Anh Pham

Building Continuous Delivery Pipeline for Microservices

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

8 May 2018

Metropolia

Author(s) Title Number of Pages Date	Anh Pham Building Continuous Delivery Pipeline for Microservices 52 pages + 3 appendices 8 May 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructor(s)	Olli Hämäläinen, Senior Lecturer

During the last two decades, there has been an impressive growth in software engineering industry. As a result, a considerable amount of applications are built every day. Meanwhile, modern technologies are also published at lightning speed to improve performance and scalability. Deploying an application using a modern software stack to a cloud platform is really challenging without a container technology standard solution. Moreover, containerised application deployment and management are much more comfortable with the help of a cluster management tool. In addition, to adapt to the need of customers, products must be delivered continuously and reliably. With the help of Continuous Delivery pipeline, the duration of development, testing, compiling, building and deployment procedures are optimised.

The objective of this project was to set up a pipeline to continuously deliver containerised applications to Google Cloud Platform with the help of Docker, Kubernetes, and CircleCI. Additionally, the meaning behind the implementation was to show the need of setting up a Continuous Delivery pipeline in every software development project.

The goal was achieved by researching and implementing a Continuous Delivery pipeline for an application which was being developed in a company. In addition, the benefits of using Docker containers, Kubernetes, CircleCI, Codecov and Google Cloud Platform in software development were investigated and showed during the implementation of the project.

Generally, the project illustrated how to set up a Continuous Delivery pipeline. Though there were several challenges, a considerable amount of benefits was shown by setting up a pipeline for a software engineering project.

Keywords	Continuous Delivery, CD pipeline, Kubernetes, Docker, CircleCI, Codecov, Google Cloud Platform, Google Kubernetes Engine, Google Container Registry

Contents

Abbreviations

API	Application programming interface.
CD	Continuous Delivery.
CI	Continuous Integration.
CLI	Command-line Interface.
GCP	Google Cloud Platform.
GCR	Google Container Registry.
GKE	Google Kubernetes Engine.
IaaS	Infrastructure as a Service.
IaC	Infrastructure as Code.
JSON	JavaScript Object Notation.
LXC	Linux containers.
MVP	Minimum viable product.
PaaS	Platform as a Service.
VM	Virtual machine.
YAML	YAML Ain’t Markup Language.

1 Introduction

An impressive development of information technology during last few decades has resulted in diverse edges such as mobile devices, cloud services, and data engineering [1, 35]. With the growth of the Internet, cloud computing has become much more common in software compared to the last two decades. The migration of backend systems to cloud service providers delivers many advantages to build an agile, scalable and economical server architecture. However, the software development process still has not adapted to this rapid change to deliver the product in a rapid and trustworthy method. [2, 85.]

Enterprises are delivering products to keep up with continuous changes of markets. Currently, customers have no reason to wait for half a year (or more) for a release to experience a new feature and give feedback on how the product should perform. [3, 78.] Consequently, the software development process is under pressure to distribute the products much more rapidly. Regularly, software is deployed in the production environment only after development has been completed and the environment is configured by the operations team. However, most of the time, there is a lack of collaboration between development and operation team. That is the reason why building application for multiple platforms takes a huge amount of time. [4, 323.]

Fortunately, the gap between engineering and operating can be shortened by using containers – a lightweight virtualisation technology. Recently, Docker container has become a defacto format because of the fast extension in software development community. [4, 323.] Docker solved the problem in packaging the software as a separate unit. In addition, Kubernetes is a tool to deploy, scale and manage Docker containers automatically [5]. Kubernetes enables users to rapidly deploy, scale and audit products in cloud platform by concise commands. Moreover, application container images can be stored in Container Registry for fast and reliable deployment or rollback. [6.] Generally, setting up a Continuous Delivery pipeline is an outstanding solution to deliver products with high quality and in a fast manner [3, 78].

The primary objective of this project was to set up a Continuous Delivery pipeline using

CircleCI as an automation platform for building and deploying an application to Kubernetes Engine in Google Cloud Platform. Software in the pipeline is packaged by Docker and deployed to Google Kubernetes Engine using Kubernetes. In addition, the code coverage reports were centralised at Codecov service.

2 Theoretical background

2.1 Microservices

During the last decade, there has been a massive growth in information technology on a variety of aspects. The race of smartphones, virtualisation and migration to cloud computing are the drivers of the progress. As a result, the software systems have become extensively complicated. Web applications nowadays typically include several components such as user-interface web, access and identity management, asynchronous tasks processor, databases, analytics and more. [1, 35.] At the same time, considerable attention has been paid to find better methods in structuring applications. Domain-driven design has presented better approaches to design systems. Continuous delivery has revealed the way to productively bring source code into the product. Provisioning, resizing and handling instances at scale became more effortless than ever thanks to virtualisation platforms and infrastructure automation. Microservices has its roots in all of them and from this world. [7, 1.]

Until recently, web applications commonly have been developed using monolithic architectures where almost all elements of the system are operated in a single process. In simple cases, the process of deployment, networking and scaling is trivial in this architecture. However, when requirements grow in complexity, they all suffer from strong barriers related to risky new development, disability to reuse part of the system and scaling issues. Microservice architecture appeared to substitute the monolith with a distributed system of independent, lightweight and narrowly focused services. [1, 35.] To have a general definition of microservices, Sam Newman stated that “Microservices are small, autonomous services that work together.” [7, 2]. In microservice architecture, the smaller the service is, the more advantages the system acquires. When services get smaller, they only focus within their explicit boundaries. As a result, chances for services to grow too large are prevented. [7, 2-3.] Moreover, each service is an independent entity. A service exposes its Application programming interface (API) for communication with others via network requests to keep the distinction between services and prevent the risks of dependences.

The whole microservice system is broken without decoupling as every change in a service should be deployed regardless of consumer's modifications. [7, 3.]

2.1.1 Benefits of microservice architecture

Microservice architecture plays a vital role in building software systems nowadays because of considerable benefits they brought in. Most of these benefits are commonly recognised at any distributed system. Nevertheless, microservices aim at bringing all these benefits to a higher level by making use of the concepts behind distributed systems and service-oriented architecture. [7, 4.]

Firstly, each service can be implemented using separate technologies since they are all independent of each other. As a result, there is no common formula for every task in the system. The most suitable programming language and framework can be chosen for implementation in each service. Moreover, when the performance of a service needs to be improved, a solution can use another technology stack to achieve the needed performance level. In addition, this solution enables the development team to rapidly take up new technology and grasp how they better the progress. This capability has been applied by many organisations to assess new technologies instantly. However, every microservice system should have a correct balance for not making this too complicated. [7, 4.]

Secondly, microservices provide a better solution for resilience. Bulkhead, a pattern to prevent failures from crashing the whole system, is an integral concept in resilience engineering. With microservices, a service can be built to control the breakdowns of services and reduce the range of capabilities correspondingly. However, this leads to the new aspect of collapses need to be handled in distributed systems such as network and machine failures. [7, 5.]

Another profit microservices bring in is agility in scaling. In monolith architecture, the system has to be scaled as a whole when a part is limited in performance. With microservices, individual services can be scaled up separately to let other parts of the system remain in smaller-scale and less powerful hardware. Additionally, this is an integral part of managing the costs effectively in the cloud. [7, 5-6.]

Deployments in monolithic applications frequently contain troubles as they all come with huge impacts and considerable risks. In monolithic architecture, the more differences between releases, the higher the risk of getting something wrong. With microservices, every change to a particular service can be deployed independently regardless of the rest of the system. As a result, the delivery time to customers has shortened thanks to having independent and faster deployments. [7, 6.]

Microservices also help organisations have better solutions in team coordination. By experience, smaller teams work more productively on smaller codebases. This approach helps companies minimise the number of developers working on the same code-base and achieve the ideal ratio between team size and productivity. Moreover, the rotation in ownership of services between teams assists developers to genuinely know about the product and parts of the system. [7, 7.]

Additionally, functionalities in microservices can be used in various ways for numerous purposes. This technique is also one of the essential parts in building a comprehensive distributed system. Customers are using software through different channels such as desktop web, mobile web, mobile app or wearable devices. As a result, software systems need to be architected in broader concepts regarding user engagements. In monolithic architecture, applications need considerable efforts in breaking down into useful recyclable functionalities. With microservices, each service can be built in different ways as they are all reusable and re-composable. [7, 7.]

Lastly, services can be rewritten or removed with minimal risks as they are all tiny and independent. Because of the ease in service management, developers can get rid of hundreds of lines of code in a single day without any worries at all. Teams maintaining microservices are totally happy to rewrite services entirely when needed as the codebase is only few hundred lines long. As a result, the cost to maintain or even replace services is always modest. [7, 7-8.]

2.1.2 Principles of microservice architecture

System design involves numerous thoughtful and accurate decisions, and they are all about give-and-take especially in microservices. For instance, when choosing a data-

store, there are many choices. However, the question is, should the development team follow with a commonly used database or adopt a new one with a better scaling solution. Some selections can be straightforward to decide immediately as information about them is always available, and sufficient. Nevertheless, the decision can be challenging for issues lacking necessary instructions. Accordingly, specifying a set of principles that guide the whole team to have better decision making is necessary. [7, 17.] The principles are different in different projects, but there are vital principles which are outlined in Figure 1 to follow for building a successful microservices system.

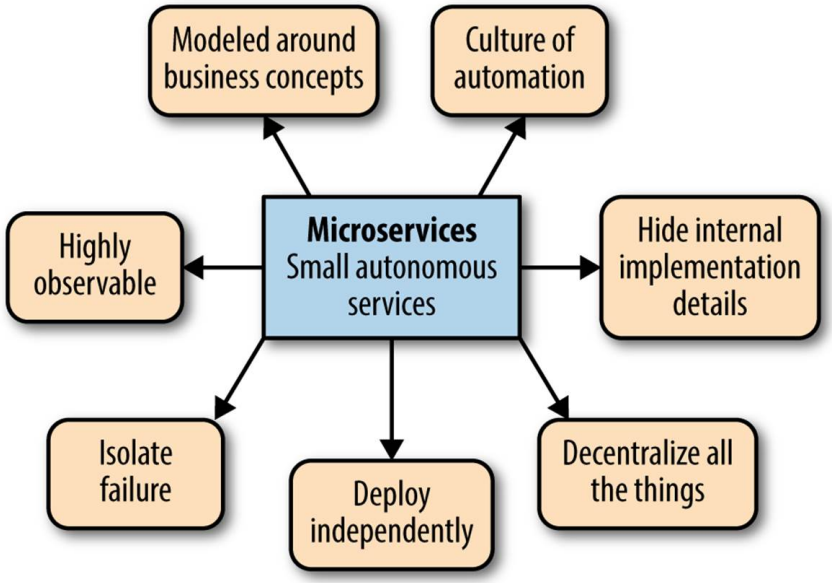

Figure 1: Principles of microservices. Reprinted from Building Microservices (2015) [7, 246].

As can be seen from Figure 1, the summary of fundamental principles to follow in building every microservices system is described clearly and concisely. These principles are described in more details in the following sections.

Modeled around business concepts

In microservices systems, services are small and independent. As a result, structuring them around business approaches increases the stability of the system. Moreover, changes in business processes are rapidly and efficiently applied to system operations. The key presented here is using business-bounded context to represent possible models properly. [7, 246.]

Culture of automation

Service administrators have a huge workload when dealing with the microservice system and possibly they can be the sources of failures. This issue is entirely understandable as there are plenty of moving pieces they have to manage. Adopting a culture of automation is vital to minimise the unexpected failures. Automated testing and deployment are crucial in every running microservices system. Furthermore, embracing usage of environment definitions and automated immutable servers creation allows the whole team to get user reactions faster after every customisation. [7, 246.]

Hide internal implementation details

Concealing the data and logic of service is exceedingly significant to make services more independent and autonomous. This method divides the models into two categories: public and private. Fortunately, this can be done partly by structuring models around bounded contexts. Besides, technology-agnosis is worth to consider for maximising the freedom of using a wide range of technology stacks. [7, 247.]

Decentralise all the things

Delegating and ensuring that teams own their services are also extensively valuable in building a good microservices system. Moreover, this principle betters development teams' capabilities in dealing with decisions made by themselves. In addition, the system architecture can apply this rule by staying away from orchestration systems or enterprise service bus approaches to keep implementation within service border. As a result, the system is more cohesive. [7, 247.]

Independently deployable

A microservice can be self-deployed at any time without extra work from other services. This principle shortens the release time to deliver new features to customers' hands rapidly. Even breaking changes are not barriers as there are many approaches to prevent system breakages such as coexist versioned endpoints, blue/green release, consumer-driven contracts and so on. In these situations, services always need to guarantee that their consumers to communicate with deprecated functionalities. [7, 248.]

Isolate failure

The microservices system resilience is better than monolithic system recovery only when all possible failures are recognised carefully beforehand. Microservices systems might be destroyed by cascading missteps as user-end influence is just only a part of the system collapse. One of the most important rules is not to treat remote requests in the same way as local requests as several sorts of failures are hidden by some means. By paying attention to anti-fragility approach, building fault-tolerant microservices system is not that hard. In practice, by setting up proper timeouts, using bulkheads pattern appropriately and being aware of potential network connection failures are all correct methods to avoid being in the situation of a demolished system. [7, 248.]

Highly observable

System monitoring is consistently a need in every application exclusively in microservices systems. To identify if the system is working correctly, observing the status of services and instances is not reliable enough. Nevertheless, system administrators need a method to observe what is running. Semantic logs and statistic aggregation into a web dashboard is a solution for administrators. Consequently, in case a problem occurs, the processing time for solving are extremely decreased. [7, 249.]

2.2 Containers

In the last decade, there has been a growing interest in cloud computing [4, 323]. Consequently, there is a variety of cloud computing definitions. Hardware resources are usually separated and shared dynamically to supply separate and multi-tenancy tiers. For better operations, there comes the appearance of techniques implementing specified models in cloud computing. [8, 81.] Hypervisor and container are commonly used technologies. On the one hand, hypervisor commonly reserves a portion of available hardware to be used by Virtual machine (VM). On the other hand, containers virtualise the operating system. [9.] In general, both containers and hypervisors are recognised as the essential technologies in cloud computing [8, 81].

Container technology is having a massive impact on distributed systems and cloud com-

puting. Containers use operating system level virtualisation technique supplying virtual run-time environment with an isolated process and networking layer. Two containers can run on the same operating system regardless of sharing resources as they have their networking layer, processes and volumes. [9.] Initially, the **chroot** command in Unix systems (established in 1979) was the motivation for containers nowadays. After that, in 1988, **jails** was implemented by extending **chroot**. A long time since then, **zones** was released with capability improvements in 2004. Containers were born after more than 6 years since then, right after the release of Solaris 11. At the same time, HP and IBM also presented similar efficient products. Nevertheless, as Linux came out as a superior open platform, containers lead the market since then with the new name - Linux containers (LXC). [8, 82.]

Virtual machines are the products of hypervisor-based virtualisation. They are usually used in cloud service platforms such as Enterprise VMware, Rackspace, Amazon Web Services (AWS) and so on. However, there are many extremely successful public cloud computing systems like Google, IBM/Softlayer, and Joyent utilising containers instead of VM. [8, 81-82.] Figure 2 illustrates the structure of VM and containers for better understanding the differences between them.

