Stan's Robot Shop

A Scalable Microservices-based Web Application in a Public Cloud

Management and Administration of IT Infrastrutures and Services

Team	**20A**
89399:	Afonso Gonçalves
90621:	Maria Filipe
89498:	Maria Martins

Information Systems and Computer Engineering
IST-ALAMEDA

2021/2022

Contents

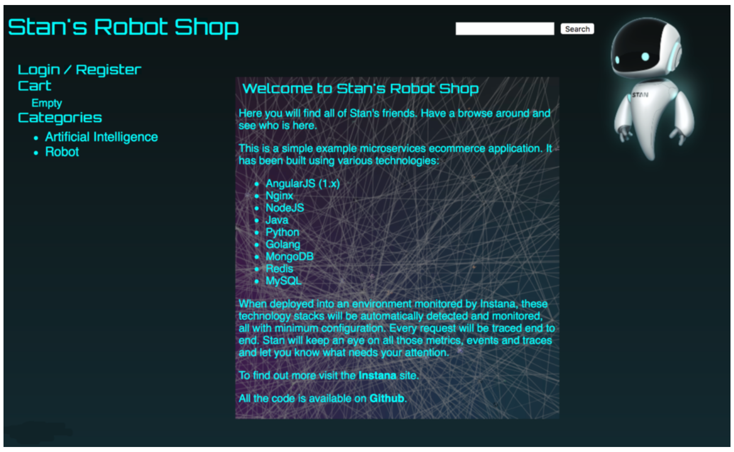
List of Figures

Chapter 1

Introduction to the application

This project was developed within the scope of Management and Administration of IT Infrastructures and Service. The students were proposed to deploy and provision a tiered Micro-services-based containerized Web Application on a Public Cloud provider, using automation tools as well as implementing instrumentation on the applications, on services and on the infrastructure components of the solution, to allow monitoring and logging features.

The Stan's Robot Shop [1] is a sample micro-service application chosen for the development of the project, that can be used as a sandbox to test and learn containerised application orchestration and monitoring techniques. It is a simple e-commerce storefront that includes: a product catalogue, a user repository, a shopping cart and a on order pipeline and was built using several technologies, such as NodeJS (Express), Java (Spring Boot), Python (Flask), Golang, PHP (Apache), MongoDB, Redis, MySQL (Maxmind data), RabbitMQ, Nginx and AngularJS.



Welcome to Stan's Robot Shop

Here you will find all of Stan's friends. Have a browse around and see who is here.

This is a simple example microservices ecommerce application. It has been built using various technologies:

- AngularJS (1.x)
- Nginx
- NodeJS
- Java
- Python
- Golang
- MongoDB
- Redis
- MySQL

When deployed into an environment monitored by Instana, these technology stacks will be automatically detected and monitored, all with minimum configuration. Every request will be traced end to end. Stan will keep an eye on all those metrics, events and traces and let you know what needs your attention.

To find out more visit the **Instana** site.

All the code is available on **Github**.

Figure 1.1: Stan's Robot Shop Website

Chapter 2

Project Architecture

Figure 2.5 details the project architecture, with all the components and their relationships. Each micro-service is replicated to assure the availability of this application. The basic interaction is as follows:

The client interacts with the web-service, which is the entry point for every service in this application. The dispatch service then balances the request load among the targeted service replicas. Each micro-service handles a task, as shown below:

- The User service handles user tasks, such as login, registering, viewing a user shopping cart, and history;

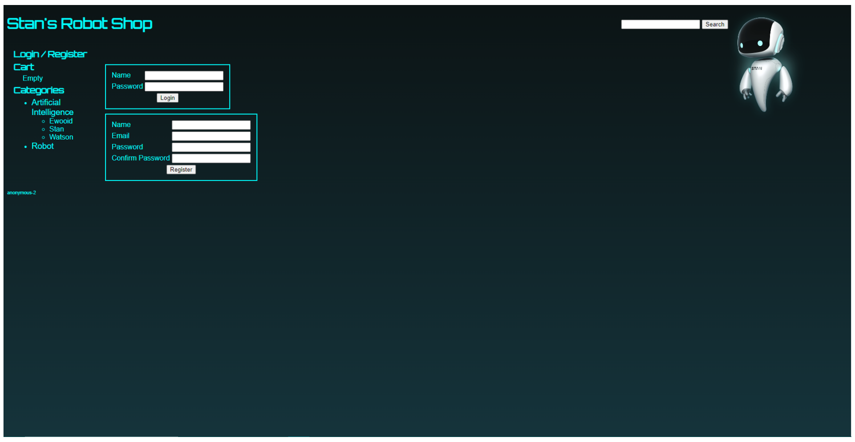
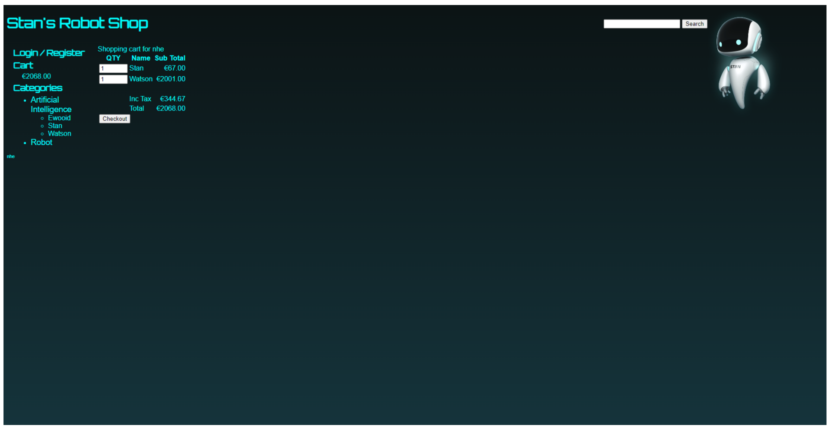


Figure 2.1: Stan's Robot Shop Login page

- The Payment service handles all the tasks related with order payments;

- The Cart service manages the shopping cart of each session;

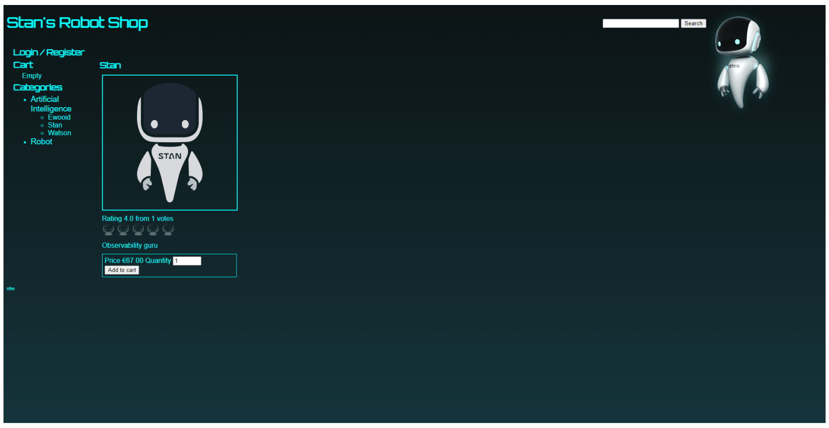


Shopping cart for nhe

QTY	Name	Sub Total
1	Stan	€67.00
1	Watson	€2001.00
	Inc Tax	€344.67
	Total	€2068.00

