

Anh Dang (D5057)

AUTOMATED APPLICATION DEPLOYMENT TO KUBERNETES ON GOOGLE CLOUD PLATFORM

Bachelor's thesis

Bachelor of Engineering, Information Technology

2021



South-Eastern Finland
University of Applied Sciences

Author (authors)	Degree title	Time
Anh Dang	Bachelor of Engineering, Information Technology	May 2021
Thesis title Automated Deployment to Kubernetes on Google Cloud Platform		35 pages
Commissioned by		
Supervisor Matti Juutilainen		
<p>Abstract</p> <p>This work explores the capabilities of an automated continuous delivery to Kubernetes in a cloud native environment. With Kubernetes becoming such a prominent technology in the software development field, its advantages provide an opportunity to further optimize the development process. In addition, the utilities of Kubernetes also facilitate the drive towards microservices architecture. All these new technologies were explored in this work and examined closely through an implementation of an example project.</p> <p>A cloud native approach through Google Cloud Platform was chosen as the example, as the author had ample experience with the cloud provider and also engaged in a similar project during his time as an intern for CloudAce Vietnam.</p> <p>The research method for this thesis relies on analysis of the components involved and how they fit in the context of automated deployment. The deployment process was closely explained so as to provide a detailed view of a functioning Kubernetes and Cloud Build system.</p> <p>The key result for the project was the correct delivery and deployment of an application to Kubernetes through a cloud native continuous delivery pipeline. Since this project served as a model for a deployment pipeline, it could be used as a reference point for developers looking to deploy their application on GKE.</p>		
<p>Keywords</p> <p>Kubernetes, continuous delivery, Google Cloud Platform, Cloud Build, automation</p>		

CONTENTS

LIST OF FIGURES.....	4
USED ABBREVIATIONS.....	5
1 INTRODUCTION.....	6
2 THEORY	7
2.1 Microservices architecture.....	7
2.2 Containerization	7
2.3 Kubernetes.....	10
2.3.1 Kubernetes components.....	11
2.3.2 Kubernetes objects.....	12
2.3.3 Workloads	14
2.4 CI/CD Pipelines.....	15
2.4.1 Application Deployment Strategies.....	15
2.5 Google Cloud Platform.....	17
2.5.1 Google Kubernetes Engine	18
2.5.2 Cloud Source Repository and Container Registry	19
2.5.3 Cloud Build.....	19
3 IMPLEMENTATION	20
3.1 Application Structure	21
3.2 Deploy to Kubernetes.....	23
3.3 Automation.....	29
3.3.1 Setting up repositories and environments	30
3.3.2 Build processes and triggers	30
3.3.3 Deployment to branches.....	35
3.3.4 Canary deployment to production.....	37
3.3.5 Deployment to the entire production environment with tags	38

4 RESULTS AND CONCLUSION	38
REFERENCES.....	40

LIST OF FIGURES

Figure 1: Virtualization.....	8
Figure 2: Containerization	9
Figure 3: Kubernetes.....	11
Figure 4: Example deployment.yaml file.....	13
Figure 5: Deployment strategies (Google 2021).....	16
Figure 6: GCP data centers as of 2021 (Google 2021)	17
Figure 7: GKE.....	18
Figure 8: Application deployment pipeline	21
Figure 9: Application structure.....	22
Figure 10: Frontend mode	23
Figure 11: Backend mode	23
Figure 12: Image built.....	24
Figure 13: frontend-production.yaml.....	25
Figure 14: backend-production.yaml	26
Figure 15: frontend.yaml.....	27
Figure 16: backend.yaml	27
Figure 17: Kubernetes system deployed	28
Figure 18: Returned webpage	29
Figure 19: Master branch	30
Figure 20: Build step	31
Figure 21: Publish step.....	32
Figure 22: Deploy step for a new development branch	32
Figure 23: Deploy step for canary deployment.....	33
Figure 24: Deploy step for live rollout.....	33
Figure 25: IAM role for Cloud Build service account.....	34
Figure 26: Build success	35
Figure 27: new-feature environment.....	36
Figure 28: Application v2.0.0	36

Figure 29: Running pods	37
Figure 30: Canary deployment serving new version.....	37
Figure 31: Live environment deployment.....	38

Used Abbreviations

CI/CD	Continuous Integration and Continuous Delivery/Deployment
GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
AWS	Amazon Web Services
SRE	Software Reliability Engineer
DevOps	Developer and Operator
VM	Virtual Machine
OS	Operating System
TLS/SSL	Transport Layer Security and Secure Socket Layer
IAM	Identity and Access Manager
SLA	Service Level Agreement

1 INTRODUCTION

In modern software development, it is clear that more and more companies are putting a larger emphasis on shorter programming cycles and faster delivery time. They are adopting more modular microservices architecture, pursuing an Agile focused development cycle or implementing automated CI/CD pipelines in their product delivery system. Along with that, the push for modern cloud hosting and open-source utilities only accelerates the modernizing process of software development as a whole. Many large corporations have stepped up and provided an entire fleet of services that can accommodate developers from start to end, such as Amazon with AWS, Microsoft with Azure or Google with Google Cloud Platform. In parallel, small companies are also very active in the market, providing high quality programs and services that would make the lives of developers, system administrators or the more modern SRE and DevOps engineers much simpler. Yet the plethora of options available sometimes confuses the average programmer rendering them unable to choose between similar products to use in their applications. This problem raises the importance of having a clear direction coordinating the use of external services in software development.

As a latecomer to the public cloud provider space, Google has been active and strategic in their growth and target customer space. Leveraging their experience in Kubernetes and coordinating large and reliable infrastructure systems, Google offers a Kubernetes managed service as a top-of-the-line selection for deploying microservices applications. Along with their obvious strength in Kubernetes, Google also offers plenty of developer tools as a service, raising the quality of life for customers opting for a full cloud native experience on Google Cloud Platform.

The main goal of this thesis is to offer a model automated deployment to the Google Kubernetes Engine, following a modern microservice oriented CI/CD architecture that would deploy the application to multiple environments suitable for testing and production right when code is merged to the program repository.

2 THEORY

This part of the thesis explains the technical theory behind the pipeline design and the services and applications used in order to achieve the goal of building a model automated delivery pipeline.

2.1 Microservices architecture

Originally, applications are put into one singular server entity that processes all the user requests and requests to the database as well as other functionalities. As the application and the development team grows, the fact that each change to the monolithic architecture would require the verification and authorization of all parties would significantly delay the development process. In addition, a small error could potentially risk paralyzing the entire production environment.

To combat this problem, development teams have adopted a more loosely coupled architecture that promotes the decoupling of different components within a monolithic server and turn them into collaborating modules called services (Richardson 2020). This adoption results in a system that consists of components that can be deployed independently, rolling out updates feature by feature and can roll back errors modularly, which helps in quickly addressing problems and returning the application to a stable state.

2.2 Containerization

Containerization is an increasingly popular method of designing and deploying Microservices applications. A container is a unit of software with the application code along with all the configuration files, libraries and dependencies packed into a sandbox that can run quickly and similarly across multiple different computing environments. Some definitive features of containers include being efficient, portable and secure. These factors help containers become a welcome alternative to the extensive use of Virtual Machines in modern application development. The research firm Gartner projects that by 2022, more than 75% of

global organizations will be running containerized applications in production, up from less than 30% in 2019 (Gartner, 2020).

The main difference between Containers and VMs is that each VM runs its own OS under the Hypervisor while Containers share the host OS and runs under the container engine layer. The differences are shown through figure 1 and 2.

