Project Report

Administration and Management of IT Services and Infrastructure

2023/2024

Afonso Bernardo 96834   João Sá 96878   Ricardo Rocha 96907

Video URL: `https://www.youtube.com/watch?v=JG_cQH59LFA`

Repository URL: `https://gitlab.rnl.tecnico.ulisboa.pt/agisit/agisit23-g13/`

1 Introduction

The goal of this project was to provision and deploy a tiered (frontend, backend) Microservices-based containerized Web Application on a public cloud provider using automation tools, and to have constant resource monitoring of the running containers. Our Web Application is a simple yet illustrative To-do List application that implements the fundamental CRUD (Create, Read, Update, Delete) operations on each of its Microservices.
We implemented both frontend and backend of our Web-Application in Python, the templating of the application using Docker, the provisioning and configuration of infrastructure using Terraform, deployment and orchestration using Google Kubernetes Engine and resource monitoring using Prometheus and Grafana using Istio as the implementation for the service mesh.

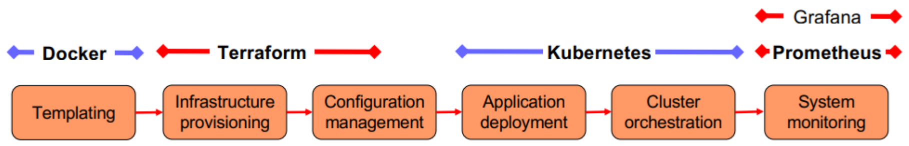


Figure 1: Overview of our automated deployment pipeline

2 Infrastructure Overview

Our infrastructure relies on the robust capabilities of Kubernetes to manage and orchestrate our services effectively. At the start of our architecture there is a Kubernetes Ingress which serves as the Load Balancer and routes user requests to the appropriate services.

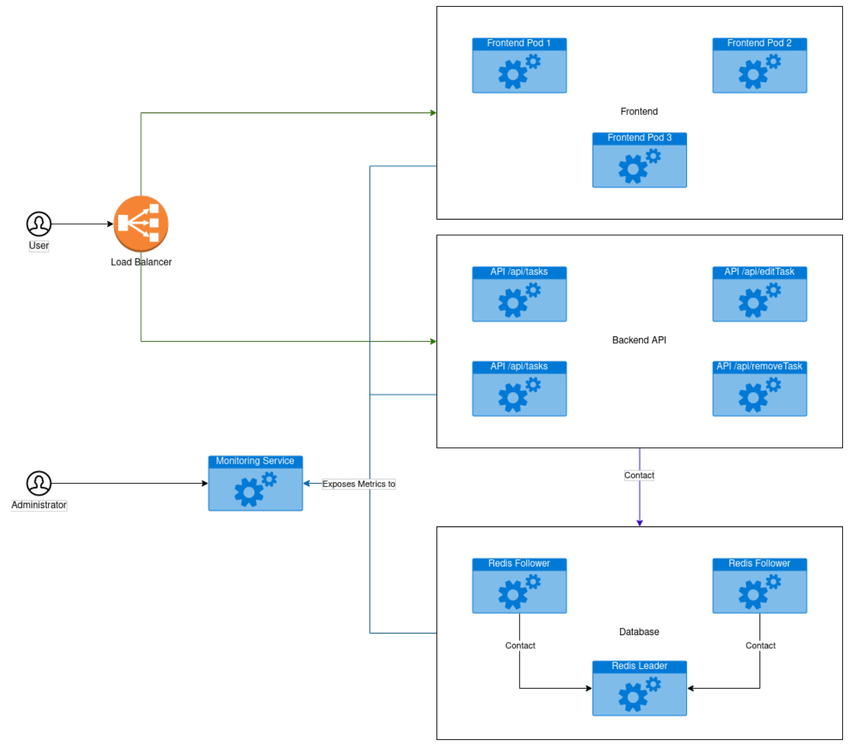
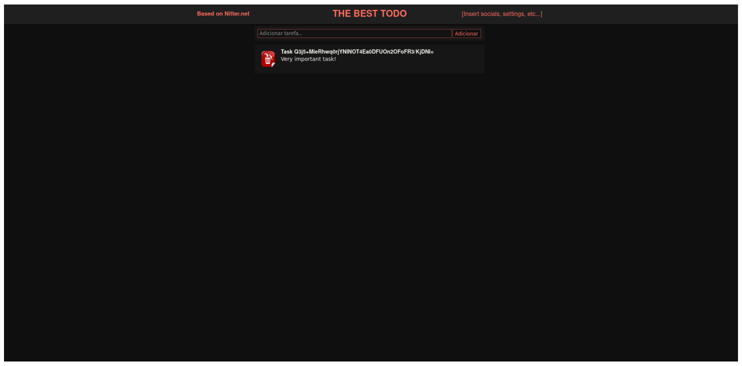


Figure 2: Overview of the Infrastructure - Microservices based Web-Application

- **LoadBalancer:** the entry point that redirects requests to our servers (frontend and backend) - implemented with **Kubernetes Ingress**
- **Frontend:** the replicas of our application that return the web application's front page (.html) - implemented as a **Flask** Web Server
- **Backend APIs:** replicated microservices that serve the application's requests - implemented as **Flask** Web Servers
- **Database:** stores our application's state - implemented with **Redis**
- **Monitoring Server:** enables observability of application and network resources used by the microservices - implemented with **Istio** service mesh, **Prometheus** to gather and **Grafana** to observe the metrics.