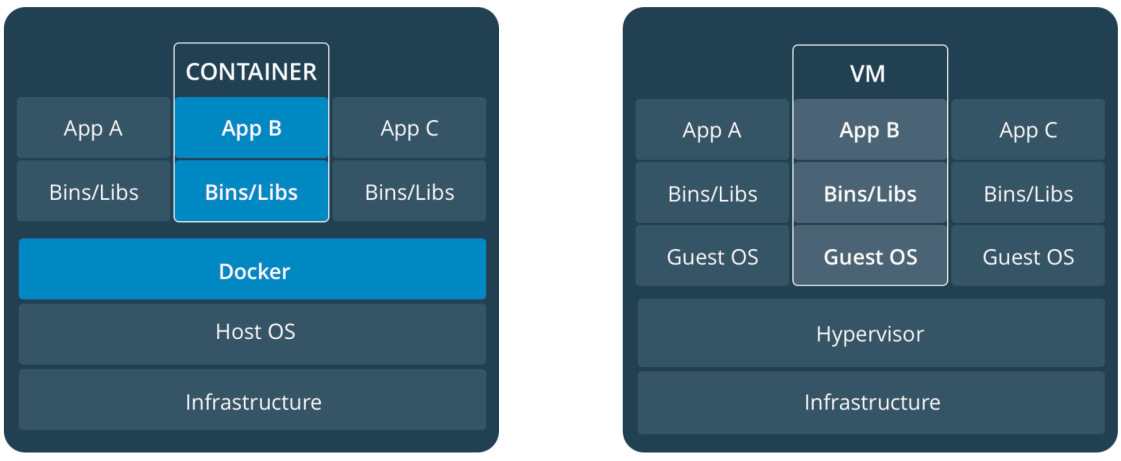

Figure 2: Virtual machines vs Containers. Reprinted from What is a Container (2018) [10].

As shown in Figure 2, both VM and containers contribute the same benefits in allocating and isolating resources but operate differently. They have an abstraction in different layers in distributed systems. Containers are an abstraction at application level while VMs are at the hardware layer. On the one hand, hypervisor technology turns single server instances

into multiple ones. On the other hand, various containers share the same kernel instance on the same machine. This is the essential difference to make containers extraordinarily lightweight and customisable compared to VM. Also, size of containers (about tens of MBs) are incredibly small compared to VMs (up-to tens of GBs). Last but not least, the starting time of containers is counted in seconds, whereas it can be minutes with VMs. In general, containers are intensely portable and efficient. [10.]

2.2.1 Docker containers

Containers provide an entirely isolated and self-sufficient application to run in a shared environment. Docker has produced an instinctive expansion in container technology space. [1, 34-35.] By supporting a separated, lightweight run-time along with API for supervising the containers and container images, Docker platform is a successful extension accommodated to existing container solutions [4, 324].

Docker containers are lightweight LXC supporting consistent development and deployment. Regularly, in the past, after successfully developed new applications using new and modern technology got stuck in deployment phase as there was no cloud service provider to support a specific language or framework. Docker solved the problem with many more benefits. Docker packs the applications with needed dependencies and running environment into a standard and self-contained unit to run smoothly in every environment. [1, 34.] The problem of conflicting and missing dependencies has been clarified with Docker. Moreover, moving from one operating system distributions to another is no longer a problem by using Docker. Currently, Docker has become a defacto format because of an extensive support from software development communities. [9.]

Most of the popular cloud service providers such as Google Cloud Platform, and Amazon Web Services are gradually increasing support for Docker. Docker is a powerful tool from which both developers and operators obtain enormous benefits. In addition, the architecture behind Docker is especially simple indicated by similarity to the client-server pattern. [11, 10.] Figure 3 on the following page demonstrates the overview of Docker’s system architecture.

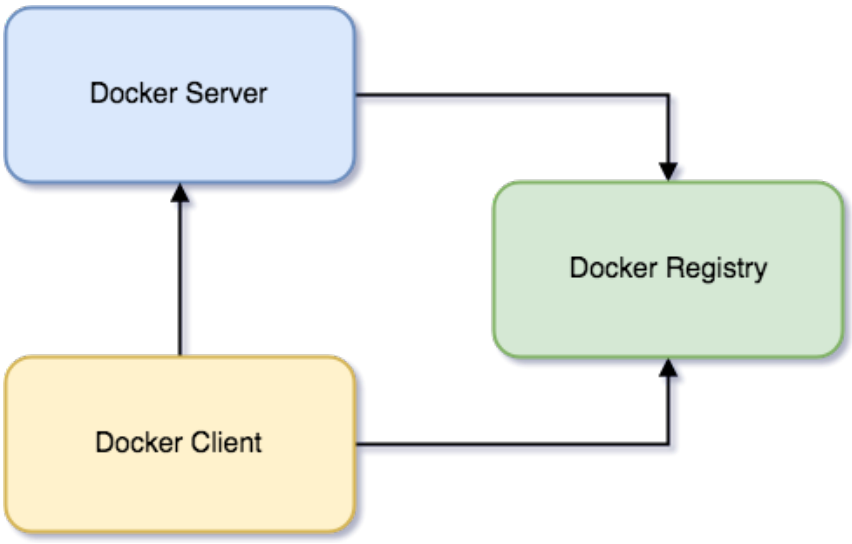

Figure 3: Docker Architecture.

Figure 3 illustrates three fundamental components in every Docker system: server, client and registry. Docker servers are responsible for container operation and management. By making use of kernel processes, Docker servers achieve required stability, and high performance. On the one hand, Docker servers take care of container environments. On the other hand, Docker clients deliver processing instructions for servers. The registry in this picture plays an essential role in storing and managing Docker images. Commonly, Docker clients initially push images to registries. Then, Docker servers are ordered to pull images from registries and run on hosting instances. [11, 10-11.]

Docker integration is possible to accomplish by adopting a workflow which not only matches the organisation and team structures, but also the product itself. Moreover, Docker is a tool to indirectly resolve the problems in communication between teams in a project. Every software development flow commonly contains the following steps: implementing, building, testing, packaging and deploying. In implementation step, Docker offers two means to control the changes: filesystem layers and image tags. While filesystem layers method records the changes produced in building container process, image tags approach handles versioning problem of built images. Both solutions pointed out and solved the common issues while developing non-containerised applications. More than that, Docker supplies a unified mechanism for building containers which makes the build procedure considerably clear and transparent regardless what technology stack is used. In addition, in the light of building containers effortlessly, the testing process is accomplished without the need of any framework and the packaging procedure is minimised to

nothing as every built image is a deployable product. Lastly, deployment workload is remarkably reduced by Docker command line interface tool. The containerised applications can now be deployed in just a single command line. In general, Docker is a solution to scale down the complication in both deployment and development stages. [11, 18-22.]

2.2.2 Container registries

Docker images are built based on the definitions and instructions included in a Dockerfile [4, 324]. According to the container best practices, Docker images typically are not built on servers. Nevertheless, built images should be stored in a place for servers to pull and effortlessly run containers. The places storing container images are called container registries. There are several approaches to adopt container registries. However, the most common methods are public and private registries. [11, 51.] In practice, almost all systems make use of both public and private ones to carry out the deployment process seamlessly and reliably [9].

Public registries are the storages to keep and supply images that the container community wants to share, such as operating system distributions, databases or ready-made base images to continue building on top of them. There are two commonly-used public registries in the market: Docker Hub and Quay.io. Both registries are good starting points for delivering containerised software. However, they are only available on the Internet, not in local networks where deployments are handled. Consequently, if applications are massively dependent on base images, the deployment can be affected by network latency which somehow delays the deployment process. This issue can be minimised by bettering image structures to contain lightweight and easy to be transferred layers around the Internet. [11, 51.]

Another solution for images storage is to host or utilise private container registries in the cloud. Private registries require a certain amount of authentication to access and interact with images stored inside it. Moreover, they also offer access management to specify roles and rights for users. This feature makes private registries more secured. Docker supports private registries by releasing the docker-registry project as a self-hosting solution with emphasises on pushing, pulling and searching images. [4, 52.] Other services in

the cloud such as Google Container Registry provide better functionalities in a graphical user interface for containers management [6].

2.3 Cluster management

Software packaging issue when using modern software stacks is resolved with container technology. However, there are still issues in containerised application cluster management in cloud particularly in Platform as a Service (PaaS) environments. [8, 84.] Noticeable problems are scheduling and scaling containers. Moreover, in microservices systems, maintaining communication between services (containers) is extremely tough. Besides, a solution to eliminate these obstacles is to start using Kubernetes which eases not only the deployment but also the cluster management process. [12, 41-42.]

At the June 2014 Google Developer Forum, Google introduced Kubernetes, an open source tool for Docker container clustering management. According to Google, Kubernetes was designed at planet scale that allows Google to execute billions of containers a week. Kubernetes guarantees to scale products without expanding operations team. Additionally, Kubernetes is flexible and mature with the applications to deliver high-quality products persistently. With Kubernetes, not only software development but also operations are simplified to produce a high-quality product efficiently and economically. [8, 84.]

In Infrastructure as a Service (IaaS) deployment and management approach, a number of VM in the cloud are ordered by an administrator. Then, runtime environment is set up in these instances to advance to ready state. In this method, the operator can choose VM configurations and scale them horizontally or vertically. However, the workload in acquiring and setting up VM are endless when the scale grows to hundreds of instances. Meanwhile, in PaaS, an administrator can select run-time configurations acquired from service providers, and source code is transferred to the runtime environment and operated inside it. Scaling is not the issue here, but the constraint here relates to the limited number of runtime VM. Kubernetes is an ideal product combining the purity of PaaS and elasticity of IaaS. [13, 129-131.] All those constraints and limitations are resolved in Kubernetes in which a container centralised supervision ecosystem was supplied. Infrastructure configurations such as computing, networking and storage resources are automatically modified

based on user's demand. Consequently, this powerful platform provides the flexibility in combining the usage in different cloud service providers or moving around between them. [14.]

Kubernetes offers a wide range of features that support containers management and discovery. First, Kubernetes automatically manages computing resources for containers by providing requirements and constraints to them without scarifying the availability. Kubernetes smartly combines high demanding and productivity workloads to raise usage but also saves resources for others. In addition, Kubernetes is designed to scale applications horizontally. Product scaling can be achieved either manually with a simple command or automatically based on resource usage. Moreover, Kubernetes provides self-healing feature which automatically cures the failed containers by replacing and rescheduling nodes. In self-healing period, Kubernetes only publishes ready-to-serve nodes to maintain the stability of the product. Kubernetes also provides automated rollout features to guarantee the availability of the product. Additionally, Kubernetes roll-back the changes automatically in case of failures occurred inside containers. Having a huge contributor community, Kubernetes continuously delivers new features and improvements that make container configurations and management easier and more efficient. [5.]

Immediately after Google's introduction, many organisations signed Kubernetes and Docker containers to be a core cloud deployment solution. There are many big names listed as Kubernetes supporters such as Microsoft, VMware, IBM and Red Hat. There are still many discussions around Docker and Kubernetes of their standing in cloud technology. [8, 84.] However, with a large supporting community, Kubernetes has a promising future. Moreover, eBay, Pokémon Go, Sound Cloud have production-ready products using Kubernetes. These are evidence to show that Kubernetes is built with 15 years of experience of executing production workloads at Google. [5.]

2.3.1 Kubernetes architecture

Scalable container coordination and software stack management are somehow impractical without Kubernetes. Infrastructure resource is utilised at application or service level to maintain the high availability and portability of the application. In addition, Kuber-

netes offers the **kubectl** tool, a concise and efficient API, to interact, manage and deploy software. [15, 43-44.] Application containers are distributed and scheduled automatically across a cluster. Figure 4 illustrates Kubernetes's core architecture.

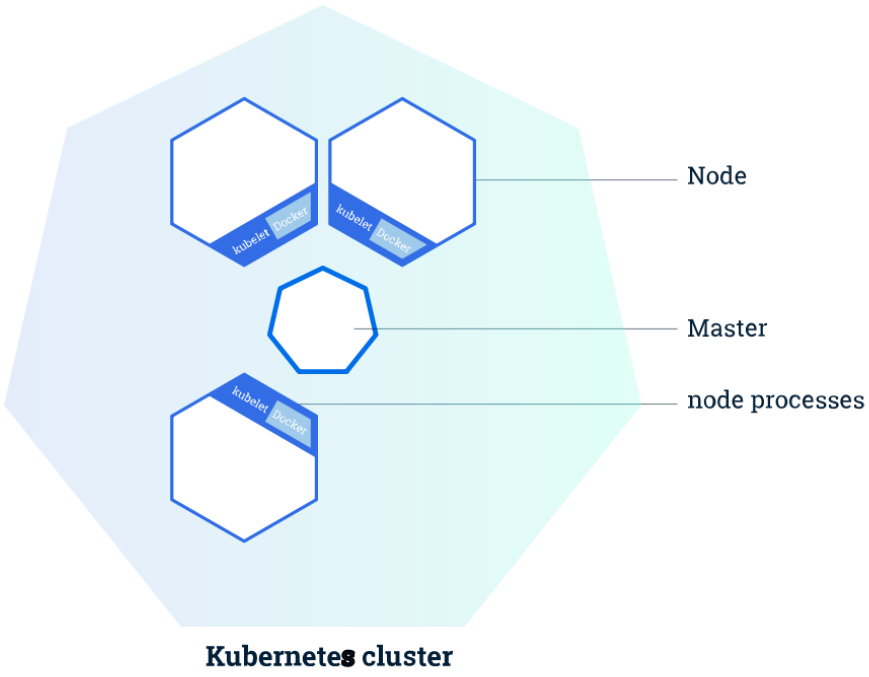

Figure 4: Kubernetes Cluster. Reprinted from Kubernetes documentation (2018) [16].

As can be seen from Figure 4, Kubernetes cluster is a group of a *master* and several nodes. *Master* is the head of the cluster. There is an API server running inside *master* to maintain the communication between *master* and the outside world for interactions related to getting and specifying the required conditions. A scheduler is also included in *master* along with API server to organise containers into target nodes. The last component inside *master* is a lightweight shared configuration store. Moreover, the cluster's states are recorded in this place for easy monitoring and modifying. [12, 45.]

In Kubernetes cluster, there are multiple nodes which are worker instances. They can be either VMs or physical machines. Every node contains at least these two essential services: kubelet and proxy. Kubelet is the node processor which connects to API server to provide cluster status and execute new tasks planned by the scheduler. The node's proxy coordinates traffic to the appropriate containers inside it. A service called cAdvisor might also exist in nodes to combine information related to computing resource and network analysis. Nodes are also known as minions in Kubernetes clusters. In a node, there are

also Pods which reside with essential services listed above. [17, 9.] Figure 5 describes the existences of Pods within a node.

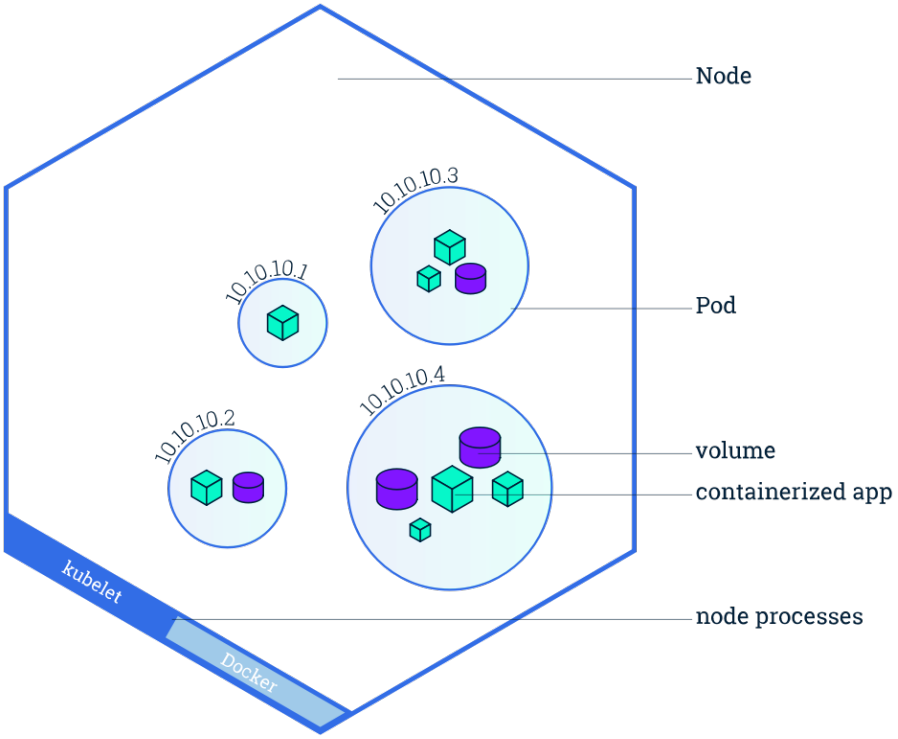

Figure 5: Kubernetes Pod. Reprinted from Kubernetes documentation (2018) [16].

