A Scalable Microservices-based Web Application in a Public Cloud

Capstone Project

Management and Administration of IT Infrastrutures and Services

Team nr.: 39

87704: Samuel Vicente

86392: Bruno Dias

92510: Lúcia Silva

Information Systems and Computer Engineering

IST-ALAMEDA

2021/2022

Contents

List of Tables

List of Figures

Acronyms

GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
GCE	Google Compute Engine
K8S	Kubernetes
IaC	Infrastructure as Code
GAR	Google Artifact Registry
MS	Micro Services
IP	Internet Protocol
VPC	Virtual Private Cloud
CI	Continuous Integration
CD	Continuous Deployment
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
API	Application Programming Interface

Chapter 1

Introduction

This project describes the process of deploying and provisioning a micro services based containerized web application in a public cloud provider, in our case, Google Cloud Platform (GCP), using Infrastructure as Code (IaC) tool Terraform.

The application to be deployed, Kubernetes Starterkit, is a simple browser-based calculator, using multiple distinct technologies, with a simple frontend and three back-end services that provide the calculator operations (addition, multiplication, etc) and the executed operations history:

- Vuecalc service (Vue.js);
- Expressed service (Express.js);
- Happy service (Hapi.js);
- Bootstorage service (Java Spring Boot);

Vuecalc is a simple calculator UI frontend application that uses the endpoints exposed by the Expressed, Happy and Bootstorage backend applications. Happy provides multiplication and division capabilities while Expressed provides addition and subtraction, both these services send the operations they execute to the bootstorage service in order to save the history of operations, this history can then be observed in the Vuecalc application. Bootstorage is a simple service whose sole purpose is to store and return the operations executed in a Redis Data Store.

Since the application was containerized, it made sense to use Kubernetes (K8S) for automatic deployment and management of the application containers. Google Kuber-

netes Engine (GKE) was used to create the cluster and Google Artifact Registry (GAR) to manage and store container images.

A video over viewing the contents of this report is available here.

Chapter 2

Methodology

This chapter describes the architecture of the implementation as well as each of its components.

2.1 Architecture

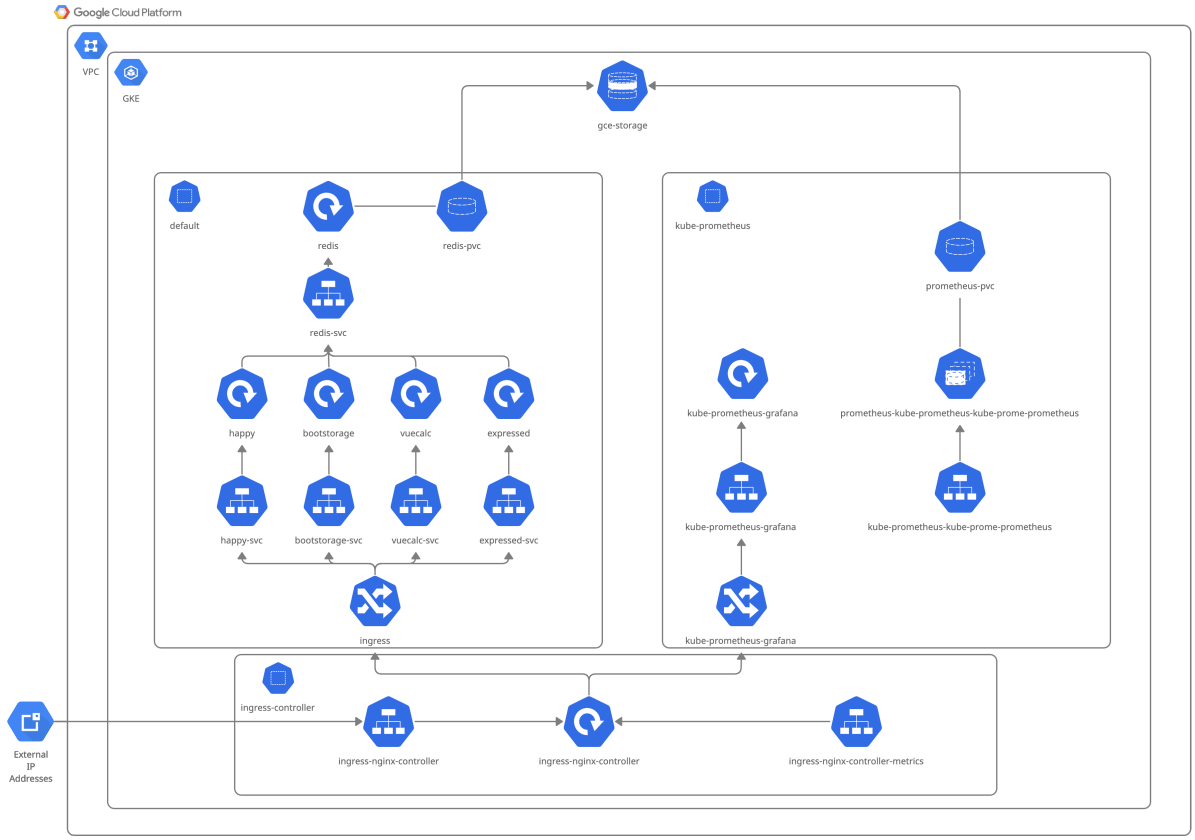

Figure 2.1: Solution architecture

We chose to build our cluster using GKE, this greatly simplifies the creation of the cluster. We deployed the cluster inside a Virtual Private Cloud (VPC).

