Expertise
and insight
for the future

Toan Thanh

Modern Production-Grade Cloud Native Pipeline with Kubernetes

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

5 May 2020

Author Title Number of Pages Date	Toan Thanh Modern Production-Grade Cloud Native Pipeline with Kubernetes 50 pages 5 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructors	Vesa Ollikainen, Senior Lecturer

During the advancement of technology in this digital era, software engineering is revolutionizing at a monumental growth, resulting in an enormous number of projects built and released every day. Deploying an application to a cloud platform is extremely tough without the help of a container technology solution. Docker and Kubernetes are changing how applications are built and deployed in the cloud.

The ultimate technical objective of this project was to build a production-grade pipeline using Docker, Kubernetes, Travis CI to deploy a fully functional application written in Javascript to Kubernetes cluster on Google Cloud Platform. Moreover, implementation highlights the need of Kubernetes in microservices' architecture and pipeline's importance in software engineering projects. Finally, the business objective of the project was to build a minimum viable product version regarding a point of sale system for a restaurant.

By investigating and building a Kubernetes pipeline piece of software using microservice architecture, the goal was successfully achieved. Moreover, the advantages and usages of Docker, Kubernetes, Travis CI and Google Cloud Platform services in modern software implementation were investigated and discussed during the scope of the project.

Keywords	Kubernetes, Container, Docker, Microservice, Continuous Integration, Continuous Delivery, Continuous Deployment, CI / CD Pipeline, Google Cloud Platform

Contents

List of Abbreviations

VM	Virtual Machine, an emulation of a computer system. It is based on computer architectures and provide functionality of a physical computer.
DevOps	Development Operations, a set of practices combining software development and information technology, used to shorten development life cycle.
CD	Continuous Delivery, software engineering approach used to produce application in short cycles.
OS	Operating System, low-level system software that supports a computer’s basic functions.
API	Application Programming Interface, a set of functions and procedures allowing the creation of applications which access the features or data of an operating system, application or other service.
GCP	Google Cloud Platform, collection of cloud computing services provided by Google, running on the same infrastructure of Google’s products.

1 Introduction

In the past, the most common architecture for most software applications were commonly monoliths, combining all different components that are all tightly coupled together into a single program from a single platform. Monolith applications experience delaying release cycle as well as infrequent update. (Newman, 2018.) Weeks or months are usually measured in cycle time in many companies, resulting in millions of dollars in costs for large companies as no revenue is delivered until the software reaches its end users (Humble and Farley, 2010). The lack of teamwork between developers and operators results in delaying release cycle. Fortunately, the evolution of containers enables us to fill the gap between those teams. Software is containerized by Docker and deployed to Kubernetes cluster on Google Cloud Platform. Kubernetes provides a production-grade container orchestration for automating deployment, scaling and management of containerized applications. Also, the role of Kubernetes in microservice architecture is crucial. (Kubernetes authors, 2019.)

1.1 Project technical objectives

The ultimate technical objective of this project is to build a production-grade pipeline for an application and deploy it to Kubernetes cluster in Google Cloud Platform hosting service. Throughout the technical implementation, the importance of Kubernetes in microservice architecture and pipeline importance in software engineering projects is highlighted. By applying the best practice of microservices with container technology, continuous delivery pipeline, along with Kubernetes as an orchestrator, the goal is to build an extremely available and highly scalable software.

1.2 Case study summary & business objective

The business objective of this thesis is to open the path for the digitalization of a traditional restaurant’s operations and services with the help of technology. The restaurant’s ambition was to expand their business towards digital dimension, including internal operation to customer acquisition and loyalty program. The main focus at the first stage in digitalization transformation plan is to build a Minimum Viable Product point of sale software, aiming to solve part of challenges in the day-to-day operation to eliminate human error between the order taking and food delivery time. There are quite a few existing solutions on the market such as Touch Bistro, Toast, etc. However, the new application is needed as the restaurant craves for the full control on new features and this point of sale system will be integrated into a bigger customized system, ranging from inventory management to customer loyalty later on.

1.3 Structure of the thesis

The following table describes the structure of the thesis and the main goals or contents in each chapter.

Chapter	Name	Main goals / contents
Chapter 1	Introduction	• Introducing the background of the thesis subject • Describing the thesis’s technical and business objectives, along with the case study summary and the structure of the thesis
Chapter 2	Theoretical background	• Providing the theoretical background for the project • Highlights the needs of Docker, Kubernetes and Continuous Delivery pipeline in software engineering applications with microservice architecture
Chapter 3	Case study	• Presenting case study used in this project, including challenges, objectives, solution, implementation and results
Chapter 4	Conclusion	• Summarizing thesis’s objectives and main results • Evaluating the project and personal learnings
	References	• Listing all the sources used in this thesis
	Appendices	• Including information, secondary code snippets supporting the main content.

2 Theoretical background

2.1 Monolithic architecture

During the revolution of technology in this digital era, software engineering is revolutionizing at a monumental growth, resulting in an enormous number of projects built and released every day, along with their extensive complexities. User-facing interface, along with access and authorization management, asynchronous task processor, databases, analytics, task queue and logging system are some typical components which modern web-based system should include nowadays. (Stubbs, Moreira and Dooley, 2015, p.35.) At the same time, we have been finding better ways to build anti-fragile systems by learning from the existing technology as well as adopting and observing new waves of revolution (Newman, 2018).

Until recently, the common architecture of software was a big monolith, combining all different components that are all tightly coupled together into a single program from a single platform (Luksa, 2018). Monolithic architecture brings enormous benefits to development team in case the application is relatively small as it enables straightforward development, testing, deployment and troubleshooting process. Furthermore, scaling applications is trivial due to the flexibility in duplicating application instances running behind a load balancer. (Newman, 2018.) The figure 1 below represents an overview of a monolithic architecture. As can be seen in the figure, all functionalities broke into multiple modules are packaged together in the single process system. On the one hand, single-process monolith not only results in a much simpler development workflow, ranging from developing, monitoring, troubleshooting to end-to-end testing, but also simplifies reusable code within the monolith itself. On the other hand, monolithic architecture tends to slow down development cycle period as more and more people works in the same piece of code. As all functionalities are packaged together, confusion between different teams on code ownership arises as concrete boundaries in the system are not drawn. Furthermore, with monolithic architecture, development teams are not able to flexibly adopt new technology such as new programming languages, database types or frameworks. (Newman, 2019.)

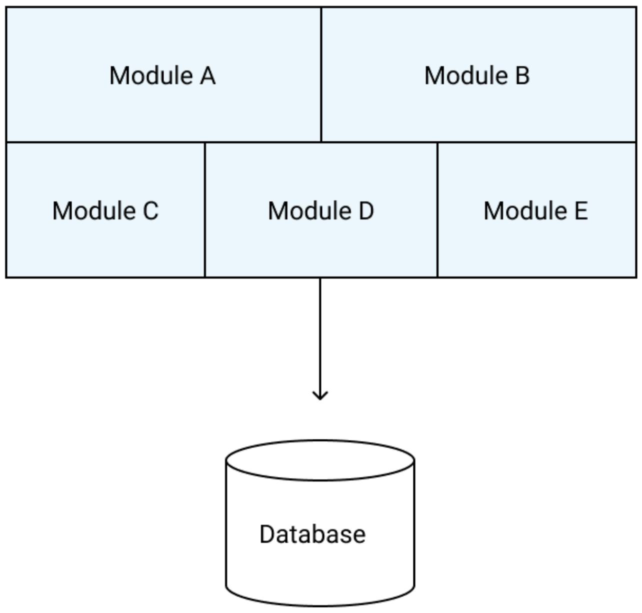


Figure 1. Monolithic architecture (Newman, 2019)

2.2 Microservice architecture

As the complexity grows in requirements due to scaling, embracing microservice architectures enables many organizations to deliver the software faster as well as give them freedom to adopt new technology, react and make different decisions. Microservices alters monoliths with a distributed system of small, loosely coupled and focused services. In microservice architecture world, each service is one independent entity. The more focused the service is, the more benefits the system achieves. As the services get smaller, the interdependence between services are reduced, resulting in the prevention of tightly packaged software risk. Each service exposes its application programming interface (API) for other services to communicate via network calls. The figure below illustrated an example of microservice architecture. (Newman, 2019.) As illustrated in the figure, the system consists of several independent services. Each backend service (account service, inventory service and shipping service) has their own dedicated database. Mobile application sends REST API call to API gateway which forwards the API call to the correct service while web browser makes API call via store front web app service. Each service in the architecture is independent and autonomous.

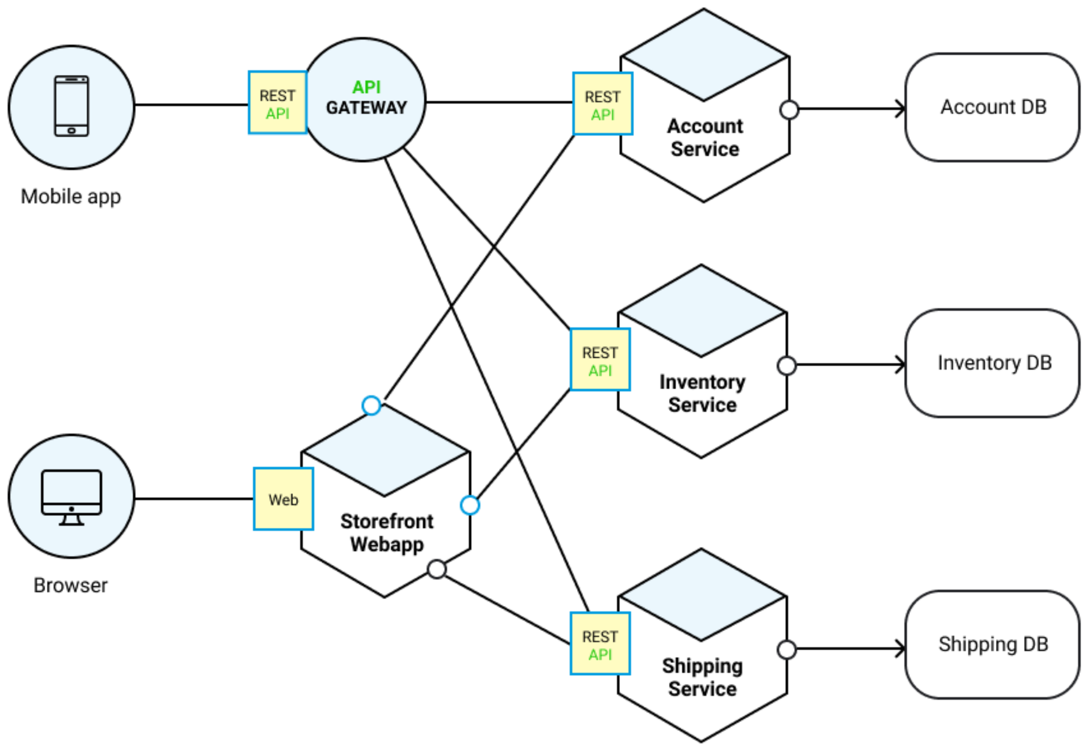


Figure 2. An example of microservice architecture (https://microservices.io)

Microservice architecture is becoming the standard in building large-scale software system for its enormous benefits. Firstly, it enables technology heterogeneity. With a system composed of independent services, development teams can freely adopt new technologies as well as choose different tech stack for each one. This allows them to pick the suitable tool for each use case instead of selecting a one-size-fits-all approach which might end up with performance issues. Resiliency is the second advantage of microservice architecture. The development team is able to isolate the problem on one service without affecting the rest of the system. By doing that, microservices enable engineers to handle failure of complete system. Thirdly, agility in scaling is another profit brought by microservices. Instead of scaling all of modules in monolithic service, development teams can just scale services independently from other services, bringing an enormous benefit in cost management. With microservices, service deployment is independent of the whole system, allowing engineers to deploy much faster. (Newman, 2019.) As a result, the delivery time to market is optimized, saving millions of dollars in opportunity cost for large companies (Humble and Farley, 2010). Microservices allow organizations to effectively divide people into teams. Smaller teams tend to perform productively on smaller codebases. The microservices approach helps companies to achieve the optimal ratio between team size and productivity. Last but not least, services can be rewritten, replaced or removed with minimal effect to the complete system as services are independent. (Newman, 2019.)

2.3 Containers

Nowadays, the operations engineers' life has become much harder with the increasing need and complexity of packaging and deploying numbers of applications (Schenker, 2018). The deployment process of an application does not only require the software itself, but also its dependencies, including libraries, sub-packages, compilers, extensions, and its configuration with settings, site-specific details, keys, database passwords, etc (Domingus and Arundel, 2019). In large enterprises, two applications running on the same production server usually experienced compatibility issue due to different version from the same framework, resulting in slow release cycles (Schenker, 2018). Several earlier attempts had been tried to tackle this problem. Configuration management systems, like Puppet or Ansible, consisting of code to install, run, configure and update the shipping software have been widely utilized. Alternatively, packaging mechanism provided by several specific languages or omnibus package had been applied; however, they did not completely resolve the dependency problem. (Domingus and Arundel, 2019.)