As illustrated in Figure 5, a Pod is a group of containers and volumes which generally share a unique IP address. Containers inside Pods are consistently scheduled, located and run in the same background on the same node. Pods are nuclear components in every Kubernetes cluster. Pods are constrained to the scheduled node and stay there until being terminated or deleted. Pods are assigned to another free node if the tied node has collapsed. Another component of a Pod is Volume. Volumes in Pods are durable as they are not tied to containers. Moreover, any container can access data inside Volumes as the existence at Pod layer. These Volumes can exist in different types and have an ability to integrate existing persistent storage methods in the cloud. [17, 10-14.]

Service is an approach to expose containerised applications in Kubernetes platform to the outside world. A Service serves as a representative for one or multiple Pods to communicate externally. Each Service in the platform is located in a virtual network by an IP address. [12, 40.] Figure 6 on the following page illustrates how Services reside inside a cluster.

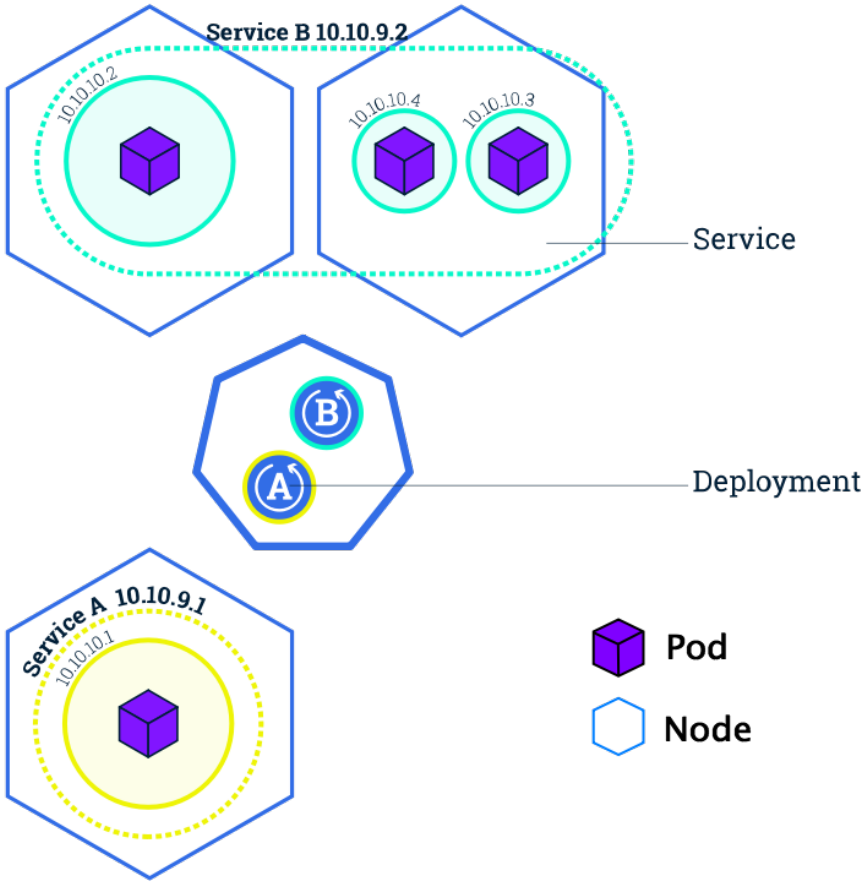

Figure 6: Kubernetes Service. Reprinted from Kubernetes documentation (2018) [16].

As can be seen from Figure 6, a Service is the entry point for users to access application resource. Consumers do not need to know the Pod's address of running software to communicate. They can interact with exposed address and port to get the direction to a proper Pod. Hence, the high-level reflection on a group of software with Pods specifications is a Service. [12, 40.]

A good distributed system platform always provides a solution for smooth scaling and effortlessly maintains the high availability of software. Kubernetes supports this feature by providing Replication Controller in the ecosystem. Replication Controller handles the number of running Pods in an organised way. Generally, Replication Controller guarantees that a setup quantity of Pods is working all the time. Moreover, the scaling issues are gone with Replication Controller as administrators only need to change the replica num-

ber and automatically, Replication Controller executes the needed actions to achieve the goal. Replication Controller is a good solution for every microservices systems by offering innovation in scaling and controlling services seamlessly and effortlessly. [12, 40.]

Application deployments in Kubernetes platform are handled by creating Deployment objects. Deployment objects carry supplementary information for containerised applications creations or modifications including an instruction of scheduling Pods. [18.] The existence of Deployment in Kubernetes cluster is illustrated specifically in Figure 7.

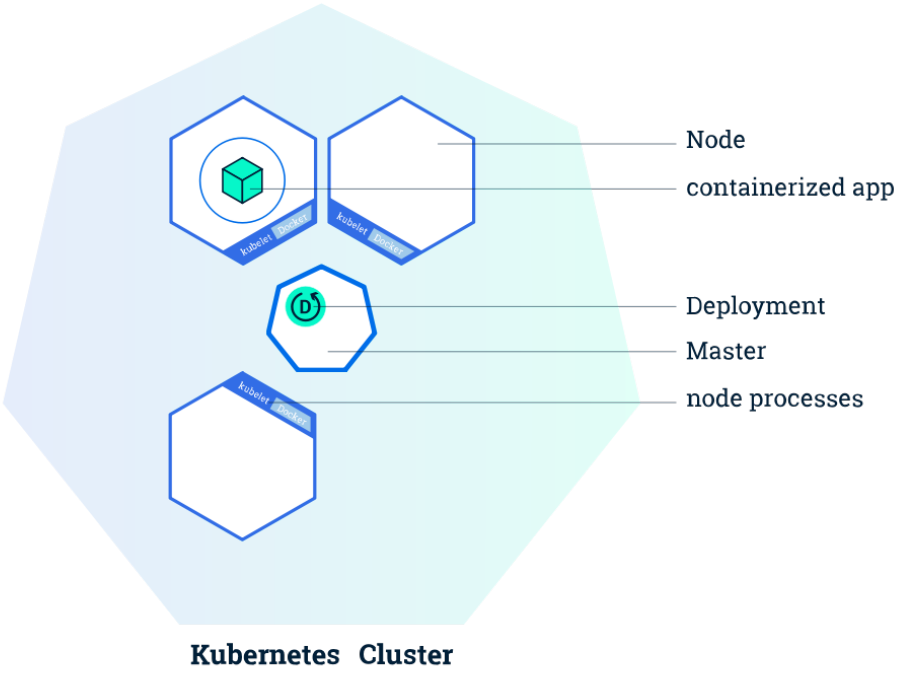

Figure 7: Kubernetes Deployment. Reprinted from Kubernetes documentation (2018) [16].

As can be seen from Figure 7, Deployment objects are kept in *master*. When a Deployment object is created, *master* schedules Pod creations onto available Nodes in the cluster. Once the Pod is created, the container image is pulled and run on it. Pods and containers inside it are being progressively monitored and scheduled to new Node in case of its failures, once they are all created. Accordingly, the Replication Controller is created along with Pod to handle the scaling of the application. In general, by starting up needed Pods and scheduling them to run consistently across Nodes, Kubernetes Deployment objects offer a new method in application deployment and management. [16.]

2.3.2 Kubernetes Objects

Kubernetes platform contains a wide range of components including Pod, Service, Replication Controller and so on. Consequently, the management of the components might be a heavy task. Fortunately, Kubernetes provides a smart method to effortlessly and smoothly manage the operations of components in a system, Kubernetes Objects. In general, they are components that live consistently in Kubernetes systems. Moreover, they are consumed to depict states of the cluster. Once Kubernetes Objects are created, the existence of the objects are guaranteed by continuously managing, reporting and scheduling. [19.]

Names and UIDs are the ways to identify Kubernetes Objects. Names are strings defined and supplied in object creation process, and UIDs are strings produced by Kubernetes system to define a particular object. In Kubernetes systems, only one given name is allowed to endure for an object in a given kind. Similarly, every object possesses a distinct UID over its lifetime. [19.]

Kubernetes also offers a method to host multiple virtual clusters in the same physical cluster. Namespace is an abstraction of virtual clusters. In general, the namespace is at the higher level than the name to differentiate a group of objects and avoid name duplication. Practically, namespaces are applied to differentiate between development environments. Same names can be used for objects across namespaces but not in the same one. [19.]

Kubernetes Objects can also be attached several labels. Labels are key-value maps to indicate object's aspects. Moreover, labels ease the objects organisation and groups selection. Labels are attached either at object establishment or later during running time. Label feature is beneficial in grouping resources such as Pods to expose through Service or aggregating logs between different Services. Different from names and UIDs, labels are not distinct. In fact, labels are produced to be utilised by multiple objects. Label selectors accomplish the process of grouping objects which carry the same labels. In another word, label selectors are the key-value explanation to group resources containing the matching labels. Equality-based and set-based approaches are pointed out as two means of selecting objects using label selector. [19.]

Using labels is a solution to attach identifying information to Kubernetes Objects. Nevertheless, annotation is the approach for defining additional metadata of objects. Similar to labels, annotations are key-value pairs, but the data inside annotations are not used to recognise or select objects. Commonly, annotations express information such as timestamps, release numbers or contact information of administrators. [19.]

2.3.3 Kubernetes Object managment

In general, Kubernetes Objects describe the cluster's state. Specifically, there are two crucial elements which are *spec* and *status* in every Kubernetes Object. The desired state of every object is contained inside *spec*. Objects creator should supply proper parameters for the *spec* section. Besides, the existing states of the objects are described inside *status* unit. This information is provided by Kubernetes platform. The actual *status* is continuously and automatically managed by Kubernetes system to match the desired state. [19.]

The object creation process is accomplished by supplying object's desired state, and necessary metadata such are name, labels and annotations to the target Kubernetes cluster. The communication to clusters is handled seamlessly through Kubernetes API. There are several required fields to be specified in every Kubernetes object which are *apiVersion*, *kind* and *metadata*. The *apiVersion* parameter points out the version of API to use in the creation process. Meanwhile, *kind* field defines the category of an object such as Pod, Service, Deployment and so on. Lastly, *metadata* describes the criteria to quickly identify objects, later on, containing a name, namespace, labels and annotations. Apparently, the *spec* field content depends on which kind of object is being created. The format for *spec* of each kind can be found in Kubernetes API Reference document. [19.]

Object deployment process starts with converting the specifications into JavaScript Object Notation (JSON) format and included in request body when using API directly. Fortunately, Kubernetes also offers **kubectl**, a powerful Command-line Interface (CLI) to ease the transmission in a convenient mean. Commonly, object parameters are defined in YAML Ain't Markup Language (YAML) format files and provided to **kubectl** to deploy into clusters. [19.] Nevertheless, there are multiple approaches for object creation and

management by **kubectl** including imperative commands, imperative object configuration and declarative object configuration. [20.]

Imperative commands usage in practice is parsing object's parameters directly through **kubectl** operation commands in the shape of arguments and flags. This approach contains considerable advantages as cluster start-up is rapid and commands are simple and easy to learn. However, this method does not keep track of history configurations or review changes before operations. When imperative object configuration is in use, object specifications are contained in YAML format files and being parsed into **kubectl** command with a specific operation such as create, delete and update. Following this approach brings several improvements such as history configurations can be recorded and reviewed in a versioning system and object's templates are provided for later usages. Nevertheless, this technique requires knowledge about object schema and YAML format and every cluster modification must be mirrored into configuration files to avoid getting lost in the next deployments. In declarative object configuration, object configurations are also saved into files. However, the operation is automatically handled and implied by **kubectl** without explicit calls. This approach brings considerable benefits as object's adjustments are preserved, and operations can be handled at the directory level. Besides, this method contains some downsides such as debugging process takes a long time to get the understanding of unexpected events. Also, updating objects partially requires the system to combine and apply states in complicated means. [20.]

Every Kubernetes object management approach contains multiple benefits and drawbacks. Nevertheless, only one approach is recommended to manage objects in Kubernetes. Combining and coupling methods lead to tremendous issues such as confusion in objects management and unexpected status and actions in the system. [20.]

2.4 Infrastructure management

The existence of virtualisation technologies and cloud computing is hugely beneficial in infrastructure management work. Virtualisation technologies separate hardware resources from what they offer such as storage, compute and networking concepts. This method utilises hardware capability distribution to a higher level as the same resource can be

shared among different applications. Also, resources can be provisioned rapidly, seamlessly and efficiently beyond hardware pool within available capacity in the infrastructure. Besides, cloud computing brings several advantages in operations work. For instance, resources allocations and provisions are accomplished not only by using scripts but also software systems. Moreover, computing resources are allocated based on users' demand. Both virtualisation technologies and cloud computing assists the hardware resource administration in the manner of availability and responsivity. [21, 1-3.]

Adopting new technologies without changing the working methods is commonly seen in many teams. As a result, they handle a huge workload by applying old working patterns without using effective means to take care of it. Moreover, as the resources are created on demand, there might be a situation too many actively operating servers to manage. This approach can lead to the result of unexpected patches or fixes. Consequently, the infrastructure contains a chain of inconsistent servers that are considerably tough to run automation tools. Nevertheless, by embracing Infrastructure as Code method, infrastructure management becomes painless. Moreover, consistency and well-maintenance are always controlled even in a large infrastructure. [21, 4-5.]

2.4.1 Infrastructure as Code

Infrastructure as Code (IaC) is a mechanism to build and manage dynamic infrastructure by making use of virtualisation technologies, cloud computing and server automation. This approach abstracts infrastructure, management tools and services to be a software system. Moreover, IaC is also applying software engineering practices to administrate infrastructure in an organised and secure manner. By adopting this method, infrastructure management teams always have a clear view of system's status, and changes are made in a reliable procedure. [21, 5.]

Taking up IaC brings enormous advantages. Firstly, IaC supports continuous changes. Infrastructure modifications should not be a restriction or an impediment. Secondly, repeated and periodic work are exempted to make the team focus on relevant tasks improving their abilities. Moreover, resources are automatically provisioned and managed based on user's demand without extra work from operations team. Additionally, there are

always failures in the system. Instead of only preventing failures, following IaC helps recover processes become considerably rapid. More than that, the workload of operations team is reduced significantly as infrastructure changes are always being scheduled. Another benefit of following IaC approach is that the system is remarkably reliable due to enhancements are built small and continuously. Finally, system issues during up-running time are always resolved by practical implementations and measurements instead of discussions in meetings. [21, 6.]

2.4.2 Principles of Infrastructure as Code

In dynamic infrastructure management, there are always disputes. For instance, servers are consistent at creation time but might not always be. Tactical fixes can lead to the variation between servers. Consequently, operation teams are not using automation tools consistently and efficiently. Additionally, the system itself can be collapsed sooner or later as there might unexpected forces destroying the system such as the storage's capacity is filled by logging files or hardware failures make several servers in downtime status. Fortunately, principles of IaC support conquering these challenges. [21, 6-10.] The key principles of IaC are explained in the following sections.