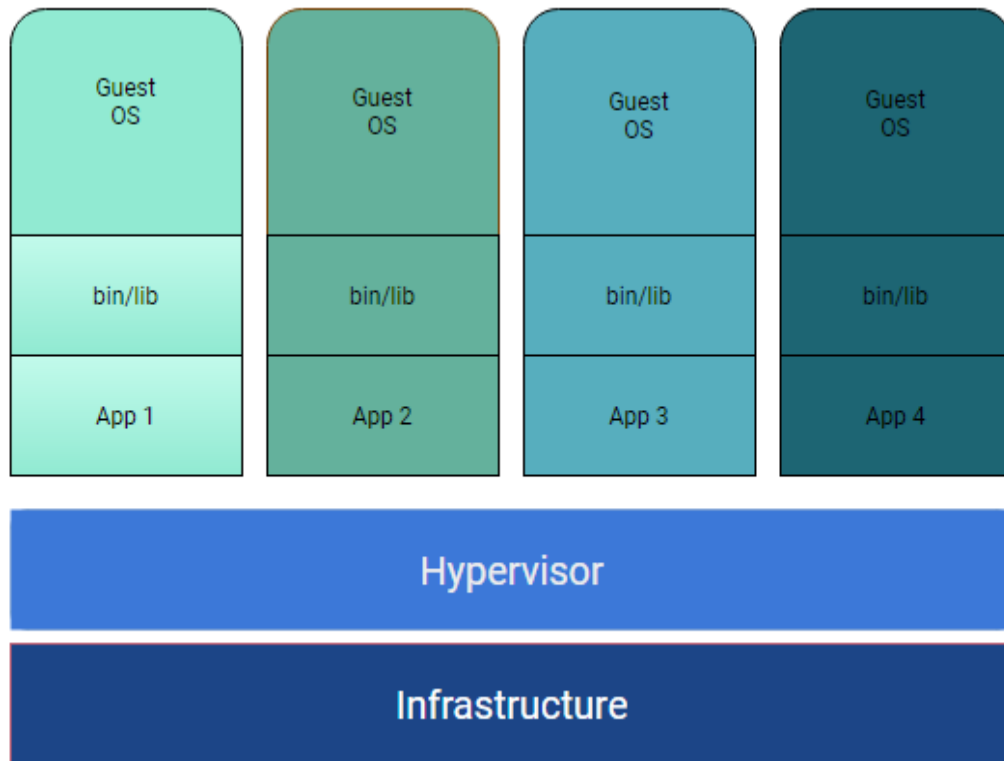


Figure 1: Virtualization

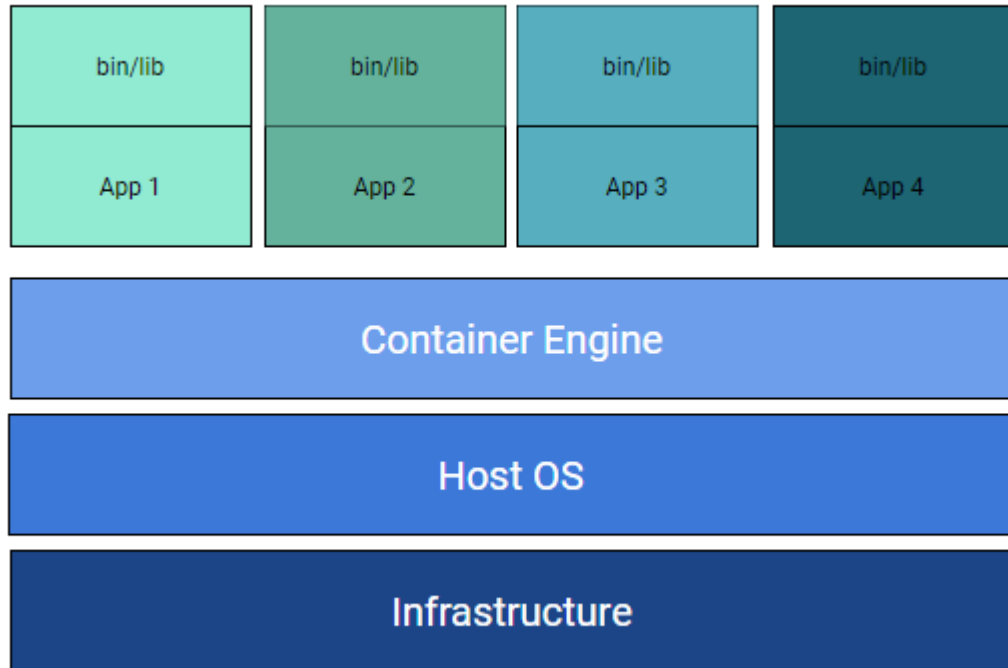


Figure 2: Containerization

Some benefits of containerization include:

- *Efficiency:* Applications running on the same containerized environment share the OS kernel. Furthermore, application layers within a container can be shared across multiple containers. Because they do not have to run a separate OS for each application, containers require much less startup time compared to VMs, therefore optimizing system performance and reducing licensing costs.
- *Portability:* A container creates a running environment that is independent from the host OS, therefore it can be run consistently across multiple machine environments and maintains consistency even when running. This feature allows some companies to run hybrid workloads across both cloud and on-premise environments through container orchestration platforms.

- *Security*: The isolation of containerized applications inherently prevents the invasion of malicious code from affecting other containers, as they run in separate environments. In addition, there are multiple security appliances that can be applied to containers to further fortify the security of application environments.

By offering the creation and operation of stable modules of small artifacts, containerization fits the description of a microservice architecture of decoupling applications into smaller size services. Containers serve as a perfect vessel for deploying such services on a large scale, and that is why Docker and Kubernetes are rising in popularity day by day.

2.3 Kubernetes

Since containerizing is such a great way to package and deploy applications, development teams and companies are using it more and more every day. However, this poses another problem. Containers are so efficient and consistently deployable that users would want to deploy multiple instances of them. Here lies a problem: if each container has to be configured manually for networking, storage and interaction with other components of the system, managing an increasingly high number of them would be unsustainable. In addition, there are some important external services such as autoscaling, OS image update, configuring TLS/SSL and so on. Managing all these container by container would be a very tall task. With this problem in mind, some engineers at Google started an open-sourced project called Kubernetes that aims to solve these issues and bring the idea of containerization closer to being a common reality. Since then, Kubernetes has grown into a highly scalable and reliable container orchestration platform that also provides a multitude of support services that greatly enhances the experience of using containers. According to “The State of Kubernetes 2020” report by VMWare, 59% of respondents were using Kubernetes in their production environment, and over 95% of respondents said that they see a clear benefit in Kubernetes adoption (VMWare 2020).

The following sections further explains some details about how Kubernetes work, based on

2.3.1 Kubernetes components

In order to manage so many additional services, Kubernetes introduces a few concepts to provide a centralized orchestration platform.