Figure 2.2: Stan's Robot Shop Shopping Cart

- The Ratings service handles the rating of every product in the shop



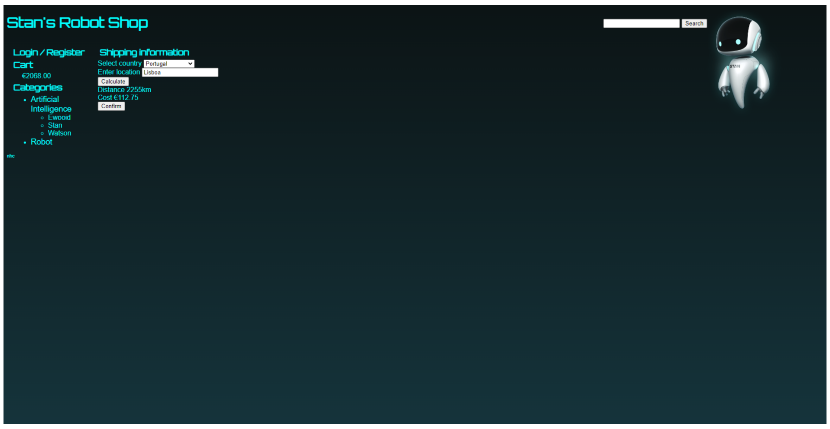
Rating 4.0 from 1 votes

Observability guru

Price €67.00 Quantity 1

Figure 2.3: Stan's Robot Shop Robot Ratings

- The Shipping service manages and allows users to track the delivery of their orders.



Shipping information

Select country Portugal

Enter location Lisboa

Distance 2255km

Cost €112.75

Figure 2.4: Stan's Robot Shop Shipping

All these services require application data, thus they all access a common (replicated) storage architecture, composed of three micro-services (Redis, MySQL and MongoDB).

This system also includes a monitoring service that regularly prompts each service to check for its health status. This way, the application managers can timely detect anomalies in the infrastructure and enhance its maintenance efficiency. Figure 2.5 displays the services and their connection.

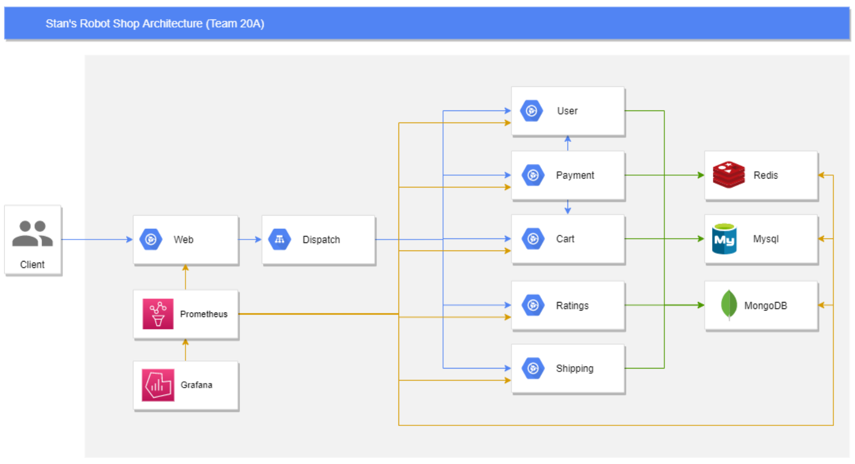


Figure 2.5: Stan’s Robot Shop Architecture

Chapter 3

Implementation

3.1 Used Technologies

This project used an **Infrastructure as Code** (IaC) approach [2] to declare and set up its infrastructure, by using **Terraform** [3], and choosing **Google Cloud Provider** (GCP) [4] to host it. In Checkpoint I, we used **Google Compute Engine** (GCE) to deploy a **Virtual Machine** (VM) for each service and **Ansible** [5] to automatically manage those nodes (install docker, pull container images and run them in each machine). However, in Checkpoint II, we changed our approach to use directly the **Google Kubernetes Engine** (GKE), which would automatically deploy and initialize our application **Docker** containers [6]. This rendered Ansible useless, as we did not require any management operation in the whole project. In the second delivery we also used **Helm** [7] to orchestrate the creation of the **Kubernetes** [8] Pods and Services. For the final delivery, we employed **Prometheus** [9] and **Grafana** [10] to, respectively, monitor the application performance and to retrieve that information within a clean visualization. The entire infrastructure is managed from a VM, which was configured using **Vagrant** [11].

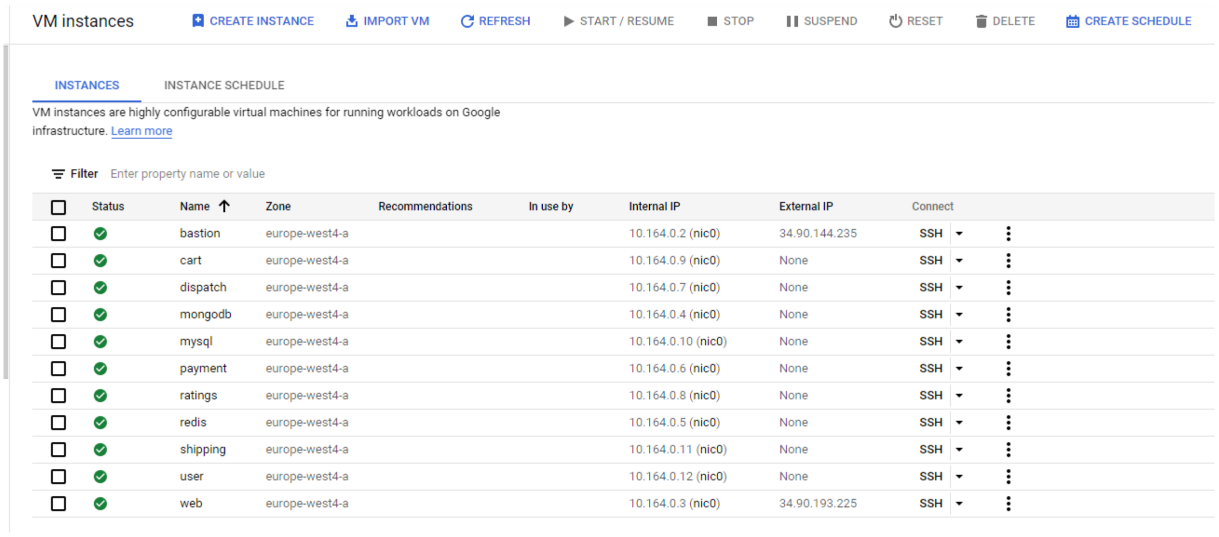
3.2 Development along Laboratories & Challenges

3.2.1 Checkpoint I

In the first iteration of this project, we declared an infrastructure containing one virtual machine per service, using **Terraform** and the **Google Compute Engine** (GCE) service. These machines were firstly connected by a simple default network, however we soon realized that this setup was not the best, since anyone could make a request to any machine. Thus, we worked on creating a better access control for this infrastructure. First, we declared a firewall that would allow only HTTP traffic to the web VM. Since Ansible would need to SSH into every machine to configure it, we also allowed SSH traffic to every machine and for debug purposes, we opted to allow ICMP traffic to every VM as well.