Hypervisor and container are widely accepted answers to this issue. In short, containers virtualize at the operating system (OS) level, while hypervisor-based solutions virtualize at the hardware level, reserving a portion hardware for their products - Virtual Machines (VM) to use. (Wong, 2016.) There are two types of hypervisor-based solutions, including type 1 virtualization and type 2 virtualization. Type 1 virtualization is called as 'bare metal hypervisors', which is a lightweight operating system is installed on a bare metal server or physical computer. In addition, type 2 virtualization is called 'hosted hypervisor'. Instead of being installed on top of hardware, a lightweight operating system is configured on top of a standard operating system. (Zomaya, 2019.) Figure 3 illustrates core differences in structure of hypervisor-based solutions (type 2 hypervisor) and containers.

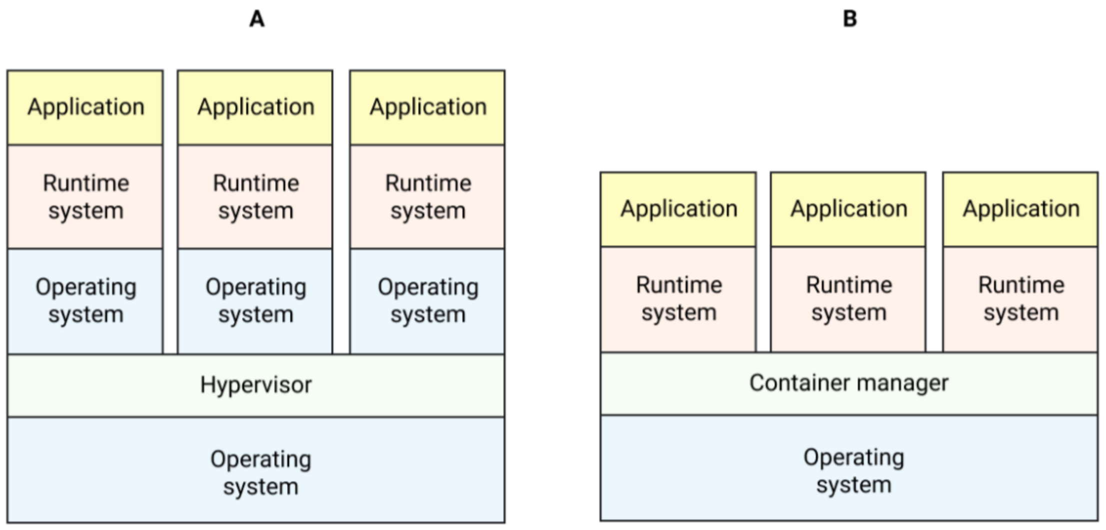


Figure 3. Hypervisor-based Solutions (Virtual Machines) vs Containers (Wong, 2016)

Figure 3 shows that VM and container provide abstraction in different layers. While container provides an abstract OS, a VM is an abstraction at hardware layer. On the VM side, hypervisor technology enables running multiple applications separately on different operating systems, whereas various applications share the same kernel space as the host machine. (Wong, 2016.) A typical VM image size is around 1 Gigabyte, resulting from lots of unrelated programs, libraries, and other unnecessary binaries. In contrast, a well-designed container image’s size might be incredibly smaller, as containers only hold the files needed to run an application. Furthermore, an optimized approach of addressable file system layers, allowing sharing and reusing resources across different containers, is used to further minimize the size of container image. (Domingus and Arundel, 2019.) Last but not least, it would only take seconds for containers to start, compared to minutes for a VM (Chamberlain, 2018). All in all, containers provide an incredibly lightweight, portable and efficient solution.

2.4 Kubernetes

Container technology offers an advantageous solution to resolve software packaging issue; however, several arising issues with containers are still remaining. Firstly, containers provide a weaker level of isolation as they share OS kernel with deep root level of authorization, which carries an enormous potential for attacks to reach underlying OS and eventually into other containers. Secondly, there was lack of container monitoring and managing tool in the industry, resulting in more complex in management compared with Virtual Machines. (Bigelow, 2015). Most importantly, problems with cluster management for containerized applications in cloud services are not tackled yet (Bernstein, 2014). As maintaining communication between containers in microservices is tough, Kubernetes is a solution to tackle these problems (Vohra, 2016, p.41-42).

In 2014, Google founded Kubernetes as an open source orchestration for containerized applications. Its vision was to become the container orchestrator that every company could use. (Domingus and Arundel, 2019.) Building and deploying reliable, scalable distributed systems are effortlessly achieved by adopting Kubernetes, whether companies are creating applications on top of public cloud infrastructure, in private data center or in any hybrid environment (Hightower, Beda and Burns, 2019). Not only the development team but also the operation team will benefit from Kubernetes to produce an exceptionally good product effectively at the minimum cost (Bernstein, 2014).

Kubernetes provides a broad range of features supporting service management and cluster management, and its core benefits can be traced back to one of these benefits, including velocity, scaling (both software and teams), abstracting infrastructure and efficiency. Kubernetes automatically operates computing resources while ensuring the application's availability. Secondly, Kubernetes enables product horizontal scaling based on resource management. Additionally, Kubernetes built-in self-healing feature, which replaces and reschedules nodes, enables failed containers to be restarted. Last but not least, having a huge support community enables Kubernetes to keep introducing new features and improvements, making cluster management and container configuration effortless. (Kubernetes authors, 2019).

2.5 Kubernetes Architecture

2.5.1 Concept

Kubernetes itself is a complicated distributed system; however, its system is actually designed based on a few repeated concepts only. The first concept is declarative configuration. (Tracey and Brendan, 2018.) Declarative is a programming paradigm focusing on the logic and output of a computation without specifying its steps to achieve the desired state (Lloyd, 1994). Kubernetes takes declarative statement from a structured JSON or YAML document and claims the ultimate responsibility for ensuring the result. The real advantage of declarative approach lies in the expression of desired state. As Kubernetes understands the desired output, it can implement autonomous action independently from user interaction, resulting self-correcting as well as self-healing behaviors. As a real world example, Kubernetes takes in a declarative YAML file stating that three copies of certain container image are required to run on different machines, with 3 cores along with a memory of 10 gigabytes per machine. Then, the Kubernetes conducts a review on all of its machines to find an optimal place to run thee container image and eventually schedules the creation of the container on the that machine. The Kubernetes's duty is not only scheduling containers, but also continuously monitoring these containers. If one out of those three containers stated in the YAML file fails to work due to the crash of its internal process, Kubernetes immediately restarts that container to maintain the desired state declared in the given statement. From a developer's point of view, this is ultimately important as the system ensure the availability without any human interception. (Tracey and Brendan, 2018).

The second Kubernetes concept is the structure built from a multitude of independent reconciliation or control loops in order to achieve above-mentioned self-healing or self-correcting behaviors. Kubernetes implements decentralized design pattern, composing a majority of controllers operating their own separate reconciliation loop.

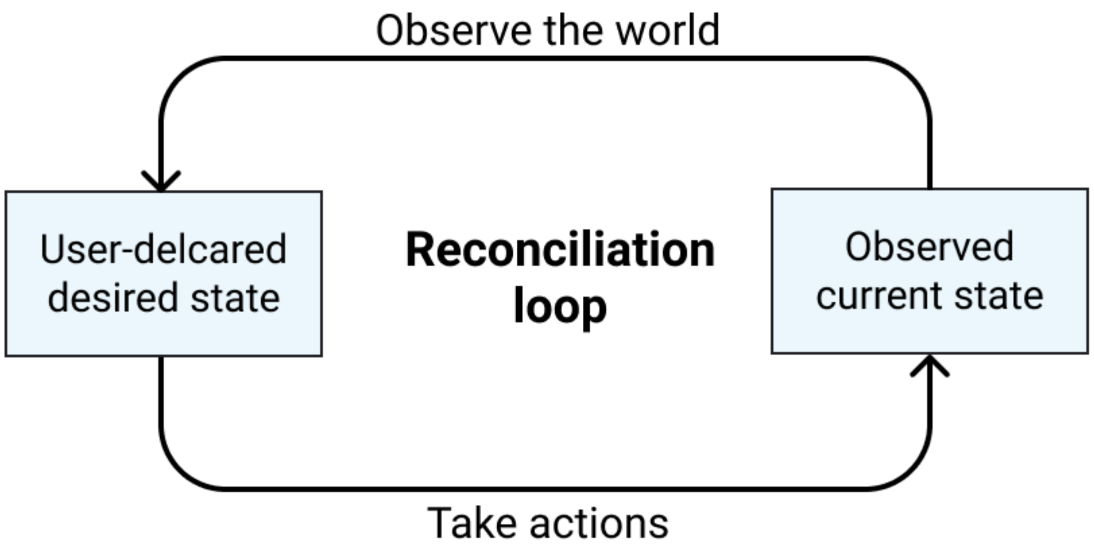


Figure 4. An overview of reconciliation loop (Tracey and Brendan, 2018)

Figure 4 illustrates the operation behind reconciliation loop. Kubernetes controllers' function is literally the same. Firstly, they observed the desired output of the application by the declarative statements sent to Kubernetes API server, followed by the observation of current state. Corresponding actions will be implemented if there is any difference to ensure the matching between current state and desired state. Each loop carries its own responsibility for certain piece of system, while each controller is completely unaware of the rest of other executions, resulting in a considerably stable system. However, the disadvantage of decentralized approach is the complex overall system behavior which requires the understanding of inner execution of independent processes when debugging. (Tracey and Brendan, 2018).

The last Kubernetes concept is implicit (or dynamic) grouping. Grouping allows users to identify a set of objects, which are persistent entities in the Kubernetes system (Kubernetes authors, 2020). When it comes to grouping Kubernetes objects together into a set, there are two approaches, which are explicit/static or implicit/dynamic grouping. While each group is defined by a static list with static grouping, the group is defined by a statement with dynamic grouping. Kubernetes implement dynamic grouping for flexibility as well as stability sake as it can handle an actively changing environment without providing consistent modifications to static lists. This dynamic grouping is accomplished via label selectors, which are key/value pairs associated with each API objects in Kubernetes. Hence, those labels can be used to classify a set objects with matching query. (Tracey and Brendan, 2018). Figure 5 below shows an illustration of labels and selection of label.

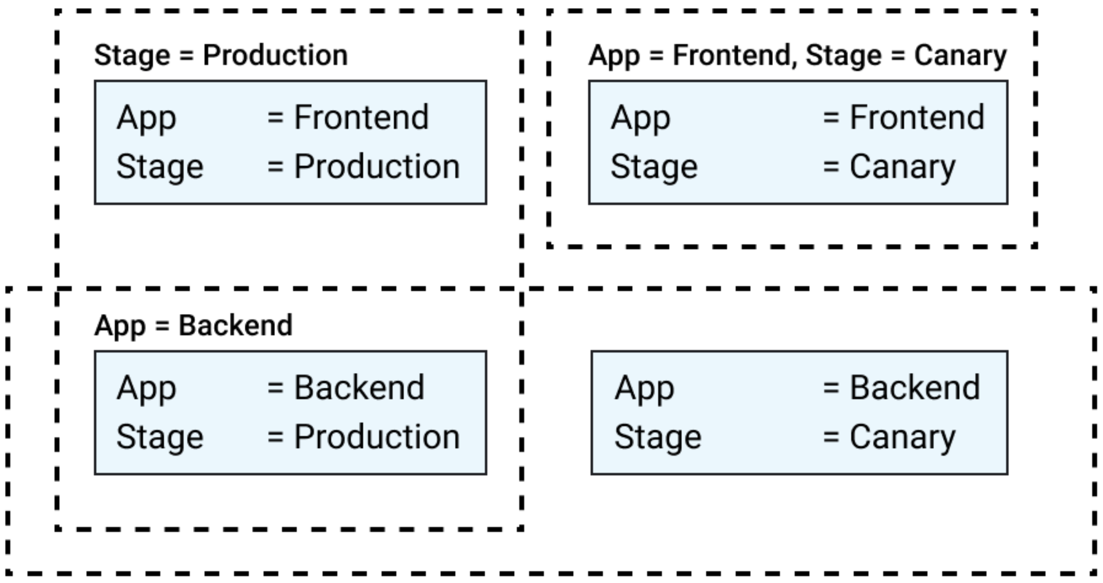


Figure 5. Illustration of labels and selection of label (Tracey and Brendan, 2018)

As can be seen from figure 5, there are four different blue boxes illustrating four running containers. Each container is associated with two different labels including app and stage, describing container’s application name as well as the stage of that application respectively. Label query or label selector in this is utilized to specify containers with the matching query. For instance, if user requires to apply certain changes to production containers only, ‘production’ stage label is queried to grab a set of matching containers without touching other running containers. Using labels enables mapping organizational structures onto loosely coupled system objects (Kubernetes authors, 2020).

2.5.2 Components

Kubernetes cluster can be divided into two different groups, including head nodes and worker nodes (Kubernetes authors, 2019). Figure 6 below shows the diagram if Kubernetes cluster consisting of all components linked together. The architecture illustrated in the figure includes one Kubernetes control plane (head node) and three worker nodes.