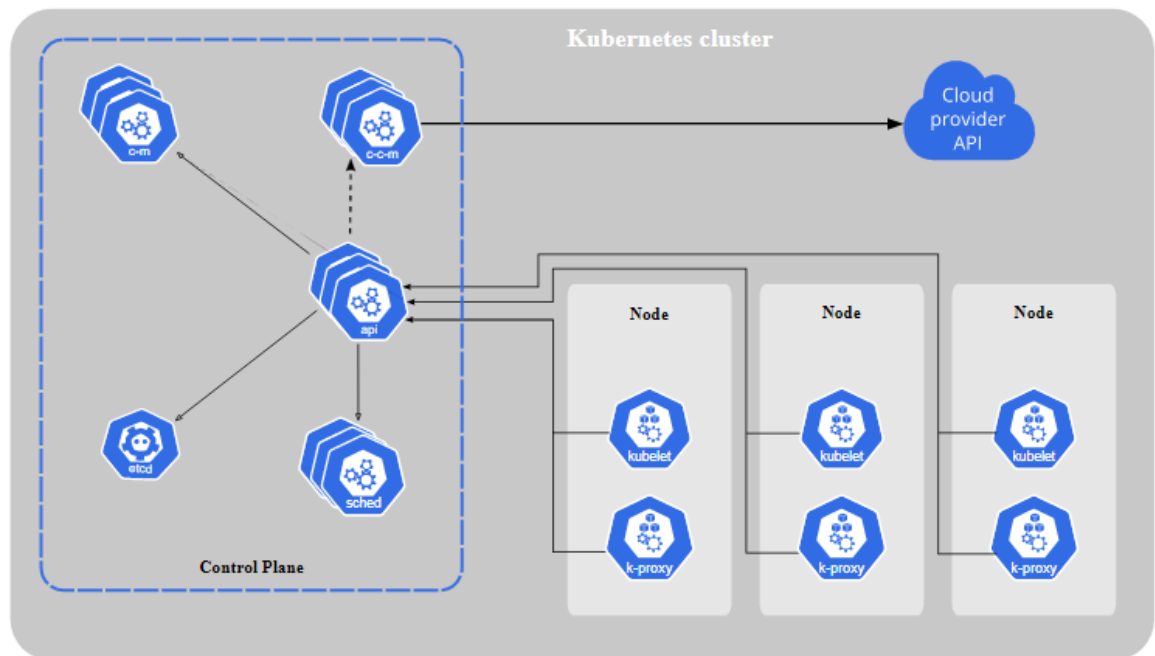


Figure 3: Kubernetes

In abstract, a Kubernetes Cluster is comprised of 2 main planes: Control plane (master nodes) and Worker nodes. The Control plane is in charge of deciding what happens over the entire cluster. It has the ability to schedule when to start a node, when to delete a node as well as keeping the cluster up to date with cluster events. A Kubernetes cluster is comprised of multiple small components both in the control plane and in the worker nodes, each in charge of a specific set of tasks that contributes to the orchestration and operation of the entire cluster

Control plane components include:

- *kube-apiserver*: This component works as a frontend to the Kubernetes API, providing an interface for all cluster components to interact with each other.
- *kube-controller-manager*: This component works as a background process that embeds the non-terminating loop that always regulates the state of the system. This is the part that defines the automation characteristic of Kubernetes, as it always checks and makes changes to the cluster state to be in sync with the desired state specified in the yaml files.
- *etcd*: Key-value storage for all cluster data.
- *kube-scheduler*: The scheduler oversees newly created pods and assigns which node to run them on.
- *cloud-controller-manager*: this is a special component used for integration with cloud providers' APIs.

Worker node components include:

- *kubelet*: This component is also known as the kube agent, which exists within a node to make sure that containers are running in a Pod.
- *kube-proxy*: This component is used to maintain network rules on nodes. With this network, pods within a cluster can reliably communicate with each other without having to use external networks.
- *Container Runtime*: This is the software that is responsible for running containers. Kubernetes currently supports Docker, containerd, CRI-O and any implementation of the Kubernetes CRI.

In addition to these crucial components, Kubernetes also provides some other features such as DNS, dashboard, cluster-level logging and monitoring for ease of management and monitoring.

2.3.2 Kubernetes objects

By definition, Kubernetes objects are persistent entities within the Kubernetes system. In Kubernetes, objects are defined within a yaml file and the Kubernetes system will constantly work to make sure that all the cluster state matches the desired state written in those yaml files.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

Figure 4: Example deployment.yaml file

As seen in figure 4, a yaml file is a declarative file that describes the desired state of an object in Kubernetes. Kubernetes will convert this format into JSON when making an API request. For a Kubernetes yaml file, there are some fields that are required in order for the object to be properly created:

- *apiVersion*: the version of the Kubernetes API that the user wants to use for this object.
- *kind*: which type of object does the user want to create.
- *metadata*: data that helps in identifying an object, such as name, uid, namespace and label.
- *spec*: the desired state of the object.

It is through the creation and maintenance of these objects that Kubernetes provision infrastructure and resource. In these object declaration files developers can choose the amount of computing resource allocated to a container, or how many containers to run for the job.

2.3.3 Workloads

A Workload is an application that runs on Kubernetes. With Kubernetes, users can run an application as one single component or multiple components that cooperate, but either way, users must run it inside a set of pods. Pods are the smallest unit of computing resource that users can assign in Kubernetes. It is a group of one or more containers with a set of common networking and storage resources. Everything inside a single Pod is always scheduled together and put in the same location. Workload resources manage a fleet of these pods as declared. Kubernetes provide several built-in workload resources:

- *Deployment and ReplicaSet*: These workloads are suitable for stateless applications since any pods within a Deployment are replaceable and a new one can easily take its place if needed.
- *StatefulSet*: This kind of workload is used for applications that do track the state of the machine somehow. If the workload needs to store persistent data, it can be connected to a PersistentVolume object for storage.
- *DaemonSet*: This type of workload provides node-local appliances. Once deployed, DaemonSet will make sure that all (or some) of the nodes have one copy of a Pod.
- *Job and CronJob*: Tasks that run to completion and then stop. Jobs are tasks that run only once and CronJobs are tasks that runs according to a schedule.

In addition to Kubernetes native workloads, developers can utilize additional third-party resources to further coordinate and manage their clusters. Some of the most used external tools include Helm for package management, Istio for service mesh management and Prometheus for monitoring. This is one of the true wonders of Kubernetes, that since it is open-sourced, everyone can join and contribute to the development of the entire ecosystem.

2.4 CI/CD Pipelines

The modern Software Development Cycle implements the Agile methodology or DevOps/SRE approach, which revolves around a continuous cycle of developing, testing and deploying the application in short sprints. In order for this process to be feasible, lots of companies implement a system called a CI/CD pipeline. A CI/CD pipeline introduces monitoring and automation to improve the process of application development, especially in the testing and deployment phases. Every part of the CI/CD pipeline can be done manually, but its true potential lies in the automation of the process.

Continuous Deployment refers to the pipeline from when code changes are merged and pushed to the repository to when the application is deployed to the servers and served to users. An automated continuous delivery pipeline provides developers with the ability to instantly react to events such as customer feedbacks or disasters. It also allows companies to test new ideas, deliver new features and rollback quickly.

2.4.1 Application Deployment Strategies

In order to reliably release new versions of an application, developers nowadays rely heavily on customer and system feedback to make sure that everything runs well before the final push to production. To achieve this, they would try to have some customers use the new version first, see if the server and the users have any feedback. There are multiple ways to go about this endeavor, called application deployment strategies.

Deployment or testing pattern	Zero downtime	Real production traffic testing	Releasing to users based on conditions	Rollback duration	Impact on hardware and cloud costs
Recreate Version 1 is terminated, and Version 2 is rolled out.	x	x	x	Fast but disruptive because of downtime	No extra setup required
Rolling update Version 2 is gradually rolled out and replaces Version 1.	✓	x	x	Slow	Can require extra setup for surge upgrades
Blue/green Version 2 is released alongside Version 1; the traffic is switched to Version 2 after it is tested.	✓	x	x	Instant	Need to maintain blue and green environments simultaneously
Canary Version 2 is released to a subset of users, followed by a full rollout.	✓	✓	x	Fast	No extra setup required
A/B Version 2 is released, under specific conditions, to a subset of users.	✓	✓	✓	Fast	No extra setup required
Shadow Version 2 receives real-world traffic without impacting user requests.	✓	✓	x	Does not apply	Need to maintain parallel environments in order to capture and replay user requests

Figure 5: Deployment strategies (Google 2021)

All the deployment strategies have their pros and cons, as seen in figure 5. For this specific project, I would want a deployment strategy that involves no downtime, with actual user traffic routed to the testing instances so that customers can test and give feedback on the newly deployed version. As seen from the chart above, A/B testing would be the perfect choice. However, A/B testing involves some additional trigger applied to the load balancer in order to serve to users based on specific conditions. Since the application I would be using does not include features that are specific to device types or user metadata, the benefits of using A/B testing compared to canary deployment would be nullified. Therefore, the canary deployment pattern would be chosen to be demonstrated in this project.