We then defined a StorageClass to use with the applications that need persistent storage, in our case, Redis and Prometheus.

All our services are cluster internal, meaning they do not have external Internet Protocol (IP) addresses, this is by choice as we deemed better to use a reverse proxy to forward the requests from outside to inside the cluster. To accomplish this task we used the NGINX Ingress Controller. This serves as both the Application Programming Interface (API) gateway and reverse proxy. We defined two ingresses, one for each namespace that needed it, the default namespace and the kube-prometheus namespace. All the services inside the default namespace apart from the redis-svc are mapped to an API path in ingress. In the kube-prometheus namespace only the Grafana service is mapped to an API path in ingress .

This diagram is a simplification of the actual component graph of our cluster, the kube-prometheus namespace, since it was deployed using a helm chart, deploys a lot of services and other components not pictured here.

2.2 Components

In this subsection we describe the main parts of the solution such as the K8S cluster itself, the alternative tool to dockerhub and all the deployments and services used, as well as the monitoring tools.

2.2.1 K8S Cluster

K8S is an open-source container-orchestration system that automates the deployment, scaling, and administration of computer applications.

2.2.2 Artifact Registry

Similarly to Docker Hub, GAR allows the creation of a repository to manage and store docker images that can be deployed into the K8S cluster. When planning the overall

solution it made sense for the docker images to share the same project as the infrastructure, so that it has direct access and in future works a pipeline to automate the process including the build and containerization of the images could be implemented using tools provided by google such as Google Cloud Build, easing and automating the all process, providing a tool that allows Continuous Integration (CI) and Continuous Deployment (CD) of the application, possibly the infrastructure itself.

2.2.3 Application Deployments and Services

Our application is composed by a set of deployments.

A deployment is used to tell K8S how to create or modify instances of the pods that hold a containerized application.

A pod is a group of one or more containers, that share storage/network resources, and a specification on how to run them.

A service is a logical abstraction for a deployed group of pods in a cluster. Since pods are ephemeral, a service enables a group of pods, that provide the same functionality, to be assigned a name and unique IP address. This way, since pods and its IP are not permanent, the service serves as the abstraction of the group that is referenced by the service name, so that each time that a pod is replaced, it is still correctly referenced.

The services used for the correct functioning of the application are those listed below:

- **Vuecalc**:Allows interaction with the calculator functionalities.
- **Expressed**:Performs addition and subtraction operations.
- **Happy**:Performs division and multiplication operations.
- **Bootstorage**:Retrieves the data and operations performed upon the calculator.
- **Redis**:Serves as the data storage for the bootstorage service.

2.2.4 Ingress

The ingress is the entity responsible for managing the access to the cluster from the outside, exposing a single IP for accessing the application services. Ingress has the ability to forward IP packets, from the client requests to the services requested. The following table specifies the routing for our cluster ingress:

Table 2.1: Routing Rules

Path	Service Name	Service Port
/	Vuecalc	2000
/api/express	Expressed	3000
/api/happy	Happy	4000
/api/bootstorage	Bootstorage	5000
/grafana	Grafana	80

2.2.5 Monitoring

For the purpose of monitoring the cluster, kube-prometheus was used, this is a collection of Grafana dashboards and Prometheus rules combined with documentation and script, in order to provide easy to operate end-to-end K8S cluster monitoring with Prometheus using the Prometheus Operator.

The Prometheus Operator includes a Custom Resource Definition that allows the definition of the ServiceMonitor, used to define in which applications, by specifying K8S labels to each of the services, metrics shall be retrieved from within Kubernetes. The controller builds the required prometheus configurations. Applications and services expose a HTTP endpoint that contains Prometheus formatted metrics, which Prometheus will retrieve.

With this, real-time metrics can be stored by Prometheus and afterwards visualized using Grafana in charts.

Chapter 3

Implementation

This chapter describes the project files, pre-requisites for deployment and the deployment procedure itself.

3.1 Project Files

Listing 3.1: Structure of the team-39A/labs/project directory

```
team-39A/labs/project
                |-- README.md
                |-- build_push.sh
                |-- gcp
                |   |-- ...
                |-- microservices
                |   |-- ...
                |-- report
                |   |-- ...
                |-- demo
                |   |-- ...
```

This is the root folder of the project, here we find:

- The build_push.sh script: this script encompasses the setup of the GAR repository and the building and pushing of the Micro Services (MS) images to said repository. This is only a convenience, in a real life scenario, this would not be

acceptable and more appropriate tools like Jenkins or Google Cloud Build would be advised;

- gcp directory: Here is where all the Terraform files reside;
- microservices directory: Here is where all the code and Dockerfiles for the MS reside;
- report directory: Here is where the checkpoints and final reports are;
- demo directory: Here is where the demo guide is;

The main focus of this project was the /gcp directory and so, we will focus on that.

