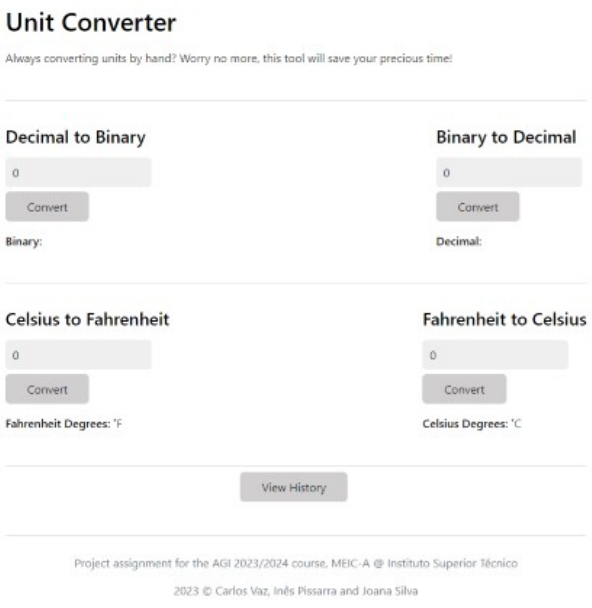
Capstone Project Report

Group: 05

Authors: Carlos Vaz (99188), Inês Pissarra (99236) and Joana Silva (99242)

---

1. Introduction



For this project we developed *Unit Converter,* which is a cloud-native app that is meant to be deployed to a Kubernetes cluster.

This application will help users convert:

- Decimal numbers to Binary and vice-versa;
- Celsius temperatures to Fahrenheit and vice-versa.

Additionally, users can also view their past operations history.

For the advanced component we chose the third option: Kubernetes as an orchestration tool.

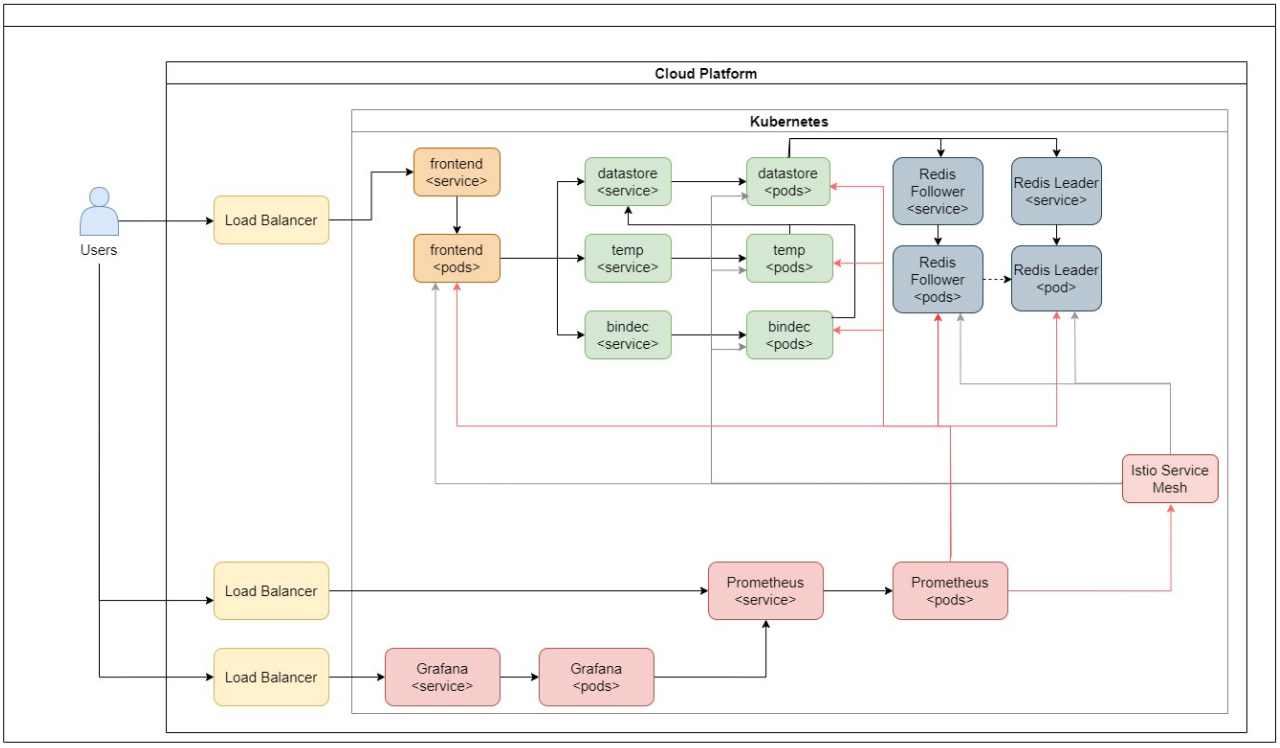
Video link: https://youtu.be/F7IJR_R9vBc