This solution would still expose too much our infrastructure, due to the fact that every machine would be exposed to the outside of the VPC. After searching about this issue and speaking to our professor, we were advised to use the **Bastion Host** architecture [12]. This architecture uses a virtual machine as the entry point of the VPC, keeping all other VMs private. This allows for a great access control to the inside of the VPC, since all the (non web) traffic passes through that machine. For this purpose, we declared a new VM and configured **Ansible** to access the **VPC** through it.

Finally, we used Ansible to perform maintenance operations, such as installing Docker and running our containers. For that task, Ansible requires the IPs of the machines to connect to, but these change at each apply. Setting those IPs manually would be a burdensome task and would be infeasible with large-scale applications with hundreds of machines. Instead, we used the terraform outputs and its templating capabilities to automatically generate a new Ansible inventory file each time it created new machines with new IPs.



VM instances are highly configurable virtual machines for running workloads on Google infrastructure. Learn more

Filter Enter property name or value

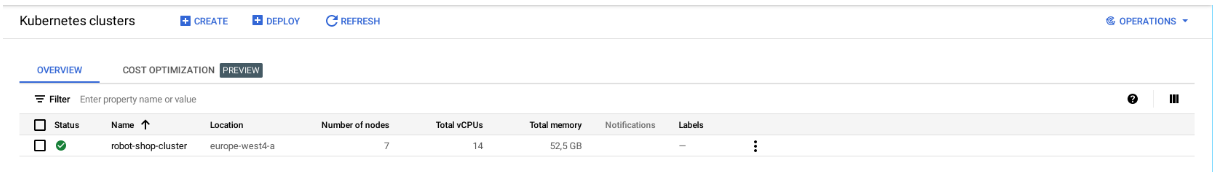
Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
✓	bastion	europe-west4-a			10.164.0.2 (nic0)	34.90.144.235	SSH
✓	cart	europe-west4-a			10.164.0.9 (nic0)	None	SSH
✓	dispatch	europe-west4-a			10.164.0.7 (nic0)	None	SSH
✓	mongodb	europe-west4-a			10.164.0.4 (nic0)	None	SSH
✓	mysql	europe-west4-a			10.164.0.10 (nic0)	None	SSH
✓	payment	europe-west4-a			10.164.0.6 (nic0)	None	SSH
✓	ratings	europe-west4-a			10.164.0.8 (nic0)	None	SSH
✓	redis	europe-west4-a			10.164.0.5 (nic0)	None	SSH
✓	shipping	europe-west4-a			10.164.0.11 (nic0)	None	SSH
✓	user	europe-west4-a			10.164.0.12 (nic0)	None	SSH
✓	web	europe-west4-a			10.164.0.3 (nic0)	34.90.193.225	SSH

Figure 3.1: VM instances in the GCP

3.2.2 Checkpoint II

In this Checkpoint, we aimed to deploy the application in the cloud. The infrastructure we had configured thus far was not the best suited since several tasks such as networking between nodes, maintenance, failure recovery, etc. were not automated. Fortunately, **Kubernetes** handles all these tasks automatically and creates a better environment to deploy, manage and scale our application, therefore we changed our approach to deploy our application in a Kubernetes Cluster. Since we were already using GCP, we decided to use the GKE.

To deploy our application, we first need to create a Kubernetes Cluster and only then declare the Services and Pods that will be running in it. For the first task, we used the *google* Terraform provider to create a cluster with 7 working nodes.



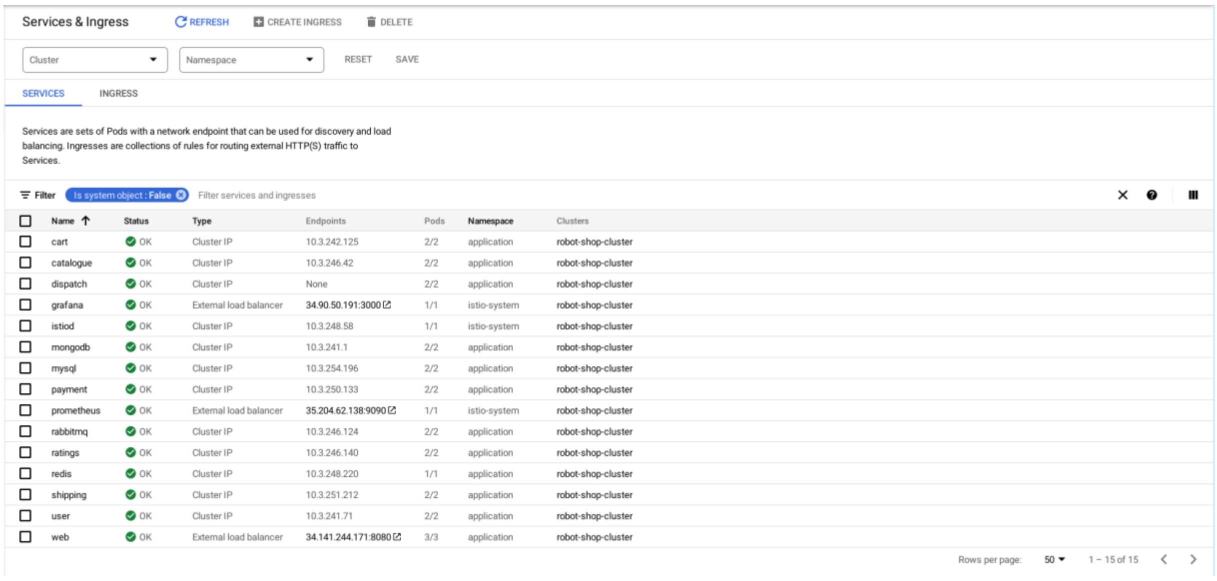
Filter Enter property name or value

Status	Name	Location	Number of nodes	Total vCPUs	Total memory	Notifications	Labels
✓	robot-shop-cluster	europe-west4-a	7	14	52,5 GB		–

Figure 3.2: Kubernetes cluster nodes

When creating the Services and Pods, instead of declaring them in the Terraform files, we took advantage of the Helm configuration existing in the source code of our application. Helm charts declare how to ”define, install and upgrade” [7] a Kubernetes application, thus deploying the Stan’s Robot Shop would be as simple as running a *helm install* command. However this required us to first create the cluster and only then to install the application in it, but we wanted to have a fully functional application with a single command. Fortunately, the Helm Terraform Provider allows application deployment in Kubernetes using Terraform, nonetheless its usage is not trivial: The deployment of the application requires cluster credentials that would only exist after the cluster creation. This dependency was solved by creating two modules: The first would create a cluster and output the required credentials. The second would deploy the application using Helm and the cluster credentials outputted by the first one. A main file would coordinate the execution of both modules and pass the required parameters (such as variables and credentials) to each of them. The materials given in the laboratories proved to be very helpful in this task since they already created a GKE cluster and deployed an application in it. We took the opportunity to use this clever module idea on our own project.