Listing 3.2: Structure of the /gcp directory

```
gcp
    |-- gke
    |   |-- ...
    |-- k8s
    |   |-- ...
    |-- registry
    |   |-- ...
    |-- main.tf
    |-- terraform.tfvars
    |-- variables.tf*-
```

On this folder we have three modules, one for each step of the infrastructure deployment. First the GAR repository must be created in order to store the container images that we will use in the Kubernetes deployments. This is the purpose of the registry module, that resides in the registry directory.

After that is created and the images are in there we can proceed to the creation of the K8S cluster itself. This is the responsibility of the gke module, defined in the gke directory.

As soon as the cluster finishes creation we can start deploying in it. The k8s module, defined in the k8s directory, defines all the things to be deployed inside the cluster (services, deployments, secrets, etc).

As was previously stated the registry module is used by the build_push.sh script and so was not included in the main.tf file.

The main.tf file is where the gke and k8s modules are configured and executed.

3.1.1 Registry module

Overview

Listing 3.3: Structure of the registry module

```
registry
    |-- provider.tf
    |-- registry.tf
    |-- variables.tf
```

In this module we define the GAR repository, for Docker images. In order to do this we must use the Terraform google-beta provider as this feature is not available in the non beta provider.

Inputs

This module needs the following information in order to execute:

- The region where the cluster will be deployed;
- The project id where the repository will be deployed;
- Json key file for accessing GCP;
- The repository name/id;

3.1.2 Gke module

Overview

Listing 3.4: Structure of the gke module

```
gke
    |-- cluster.tf
```

```
    |-- outputs.tf
    |-- provider.tf
    |-- templates
    |   |-- kubectl.tmpl
    |-- variables.tf
    |-- vpc.tf
```

In this module we define the K8S cluster. To do this we used Terraform google provider.

We first created a VPC network, by default GCP blocks all access from outside this network, effectively separating our cluster from the internet, when we add resources to this network, for instance, the cluster, GKE creates the necessary firewall rules in order to allow communication with the kube-apiserver, or ingress endpoint. We then define a subnetwork of this network with two IP ranges, one for the pods and another for the services.

Now that we have the VPC network we can define the cluster itself.

This cluster will be deployed in the network created and the subnetwork's IP ranges allocated to the services and pods. Unlike in the labs, we opted to create a separate node pool, this enables us to add or remove nodes without having to redeploy the entire cluster.

Inputs

This module needs the following information in order to execute:

- The region where the cluster will be deployed;
- The zone where the cluster will be deployed;
- Json key file for accessing the google cloud;
- The project id where the cluster will be deployed;
- Number of worker nodes for the cluster;
- Type of machine to use for the kubernetes nodes;
- A map of ip ranges for the pods and services;

Outputs

We output the following values out of the module:

- The IP address of the cluster's Kubernetes master;
- The private key used by clients to authenticate to the cluster endpoint;
- The public certificate used by clients to authenticate to the cluster endpoint;
- The public certificate that is the root of trust for the cluster;

This module creates a shell script with the necessary gcloud command to configure kubectl for the local machine. This is for convenience.

3.1.3 K8S module

Overview

Listing 3.5: Structure of the k8s module

```
k8s
    |-- bootstorage.tf
    |-- expressed.tf
    |-- happy.tf
    |-- ingress.tf
    |-- monitoring.tf
    |-- output.tf
    |-- provider.tf
    |-- pv.tf
    |-- redis.tf
    |-- resources
    |   |-- nginxdash.json
    |-- templates
    |   |-- ingress-nginx.tmpl
    |   |-- ingressIP.tmpl
    |   |-- kube-prometheus-stack.tmpl
    |-- variables.tf
    |-- vuecalc.tf
```

In this module we define the things that will get deployed in the cluster. To do this we used Terraform google provider and Terraform helm provider.

We can divide these files into 4 categories:

- Monitoring:
  - monitoring.tf;

    The monitoring stack used was the kube-prometheus-stack helm chart that provides us with a simple way of deploying and configuring the monitoring stack.

    This chart was deployed in the kube-prometheus namespace.

    By default this chart does not configure the monitoring for the ingress-nginx controller, so we had to add this capability. To do so we had to set a value so that Prometheus goes looking for serviceMonitors outside the helm specified namespace. We also added a persistent volume to Prometheus.

    Since ingress-nginx is not supported by default we also had to add an additional grafana dashboard to view the data gathered by Prometheus. We did this by defining a configMap with the JavaScript Object Notation (JSON) definition of the dashboard. Grafana is configured to used configMaps that have a specified labels in their metadata.

    We also configure Grafana to enable Ingress on the /grafana path and to use the nginx controller.

- Micro Services:

  These files define the deployments and services of our application.

  All the services are of type ClusterIP as we do not want to expose them directly to outside the cluster.