2.5 Google Cloud Platform

Google Cloud Platform is one of the Leaders for Cloud Infrastructure and Services in 2020 according to Gartner's report (Raja et al. 2020). With the third largest market share, GCP is proving itself to be a growing competitor to Microsoft's reliable Azure and Amazon's giant AWS. GCP provides a wide range of products, from traditional VMs and storage to managed services such as Google Kubernetes Engine, CloudSQL and so on. In addition, GCP also provides powerful cloud Big Data and data analytics platform with BigQuery. Google also wants to make GCP an ecosystem for developers with all open-sourced services like Kubernetes and the push for Anthos for hybrid deployment to Kubernetes for GKE. With such a strong base for deploying Kubernetes, I found it deploying a pipeline around Google Kubernetes Engine a highly simple yet effective and rewarding process for companies looking to develop a cloud native application.



Figure 6: GCP data centers as of 2021 (Google 2021)

Figure 6 shows the global scale of Google Cloud Platform and Google's plan to further expand its operation scope. Compared to other cloud providers, Google has already dedicated much resource into building its own network, connecting data centers with Google's own infrastructure to avoid having to utilize external internet traffic. This dedication shows their true willingness to enter the market

and potentially to be a large player, bringing more and more options that would benefit users and developers.

2.5.1 Google Kubernetes Engine

Google Kubernetes Engine is GKE's managed Kubernetes service on GCP. By managed service, Google means that the provision, creation and management of sufficient infrastructure would be Google's responsibility and customers only need to focus on the application and products that they would be running on top of the provisioned resources. Google makes sure that the resources provisioned would be guaranteed to function properly with an SLA for each managed service product. The GKE environment consists of multiple machines grouped together to form a cluster. Google Kubernetes Engine management scope

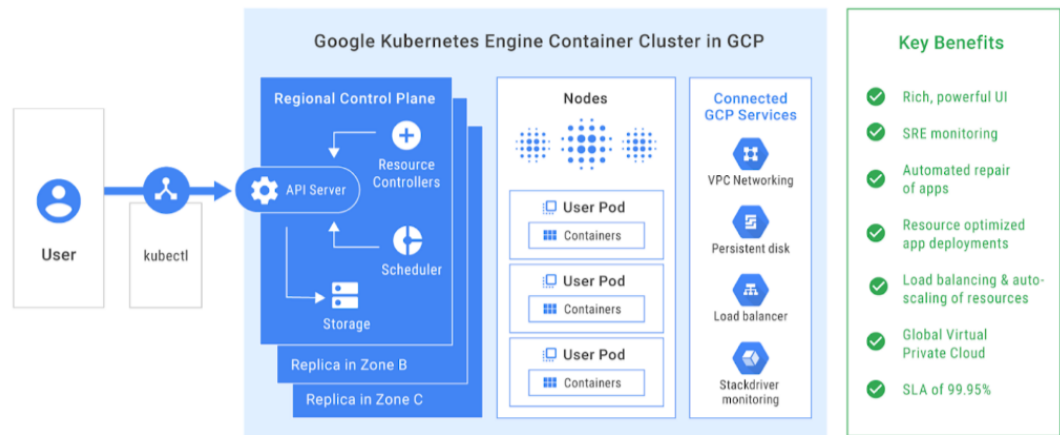


Figure 7: GKE

By offloading the management of the control plane to GKE, developers eliminate the high overhead of setting up management components like the scheduler or the api server, as well as the need for setting up utilities such as autoscaling, auto-healing and OS image management, to name a few. Previously the management overhead for running a Kubernetes cluster has been a significant problem. There are so many problems that needed to be addressed and so many detailed components that could raise a conflict. To name a few, problems might arise when creating master node replicas, bootstrapping a high availability etcd

storage cluster or configuring remote access with role-based access control. Furthermore, GKE has strong integration with other GCP services such as load balancing or centralized monitoring, which immensely simplify the operation and deployment process for companies.

2.5.2 Cloud Source Repository and Container Registry

Cloud Source Repository is Google's own version control platform hosted on Google Cloud Platform. CSR is a git platform, not too different from Github, Gitlab or Bitbucket. Each project on GCP has its own Cloud Source Repository and CSR can also integrate with both Github and Bitbucket. To Google, CSR is more like an extension to GCP rather than a direct opponent to Github or Bitbucket. Because it is a natural extension to GCP, CSR provides seamless integration with Cloud Build, App Engine, Cloud Pub/Sub and Cloud Monitoring and Logging to facilitate deployment automation to users' workflow. (Hajdarbegovic 2015)

Container Registry is GCP's repository for Container Images. More than just an Image repository, the Container Registry also provides additional services like build automation for Cloud Build or direct deployment to GKE, App Engine, Cloud Function or Firebase. Furthermore, Container Registry provides an in-depth vulnerability scanning service that checks the container images for all possible vulnerabilities available on Google's up-to-date database.

2.5.3 Cloud Build

Cloud Build is a service that executes the code builds on Google Cloud Platform's infrastructure. Cloud Build can import source code from multiple different repositories or even cloud storage spaces to produce artifacts such as Docker containers or Java archives.

In order to run Cloud Build, users can write a build config file to provide Cloud Build with instruction on what task to perform. Users can configure builds to fetch dependencies, run tests, analysis and create artifacts with tools such as docker, gradle, maven, bazel and gulp. Cloud Build executes this config as a series of

build steps, where each step is run inside a Docker container. Executing build step is largely similar to executing commands in a script.

In addition to writing build steps, Cloud Build and the Cloud Build community provides a set of supported open-source build steps as well as community-contributed build steps for everyone to use as directly or use as reference for their own file.

3 IMPLEMENTATION

This section will explain my implementation of an autonomous Continuous Delivery pipeline. The project is fairly simple. As shown in figure 8, there are two logical environments for code deployment, production and development. Whenever the developers push the code to a branch of the repository, the Cloud Build will build the container image and apply the container image to the cluster according to the branch of deployment. In addition, I would be deploying my application in different ways to show the flexibility of such a pipeline. For the main production environment, I would create the environments first and then deploy to it. I would also use canary deployment to roll out new versions of the application to show the possibility of having this done to a multitude of deployment strategies. For the development environment, I would create resources programmatically through the automated pipeline to show that Cloud Build can make CI/CD very flexible for development teams to automate everything as will.

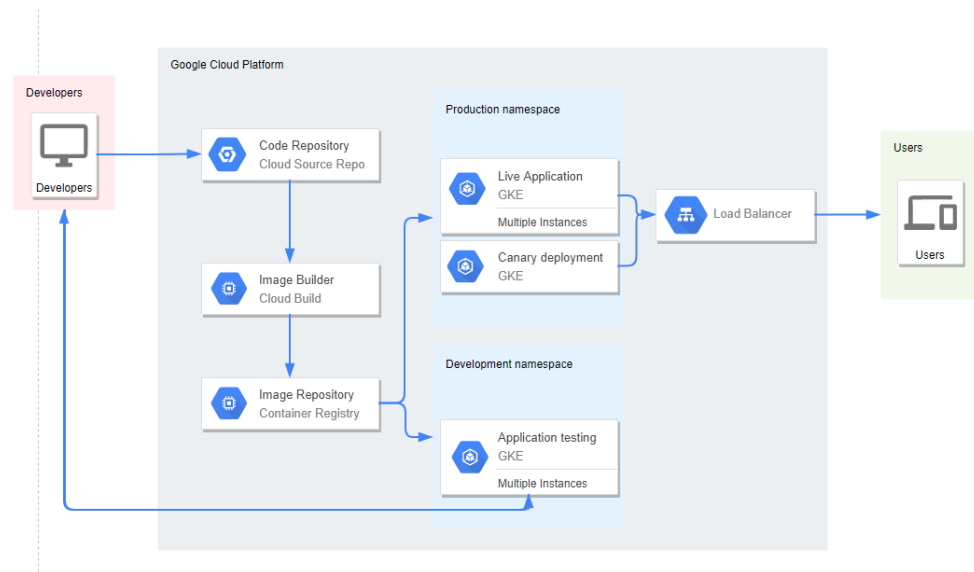


Figure 8: Application deployment pipeline

3.1 Application Structure

The application I would be using for the demo is an open-sourced application made by Google called gceme. This application would return the VM metadata of the Compute Engine instance that hosts the application. In addition, the application would also show its version number, so that users can track whether their code was deployed to the environments properly. Gceme is an application built specifically for the testing of a Deployment pipeline, and that is why I chose to run this application in this project.

