform controller and potentially have a simpler, command line based tool. A quick summary of what to expect to be implemented with each controller is as follows.

Terraform Controller - Week One to Three

Initial sub features

- Terraform initialisation and authentication with Google Cloud.
- Terraform infrastructure creation.
- Terraform infrastructure destruction.
- Terraform state management.

Kubernetes Controller - Week Four to Six

Initial sub features

- Configuring the prerequisite RBAC resources on Kubernetes.
- Deploying Helm charts onto Kubernetes cluster.
- Reporting Kubernetes health metrics.

Infrastructure Controller - Week Seven to Nine

Initial sub features

- Wrap the Terraform and Kubernetes controllers into a single endpoint.
- Exposing relevant health checks for infrastructure assets.

- Call the relevant CRUD operations for Kubernetes and Terraform controllers.

Platform Controller - Week Ten to Twelve

Initial sub features

- Managing delivery of front end to users.
- Creating authentication service.
- Back end management service, responsible for talking to infrastructure controller.

These are rough sub features. In reality, each feature will be further refined down into executable tasks. Furthermore, extensive research will be taken to ensure we are not missing any critical functionality for the project as each controller is under development.

4.5 Evaluation

Headings we will evaluate the success of this project on include -

- Was a minimum viable product achieved?
- In terms of core features, how many of these were achieved?
- Did we expand on and deliver greater functionality beyond a minimum viable product?
- Did we solve the problem we were presented with at the beginning of the project?

The obvious goal that we need to measure against is did we deliver a minimum viable product. When we say minimum viable product, we mean did we deliver something which at a minimum, delivers on the promised functionality. For example, at a minimum, we want the project to be operational on Digital Ocean. If we deliver on multiple cloud provider support, that is even better. But if that support means that a separate feature, for example the Kubernetes automation is incomplete, then that is a failure. The goal of minimum viable product should always be the priority. Once the minimum viable product is achieved, we can focus on delivering additional, individual features.

As we will be using feature driven development, we will be breaking down the core aspects of the project into features. We will be taking these features for a fixed time

duration and working on implementing them. Features may be classed into core features and additional features. Core features will mirror our evaluation goal of minimum viable product and directly contribute to achieving that goal. We will agree a fixed number of core features (for example, ten core features) and as we complete these, we will measure the percentage completion of the project. We can map this to our implementation schedule and determine if we have made sufficient progress. If we discover that we are lacking in core feature development, we can then prioritise and ensure we deliver core features that contribute to the overall project completion percentage.

There is a wide array of additional features that can be developed for the project. Primarily, support beyond Java and Gradle based pipelines for example. Our priority remains on solving the original problem. Yet if we deliver support for additional languages and technologies, no doubt there will be additional use cases for the final deliverable of the project.

Finally, we must not lose sight of the original problem we intended to solve in this project. We need to deliver a platform that automates the end to end creation of CI/CD pipelines. If we cannot deliver on this goal, irrespective of the number of features delivered, then we have failed on the initial problem this project intended to solve.

4.6 Prototype

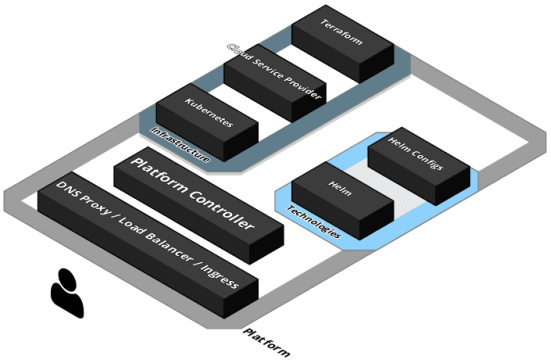

FIGURE 4.1: Platform Overview - Pre Implementation Phase

At this early stage of development, a test script was created. This script showcases what is already possible for deploying infrastructure and the required tools. This script can be located in the Appendices of this document.