Kubernetes allows the developer to define the number of replicas of each service. The present Helm configuration only set one replica per service, so we changed the Helm Chart to create 3 replicas for the web service and 2 replicas for every other service. We set 3 replicas for the web service since it is the frontend and would be the service that had more load in the application.



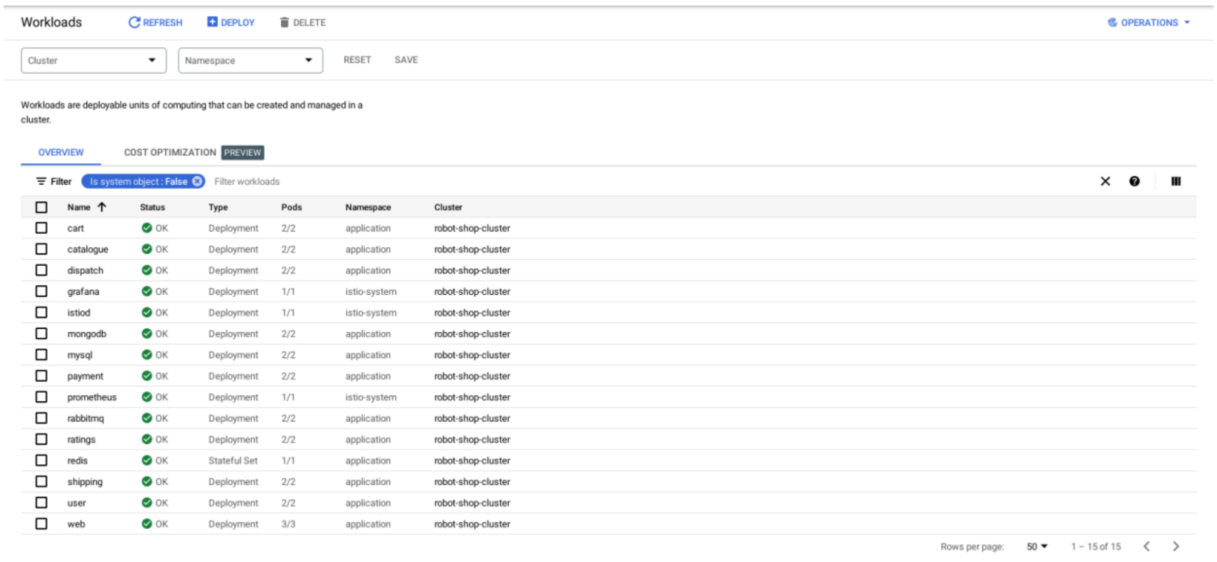
Services are sets of Pods with a network endpoint that can be used for discovery and load balancing. Ingresses are collections of rules for routing external HTTP(S) traffic to Services.

Filter: Is system object : False — Filter services and ingresses

Name	Status	Type	Endpoints	Pods	Namespace	Clusters
cart	✓ OK	Cluster IP	10.3.242.125	2/2	application	robot-shop-cluster
catalogue	✓ OK	Cluster IP	10.3.246.42	2/2	application	robot-shop-cluster
dispatch	✓ OK	Cluster IP	None	2/2	application	robot-shop-cluster
grafana	✓ OK	External load balancer	34.90.50.191:3000	1/1	istio-system	robot-shop-cluster
istiod	✓ OK	Cluster IP	10.3.248.58	1/1	istio-system	robot-shop-cluster
mongodb	✓ OK	Cluster IP	10.3.241.1	2/2	application	robot-shop-cluster
mysql	✓ OK	Cluster IP	10.3.254.196	2/2	application	robot-shop-cluster
payment	✓ OK	Cluster IP	10.3.250.133	2/2	application	robot-shop-cluster
prometheus	✓ OK	External load balancer	35.204.62.138:9090	1/1	istio-system	robot-shop-cluster
rabbitmq	✓ OK	Cluster IP	10.3.246.124	2/2	application	robot-shop-cluster
ratings	✓ OK	Cluster IP	10.3.246.140	2/2	application	robot-shop-cluster
redis	✓ OK	Cluster IP	10.3.248.220	1/1	application	robot-shop-cluster
shipping	✓ OK	Cluster IP	10.3.251.212	2/2	application	robot-shop-cluster
user	✓ OK	Cluster IP	10.3.241.71	2/2	application	robot-shop-cluster
web	✓ OK	External load balancer	34.141.244.171:8080	3/3	application	robot-shop-cluster

Rows per page: 50 1 – 15 of 15

Figure 3.3: Pods and their rules



Workloads REFRESH DEPLOY DELETE OPERATIONS

Cluster Namespace RESET SAVE

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

OVERVIEW COST OPTIMIZATION PREVIEW

Filter Is system object : False Filter workloads

Name	Status	Type	Pods	Namespace	Cluster
cart	OK	Deployment	2/2	application	robot-shop-cluster
catalogue	OK	Deployment	2/2	application	robot-shop-cluster
dispatch	OK	Deployment	2/2	application	robot-shop-cluster
grafana	OK	Deployment	1/1	istio-system	robot-shop-cluster
istiod	OK	Deployment	1/1	istio-system	robot-shop-cluster
mongodb	OK	Deployment	2/2	application	robot-shop-cluster
mysql	OK	Deployment	2/2	application	robot-shop-cluster
payment	OK	Deployment	2/2	application	robot-shop-cluster
prometheus	OK	Deployment	1/1	istio-system	robot-shop-cluster
rabbitmq	OK	Deployment	2/2	application	robot-shop-cluster
ratings	OK	Deployment	2/2	application	robot-shop-cluster
redis	OK	Stateful Set	1/1	application	robot-shop-cluster
shipping	OK	Deployment	2/2	application	robot-shop-cluster
user	OK	Deployment	2/2	application	robot-shop-cluster
web	OK	Deployment	3/3	application	robot-shop-cluster