  All the deployments have startup, readiness and liveness probes and one replica.

  All the components of this category are defined inside the default namespace.

  - bootstorage.tf;

    Here we define the bootstorage service and deployment. We also define a secret for bootstorage to use when connecting with the Redis service.

  - vuecalc.tf;

    Here we define the vuecalc service and deployment. For this deployment the application requires us to define the base Uniform Resource Locator (URL) for the multiple API we offer. To do this we needed to set multiple environment variables.

  - happy.tf;

    Here we define the happy service and deployment.

  - expressed.tf;

    Here we define the bootstorage service and deployment.

- Storage:

  - pv.tf;

    Here we define the storageClass and the persistentVolumeClaim for the redis deployment. The storageClass is provisioned by Google Compute Engine (GCE) and is of type pd-ssd. The reclaim policy is retain so that the storage does not get deleted when it is released.

  - redis.tf;

    In this file we define the redis deployment of one replica.

    We specify the startup args so that redis will periodically dump to disk its internal state. We give it the volume claimed to mount on the /data directory.

    We also define a headless service, a service with no clusterIP that instead will return the ip of the pod running the application, in this case, Redis.

- Ingress:

  - ingress.tf;

    In order to access our services, which only have an internal clusterIP we must expose them through Ingress. We could do this only using it as an API gateway with the cloud provider controller doing the work, meaning that we would have to expose the services to outside the cluster, or we could define the controller ourselves inside the cluster meaning that along with the API

gateway functionality we could also use it as a reverse proxy. We chose the latter.

Our ingress controller of choice was the nginx-ingress-controller. We deployed it making use of the ingress-nginx helm chart.

The controller was deployed inside its own namespace ingress-controller.

Since we wanted to gather metrics about the controller we had to enable that functionality.

In this file we also define the Ingress for the services the application offers, this Ingress is defined in the default namespace, where the services are defined. We must state in the Ingress metadata for it to use the nginx controller.

Inputs

This module needs the following information in order to execute:

- The IP address of the cluster's Kubernetes master;
- The private key used by clients to authenticate to the cluster endpoint;
- The public certificate used by clients to authenticate to the cluster endpoint;
- The public certificate that is the root of trust for the cluster;
- The region where the image repository is deployed;
- The project id where image repository is deployed;
- The Grafana access password for the admin user;

Outputs

This module creates a markdown file with the IP address of the ingress created.

3.2 Deployment

The next subsections describe the steps needed to correctly deploy and configure the MS application into the K8S cluster.

First we will go through all the necessary pre-requisites, followed by the creation of a repository to store the docker images, and lastly the infrastructure provisioning and deployment.

3.2.1 Pre-requisites

In order to deploy our project there is a set of requirements that must be met.

First, lets overview what software needs to be installed in the machine that will preform the deployment. All software used was on the latest version as of November 2021;

- Terraform;
- Docker Engine;
- npm;
- JDK8;
- maven;
- gcloud;
- awk;
- bash;

Now that the required software is installed we can move on to setting up the GCP environment:

1. Create a new project;
2. Enable GKE API;
3. Enable GAR API;

4. Add Kubernetes Engine Service Agent Role to the Compute Engine default service account;
5. Create new key for the Compute Engine default service account, in the JSON format;
6. Save this key in the /project/gcp directory;

We can now set up the /project/gcp/terraform.tfvars file with the missing values, or change the already present ones.

Listing 3.6: terraform.tfvars

```
region        = "europe-west1"
zone          = "b"
project       = "<PROJECT_ID>"
credentials   = "<JSON_KEY_PATH>"
node_count    = 3
machine_type = "n1-standard-2"
secondary_ip_ranges = {
"pod_ip_range"      = "10.0.0.0/14"
"services_ip_range" = "10.4.0.0/19"
}
registry = "myrepo" # GAR repository ID to be created
grafana_password = "grafanamonitoring"
```

For a more detailed step-by-step please refer to the video that accompanies this report.

3.2.2 Artifact Registry Repository Creation

In order to proceed to the repository creation using GAR, a script located in the root of the project called `build_push.sh` was created, responsible for:

- Authenticating with gcloud;
- Setting the project in gcloud;
- Create the GAR repository in the GCP project;

- Configure Docker to use this repository;
- Build the MS images and push them to this repository;

```
$ ./build_push.sh
```

Figure 3.1: Google Artifact Registry

3.2.3 K8S cluster Provisioning and Deployment

Go to the /project/gcp directory of the project and follow the following steps:

The following commands create and provision the infrastructure. The outputs obtained should be similar to the ones presented.

```
$ terraform init
```

```
Initializing modules...

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/google from the dependency lock file
- Reusing previous version of hashicorp/local from the dependency lock file
- Reusing previous version of hashicorp/kubernetes from the dependency lock file
- Reusing previous version of hashicorp/helm from the dependency lock file
- Using previously-installed hashicorp/google v4.1.0
- Using previously-installed hashicorp/local v2.1.0
- Using previously-installed hashicorp/kubernetes v2.6.1
- Using previously-installed hashicorp/helm v2.4.1

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Figure 3.2: Terraform Init