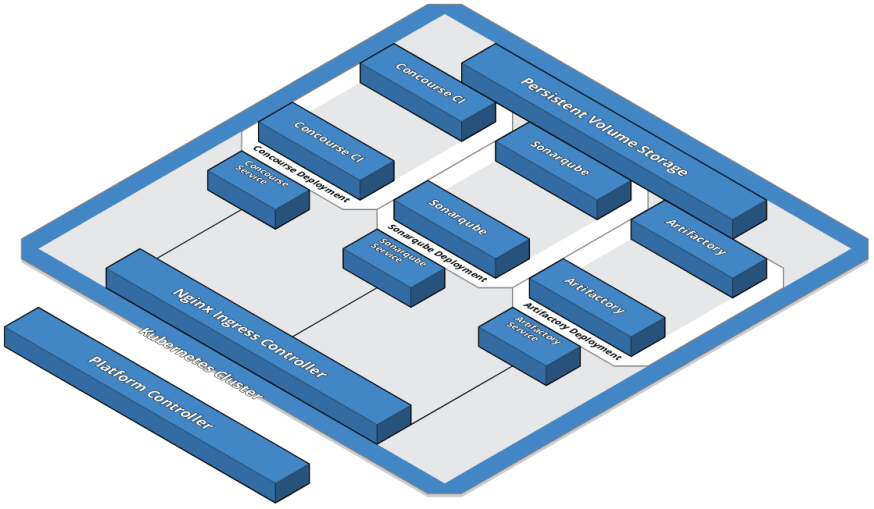

FIGURE 4.2: Kubernetes Overview - Pre Implementation Phase

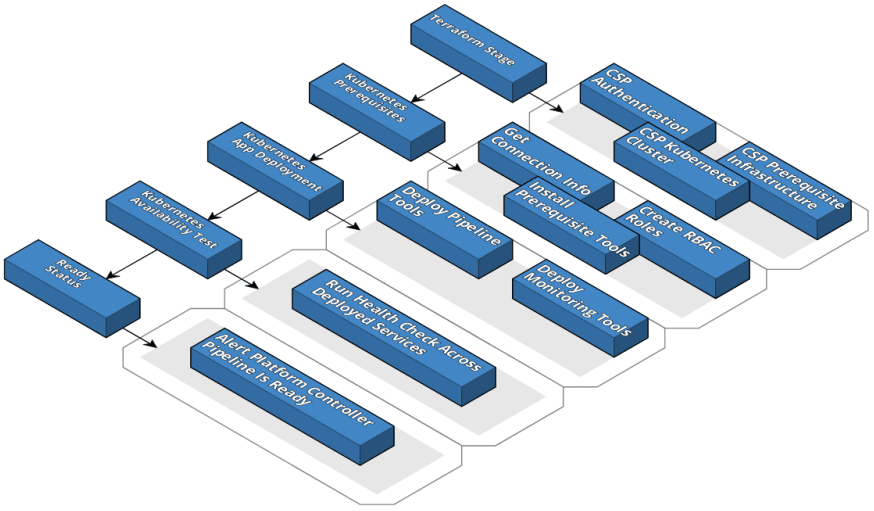

FIGURE 4.3: Pipeline Creation Process - Pre Implementation Phase

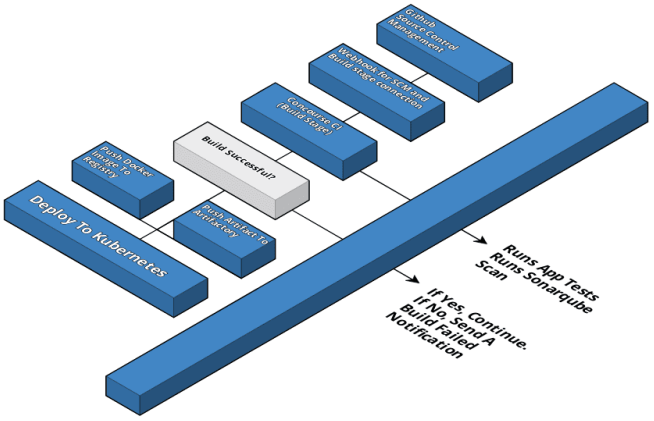

FIGURE 4.4: Code Process by Pipeline - Pre Implementation Phase

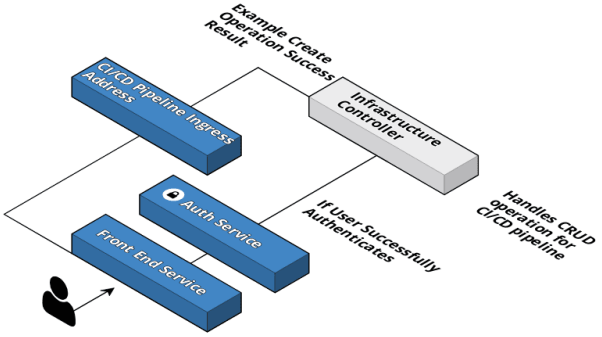

FIGURE 4.5: Platform Controller Overview - Pre Implementation Phase

Chapter 5

Implementation Retrospective - Expectations versus Reality

5.1 Introduction

In this chapter, we will be taking a look back at the implementation phase as a whole. In the research phase, we created implementation plans, had a general idea of a time frame for implementation, among other things. After completing the implementation phase, we are now going to take the time to review these aspects and see what was the reality of the implementation phase in comparison to our expectations.

5.2 Implementation Approach

5.2.1 Storage Controller

Early on in development, we identified the requirement for a storage layer in our project. As it turns out, two controllers relied on files to handle their operations. Terraform Controller, required Terraform files for operation and outputted a Kubernetes configuration file. The Kubernetes Controller also required the Kubernetes configuration file that was generated by the Terraform Controller. In a normal, monolithic application, this would not present as an issue since the application would be contained and run from the one context. But since every controller in our project is an independent service, we were required to develop a new controller that could fulfil these newly discovered requirements. These requirements include,

- Enable the uploading and downloading of specific files from one service, to the storage service.
- Storing these files in an accessible manner
- Maintaining the cloud agnosticism that forms a critical pillar of this project.

To satisfy these requirements, we began to work on the Storage Controller. This controller communicates via REST as is standard with the other controllers of the project. At the heart of the Storage Controller is Minio[27]. Minio is a storage service that exposes an S3 complaint API. As a result, Minio can sit on top of virtually any storage layer and maintain the same exposed API, requiring no change in the code of the application. We could of made our implementation easier and directly stored these files on Amazon Web Services. However, as mentioned this would cause us to no longer be cloud agnostic. So as a result, we opted to use Minio.