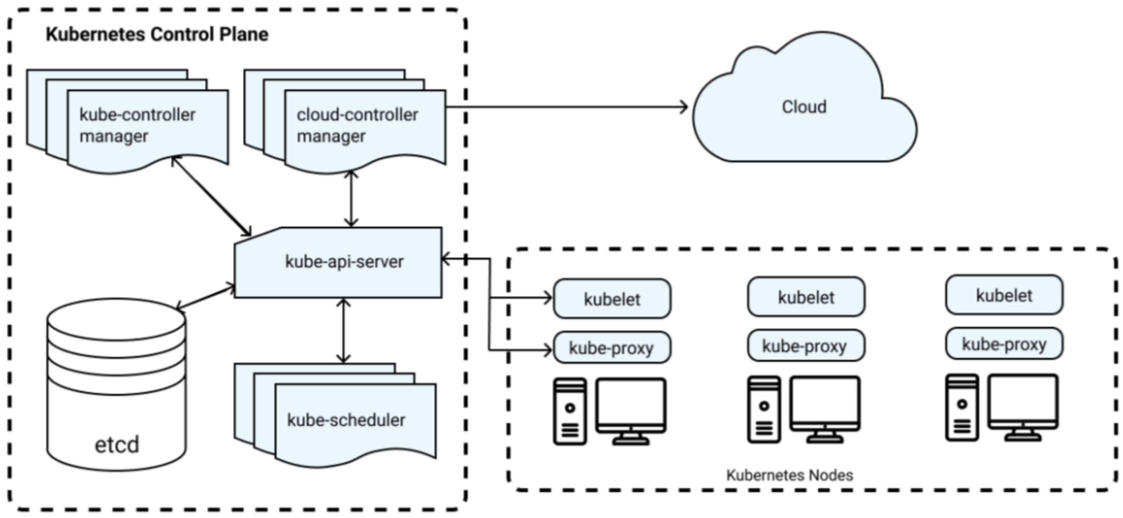


Figure 6. Kubernetes cluster

The implementation of Kubernetes always divides its fleet of machines into two groups, including head nodes and worker nodes. Head nodes (control plane nodes) contain most of the Kubernetes infrastructure's fundamental components like the API server as well as etcd while a limited selection of Kubernetes components implementing the cluster's actual work are running on the worker nodes. The following discussion breaks the Kubernetes components in more detail.

Head Node / Kubernetes Control Plane Components

As shown in figure 6, control plane nodes (or head nodes) includes most of the components that implement Kubernetes such as etcd and Kubernetes API server. The control plane is considered as the cluster's brain as it runs all of the tasks needed for Kubernetes to accomplish its responsibility, ranging from scheduling containers, managing services to serving API requests, and more.

The etcd functions as a persistent and highly available key-value database where it stores all Kubernetes data regarding cluster state, such as the existence of nodes or resources on cluster and so on (Domingus and Arundel, 2019). Raft, a consensus algorithm is implemented by etcd servers, ensuring the data maintenance and recovery in case server storage fails (Tracey and Brendan, 2018.)

While etcd is the heart, the API server is the hub of Kubernetes cluster. It handles all communications between API objects persisted in etcd and clients, resulting in the central touch point for different components inside the cluster. (Tracey and Brendan, 2018.)

Last but not least, controller managers are needed to execute reconciliation control loops that implement several pieces of Kubernetes system (kube-controller-manager) or interact with the underlying cloud providers (cloud-controller-manager). (Tracey and Brendan, 2018). The controller manager is the most diverse components, including Node Controller, Replication Controller, Endpoints Controller, Route Controller, Service Controller, Volume Controller as well as Service Account and Token Controllers (Kubernetes authors, 2019).

Node Components

In addition to head node, each cluster consists of at least one worker node, which actually run user workloads. A few components presenting on these nodes are essential needed to perform worker node’s functionality.

The first component that runs on all worker nodes in Kubernetes is Kubelet, which ensures containers created by Kubernetes are running in a pod (Kubernetes authors, 2019). Additionally, Kubelet monitors, transfers the health state of these containers back to the API server as well as restarts the container if it fails health check. In addition to Kubelet, kube-proxy is the second component running on all machines in cluster. (Managing K8s.) kube-proxy is a network proxy, responsible for network communication between pods inside cluster and between Pods and the internet (Domingus and Arundel, 2019).

Scheduled components

There are several components scheduled to Kubernetes cluster after its initialization, including the cluster DNS services, such as the Kubernetes Service load balancer infrastructure, dashboard, automatic certificate agents, and container monitoring. (Tracey and Brendan, 2018.)

2.6 Kubernetes Objects

2.6.1 Understanding Kubernetes Objects

Kubernetes provides a broad range of objects including Pod, Service, Deployment and so on. Kubernetes Objects are persistent entities representing the state of cluster by describing running containerized applications with available resources and behavioral policies attached to those applications. While the Pod object is the elemental Kubernetes work unit, defining a single or group of containers scheduled together, the Deployment is the Kubernetes resource used to declaratively specify Pods as well as schedule, deploy, update and restart those Pods whenever needed. Furthermore, a Service functions as a load balancer or a proxy, routing traffic to its corresponding Pods via an assigned IP address or DNS name. (Domingus and Arundel, 2019.) Once objects are initialized, Kubernetes ensures their existences by frequently managing, reporting and scheduling. (Kubernetes authors, 2019.)

There are also other important concepts in Kubernetes Objects' world. Names and UIDs are usually used to recognize Kubernetes Objects. While a name is defined in initialization process that refer that object, an UID is generated by Kubernetes system to distinguish that object from other similar entities. Moreover, namespaces are virtual clusters in the same physical cluster, which are utilized to separate between objects as well as prevent name duplication. (Kubernetes authors, 2019.)

As discussed in section 2.5.1, with implicit/ dynamic grouping, labels attach identifying metadata for Kubernetes objects, providing the groundwork for objects grouping via label selector. Kubernetes takes advantage of label selector to filter objects based on a set of labels. Apart from names and UIDs, labels are introduced to be used by multiple objects. Lastly, annotations provide a solution to store additional metadata such as timestamps, release numbers or administrator contact information for Kubernetes objects. Annotations provide key/value metadata storage in a similar manner to labels; however, these metadata are not used to identify or select objects, but only used by external tools such as third-party schedulers or monitoring tools. As labels and annotations are both important, utilizing them properly releases the ultimate power of Kubernetes's flexibility as well as sets the great foundation for constructing deployment workflows or automation tools. (Hightower, Beda and Burns, 2019.)