```
$ terraform plan
```

```
  # module.k8s.local_file.ingressIP will be created
  + resource "local_file" "ingressIP" {
      + content              = (known after apply)
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "ingressIP.md"
      + id                   = (known after apply)
    }

Plan: 25 to add, 0 to change, 0 to destroy.
```

Figure 3.3: Terraform Plan

```
$ terraform apply auto-approve
```

```
module.k8s.kubernetes_deployment.vuecalc: Still creating... [30s elaps
module.k8s.kubernetes_deployment.vuecalc: Still creating... [40s elaps
module.k8s.kubernetes_deployment.vuecalc: Still creating... [50s elaps
module.k8s.kubernetes_deployment.vuecalc: Creation complete after 56s

Apply complete! Resources: 25 added, 0 changed, 0 destroyed.
```

Figure 3.4: Terraform Apply

- Now, in order to access the cluster and run commands, kubectl is used, and can be configured with the following command. Zone and ProjectID correspond to your project's specifications.

  Alternatively we can run the kubectl.sh file in the /gcp directory that already has these values set.

```
$ gcloud container clusters get-credentials vpc-native-cluster --
   zone <Zone> --project <ProjectId>
```

```
$ ./kubectl.sh
Fetching cluster endpoint and auth data.
kubeconfig entry generated for vpc-native-cluster.
```

- Using kubectl, the namespaces can be retrieved.

```
$ kubectl get namespaces
```

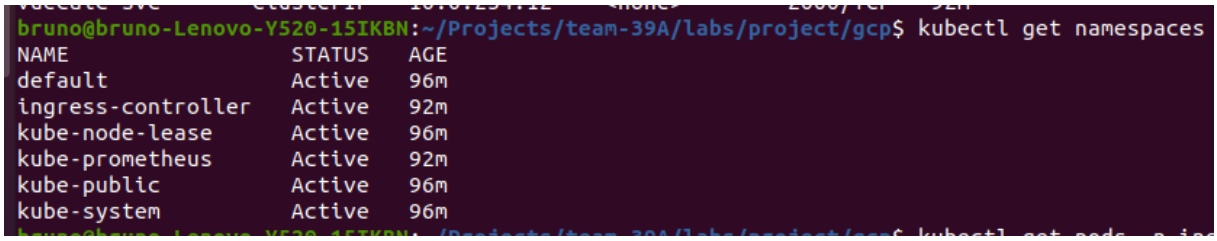

Figure 3.5: Cluster namespaces

- Check default namespace, namespace in which our microservices app is located, with which it is displayed all the components, particularly the pods and services referring to it.

  As we can see in figure 3.6, none of the components is exposed to the internet.

```
$ kubectl get all -n default
```

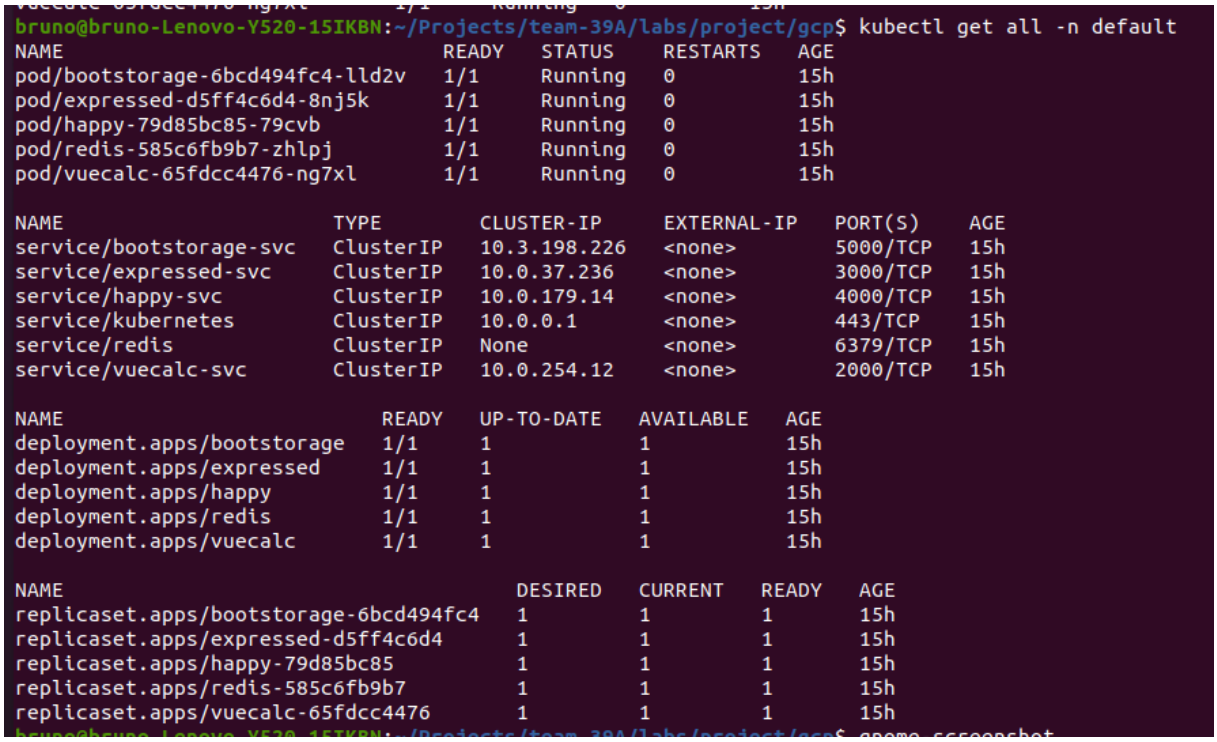

Figure 3.6: Application namespace

- In order to access the application, we need to retrieve the IP exposed by the Ingress and we can get it by looking at the correspondent namespace. In the services, the nginx-controller exposes an IP.

  Alternatively we can cat the ingressIP.md file in the /gcp directory that already has external IP of the Ingress endpoint.