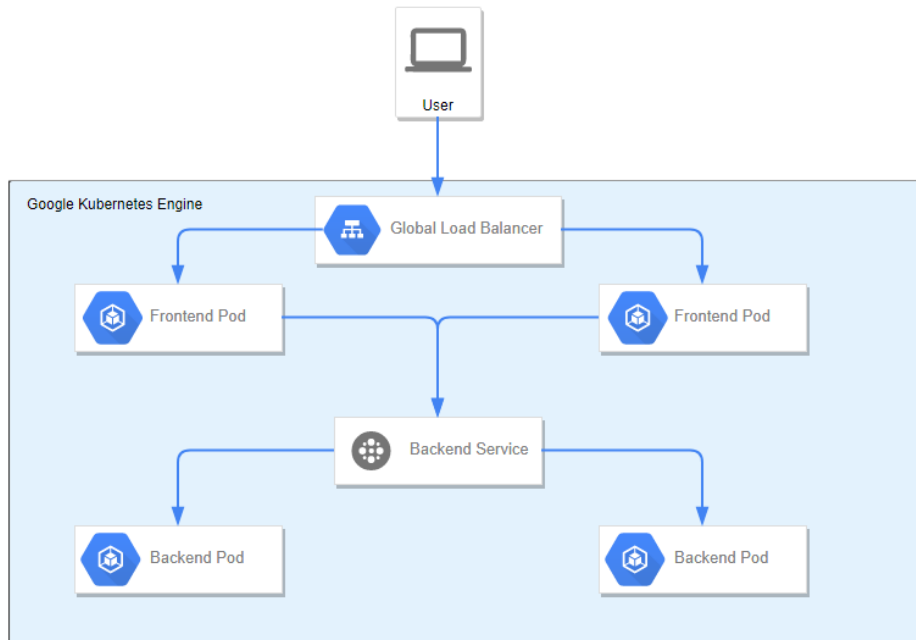


Figure 9: Application structure

As shown in figure 9, the application works as follows: When the user sends a request to the load balancer, it automatically sends the package to one of the frontends. Then, the frontend sends the request to the backend service which in turn sends the request to the backend pod. From there, the backend pod returns its Instance metadata to the user.

This single application has 2 modes of operation, frontend mode and backend mode. While creating the deployment, users can declare which mode of operation they would want to use.

```

func frontendMode(port int, backendURL string) {
    log.Println("Operating in frontend mode...")
    tpl := template.Must(template.New("out").Parse(html))

    transport := http.Transport{DisableKeepAlives: false}
    client := &http.Client{Transport: &transport}
    req, _ := http.NewRequest(
        "GET",
        backendURL,
        nil,
    )
    req.Close = false
}

```

Figure 10: Frontend mode

```

func backendMode(port int) {
    log.Println("Operating in backend mode...")
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        i := newInstance()
        raw, _ := httputil.DumpRequest(r, true)
        i.LBRequest = string(raw)
        resp, _ := json.Marshal(i)
        fmt.Fprintf(w, "%s", resp)
    })
    http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
    })
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%v", port), nil))
}

```

Figure 11: Backend mode

With the application components ready for deployment, the next step is to build a first Image and deploy the application to a GKE cluster.

3.2 Deploy to Kubernetes

First, in order to store my Container Image for the cluster to pull from, I created a Container Register repository called on my GCP project. Afterwards, I proceeded to build and push the container image version 1.0.0 to the image repository as shown in the figure below.

```
anhdang@cloudshell:~/anhdang-thesis (protean-unity-305602) $ docker build -t gcr.io/$PROJECT/gceme:1.0.0 .
Sending build context to Docker daemon 203.3kB
Step 1/1 : FROM golang:onbuild
onbuild: Pulling from library/golang
ad74af05f5a2: Pull complete
2b032b8bbe8b: Pull complete
a9a5b35f6ead: Pull complete
25d9840c55bc: Pull complete
d792ec7d64a3: Pull complete
be556a93c22e: Pull complete
3a5fce283ale: Pull complete
0621865a0c2e: Pull complete
Digest: sha256:c0ec19d49014d604e4f62266afd490016b11ceec103f0b7ef44875801ef93f36
Status: Downloaded newer image for golang:onbuild
```

Figure 12: Image built

For this application, the GKE cluster is split into 4 main parts. The first part is the Load Balancer service that exposes the Frontend pods to the Internet. The second part is the deployment for Frontend pods. For these pods, I first created the deployment without declaring the desired number of pods and then I enabled autoscaling for the frontend pods. With this architecture, we could fully leverage the use of a load balancer, automatically splitting traffic to all pods within the deployment. The third part of the cluster is the Backend services, which acts as a proxy between the Frontend pods and Backend pods. The following figures show the yaml files for deploying the Kubernetes cluster. In the two deployment.yaml files, focus on the spec/spec/containers part where the specification for the containers is declared. The difference between frontend and backend pods are in the command part where I inserted the bash script commands to choose to deploy the application as either the backend or the frontend pod to serve traffic. In the other two yaml files, we only need to focus on the type of service that they would be creating, which is a load balancer and a service proxy called ClusterIP to connect frontend services with backend pods.


```

kind: Deployment
apiVersion: apps/v1
metadata:
  name: gceme-frontend-production
spec:
  replicas:
  selector:
    matchLabels:
      app: gceme
      role: frontend
      env: production
  template:
    metadata:
      name: frontend
      labels:
        app: gceme
        role: frontend
        env: production
    spec:
      containers:
      - name: frontend
        image: gcr.io/protéan-unity-305602/gceme:1.0.0
        resources:
          limits:
            memory: "500Mi"
            cpu: "100m"
        imagePullPolicy: Always
        readinessProbe:
          httpGet:
            path: /healthz
            port: 80
        command: ["sh", "-c", "app -frontend=true -backend-service=http://gceme-backend:8080 -port=80"]
        ports:
        - name: frontend
          containerPort: 80

```

Figure 13: frontend-production.yaml

About the Kubernetes yaml files, first it must define the type of object that this file is about, in case of figure 13, it is a deployment yaml file. Next is the apiVersion for Kubernetes, which normally is version one. After that is the specification for the entire deployment, which has the template for the pods used. In the template part, the file defines containers that it would deploy, what image that container would run and other commands to operate that container.

The frontend pod will send traffic towards the backend service to port 8080 through port 80 as defined in the spec/template/spec/container/command part of the yaml file in figure 13.

```

kind: Deployment
apiVersion: apps/v1
metadata:
  name: gceme-backend-production
spec:
  selector:
    matchLabels:
      app: gceme
      role: backend
      env: production
  replicas: 1
  template:
    metadata:
      name: backend
      labels:
        app: gceme
        role: backend
        env: production
    spec:
      containers:
        - name: backend
          image: gcr.io/protean-unity-305602/gceme:1.0.0
          resources:
            limits:
              memory: "500Mi"
              cpu: "100m"
            imagePullPolicy: Always
          readinessProbe:
            httpGet:
              path: /healthz
              port: 8080
          command: ["sh", "-c", "app -port=8080"]
          ports:
            - name: backend
              containerPort: 8080

```

Figure 14: backend-production.yaml

The backend port will open port 8080 to wait for service calls from frontend as seen in figure 14.

```
kind: Service
apiVersion: v1
metadata:
  name: gceme-frontend
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: gceme
    role: frontend
```

Figure 15: frontend.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: gceme-backend
spec:
  ports:
  - name: http
    port: 8080
    targetPort: 8080
    protocol: TCP
  selector:
    role: backend
    app: gceme
```

Figure 16: backend.yaml

In figure 15 and 16, services such as load balancers and cluster IP services are also declared as object to be created with specifications on which port to open and which pods to select as their backend.