2.6.2 Working with Kubernetes Objects

Generally, the state of cluster is described by Kubernetes Objects with two equally vital elements which are spec and status. While properly parameter-supplied spec specifies the desired states of that object, status unit holds the information of the object's existing states, provided by Kubernetes system. The current status is continuously monitored and automatically updated by Kubernetes system to ensure its matching with desired state. The communication between objects and clusters is executed smoothly by Kubernetes API. Object initialization process requires object's desired state defined in spec field, along with several necessary parameters which are apiVersion, kind and metadata. The apiVersion field refers to the Kubernetes API version used during the creation process, whilst kind parameter specifies the category of object. As discussed in 2.6.1 section, metadata field holds information supporting unique object identification such as a name string, UID and possibly a namespace. (Kubernetes authors, 2019.) The script below is an example of declaring a deployment object.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: frontend
          image: toanthanh/frontend:latest
          ports:
          - containerPort: 8888
```

Script 1. Example declaring Deployment object

This YAML file will be then created an object of kind Deployment to Kubernetes cluster with frontend name specified in metadata field. The creation of Deployment object is supported with extensions/v1beta1 API version provided in Kubernetes. Spec field specifies the desired states, aiming to create one (number provided in replicas field) pod running container built from frontend image with latest tag pull from Docker Hub repository of toanthanh account.

In order to create objects and interact with API, Kubernetes introduced a powerful command-line tool called kubectl (Hightower, Beda and Burns, 2019). In practice, necessary object fields are specified in YAML Ain't Markup Language (YAML in short) format files called manifests and then provided to kubectl in order to initiate deployment process in clusters. Then, specifications in YAML file are converted into JavaScript Object Notation (JSON) format, followed by injection into request body when making API request. (Kubernetes authors, 2019.) Every object contained in Kubernetes is served by a RESTful resource and exists at an exclusive HTTP path. The kubectl command initiates HTTP requests to these URLs in order to interact with Kubernetes Objects and make changes accordingly. Generally, kubectl is considered as Swiss Army knife of Kubernetes with several basic commands including get, describe, apply, delete. Therefore, kubectl allows users to apply manifests, query resources information, update objects with changes, delete resources and some multiple other tasks. (Domingus and Arundel, 2019.)

For almost all programming languages and operating systems, software installation and maintenance are made easy with the help of its own package manager, such as Debian's apt, Python's pip or JavaScript's npm (Boucheron, 2018). In Kubernetes world, the most popular package manager is called Helm. Helm packing format is known as charts, consisting of several YAML configuration files and templates that are converted into Kubernetes manifest. Helm enables developers and operators to effortlessly package, install, configure as well as deploy applications and services into clusters. (Helm authors, 2019.) By using Helm, the process of configuring and deploying applications in Kubernetes is simplified by letting users maintain only a single set of values and a single set of templates needed for Kubernetes YAML file generation, rather than having to maintain all the raw YAML files. Helm is becoming the standard of Kubernetes with its stability and widespread adoption as it is currently maintained by Cloud Native Computing Foundation projects (Domingus and Arundel, 2019).

2.7 Deploying Kubernetes with Continuous Delivery pipeline

Continuous Delivery is one of DevOps principle keys to build a production-grade application (Domingus and Arundel, 2019). Kubernetes is a tool to orchestrate running containers in the cloud and all of the changes made to running containers as well as Kubernetes configuration files are executed with the help of a continuous delivery pipeline.

2.7.1 Continuous Delivery

Traditionally, the most critical problem of software engineers has been the risky and daunting process of releasing applications as it directly affects application end users. Furthermore, the application delivered no business value until consumed by users. (Humble and Farley, 2010.) The software market has become increasingly competitive compared to the last two decades. To maintain the advantage in this fierce fight, the rapid process of developing and delivering world-class applications has been placed at the first concern. (Shahin, Babar & Zhu, 2017.) With the help of Continuous Delivery (CD), development teams are able to deliver reliable applications safely in short cycles, resulting in continuously bringing application latest updates in an efficient way. Not only bringing the value to development team, adopting CD can also be beneficial for multiple aspects across the company. There are several key benefits of utilizing CD in product development process. (Chen, 2015.)

The first benefit of adopting CD is accelerated time to market. Without the help of CD practice, weeks or months are usually measured in cycle time in many companies (Humble and Farley, 2010).

CD enables companies to stay ahead of competitive market by frequently introduce new enhancements that reinforce the relationship between companies and their end users. The software is able to be released many times a day without any challenges and the development cycle is now drastically reduced to less than a week compared to previous months or weeks. (Chen, 2015.)

The CD’s second benefit is building the right product. Continuous releases let development teams receive user feedback rapidly, enabling them to realize and focus only on high business value features. In the past, teams used to spend extensive amount of time and effort on disadvantageous features without discovering these until after next cycle. Obviously by adopting CD, the user-centric and innovation are placed in the core value of companies for digital transformation. (Chen, 2015.)

Productivity and efficiency have been increased considerably by implementing CD (Chen, 2015). Traditionally, the process of releasing and managing products requires the close cooperation between developers and operators. CD blurs the lines between those team and ingrates them all into overall application performance responsibility. (Humble and Farley, 2010.) All processes and releases involved happens only at the push of a button, empowering the whole team to be highly product as well as efficient by removing manual and repetitive tasks (Itkonen, Udd, Lassenius, Lehtonen, 2016).

Reliable releases lie amongst the core of CD benefits as the risks involved with release process have drastically reduced. Previously, human errors are the most popular factor leading to mistakes during deployment. (Itkonen, Udd, Lassenius, Lehtonen, 2016.) By adopting CD, these mistakes have been automatically detected before exposing to production environment. The difference is minor enough for immediate bug identification and fixes. (Chen, 2015.) Therefore, the simplicity and fast release process has relieved the previous stresses put on development team (Humble and Farley, 2010). Furthermore, release failures are backed by automatic rollback offered by CD, resulting in improving reliable releases (Chen, 2015).

Product quality is dramatically improved as the number of errors has reduced by over 90 percent with the assistance of CD. Previously, bug tracking and fixing requires 30 percent of the team. In contrast, currently the code experiences a series of tests, and bugs will be fixed rapidly before moving to other tasks. Rare production bugs are immediately added to team current sprint plan and fixed within days, compared to waiting months formerly. (Chen, 2015.)

Last but not least, by adopting CD, the relationship between companies and their customers has been enormously strengthened. The enhance in cooperation of developers and operators generates a smooth workflow, resulting in the best products served to companies' end users. Earning the trust from customers might be one of the most rewarding benefits offered by CD. (Chen, 2015.)

2.7.2 Continuous Delivery Pipeline

As discussed in the previous section, Continuous Delivery (CD) practice aims to develop a process which is responsible for delivering new software enhancements to end users constantly and reliably. By the stable and effortless deployment process, CD not only plays in important role in ensuring the high quality for the application, but also strengthens and improves the cooperation between developers and operators, leading to better development cycle. All of those above-mentioned benefits are achieved by building a Continuous Delivery Pipeline. (Phillips, 2014.) A CD pipeline is an automated manifestation of the process of getting application from code to end users (Humble and Farley, 2010). There is no silver bullet or general blueprint for CD pipeline as it differs with various applications. Nevertheless, every pipeline commonly involves integration stage, followed by building, testing and deployment stages. (Phillips, 2014.)

The Figure 7 below illustrates an example of CD pipeline. As the figure shows, this example consists of six different stages and the promotion from one step to another is enabled automatically or manually.

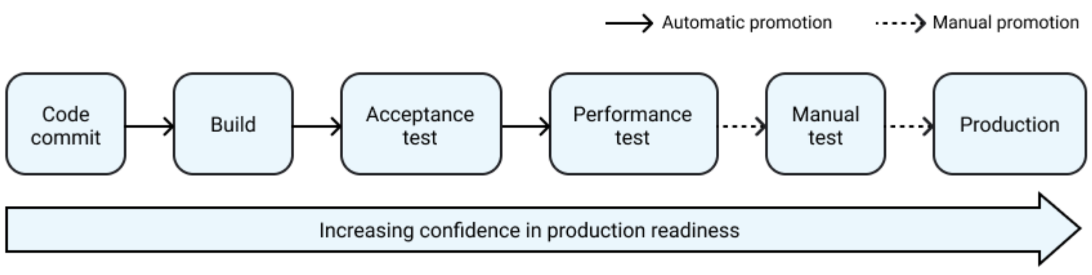


Figure 7. An example of Continuous Delivery pipeline (Chen, 2015)

The first stage of CD pipeline is code commit. Whenever developers push their codes to the code repository, the CD pipeline is triggered by a pre-defined hook to compile the code as run as conduct unit tests. The CD pipeline stops in case there is any errors, then the develop will be informed. The next run will be executed as soon as fixes are pushed to the remote repository. If the stage completes without any error, the advancement to next build stage will be conducted automatically. (Chen, 2015.)

Build is the CD pipeline second stage. Unit tests are run again in order to generate code coverage report, followed by the execution of integration tests and multiple code analysis. At the end of the successful run, the artifacts are built and uploaded to the repository in charge of deployment or distribution process. CD pipeline eliminates the software error occurred while deploying to multiple different environments. The pipeline automatically switches to next stage if the execution completes successfully. (Chen, 2015.)

The CD pipeline automatically continues with a series of test stages, including acceptance test and performance test. The acceptance test stage guarantees the matching between software and user requirements. Several tasks including provisioning and configuring the servers, along with deploying the actual application to them are manually required to create the acceptance test environment. Undoubtedly, this time-consuming setup required human resource with time resource to be accomplished. With the help of CD pipeline, these needs are eliminated, as the pipeline automatically sets up the test environments in the matter of minutes. Secondly, the performance test stage determines the level of software’s performance affected by code change. The CD pipeline assists to conduct performance test in each code commit passed the previous stages, whereas this kind of test was performed only before big releases previously. Lastly, the manual testing environment is automatically created for testers in the manual test stage, whereas human activity was required in setting up phase previously. Furthermore, the testers are notified with message containing the required information to access the pre-setup environment. Like any other stages, the pipeline will exit if there is any error in any stage. After completing all test stages, the application has passed all of the compulsory checks and is ready to be delivered to end users. (Chen, 2015.)

The final stage of the CD pipeline is production. In the past, there were usually failures in this step due to errors in the deployment process or scripts. With the help of CD, as the deployment process as well as scripts have been gone through a series of tests in previous stages, the application is deployed into production only with the click of a button. (Chen, 2015.)

2.7.3 Continuous Delivery with Kubernetes

Continuous Delivery pipeline plays a vital role in deploying a well-functioning Kubernetes cluster as manual Kubernetes application management leads to fragile deployment updates, resulting in the decrease in agility of application delivery. Continuous Delivery starts with a version control, which is a tool to maintain application and configuration code changes history. Then, a series of tests in the pipeline is executed to quickly provide immediate loops of feedback for code changes that break the build, limiting the delivery of bad code into production environment. The pipeline proceeds with container building phase if all test suites have been passed to create an artifact to deploy to an environment. There are multiple approaches to minimize the size of the container image such as multistage build for removing the unnecessary dependencies for the application to run, distroless base images for removing redundant binaries and shells, as well as optimize base images. The successfully built container needs to have proper tags for development team to effortlessly identify the version of the image deployed to a certain environment. Usually, Git Hash tagging strategy is utilized in this execution. (Vallalba, Strebel, Evenson & Burns, 2019.)

Till this point, containers built from the pipeline are ready to be deployed to environments. Containers exist as immutable objects which can be promoted from dev, staging and production environments. Previously, development teams always encountered configuration drift issue with libraries and versioning of components diverging in each environment. Kubernetes tackles this problem by having a declarative way to describe Deployment objects which are versioned as well as deployed consistently.

There are several deployment strategies in Kubernetes world, including rolling updates, blue/green deployments and canary deployments. Rolling updates are built-in functionality in Kubernetes, enabling users to trigger an update to running application without down time. Figure 8 below illustrates an example of Kubernetes rolling update. In the example, rolling updates start by creating a second version of Deployment object which then creates pods running the second version of application (frontend:v2). Then, service object immediately terminates connection to Deployment 1 object and directs all traffic to newest updated second version. There are two points that require attention while implementing this strategy. Firstly, rolling updates strategy can cause connection dropping. Readiness probes and presto life cycle hooks provided by Kubernetes are utilized to ensure the traffic connection to the new pod objects. Secondly, as there are two versions of the application at the same time during the updates, the application’s database schema should support both versions of the application. (Vallalba, Strebel, Evenson & Burns, 2019.)

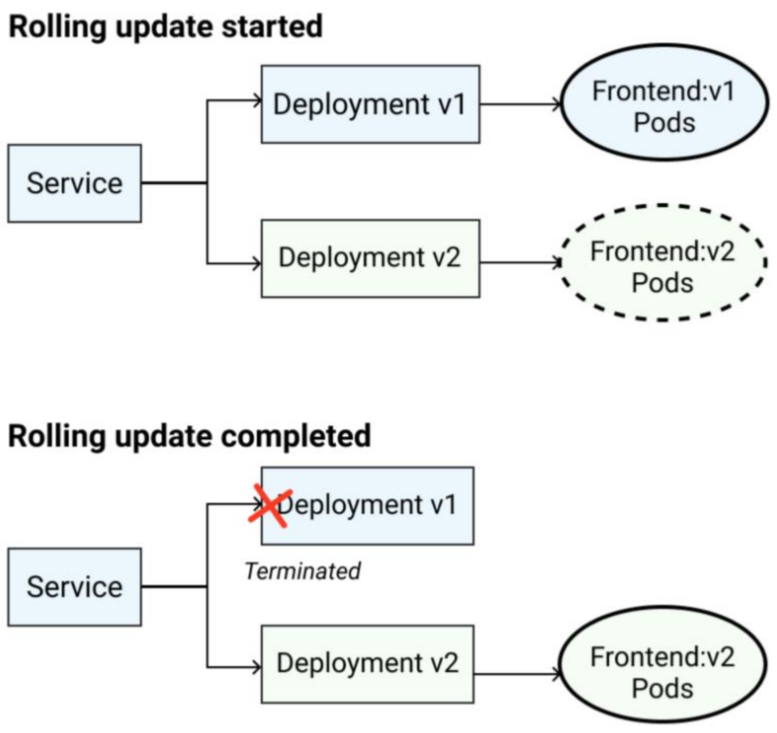


Figure 8. A Kubernetes rolling update (Vallalba, Strebel, Evenson & Burns, 2019)

The second deployment strategy is blue/green rollout. Blue/green deployments enables the application release in a predictable manner, as it grants development teams control when to shift the traffic over to the new environment. This approach requires the internal infrastructure to have enough capacity to deploy both existing and new versions of the application at the same time. Blue/green deployment brings a massive advantage in quickly switching back to the previous version of the application. However, database migrations require development team to consider in-flight transactions as well as schema update compatibility. (Vallalba, Strebel, Evenson & Burns, 2019.) Figure 9 below depicts a blue/green deployment.

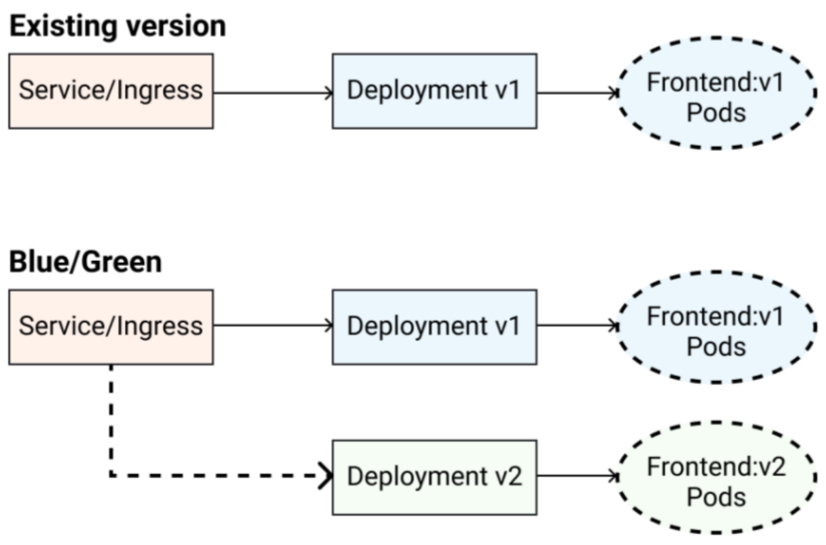


Figure 9. A blue/green deployment (Vallalba, Strebel, Evenson & Burns, 2019)