2.1 Frontend

The frontend page serves as the entry point for users to access our application. It provides the interface for various tasks and operations. The reason for using three frontend pods is that this is certainly the feature that will have the most amount of requests.



Task Q3j5+MieRhwq0rjYNINOT4Ea0DFUOn2OFoFR3/KjDNI=
Very important task!

Figure 3: Overview of our web application frontend with 1 task

This page makes use of html and javascript and with that, the user can proceed by creating several tasks, that each will generate a request (from the user side) to one of the backend API's endpoints.

2.2 Backend APIs

Our backend API comprises four distinct pods, each running a Flask server. These servers are designed to handle different types of requests, ensuring efficient workload distribution. Consequently, various requests are seamlessly redirected to the appropriate server, offering high performance and scalability. The idea is that every single service that is running is a very simple service, with just a few lines of code, making it really simple to debug and understand in case of failure or crash. All of the api services return data in the same format, json.
The API endpoints are:

- **/api/addTask:** This endpoint allows for a post request that will add a new task to the database.
- **/api/tasks:** This endpoint allows for a get request that will return a json with all the items in the database.
- **/api/editTask:** This endpoint allows for a post request that will edit a task, changing its description on the database.
- **/api/removeTask:** This endpoint allows for a post request that will remove a task from the database.

2.3 Database

Our database infrastructure is implemented using Redis, a distributed and replicated cache/database known for its speed and reliability.
All backend servers interact with Redis, ensuring data consistency and integrity across the system. Our system relies on a single Redis leader and two followers.

2.4 Monitoring Server

The monitoring server gives the system administrator a complete view over the application and network resources used by the microservices-based web application.
The way it is implemented is by having a side-car container in each of the pods, responsible for reporting the selected metrics to the Prometheus server. This side-car containers are connected through a service mesh implemented by Istio.
The metrics are available to be observed in a web-page provided by the Grafana web-server that was set up.

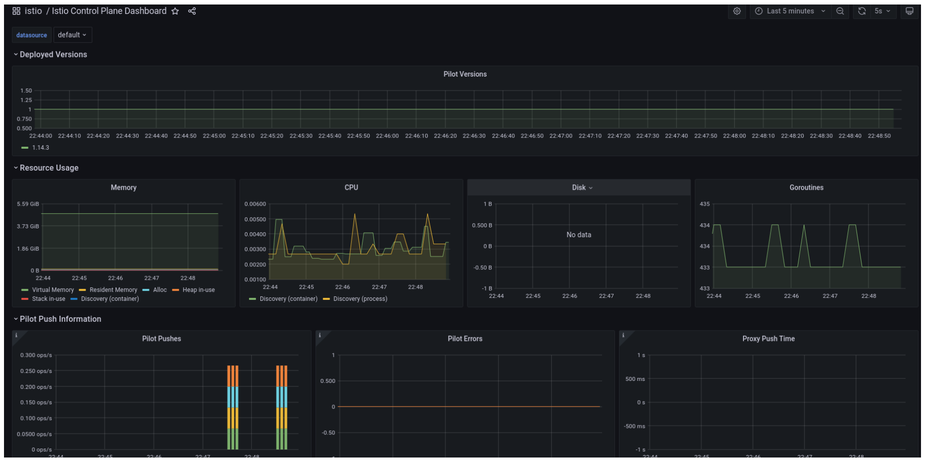


Figure 4: Overview of our Grafana dashboard for the Control Plane

3 Automated Deployment

All the infrastructure described above is automatically deployed using the pipeline presented in **Figure 1** from a special (virtual) machine residing on a local machine called 'Management Node'. It is running Ubuntu and is booted up using vagrant giving us the flexibility to deploy this infrastructure from any machine that is able to handle virtualization.
As soon as the Management Node is set up and accessible through ssh, we ssh inside it and run **terraform init** and **terraform apply** to provision and configure our infrastructure.

We chose to provision our infrastructure in Google Cloud Platform. When we run **terraform apply** the following steps occur:

1. The Docker images that represent the frontend and each of the microservices are built and pushed to Google Artifact Registry.
2. A Kubernetes Cluster is created with 3 nodes using Google Kubernetes Engine (GKE).
3. The Docker images are pulled from the Google Artifact Registry and the Pods running those images are set up (with a container that is respective to that image and one side car container used by the service mesh for service monitoring)
4. All the networking, load balancing, and ingress rules previously mentioned are set up.
5. Terraform will assign a duckdns domain to the load balancer, so that clients can always use the same domain even when IP changes. This domain is also used to have a TLS certificate on our website, so we can use HTTPS.
6. All the important information regarding the created cluster, such as the load balancer IP / DNS name is output to the ssh terminal.