Rows per page: 50 1 – 15 of 15

Figure 3.4: Workloads

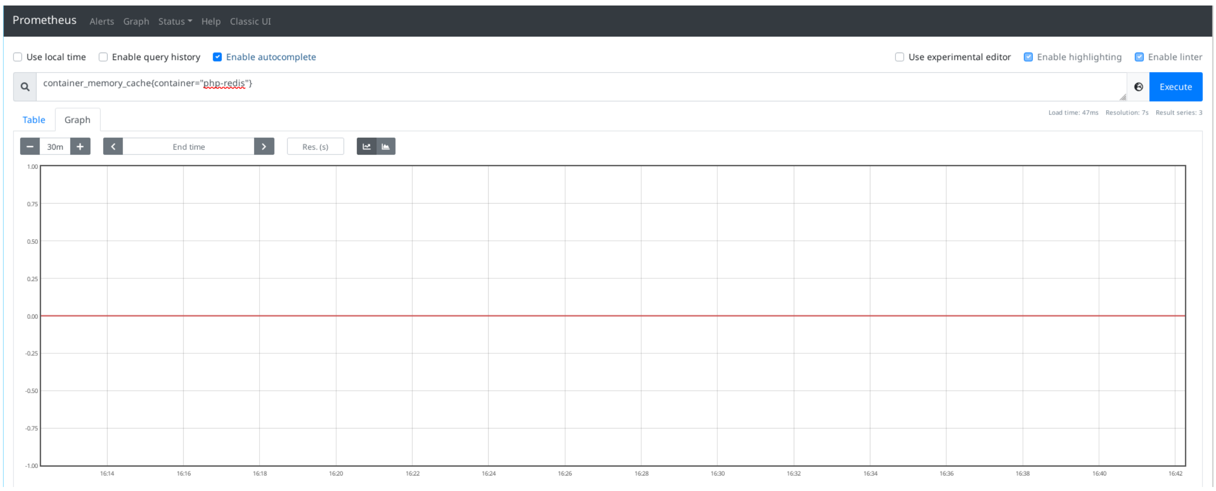
In the process of deploying the application in the GKE we ran into two minor issues: First, the Helm deployment would fail every time it's duration was superior than 5 minutes, due to a timeout. After reading the documentation [13] and approaching our teacher, we configured the helm timeout to a greater interval. After this fix, we still had trouble executing the *terraform apply* command due to Terraform attempt to access resources reported in the *tfstate* that did not exist anymore. After discussing with our teacher, we concluded that this inconsistency happened since we were applying a new Terraform configuration without destroying the previous one. This would cause some conflicts since Terraform would expect some resources to exist, while they didn't. Our teacher advised us to run *terraform destroy* every time a deploy went wrong and only then run *terraform apply*. After having this practice, we had no more troubles deploying our application and acquired knowledge on this subject!

3.2.3 Final Delivery

The objective for the last delivery was to deploy the same Cloud-Native containerized application of the previous checkpoint, in a High-Available and scalable cloud-based infrastructure and to add a Service Mesh for observability (telemetry and monitoring), traffic management, security, and policing. A Service Mesh is an infrastructure layer that handles and abstracts a safe, fast and reliable inter service communication [14]. We used the ISTIO Service Mesh, which does this by ”adding a sidecar proxy that intercepts all network communication between micro-services” [15].

We first tried to monitor our application using Instana [16], since our source code was built to work with this platform. However, this required creating a new Instana account and this would increase the complexity of the project setup. Therefore, we aimed to install Prometheus and Grafana instead. Having ISTIO installed, it was possible to deploy these tools to monitor our infrastructure and get alerts if something went wrong. We used the default configuration used in the course laboratories because it already covers a wide range of metrics and we did not required any other metric or visualization that was not covered by Prometheus or Grafana.

ISTIO was installed using the Terraform Helm Provider, since its package already contained useful charts for this task. For Prometheus and Grafana, we already had *yaml* files that configured how these tools would be deployed. We used the Terraform Kubectl Provider to load that configuration into the cloud micro-services.



```
container_memory_cache{container="php-redis"}
```

Figure 3.5: Prometheus graph for the container memory cache metric from the php-redis

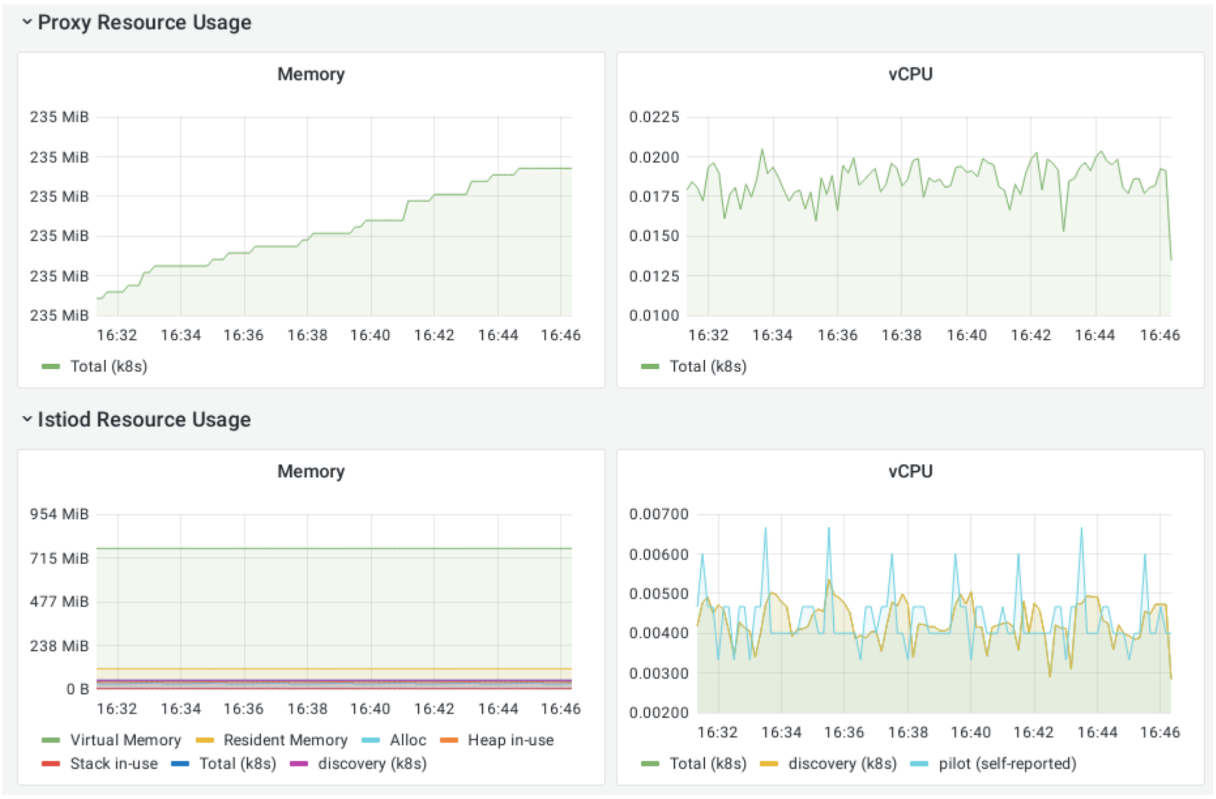


Figure 3.6: Grafana proxy resource usage and istiod resource usage

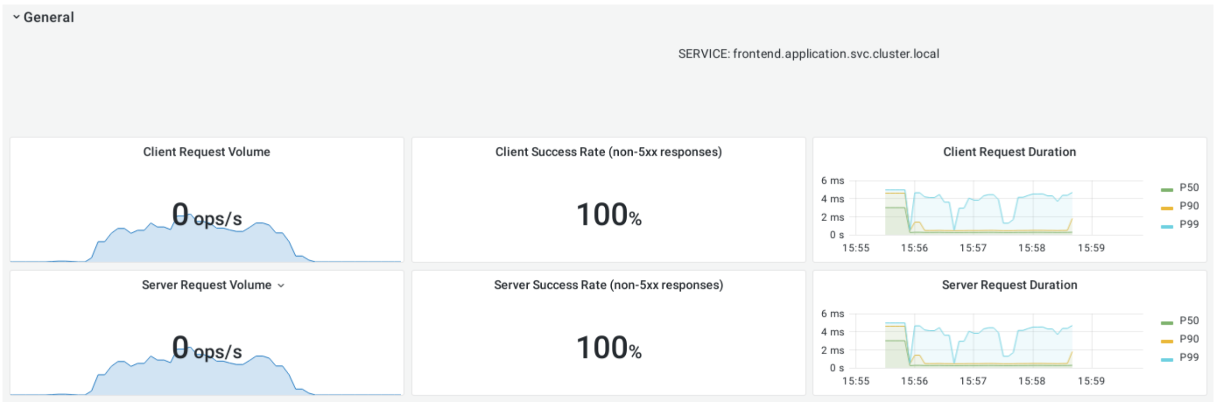


Figure 3.7: Some metrics on the running services

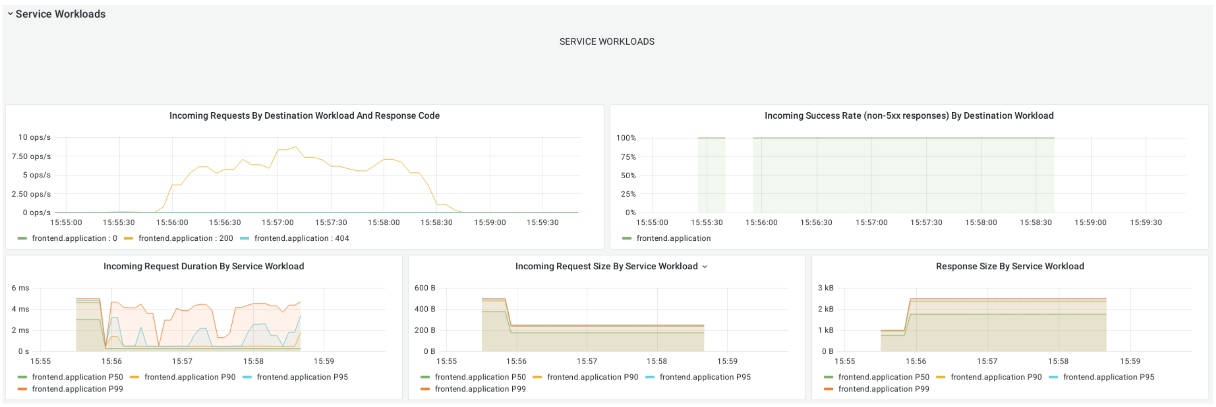

Service Workloads
SERVICE WORKLOADS
Incoming Requests By Destination Workload And Response Code
frontend.application : 0
frontend.application : 200
frontend.application : 404
Incoming Success Rate (non-5xx responses) By Destination Workload
frontend.application
Incoming Request Duration By Service Workload
Incoming Request Size By Service Workload
Response Size By Service Workload
frontend.application P50
frontend.application P90
frontend.application P95
frontend.application P99

Figure 3.8: Service workloads

3.2.4 Optimal components

We were given the choice to implement one of the following components for an extra percentage of the final grade:

1. Create and run a Continuous Integration(CI)/Continuous Deployment (CD) pipeline for the Web Micro-services-based Application, using for example Jenkins, in order to test, package and deploy the application.

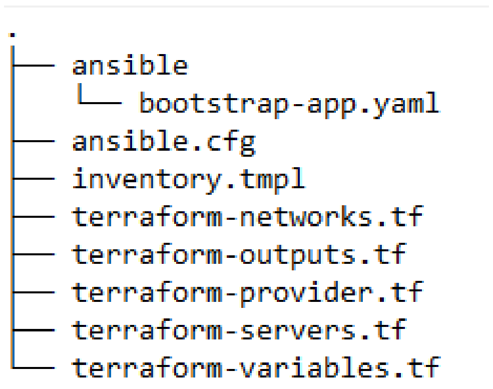
2. Deploy the Web Micro-services-based Application, with high availability, which requires Balancing and a database backend service with persistent storage.

3. Use orchestration tools such as Docker Compose, Nomad, Kubernetes, to *automatically* deploy, scale, have adequate networking (i.e., a Service Mesh ensuring a streamlined communication process in between the micro-services), and restart services if they are stopped.

The components 2 and 3 were implemented along the development of the project. The second component was fulfilled since the chosen application already had this characteristic. For the third component, as mentioned before, we use Kubernetes and for the final delivery we opted to use the ISTIO Service Mesh for observability, traffic management, security, and policing.

3.3 Project files

The project started by adding the following files to the previous base configuration and code of **Stan's Robot Shop** in **Checkpoint 1**:



```
.
├── ansible
│   └── bootstrap-app.yaml
├── ansible.cfg
├── inventory.tmpl
├── terraform-networks.tf
├── terraform-outputs.tf
├── terraform-provider.tf
├── terraform-servers.tf
└── terraform-variables.tf
```

Figure 3.9: File Tree Checkpoint I

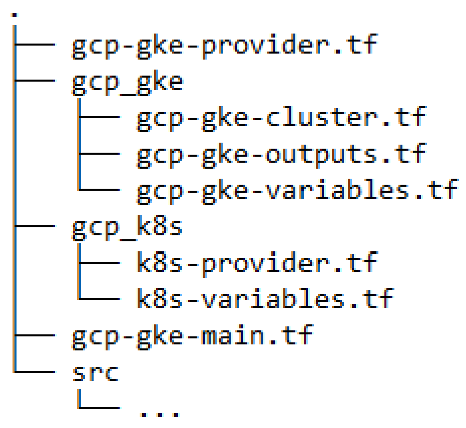
The terraform configuration was split across different files, according to the configuration purpose. Each file and it's purpose:

- **terraform-variables.tf** sets up variables that will be used in the rest of the configuration, to simplify future alterations that we may want to do; It also defines the list of services that our project will run
- **terraform-provider.tf** configures the google provider, by setting the project it will act on, the credentials to do so and the region it will act on
- **terraform-servers.tf** declares the instances that will be launched and their configuration. It launches one VM per service declared in the terraform-variables.tf file
- **terraform-networks.tf** declares the firewall rules to apply in the generated VPC: By default it blocks every connection, except:

1. ICMP messages
2. SSH from the outside of the VPC to the bastion host
3. SSH from the bastion to any machine inside the VPC
4. HTTP/S connections to the web machine. This rule was disabled at the same since we didn't have a web server ready yet