5.2.2 Platform Controller

Heading towards the second half of the implementation phase, we began to question was the functionality of the Platform Controller really required. As a reminder, the Platform Controller was planned to be primarily responsible for serving a front end to the end user and managing an authentication service. Based on the user request, the Platform Controller would then talk to the Infrastructure Controller to create the required resources. Below we talk about why we decided to remove the Platform Controller from our project.

We figured that for starters, it would not make sense to have the Platform Controller, send a request to the Infrastructure Controller and then begin creating infrastructure. It would be simpler and reduce the complexity of the project if we just had the Infrastructure Controller handling this through its existing, centralised endpoint.

In terms of serving a front end, we believe that with our target audience of developers that this is not really a high requirement. When it came to the critical period of the project, we wanted to invest our time in ensuring the core functionality of a minimum viable product was in place. We did this in lieu of attempting to design a front end and engineer the required communication to and from the back end. The required end point for the infrastructure controller can be visited inside a browser, which already enables an easy method of communication to the project.

Overall, these were the determining factors that resulted in us removing the Platform Controller from the project. The Infrastructure Controller exposes a singular endpoint

which we can poll to create the infrastructure platform. It was thanks to the removing of the Platform Controller which enabled us to complete the minimum viable product.

5.2.3 Technologies

When it came to our list of planned technologies, the vast majority of these were successfully implemented in the project. However, either due to a change in implementation or lack of availability, some of these technologies were not used. Credhub for example, a secrets management tool, was not implemented. This was based on the fact that Kubernetes has an object type of Secrets which is designed for the same purpose as Credhub. As a result, we opted not to use Credhub. We also did not include a webhook as part of the project implementation. The webhook fell in the realm of pipeline configuration where in reality, the focus for this project was infrastructure deployment and configuration. We also opted not to include a Docker registry as part of the project. This registry could be included with the jFrog Artifactory, provided a paid license was activated. Or, for reasons of availability, the registry could simply be configured with a third party.

5.2.4 Platform

A few changes were made to our approach when it came to the idea of the platform. For starters, the mere idea of what the Platform Controller would be doing was rolled into the Infrastructure Controller, as previously discussed. Our goal of implementing a CRUD for the platform fell short. We successfully implemented the creation and deletion aspects. But, when it came to the concept of reading and updating a platform, these aspects were never implemented in the current, project prototype.

There was no implementation of an Elasticsearch, Logstash and Kibana stack due to time constraints. Another interesting constraint would be that of power for the cluster, we estimate that we would have to double the current power levels of the Kubernetes cluster to support the stack. Interestingly, we believe this would also increase the spin up time of the cluster, which would have a direct knock on effect on the time it takes to spin up the platform.

5.3 Implementation Retrospective

5.3.1 Schedule Retrospective

Originally in our implementation plan, we specified that each controller would take an estimated three weeks of development. We estimated this, based on the initial sub features for each controller and the length of the semester. At the start of each implementation window for a controller, we expanded on the sub features to bring them to an implementable state. These features, in line with our implementation methodology, were then broken down into smaller tasks for working on. Typically, the stated implementation was attempted in a simpler manner, before we attempted the more complicated implementation which included inter controller communication. We felt that this was a smart approach since the simpler implementation took a shorter amount of time. This then lead to the complex implementation being easier as we had the basis of the functionality figured out.

Overall, the implementation schedule was adhered to, but several roadblocks put the project at risk of not being completed on time. As previously mentioned, the Storage Controller was not discovered as a requirement till the end of the Terraform Controllers development window. The Storage Controller proved be a hard component to implement. It could not be shelved till the end as the Terraform and Kubernetes Controllers were both dependent on it. It also had to adhere to our cloud agnosticism trait which meant a simple quick fix was not on the cards. The schedule was not designed to accommodate an unexpected requirement such as the Storage Controller. The schedule also was not cognisant of the workload required from other modules during the semester. Attempting to load balance other assignments while trying to implement components of the project on time was exceptionally hard.

However, one saving grace that enabled the implementation to succeed was ProcessBuilder. ProcessBuilder is a feature in Java which allows us to run applications as a process on the operating system. Coincidentally, it became very easy to achieve the projects functionality by wrapping the tools that are traditionally used manually. As well, Spring IO proved to be an excellent choice for a basis of the project. It made the implementation of communication via REST to be exceptionally simple and there was extensive documentation available online to assist with troubleshooting. The speed of implementation thanks to ProcessBuilder and Spring IO meant that irrespective of the delays encountered, the project was able to be completed on time. Below you can see each controller with a comment on how they adhered to their schedule.

- Terraform Controller - Fell within its planned schedule, but did not achieve full functionality until the Storage Controller was completed. Managed to convert from Google Cloud Platform to DigitalOcean during the planned schedule.
- Kubernetes Controller - Took longer to fully implement due to dependencies on Storage Controller, but core functionality was implemented on time.
- Infrastructure Controller - Was finished ahead of schedule due to less work required in achieving a full implementation.
- Storage Controller - Was never allocated a time slot for implementation, but took the longest due to researching various solutions in an attempt to get this controller working.

