ASSIGNMENT 10 – RISULTATI AGGIORNATI
MAIN TOPICS: KUBERNETES, Helm, Istio, Prometeus, Grafana

MODULE 1: HELM

I've created a virtual machine called LAB12vm in order to perform the overall activity without interfering with other projects.
After having installed all the packages:

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-4 |
VERIFY_CHECKSUM=false bash
```

```
helm version

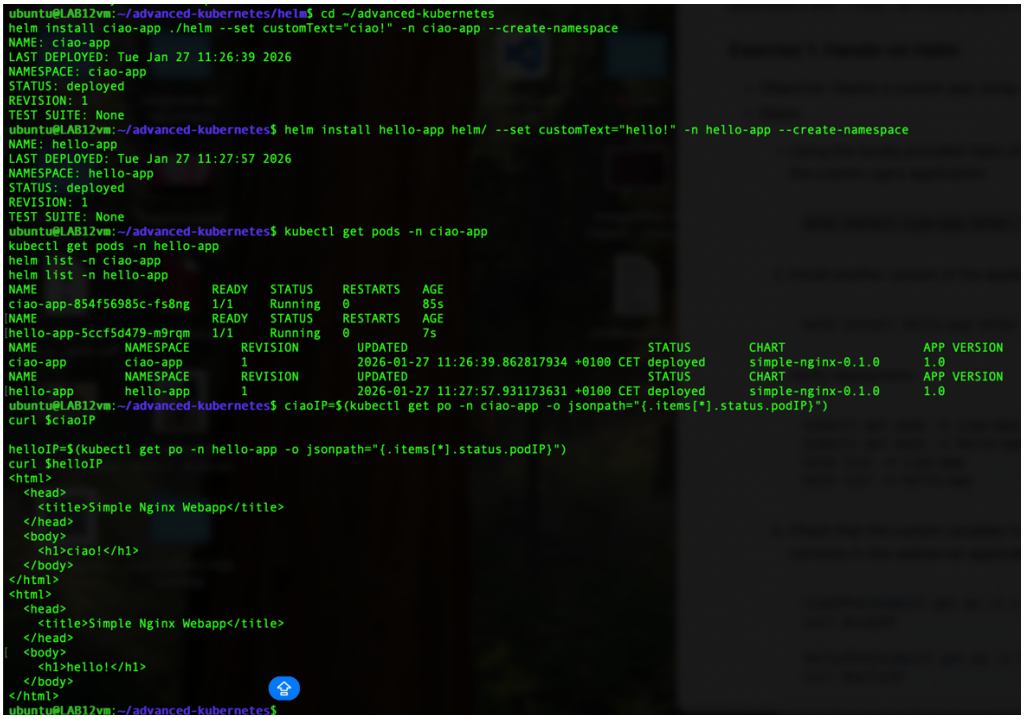
helm install ciao-app helm/ --set customText="ciao!" -n ciao-app --create-
namespace
```

If there's an error in the installation I have to run the following commands:

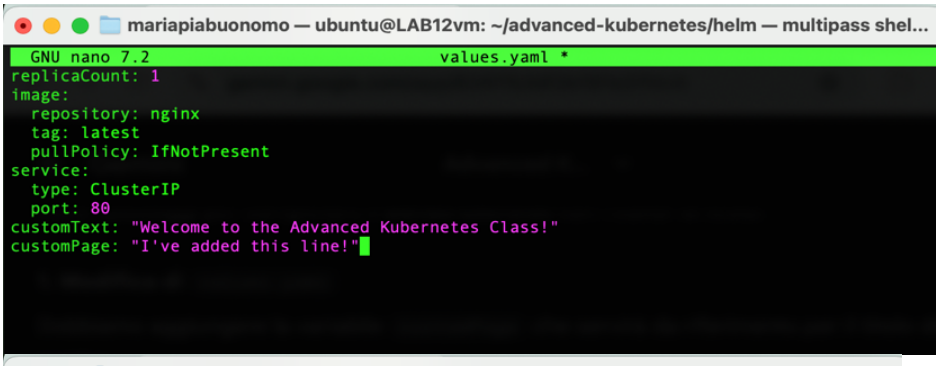
`sudo k0s kubeconfig admin > ~/.kube/config` HERE WE EXTRACT THE ADMINISTRATIVE CREDENTIALS GENERATED BY K0S TO SAVE THEM IN A CONGIF FILE IN THE HIDDEN DIRECTORY .KUBE (IN THE HOME DIRECTORY) IN ORDER TO LET KUBECTL AND HELM TALK TO THE CLUSTER

`chmod 600 ~/.kube/config` THIS MAKES THE PERMISSION SO THAT MY USER CAN READ OR WRITE THAT FILE.

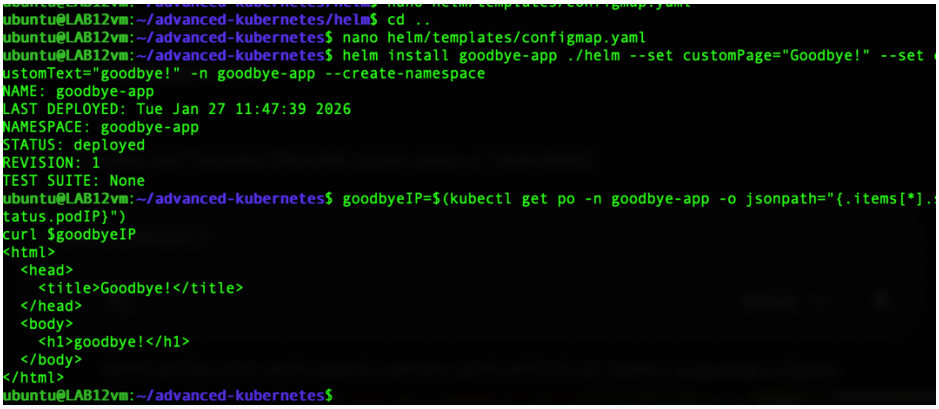
At first I create a RELEASE based on a CHART with the parametrization ("—set ..."). This, by overwriting in the values.yaml default file with a personalized text without modifying the code of the Chart manually.
Creation of the namespace ("-n" ecc...) to isolate the resources and avoid conflicts.
Then, verification of the status and pods.
With "jsonpath..." I extract the IP address inside the pod directly from the API of Kubernetes (advanced way to filter the JSON data); the "curl" after demonstrates that the ConfigMap generated by Helm has injected the personalized text correctly in the web server Nginx.



To perform the Bonus exercise in Module 1 on Helm, I enter the file *helm/values.yaml* to include a new line *customPage*. And then I connect the variable to the HTML template of ConfigMap.



And finally I have installed the Goodbye application:



Then I have uninstalled the packages as suggested by the Exercise text.

```
helm uninstall -n ciao-app ciao-app
helm uninstall -n hello-app hello-app
helm uninstall -n goodbye-app goodbye-app
kubectl delete ns ciao-app
kubectl delete ns hello-app
kubectl delete ns goodbye-app
```

MODULE 2: ISTIO

I have installed Istio, Prometheus and Grafana as requested by the exercise.

```
curl -L https://istio.io/downloadIstio | ISTIO_VERSION=1.28.0 sh - # Downloads and extracts the specified Istio version.
export PATH="$PWD/istio-1.28.0/bin:$PATH" # Adds Istio binaries to your PATH for easy access.
istioctl install --set profile=minimal -y # Installs Istio with the minimal profile, suitable for labs and quick demos.
```