Reproducibility

Every component of the system should be reconstructed seamlessly and faithfully. The needed information about components should be recorded in scripts and provisioning tools. Reproducibility principle minimises the risks of continuous modifications and scaling issues. [21, 12.]

Consistency

Infrastructure elements offering the same service should be built and rebuilt consistently. Normally, there might be modifications after creation time. However, maintaining the consistency of the infrastructure plays an important role in executing automation later on. [21, 13.]

Repeatability

Changes in infrastructure should always be able to repeat as of Reproducibility principle. Modifications should be recorded into scripts or configuration tools rather than run commands manually on specific resources. Scripting culture must be followed strictly to gain the efficiency and reliability of infrastructure management. [21, 13.]

Disposability

Infrastructure components can be collapsed unexpectedly regardless of any announcement. The problem can be the result of hardware failures, scaling issues or even reducing storage size. If every element of infrastructure can be disposed of affecting others, the upgrading or reconfiguring resource processes can be achieved efficiently and effortlessly. [21, 14.]

Service continuity

The availability of a service hosted in the infrastructure should consistently be maintained regardless the absence of one or multiple elements in the system. The dependencies of the service should always be recognised and decoupled from the infrastructure. Despite the changes in the infrastructure, the service itself should be able to handle requests from users. [21, 15.]

Self-testing systems

Automated infrastructure testing is one of the essential practices applied from software development. The most valuable advantage of adopting this is to receive feedback on every change rapidly without breaking the system. This principle encourages teams to improve infrastructure at high speed regularly. [21, 15-16.]

Self-documenting systems

By adopting IaC, the procedure for implementing changes are saved in the scripts or management tool handling the process. Clearly, they are all self-documented. Consequently, the documentation outside of these scripts or tools is considerably reduced by

only describing the tool's entrance and setting up instructions. [21, 16.]

Small changes

There are several advantages to release tiny and accumulative modification rather than big one. For instance, small changes ease the testing process to guarantee the solidity of the system. Moreover, in case of modification mistakes, determining, repairing or reversing changes is more effortless and efficient. Additionally, operation teams are motivated by delivering fixes and improvements continuously and speedily. [21, 17.]

Version all the things

IaC offers the way to configure infrastructure by using the source code. As a result, keeping versions of the configurations is vital. There are multiple reasons behind versioning changes to the infrastructure system. For instance, the modification history is all being kept in version control system. As a result, in case of a change got broken, the reverting work size is minimal. Additionally, every team member is aware of changes due to its visibility in version control system. Lastly, by using version control system, automation system integration is considerably effortless, and changes are deployed at a high pace. [21, 18.]

2.5 Automation of software engineering

The process of releasing applications is commonly the most anxious stage across development teams. Clearly, this step is risky and frightening as it affects directly to users consuming the applications. [22, 4.] Today, the complexity of software is considerably high, and independent of its size. Thus, manual deployment is selected as the primary method by many firms. This practice is noticeably time-consuming and unreliable as the process entirely depends on human efforts. [22, 5.] Additionally, the applications are commonly not deployed into staging environment frequently. The software is only deployed to this environment after the completion of the development process. This practice isolates testing, deployment and release activities out of development cycle. Therefore, considerable defects are found during testing process which obviously influences significantly to the release schedule. [22, 7-8.] Another bad habit commonly seen is that configurations

of production environment are handled manually by the operations team. This approach is potentially dangerous as the deployment can be successful many times in staging environment but not in production environment. [22, 9-10.] Many organisations are looking for a solution to minimise these risks. Meanwhile, they are using above practices to deal with unexpected issues every day and could not get out of it. [22, 5.]

Software delivery procedures should be handled regularly and rapidly at low-cost and high reliability. The described goals can be achieved strictly and effortlessly by adopting automation engineering approaches. The main idea behind these methods is to offer a set of practices making the software releases autonomous and continuous. Not only product's feedback but also system feedback are returned rapidly by applying these techniques. This method makes the project teams response with customers faster to enhance the experience over time. [22, 10-16.]

2.5.1 Continuous Delivery

The software market is much more competitive today in comparison to the last two decades. Consequently, the necessity of developing and delivering high-quality applications at a rapid pace is always placed at the highest priority. [23, 3909-3910.] Continuous Delivery (CD) is a powerful method to keep development team delivers reliable software in a short period and guarantee that the software can be released safely. CD, a software automation engineering approach, allows enterprises to constantly bring software enhancements to the customers in a fast, efficient and reliable way. Moreover, CD brings considerable advantages not only to development team but also product quality and business aspects. [24, 50.] Figure 8 on the following page indicates six key benefits by adopting CD.

Figure 8: CD's benefits. Reprinted from IEEE Software (2015) [24, 52].

As illustrated in Figure 8, embracing CD in the development process is extremely beneficial. These essential benefits are specified in more detail in the following sections.

Accelerated time to market

Without adoption of CD, the applications release time can be several months. Fortunately, CD practices have decreased this considerably. By applying CD in the development process, the software can be released several times a day without any obstacles. Moreover, this helps organisations rapidly deliver new features to the market to keep the competitiveness at a high level. The development cycle from concepts to practices nowadays is less than a business working week compared to months in the past. [24, 52.]

Building the right product

In the old days, development teams were often spending a considerable amount of time to implement features but only recognised redundant ones after a big release. Nevertheless, by delivering products quickly and regularly to customers, developers receive customers' feedback faster. Hence, the team can assess what are the essential and appropriate features to focus on and implement. Clearly, the workload is redistributed, and the product is close to customer needs. [24, 52.]

Improved productivity and efficiency

CD automates the testing procedures and eases the deployment process. In general, the story of starting up testing environment and fixing it is redundant with CD. [24, 52.] With CD, old versions can be run in a separate environment to confirm the changes or reproducing defects if needed [22, 17]. More than that, releases can be automatically carried out by a button click. The whole team become extremely productive and efficient in working on interesting parts instead of doing repetitive tasks. [24, 53.]

Reliable releases

Manual deployment and release are commonly risky because of failures related to the human factor. Frequently, there are mistakes during deployments in this approach. Thus, the releases are obviously unreliable and often crashed. [25.] Fortunately, these risks have been noticeably reduced by CD. By adopting CD, the release procedure is automatically and continuously guaranteed. Hence, most errors are caught before production deployments. Moreover, by delivering product continuously, the difference is small enough to identify and fix application bugs quickly. [24, 53.] Consequently, the stress is minimised in release day not only by the short release duration but also the simplicity of the process [22, 20-21]. Importantly, CD offers a way to automatically rollback in case of release failures. This feature is extremely beneficial to consistently keep the product in good shape and downsize the risky changes. [24, 53.]

Improved product quality

With the assistance of automated testing, the number of defects is reduced remarkably (more than 90% in several cases). In the past, the workload spent on fixing bugs could be one-third of the whole team. Nevertheless, by adopting CD, rarely any developer works on defects discovered by end customers. Additionally, as the release process is simple and is frequently handled, and bugs are resolved rapidly. Moreover, the manual configurations failures are also resolved as of the effective automation deployment processes. [24, 53.; 25.]

Improved customer satisfaction

The low quality and release problems were the challenges to make a close-knit connection between different departments in an organisation. However, with the help of CD, the software is now deployed continuously, frequently, reliably, and securely. The relationship between teams has been massively improved. Not only developers and operators, but also testers and customer supporters are happy with their job. After CD adoption, the collaboration has increased dramatically which also makes the quality product at the same time. As a result, customers are always served with the best services. [24, 53.; 25.]

2.5.2 Continuous Delivery pipeline

The main concept behind CD is developing a process that delivers new functionalities to customers frequently, accumulatively and reliably. CD ensures the quality of software and improves the productivity of development cycle by the stable and straightforward deployment process. All these benefits and ideas are implemented practically by building a Continuous Delivery pipeline. CD pipeline describes different stages of application delivery process. Each stage focuses on assessing and giving feedback based on the quality of the product at that point in different aspects. The pipeline is different for various applications. Nevertheless, every pipeline should commonly have these stages: integration, building, testing and deployment. [26.] Figure 9 illustrates a sample continuous delivery pipeline.

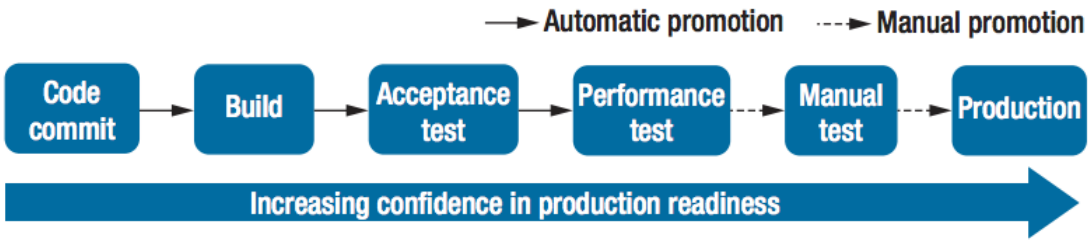

Figure 9: A sample CD pipeline. Reprinted from IEEE Software (2015) [24, 51].

As can be seen from Figure 9, this sample CD pipeline contains six stages and the advancement from one step to the next one is able to be either automatically or manually. These stages are explicitly explained in the following sections.

Code commit

Consistently, every time developers push their changes to the source code repository, the pipeline is triggered to compile the source code and to run unit tests. The CD pipeline stops whenever this stage occurred an error and informs developers about those failures. After fixes from developers are checked-in and pushed to the remote repository, the pipeline is triggered to start a new run. This stage presents initial feedbacks instantly on the changes made by developers. If everything completes nicely, the pipeline moves to build stage automatically. [24, 51.]

Build

Unit testing is conducted again in build stage to generate a code coverage report. The build stage also takes integration tests into account along with the analysis of source code. After that, applications are built into a batch of binaries and upload them to a repository for bettering the distribution. This step helps in finding and resolving considerable defects occurred when the artifacts were different between development, testing and production environment in the past. The pipeline automatically moves to acceptance test stage after passing all the steps successfully and smoothly. [24, 51.]

Acceptance test

After successfully passed the build stage, the pipeline sets up an environment similar to production one and deploy the application there to execute multiple test suites. In the past, the environment creation time can be up to several months (with big projects) or at least half a day (with small applications) as they are handled manually. The duration of supplying, configuring servers and deploying applications is now cut down to several minutes by using CD pipeline with automation scripts. Basically, this stage primarily guarantees all application requirements are satisfied. Similar to previous stages, the pipeline stops in case of failures happened and notify developers. Otherwise, the pipeline goes on to performance test unquestionably. [24, 51.]

Performance test

In the past, performance tests were only carried out before big releases but not in the development cycle as environment creation was considerably time-consuming. However, by using automation scripts, the CD pipeline now automatically creates a new environment and deploy the application into it effortlessly. After that, the pipeline executes a series of performance tests in the created environment. The result is then being reported to the development team to assess the effects related to application's performance made by those commits. Clearly, investigating and fixing problems at this stage is extremely reasonable compared to doing so before big releases. After the performance test result reaches an acceptable level, the pipeline is manually triggered to the next stage - manual test. [24, 52.]

Manual test

Though automated testing is effectively inclusive, there is still a need for manual testing. At this stage, testers might be technical or business people. However, they do not need to set up the environment themselves. The pipeline automatically creates the test environment, deploys the application, then notifies via email for instance on the necessary information to access it. After the testing are accomplished successfully and sufficiently, the quality of binaries is guaranteed. Consequently, the built binaries batch is advanced to be a release candidate and ready to be deployed to the production environment. [24, 52.]

Production

Lastly, by clicking a button, the application is automatically deployed into the production environment. The story of deployment failures is resolved by using CD as there is no manual deployment step. Moreover, the deployment scripts have been checked all over again in previous stages to confirm there is no problem. With CD pipeline, reliable software is delivered to customers rapidly at a high pace. [24, 52.]

Undoubtedly, CD brings numerous benefits into the development process and cycle on a different scale. CD gains a considerable amount of attraction and investment into it nowadays. However, CD implementation still contains noticeable disputes. The obstacles might come from organisations, processes, or techniques. The most significant obstacle

relates to organisation structure as the release actions are accomplished by multiple departments with different access and working culture. The collaboration between teams should be encouraged, and the company's structure might be reformed to adopt CD practices effectively. Additionally, the process should be concise and flexible enough to promote the features to be released in time. Last but not least, building a CD pipeline requires integrating different tools and technologies. Consequently, enclosed dependencies might also be an obstacle at a specific time. The project should follow broadly approved standards and build an intense vendor ecosystem to apply CD practices smoothly. In general, there is a huge demand to deal with such as organisation understanding, development researches and practices to overcome all these challenges. [24, 53-54.]

3 Case study

3.1 Case summary

This case study was carried out to solve the problem in integration and deployment process in a software development team. The case company has about 20,000 employees allocated in multiple countries around the world. The company started a project to produce a new e-learning platform for their internal use. Nevertheless, the development team concentrated on the development process, but did not take source code integration and deployment procedure into account. Consequently, the anxiety feeling was built up in every team member about how to achieve the release deadlines ahead. Besides, developers did not strictly follow the coding conventions resulting in an unformatted code-base which is hard to maintain in the future. Additionally, the deployment was still handled manually which lead to time-consuming releases and potential failures in the procedure.

The objective of this case study is to solve all the challenges the development team was tackling with every day - source code integration and manual deployments. Fortunately, there were several solutions to get rid of those troubles. The first solution could be developing automated scripts for developers to run before pushing their commits to the remote repository. In addition, the deployment scripts could be implemented to avoid human mistakes in the release procedure. However, this solution still contains several manual tasks that may lead to many other potential failures in the future. A better solution was to implement a CD pipeline which not only increases the development productivity, but also resolves the manual task issues. The pipeline can contain multiple stages, and troubles are resolved progressively. The latter solution was chosen to be implemented in this case.

By having a clear plan and a good design, a proof of concept of CD pipeline was implemented successfully. The pipeline was running by integrating multiple services such as CircleCI, Codecov, GCR, and GKE. The pipeline workflow was configured based on the initial design without any obstacle. In general, the pipeline was set up successfully, and proved the concept behind CD to deliver product in a fast, efficient and reliable manner.

3.2 Case challenges

The architecture of the application was designed to follow microservices as the system contains considerable integrations between services which have already been used inside this company. In this software, services were decided to be packaged as Docker images. The product was expected to be deployed and hosted in a cloud computing platform. At that point, the project was pretty close to the Minimum viable product (MVP) release. However, the team was focusing on developing features and did not pay enough attention to integration and deployment processes. Consequently, the whole team was being stressed and confused about the deadlines ahead.

The first and foremost challenge was about code-base integration and testing. Though there were several unified conventions in the team related to code formatting, developers still easily had many mistakes in development procedures. As a result, the peer-reviewing processes took a remarkable time, and productivity of the whole team decreased. Additionally, the testing phase was also handled manually by developers before committing the changes. As a consequence, the tests were run differently in developer machines. More than that, developers often forgot to run tests before commits because of time-consuming. Consequently, potential defects started rising. Importantly, coverage testing was not reported at all and developers has no way to audit the quality status of the current code-base.

In addition, there were several impediments at that point in deployment stage. Deployments were handled manually. Clearly, it was extremely time-consuming (at least half a day) to deploy new application version. The combination of manual deployments and manual infrastructure configurations led to the nervous feeling when a developer in the team was assigned to deployment task. Moreover, as the architecture of the application is microservices, the configuration and deployment are remarkably complicated and took abundant time when accomplishing manually. Consequently, the manual testing duration was exceedingly long just because of the deployment process. However, there are solutions to overcome all these challenges.