The last deployment strategy is canary deployment method. This approach is relatively similar to Blue/green strategy; however, it enables a much flexibility in directing traffic to a new release. In practice, this method allows development team to test new features with a subset of users, reducing the risk of delivering broken features to all user base. For example, 10% of traffic can be directed to the new version of the application before shifting a bigger percentage of users there. Canary deployments also work well with advanced techniques to release to certain specific region of users or target users with certain specific profile. (Vallalba, Strebel, Evenson & Burns, 2019.)

2.8 Monitoring Kubernetes

One of the core engineering practices in software system is monitoring. Furthermore, the rise in microservice systems provokes the importance of monitoring. The production Kubernetes cluster experiences the devastating risk without being monitored with a good strategy. (Tracey and Brendan, 2018.) The first and foremost objective of monitoring is reliability. Reliability in this case refers to both Kubernetes cluster and applications running in the cluster. The second feature of monitoring system is providing observability into Kubernetes cluster as it plays a critical role in determining and tracing problems within the system before they end up being incidents. Black-box and white-box are among two common monitoring techniques. Black-box method focuses on monitoring an application from the outside and is often used for components like storage, CPU and memory, whereas white-box method put concentration on monitoring the application state such as total HTTP requests, total of 500 errors, latency of requests, etc. (Burns & Tracey, 2019.) Prometheus tool (https://prometheus.io/) is commonly used to monitor Kubernetes cluster in the inside, while Uptime Robot tool (https://uptimerobot.com/) is utilized to mimic user behavior to check if the application is available to end users (Domingus and Arundel, 2019).

There are several metrics needed to be monitored and those metrics can be grouped into four different layered approaches, including physical or virtual nodes, cluster components, cluster add-ons and end-user-applications. Regarding nodes, metrics that development team might want to monitor are CPU utilization, memory utilization, network utilization and disk utilization. Etcd latency should be put into consideration when monitoring cluster components. Cluster auto scaler and Ingress controller are cluster add-ons related critical metrics to monitor. Last but not least, container memory utilization and saturation, along with container CPU utilization, container network utilization and error rate, combining with application framework-specific metrics are targeted as application layer is being monitored. In general, using this monitoring layered approach enables a more targeted approach to trace problems. (Burns & Tracey, 2019.)

3 Case study

3.1 Case summary

This case study was implemented in order to digitalize a traditional restaurant's operations and services. The case restaurant has more than 10 employees in total and on average 5000 active customers a month. Previously, the process of taking order from customer was a time-consuming process as human manual activity was required. Furthermore, after taking order from customer with pen and paper, the waitress needed to go upstairs to deliver the order paper to kitchen area. This complicated process required lots of human interaction without any help of technology, resulting in lots of errors and mistakes. Several mistakes including the missing of order notes transferred by the waitress or the missing of order notes in the middle of the food preparation by chef and kitchen assistances. The restaurant started a digital project aiming to transform internal processes by applying technology into daily operations.

The objective of this implementation is to solve part of challenges in the day-to-day operation to eliminate human error with the help of a technology solution. The main focus at the first stage in digitalization transformation plan is to build a point of system software. After the scoping meeting, this system's proof of concept should include several core features, such as item with variants listing, the ability to take order and sell from a smartphone or tablet, kitchen display system for meal preparation as well as order management. Several future features have also been put into consideration.

By having a clear vision agreed from the scoping meeting, a proof of concept of the application was implemented successfully. The frontend of the application is written using ReactJS, while the backend of the software is powered by NodeJS. The development and deployment process are run by a CD pipeline hosted on Travis CI (https://docs.travis-ci.com/). Furthermore, as the system needs to be highly available and scalable, the existence of several technology, consisting of Docker container, Docker container registry and Kubernetes hosted on Google Kubernetes Cluster (GKE) fulfill this requirement. In general, the implementation of the software, including backend, frontend, CD pipeline and Kubernetes cluster was successful, resulting in the first step towards the complete digitalization in the restaurant’s internal processes as well as service in the future.

3.2 Case challenges

The software was decided to be deployed in in of the most reliable cloud computing platform. Along with the complexity of architecting and setting up the project from scratch, integration as well as deployment process needs to be put into consideration. There were several challenges needed to be solved at the beginning of the project.

The first challenge was deciding the architecture of the application. Previously, applications were usually designed based on monolithic architecture as they are effortless to build, test and deploy. A monolithic application is a self-contained, single-tiered software application. With this application, as we expected things to growth at a rapid pace in the near future due to a number of features it requires, monolithic architecture seemed not to be an appropriate solution this time.

Technology stack was the second challenge in this project. In the first phase, this application frontend is utilized only by restaurant’s staff; however, the idea is to publish the application so customers visiting the restaurant can order their food at their fingertip without waiting for waitress. Therefore, the application should support multiplatform including tablet, mobile or desktop, regardless of device’s operating system. Furthermore, the client side should maintain user state as well as persistent connection to the backend and database of the application. Database type selection also lay amongst the problems within technology stack scope.

Last but not least, all problems in operating and deployment stage should be foreseen and put into foremost consideration. As the application plays a crucial role in the daily operation, it was expected to be fault-tolerant and highly available. Any outage would result in hundreds of other challenges, risking customer satisfaction and putting monthly financial accounting at risk. As the number of developers will rise over time, setting up a deployment pipeline was put on top of priority list. This approach was used to not only ensure seamless development experience but also eliminate time-consuming process of peer-reviewing, testing and environment management and debugging. In addition, a deployment pipeline enabled the development team to release new versions as quickly as possible, allowing developers to focus on customer-oriented issues and features, rather than spending hundreds of hours debugging an error at deployment process.

3.3 Technology solutions

As discussed in the previous section, there were several challenges described in the project. The first solution aiming to solve the architecture was microservice architecture. The development team decided to design the application following microservice architecture as the system will consist of multiple integrations between services in the near future. Compared to big monolithic application where all services are bundled into a gigantic system, with the help of microservice architecture, each service has its own function and are loosely couple with other services. Across the whole application, each service is packaged as a Docker image as single unit of deployment.

Regarding the tech stack, JavaScript was chosen as the primary programming language in this project. As the application should be available on different platforms and the budget was restrict, the web application approach was adopted. React, which is a library for building user interfaces, was used in the frontend of the application due to its popularity and modular approach on writing components. On the backend side, Express.js framework powered by Node.js, which is prominently used to build APIs, was the final selection in this project to match JavaScript stack. In order to maintain persistent connection between frontend and backend, Socket.io library was the good match as it enabled real-time, bi-directional communication between clients and servers. Regarding the database type, as there will be a large number of read-write operations and the application will deal with a large amount of data with flexibility in data modelling, NoSQL databases were the most suitable type in this scenario. Therefore, the team decided to go with MongoDB, which is a the most popular document-oriented NoSQL database.

Lastly, as highly availability and fault tolerance were put into top priority when developing this software, Kubernetes, which is a container orchestration in large scale microservice application, was adopted due to its core functionalities. Kubernetes aimed to tackle several problems, including service discovery and load balancing, horizontal scaling and self-healing. As a result, these problems mentioned previously would be completely resolved. Furthermore, the help of continuous delivery pipeline was needed in collaboration with Kubernetes. The continuous delivery pipeline not only facilitated us to prevent failures in release stage to production but also made the final deployment to end users effortless with a single click of a button.

3.4 Implementation

All of the problems have been tackled by those solutions discussed in the previous section. The implementation of the whole application will be described in the following parts. First of all, an overview of software architecture is explained for a basic understanding of the software. Then, tools along with services needed to build the Kubernetes cluster and the pipeline of software are mentioned. Last but not least, the process of creating Kubernetes cluster with the continuous delivery is discussed in detail.

3.4.1 Software architecture

Designing the overview of software architecture plays an evitable role in developing any product. The software is a point of sale (POS) system for both internal and customer use in a restaurant. Figure 10 below illustrates the architecture of the Minimum Viable Product (MVP) version of the software.

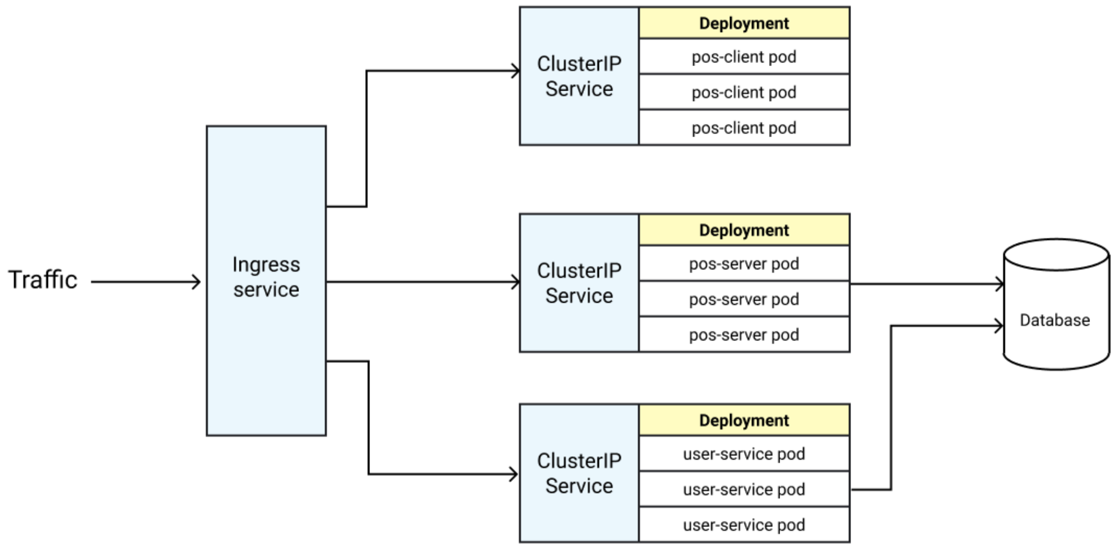


Figure 10. Application architecture

As can be seen from the figure 10, the application contains three services, including a client with persistent connection to its server responsible for order management connected with kitchen display system, a backend API and database, and a user service. The traffic is redirected by Ingress service to the correct service’s Cluster IP service inside the Kubernetes cluster. This architecture was designed to provide the flexibility on adding or removing services in the Kubernetes cluster in the future. Each service deployment unit contains three co-scheduled pods representing three different running containers of the corresponding Docker image. The three pods keep the application highly available as traffic redirected from Ingress is load-balanced by Kubernetes and routed to the available pod. The pod itself does not inherit self-healing ability, therefore pods are created, controlled and monitored by Deployment Kubernetes object. Deployment object ensures the availability of service by immediately restarting the container in case it is shut down for any reason. During the scope of this thesis, the author will main discuss about **pos-client** service, **pos-server** service and how they work with each other under the orchestration of Kubernetes.

3.4.2 Services and tools

Last decade experienced a revolution of cloud computing technology, with the introduction of a variety of different products on the market to build containerized, distributed systems on the cloud. The project utilized the help of multiple available tools, some of those are industry-standard solutions. Those tools and services are introduced explicitly in the next sections.

Google Cloud Platform

Google Cloud Platform (GCP) is a provider of a series of cloud computing services by Google. GCP enables customers to remove the hassle of managing physical infrastructure, along with provisioning servers as well as configuring networks. With GCP, Google offers a wide range of essential benefits, including automated processes, compelling data analytics, hybrid and multi-cloud flexibility and scalable security at an affordable pricing with creative control strategy. Lately, Google has been intensively working on developing container services towards microservice trends in software engineering. Some of these services includes Kubernetes Engine, Istio, Anthos, Container Registry. (Google Cloud Platform authors, 2020.)

Google Kubernetes Engine

Google Kubernetes Engine (GKE) is a controlled platform for containerized applications. These applications can include stateful, stateless, Artificial Intelligence, Machine Learning, complex and simple web applications or any kind of API or backend services. GKE enables rapid development by offering ability to remove operational bottlenecks with auto-repair, auto-upgrade and release channels. Furthermore, GKE provides auto-scaling feature, enabling systems to handle abrupt increase in demand on services. GKE is considered as a modern and smart way to deploy Docker containers, with the help of Kubernetes for effortless management and scaling experience. (Google Kubernetes Engine authors, 2020.)

Docker Hub Container Registry

Docker Hub is world’s largest service for container images that is compatible with popular continuous delivery systems, including Jenkins, Circle CI or Travis CI. Docker Hub offers several convenient features, including auto build and auto test. Auto build enables Docker Hub to pull code from version control management tools like Github or Bitbucket, followed by locating Dockerfile and finally build, tag and push the image into the container. In addition, auto test initiates tests after building the image, stopping the process of pushing new image if anything fails. Furthermore, the security and authentication are also put into top priority with the help of access tokens management and two-factor authentication feature. Private images are accessible only by project members. (Docker authors, 2020.)

Travis CI

Travis CI is a hosted solution offering continuous integration service to build and test software projects hosted on Github. CI is included in the name instead of CD as at first it was developed to solely solve the continuous integration (CI) problem, with an intensive focus on building and testing. However, the change occurred as later they expanded their platform into a continuous delivery and continuous deployment platform. Travis CI tool provides assistance in the pipeline automation from code commit stage till final deployment stage. Also, Travis CI with notification system alerts the developers about build status as well as errors so they can act rapidly. (Travis CI authors, 2020.)

Minikube

Minikube is a tool which creates a local Kubernetes cluster on operating systems including macOS, Linux and Windows provided by Google. Minikube aims to be the best tool for local Kubernetes application experience by supporting several integral Kubernetes features, including Load Balancer, Multi-cluster, NodePorts, Persistent Volumes, RBAC,... (Minikube authors, 2019.)

Skaffold

Skaffold is a developer-focused command line tool developed by Google to provide a swift local development workflow. In local environment, containers are automatically rebuilt and deployed changes into local or remote cluster if any code change is detected. By using Skaffold, enormous amount of time is saved as changes are immediately rendered on local environment without flowing through continuous pipeline and container registry. (Domingus and Arundel, 2019.)

3.4.3 Implementation

Service image configuration

Docker can automatically build images by reading the information in Dockerfile, which is a text document containing all the commands used to assemble an image. The scripts below show the configuration of Dockerfile for client and server services.