```

anh dang@cloudshell:~/anh dang-thesis (protean-unity-305602)$ kubectl get all -n production
NAME                                READY   STATUS    RESTARTS   AGE
pod/gceme-backend-canary-6ff5dc6cc5-p5xsh   1/1     Running   0           2m29s
pod/gceme-backend-production-7b8d85d95b-h4hrs  1/1     Running   0           2m52s
pod/gceme-frontend-canary-74569bb4f9-8lbtk    1/1     Running   0           2m29s
pod/gceme-frontend-production-74c984ff46-75cjj  1/1     Running   0           119s
pod/gceme-frontend-production-74c984ff46-7xsmf  1/1     Running   0           119s
pod/gceme-frontend-production-74c984ff46-92kdc  1/1     Running   0           2m52s
pod/gceme-frontend-production-74c984ff46-jhwmx  1/1     Running   0           119s

NAME                                TYPE                CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
service/gceme-backend                ClusterIP           10.3.255.240 <none>        8080/TCP       2m20s
service/gceme-frontend                LoadBalancer       10.3.241.192 34.80.217.174 80:30038/TCP   2m20s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/gceme-backend-canary  1/1     1             1           2m29s
deployment.apps/gceme-backend-production  1/1     1             1           2m52s
deployment.apps/gceme-frontend-canary    1/1     1             1           2m29s
deployment.apps/gceme-frontend-production  4/4     4             4           2m52s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/gceme-backend-canary-6ff5dc6cc5  1         1         1       2m29s
replicaset.apps/gceme-backend-production-7b8d85d95b  1         1         1       2m52s
replicaset.apps/gceme-frontend-canary-74569bb4f9  1         1         1       2m29s
replicaset.apps/gceme-frontend-production-74c984ff46  4         4         4       2m52s

```

Figure 17: Kubernetes system deployed

In addition to the production environment with four active frontend pod, one instance of frontend and backend canary deployment was also created to serve 20% of users with the canary version of the application.

We can see from figure 17 that we have a Load Balancer service for the frontend that is exposing our application to the internet through the external IP address 34.80.217.174 and a ClusterIP service that is exposing our backend to our frontend through the TCP port 8080.

With the Kubernetes system deployed, I could access the external IP address of the frontend Load Balancer. The web page is served as in figure 18, whereas it shows the information of the Compute Engine instance that the backend pod was running on. In addition, it shows that the application version that was running was version 1.0.0. This is important since I would later deploy a different version of the application and it would show that the application was deployed as desired.

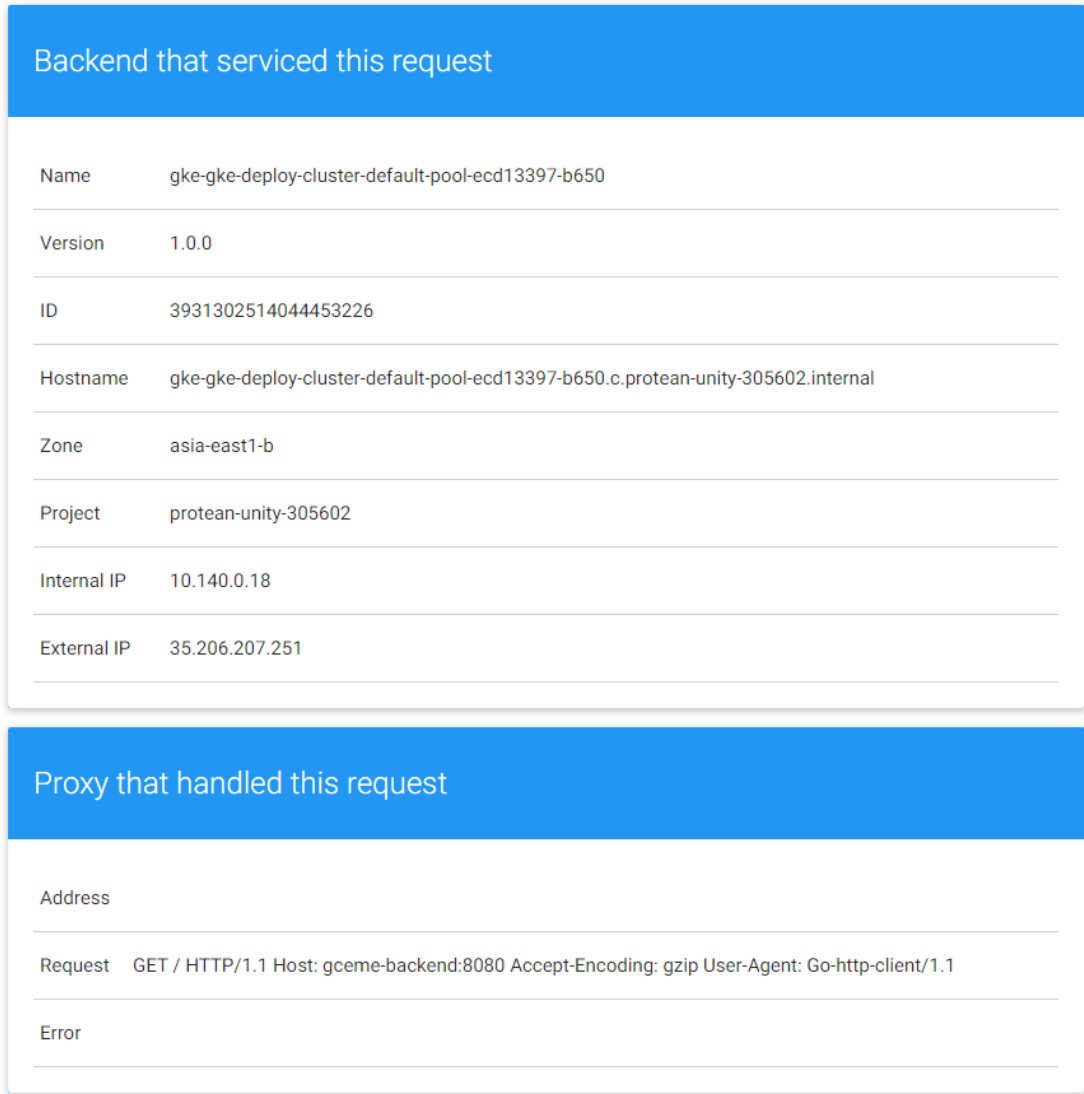


Figure 18: Returned webpage

As I have had the application up and running as desired, the next step is to prepare the environment for automation on Cloud Build.

3.3 Automation

In this section, set up the automation pipeline on Cloud Build to automate the deployment process to Kubernetes, from when the code was pushed to the repository to when the system serves the requests to end-users.

3.3.1 Setting up repositories and environments

I began this step by creating a repository on Cloud Source Repository called default. Then, I initialized git on my local application file and set the default repository I just created as remote. After that, I pushed the code to the repository master branch for the initial commit. Now, the codebase is available on the repository and ready for setting up a build pipeline.

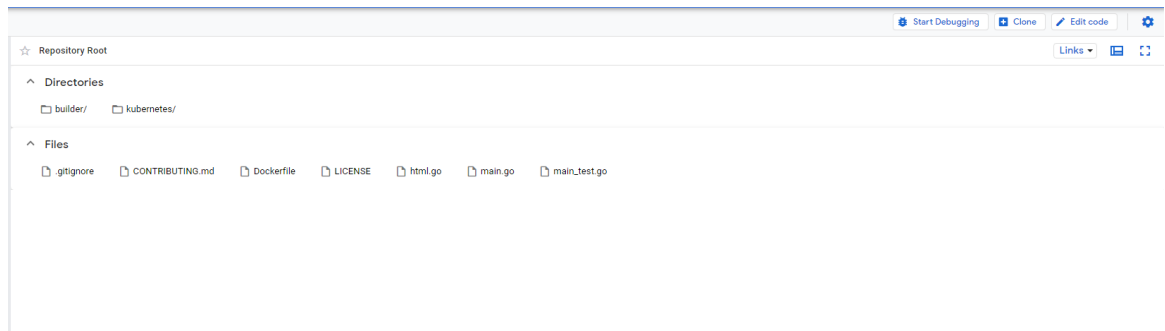


Figure 19: Master branch

The master branch on the repository contains the files on figure 19. The main application file is main.go and from there I would update the code version. From the figure, we could also see I made the kubernetes/ directory for storing yaml files regarding the GKE cluster, while the builder/directory is for storing yaml files to run the Cloud Build pipeline.

3.3.2 Build processes and triggers

Cloud Build has a system to listen to events from other places, which is called triggers. Here, I created three triggers in place so that in the event of codes being pushed to different branches of the code repository, the pipeline will automatically build and deploy accordingly to appropriate environments.

One trigger would be for when there is a code pushed to a new branch other than the master branch. Codes of this nature are normally pushed to run on the Development environment, so my trigger is to create the Development environment with the application from the new branch created.

Another trigger is for deploying to the deployment environment itself. However, since the pipeline uses a canary deployment rollout scheme, the build process only deploys the application to canary deployments and serves a small portion of traffic to users for live testing purposes.

The last trigger runs when the code was merged directly to the master branch, effectively announcing that the applied version should be the live version served to all customers. Cloud Build will perform accordingly and apply patches to all deployment namespace pods and serve the newest live version.

The following figures are all from the build.yaml files to deploy from different branches to different parts of the cluster.