3.3 Solutions

There were many challenges in development cycle described previously in the project. There are solutions to resolve all of these disputes. The first solution could be implementing automated scripts to get rid of the problems. The first obstacle could be overcome by executing a script that formats the whole code-base, run the test suites and generate reports into files. This script can be run manually several times during development phase to maintain the quality of the code-base. This script could also be triggered to run before developers commit their changes. Another script could be used to automate the deployment procedure with detailed information about infrastructure configurations. This script would be used in deployment process to deploy the product to proper destination. These scripts somehow automated several processes in the development cycle and resolved the challenges quite nicely.

There is also another approach to deal with the disputes presented earlier, Continuous Delivery (CD) adoption. In this case, a CD pipeline could be built to not only increase the productivity of the development cycle but also resolve the manual tasks issues. The pipeline regularly contains multiple stages and problems are resolved gradually in different stages. Firstly, the code formatting and testing matters could be quickly figured out by executing proper commands and committing changes in build stage of CD pipeline. The code coverage report could be uploaded to a separate centralised place for developers to monitor the quality of the code-base easily. Additionally, the deployment procedure could be automatically handled by CD pipeline easing the manual testing and release stage by automated scripts. This solution helps to test the code-base effortlessly and to deploy the application to the desired environment frequently.

Clearly, both solutions are able to resolve the issues accordingly. Nevertheless, the first solution can lead to other obstacles which might take a longer time to be resolved. For instance, there might be conflicts in code coverage report files between developers or the deployment is still attached to a specific developer. On the other side, the second solution completely wipes out manual tasks and keeps the reports out of the code-base in a centralised place. Developers are not required to execute any script before committing their changes. Moreover, the failures in release stage are disappeared by the appearance of the release button in the CD pipeline which is extremely beneficial.

3.4 Implementation

As discussed previously, the solution to implement a CD pipeline is better. The implementation of the pipeline is explained in the following sections. First, an overview of the software system is described to have a better understanding of the application. Then, tools and services required to build the pipeline are mentioned with reasons why should they were used. Last but not least is the actual process of building the pipeline with detailed information on different stages.

3.4.1 Application architecture

Understanding the application system and its behaviours play an important role in designing and implementing a suitable CD pipeline for the use case. The application, in this case, is a learning platform for internal usages of about 20,000 employees. The architecture of the application is microservices due to considerable integrations with existing services. Figure 10 illustrates an overview of the desired application system at MVP release.

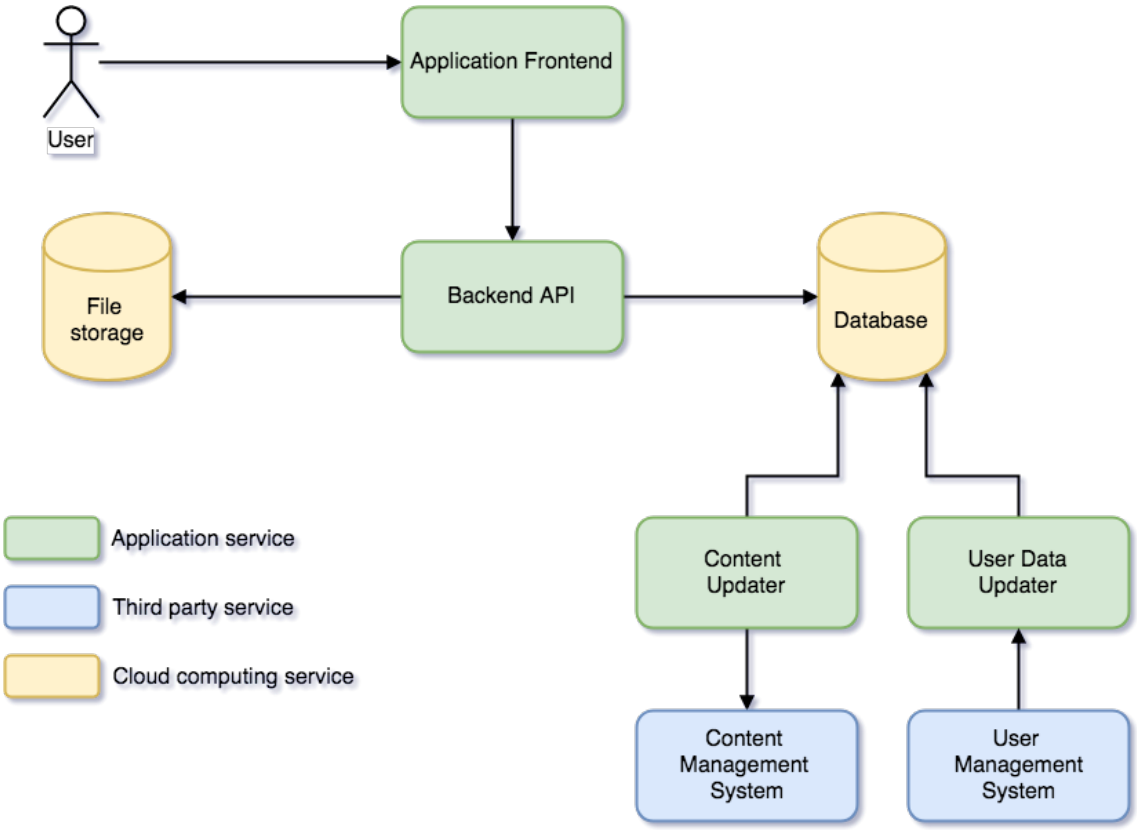

Figure 10: Software architecture at MVP release.

As can be seen from Figure 10 on the preceding page, the architecture of the application and its behaviours has been explained concisely. The application contains four services (frontend, backend API, content updater and user data updater). User data updater service brings up to date data related to users to application database whenever being requested by user management system. Meanwhile, content updater service frequently pulls the latest content from the content management system to update existing relevant data in the database. Different from the two services above, backend API and frontend service do not communicate with third-party services. This service supplies resources in the database and file storage for the usage of frontend service. An elegant user interface is offered by frontend service to improve the user experience of the application. All these services work closely together to deliver a proper application for customers.

3.4.2 Services and tools

Due to limited time for implementation, the pipeline was constructed with the help of several available platforms in the market. These supporting services and tools are explicitly explained in the following sections.

Google Cloud Platform (GCP)

GCP comprises a chain of cloud computing services provided by Google. GCP help customers to be free from managing physical infrastructure, provisioning servers and configuring networks. There are many essential benefits supplied by Google such as future-proof infrastructure, compelling data and analytics, scalable security and customer-friendly pricing. Google provides a wide range of cloud computing products, from IaaS, PaaS, Network as a Service to Serverless computing. Recently, Google has been focusing on developing container services such as Kubernetes Engine and Container Registry with attractive pricing plans and powerful features. [27.]

Google Kubernetes Engine (GKE)

GKE is a controlled environment for containerised software deployment. Google's latest innovations in development productivity, resource efficiency and open source flexibility are brought into this product. GKE activates fast development by providing abilities to

deploy, update and manage applications and services efficiently and smartly. In addition, GKE auto-scaling feature enables applications to handle sudden increment users' demand on services. In GKE, Google also runs an upstream Kubernetes to ease developers or operations team effort in system components customisation including monitoring, logging and CD. GKE is a smart way for Docker containers to be deployed, managed and scaled easily on GCP powered by Kubernetes. [28.]

Google Container Registry (GCR)

GCR is a fast and private Docker image storage on GCP. GCR is a private Docker registry that compatible with common CD systems such as Jenkins, Travis and CircleCI. Private images are stored in Google Cloud Storage and cached in Google's data centres, this help GKE to speedily retrieve and deploy intended containers. In addition, the privacy and security are always maintained as images are hosted under cloud platform projects with identity and access management. Only project members can access private images. With GCR, worries about the confidentiality or performance issues of container registry are all clarified. [6.]

CircleCI

CircleCI is a modern continuous delivery platform. The name includes CI instead of CD as initially it was developed to focus only on Continuous Integration (CI) (aims at building and testing) but later on expanded to be a CD platform. CircleCI is powerful, flexible and controllable. This tool aids the pipeline automation from source code commit phase till deployment stage. In addition, CircleCI offers notification integration which makes the whole team to be updated about build status and act rapidly based on the report. [29.] Recently, CircleCI release version 2.0 which provides several advantageous features such as computing capability customisations for different builds and job orchestration [30].

Codecov

Codecov is a code coverage platform which focuses on code quality and unification. Codecov distributes reports straight into development workflow to advance the code quality in some way, particularly in pull requests where new features are implemented and waiting for the release. Codecov offers several beneficial features such as pull request

comments and combining reports. By experience, the code review duration becomes shorter with the usage of Codecov. Code coverage is lifted up by Codecov. [31.]

3.4.3 Pipeline implementation

According to the application architecture and CD requirements, the pipeline was designed to contain four stages including code commit, build, acceptance testing, and release. Figure 11 indicates the implementation of the pipeline in combination with previously listed services.

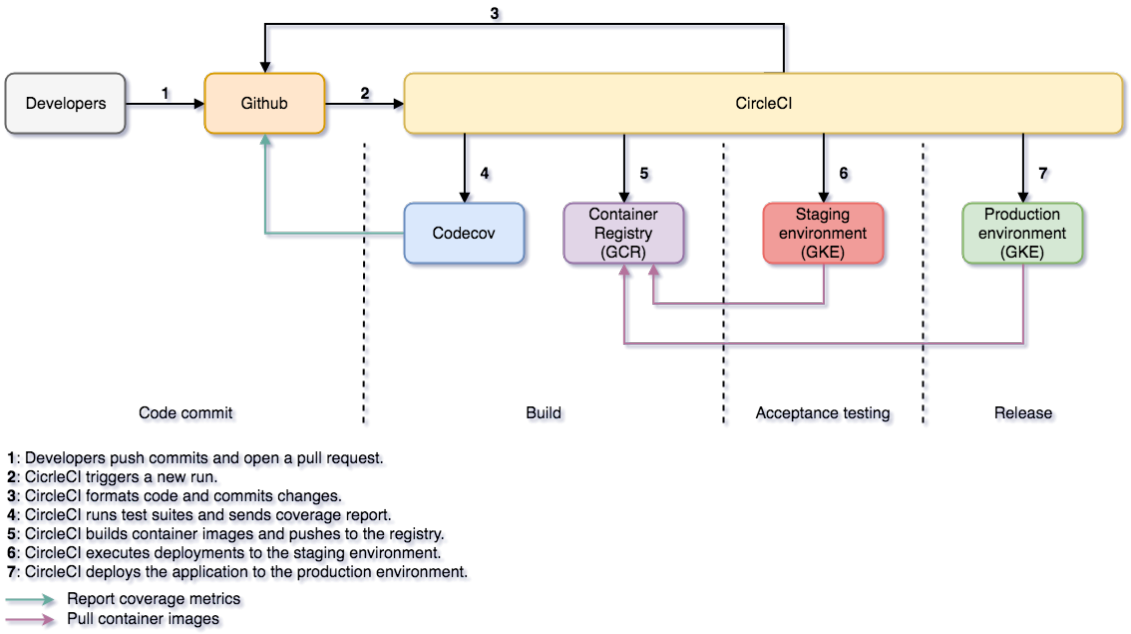

Figure 11: CD pipeline implementation.

As illustrated in Figure 11, the starting point of the pipeline is code commit stage in which developers push their changes to a remote repository in Github and open a pull request. The pull request is a way for developers to inform others that the feature implementation or defect fixes are completed. At this point, the beginning of the build stage is automatically triggered with the help of CircleCI. This stage starts with checking out the source code and format them to follow team's coding conventions strictly. In case there is any change in formatting steps, the build fails. Moreover, those changes are committed back to the repository before starting a new run. In the next run, the code does not cause the

build to fail again. The process continues with compiling source code, running unit test and integration test suits, building Docker images and pushing the built images to remote registry hosted by GCR. During these processes, the code coverage report is generated and uploaded to Codecov. If any step contains failures, the process stops and informs developers about them. Otherwise, the pipeline is automatically moved to the next stage - acceptance testing. In this stage, CircleCI executes a script to set up a staging environment in a cluster hosted at GKE and deploying the application there. Then, developers are informed with the instruction to access this environment to test the product. After the testing was fulfilled and the implementation got approved, the code changes are merged into the master branch and start a new pipeline run to prepare for the release. The acceptance testing is accomplished one more time in the run of the master branch. At this point, by clicking a release button, the process move to release stage and the application is deployed to the production environment. In case of any failures in this stage, the pipeline executes a script to roll back to the previous version. The production software is guaranteed for its availability and reliability.

After got the approval from the development team, the pipeline implementation was conducted. The execution consists of four main steps which are infrastructure acquisition, script development, code coverage software integration, and CD platform configuration. These steps are explained more detail in the following sections.

Infrastructure acquisition

This step contains several phases starting from cloud project creation to registry configuration and cluster management. GCP was decided to be the cloud service for hosting required infrastructure and services in this project. Fortunately, GCP provides a CLI namely **gcloud** to make services management extremely effortless.

Firstly, a Google Cloud project was needed to start using GCP services. The project creation was seamlessly achieved by executing the following command: `gcloud projects create cd-pipeline-poc`. A cloud project with the ID of **cd-pipeline-poc** was created. From this point, cloud services are deployed inside this project in GCP. A web user interface is also provided by GCP to interact with project and services inside it. After project creation was succeeded, the project was available to access. Figure 12 on the next page

illustrates the dashboard of the project after creation in GCP.

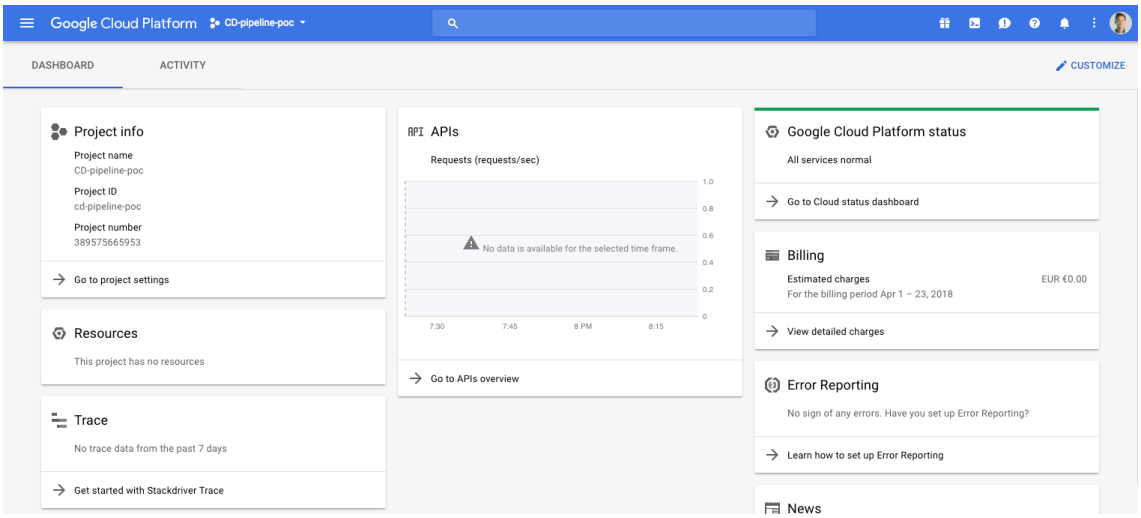

Figure 12: Cloud project dashboard.

Secondly, GCR needed to be configured to be available for storing Docker images. By clicking into Container Registry in the sidebar menu, the GCR dashboard is showed up (see Figure 13).