```
# client service Dockerfile
FROM node:12.10.0-alpine
WORKDIR /usr/client
COPY package.json ./
RUN npm install
COPY . .
RUN npm run build
CMD ["npm", "run", "start"]
---

# server service Dockerfile
FROM node:12.10.0-alpine
WORKDIR /usr/server
COPY package.json ./
RUN npm i
COPY . .
CMD ["npm", "start"]
```

Script 2. Client & Server images configuration

Service’s deployment and cluster IP configuration

As described in figure 10 regarding the application architecture, the pods are initiated as well as managed by Deployment object. Furthermore, the cluster IP service for each service needs to be setup so that the load balancer can communicate with the correct service. The configuration of deployment objects, along with cluster IP service for client and server services are shown in the scripts below.

```
# client-cluster-ip-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: client-cluster-ip-service
spec:
  type: ClusterIP
  selector:
    component: web
  ports:
    - port: 3000
      targetPort: 3000
```

Script 3. Client service configuration

```yaml
# client-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client-deployment
  labels:
    app: client
spec:
  replicas: 3
  selector:
    matchLabels:
      app: client
  template:
    metadata:
      labels:
        app: client
    spec:
      containers:
        - name: client
          image: toanthanh/pos-client:latest
          ports:
            - containerPort: 3000
```

Script 4. Client deployment object configuration

```yaml
# server-cluster-ip-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: server-cluster-ip-service
spec:
  type: ClusterIP
  selector:
    component: server
  ports:
    - port: 5000
      targetPort: 5000
```

Script 5. Server service configuration

```yaml
# server-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: server-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      component: server
  template:
    metadata:
      labels:
        component: server
    spec:
      containers:
        - name: server
          image: toanthanh/pos-backend
          ports:
            - containerPort: 5000
          env:
            - name: DB_URL
              valueFrom:
                secretKeyRef:
                  name: db_url
                  key: DB_URL
```

Script 6. Server deployment object configuration

Ingress service configuration

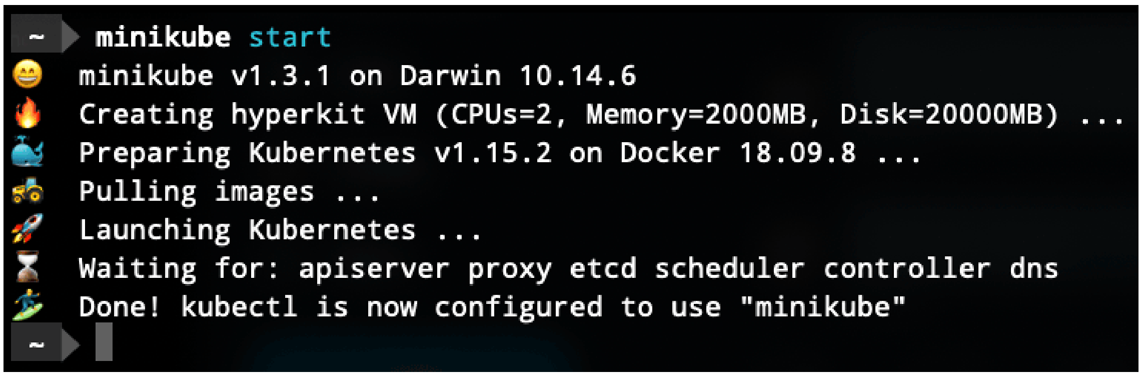
Ingress service main job is to expose HTTP as well as HTTPS routes from outside the cluster to correct services inside the cluster. In this case, it redirects the traffic to https://<cluster-ip>/ to the client service while the traffic to https://<cluster-ip/api/*/ will be routed to server service. The script below illustrates the detailed configuration of Ingress service.