```
### Build
- id: 'build'
  name: 'gcr.io/cloud-builders/docker'
  entrypoint: 'bash'
  args:
    - '-c'
    - |
      docker build -t gcr.io/$PROJECT_ID/gceme:$TAG_NAME .
```

Figure 20: Build step

Figure 20 describes the Build step for the process. Here, the yaml file runs a bash script that runs the command *docker build -t* to build the docker image and tag it with a name.

```

### Publish
- id: 'publish'
  name: 'gcr.io/cloud-builders/docker'
  entrypoint: 'bash'
  args:
    - '-c'
    - |
      docker push gcr.io/$PROJECT_ID/gceme:$TAG_NAME

```

Figure 21: Publish step

Figure 21 describes the publishing process to push the image to cloud source repository.

```

### Deploy
- id: 'deploy'
  name: 'gcr.io/cloud-builders/gcloud'
  env:
    - 'CLOUDSDK_COMPUTE_ZONE=${_CLOUDSDK_COMPUTE_ZONE}'
    - 'CLOUDSDK_CONTAINER_CLUSTER=${_CLOUDSDK_CONTAINER_CLUSTER}'
    - 'KUBECONFIG=/kube/config'
  entrypoint: 'bash'
  args:
    - '-c'
    - |
      CLUSTER=$(gcloud config get-value container/cluster)
      PROJECT=$(gcloud config get-value core/project)
      ZONE=$(gcloud config get-value compute/zone)

      gcloud container clusters get-credentials "${CLUSTER}" \
        --project "${PROJECT}" \
        --zone "${ZONE}"

      sed -i 's|gcr.io/protan-unity-305602/gceme:.*|gcr.io/$PROJECT_ID/gcme:${BRANCH_NAME}-${SHORT_SHA}|' ./kubernetes/deployments/dev/*.yaml

      kubectl get ns ${BRANCH_NAME} || kubectl create ns ${BRANCH_NAME}
      kubectl apply --namespace ${BRANCH_NAME} --recursive -f kubernetes/deployments/dev
      kubectl apply --namespace ${BRANCH_NAME} --recursive -f kubernetes/services

```

Figure 22: Deploy step for a new development branch


```

### Deploy
- id: 'deploy'
  name: 'gcr.io/cloud-builders/gcloud'
  env:
    - 'CLOUDSDK_COMPUTE_ZONE=${_CLOUDSDK_COMPUTE_ZONE}'
    - 'CLOUDSDK_CONTAINER_CLUSTER=${_CLOUDSDK_CONTAINER_CLUSTER}'
    - 'KUBECONFIG=/kube/config'
  entrypoint: 'bash'
  args:
    - '-c'
    - |
      CLUSTER=$(gcloud config get-value container/cluster)
      PROJECT=$(gcloud config get-value core/project)
      ZONE=$(gcloud config get-value compute/zone)

      gcloud container clusters get-credentials "${CLUSTER}" \
        --project "${PROJECT}" \
        --zone "${ZONE}"

      sed -i 's|gcr.io/protean-unity-305602/gceme:.*|gcr.io/$PROJECT_ID/gceme:${BRANCH_NAME}-${SHORT_SHA}|' ./kubernetes/deployments/dev/*.yaml

      kubectl get ns ${BRANCH_NAME} || kubectl create ns ${BRANCH_NAME}
      kubectl apply --namespace ${BRANCH_NAME} --recursive -f kubernetes/deployments/dev
      kubectl apply --namespace ${BRANCH_NAME} --recursive -f kubernetes/services

```

Figure 23: Deploy step for canary deployment

```

### Deploy
- id: 'deploy'
  name: 'gcr.io/cloud-builders/gcloud'
  env:
    - 'CLOUDSDK_COMPUTE_ZONE=${_CLOUDSDK_COMPUTE_ZONE}'
    - 'CLOUDSDK_CONTAINER_CLUSTER=${_CLOUDSDK_CONTAINER_CLUSTER}'
    - 'KUBECONFIG=/kube/config'
  entrypoint: 'bash'
  args:
    - '-c'
    - |
      CLUSTER=$(gcloud config get-value container/cluster)
      PROJECT=$(gcloud config get-value core/project)
      ZONE=$(gcloud config get-value compute/zone)

      gcloud container clusters get-credentials "${CLUSTER}" \
        --project "${PROJECT}" \
        --zone "${ZONE}"

      sed -i 's|gcr.io/protean-unity-305602/gceme:.*|gcr.io/$PROJECT_ID/gceme:$TAG_NAME|' ./kubernetes/deployments/prod/*.yaml