- **terraform-outputs.tf** automatically generates an Ansible inventory file according to inventory.tmpl template. Terraform uses the IP addresses created by the provider to populate the inventory file
- The ***ansible*** **directory** would contain the playbooks used in this project. At the time, we only had the bootstrap.yml playbook, which sets up the launched instances

For the **second Checkpoint** the following files were added:



```
.
├── gcp-gke-provider.tf
├── gcp_gke
│   ├── gcp-gke-cluster.tf
│   ├── gcp-gke-outputs.tf
│   └── gcp-gke-variables.tf
├── gcp_k8s
│   ├── k8s-provider.tf
│   └── k8s-variables.tf
├── gcp-gke-main.tf
└── src
    └── ...
```

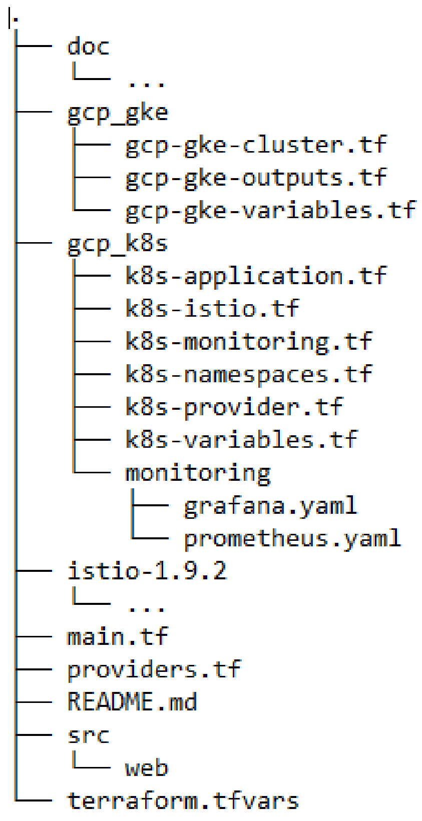
Figure 3.10: File Tree Checkpoint II

Each file and it's purpose:

- **gcp-gke-provider.tf** configures the google provider, setting up the credentials for the deployments
- **gcp_gke directory** contains the terraform module that declares and configures the GKE cluster. The *variables* file declares the variables the module will use; The *cluster* file creates the GKE cluster and the *outputs* file declares the values this module will output.

- **gcp_k8s directory** contains the terraform module that deploys the pods and services into the GKE cluster. The helm provider greatly simplifies this module! The *variables* file declares the values this module expects to receive and the *provider* file configures the Helm and Kubernetes providers and deploys the application in the GKE.
- **gcp-gke-main.tf** orchestrates the invocation of the modules mentioned above, guaranteeing that the pods and services are only deployed after a cluster is created and passes the K8s module the right credentials generated by the other one.
- The **src directory** contains the source code for the Stan's Robot Shop

Finally, we present in Figure 3.11 the final file structure of the project, corresponding to the Final Delivery. We explain the added files in detail below



```
|.
├── doc
│   └── ...
├── gcp_gke
│   ├── gcp-gke-cluster.tf
│   ├── gcp-gke-outputs.tf
│   └── gcp-gke-variables.tf
├── gcp_k8s
│   ├── k8s-application.tf
│   ├── k8s-istio.tf
│   ├── k8s-monitoring.tf
│   ├── k8s-namespaces.tf
│   ├── k8s-provider.tf
│   ├── k8s-variables.tf
│   └── monitoring
│       ├── grafana.yaml
│       └── prometheus.yaml
├── istio-1.9.2
│   └── ...
├── main.tf
├── providers.tf
├── README.md
├── src
│   └── web
└── terraform.tfvars
```

Figure 3.11: File Tree of the final delivery

- **istio-1.9.2** contains the Istio installation
- **gcp_k8s/application.tf** declares and deploys the application infrastructure (using Helm)

- **gcp_k8s/istio.tf** declares the Istio Service Mesh using Helm and the charts provided in the Istio installation directory;
- **gcp_k8s/monitoring** contains configuration files for Prometheus and Grafana. These files have information such as how to deploy each service, which dashboards to present to the users, from which sources to collect data, which services to probe, etc.
- **gcp_k8s/monitoring.tf** deploys the monitoring services according to their configuration present in the *directory*.
- **gcp_k8s/namespace.tf** declares the *istio-service* and *application* namespaces

Chapter 4

Pre-Requisites & Deployment

In this section we show, step by step, how to deploy our application. The design of our solution aimed at minimizing the effort in the *Deploy* phase, since it would be the most used one. We consider this objective completed since it only requires one command to deploy the entire application. A demo of all these steps can be found in a YouTube video we made for this purpose (`https://youtu.be/H4zqKkkNlzQ`)

4.1 Pre-Requisites

In order to deploy this projects infrastructure, there are some steps that need to be completed:

- Install **vagrant** in your machine. Head over to the Vagrant downloads page and get the appropriate installer or package for your platform. Install the package using standard procedures for your operating system.
- Create a project in your GCP account. For that purpose select on the top Menu Bar the Organization/Projects drop down button, that will open a window for selecting and/or creating a Project,
- Create a service account key for the project, to do so:
  - Go to the APIs & Services dashboard and enable the Kubernetes Engine API
  - Go to the IAM & Admin > Service Accounts page, select the default service account and then, on **Actions**, click on **Manage keys**;

- Click on **Add Key** > **Create new Key** and select a JSON. Then click **CREATE**. Make sure you save it on your project directory and it will not be shared in your repositories;

- Authorize Google Cloud SDK in your mgmt machine to access the GCP project, to achieve this:
  - **ssh** into the **mgmt** machine and go to the project directory (**cd /labs/project**)
  - Run **gcloud auth login**
  - Click on the outputted link and proceed with the authentication
  - Copy the code given after authentication and paste it in the terminal
  - Run **gcloud config set project** <**project_ID**>, replacing <**project_ID**> with the actual project ID

- Grant Kubernetes Engine Service Agent role to your service account:
  - Go to IAM & Admin > IAM
  - Select your service account (ending with *@developer.gserviceaccount.com*)
  - Click on the outputted link and proceed with the authentication
  - Click *Edit* and then on *Add Another Role* and select *Kubernetes Engine Service Agent* role
  - Finally, click on *Save*

- Then you should update the project variables by:
  - Create a new file named **terraform.tfvars** with the following contents, the region presented was the one chosen since it's the closest to our country:

```
project = ""
credentials_file = ""
workers_count = "3"
region = "europe-west4-a"
```