```
# ingress-service.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-service
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
      - http:
          paths:
            - path: /?(.*)
              backend:
                serviceName: client-cluster-ip-service
                servicePort: 3000
            - path: /api/?(.*)
              backend:
                serviceName: server-cluster-ip-service
                servicePort: 5000
```

Script 7. Ingress service configuration

Local Development with Minikube and Skaffold

As introduced in the previous section, Minikube is utilized to stimulate a production Kubernetes cluster on local environment. The figure 11 below illustrates the command of initiating local cluster with Minikube and the process of creating it.



```
~ ▶ minikube start
😄  minikube v1.3.1 on Darwin 10.14.6
🔥  Creating hyperkit VM (CPUs=2, Memory=2000MB, Disk=20000MB) ...
🐳  Preparing Kubernetes v1.15.2 on Docker 18.09.8 ...
🚜  Pulling images ...
🚀  Launching Kubernetes ...
⌛  Waiting for: apiserver proxy etcd scheduler controller dns
🏄  Done! kubectl is now configured to use "minikube"
~ ▶
```

Figure 11. Minikube command and Kubernetes cluster creation process.

Skaffold is the tool used to enhance the local development experience by listening all changes in the project root directory, followed by building and deploying the application to local Kubernetes cluster. Skaffold enables the rapid container update in local Kubernetes cluster without the need of running separate build command, pushing the latest built version to container registry and pulling it back on local cluster. With the assistance of Skaffold, not only an enormous amount of time for the developer are saved, but also the efficiency is boosted. The script below illustrates the configuration file to update the containers inside the Kubernetes cluster with the latest image built from the changes. The script is executed by running command “skaffold dev” in the terminal.

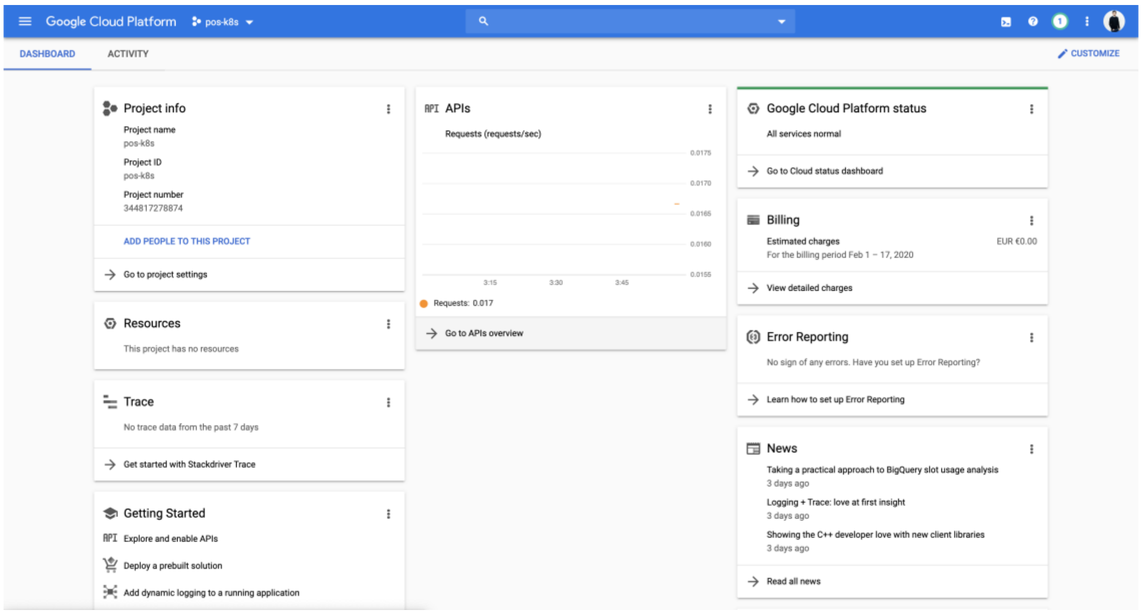
```
# skaffold.yaml
apiVersion: skaffold/v1beta2
kind: Config
build:
  local:
    push: false
  artifacts:
    - image: toanthanh/pos-client
      context: client
      docker:
        dockerfile: Dockerfile
      sync:
        "**/*.js": .
        "**/*.css": .
        "**/*.html": .
    - image: toanthanh/pos-server
      context: backend
      docker:
        dockerfile: Dockerfile
      sync:
        "**/*.js": .
deploy:
  kubectl:
    manifests:
      - k8s/client-deployment.yaml
      - k8s/server-deployment.yaml
      - k8s/server-cluster-ip-service.yaml
      - k8s/client-cluster-ip-service.yaml
```

Script 8. Skaffold configuration

Infrastructure Setup

This phase contains multiple steps, starting with cloud project creation and configuration management to cluster management. The development team decided to go with Google Cloud Platform as the hosting infrastructure in this project. The service management with Google Cloud Platform was implemented extremely effortless with the help of the default command-line interface **gcloud**.

First of all, a Google Cloud project was created as the starting point of using Google Cloud Platform services. The creation of project was seamlessly accomplished by executing the simple command from the command-line interface: `gcloud projects create pos-k8s`. As a result of the command, a project with the ID of **pos-k8s** was created and all cloud services are utilized under this project in Google Cloud Platform. After the project initiation was completed, a web user interface provided by Google Cloud Platform was available to interact with the project as well as the services applied in the project. Figure 12 below shows the dashboard of the project provided by the Google Cloud Platform for the user to access after the creation in Google Cloud Platform.



Project info

Project name
pos-k8s

Project ID
pos-k8s

Project number
344817278874

ADD PEOPLE TO THIS PROJECT

Go to project settings

Resources

This project has no resources

Trace

No trace data from the past 7 days

Get started with Stackdriver Trace

Getting Started

Explore and enable APIs

Deploy a prebuilt solution

Add dynamic logging to a running application

APIs

Requests (requests/sec)

Requests: 0.017

Go to APIs overview

Google Cloud Platform status

All services normal

Go to Cloud status dashboard

Billing

Estimated charges EUR €0.00
For the billing period Feb 1 – 17, 2020

View detailed charges

Error Reporting

No sign of any errors. Have you set up Error Reporting?

Learn how to set up Error Reporting

News

Taking a practical approach to BigQuery slot usage analysis
3 days ago

Logging + Trace: love at first insight
3 days ago

Showing the C++ developer love with new client libraries
3 days ago

Read all news

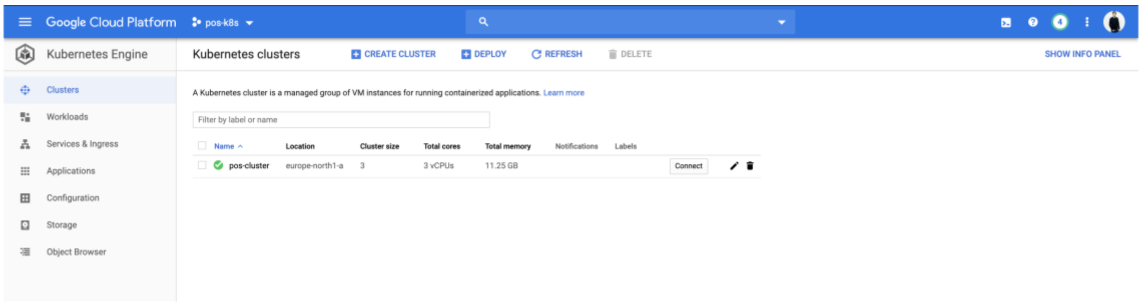
Figure 12. Cloud project dashboard

Then, the existence of a Kubernetes cluster was needed in order to deploy the application. Again, the cluster creation process was made seamlessly effortless by utilizing **gcloud** command-line interface. The script below illustrates the command executed to initiate a cluster with **gcloud**.

```
gcloud container clusters create pos-cluster \
    --project "pos-k8s" \
    --zone "europe-north1-a" \
    --no-enable-basic-auth \
    --cluster-version "1.13.12-gke.25" \
    --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" \
    --subnetwork "default" \
    --enable-legacy-authorization \
    --addons HorizontalPodAutoscal-
ing,HttpLoadBalancing,KubernetesDashboard \
    --enable-autoupgrade \
    --enable-autorepair
```

Script 9. Google Cloud Kubernetes cluster creation configuration

The cluster creation took several minutes for the Google Kubernetes Engine to process the infrastructure acquired from the script, enable Kubernetes API inside the project and install the software. After the success of cluster creation, the cluster as well as its status were displayed in the Google Kubernetes Engine dashboard.



A Kubernetes cluster is a managed group of VM instances for running containerized applications. Learn more

Name	Location	Cluster size	Total cores	Total memory	Notifications	Labels
pos-cluster	europe-north1-a	3	3 vCPUs	11.25 GB		

Figure 13. Google Kubernetes dashboard

As illustrated from the Figure 13, the newly created *pos-cluster* cluster was ready for deployments. In order to assign the deployment process in the pipeline the possibility to handle the cluster as well as deploy applications inside it, a separate account containing the correct rights were needed. In Google Cloud Platform terms, these accounts are called service accounts. In the scope of this project, only 1 service account was needed to deploy applications to the Kubernetes cluster. The execution script of this service account creation is illustrated in Listing in Appendix 1. As a result, a downloadable JSON file containing the service account's private key was available to be downloaded. This JSON file then was encrypted to be used in Continuous Delivery pipeline without exposing confidential information when being committed to Github. All in all, the infrastructure setup step was completed by achieving a running cluster and a service account with the correct roles to deploy applications to the cluster.

Continuous Delivery platform configuration

The scripts specified in the previous section were ready to be run in the actual pipeline. The last step of the implementation phase was to config the Continuous Delivery pipeline flow with the assistance of Travis CI platform. Travis CI provides an effortless way to define and trigger Continuous Delivery pipeline by simply putting a configuration file (**.travis.yml**) in the project root directory. The detail of the configuration file in YAML format is displayed in the script below.

```
sudo: required
services:
  - docker
env:
  global:
    - SHA=$(git rev-parse HEAD)
    - CLOUDSDK_CORE_DISABLE_PROMPTS=1
before_install:
  - openssl aes-256-cbc -K $encrypted_0c35eebf403c_key -iv
$encrypted_0c35eebf403c_iv -in service-account.json.enc -out service-
account.json -d
  - curl https://sdk.cloud.google.com | bash > /dev/null;
  - source $HOME/google-cloud-sdk/path.bash.inc
  - gcloud components update kubectl
  - gcloud auth activate-service-account --key-file service-
account.json
  - gcloud config set project pos-k8s
  - gcloud config set compute/zone europe-north1-a
  - gcloud container clusters get-credentials pos-cluster
  - echo "$DOCKER_PASSWORD" | docker login -u "$DOCK-ER_USERNAME" --
password-stdin
  - docker build -t toanthanh/react-test -f ./client/Dockerfile.dev
./client

script:
  - docker run toanthanh/react-test npm test -- --coverage

deploy:
  provider: script
  script: bash ./deploy.sh
  on:
    branch: master
```

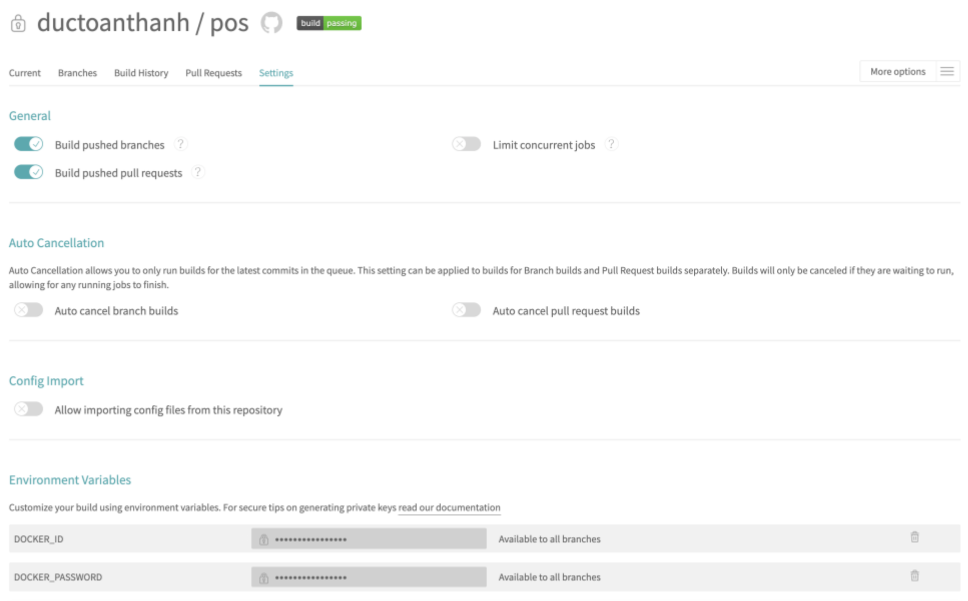
Script 10. Travis CI configuration

In the env section, the SHA variable was the unique identifier of the commit used in Github. By using SHA, the Docker image was added with a unique tag along with the default latest tag whenever the commit was created, allowing the Kubernetes cluster always stayed updated with the latest version of the application. The **before_install** job lifecycle was used to setup **gcloud** command-line tool with the correct information of project and cluster inside the container running the deployment process. The desired result of this step was to be able to access the correct cluster with the encrypted service account rights and to be ready for actual application deployments. Then, the **script** job lifecycle was used to run tests. If there was any error in this stage, the pipeline exited and returned an exit code of 1. Otherwise, the deployment process continued with the **deploy** job lifecycle. As multiple commands needed to be executed in this phase ranging from image building and tagging to Kubernetes cluster deploying, an executable wrapper script was created. The details of executable file was shown in below.