5.3.2 Implementation Retrospective

For starters, actually getting up and running with the development of the project proved to be the first challenge. Taking on a development such as this is no easy matter, where one is the judge, jury and executioner so to speak. We knew that we were responsible for the development of the project and that it would be our failing if it was not completed. So while we had this great fear initially, this was coupled with great excitement. To our knowledge, this was not attempted before. So to potentially be one of the first implementations of such a platform was a great feeling. Once we began the implementation, we gained more and more confidence in our own skills to complete the project, this confidence is what carried us on throughout the implementation phase.

One other item that came up was simply a question of how we would approach the implementation. We knew that the project would be developed using Java and Spring IO. But as previously mentioned, with this being a new undertaking we had no previous developments to go from. If there was an easy way of doing this kind of work, most likely it would have been done before. Fortunately for us, we discovered the ProcessBuilder in Java. This discovery really muted any voices that doubted the possibility of this project being implemented. ProcessBuilder simplified development greatly and while we still encountered issues in the project life cycle, we were able to iterate and release project features quickly.

The implementation schedule we created was very tight. As a result of this, unexpected variables were quite hard to accommodate. For example, the discovery of the Storage Controller as a required feature definitely threw a spanner into the works. Not only was this feature not planned for, but it proved to be exceptionally difficult for us to implement. This difficulty also lead to delays on the other components reaching an

initial, 1.0.0 release. Storage Controller development ate into other project component development time. If this feature was planned for and or implemented quickly, additional functions could have been added to other parts of the project to make the project even more robust and feature rich.

Following this, we then struggled with feature prioritisation. As mentioned, the Storage Controller severely impacted the development schedule. We could not ignore it since there was other features dependent on its features. How we prioritise the features of the project, ultimately will decide how much we would have implemented by the end of this project phase. While we identified features in several components that could be viable for a 1.1.0 release, we elected to not prioritise this. Instead, we prioritised getting the minimum viable product for the project implemented. Using a workaround, we could test the other components of the project without the Storage Controller which enabled an easier development environment. Once the Storage Controller was implemented, we continued with our minimum viable product prioritisation since at this stage, this phase of the project was reaching its conclusion.

Balance proved to be more of an issue than anticipated. When we talk about balance, we are referring to the balancing of our workloads in the context of college. We were aware that the implementation phase of this project was equivalent to two modules. Therefore, we should be devoting 14 hours of work per week to the project. While we did not exactly measure our time spent on the project, we can confirm that it got more time than other modules. The issue, is that it most certainly got more time than it should of and as a result, other modules suffered. We believe the core reason for this was down to personal interest and investment in the project as versus other modules. This project represents almost everything the author is interested in, in terms of technologies. As a result, it was naturally easier to work on the project, which rarely felt like work. Looking back, greater restraint should of most likely been applied so that the other modules being undertaken received proper development and learning time.

Most likely, the single greatest issue encountered from the authors perspective, was a series of health issues that occurred in the middle of the semester and thus in the middle of the project development lifecycle. These issues, compounded into a complete inability to work on any form of development across college and personal workloads. As a result, we fell behind on the implementation of the project and we believe that had this not occurred, greater functionality would be present in the project. Nevertheless, we believe that the prioritisation of our own health was the only call that could be made in this case. With this context in mind, we are still exceptionally pleased with the current implementation of the project.

As a result of the aforementioned, our next struggle was accepting that some features would simply not be implemented. Having expectations of being able to release new minor versions of our project components were dashed early on in the implementation phase lifecycle. A variety of new features were planned in the Terraform Controller for example, but these were shelved as implementation issues arose (eg - Storage Controller) In the end, the prioritisation of the project MVP was the correct call to make.

Troubleshooting project issues proved to be troublesome. One of the most serious issues encountered was with Kubernetes. We discovered that while we could successfully launch all the required tools, they were not launching correctly. We would frequently encounter issues with every deployment and or test, which meant we could never truly confirm if the platform was indeed working correctly. Towards the end of the project implementation phase, we had an epiphany. Thanks to a non related joke from a colleague, we realised that maybe our issue was an issue of resources. Kubernetes has a concept of resource requests and limits. These are designed to give applications a portion of the underlying nodes resources. This is done in part to ensure that one application does not consume all the resources of a node or that a node becomes overwhelmed. With this knowledge once again fresh in our minds, we ran our next test with several magnitudes of additional power. On this test, the entire platform came to life. It was one of the most troubling issues with the project as if it was not resolved, it would mean the project implementation was not a success. The moment when everything worked is something that we will most certainly cherish.

Finally, our expectation of implementing a structure of serving the platform applications at a /service URL was much harder than anticipated. While we were able to create these mappings, the applications were not able to resolve to these URLs. As a result, we had to change course and implement a sub-domain name approach to making the platform reachable. Once we did this, we were able to resolve each service to a sub-domain name.

5.4 Risk Assessment - Risks Experienced, Adverted and Resolved

In this section, we will be reviewing our risk assessment that we completed before the implementation phase. We will see what risks did we in fact experience, how did we advert certain risks and if a risk was encountered, how did we resolve it.

5.4.1 Risk One - Terraform IAM Creation

- Risk Status - Encountered and mitigated successfully.

- Explanation - We encountered this risk in the Terraform stage. While we did move from using Google Cloud Platform to DigitalOcean the risk still presented itself. Nevertheless, we mitigated this risk by creating a DigitalOcean API token. We then placed this token in a variables file in Terraform which allowed us to programmatically access DigitalOcean.