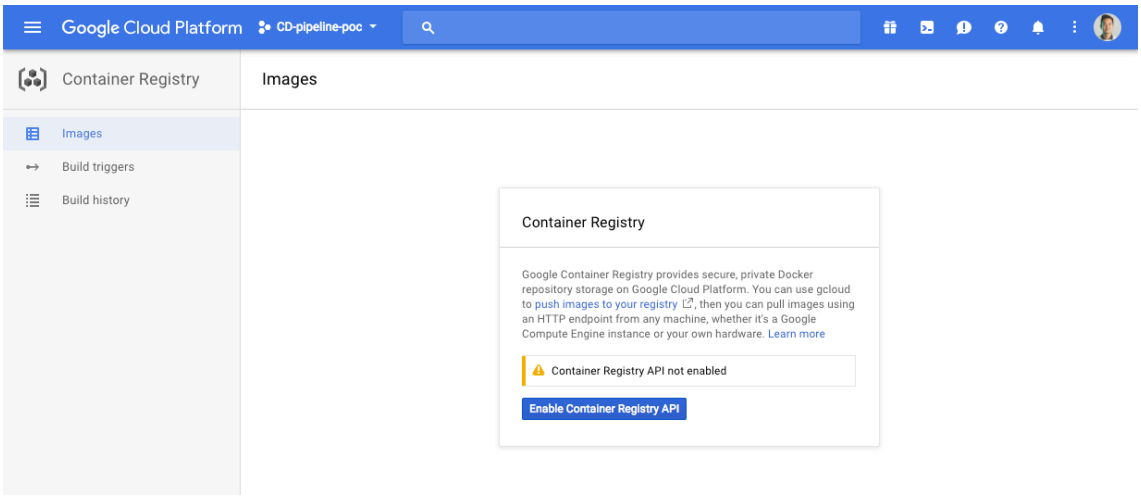

Figure 13: Container Registry dashboard.

As can be seen from Figure 13, Container Registry raised up a warning that Container Registry API was not enabled. By activating this API, pipeline runner could connect and push Docker images to the registry by using CLI. After completed enabling the Container Registry API, the registry came to ready-to-use state without any warning.

Then, a Kubernetes cluster was required to deploy application there. The process of creating a cluster was completed seamlessly by using **gcloud** CLI. The cluster creation is explained in detail in Listing 1 in Appendix 1. The cluster creation was taking several

minutes for GKE to acquire needed infrastructure, install software in it. After cluster was successfully created, the cluster along with its status was displayed in the GKE dashboard (see Figure 14).

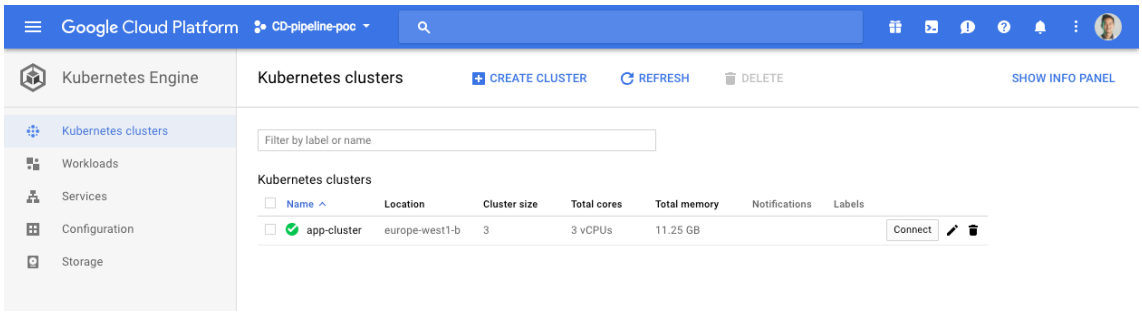

Figure 14: Kubernetes Engine dashboard.

As can be seen from Figure 14, the cluster was ready to be used. However, a separate account which contains only the right of handling cluster was needed for the pipeline runners as the deployment process was planned to be automatic. In GCP, these accounts are called service accounts. In this project, there was a need for 2 separate service accounts. The first account was used to push images to GKE. The other one was used for the deployment process. The execution of this step is described in detail in Listing 2 in Appendix 1. In general, the infrastructure acquisition tasks were done resulted in a running cluster, a private container registry and two service accounts.

Script development

Several scripts are needed to reduce the pipeline configuration time and make it work efficiently. The scripts usually relate to the tasks in pipeline stages. Fortunately, the code-base at that time had a Makefile which contains commands for formatting, linting, testing, compiling and building source code. The remaining work was to implement a deployment script which required to modify the build commands and create configuration template files. The build execution was changed to attach proper tags to Docker images and push them to GCR. The build script modifications are explained in detail in Listing 3 in Appendix 2.

Additionally, the deployment configuration templates and scripts were demanded to be implemented. According to the application architecture, there were four services in the system. Consequently, there are four configuration template files are implemented. In general, these files share the same format but different parameters in the number of repli-

cation for services. Besides, the deployment script is changed for the compatibilities with CircleCI. The detailed configurations of the deployment script is presented in Listing 4 in Appendix 2.

Code coverage software integration

Codecov was planned to use in this project as a centralised coverage report storage. The integration was achieved by placing a Codecov configuration file into the root of the code-base, setting up a needed environment variable, and include a command in the test script. The Codecov configuration files can be validated by an open API provided by Codecov. After the test suits were executed, to report the code coverage statistics, the following command needs to be run: `bash <(curl -s https://codecov.io/bash)`. Nevertheless, the reports needed to be uploaded to a particular address. A token received from Codecov interface specifies this address. The token has to be stored in the pipeline runner environment as an environment variable called **CODECOV_TOKEN** to execute the upload process accurately. Generally, the code coverage software integration was accomplished at this point.

CD platform configuration

At this point, the scripts were ready to be executed in pipeline stages. The remaining work was to config the pipeline flow with the help of CircleCI platform. Fortunately, CircleCI offers a simple way to define CD pipeline flow by placing a configuration file (**config.yml**) under the directory **.circleci** in the project root. In general, the configuration file constructes the pipeline into 4 phases (*build*, *acceptance-testing*, *approved*, and *deploy*). The detail implementation of the configuration file is explained in detail in Listing 5 in Appendix 3.

After the configuration file was created, there were several environment variables need to be set up for pipeline runner to work. By navigating to CircleCI dashboard, these environment variables were set up and ready for the first build. Figure 15 on the following page illustrates the environment variables configuration in CircleCI.

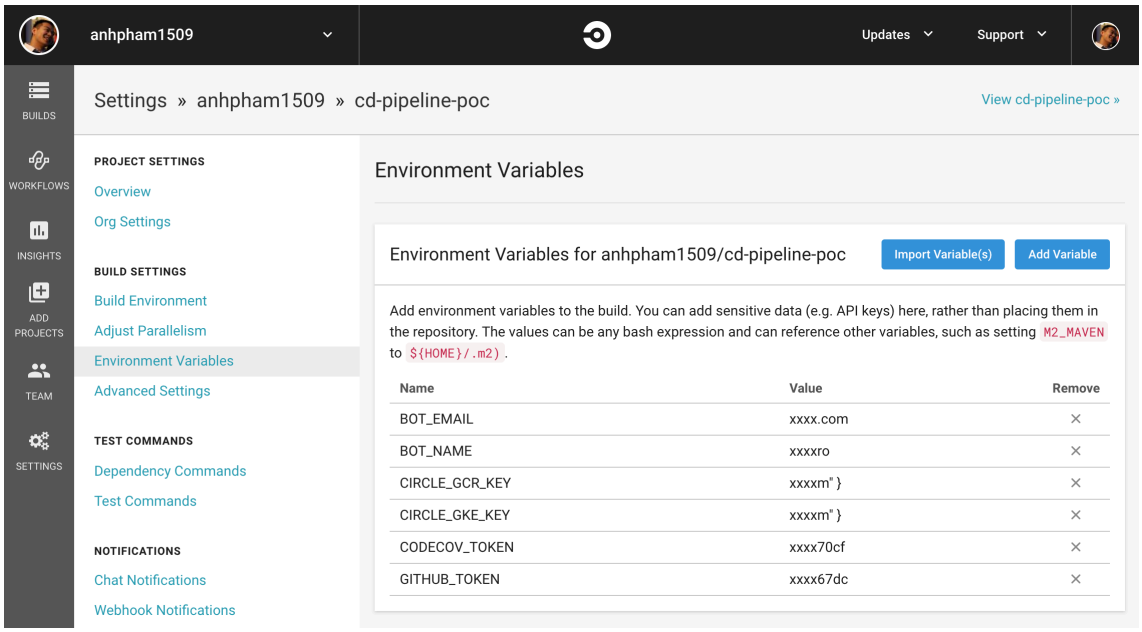

Figure 15: Environment variables configuration in CircleCI.

At this point, the pipeline environment was ready to start the first build. By adding Github project in CircleCI dashboard, the code-base was triggered to start the first run (see Figure 16).

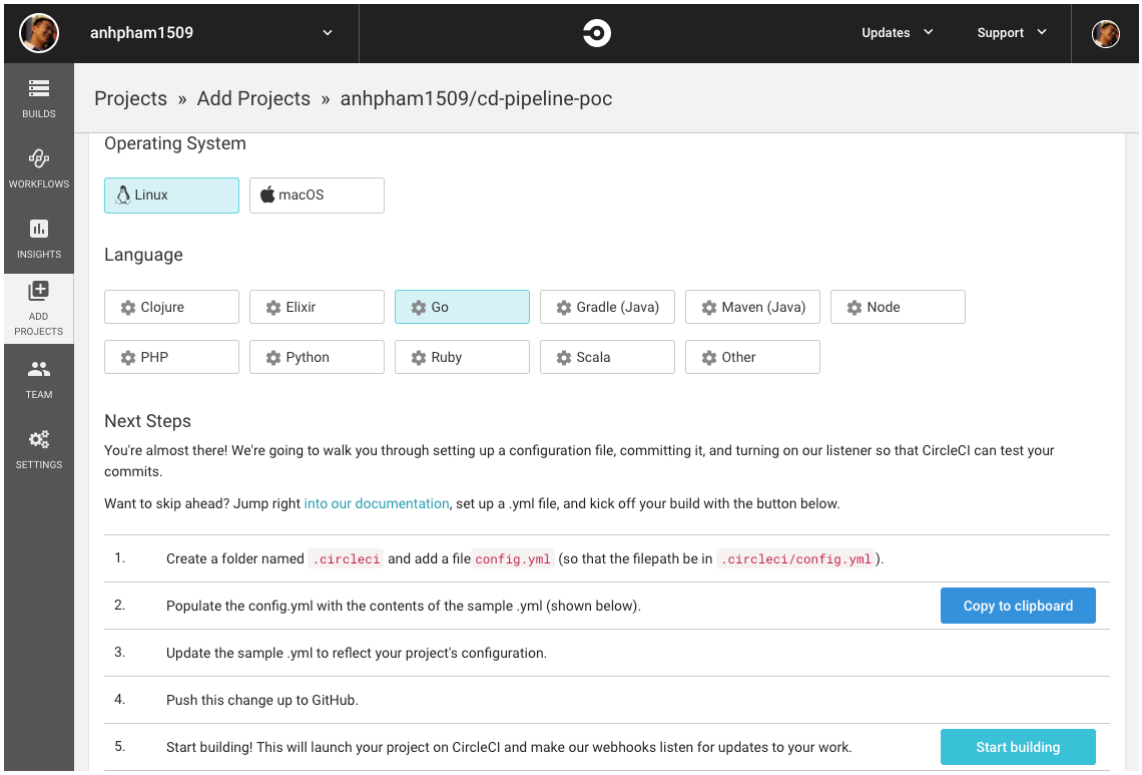

Figure 16: Add new project in CircleCI.

Finally, the CD pipeline was implemented with configurations not only in infrastructure but also software integration. Since then, the builds can be audited from CircleCI dashboard

by developers to figure out failures rapidly and correct them accordingly.

3.5 Results

By having a clear plan, a proof of concept of CD pipeline was implemented with success. The pipeline was built by the integration of services and platforms namely CircleCI, Codecov, GCR, and GKE. The pipeline workflow was set up based on designed stages without any obstacle. The outcomes of this project are illustrated in the following figures.

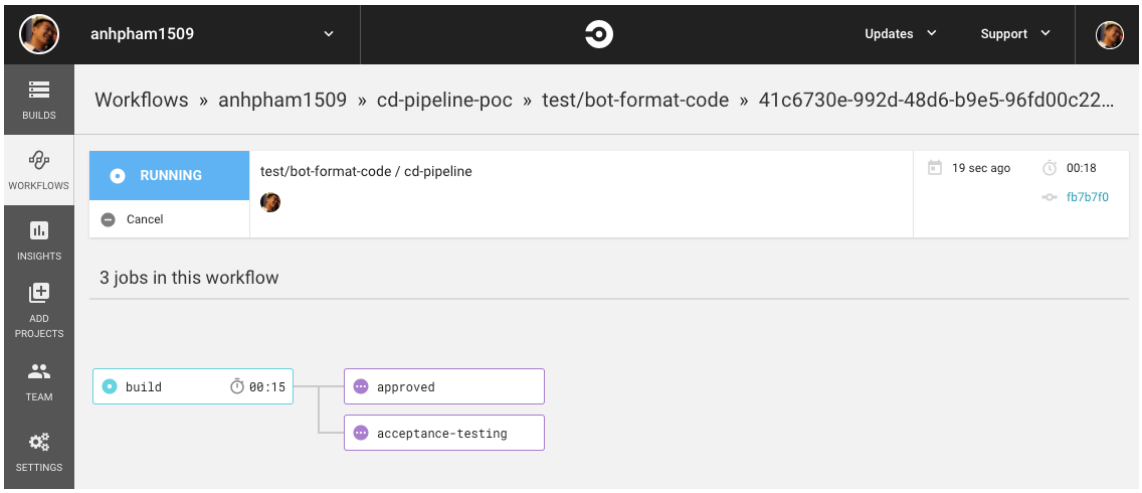

Figure 17: Workflow in a feature branch.

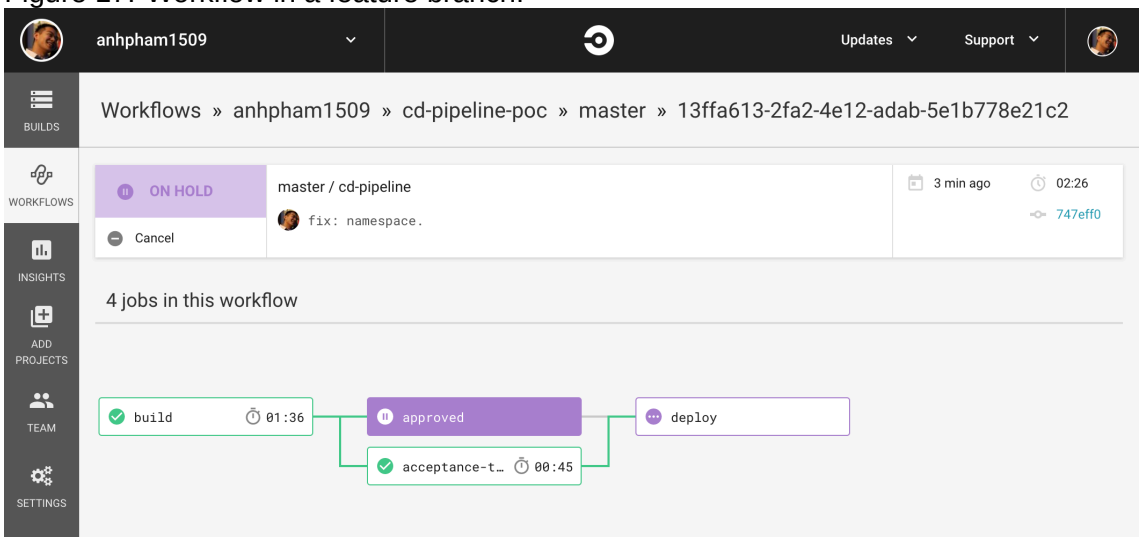

Figure 18: Workflow in the master branch.