```
$ kubectl get all -n ingress-controller
```

```
$ cat ingressIP.md
# Ingress IP address:
34.79.197.230
```

```
bruno@bruno-Lenovo-Y520-15IKBN:~/Projects/team-39A/labs/project/gcp$ kubectl get all -n ingress-controller
NAME                                            READY   STATUS    RESTARTS   AGE
pod/ingress-nginx-controller-777d8b54b7-ktgzm   1/1     Running   0          15h

NAME                                         TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)                      AGE
service/ingress-nginx-controller             LoadBalancer   10.0.171.103   34.79.197.230   80:31699/TCP,443:32474/TCP   15h
service/ingress-nginx-controller-admission   ClusterIP      10.0.151.101   <none>          443/TCP                      15h
service/ingress-nginx-controller-metrics     ClusterIP      10.0.134.22    <none>          10254/TCP                    15h

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ingress-nginx-controller   1/1     1            1           15h

NAME                                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/ingress-nginx-controller-777d8b54b7   1         1         1       15h
```

Figure 3.7: Ingress

- We then copy the correspondent IP to a browser and we can access the application UI.

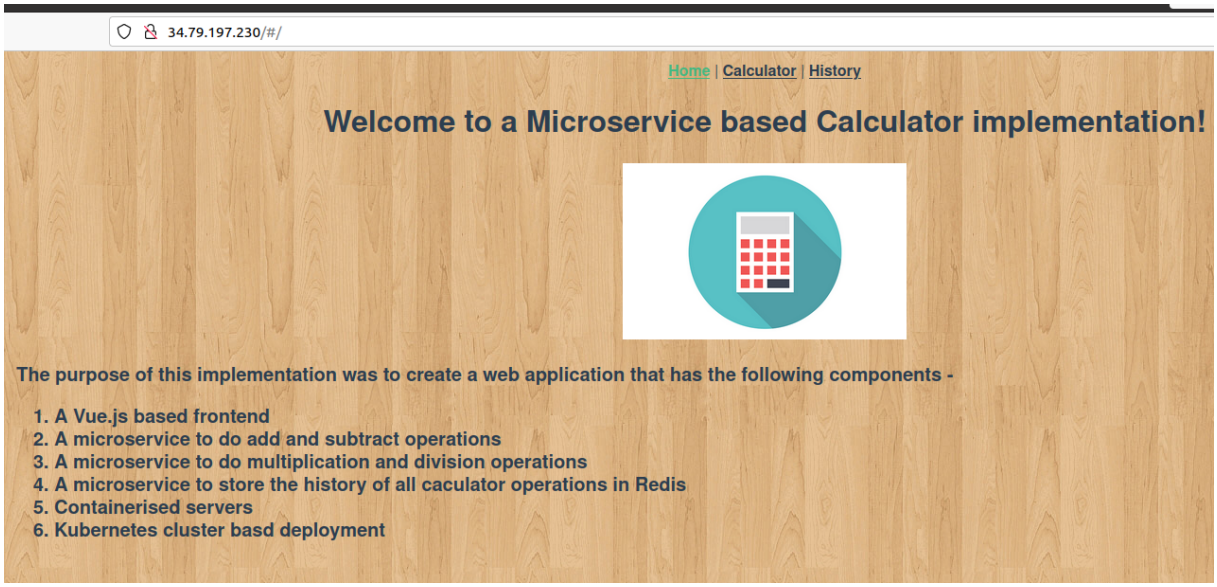

Figure 3.8: Application Interface

- Access the calculator by clicking on ”Calculator” link in to top and the command history by clicking on ”History”.

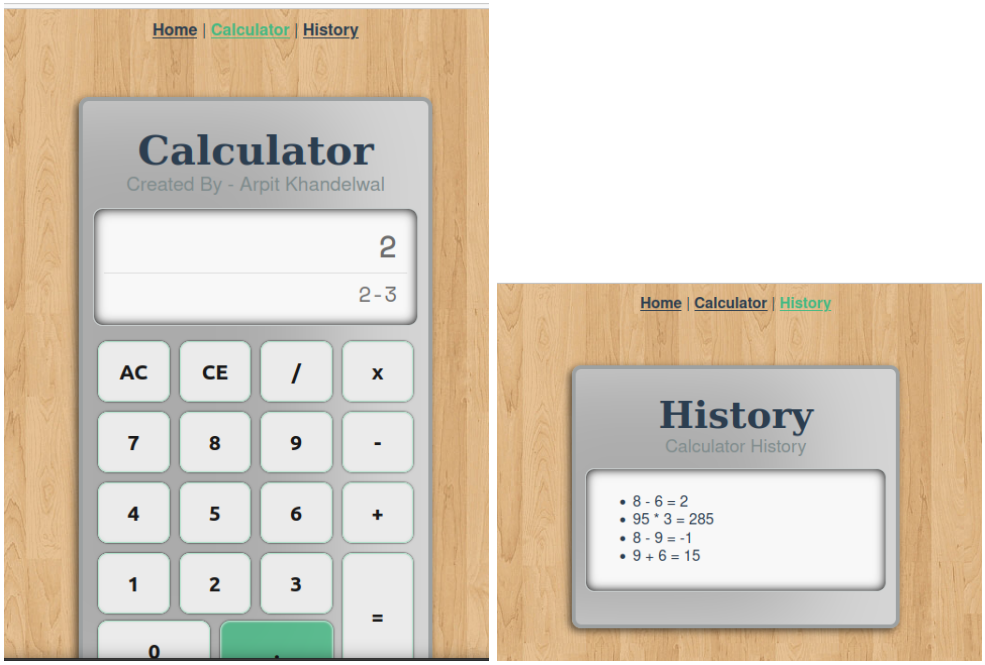

Figure 3.9: Calculator and History

3.3 Monitoring

- Here we can see all the different components of kube-prometheus.