5.4.2 Risk Two - Acquiring Kubernetes credentials from cluster created by Terraform

- Risk Status - Not encountered.
- Explanation - This risk was not encountered in thanks to a resource type in Terraform. This resource type allows for the exporting of a Kubernetes configuration from a newly created cluster. As a result, this risk was successfully averted.

5.4.3 Risk Three - Kubernetes Role Based Access Control

- Risk Status - Not encountered.
- Explanation - We did not encounter this risk in part to giving Helm the correct level of permissions needed to allow applications to create their own sets of permissions.

5.4.4 Risk Four - Kubernetes Nginx Ingress

- Risk status - Encountered and mitigated successfully.
- Explanation - To mitigate this risk, we had to change the implementation approach to use sub-domain names. While this compromised the objective of having the services resolvable from a /service address, we still achieved the core functionality of the project.

5.4.5 Risk Five - Helm Chart Installation

- Risk status - Not encountered but mitigated.
- Explanation - While we did not directly encounter this risk, we were able to build a mitigation into the implementation, in the event of it occurring. A separate, unaccounted for risk with Helm did occur which related to Helm chart installation. Namely, not enough cluster resources available for the installation of applications. This risk too, was successfully mitigated.

5.4.6 Risk Six - Helm Value File Generation

- Risk status - Not encountered.
- Explanation - The project prototype did not build support for utilising custom Helm chart values, as a result we did not encounter this risk.

5.4.7 Risk Seven - Service Health Checking

- Risk status - Encountered and mitigated.
- Explanation - While we did encounter issues with health checks, we were able to implement a rudimentary health check using existing tools. These tools involve using a feature in Kubernetes called roll-out status. This allows us to wait for a deployment in Kubernetes to be considered ready. As a result of this, the platform deploys a service and then waits for that to be considered ready before moving onto the next task.

5.4.8 Risk Eight - DNS Record Creation and Propagation Times

- Risk status - Encountered and mitigated.
- Explanation - We managed to overcome this risk by manually creating DNS records in CloudFlare. We have found that with the Automatic TTL feature, the platform is able to respond to requests within a minute of the records being updated.

5.4.9 Risk Nine - Authentication Service

- Risk status - Not encountered.
- Explanation - An authentication service was not implemented as part of the project prototype. As a result, we did not encounter this risk.

5.4.10 Risk Ten - Inter Service Communication

- Risk status - Mitigated.
- Explanation - We did not encounter this risk, but as we implemented inter service communication through a simple manner via sub-domains, we do not foresee any issues.

5.4.11 Risk Eleven - Service Configuration

- Risk status - Not encountered.
- Explanation - The goal of the minimum viable product only covered infrastructure provisioning and deployment. Not automating the configuration of individual services.

5.4.12 Risk Twelve - Docker Registry Location

- Risk status - Not encountered.
- Explanation - Our prototype currently, does not use Docker when we run the platform. While we believe it would be easy to implement, this risk could most certainly be mitigated by using the image registry that comes with our version control system.

5.4.13 Risk Thirteen - Webhook Configuration

- Risk status - Not encountered.
- Explanation - We found no relevant feature in our implementation development that mandated the configuration of a webhook.

5.4.14 Risk Fourteen - Cloud Platform Networking

- Risk status - Not encountered.
- Explanation - By assuming a green field cloud deployment scenario for the prototype, this risk was not encountered during the development lifecycle.

5.5 A Retrospective Of Our Functional and Non Functional Requirements

We will now review the requirements we set out for our project before the implementation phase. We will take a look at each requirement and comment on what we achieved as regards that particular requirement.

5.5.1 Functional - Create an API for installing specific technologies onto infrastructure provisioned by a separate API

This requirement was achieved via the Terraform and Kubernetes Controllers. The Terraform Controller creates the infrastructure required for the project. While the Kubernetes Controller installs the various technologies onto said infrastructure. Since the two controllers are indeed two separate APIs, we implemented this requirement.

5.5.2 Functional - Create an API responsible for generating configurations for specific technologies

Looking back, this requirement appears to be a bit vague and could be misinterpreted. Based on our notes, we believe this related to generating custom configurations for the tools deployed by the platform. In an earlier section, we touched on how this particular feature was in fact out of scope for the project. Nevertheless, the Terraform Controller does technically generate a configuration file for a specific technology, Kubernetes in this case.

5.5.3 Functional - Create an application which centralises the previously created APIs

This requirement was achieved and is represented in the form of the Infrastructure Controller. While the Infrastructure Controller gained additional functionality in the form of the Platform Controller features, it still accomplishes the requirement that was set out above.

5.5.4 Functional - Create an authentication service responsible for user management

This requirement was not achieved. In part due to the minimum viable product being what was implemented in the end, along with the minimum viable product being redefined to not include authentication. We made this decision in light of reduced time available to implement features and that in retrospect, this should of been a non functional requirement.

5.5.5 Functional - Create a platform which enables a CRUD with the technologies created by the previous application.

This objective was not achieved. Primarily, this was due to the final implementation only allowing the creation of one platform. However, the thought process to enable multiple platform creation is there. An example, would be to generate a random identifier at the time of spin up. This identifier would be passed between every stage and then stored in some form of database. Then, the identifier would be sent to the platform to determine what action to take on what infrastructure.