As described in Figure 17 and 18, the pipeline workflow was different between the master branch and another branch. There was one more stage called deploy for the release stage when running in master branch and this stage was only reached by pressing **Approve** button. After a successful release, the workflow is shown as in Figure 19.

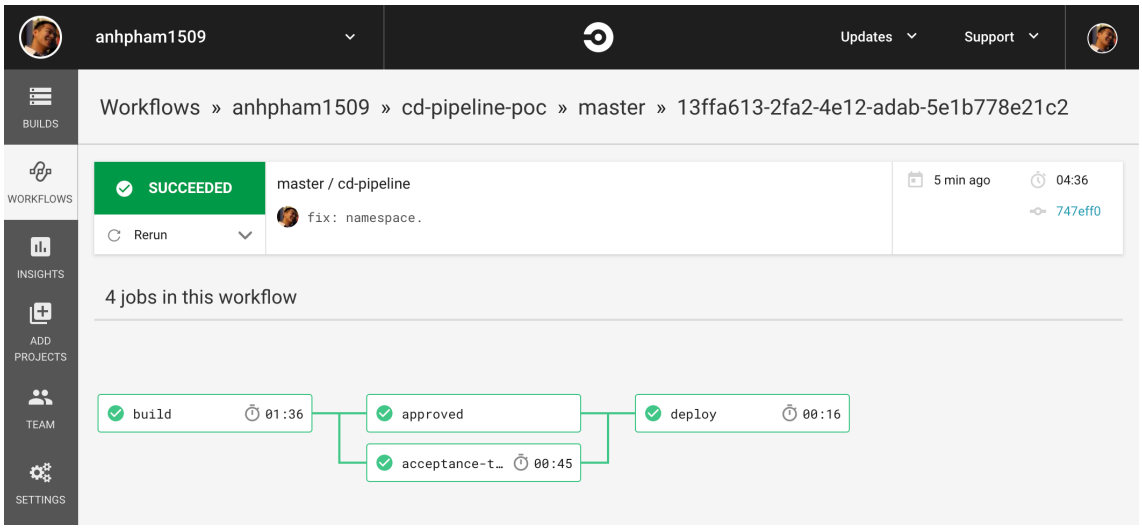

Figure 19: Successful release.

In addition, the source code was formatted automatically, and changes were committed to the target branch (see Figure 20).

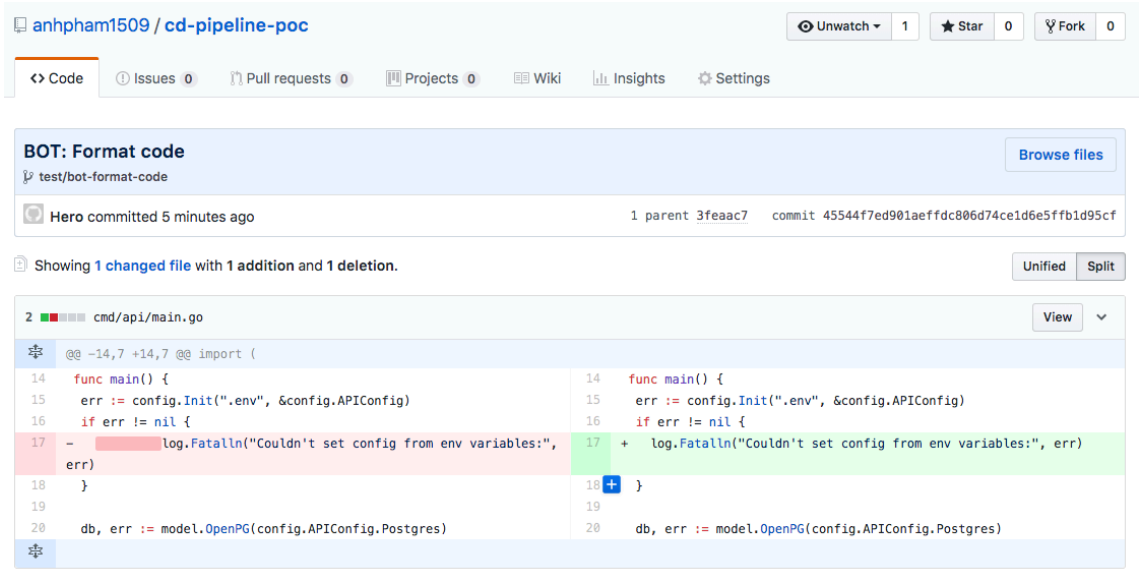

Figure 20: *Hero*'s commit.

As illustrated in Figure, one commit from *Hero* had been automatically pushed correctly. This commit fixed the indentation for one line of code. Moreover, the code coverage reports were uploaded to Codecov. Since then, developers focused on looking at Codecov dashboard instead of finding which lines of code inside code-base were not thoroughly tested. Figure 21 on the following page indicates a sample of a coverage report developers could get from Codecov dashboard.

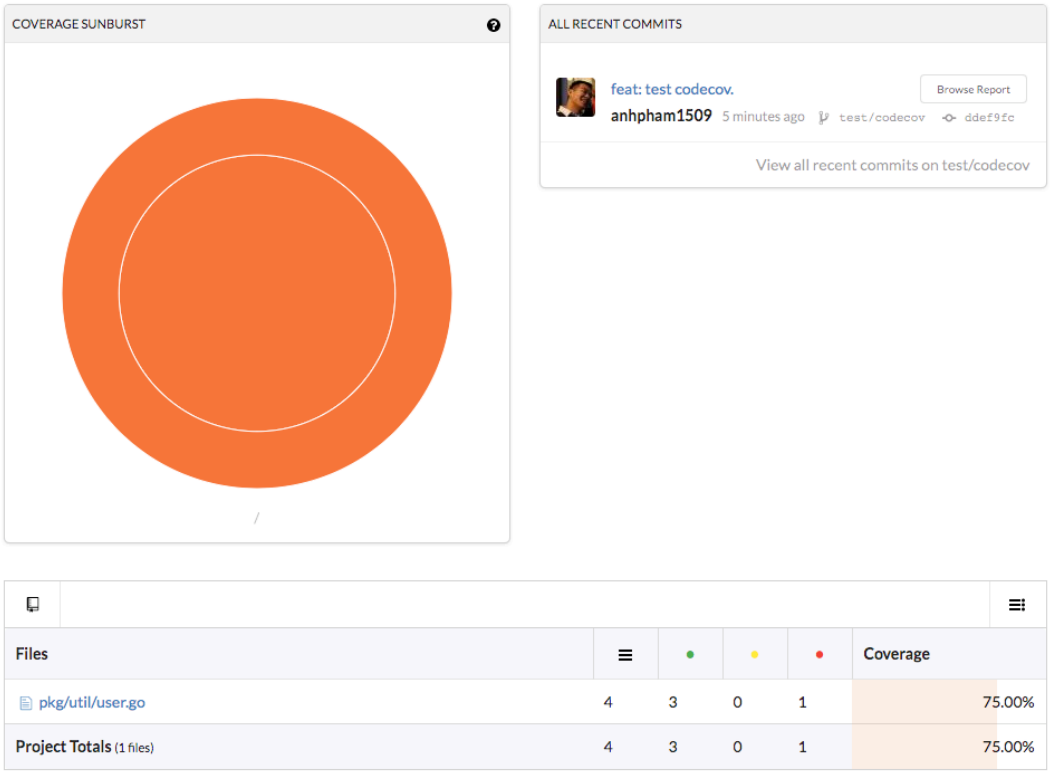

Figure 21: Sample Codecov report.

In general, the pipeline was set up successfully and proved the concept behind CD to make software delivery more rapid, efficient and reliable. The development team immediately adopted this CD pipeline and got advantageous results. According to CircleCI dashboard, the success rate of builds was extremely high with a minimised amount of execution time (see Figure 22).

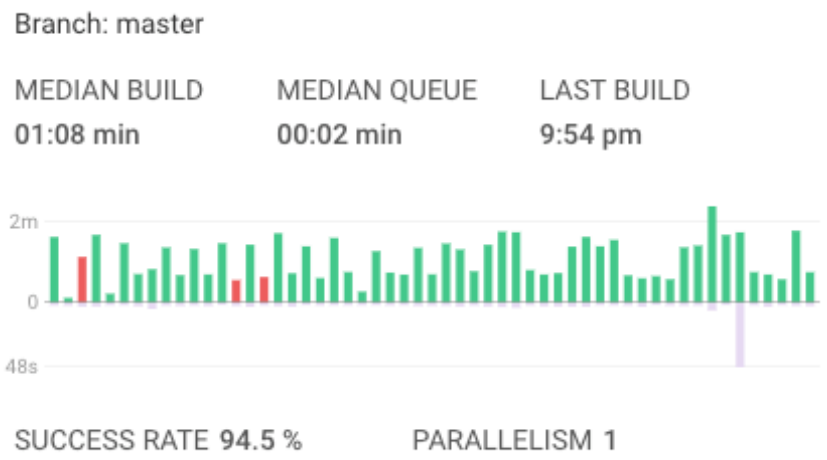

Figure 22: Builds statistics.

3.6 Future development

Though the implemented CD pipeline was a success which makes the release procedure of the application efficient and reliable, several improvements still needed to be developed. For instance, the pipeline is still missing a performance testing step which is necessary to identify and improve the capability of the application. Moreover, the availability of pipeline currently still relies on several dependency services. Implementing a separate and controllable system could be taken into consideration if resource and budget are allowed to do so. Alternatively, one system which gathers highlighted information from services can be implemented in the future. The system can avoid the situation that developers need to browse multiple services to get the final result. In general, the pipeline at that point was still at the first version, and there was a considerable demand to better the workflow to achieve the goal of releasing application regularly and keeping its performance and reliability.

4 Conclusion

In recent years, a considerable amount of applications are built. Consequently, the release cycle must be shortened much more to adapt to the needs of the market. Software development teams need the help of a CD pipeline to maintain the application reliability and release the product rapidly. Moreover, new technologies are published continuously with significant improvements not only in scalability but also in performance. However, deploying applications using modern software stack is tough without container technology. Recently, Docker has become the industry standard for container technology and commonly used by many products [8, 84]. To deliver and manage containerised applications, Kubernetes, which initially developed by Google, helps a lot in clustering management and is endorsed by many enterprises such as Microsoft, VMware, IBM and so on [14]. More than that, Google has been developing cloud services supporting containerised applications registry and deployment which makes application deployment and management easier [27]. Building a CD pipeline has become an essential and demanding component in every software engineering project.

In conclusion, the project initially described how to set up a pipeline to continuously deliver Docker containerised applications to Google Cloud Platform using CircleCI and Kubernetes. By setting up the pipeline, not only the software development time is reduced significantly, but also the customer satisfaction has been enhanced a lot. Product users are always served with desired products rapidly and reliably. However, implementing CD pipeline still contains many impediments. The obstacles might come from organisations, processes or techniques [24, 53-54]. Fortunately, there are methods to avoid these challenges. Firstly, enterprises should have a collaborative culture among teams to avoid miscommunication. Also, the process should be concise and flexible enough to promote features being released on time. Lastly, building a CD pipeline needs to integrate many different tools and technologies. Consequently, enclosed dependencies might also be an obstacle at a specific time in development lifetime. The project should follow broadly approved standards and build an intense vendor ecosystem. [24, 53-54.]

References

1 Stubbs J, Moreira W, Dooley R. Distributed Systems of Microservices Using Docker and Serfnode. 2015 7th International Workshop on Science Gateways. 2015 June;p. 34–39.

2 Soni M. End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery. 2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM). 2015 November;p. 85–89.

3 Virmani M. Understanding DevOps bridging the gap from continuous integration to continuous delivery. Fifth International Conference on the Innovative Computing Technology (INTECH 2015). 2015 May;p. 78–82.

4 Cito J, Schermann G, Wittern J, Leitner P, Zumberi S, Gall H. An Empirical Analysis of the Docker Container Ecosystem on GitHub. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). 2017 May;p. 323–333.

5 Kubernetes Authors. Kubernetes | Production-Grade Container Orchestration. [online]; 2018. Available from: https://kubernetes.io/ [cited 14 April 2018].

6 Google Inc. Container Registry - Private Docker Registry | Google Cloud. [online]; 2018. Available from: https://cloud.google.com/container-registry/ [cited 14 April 2018].

7 Newman S. Building Microservices. O'Reilly Media; 2015.

8 Bernstein D. Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Computing. 2014 September;1(3):81–84.

9 Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux J. 2014 March;2014(239).

10 Docker Inc. What is a Container. [online]; 2018. Available from: https://www.docker.com/what-container [cited 14 April 2018].

11 Matthias K, Kane S. Docker: Up and Running. O'Reilly Media; 2015.

12 Vohra D. Kubernetes Microservices with Docker. Apress; 2016.

13 Djemame K, Bosch R, Kavanagh R, Alvarez P, Ejarque J, Guitart J, et al. PaaS-IaaS Inter-Layer Adaptation in an Energy-Aware Cloud Environment. IEEE Transactions on Sustainable Computing. 2017 April;2(2):127–139.

14 Kubernetes Authors. What is Kubernetes? [online]; 2018. Available from: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ [cited 14 April 2018].

15 Baier J. Getting Started with Kubernetes. Packt Publishing; 2017.

16 Kubernetes Authors. Overview | Kubernetes. [online]; 2018. Available from: https://kubernetes.io/docs/tutorials/kubernetes-basics/ [cited 14 April 2018].

17 Rensin D. Kubernetes - Scheduling the Future at Cloud Scale. O'Reilly Media; 2015.

18 Kubernetes Authors. Deployments | Kubernetes. [online]; 2018. Available from: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ [cited 14 April 2018].

19 Kubernetes Authors. Understanding Kubernetes Objects | Kubernetes. [online]; 2018. Available from: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/ [cited 14 April 2018].

20 Kubernetes Authors. Kubernetes Object Management | Kubernetes. [online]; 2018. Available from: https://kubernetes.io/docs/concepts/overview/object-management-kubectl/overview/ [cited 14 April 2018].

21 Morris K. Infrastructure as Code: Managing Servers in the Cloud. O'Reilly Media; 2016.

22 Humble J, Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education; 2011.

23 Shahin M, Babar MA, Zhu L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access. 2017;5:3909–3943.

24 Chen L. Continuous Delivery: Huge Benefits, but Challenges Too. IEEE Software. 2015 March;32(2):50–54.

25 Itkonen J, Udd R, Lassenius C, Lehtonen T. Perceived Benefits of Adopting Continuous Delivery Practices. Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. 2016;p. 42:1–42:6.

26 Phillips A. The Continuous Delivery Pipeline — What it is and Why it's so Important in Developing Software. [online]; 2018. Available from: https://devops.com/continuous-delivery-pipeline/ [cited 20 April 2018].

27 Google Inc. Business Solutions for Enterprise | Google Cloud. [online]; 2018. Available from: https://cloud.google.com/why-google-cloud/ [cited 22 April 2018].

28 Google Inc. Google Kubernetes Engine | Google Cloud. [online]; 2018. Available from: https://cloud.google.com/kubernetes-engine/ [cited 22 April 2018].

29 Circle Internet Services Inc . CircleCI product and features. [online]; 2018. Available from: https://circleci.com/product/ [cited 22 April 2018].

30 Circle Internet Services Inc . Administrator’s Overview - CircleCI. [online]; 2018. Available from: https://circleci.com/docs/2.0/overview/ [cited 22 April 2018].