      kubectl get ns production || kubectl create ns production
      kubectl apply --namespace production --recursive -f kubernetes/deployments/prod
      kubectl apply --namespace production --recursive -f kubernetes/services

```

Figure 24: Deploy step for live rollout

In figure 22, 23 and 24 are deployment steps for different ways to deploy. All three begin by getting the environment variable for the cluster name, project name and running zone to operate on.

Next, they get the credentials for the Kubernetes cluster in order to be allowed to operate on the cluster. However, the Cloud Build API itself has to be allowed through Cloud Identity and Access Management system by a service account with the appropriate roles and permissions to interact with GKE and its clusters.

Member	Project
598529456617@cloudbuild.gserviceaccount.com	My First Project
Role Cloud Build Service Accou... ▼ Can perform builds	Condition Add condition
Role Kubernetes Engine Develo... ▼ Full access to Kubernetes API objects inside Kubernetes Clusters.	Condition Add condition

[+ ADD ANOTHER ROLE](#)

Figure 25: IAM role for Cloud Build service account

There is an interesting command in this deployment phase, which is the `sed -i` command. With this, whenever a new image is created, Cloud Build will replace the image repository address of the previously live image with newly built container image. There is also another way to do this, which is to let the deployment pull the latest image, so that whenever a new image is pushed to the Container Registry, the cluster will automatically update as well. However, letting Kubernetes always pull the latest image is not always a good thing, since there could be a mistake in the code pushed. Therefore, a substitution command to the `deployment.yaml` file should do the trick.

From there, the rest of the deployment files are commands directly to the Kubernetes cluster to deploy our code to the appropriate environment. The development branch creates a new namespace and creates an environment there, the canary deployment rolls out canary deployment and split traffic and the live tag deployment deploys the tags to the entire production environment.

✓ Successful: 01a14db8

Started on May 13, 2021, 3:00:13 PM

Steps	Duration
✓ Build Summary 3 Steps	00:01:28
✓ 0: build bash -c docker build -t gcr.io/protean-unity-305602/gcme:ne...	00:00:31
✓ 1: publish bash -c docker push gcr.io/protean-unity-305602/gcme:new-...	00:00:36
✓ 2: deploy bash -c CLUSTER=\$(gcloud config get-value container/cluste...	00:00:13

Figure 26: Build success

The figure 26 above shows that our Build step is complete, and the application is rolled out accordingly.

3.3.3 Deployment to branches

With everything setup, it was time to run and test the whole deployment pipeline. Here, developers push the code to a new branch on the repository. Cloud Build will be triggered to create a new environment on Kubernetes with the same name as the new branch

For the development environment, I push the code with version 2.0.0 and changed the application color scheme to a new branch called “new-feature”. Since this branch did not exist before and needs an environment to run on, Cloud Build creates a new namespace for developers to test out the application. The information about the namespace is in figure 27 below.

```

anh dang@cloudshell:~/anh dang-thesis (protean-unity-305602)$ kubectl get all -n new-feature
NAME                                READY   STATUS    RESTARTS   AGE
pod/gceme-backend-dev-568f7bb58c-p7r4q  1/1     Running   0           44s
pod/gceme-frontend-dev-7f75fcc948-wxxwj  1/1     Running   0           43s

NAME                                TYPE                CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
service/gceme-backend               ClusterIP           10.3.251.158  <none>         8080/TCP         41s
service/gceme-frontend               LoadBalancer       10.3.241.48   35.194.231.68  80:32518/TCP    40s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/gceme-backend-dev    1/1     1             1           44s
deployment.apps/gceme-frontend-dev    1/1     1             1           43s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/gceme-backend-dev-568f7bb58c  1         1         1       44s
replicaset.apps/gceme-frontend-dev-7f75fcc948  1         1         1       43s

```

Figure 27: new-feature environment

Backend that serviced this request

Name	gke-gke-deploy-cluster-default-pool-4fb85b45-2t6j
Version	2.0.0
ID	9174091039828198932
Hostname	gke-gke-deploy-cluster-default-pool-4fb85b45-2t6j.c.protean-unity-305602.internal
Zone	asia-east1-b
Project	protean-unity-305602
Internal IP	10.140.0.25
External IP	35.206.246.226

Proxy that handled this request

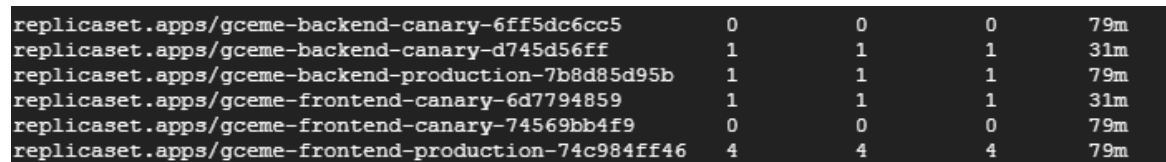
Address	
Request	GET / HTTP/1.1 Host: 10.3.251.158:8080 Accept-Encoding: gzip User-Agent: Go-http-client/1.1
Error	

Figure 28: Application v2.0.0

From figure 28, we could see that the color for the application was changed along with the version number. This shows that the environment is updated and ready for developers to test and work on.

3.3.4 Canary deployment to production

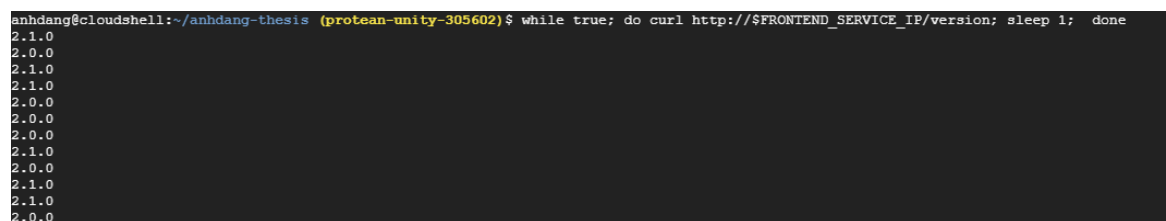
After the developing phase, when everything is ready for release, the application is merged to the master branch. However, this does not mean that it was ready for a full release. For the deployment model of this project, the application has to go through a phase of canary live testing to get user feedback. In this phase, some pods will run a version of the application for testing. The trigger in place for this process deploys the code merged to master branch of the repository to the production environment.



replicaset.apps/gceme-backend-canary-6ff5dc6cc5	0	0	0	79m
replicaset.apps/gceme-backend-canary-d745d56ff	1	1	1	31m
replicaset.apps/gceme-backend-production-7b8d85d95b	1	1	1	79m
replicaset.apps/gceme-frontend-canary-6d7794859	1	1	1	31m
replicaset.apps/gceme-frontend-canary-74569bb4f9	0	0	0	79m
replicaset.apps/gceme-frontend-production-74c984ff46	4	4	4	79m

Figure 29: Running pods

As seen in figure 29, 4 backend pods were running the live version and one backend canary pod was running the new version for testing.



```
anh dang@cloudshell:~/anh dang-thesis (protean-unity-305602)$ while true; do curl http://$FRONTEND_SERVICE_IP/version; sleep 1; done
2.1.0
2.0.0
2.1.0
2.1.0
2.0.0
2.0.0
2.0.0
2.0.0
2.1.0
2.0.0
2.1.0
2.1.0
2.1.0
2.0.0
```

Figure 30: Canary deployment serving new version

In figure 30, I ran a command to continuously request the application to return the value of the version that the pod serving the request was running. As we can see, some of the pods were running a newer version while most were running the old 2.0.0 version.

Furthermore, not all applications would have such a simple frontend and backend architecture, so a thorough analysis of each application is required in order to be able to design a working pipeline for automation. For example, compared to the production application that this implementation was based on, the Kubernetes cluster design was significantly different and therefore the Cloud Build trigger had to be altered accordingly. There are also additional trending utilities for a CI/CD pipeline that involve new models such as DevOpsSec, the alignment of all development, operation and security to the process. This involves newer concepts such as Compliance as code and has services to run such as Chef Inspec.

References

Richardson, C. 2020. Pattern: Microservices architectures. WWW document. Available at <https://microservices.io/patterns/microservices.html> [Accessed 16 April 2021].

VMWare. 2020. The state of Kubernetes 2020. WWW document. Available at <https://k8s.vmware.com/state-of-kubernetes-2020/> [Accessed 10 May 2021].

The Kubernetes Author. 2020. Kubernetes concepts. WWW document. Available at <https://kubernetes.io/docs/concepts/> [Accessed 15 May 2021].

Google. 2021. Application deployment and testing strategies. WWW document. Available at <https://cloud.google.com/architecture/application-deployment-and-testing-strategies> [Accessed 16 May 2021].

Hajdarbegovic, N. 2015. Google Cloud Source Repository vs. Bitbucket vs. Github: a worthy alternative? WWW document. Available at <https://www.toptal.com/git/google-cloud-source-repositories-vs-github-a-worthy-alternative> [Accessed 2 May 2021]