  - Set the **project** variable to your project ID

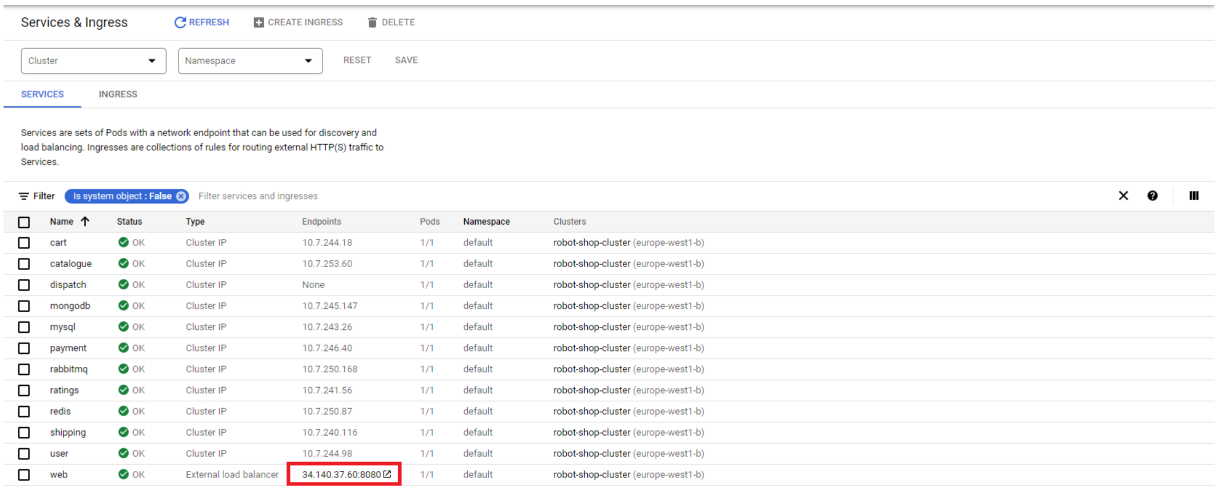
    - Set the **credentials_file** variable to the path to the key file your previously saved
    - Feel free to change the number of worker nodes in the cluster (**workers_count**) and the region (**region**) where the cluster will be deployed

- Run **terraform init**
- Install Istio, by running ”*curl -L https://istio.io/downloadIstio | ISTIO_VERSION=1.9.2 sh -*”

4.2 Deployment

As for the deployment of the project, which is final and essential step for the use of Stan’s Robot Shop with our added characteristics, it is necessary to:

- Run **terraform apply** to deploy the infrastructure and insert **yes** if you want to apply that plan
- Go to the GCP Project dashboard (make sure you have your project selected) and on the left side, select **Kubernetes Engine** > **Services & Ingress**
- Find the line that contains the **web** service and click on the IP present in the **Endpoints** column



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	Clusters
cart	✓ OK	Cluster IP	10.7.244.18	1/1	default	robot-shop-cluster (europe-west1-b)
catalogue	✓ OK	Cluster IP	10.7.253.60	1/1	default	robot-shop-cluster (europe-west1-b)
dispatch	✓ OK	Cluster IP	None	1/1	default	robot-shop-cluster (europe-west1-b)
mongodb	✓ OK	Cluster IP	10.7.245.147	1/1	default	robot-shop-cluster (europe-west1-b)
mysql	✓ OK	Cluster IP	10.7.243.26	1/1	default	robot-shop-cluster (europe-west1-b)
payment	✓ OK	Cluster IP	10.7.246.40	1/1	default	robot-shop-cluster (europe-west1-b)
rabbitmq	✓ OK	Cluster IP	10.7.250.168	1/1	default	robot-shop-cluster (europe-west1-b)
ratings	✓ OK	Cluster IP	10.7.241.56	1/1	default	robot-shop-cluster (europe-west1-b)
redis	✓ OK	Cluster IP	10.7.250.87	1/1	default	robot-shop-cluster (europe-west1-b)
shipping	✓ OK	Cluster IP	10.7.240.116	1/1	default	robot-shop-cluster (europe-west1-b)
user	✓ OK	Cluster IP	10.7.244.98	1/1	default	robot-shop-cluster (europe-west1-b)
web	✓ OK	External load balancer	34.140.37.60:8080	1/1	default	robot-shop-cluster (europe-west1-b)

Figure 4.1: GCP Project Dashboard Services

- You will enter in the Stan's Robot Shop website, will all the services available and at your disposal.

If you want to check the status of your cluster via command line, after having the cluster deployed, do the following:

- SSH into the **mgmt** machine and go to your project directory
- In the browser, go to the GCP console and select your project
- In the sidebar, go to **Kubernetes Engine** > **Clusters**
- In the **Actions** option of the target cluster, click **Connect**
- Copy the **gcloud** command given and run it in the **mgmt** machine

Chapter 5

Conclusion

We can conclude this project on a very positive note. It was the perfect way of gaining real insight on how to deploy micro-services based applications and what issues we may encounter during that process. More than ten different technologies were used in order to achieve all the goals we proposed: having a frontend, a backend service, a datastore and a monitoring service as part of a functional system. In summary, our application was deployed in a Kubernetes Clusters using GKE and the cluster with the working nodes was provided by Terraform. In the end, a ISTIO Service Mesh was added and Prometheus and Grafana were used as the monitoring services. With all this, we learned the value of IaC and monitoring. IaC is invaluable in cost reduction, achieving better speed through faster executing and finally in reducing risk by removing human error from the equation. On top of that, we learned that monitoring is useful to detect problems in the infrastructure and also to maintain efficiency. Finally, it's possible to state that the goal of deploying a micro-services based containerized Web Application on a Public Cloud provider using automation tools as well as implementing monitoring tools was achieved with success.

Bibliography

[1] Instana. (2021) Sample microservice application. [Online]. Available: https://github.com/instana/robot-shop/

[2] I. C. Education. (2019) Infrastructure as code. [Online]. Available: https://www.ibm.com/cloud/learn/infrastructure-as-code

[3] HashiCorp. (2020) Terraform. [Online]. Available: https://www.terraform.io/

[4] Google. (2021) Cloud computing services. [Online]. Available: https://cloud.google.com/

[5] R. Hat. (2021) Ansible is simple it automation. [Online]. Available: https://www.ansible.com/

[6] Docker. (2021) Empowering app development for developers — docker. [Online]. Available: https://www.docker.com/

[7] Helm. (2021) Helm. [Online]. Available: https://helm.sh/

[8] Kubernetes. (2021) Kubernetes. [Online]. Available: https://kubernetes.io/

[9] Prometheus. (2021) Prometheus - monitoring system time series database. [Online]. Available: https://prometheus.io/

[10] G. Labs. (2021) Grafana: The open observability platform. [Online]. Available: https://grafana.com/

[11] HashiCorp. (2021) Vagrant by hashicorp. [Online]. Available: https://www.vagrantup.com/

[12] Google. (2021) Securely connecting to vm instances. [Online]. Available: https://cloud.google.com/solutions/connecting-securely

[13] HashiCorp. (2021) Docs overview: Helm. [Online]. Available: https://registry.terraform.io/providers/hashicorp/helm/latest/docs

[14] NGINX. (2018) What is a service mesh? [Online]. Available: https://www.nginx.com/blog/what-is-a-service-mesh/

[15] Istio. (2021) Istio. [Online]. Available: https://istio.io/

[16] Instana. (2021) Instana - enterprise observability and apm for cloud-native applications. [Online]. Available: https://www.instana.com/