```
kubectl get pods -n istio-system
```

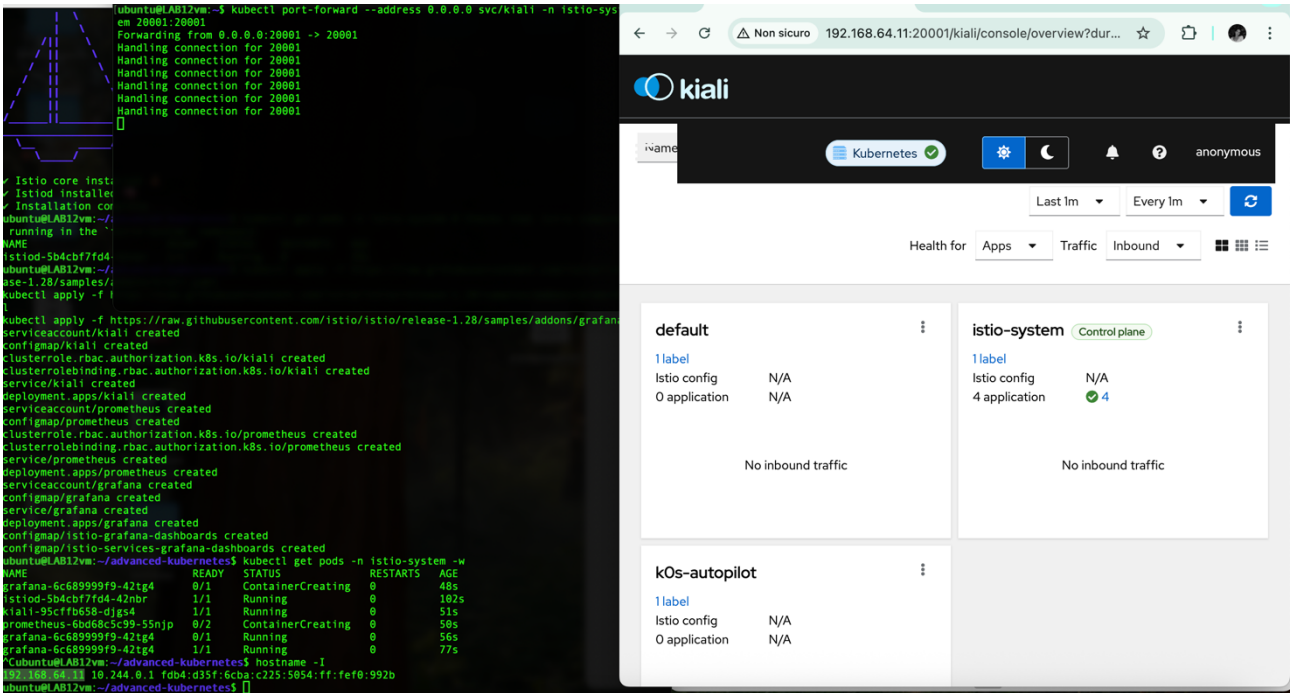
```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.28/samples/addons/kiali.yaml
kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.28/samples/addons/prometheus.yaml
kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.28/samples/addons/grafana.yaml
```

```
ubuntu@LAB12vm:~/advanced-kubernetes$ kubectl get pods -n istio-system -w
NAME                          READY   STATUS              RESTARTS   AGE
grafana-6c689999f9-42tg4      0/1     ContainerCreating   0          48s
istiod-5b4cbf7fd4-42nbr       1/1     Running             0          102s
kiali-95cffb658-djgs4         1/1     Running             0          51s
prometheus-6bd68c5c99-55njp   0/2     ContainerCreating   0          50s
grafana-6c689999f9-42tg4      0/1     Running             0          56s
grafana-6c689999f9-42tg4      1/1     Running             0          77s
^Cubuntu@LAB12vm:~/advanced-kubernetes$
```

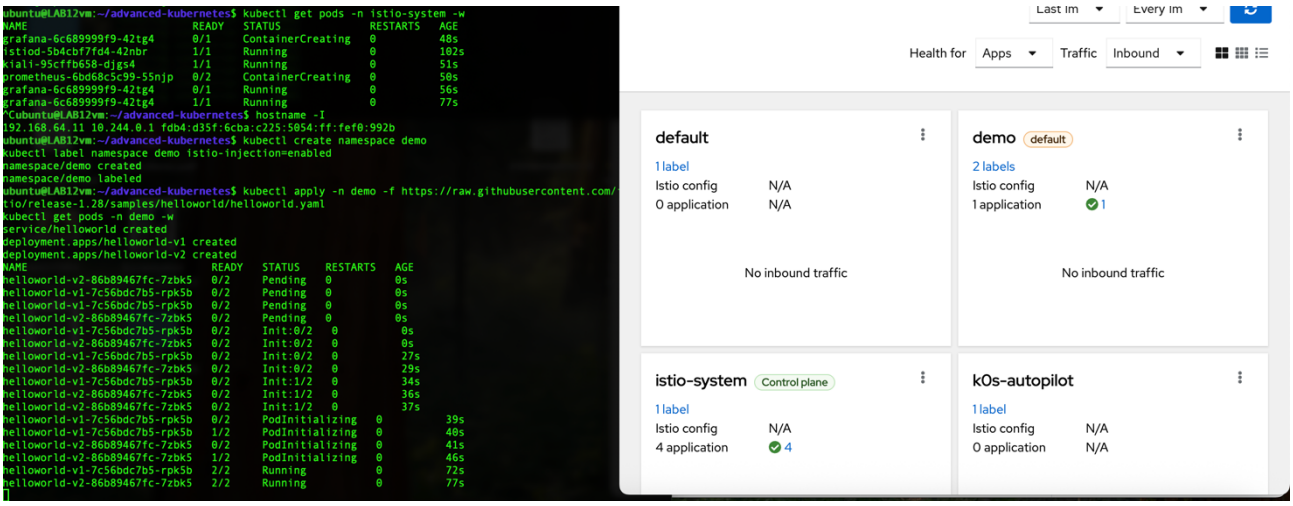
I have checked which was my IP address for my VM with :

hostname -I

Discovered that it was 192.168.64.11. So I opened another terminal (up on the left in the photo) with that command that temporarily forwards the Kiali service to my machine.
Then I went on the browser: http:// 192.168.64.11:20001 to open the Kiali Dashboard.



Then I have created the demo namespace with the helloworld sample app in it. The sample app (`helloworld.yaml`) is a simple HTTP service used for demonstration and testing Istio features.

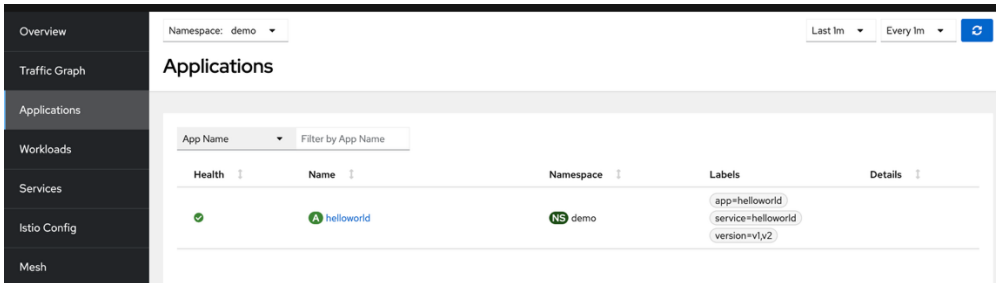


Each pod should have two containers: the application and the istio-proxy, The Istio sidecar (Envoy) has been injected automatically! Istio uses automatic sidecar injection to add an Envoy proxy container to each pod in a labeled namespace. This enables traffic management, security, and observability features for all services in that namespace.