5.5.6 Non-Functional - All services created should be agnostic of any specific CSP.

This non functional requirement, if given more research could certainly be confirmed as implemented or not. Currently, our Terraform Controller will output a Kubernetes configuration file based on the Kubernetes cluster created by DigitalOcean. We believe that theoretically, the Terraform file could hold details for any CSP based Kubernetes cluster and we will still get a Kubernetes configuration file as output. We have yet to test this however and thus it remains just a theory.

5.5.7 Non-Functional - All services should use standard HTTP codes to determine operation success or failure

While this requirement was not directly implemented, indirectly HTTP codes are indeed in use. As requests are sent to controllers, they will return a 200 HTTP code for an OK status while any form of error will return an error based HTTP code such as a variation of a 400 or 500 HTTP response.

5.5.8 Non-Functional - The platform should continuously poll created technologies to determine health status

Similar to the above requirement with HTTP codes, we indirectly implemented this requirement when the platform is creating the required services on the Kubernetes cluster. We use a feature known as rollout status, which waits on the specified service to show a status of Ready. Once this status is received, we assume the service to be healthy and we then move onto the next planned action.

5.6 Implementation Phase Diagrams

Below you will find various diagrams that represent the state of this work at the conclusion of the implementation phase.

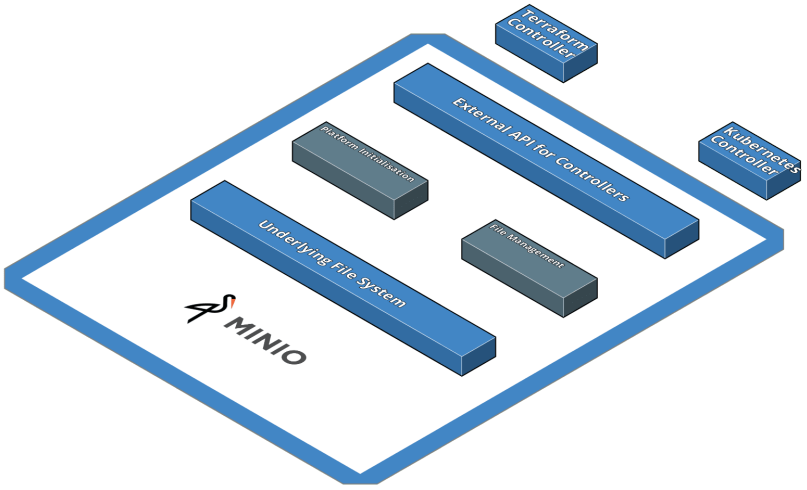

FIGURE 5.1: Storage Controller

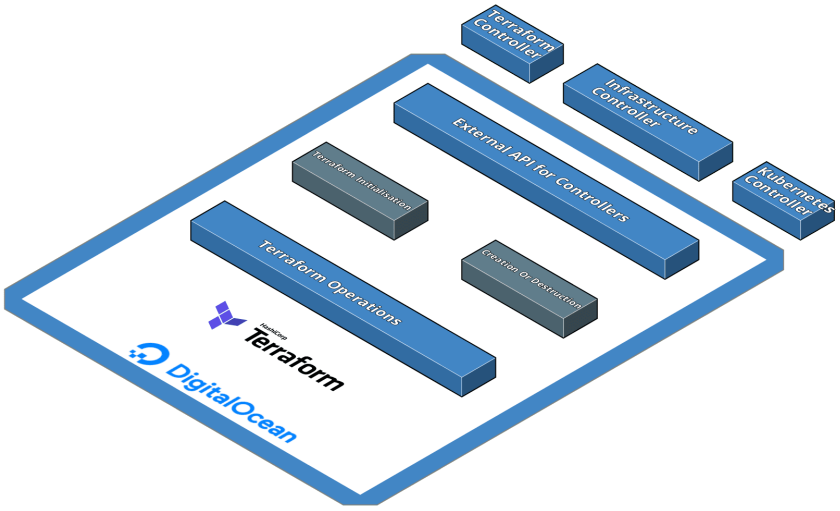

FIGURE 5.2: Terraform Controller

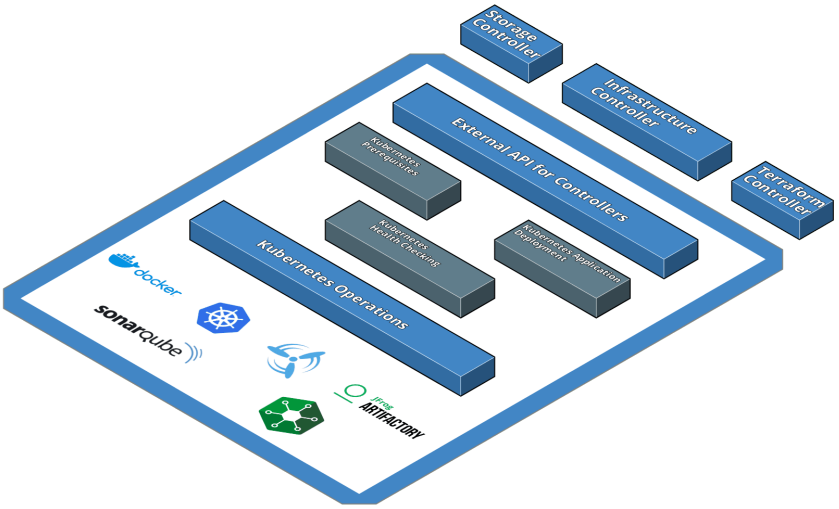

FIGURE 5.3: Kubernetes Controller

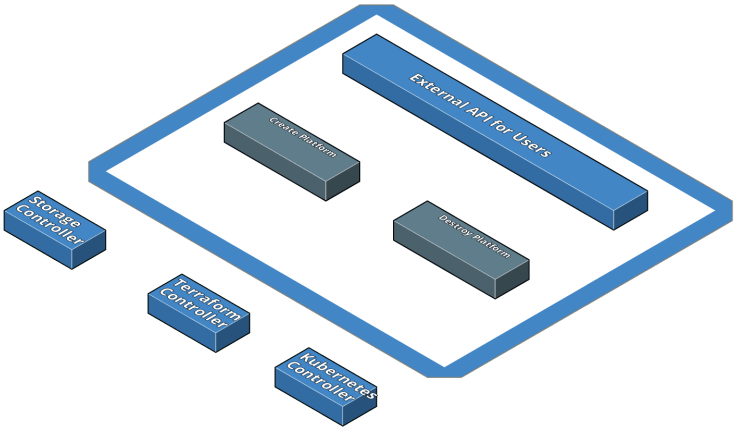

FIGURE 5.4: Infrastructure Controller

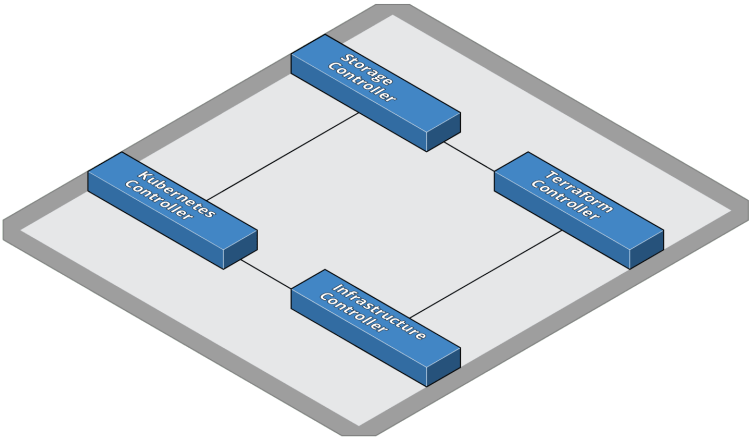

FIGURE 5.5: Platform Overview Post Implementation Phase

Chapter 6

Conclusions and Future Work

6.1 Discussion

We initially encountered a problem with the sourcing of material to read and gleam information from for this project. Trying to search directly for what we are trying to accomplish proved exceptionally challenging. While this did lead us to a conclusion that what we are trying to do was not done before, it was still a hurdle to overcome. We eventually decided to broaden our overall scope of material which proved beneficial. Finding one to two piece of anchor material, we could use this as a basis going forward for additional material. In the end, we found a reasonable amount of resources online that enabled our project.