```
# deploy bash script
docker build -t toanthanh/pos-client:latest -t toan-thanh/pos-
client:$SHA -f ./client/Dockerfile ./client
docker build -t toanthanh/pos-server:latest -t toan-thanh/pos-
server:$SHA -f ./backend/Dockerfile ./backend

# push images with latest
docker push toanthanh/pos-client:latest
docker push toanthanh/pos-server:latest

# push images with SHA tag
docker push toanthanh/pos-client:$SHA
docker push toanthanh/pos-server:$SHA

# deploy applications to Kubernetes
kubectl apply -f k8s
kubectl set image deployments/server-deployment server=toanthanh/pos-
server:$SHA
kubectl set image deployments/client-deployment client=toanthanh/pos-
client:$SHA
```

Script 10. Executable script in deploy hook in Travis CI configuration

The previous script was used to build and tag images, push them to Docker Hub, followed by enabling Kubernetes cluster set in **before_install** lifecycle to pull the image from Docker Hub and put them into container orchestrated by Kubernetes. In order for the whole process to work, environment variables needed to be declared in Travis CI. These environment variables were effortlessly setup by navigating to the setting page in Travis CI dashboard. Figure 14 below illustrates the configuration of environment variables in Travis CI.



ductoanthanh / pos build passing

Current Branches Build History Pull Requests Settings More options

General

Build pushed branches ✓ | Limit concurrent jobs
Build pushed pull requests ✓

Auto Cancellation

Auto Cancellation allows you to only run builds for the latest commits in the queue. This setting can be applied to builds for Branch builds and Pull Request builds separately. Builds will only be canceled if they are waiting to run, allowing for any running jobs to finish.

Auto cancel branch builds | Auto cancel pull request builds

Config Import

Allow importing config files from this repository

Environment Variables

Customize your build using environment variables. For secure tips on generating private keys read our documentation

DOCKER_ID	••••••••••••••••	Available to all branches
DOCKER_PASSWORD	••••••••••••••••	Available to all branches

Figure 14. Environment variables configuration in Travis CI

3.5 Evaluation

The implementation of the point of sale minimum viable product was a success with running application on Kubernetes cluster hosted on Google Cloud Platform. The development team successfully developed its core features as well as efficient and reliable release process. The outcome of the project met the expectation of both business objectives as well as technical objectives.

Business-wise, the application brought a solution to one the most challenging day-to-day operation, enabling smooth order transition from counter to kitchen area. The application with basic features including item listing, order placing, kitchen display system was created to immediately solve existing problems, reducing manual work for customer service employees, helping them to provide better user experience. Kitchen area now had the system to track orders sent from the counter personnel. Additionally, chefs could notify customer service people as soon as the food is ready to be served by simply marking the order as done. The success of this proof-of-concept project not only transformed daily operation by technology, but also opened door for future opportunities with the aim of providing the best possible point of system service by adding features like warehouse management, loyalty program application for end users. The layout of the applications is illustrated in the following pictures.

Figure 17 illustrates the counter area user interface. Items were listed based on category (appetizer, main, drink or dessert) and the prices were adjusted based on time, whether the current time was lunch, dinner or weekend. On the right panel, they could effortlessly monitor all current orders and prices to charge later on without asking the customers what they had previously.

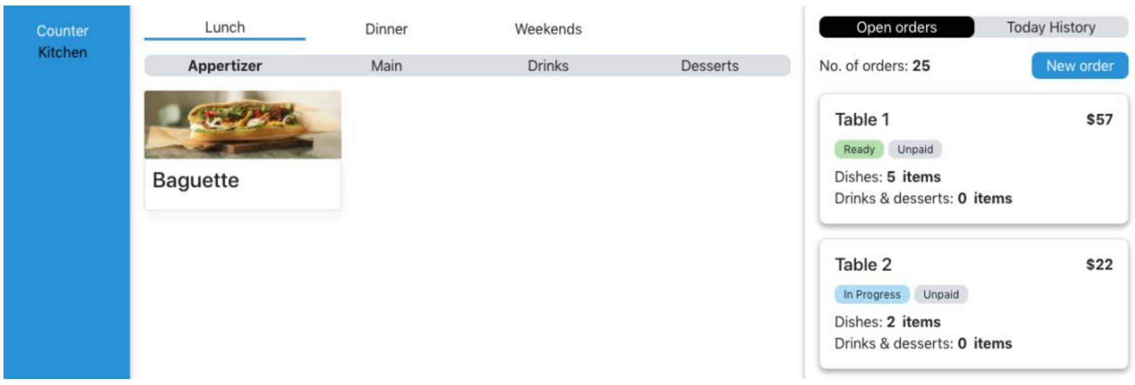
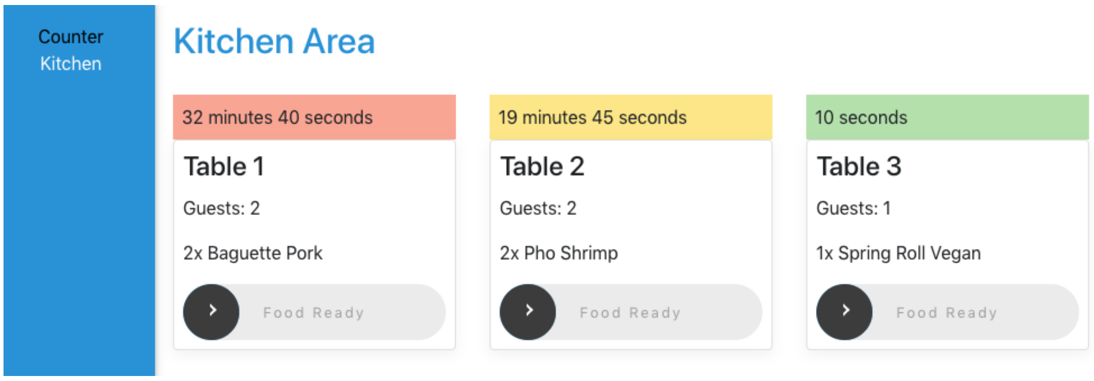


Figure 15. Application’s counter user interface

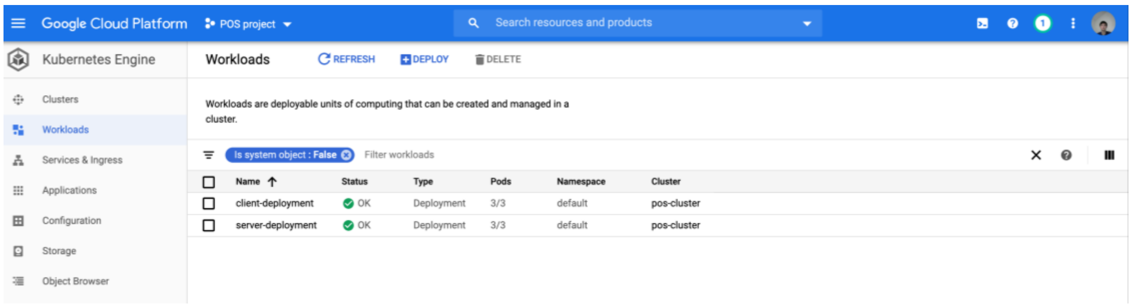
The figure 18 describes the kitchen area user interface. Orders were displayed and sorted by time from left to right, with latest orders appeared in the end. The order card’s header showed the time since customer placed an order and the header background color was adjusted accordingly. As the restaurant aimed to serve the customer as soon as possible, and time-from-order-till-serve was one of the important metrics to customer satisfaction, showing the time enabled chefs to focus and deliver the good quality food within the satisfaction time.



32 minutes 40 seconds	19 minutes 45 seconds	10 seconds
Table 1	Table 2	Table 3
Guests: 2	Guests: 2	Guests: 1
2x Baguette Pork	2x Pho Shrimp	1x Spring Roll Vegan

Figure 16. Application's kitchen area visual

Looking from technical perspective, the implementation also met the initial objectives set from the beginning. Firstly, the application was successfully designed with microservice architecture. Two current running services were client and server. Each service was loosely coupled and packaged into a Docker container as a single unit of deployment. Future services can be easily integrated into the system. The development team successfully deployed working containers into Kubernetes cluster. Figure 19 below illustrates Kubernetes's workload.

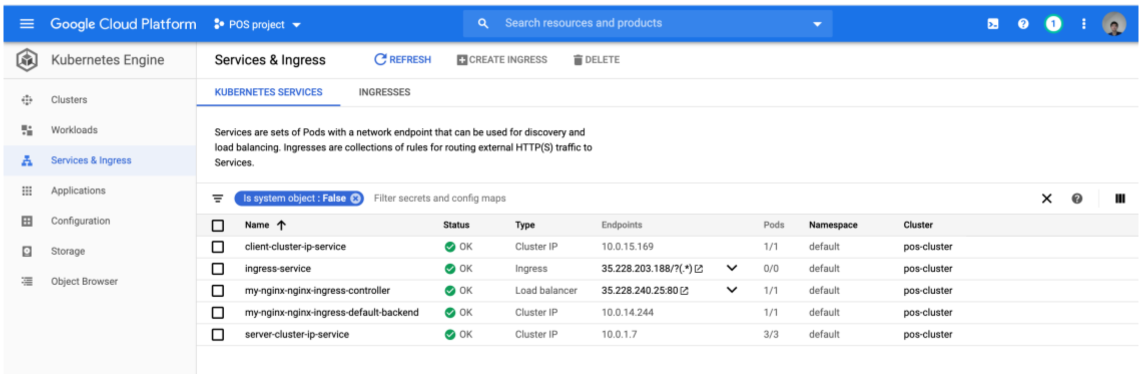


Workloads are deployable units of computing that can be created and managed in a cluster.

Name	Status	Type	Pods	Namespace	Cluster
client-deployment	OK	Deployment	3/3	default	pos-cluster
server-deployment	OK	Deployment	3/3	default	pos-cluster

Figure 17. Kubernetes cluster's workload

As shown in the figure, two services were running on six different pods managed directly by client-deployment and server-deployment objects. In case one pod was down for any reason, the client-deployment or server-deployment objects will restart the pod immediately, providing the zero-downtime feature for the whole application. The figure 20 shows Kubernetes cluster's services and ingress.



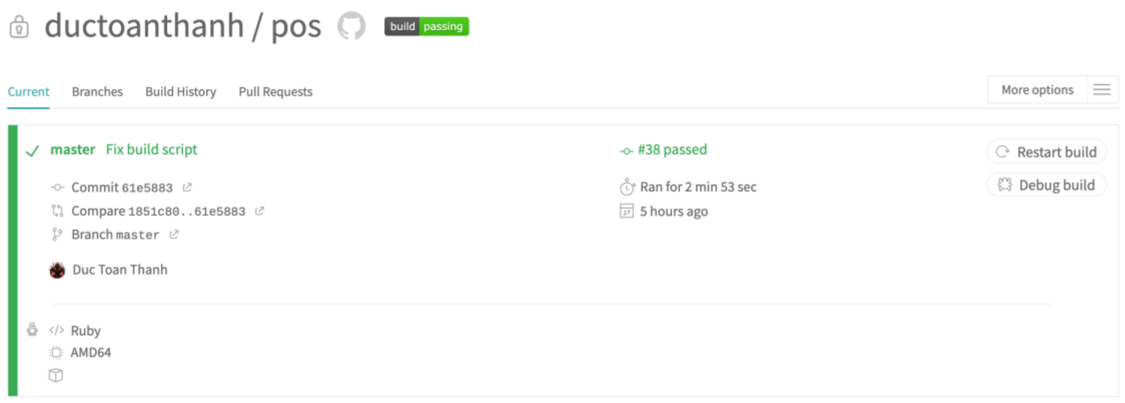
Services are sets of Pods with a network endpoint that can be used for discovery and load balancing. Ingresses are collections of rules for routing external HTTP(S) traffic to Services.

Name	Status	Type	Endpoints	Pods	Namespace	Cluster
client-cluster-ip-service	OK	Cluster IP	10.0.15.169	1/1	default	pos-cluster
ingress-service	OK	Ingress	35.228.203.188/?(.*)	0/0	default	pos-cluster
my-nginx-nginx-ingress-controller	OK	Load balancer	35.228.240.25:80	1/1	default	pos-cluster
my-nginx-nginx-ingress-default-backend	OK	Cluster IP	10.0.14.244	1/1	default	pos-cluster
server-cluster-ip-service	OK	Cluster IP	10.0.1.7	3/3	default	pos-cluster

Figure 18. Kubernetes cluster’s services & ingress

Referring back to application architecture (figure 9), the Ingress service received outside traffic and forwarded the traffic to the correct services. As mentioned previously, each service ran inside three different pods and each pod had its own IP address. As IP address of each pod was dynamic, client-cluster-ip-address and server-cluster-ip-address maintained the unique endpoints to these sets of pods, therefore traffic received from Ingress service could be load-balanced to the correct place.

In general, a fully-functioning, highly available, fault-tolerant and effortlessly scalable application was successfully deployed to Kubernetes cluster in Google Kubernetes Engine. Furthermore, deployment process - one of the objectives stated in the beginning, was also be solved with the help of a delivery pipeline. The delivery pipeline helped the team to avoid pushing bad codes into production environment, which prevented outages. In addition, it enabled new versions to be released as quickly as possible, allowing development team to focus on features, rather than spending hours experiencing production downtime as well as debugging error at deployment process. The code after merged into master branch will automatically deploy to Kubernetes cluster without any manual configuration. As the development team will grow time after time, setting up a delivery ensured the seamless development workflow in the future. Figure 21 shows the user interface of a successful build in Travis CI tool.


ductoanthanh / pos

build passing

Current Branches Build History Pull Requests More options

master Fix build script

Commit 61e5883

Compare 1851c80..61e5883

Branch master

Duc Toan Thanh

#38 passed

Ran for 2 min 53 sec

5 hours ago

Restart build

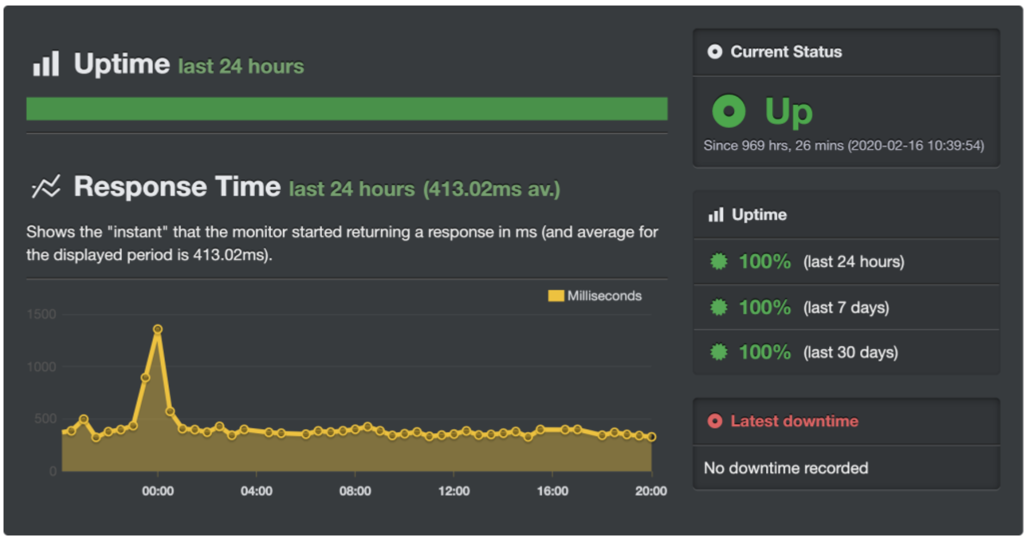
Debug build

Ruby

AMD64

Figure 19. A Travis CI successful build

Last but not least, even though Kubernetes cluster itself provided zero-downtime feature, setting up an external monitoring tool is important to provide a second layer of external monitoring. Therefore, Uptime Robot tool has been configured to track the availability of the application. Constant checks were sent with the interval of 5 minutes and development team will be alerted in case the cluster is down. The application has performed 100% uptime since the tool was configured. Figure 22 illustrates application uptime.


Uptime last 24 hours

Response Time last 24 hours (413.02ms av.)

Shows the "instant" that the monitor started returning a response in ms (and average for the displayed period is 413.02ms).

Current Status

Up

Since 969 hrs, 26 mins (2020-02-16 10:39:54)

Uptime

100% (last 24 hours)

100% (last 7 days)

100% (last 30 days)

Latest downtime

No downtime recorded

Figure 20. Application uptime monitored by Uptime Robot

3.6 Future development

Though the implementation of the project achieved the initial goals regarding both business objectives as well as technical objectives, there are several rooms for improvement in the future. Firstly, regarding feature point of view, as the application is serving only one restaurant, the multi tenancy architecture can be utilized to transform the application into a large-scale Software as a Service (SaaS) solution. Also, the authentication, role-based access, analytics features should be put into development backlog to give users an overview of sales numbers as well as fully support multiple users with different access rights. More unit tests should be put into consideration for effortless application debugging process in the long run. Lastly, from infrastructure perspective, as the current pipeline depends on several services such as Travis CI and Docker Hub, implementing a centralize and controllable system should be put into consideration. This effort can be done by moving the pipeline into Cloud Build as well as utilizing Google Container Registry, all provided by Google Cloud Platform. All in all, the application and infrastructure setup are still at the first version, and there are considerable improvements can be implemented in the future.

4 Conclusion

This thesis focuses on building a production-grade pipeline for an application deployed to Kubernetes cluster hosted in Google Cloud Platform while discussing the advantages of using Docker container technology with Kubernetes as container orchestration. In the past few years, as technology is the drive of industrial revolution, a considerable amount of software has adopted microservice architecture. Docker now has been recognized as the new standard for container technology (Bernstein, 2014). Besides, Kubernetes are transforming the creation and deployment of applications by fundamentally giving developers more velocity, efficiency and agility (Hightower, Beda and Burns, 2019). Kubernetes offers enormous help not only in delivering containerized application, but also in clustering management (Kubernetes Authors, 2019). Furthermore, Kubernetes also plays an important role in every microservice software project nowadays.

To sum up, the thesis successfully illustrated how to setup a modern production-grade pipeline to deliver containerized application to Kubernetes clusters in Google Cloud Platform. Moreover, the project outcome meets not only the technical but also the business objectives set for this study. By using Docker container technology, dependency resource management and isolation of environments problems are completely resolved. Kubernetes with rich feature set and application support, combined with outstanding community and industry support will be ultimately beneficial for all applications. However, there are still several impediments while implementing Kubernetes. First of all, Kubernetes can be an overkill solution for small applications with simple architecture. Secondly, Kubernetes learning curve is steep enough so that it might reduce productivity for developers in the transition phase. Last but not least, Kubernetes can be a much more expansive approach than its alternatives, especially when it comes to simple applications. All in all, applying Kubernetes in software projects seems to bring organizations and companies flexibility, power and scalability regarding human resources and costs.

References

Bernstein, D. (2014) ‘Containers and Cloud: From LXC to Docker to Kubernetes’ IEEE Cloud Computing 1(3) pp.81–84 [Online] Available at: https://ieeexplore.ieee.org/document/7036275 (Accessed: 09 December 2019)

Bigelow, S. (2015). Five cons of container technology. [Online]. Available at: https://searchservervirtualization.techtarget.com/feature/Five-cons-of-container-technology (Accessed: 16 March 2020)

Boucheron, B. (2018). An Introduction to Helm, the package manager for Kubernetes. [Online]. Available at: https://www.digitalocean.com/community/tutorials/an-introduction-to-helm-the-package-manager-for-kubernetes (Accessed: 16 January 2020)

Chamberlain, D. (2018). Containers vs. Virtual Machines (VMs): What’s the Difference? [Online] Available at: https://blog.netapp.com/blogs/containers-vs-vms/ (Accessed: 09 December 2019)

Docker Authors. (2019). Docker Hub. [Online] Available at: https://docs.docker.com/docker-hub/ (Accessed: 31 December 2019)

Domingus, J. & Arundel, J. (2019). Cloud Native DevOps with Kubernetes. [Online] Available at: https://learning.oreilly.com/library/view/cloud-native-devops/9781492040750/ch01.html (Accessed: 08 December 2019)

Google Inc. (2019). Google Kubernetes Engine. [Online]. Available at: https://cloud.google.com/kubernetes-engine/ (Accessed: 10 March 2020)

Hightower, K., Beda, J. and Burns, B. (2019). Kubernetes: Up and Running, 2nd Edition. [Online] Available at: https://learning.oreilly.com/library/view/kubernetes-up-and/9781492046523/ (Accessed: 15 December 2019)

Humble, J. & Farley, D. (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation [Online] Available at: https://learning.oreilly.com/library/view/continuous-delivery-reliable/9780321670250/ (Accessed: 08 December 2019)

Itkonen, J., Udd, R., Lassenius, C., Lehtonen, T. (2016). Perceived Benefits of Adopting Continuous Delivery Practices. [Online] Available at: https://dl.acm.org/doi/10.1145/2961111.2962627 (Accessed: 15 January 2020)

Kubernetes Authors. (2019). Kubernetes Object Management. [Online] Available at: https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/ (Accessed: 19 December 2019)

Kubernetes Authors. (2019). Kubernetes | Production-Grade Container Orchestration [Online] Available at: https://kubernetes.io/ (Accessed: 08 December 2019)
Kubernetes Authors. (2019). Understanding Kubernetes Objects. [Online] Available at: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/ (Accessed: 19 December 2019)

Luksa, M. (2018). Kubernetes in Action. [Online] Available at: https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/ (Accessed: 02 December 2019)

Newman, S. (2015). Building Microservices. [Online] Available at: https://learning.oreilly.com/library/view/building-microservices/9781491950340/ (Accessed: 03 December 2019)

Newman, S. (2019). Monolith to Microservices. [Online] Available at: https://learning.oreilly.com/library/view/monolith-to-microservices/9781492047834/ (Accessed: 03 December 2019)

Phillips, A. (2014). The continuous delivery pipeline – What is it and Why it's so important in developing software. [Online]. Available at: https://devops.com/continuous-delivery-pipeline/ (Accessed: 15 January 2020)

Richardson, C. (2018). Microservices patterns. [Online] Available at: https://learning.oreilly.com/library/view/microservices-patterns/9781617294549/ (Accessed: 04 December 2019)

Schenker, G. N. (2018). Learn Docker - Fundamentals of Docker 18.x. [Online] Available at: https://learning.oreilly.com/library/view/learn-docker-/9781788997027/5bbf53c5-e15c-4c56-90df-9d3efe2204bb.xhtml (Accessed: 08 December 2019)

Stubbs, J., Moreira, W. and Dooley, R. (2015) 'Distributed Systems of Microservices Using Docker and Serfnode' 2015 7th International Workshop on Science Gateways pp.34–39 [Online] Available at: https://ieeexplore.ieee.org/document/7217926 (Accessed: 04 December 2019)

Tracey, C. and Burns, B. (2018). Managing Kubernetes. [Online] Available at: https://learning.oreilly.com/library/view/managing-kubernetes/9781492033905/ (Accessed: 15 January 2020)

Travis CI Authors. (2020). Travis CI Documentation. [Online] Available at: https://docs.travis-ci.com/ (Accessed: 10 March 2020)

Villalba, E., Strebel, D., Evenson, L. and Burns, B. (2019). Kubernetes Best Practices: Blueprints for Building Successful Applications on Kubernetes. [Online]. Available at: https://learning.oreilly.com/library/view/kubernetes-best-practices/9781492056461/ (Accessed: 18 March 2020)

Vohra, D. (2016) Kubernetes Microservices with Docker [Online] Available at: https://learning.oreilly.com/library/view/kubernetes-microservices-with/9781484219072 (Accessed: 10 December 2019)

Wong, W (2016). What’s the difference between Containers and Virtual Machines [Online] Available at: https://www.electronicdesign.com/technologies/dev-tools/article/21801722/whats-the-difference-between-containers-and-virtual-machines (Accessed: 08 December 2019)

Zomaya, D. (2019). Container vs. Hypervisor: What’s the Difference? [Online]. Available at: https://www.cbtnuggets.com/blog/certifications/cloud/container-v-hypervisor-whats-the-difference (Accessed: 16 March 2020)