```
kubectl describe pod –n demo | grep –E "container.*istio–proxy"
```

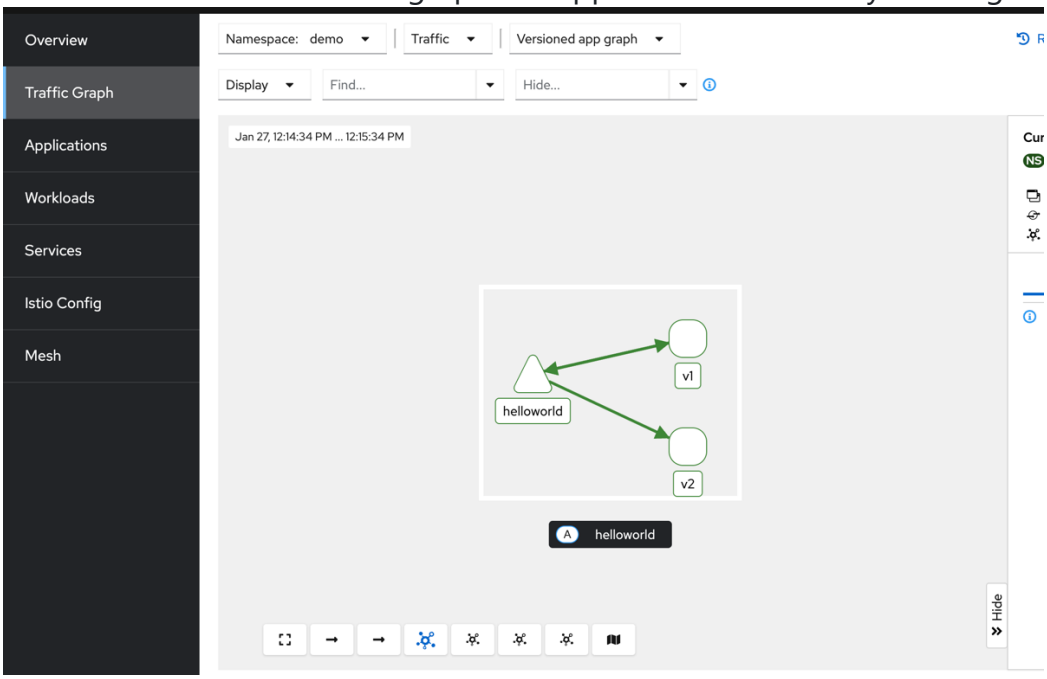
I am using this command to validate the automatic sidecar injection. Since I labeled the namespace with istio-injection=enabled, Istio intercepts the Pod creation and adds an Envoy proxy (istio-proxy). Without this container, the Pod would not be part of the Service Mesh, and I would be unable to monitor traffic or apply security policies like mTLS.
SO: Istio has injected automatically a second container (Proxy Envoy) inside the same pod.

Now in the Kiali dashboard we can see in Applications the helloworld correctly created.

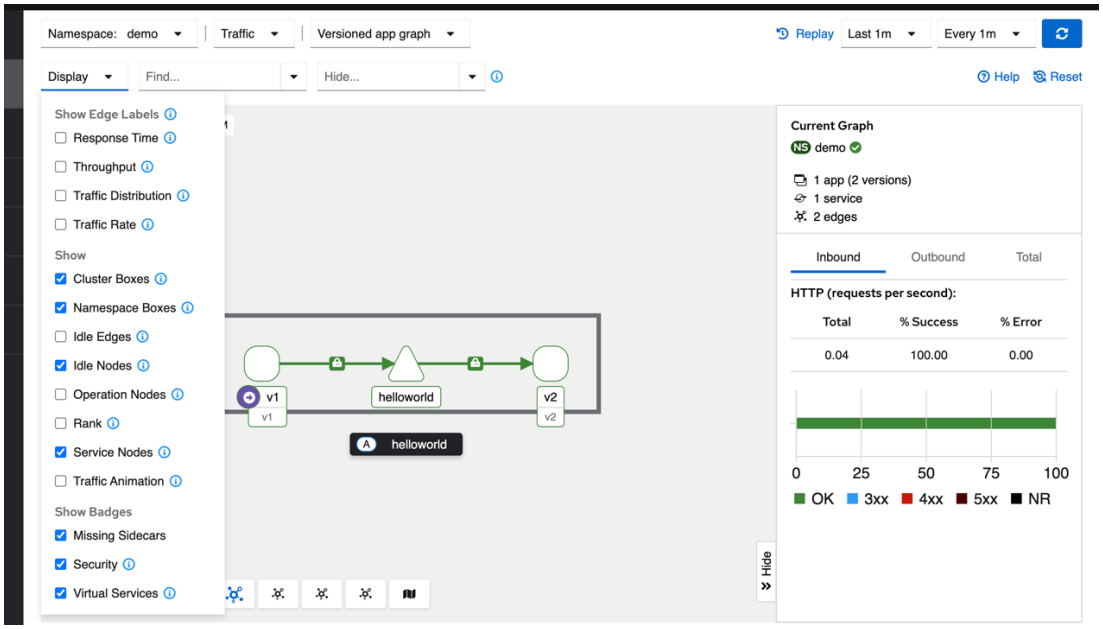


And if I generate traffic with

```
istio/helloworld–traffic.sh
```

I can see from Kiali that the traffic graph has appeared and correctly working with green links. 

Then, if we enable the security mTLS given by Istio with **kubectl apply -f istio/mtls.yaml**
And reload the application and the traffic with:
kubectl rollout restart deployment -n demo helloworld-v1 helloworld-v2
istio/helloworld-traffic.sh



The PeerAuthentication policy configures Istio to require mTLS for all workloads in the namespace.

Now let's try to elaborate complex application traces:

(form the text...)

The travel demo application consists of several microservices deployed across three namespaces. Label each namespace to enable automatic Envoy sidecar injection for observability and traffic management.

kubectl create namespace travel-agency

kubectl create namespace travel-portal

kubectl create namespace travel-control

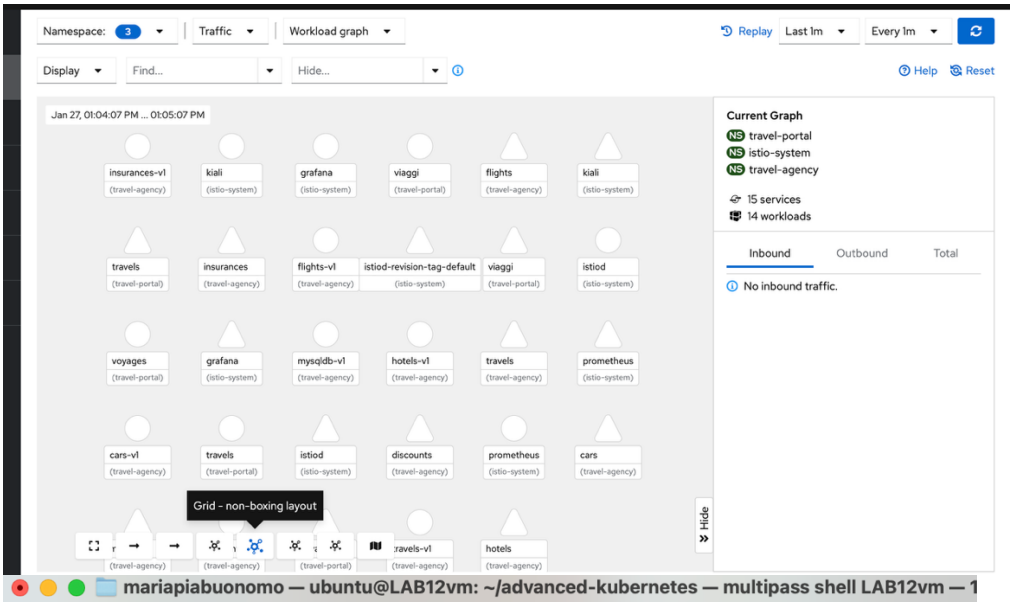
kubectl label namespace travel-agency istio-injection=enabled

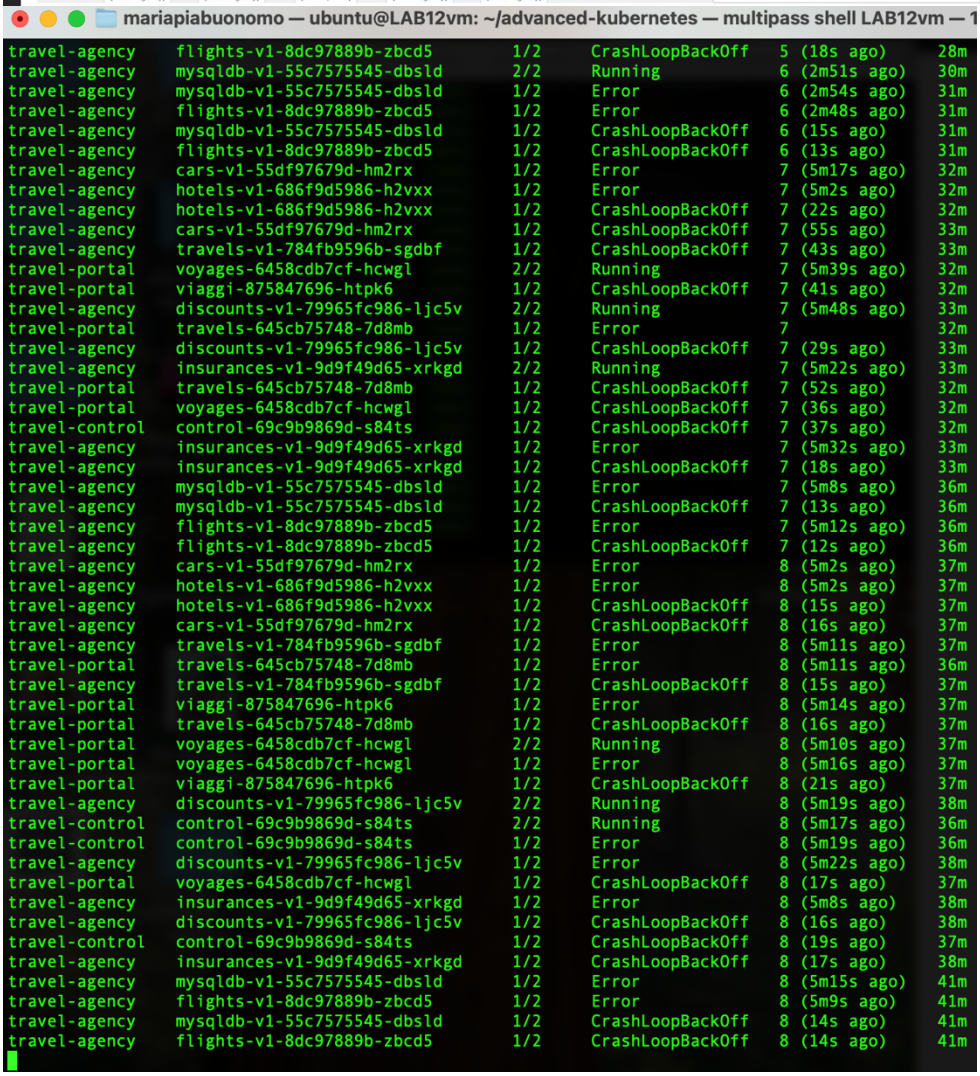
kubectl label namespace travel-portal istio-injection=enabled

kubectl label namespace travel-control istio-injection=enabled

Apply the manifests for each part of the application. These will create multiple interconnected services for a realistic microservices scenario.