Gaining a thorough understanding of all the individual components in play with our project was the next hurdle. While the concept is generally simple from a high level perspective, a view of the implementation aspects of it proved more complex. In the end, we took an approach of looking at the entire project in layers and then determining inter connection points between each layer. As a result, we could easily divide the project into certain areas and focus on implementing these areas for the project.

When it came to handling the configuration and deployment of services, we were concerned as to how it could be implemented. This particular component of the project is critical to ensuring the success of the entire project. If we could not find a solution, then certainly the project would be deemed a failure. Fortunately, when we reviewed the available tooling for Kubernetes, we were reminded of Helm. While we do find that there are several issues with Helm, we believe it to be the best solution currently for implementing this component of the project.

Finally, we think the most impactful event during this phase of research was the comparison of containers and virtual machines. Being able to gain a more in depth understanding of the core differences between these two technologies proved invaluable. With the conclusion in hand that containers are the agreeable choice for this project, we are able to proceed with the implementation in the second semester.

6.2 Conclusion

We reached two overarching conclusions during this phase of the report. For starters, we believe that more automation is required in the area of CI/CD. While we saw automation on a small scale, we believe that the automation must stretch from the cloud provider all the way to software configuration. Even going further beyond this by creating a platform that executes on this goal. Executing on automation at this scale will enable software engineering to achieve far greater velocity from the initial idea, all the way to the initial release.

The second overarching conclusion, is that when implemented correctly, containers are the way forward. We believe that solutions which still rely on state should consider a move to containers more carefully than others. But nevertheless, in dedicated container environments where we work on a container engine as versus a hypervisor, the performance benefits of containers far outstrip that of full, virtual machines.

6.3 Future Work

In the future, the obvious next step would be to implement support for other languages. Supporting Java with Gradle is a good first step, but it would be better to support other languages and build technologies. For example, we could expand Java support out to include Maven. Or, we could start to support Python for example. The overall expansion of the project to support more software archetypes would be the primary goal in the future.