```
$ kubectl get all -n kube-prometheus
```

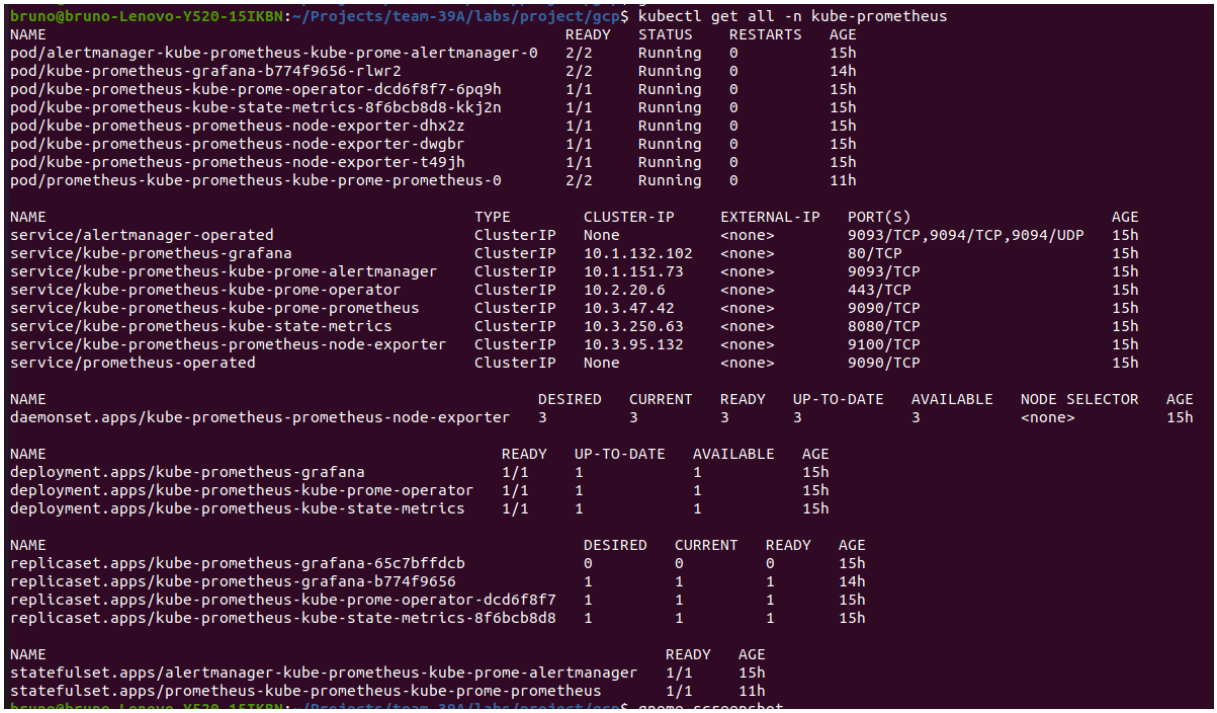

Figure 3.10: Kube-prometheus

- In order to access Grafana, enter the IP address retrieved in last section followed by /grafana. The default username is ”admin”. In the tool we can access different dashboards to check metrics from our system, which allows for monitoring different aspects of the cluster.

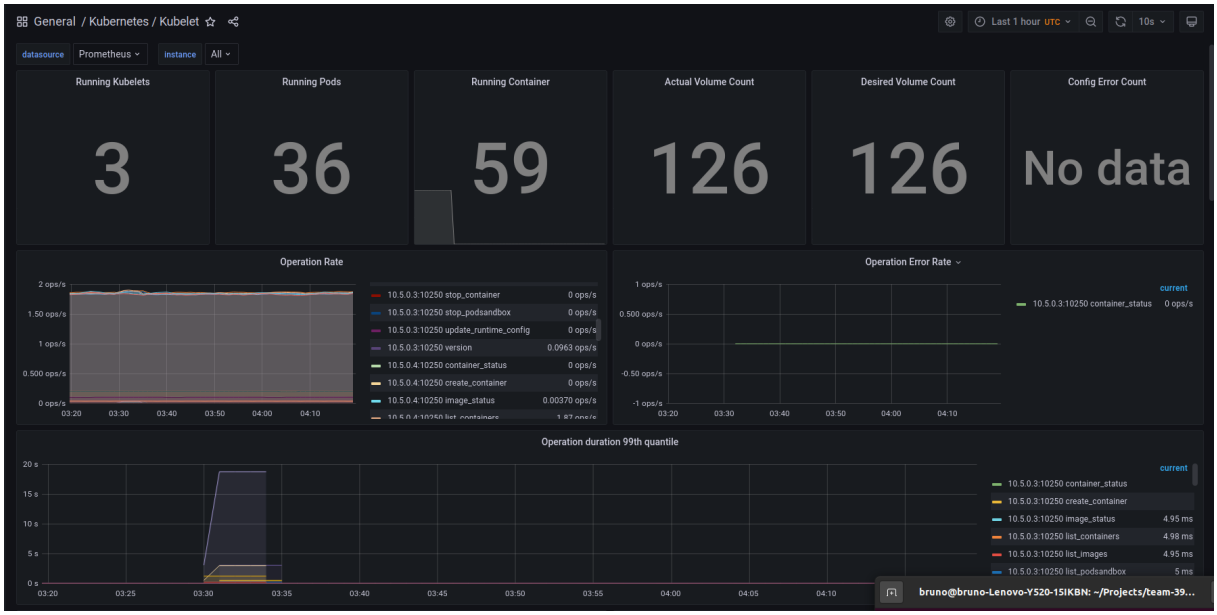

Figure 3.11: Grafana

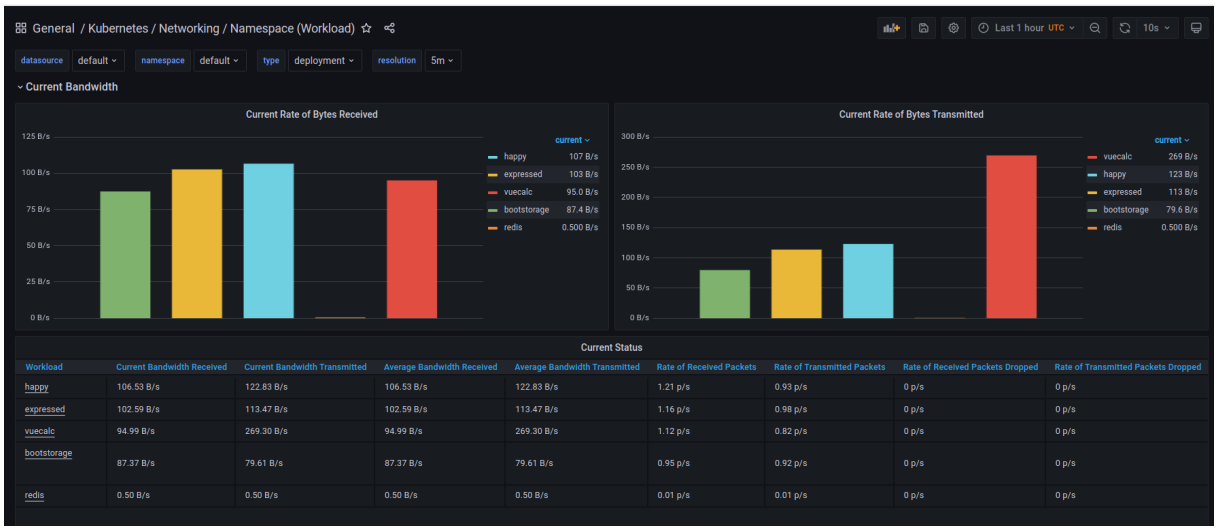

Figure 3.12: Grafana Micro-Services Workload

Chapter 4

Final Remarks

An improvement to be made would be to add TLS to the application, this could be done using Cert-Manager.

Another improvement would be to add NGINX Service Mesh to our cluster, this would allow securing the communication between pods, traffic shaping so that when deploying new versions only a small amount of traffic goes to them in case of bugs and better visualization.

Finally our project only has one environment, in a real life case we would have to have multiple, one for development, staging and production.