```
kubectl apply -f <(curl -L
https://raw.githubusercontent.com/kiali/demos/master/travels/travel_agency.yaml)
-n travel-agency
kubectl apply -f <(curl -L
https://raw.githubusercontent.com/kiali/demos/master/travels/travel_portal.yaml)
-n travel-portal
kubectl apply -f <(curl -L
https://raw.githubusercontent.com/kiali/demos/master/travels/travel_control.yaml
) -n travel-control
```





I had to reload the overall VM because the quantity of logs to run is very high so it gives me the CrashLoopBackOff error, signi of very small memory in my VM so that the processing is very difficult to perform for my PC.

MODULE 3: TELEMETRY

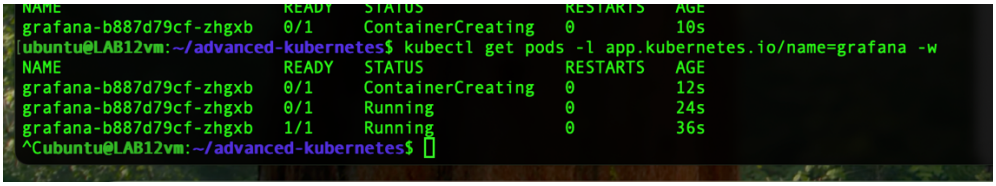
I have installed Grafana and Prometheus using Helm as the guide suggested. Prometheus was successfully installed but Grafana gave me a CrashLoopBack and, by checking the logs with:

kubectl logs grafana-7f4875785c-z7pkr

kubectl describe pod grafana-7f4875785c-z7pkr

I performed a clean reinstallation because the Exit Code 139 suggested a segmentation fault or binary incompatibility. By redeploying the chart and disabling persistence, I ruled out potential conflicts with the VM's file system or corrupted configuration states.
The latest Grafana image (12.3.1) was triggering a Segmentation Fault (Exit Code 139) due to an instruction set incompatibility within the emulated ARM64 environment of this VM. I resolved this by performing a downgrade to version 10.4.0 and explicitly defining Resource Requests, ensuring the container starts with a stable binary and guaranteed memory allocation.

```
helm install grafana grafana/grafana \
 --set image.tag=10.4.0 \
 --set adminPassword=admin \
 --set service.type=NodePort \
 --set service.nodePort=30405 \
 --set persistence.enabled=false \
 --set resources.requests.memory=256Mi \
 --set resources.requests.cpu=100m
```



In order to retrieve how to access to Prometheus and Grafana I have applied the following commands suggested by the guide:

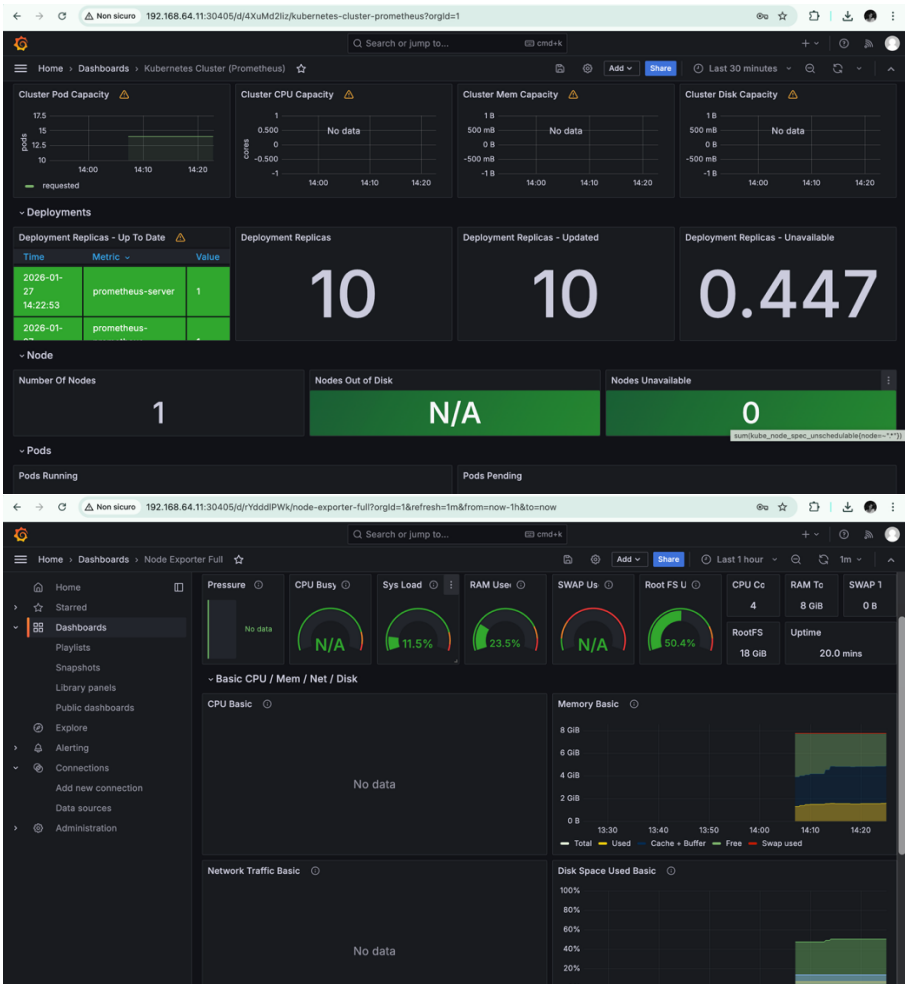
```
export PNODE_PORT=$(kubectl get --namespace default -o
jsonpath="{.spec.ports[0].nodePort}" services prometheus-server)
export PNODE_IP=$(kubectl get nodes --namespace default -o
jsonpath="{.items[0].status.addresses[0].address}")
echo "Prometheus UI: http://$PNODE_IP:$PNODE_PORT"