A more subtle aim would be to support multiple cloud service provider Kubernetes services. On occasion, certain annotations may be required to get the full support of the underlying CSP. Along with Terraform support for these providers, it would be an ideal scenario if the end user could pick what cloud provider they wish to deploy too when they are using the platform. Another challenge would be to ensure we could support open source cloud software that is running a Kubernetes offering.

Bibliography

[1] L. Columbus". "state of enterprise cloud computing, 2018". [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2018/08/30/state-of-enterprise-cloud-computing-2018/

[2] "Brocade". "it professionals weight in on enterprise automation". [Online]. Available: https://www.networkworld.com/article/3176683/lan-wan/it-professionals-weigh-in-on-enterprise-automation.html

[3] K. Weins". "cloud computing trends: 2018 state of the cloud survey". [Online]. Available: https://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2018-state-cloud-survey

[4] I. Buchanan". "why agile isn't agile without continuous delivery". [Online]. Available: https://www.atlassian.com/continuous-delivery/why-agile-development-needs-continuous-delivery

[5] X. Labs". "the ultimate list of continuous integration (ci)". [Online]. Available: https://xebialabs.com/the-ultimate-devops-tool-chest/the-ultimate-list-of-continuous-integration-(ci)/

[6] R. Miller". "as kubernetes surged in popularity in 2017, it created a vibrant ecosystem". [Online]. Available: https://techcrunch.com/2017/12/18/as-kubernetes-surged-in-popularity-in-2017-it-created-a-vibrant-ecosystem/

[7] "Jenkins". "jenkins press information". [Online]. Available: https://jenkins.io/press/

[8] M. Soni, "End to end automation on cloud with build pipeline: the case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery," in *Cloud Computing in Emerging Markets (CCEM), 2015 IEEE International Conference on.* IEEE, 2015, pp. 85–89.

[9] "Jenkins". "jenkins - a ci server tool". [Online]. Available: https://jenkins.io/

[10] "Chef". "chef: Deploy new code faster and more frequently. automate infrastructure and applications — chef". [Online]. Available: https://chef.io/

[11] "Ansible". "ansible is simple it automation". [Online]. Available: https://www.ansible.com/

[12] "Docker". "enterprise container platform — docker". [Online]. Available: https://www.docker.com/

[13] "Terraform". "terraform by hashicorp". [Online]. Available: https://www.terraform.io/

[14] "Kubernetes". "production-grade container orchestration - kubernetes". [Online]. Available: https://kubernetes.io/

[15] E. Wibowo, "Cloud management and automation," in *Rural Information & Communication Technology and Electric-Vehicle Technology (rICT & ICeV-T), 2013 Joint International Conference on.* IEEE, 2013, pp. 1–4.

[16] "Techopedia". "what is a service catalog? - definition from techopedia". [Online]. Available: https://www.techopedia.com/definition/29355/service-catalog

[17] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on.* IEEE, 2014, pp. 610–614.

[18] "Ubuntu". "the leading operating system for pcs, iot devices, servers and the cloud — ubuntu". [Online]. Available: https://www.ubuntu.com/

[19] "Windows". "windows — official site for microsoft windows 10 home and pro os, laptops, pcs, tablets and more". [Online]. Available: https://www.microsoft.com/en-ie/windows

[20] "Docker". "docker tag — docker documentation". [Online]. Available: https://docs.docker.com/engine/reference/commandline/tag/

[21] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[22] "Techopedia". "what is a vcpu? - definition from techopedia". [Online]. Available: https://www.techopedia.com/definition/30859/vcpu

[23] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[24] L. Containers". "linux containers". [Online]. Available: https://linuxcontainers.org/

[25] A. Salihefendic". "systemist: A modern productivity workflow". [Online]. Available: https://blog.doist.com/systemist-a-modern-productivity-workflow-b6ec9ef05d3f

[26] "Todoist". "todoist the best to do list app 'i&' task manager". [Online]. Available: https://todoist.com/

[27] "Minio". "minio — object storage for ai". [Online]. Available: https://min.io/

Appendix A

Code Snippets

Our prototype script does the following tasks.

- Runs the terraform init command
- Runs the terraform plan command
- Runs the terraform apply command
- We then move to gcloud to acquire credentials to access the created Kubernetes cluster.
- We run a kubectl get nodes to confirm that we can indeed reach the cluster.
- We then create the necessary prerequisites for Helm
- We then install Helm onto the cluster
- We then deploy the charts of several applications. Concourse, Artifactory and Sonarqube

For this prototype stage, we are only demonstrating the capabilities of Terraform and the capabilities of Helm to deploy applications.

```
#!/bin/bash
export GOOGLE_CLOUD_KEYFILE_JSON=~/prototype/terraform-service-account.json
terraform init
terraform plan
terraform apply -auto-approve
gcloud container clusters get-credentials marcellus-wallace
kubectl get nodes
kubectl create -f /home/evan/kubernetes/infrastructure/tiller/rbac-config.yml
helm init --service-account tiller
sleep 45 # Sleep to allow for Tiller pods to be ready
helm install --name artifactory jfrog/artifactory --version 7.8.3
helm install --name concourse stable/concourse --version 3.0.1
helm install --name sonarqube stable/sonarqube --version 0.12.0
sleep 45 # Sleep to allow app pods to be ready
kubectl get pods --all-namespaces
```

FIGURE A.1: Prototype Script