31 Codecov. About code coverage. [online]; 2018. Available from: https://docs.codecov.io/docs/about-code-coverage/ [cited 22 April 2018].

1 Infrastructure configuration

1.1 Cluster creation

A Kubernetes cluster is published in Google Kubernetes Engine (GKE) after the execution of the command in Listing 1.

```
gcloud container clusters create "app-cluster" \
  --project "cd-pipeline-poc" \
  --zone "europe-west1-b" \
  --no-enable-basic-auth \
  --cluster-version "1.9.6-gke.1" \
  --machine-type "n1-standard-1" \
  --image-type "COS" \
  --disk-size "20" \
  --scopes "gke-default" \
  --num-nodes "3" \
  --enable-autoscaling \
  --min-nodes "3" \
  --max-nodes "5" \
  --network "default" \
  --enable-cloud-logging \
  --enable-cloud-monitoring \
  --subnetwork "default" \
  --enable-legacy-authorization \
  --addons HorizontalPodAutoscaling,HttpLoadBalancing,KubernetesDashboard
      \
  --enable-autoupgrade \
  --enable-autorepair
```

Listing 1: GKE cluster creation command

As can be seen from Listing 1, a new cluster is created with ID of **app-cluster** inside **cd-pipeline-poc** project. The cluster uses Kubernetes version 1.9.6 and runs in **europe-west1-b** zone. In addition, the cluster contains node auto-scaling feature specifically minimum of 3 and maximum of 5 nodes. Several add-ons are also attached to the cluster such as load balancer, pod auto-scaling or Kubernetes dashboard. Moreover, this cluster is set up to upgrade and repair nodes and master automatically in case of failures in operation time.

1.2 Service account creation

Base on the need of CD pipeline, there are two service accounts need to be created. These accounts are created by executing the script in Listing 2.

```
1  PROJECT="cd-pipeline-poc"
2  PROJECT_DOMAIN="cd-pipeline-poc.iam.gserviceaccount.com"
3  GCR_PUSHER="circleci-gcr-pusher"
4  GCR_PUSHER_ACCOUNT="${GCR_PUSHER}@${PROJECT_DOMAIN}"
5  GKE_CLUSTER_MANAGER="circleci-gke-cluster-manager"
6  GKE_CLUSTER_MANAGER_ACCOUNT="${GKE_CLUSTER_MANAGER}@${PROJECT_DOMAIN}"
7
8  # Create service account to access GCR
9  gcloud iam service-accounts create ${GCR_PUSHER} \
10   --project=${PROJECT} \
11   --display-name="Service account for CircleCI to push Docker images to
         GCR" \
12
13 # Generate a secret key to authenticate
14 gcloud iam service-accounts keys create \
15   --project=${PROJECT} \
16   --iam-account=${GCR_PUSHER_ACCOUNT} \
17   --key-file-type="json" \
18   circleci-gcr-pusher-key.json
19
20 # Grant proper roles
21 gcloud projects add-iam-policy-binding ${PROJECT} \
22     --member=${GCR_PUSHER_ACCOUNT} \
23     --role "roles/storage.admin"
24
25 gcloud projects add-iam-policy-binding ${PROJECT} \
26     --member=${GCR_PUSHER_ACCOUNT} \
27     --role "roles/storage.objectCreator"
28
29 gcloud projects add-iam-policy-binding ${PROJECT} \
30     --member=${GCR_PUSHER_ACCOUNT} \
31     --role "roles/storage.objectViewer"
32
33 # Create service account to access GKE
34 gcloud iam service-accounts create ${GKE_CLUSTER_MANAGER} \
35   --project=${PROJECT} \
36   --display-name="Service account for CircleCI to manage cluster hosted in
         GKE"
37
38 # Generate a secret key to authenticate
39 gcloud iam service-accounts keys create \
40   --project=${PROJECT} \
41   --iam-account=${GKE_CLUSTER_MANAGER_ACCOUNT} \
42   --key-file-type="json" \
```

```
43    circleci-gke-cluster-manager-key.json
44
45  # Grant proper roles
46  gcloud projects add-iam-policy-binding ${PROJECT} \
47      --member=${GKE_CLUSTER_MANAGER_ACCOUNT} \
48      --role "roles/container.developer"
```

Listing 2: Service account creation script

As can be seen from Listing 2, two service accounts are created and granted proper roles. Specifically, service account **circleci-gcr-pusher** is created with three access roles to manage the registry. Besides, account **circleci-gcr-pusher** is created with only one access to handle the cluster management tasks. Two access keys are also generated in JSON format to authenticate and connect to cloud services securely from CircleCI later on.

2 Script development

2.1 Build script

The build execution in the code-base was changed to attach proper tags to Docker images and push them to GCR (see Listing 3).

```
PROJECT="cd-pipeline-poc"
DOCKER_REGISTRY="eu.gcr.io"
DOCKER_TAG=`git rev-parse HEAD`
KEY_FILENAME="gcr-service-key.json"

# Compile api service to binary
CGO_ENABLED=0 GOOS=linux go build -installsuffix cgo -o api cmd/api

# Compile frontend service to binary
CGO_ENABLED=0 GOOS=linux go build -installsuffix cgo -o sap cmd/frontend

# Compile cms service to binary
CGO_ENABLED=0 GOOS=linux go build -installsuffix cgo -o cms cmd/cms

# Compile sap service to binary
CGO_ENABLED=0 GOOS=linux go build -installsuffix cgo -o sap cmd/sap

# Build api Docker image
docker build . \
  --tag ${DOCKER_REGISTRY}/${PROJECT}/api:${DOCKER_TAG} \
  --file apps/api/Dockerfile

# Build frontend Docker image
docker build . \
  --tag ${DOCKER_REGISTRY}/${PROJECT}/frontend:${DOCKER_TAG} \
  --file apps/frontend/Dockerfile

# Build cms Docker image
docker build . \
  --tag ${DOCKER_REGISTRY}/${PROJECT}/cms:${DOCKER_TAG} \
  --file apps/cms/Dockerfile

# Build sap Docker image
docker build . \
  --tag ${DOCKER_REGISTRY}/${PROJECT}/sap:${DOCKER_TAG} \
  --file apps/sap/Dockerfile

# Get authentication key to file
```

```
39 echo ${CIRCLE_GCR_KEY} > ${HOME}/${KEY_FILENAME}
40
41 # GCR Authentication
42 docker login https://${DOCKER_REGISTRY} \
43   -u _json_key --password-stdin < ${HOME}/${KEY_FILENAME}
44
45 # Push api Docker image to GCR
46 docker push ${DOCKER_REGISTRY}/${PROJECT}/api:${DOCKER_TAG}
47
48 # Push frontend Docker image to GCR
49 docker push ${DOCKER_REGISTRY}/${PROJECT}/frontend:${DOCKER_TAG}
50
51 # Push cms Docker image to GCR
52 docker push ${DOCKER_REGISTRY}/${PROJECT}/cms:${DOCKER_TAG}
53
54 # Push sap Docker image to GCR
55 docker push ${DOCKER_REGISTRY}/${PROJECT}/sap:${DOCKER_TAG}
```

Listing 3: Updated build script

As can be seen from Listing 3, the build process is divided into four steps. Initially, the source-code of services are compiled into binary files. Then, four Docker images are built with appropriate tags matching the registry address and project ID. After that, a secured connection to GCR is established using service account key created previously. Lastly, these built images are pushed to the remote registry with proper tags.

2.2 Deployment script

The deployment script presented in Listing 4 is the updated version which is used for automatic deployment in CircleCI.

```
1  CLUSTER_NAME="app-cluster"
2  PROJECT="cd-pipeline-poc"
3  DOMAIN="cd-pipeline-poc.iam.gserviceaccount.com"
4  GKE_SERVICE_ACCOUNT="circleci-gke-cluster-manager@${DOMAIN}"
5  KEY_FILENAME="gke-service-key.json"
6
7
8  # Get authentication key to file
9  echo ${CIRCLE_GKE_KEY} > ${HOME}/${KEY_FILENAME}
10
11 # Authenticate to Google Cloud
12 gcloud auth activate-service-account ${GKE_CLUSTER_MANAGER_ACCOUNT} \
13   --key-file=${HOME}/${KEY_FILENAME}
14
```

```
15 # Get cluster credential
16 gcloud container clusters get-credentials ${CLUSTER_NAME} \
17   --project ${PROJECT} \
18   --zone europe-west1-b
19
20 # Get list of configuration files
21 FILES=$(find kubernetes -type f -name "*.yml" -maxdepth 1)
22
23 # Clean deployed configuration files
24 rm -rf kubernetes/dist
25
26 # Create directory for storing deployment configurations
27 mkdir -p kubernetes/dist
28
29 for file in ${FILES}
30 do
31   fileName=$(echo ${file} | sed 's/kubernetes\///')
32   # Inject environment variable into kubernetes configuration files
33   envsubst < ${file} > kubernetes/dist/${fileName}
34 done
35
36 REPLACE_SPLASH="sed 's/\//-/g'"
37 REPLACE_UNDERSCORE="sed 's/_/-/g'"
38 REPLACE_COMMA="sed 's/\./-/g'"
39 TO_LOWERCASE="awk '{print tolower($0)}'"
40
41 if [[ ${CIRCLE_BRANCH} != "master" ]] then
42   # Generate unified namespace based on branch name
43   UNIFIED_BRANCH=$(echo ${CIRCLE_BRANCH} | \
44     ${REPLACE_SPLASH} | \
45     ${REPLACE_UNDERSCORE} | \
46     ${REPLACE_COMMA} | \
47     ${TO_LOWERCASE})
48
49   NAMESPACE=${UNIFIED_BRANCH}-${CIRCLE_PR_NUMBER}
50
51   # Create a namespace if not exist
52   if [[ $(kubectl get namespace ${NAMESPACE}) != 0 ]]; then
53     kubectl create namespace ${NAMESPACE}
54   fi
55 else
56   # Using production namespace if the current build run on master
57   NAMESPACE="production"
58 fi
59
60 # Deploy new version
61 kubectl apply \
62   --namespace ${NAMESPACE} \
63   -f kubernetes/
64
```

```
65 # Clean deployment configuration files
66 rm -rf kubernetes/dist
```

Listing 4: Deployment script

As described in Listing 4, the deployment process contains several steps. Initially, the authentication to GCP is established to get the credentials for cluster interactions. After that, configuration files are prepared for the deployment based on the created templates. Then, a Kubernetes namespace is created if it does not exist and the deployment to GKE happens in this step. The last step is to clean all generated configuration files.

3 CircleCI configuration

Listing 5 describes the CircleCI configuration file used in this project.

```
1  version: 2
2  machine:
3    timezone: Europe/Helsinki
4  jobs:
5    build:
6      docker:
7        - image: circleci/golang:1.10.0
8      working_directory: /go/src/github.com/anhpham1509/cd-pipeline-poc
9      steps:
10       - checkout
11       - run:
12           name: Setup bot user for git
13           command: |
14             git remote add pushback
                   https://${GITHUB_TOKEN}@github.com/anhpham1509/cd-pipeline-poc.git
15             git config --global user.name ${BOT_NAME}
16             git config --global user.email ${BOT_EMAIL}
17       - run:
18           name: Install dep
19           command: |
20             DEP_VERSION=0.4.1
21             curl -L -s
                   https://github.com/golang/dep/releases/download/v${DEP_VERSION}/dep-
                   -o ${GOPATH}/bin/dep
22             chmod +x $GOPATH/bin/dep
23       - restore_cache:
24           keys:
25             - deps-{{ arch }}-{{ checksum "Gopkg.lock" }}
26       - run:
27           name: make install
28           command: |
29             make install
30             if [[ ${CIRCLE_BRANCH} != "master" ]]; then
31               if [[ $(git status --porcelain) ]]; then
32                 git add --all
33                 git commit -m "BOT: Make install"
34               fi
35             fi
36       - save_cache:
37           key: deps-{{ arch }}-{{ checksum "Gopkg.lock" }}
38           paths:
39             - vendor
```

```
40        - run: make setup
41        - run:
42            name: make format
43            command: |
44              if [[ ${CIRCLE_BRANCH} != "master" ]]; then
45                make format
46                if [[ $(git status --porcelain) ]]; then
47                  git add --all
48                  git commit -m "BOT: Format code"
49                fi
50              fi
51        - run: make lint
52        - run: make test
53        - run:
54            name: Upload coverage to codecov.io
55            command: bash <(curl -s https://codecov.io/bash)
56        - run:
57            name: Push changes
58            command: |
59              if [[ $CIRCLE_BRANCH != "master" ]]; then
60                if [[ $(git log origin/${CIRCLE_BRANCH}..${CIRCLE_BRANCH})
                      ]]; then
61                  echo "Pushing autobot fixes"
62                  git push --set-upstream pushback $CIRCLE_BRANCH
63                  exit 1
64                fi
65              fi
66        - setup_remote_docker:
67            docker_layer_caching: true
68        - run: make build
69
70    acceptance-testing:
71      docker:
72        - image: google/cloud-sdk:198.0.0-alpine
73      working_directory: /cd-pipeline-poc
74      steps:
75        - checkout
76        - run:
77            name: Install kubectl
78            command: |
79              if [[ $(kubectl version) != 0 ]]; then
80                gcloud components install kubectl
81              fi
82        - run:
83            name: Install envsubst
84            command: |
85              if [[ $(envsubst -V) != 0 ]]; then
86                set -x
87                apk add --update libintl
88                apk add --virtual build_deps gettext
```

```
89                cp /usr/bin/envsubst /usr/local/bin/envsubst
90                apk del build_deps
91              fi
92        - run:
93            name: Staging deployment
94            command: |
95              UNIFIED_NAMESPACE=`echo ${CIRCLE_BRANCH} | sed 's/\//-/g' |
                    sed 's/_/-/g' | sed 's/\./-/g' | awk '{print
                    tolower($0)}'`
96              scripts/deploy.sh staging-${UNIFIED_NAMESPACE}
97
98    deploy:
99      docker:
100       - image: google/cloud-sdk:198.0.0-alpine
101     working_directory: /cd-pipeline-poc
102     steps:
103       - checkout
104       - run:
105           name: Install kubectl
106           command: |
107             if [[ $(kubectl version) != 0 ]]; then
108               gcloud components install kubectl
109             fi
110       - run:
111           name: Install envsubst
112           command: |
113             if [[ $(envsubst -V) != 0 ]]; then
114               set -x
115               apk add --update libintl
116               apk add --virtual build_deps gettext
117               cp /usr/bin/envsubst /usr/local/bin/envsubst
118               apk del build_deps
119             fi
120       - run:
121           name: Production deployment
122           command: scripts/deploy.sh "production"
123
124 workflows:
125   version: 2
126   cd-pipeline:
127     jobs:
128       - build
129       - acceptance-testing:
130           requires:
131             - build
132       - approved:
133           type: approval
134           requires:
135             - build
136       - deploy:
```

```
137             requires:
138               - acceptance-testing
139               - approved
140             filters:
141               branches:
142                 only: master
```

Listing 5: CircleCI configuration file

As can be seen from Listing 5, the pipeline workflow in CircleCI contains 4 jobs which are *build*, *acceptance-testing*, *approved*, and *deploy*. In *build* job, the sequence starts with checking out source code, setting up an environment, installing dependencies and tools. After that, formatting and linting source code are executed. Then, unit test suites and integration tests are run, and the result is reported to Codecov. In case of any change during previous steps, those changes are committed and pushed back to the repository in Github. The last step of *build* job is to build and push Docker images to GCR. Afterwards, the application deployment to staging environment happens in *acceptance-testing* job. The *approved* job was there as an approval condition before production deployment. The process is held there until got an approval. Consequently, the pipeline runner is automatically triggered to move to *deploy* job. In this job, the application is deployed into the production environment, and all new changes are available to customers.