# Get the Grafana URL to visit by running these commands in the same shell:
export GNODE_PORT=$(kubectl get --namespace default -o
jsonpath="{.spec.ports[0].nodePort}" services grafana)
export GNODE_IP=$(kubectl get nodes --namespace default -o
jsonpath="{.items[0].status.addresses[0].address}")
echo "Grafana UI: http://$GNODE_IP:$GNODE_PORT"
```

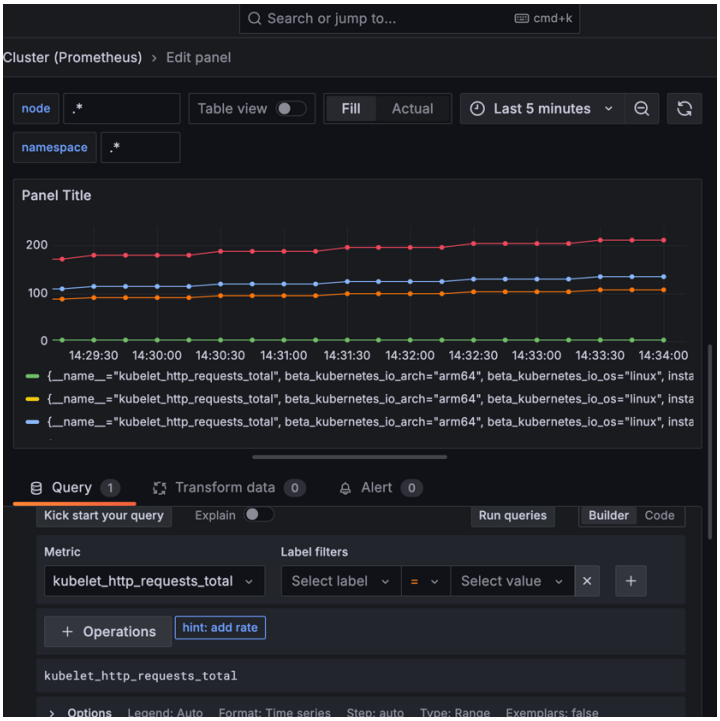
Prometheus UI: http://192.168.64.11:30303
Grafana UI: http://192.168.64.11:30405

So I open Grafana with that link and add Prometheus as a Data Source in Connection>data sources>Add data source and by pasting the previous link given by Prometheus.

Then, I have added with their JSON file the two dashboards: the Cluster monitoring and the Node Exporter Full.



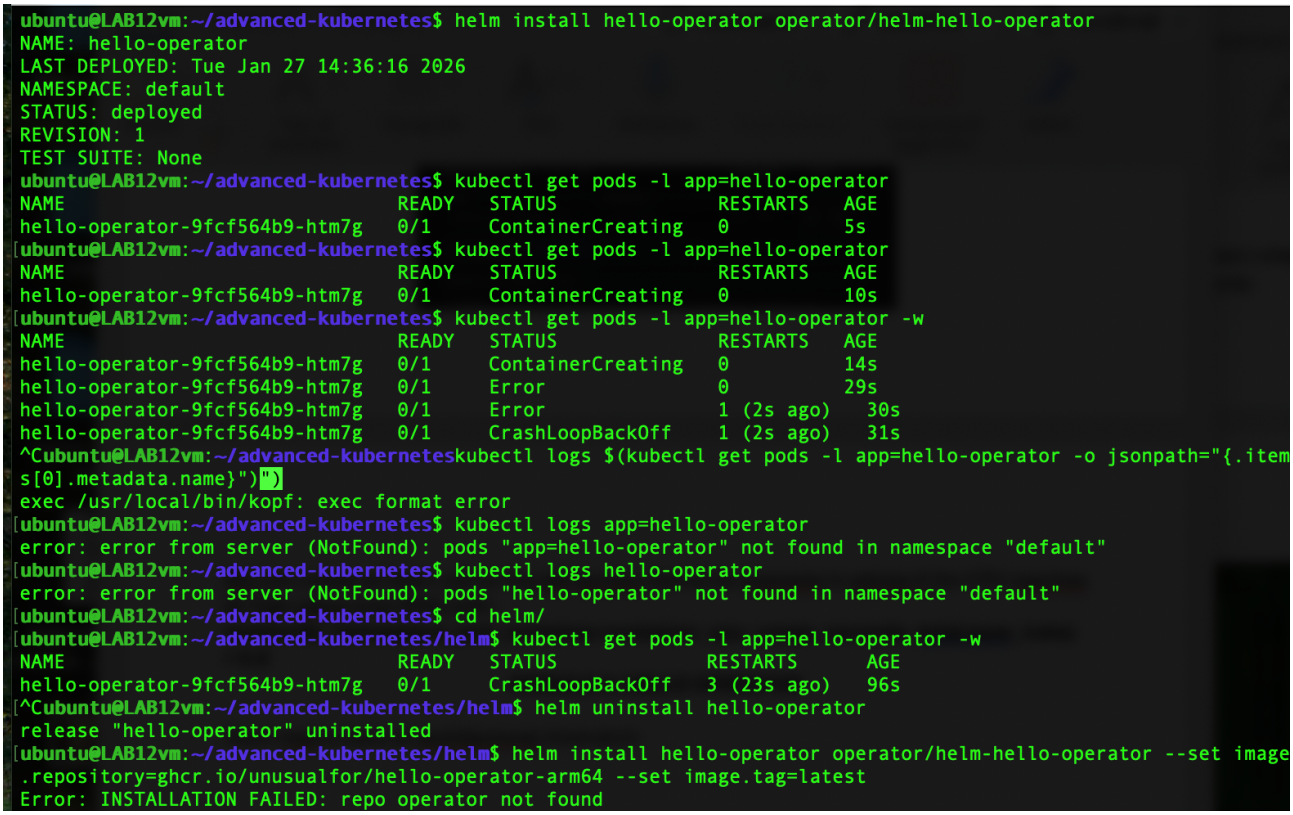
Then, if we go to Add (in alto a destra) and Visualization, we can add some metrics. For example how many times we used curl, for KUBELET**: kubelet_http_requests_total**.



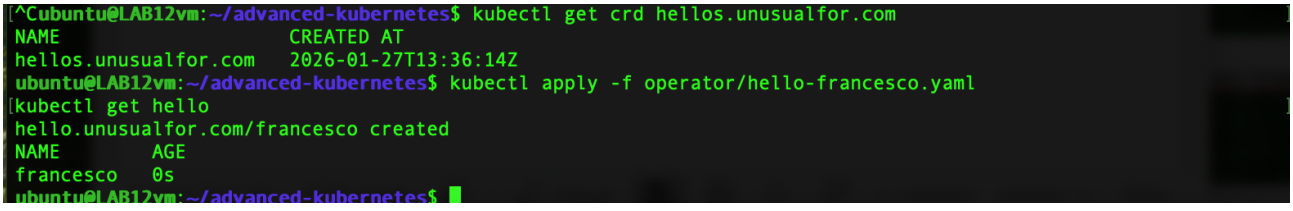
We can create an Alert rule (in the left section Alerts) for example to advise if the CPU reaches an high value (over 80%) with:
sum(node_namespace_pod_container:container_cpu_usage_seconds_total:sum_irate) > 0.8

MODULE 4: HELLO OPERATOR

Here I had to face an architectural mismatch:



In fact, by looking at the pods I'm facing a problem between arm64 and x86_64. even though the Operator pod is currently crashing due to architecture incompatibility I have successfully extended the Kubernetes API. By running kubectl get crd, I can show that the cluster now 'understands' what a Hello resource is. I have applied a Custom Resource named francesco, and Kubernetes has stored it in its database (etcd).



HOW DID I PERFORM THE TROUBLESHOOTING FOR THIS PART:

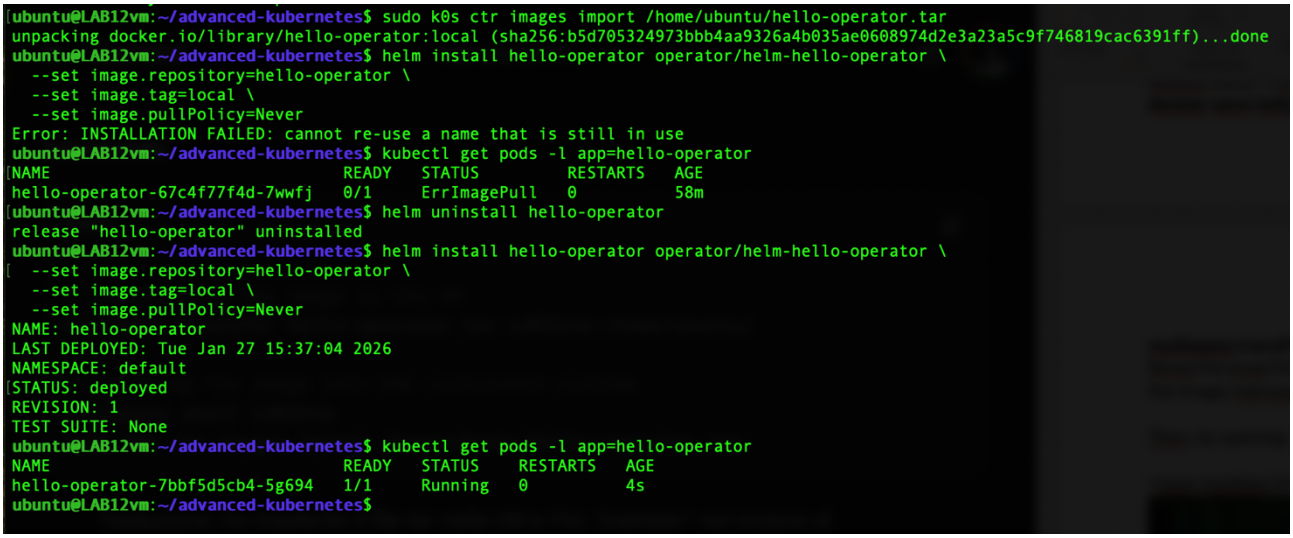
The default container image provided in the lab was compiled for x86_64, causing an Exec format error on my MacBook Air (M3). I performed a local build to ensure the binary was compatible with the ARM64 architecture of my host and VM.

In my normal terminal:

cd ~/advanced-kubernetes/operator/app
docker build -t hello-operator:local .
docker save hello-operator:local > hello-operator.tar

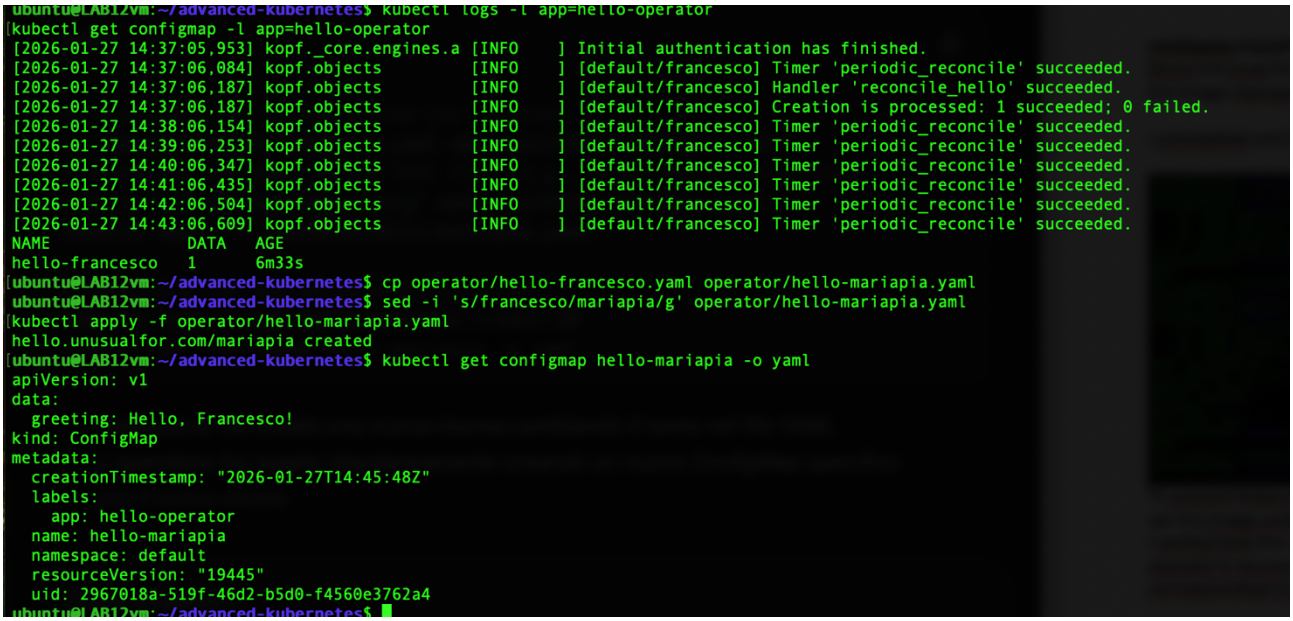
multipass transfer hello-operator.tar secondaVM:/home/ubuntu/
Since the local Docker image was not available inside the isolated Multipass VM, I transferred the image manually and imported it directly into the containerd runtime of the k0s cluster.

I uninstalled and installed another time helm hello operator:



To prevent Kubernetes from trying to pull the incompatible image from the remote registry, I set the image.pullPolicy to Never, forcing the cluster to use the local sideloaded image.
I verified that the operator was running correctly. Upon applying a Custom Resource (CR), the operator's reconciliation loop successfully detected the event and automatically created a corresponding ConfigMap.

I tried to create a new resource from the template:



I modified the name of the YAML file and see the final output of the new ConfigMap specific for my name (new user).
I have deleted all to free my previous environment. The docker image too from my PC:
helm uninstall hello-operator

kubectl delete crd hellos.unusualfor.com
docker rmi hello-operator:local

And from the VM:
kubectl get configmap -l app=hello-operator
helm uninstall hello-operator
kubectl delete crd hellos.unusualfor.com
rm /home/ubuntu/hello-operator.tar
sudo k0s ctr images rm docker.io/library/hello-operator:local
****

ASSIGNMENT

Assignment: Telemetry

Objective: Deploy a custom telemetry app, ensure Prometheus scrapes its metrics, and create a Grafana dashboard for visualization and analysis.

Prerequisites:

- Prometheus and Grafana are installed and running as per Module 3
- You have access to your cluster via kubectl and Helm

Steps:

1. **Deploy the telemetry app using Helm in the assignment namespace:**
    - This command installs the telemetry app using the provided Helm chart, creating the assignment namespace if it does not exist.