2. Architecture

Unit Converter is mainly composed of 4 microservices, all written in Python using the Flask framework and microservices for a Redis database:

- **Frontend:** is the web app entry point and exposes an HTTP server. Automatically generates an user ID and saves it as a cookie. Sends requests to *bindec* and *temp* to process conversions of user input, and requests the user's operation history from *datastore*.

- **Bindec:** has two endpoints, one for converting decimal numbers to binary and another for converting binary numbers to decimal. After executing the conversion, it makes a request to *datastore* to register the new operation.
- **Temp:** has two endpoints, one for converting Celsius temperatures to Fahrenheit and another for converting Fahrenheit temperatures to Celsius. After executing the conversion, it makes a request to *datastore* to register the new operation.
- **Datastore:** communicates with the Redis database microservices to store new user operations (write to *redis-leader*), and to retrieve the user's operation history (read from *redis-follower*).
- **Redis-leader and Redis-follower:** Redis database in a leader-follower configuration for persistent storage of the users' operation history. The Redis services store data in a dictionary where the keys are the user ids and the values are a list of operations that can be manipulated like a stack (the most recent operations are on top).



Our solution doesn't strictly follow the 'database-per-service' pattern in which every service should have their own database. In our case, that would mean that there should be a total of 3 databases: one for the user ids, one for the temperature operations and another one for the binary and decimal operations. However, that solution would increase the complexity and the traffic of the application, since the services would need to make more requests to retrieve information that was kept on the other databases which they don't own.

Instead, we only use one database that stores the whole data. To access the database, we created the *datastore* microservice, so that the other microservices can be abstracted from the *redis-leader* and *redis-follower* datastore.

(See chapters 5.2 and 6 for Istio Service Mesh, Prometheus and Grafana services)

3. Tools Used

The project contains the necessary files to deploy the app to a Kubernetes cluster on Google Cloud Platform's Google Kubernetes Engine, using Terraform. The app is deployed to a Kubernetes cluster with the Istio Service Mesh installed and configured. Prometheus and Grafana are also installed to monitor the cluster.

Terraform also builds the Docker images of the app's microservices and pushes them to Google Cloud's Artifact Registry.

Vagrant is used to launch a local Management Node VM that will be used to deploy the app to the Kubernetes cluster.

To stress the microservices we used the jmeter tool which we use in the video to demonstrate the autoscaling.

4. The Provisioning and Deployment Modules

The provisioning and deployment files can be found in the *gcp* directory and are divided in 2 modules:

- ***gcp_gke*** module specifies the definition of the kubernetes cluster which includes the region where the cluster is going to be deployed and the number of nodes per zone (in our case: 1 node per zone, 3 nodes in total). We only deployed the cluster in 1 region (europe-central2) since our credits only allow that. In theory it would be better to deploy the cluster in multiple regions in order to have the machines closer to the client;
- ***gcp_k8s*** module specifies the application configuration: pods, services, horizontal pod autoscalers. It also configures the deployment of Istio Service Mesh, together with Prometheus and Grafana monitoring tools.
    - ***k8s-provider.tf*** has a list of providers: *docker, helm, kubectl and kubernetes*;
    - ***k8s-docker.tf*** builds and pushes the images of the microservices written in python (frontend, bindec, temp and datastore) to a private Google Cloud repository using Docker as the provider;
    - ***k8s-namespaces.tf*** defines 2 namespaces:
        - ‘application’ for the application services;
        - ‘istio-system’ for Istio Service Mesh, Prometheus and Grafana.
    - ***k8s-pods.tf*** defines the pods to be deployed and the initial number of replicas (in this case, we only start with 1 replica for each microservice);
    - ***k8s-services.tf*** defines a set of services which include the microservices and datastore;
    - ***k8s-hpa.tf*** configures horizontal pod autoscalers. All services, except *redis-leader*, can have up to 3 replicas depending on CPU metric;

- ***k8s-istio.tf*** deploys the Istio Service Mesh;
- ***k8s-monitoring.tf*** and ***monitoring*** folder configure Prometheus targets and Grafana dashboards.

4.1 Deployment and StatefulSet

Since the application microservices are all stateless they are configured as ***Deployment*** type. The *redis-leader* is a ***StatefulSet,*** which means that a volume keeps all the data stored. If the *redis-leader* pod needs to restart, the next pod to be created will mount the volume of the previous pod, which means that the data will persist between restarts.

4.2 Services

To make requests to other services and to abstract to which replica the request will be forwarded to, we group the identical replicas inside a service. Then the requests will be routed to the service that will load balance the requests between the available replicas.

The *frontend, Prometheus* and *Grafana* services are of ***LoadBalancer*** type, which means that the clients connect to a load balancer that exposes an external endpoint which in turn forwards the request to the service. All the other services are by default ***ClusterIP*** which means that they are only accessible from inside the cluster.

5. Advanced Component

5.1 Horizontal Pod Autoscaler

We decided to use a simple raw metric as a criterion to auto scale (***target_cpu_utilization_percentage***). Therefore, to automatically scale the pods we configured the ***kubernetes_horizontal_pod_autoscaler_v1*** resource for each deployment.

The only service that doesn't scale is the *redis-leader* service, since it can't have more than one replica at a time, or otherwise, it would be necessary to sync the multiple *redis-leader* replicas to keep the data coherent.

The other application services can scale up to 3 replicas according to the CPU metric. If the average CPU that the replicas consume, within a particular service, is more than 80% of the requests resource configuration, then another replica will be automatically deployed. In our case that means that if the replicas use more than 80% of 0.2 CPU (0.1 CPU from microservice's container + 0.1 CPU from Istio sidecar container) then another replica is added to the service.

This configuration also enables automatic scale down. If the ***target_cpu_utilization_percentage*** is below 80%, then the number of replicas will decrease over time.

To check the CPU metric at real time one can run '*kubectl get hpa -n application*'.

The video submitted has a demonstration that showcases the auto scaling feature.

5.2 Istio Service Mesh

We used Istio Service Mesh which exposes an endpoint that creates metrics for each target component.

In the ***k8s-istio.tf*** file we configured the deployment of the Istio Service Mesh using Helm Charts. By using the service mesh in service-to-service communication, we can collect data and metrics, such as the CPU and memory usage, that can be used by Prometheus and Grafana to be presented in the dashboards.

It was necessary to create a new namespace, so that a sidecar container of the Istio Service Mesh could be automatically injected to all the target pods belonging to the 'application' namespace. Because of that, all the pods will have 2 containers running.

6. Monitoring

We used Prometheus to scrape the endpoints and collect data. Finally, Grafana was used to create dashboards to show the retrieved data.

In ***k8s-monitoring.tf*** we use the ***kubectl_manifest*** resource to apply the configurations in ***prometheus.yaml*** and ***grafana.yaml*** files.

In the ***prometheus.yaml*** file we configured 5 types of targets that specify where Prometheus will collect the metrics: apiservers (unit-converter cluster's private endpoint), nodes (the 3 nodes deployed), cadvisor (collects CPU, memory and network usage metrics for all containers running on the 3 nodes), pods (which include the application pods, but also the Istio Service Mesh pod) and prometheus. All of these targets are needed in order to retrieve the statistics that are going to be presented in the grafana dashboards.

In the ***grafana.yaml*** file there are 5 different dashboards declared to show multiple metrics collected by Prometheus.

The **Istio Workload Dashboard** is the most important dashboard where we can check the inbound and outbound services that communicate with a particular service. We can see the incoming/outgoing requests (inside Inbound Workloads and Outbound Services sections) to verify the success rate, the services daily usage pattern and the average duration of the requests. All of these metrics are relevant to identify if a service is overloaded.

The **Istio Service Dashboard** complements the previous dashboard by showing the global incoming requests instead of dividing by service. If we just want the global requests statistics, such as the volume of requests and the success rate of the entire service mesh, then the **Istio Mesh Dashboar**d provides those metrics.

With the **Istio Performance Dashboard** we can monitor the sidecars proxy that were injected in the application pods and the Istio Control Plane to check the average memory and CPU usage.

Finally, the **Istio Control Plane Dashboard** presents data related to the Istio container which includes: the CPU and memory usage, the number of errors and the number of sidecar injections. With these metrics we can identify if something is wrong with the service mesh control plane.