```
helm install assignment-app oci://ghcr.io/unusualfor/demo-app -n assignment --create-namespace
```

2. **Verify the app is running in the assignment namespace:**
    - Check that the pod and service for the demo app are present and running.
3.
```
kubectl get pods -n assignment -l app=demo-app
kubectl get svc -n assignment demo-app
```

4. **Test metrics endpoint:**
    - Retrieve the ClusterIP of the demo-app service:

```
CLUSTER_IP=$(kubectl get svc -n assignment demo-app -o jsonpath='{.spec.clusterIP}')
```

    - Access the metrics endpoint exposed by the app:

```
curl http://$CLUSTER_IP:8000/metrics
```

- You should see Prometheus-formatted metrics output. If not, check the pod logs and service configuration.

5. **Check Prometheus targets:**

   - Prometheus is exposed via NodePort. Retrieve the Prometheus server URL:

   - ```
     export PNODE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].nodePort}" services prometheus-server)
     ```
   - ```
     export PNODE_IP=$(kubectl get nodes --namespace default -o jsonpath="{.items[0].status.addresses[0].address}")
     echo "Prometheus UI: http://$PNODE_IP:$PNODE_PORT"
     ```
   - Query Prometheus targets to confirm your app appears in activeTargets and is up:
     ```
     curl http://$PNODE_IP:$PNODE_PORT/api/v1/targets | jq . | grep assignment
     ```

     or just access the following and filter per *assignment*:

     ```
     echo "Prometheus UI - Targets: http://$PNODE_IP:$PNODE_PORT/targets"
     ```
   - If your app does not appear or is not up, check the app's service annotations and Prometheus scrape configuration.

6. **Build a dashboard in Grafana:**

   - After confirming metrics are available, create a custom dashboard or panel in Grafana using the metrics exposed by your app (e.g., request count, latency, memory usage). Check again with

7. ```
   CLUSTER_IP=$(kubectl get svc -n assignment demo-app -o jsonpath='{.spec.clusterIP}')
   curl http://$CLUSTER_IP:8000/metrics
   ```

   which are the available metrics and use them appropriately (hint: some of them start with *demo_app*).

   - Example tasks:

     - Visualize request rate over time
     - Show average or maximum latency
     - Display current memory usage

   - Use Prometheus as the data source and select your app's metrics for visualization.

   - Make sure to create multiple panels and, for each panel itself try to relate some metrics together (e.g. how many requests were created and which was the request latency, shown in the same panel).
   ```

As in the previous operation part, because of mismatch of architectures with the creation of the new demo app, I had to create an image on my personal docker and copy it on the VM in this way:

```
Host-006:esercizidistr mariapiabuonomo$ cd advanced-kubernetes/
Host-006:advanced-kubernetes mariapiabuonomo$ docker build -t demo-app:local .
[+] Building 0.0s (1/1) FINISHED                                                docker:desktop-linux
 => [internal] load build definition from Dockerfile                                           0.0s
 => => transferring dockerfile: 2B                                                             0.0s
ERROR: failed to build: failed to solve: failed to read dockerfile: open Dockerfile: no such file or directory
Host-006:advanced-kubernetes mariapiabuonomo$ cd app
-bash: cd: app: No such file or directory
Host-006:advanced-kubernetes mariapiabuonomo$ ls
assignment      helm            infra           istio           operator        README.md
Host-006:advanced-kubernetes mariapiabuonomo$ cd assignment/
Host-006:assignment mariapiabuonomo$ ls
app     helm
Host-006:assignment mariapiabuonomo$ cd app/
Host-006:app mariapiabuonomo$ docker build -t demo-app:local .
[+] Building 6.5s (10/10) FINISHED                                              docker:desktop-linux
 => [internal] load build definition from Dockerfile                                           0.0s
 => => transferring dockerfile: 152B                                                           0.0s
 => [internal] load metadata for docker.io/library/python:3.10-slim                            1.8s
 => [auth] library/python:pull token for registry-1.docker.io                                  0.0s
 => [internal] load .dockerignore                                                              0.0s
 => => transferring context: 2B                                                                0.0s
 => [1/4] FROM docker.io/library/python:3.10-slim@sha256:f5d029fe39146b08200bcc73595795ac19b85997ad0e5001a0  0.0s
 => => resolve docker.io/library/python:3.10-slim@sha256:f5d029fe39146b08200bcc73595795ac19b85997ad0e5001a0  0.0s
 => [internal] load build context                                                              0.0s
 => => transferring context: 901B                                                              0.0s
 => CACHED [2/4] WORKDIR /app                                                                  0.0s
 => [3/4] COPY app.py .                                                                        0.0s
 => [4/4] RUN pip install prometheus_client psutil                                             4.2s
 => exporting to image                                                                         0.4s
 => => exporting layers                                                                        0.3s
 => => exporting manifest sha256:2a65a547a67b54e05c83410e692b1e52d0ef1f4829c4dc6a8ff3c0628f506aab  0.0s
 => => exporting config sha256:a65a67fe4d8061b5e1f726c4851d61b133fef660beed82ce9f82c6bbdf024a14  0.0s
 => => exporting attestation manifest sha256:e98061a27883f6ecf7d30e7aba370d8a8cee8629a600c60a3fbb22105e83f5  0.0s
 => => exporting manifest list sha256:575d5eac34dcc9c7b8061965c16cdd08dbee146567c046f85fe0a5ac44c6721b  0.0s
 => => naming to docker.io/library/demo-app:local                                              0.0s
 => => unpacking to docker.io/library/demo-app:local                                           0.1s
Host-006:app mariapiabuonomo$ docker save demo-app:local > demo-app.tar
Host-006:app mariapiabuonomo$ multipass transfer demo-app.tar LAB12vm:/home/ubuntu
Host-006:app mariapiabuonomo$
```

Then I went back to the VM in order to open the docker files and use the demo app by uninstalling and re-installing it. (in the following screenshot I put the wrong name at first)

```
ubuntu@LAB12vm:~/advanced-kubernetes$ sudo k0s ctr images import /home/ubuntu/demo-app.tar
unpacking docker.io/library/demo-app:local (sha256:575d5eac34dcc9c7b8061965c16cdd08dbee146567c046f85fe0a5ac44c6721b)...done
ubuntu@LAB12vm:~/advanced-kubernetes$ helm uninstall demo-app
Error: uninstall: Release not loaded: demo-app: release: not found
ubuntu@LAB12vm:~/advanced-kubernetes$ helm install assignment-app oci://ghcr.io/unusualfor/demo-app -n assignment --create-namespace
Pulled: ghcr.io/unusualfor/demo-app:0.1.0
Digest: sha256:001f32c6dfa671ff3ca8fbc88683ba2755c9ae1d32997c81c465d29f0f471581
Error: INSTALLATION FAILED: cannot re-use a name that is still in use
ubuntu@LAB12vm:~/advanced-kubernetes$ helm uninstall demo-app
Error: uninstall: Release not loaded: demo-app: release: not found
ubuntu@LAB12vm:~/advanced-kubernetes$ helm uninstall assignment-app -n assignment
release "assignment-app" uninstalled
ubuntu@LAB12vm:~/advanced-kubernetes$ helm install assignment-app oci://ghcr.io/unusualfor/demo-app -n assignment \
  --set image.repository=demo-app \
  --set image.tag=local \
  --set image.pullPolicy=Never
Pulled: ghcr.io/unusualfor/demo-app:0.1.0
Digest: sha256:001f32c6dfa671ff3ca8fbc88683ba2755c9ae1d32997c81c465d29f0f471581
NAME: assignment-app
LAST DEPLOYED: Tue Jan 27 16:06:02 2026
NAMESPACE: assignment
STATUS: deployed
REVISION: 1
TEST SUITE: None
ubuntu@LAB12vm:~/advanced-kubernetes$ kubectl get pods -n assignment -l app=demo-app -w
NAME                        READY   STATUS    RESTARTS   AGE
demo-app-545c9b6cc6-ll2gv   1/1     Running   0          18s
^Cubuntu@LAB12vm:~/advanced-kubernetes$
```

Now that is running I first test the endpoint:

```
^Cubuntu@LAB12vm:~/advanced-kubernetesCLUSTER_IP=$(kubectl get svc -n assignment demo-app -o jsonpath='{.spec.clusterIP}')")
curl http://$CLUSTER_IP:8000/metrics
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 391.0
python_gc_objects_collected_total{generation="1"} 133.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable objects found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 46.0
python_gc_collections_total{generation="1"} 4.0
python_gc_collections_total{generation="2"} 0.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="10",patchlevel="19",version="3.10.19"} 1.0
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.84029184e+08
# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 2.2786048e+07
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.76952636198e+09
# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.23
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 6.0
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 999999.0
# HELP demo_app_requests_total Total requests to demo app
# TYPE demo_app_requests_total counter
demo_app_requests_total 37.0
# HELP demo_app_requests_created Total requests to demo app
# TYPE demo_app_requests_created gauge
demo_app_requests_created 1.769526362925001e+09
# HELP demo_app_request_latency_seconds Request latency in seconds
# TYPE demo_app_request_latency_seconds histogram
demo_app_request_latency_seconds_bucket{le="0.005"} 0.0
demo_app_request_latency_seconds_bucket{le="0.01"} 0.0
demo_app_request_latency_seconds_bucket{le="0.025"} 0.0
demo_app_request_latency_seconds_bucket{le="0.05"} 0.0
demo_app_request_latency_seconds_bucket{le="0.075"} 0.0
```

And then I find the Prometheus and Grafana URL as done in the previous module 3.

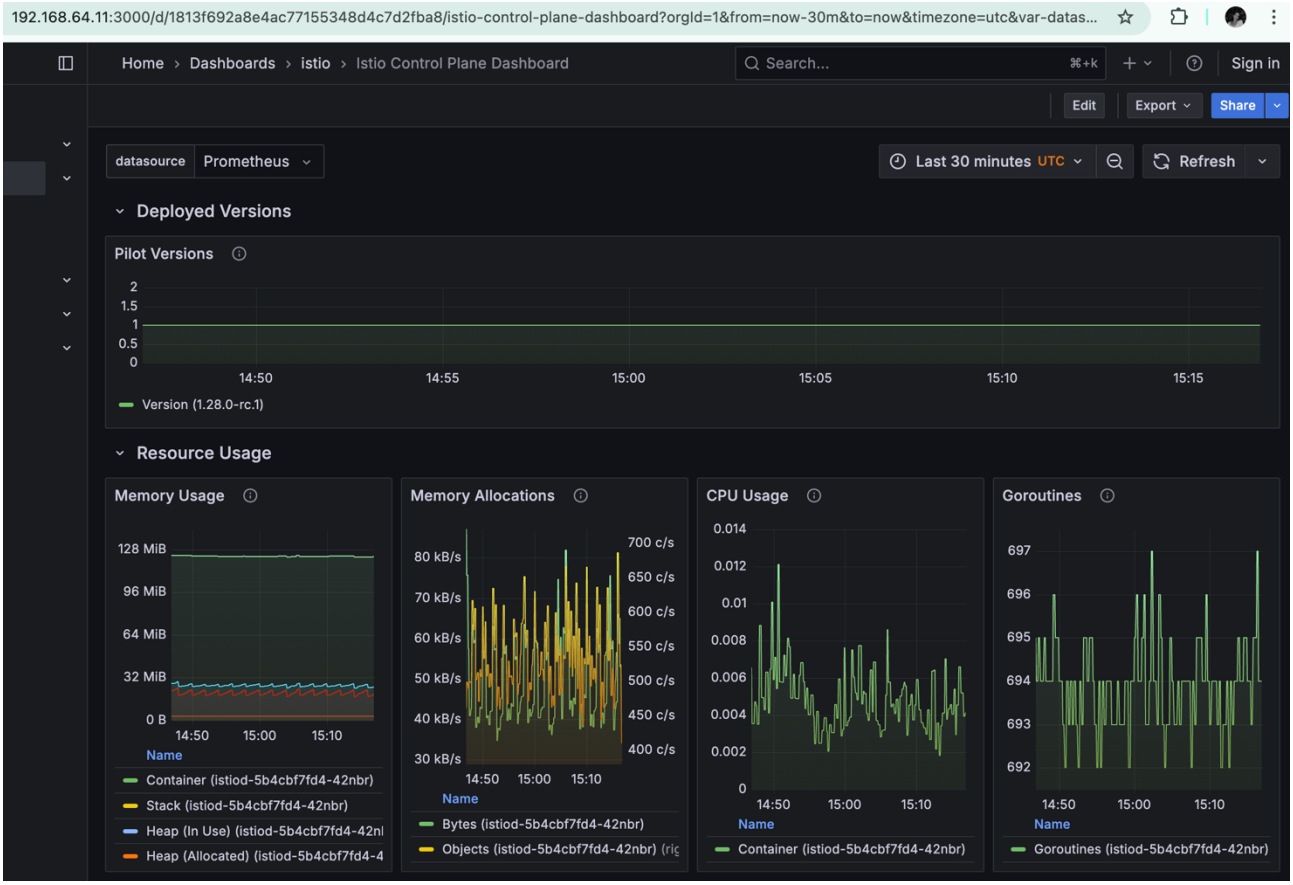
export PNODE_PORT=$(kubectl get -n default svc prometheus-server -o jsonpath="{.spec.ports[0].nodePort}")
export PNODE_IP=$(kubectl get nodes -o jsonpath="{.items[0].status.addresses[0].address}")
echo "Prometheus UI: http://$PNODE_IP:$PNODE_PORT/targets"

The output is:
Prometheus UI: http://192.168.64.11:30303/targets
To open Prometheus I just click on the link.

So I open another terminal of my VM and start the forwarding to the wanted port.
For Grafana I have:
http://192.168.64.11:3000

```
[^Cubuntu@LAB12vm:~/advanced-kuberneteskubectl port-forward -n istio-system svc/grafana 3000:3000 --address 0.0.0.0
Forwarding from 0.0.0.0:3000 -> 3000
Handling connection for 3000
Handling connection for 3000
Handling connection for 3000
Handling connection for 3000
Handling connection for 3000
Handling connection for 3000
Handling connection for 3000
```

Here I will find all the Istio dashboards.



Examples of results:

During the traffic generation phase, I performed connectivity testing using curl against the Service ClusterIP. I ensured the application was reachable on port 8000 by verifying the Pod readiness probe and the Service selector matching the assignment namespace. So I performed this while cycle and visualize possible results on Grafana:

The while loop used for traffic generation is a critical part of the observability test.

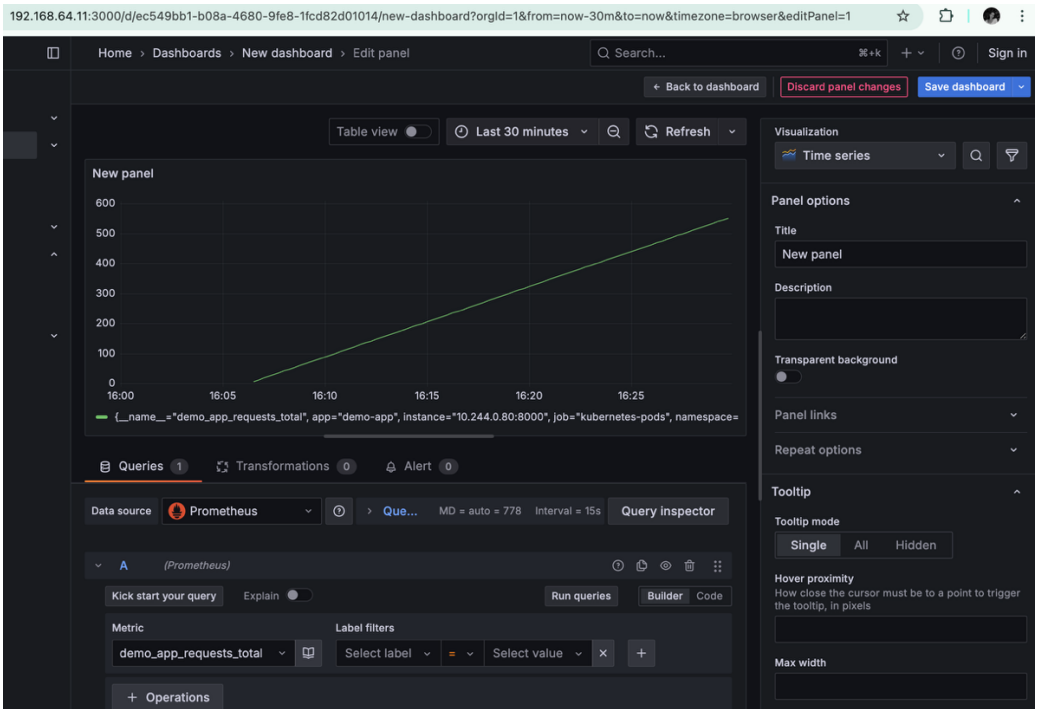
- curl -s -m 2 http://$CLUSTER_IP:8000/hash: This command performs a silent request to the application's /hash endpoint with a 2-second timeout.
- || echo "Connection ERROR": This is a defensive programming practice to immediately signal any network failure or pod crash during the test.

By sending requests every second, I am creating a constant load that allows Prometheus to calculate stable rates (per-second averages) rather than showing erratic spikes.

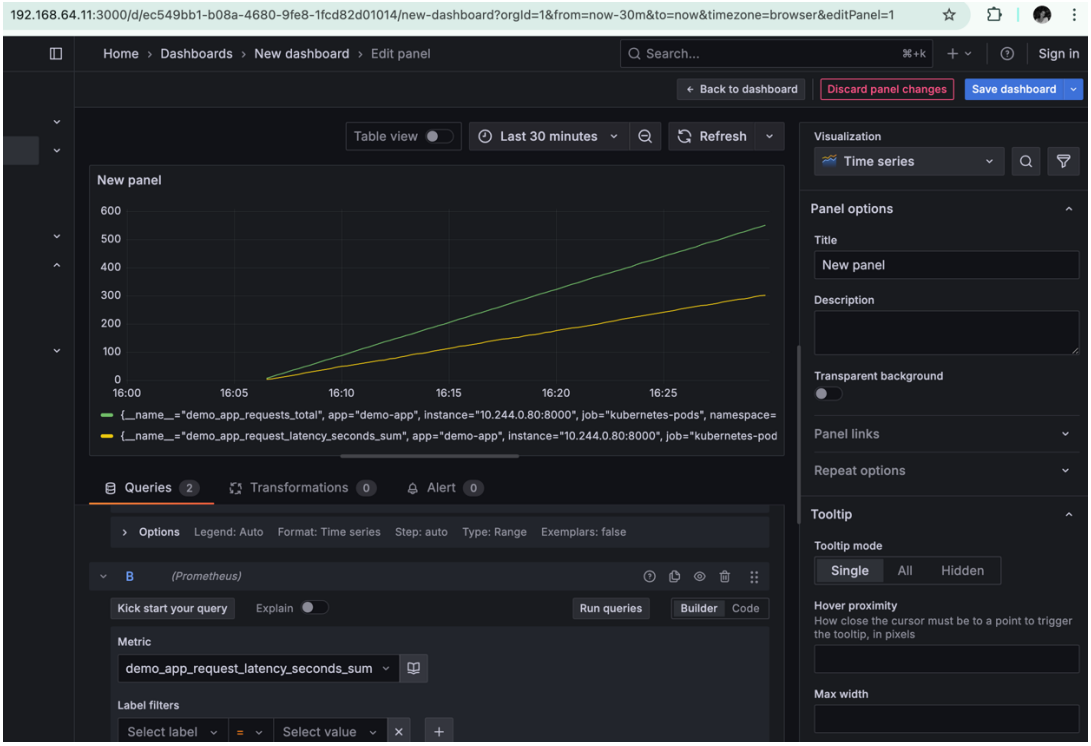
```
while true; do
  echo -n "$(date +%H:%M:%S) – Sending the request... "
  curl -s -m 2 http://$CLUSTER_IP:8000/hash || echo "Connection ERROR"
  sleep 1
done
```

When running the curl command, the application returns raw Prometheus metrics.

demo_app_requests_total: This is a Counter. It only goes up. In the results, it shows over 500 requests, confirming the application has been active and stable.



The metric demo_app_request_latency_seconds_sum represents the cumulative time, in seconds, that the application has spent processing all incoming requests since the last restart.



This i show much the Cpu was used: