Cloud computing

Microservices Project

Lorteau Erwan

2023-2024 - Mosig M2 DSAI

Deployment

First, we follow the deployment guideline in the original project git repository. Therefore we create a Google cloud project and directly setup the zone and the ID of the project

- Zone : europe-west10 (choosen after a lookup at Gcloud compute zone list)
- ID : lorteau-microservices-demo

I haven't been able to used the recommended setup (machine type: e2-standard-2, number of nodes: 4, as it require up to 1200GB and we cannot increase that limit during the free trial)

Autopilot : Autopilot mode allows provisioning and managing the infrastructure automatically (nodes etc). It also does scaling, adjusting the resources of each pod based on the workload.

Using the external ip address, we're capable of accessing the shop on the browser
Beside that, we also check the log of the loadgenerator to verify the app is properly running

- kubectl logs loadgenerator-6df4597d5b-vzk2c

Overall, everything works properly so far.

Explain the configuration files

I picked the payment service To display the config file

- kubectl get service frontend -o yaml

Here is an explanation of the main lines :

  - *ApiVersion* specify the version
  - *Kind* specify if the following configuration is for a service, or a deployement
  - *name* contain the name of the service
  - *namespace* is the namespace in which it is created
  - *clusterIP* is the address assigned to this service within the cluster
  - *ports* are the ports in which the service is available,
  - *name* is the name attached to the port
  - *targetPort* is the port in which the service send request
  - *protocol*: The protocol used by the service (default is TCP)
  - selector : select the pod to which the service will route (link service/pod)
  - creationTimeStamp, uid are system generated
  - type : clusterIP make the service accessible within the cluster (define scope)

Targetting minimal deployment

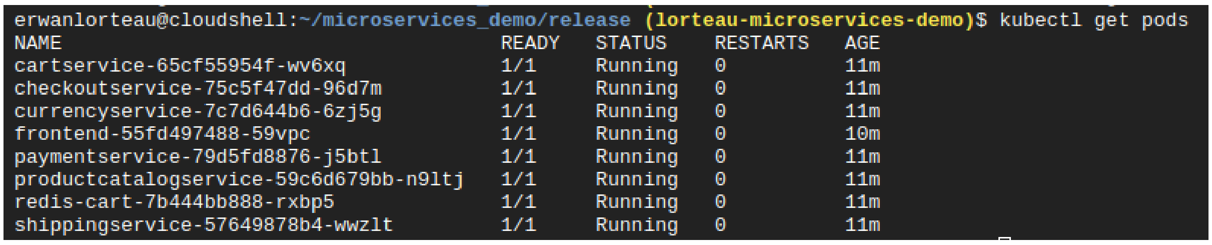
I identified four unnecessary services for a minimal deployment

1. adservice
2. recommandationservice
3. emailservice
4. loadgenerator

The currency service's usefulness could be debated but I decided to keep it since we want customers to be able to pay in any currency. To apply the change, we first need to redefine the yaml configuration file in release/. We simply remove every configuration (deployement & service) of the services we want to delete. The resulting file is saved inside the project as kubernetes-manifests.yaml and I also kept the old one and renamed it. They are available in /release

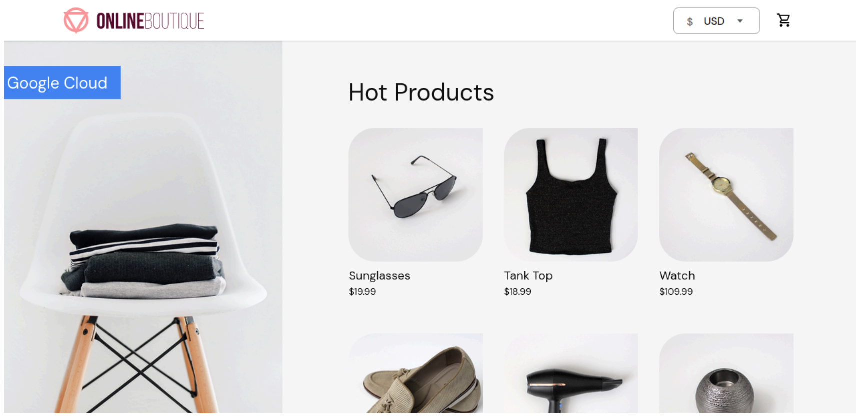
Secondly, we need to apply the changes on the cluster

1. In my case, I was redoing the deployment on a new Gcloud project so I didn't had to stop all the pods, however you need to do it if you are running on the same
2. Then, delete the old conf kubectl delete -f ./release/kubernetes-manifests.yaml
3. And apply the change : kubectl apply -f ./release/kubernetes-manifests.yaml
4. We can observe the essential pods are now runnning



```
erwanlorteau@cloudshell:~/microservices_demo/release (lorteau-microservices-demo)$ kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
cartservice-65cf55954f-wv6xq             1/1     Running   0          11m
checkoutservice-75c5f47dd-96d7m          1/1     Running   0          11m
currencyservice-7c7d644b6-6zj5g          1/1     Running   0          11m
frontend-55fd497488-59vpc                1/1     Running   0          10m
paymentservice-79d5fd8876-j5btl          1/1     Running   0          11m
productcatalogservice-59c6d679bb-n9ltj   1/1     Running   0          11m
redis-cart-7b444bb888-rxbp5              1/1     Running   0          11m
shippingservice-57649878b4-wwzlt         1/1     Running   0          11m
```

Same thing from the browser, working as well before and after the changes :



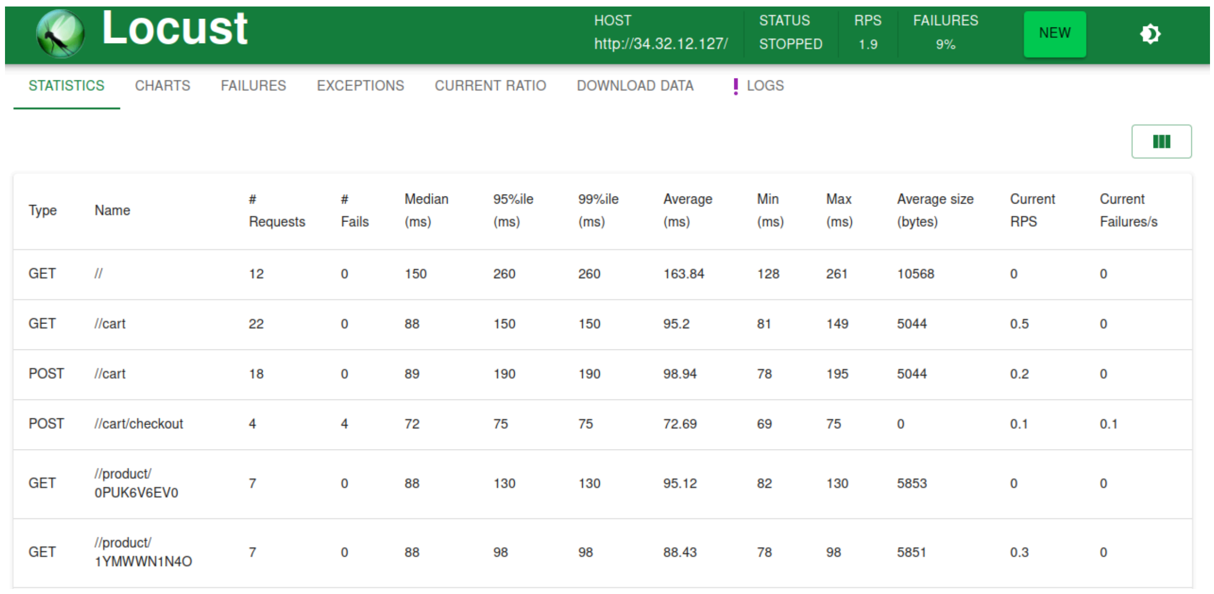
Deploying the load generator on a local machine

We simply download the archive, move to src/loadgenerator

The we build and run the container :

1. We launch locus using the command given in their website documentation : docker run -p 8089:8089 -v $PWD:/mnt/locust locustio/locust -f /mnt/locust/locustfile.py
2. Access the GUI on localhost:8089

We can access the GUI and perform some test :



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	//	12	0	150	260	260	163.84	128	261	10568	0	0
GET	//cart	22	0	88	150	150	95.2	81	149	5044	0.5	0
POST	//cart	18	0	89	190	190	98.94	78	195	5044	0.2	0
POST	//cart/checkout	4	4	72	75	75	72.69	69	75	0	0.1	0.1
GET	//product/0PUK6V6EV0	7	0	88	130	130	95.12	82	130	5853	0	0
GET	//product/1YMWWN1N4O	7	0	88	98	98	88.43	78	98	5851	0.3	0

Here, we just provided the frontend ip address and ran a few seconds of test to see if locust was able to reach the app. We can now inject load from here.

Deploying automatically the load generator in Google Cloud

I used terraform to automatically deploy an e2 VM and perform the following automatically

1. Install docker, git
2. Pull the repository
3. Build the locust image using the locustfile.py

Please note I’m not using the Dockerfile provided in the git repository but rather the locust file and the docker image provided by locust directly. (It was better documented and worked first try).

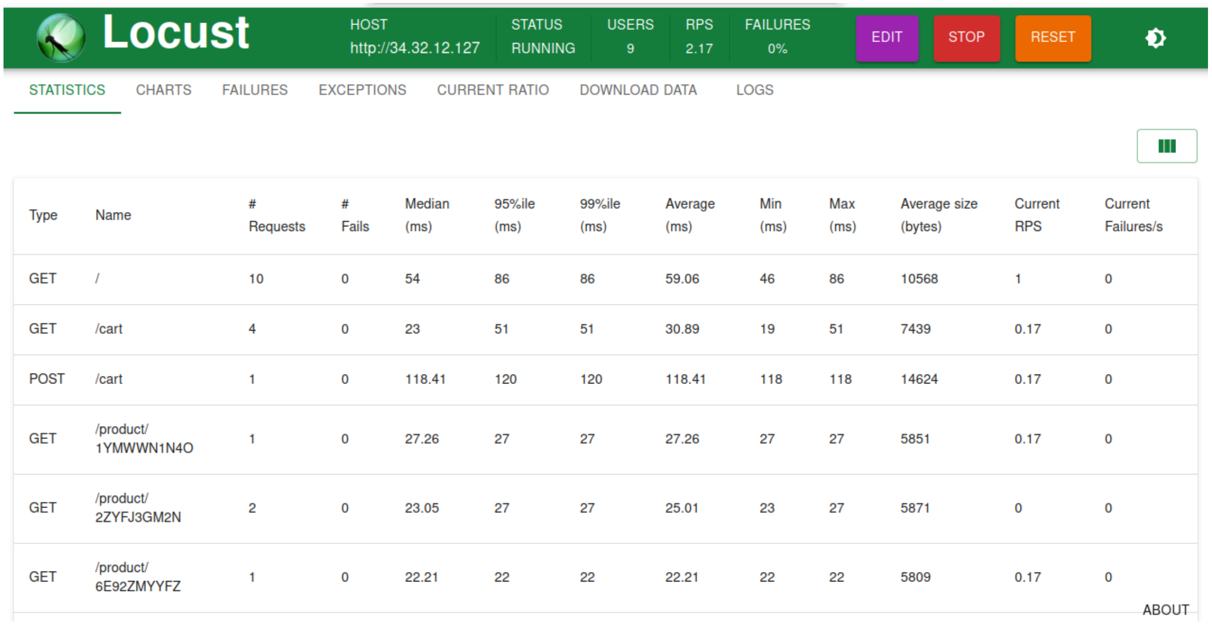
The script is available in the archive in terraform/terraform_vm_ script.tf. It simply used a declarative approach to instantiate a VM on the gcloud project as it allows a quick deployment of the VM.. It also installs git, docker, and pull the git repository, builds the locust image with the corresponding locustfil at booting using a script which is executed everytime the vm starts. The script is quite explicit, and the version in the archive implements 3 VM (using count) in for the next steps

To use /mofidy the script

1. terraform init
2. Do some modifications
3. terraform validate
4. terraform plan
5. terraform apply

Note : I used google-cloud-cli to authenticate terraform to the google cloud accound

Then, go to the web browser, find the IP address of the machine on the gcloud conole and go to port 8089 to access the GUI. Here is a quick overview : It allows us to specify the number of fake users, the amount that connects each second and the maximum number of fake users to reach. It is now easy to provision several VM's to inject load.



HOST	STATUS	USERS	RPS	FAILURES
http://34.32.12.127	RUNNING	9	2.17	0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	10	0	54	86	86	59.06	46	86	10568	1	0
GET	/cart	4	0	23	51	51	30.89	19	51	7439	0.17	0
POST	/cart	1	0	118.41	120	120	118.41	118	118	14624	0.17	0
GET	/product/ 1YMWWN1N4O	1	0	27.26	27	27	27.26	27	27	5851	0.17	0
GET	/product/ 2ZYFJ3GM2N	2	0	23.05	27	27	25.01	23	27	5871	0	0
GET	/product/ 6E92ZMYYFZ	1	0	22.21	22	22	22.21	22	22	5809	0.17	0

Monitoring the application and the infrastructure

For this part, in order to install prometheus I deleted the old cluster to remake one with no autopilot as it was conflicting with prometheus default installation. I used the recommended e2 instances.

Furthermore, I installed grafana and prometheus on the gcloud shell :

1. helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
2. helm repo update
3. helm install prometheus prometheus-community/prometheus

4. helm repo add grafana https://grafana.github.io/helm-charts
5. helm repo update
6. helm install grafana grafana/grafana

Then I obtain the grafana credentials and change the service type to LoadBalancer (I was facing cors errors while linking it to prometheus as a datasource). I can thus access the browser UI. To access prometheus GUI you have forward the pod of the prometheus server on port 9090 (local in cloud shell).

On the grafana dashboard, I can connect it to prometheus and use the prometheus data sourse to create charts to monitor the app. I did two basic dashboards, one for the pod and one for the nodes, each of them displaying various informations

Pod dashboard

1. Memory usage in bytes :
   sum(container_memory_usage_bytes{container!="POD",container!=""}) by (pod)

2. Disk read operations in bytes :
   sum(rate(container_fs_reads_bytes_total{pod!=""}[5m])) by (pod) (turn out there is very little)
3. Dis write operations in bytes :
   sum(rate(container_fs_writes_bytes_total{pod!=""}[5m])) by (pod)
4. network received bytes :sum(rate(container_network_receive_bytes_total{pod!=""}[5m])) by (pod)
5. network sent bytes :
   sum(rate(container_network_transmit_bytes_total{pod!=""}[5m])) by (pod)
6. pod restart count : sum(kube_pod_container_status_restarts_total{pod!=""}) by (pod)
7. cpu usage :
   sum(rate(container_cpu_usage_seconds_total{container!="POD",container!=""}[5m])) by (pod)

Node dashboard

8. CPU % usage : 100 - (avg by (instance) (irate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)
9. Memory % usage : (node_memory_MemTotal_bytes - node_memory_MemAvailable_bytes) / node_memory_MemTotal_bytes * 100

Of course we could consider more elaborate metrics, here my purpose was only to provide the most important one, allowing us to identify potential problems like bottlenecks, network issues, etc.

Performance evaluation

The performance evaluation will consist of 3 VM running each instance of the loadgenerator. Each VM runs the same amount of “fake'” clients, meaning that we’re just splitting the work among VM’s.

They are all e2 instances, following the recommended specifications (8gb ram, 2cores). All the VM are located in the same region as the cluster, but aren't within the cluster. We’ll use the previously configured terraform script to deploy them

<u>Since there was no further instruction in the lab, I performed the test set on the minimal deployment.</u>

We’ll perform the following tests :

- Stress test : continuously increasing the amount of user until it crashes (+1user/vm/sec)
- Spike test : Scaling the number of users up to 200 users (same)
- Endurance test on 100 users (10 min”)

We’ll oberve for the stress test

- The maximum amount of user the app can handle (knowing we provisioned 4 nodes, without autopilot)
- The possible cause (lack of memory, cpu…)

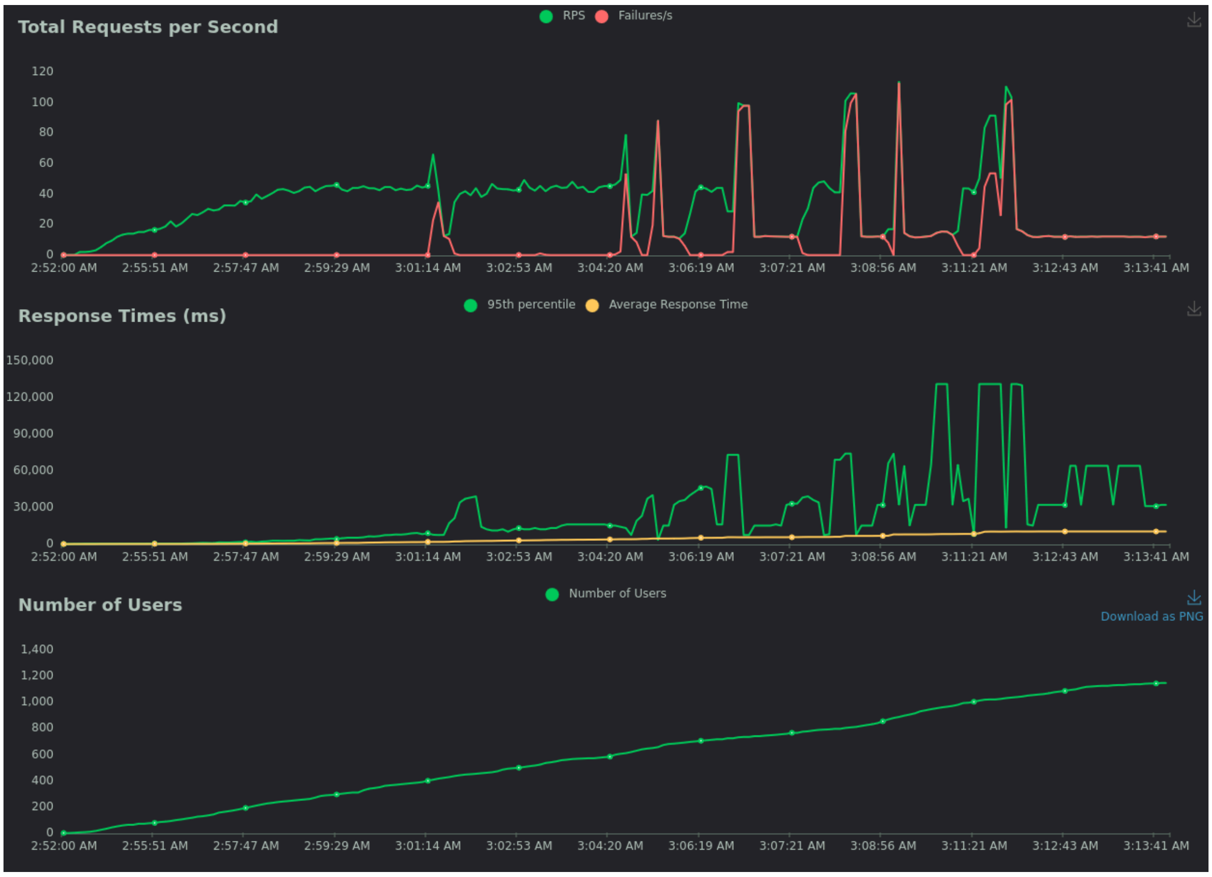
For the spike test

- Is the workload balanced among nodes
- Which pods are under pressure (high disk/network/cpu/memory usage)

We’ll observe for the endurance test

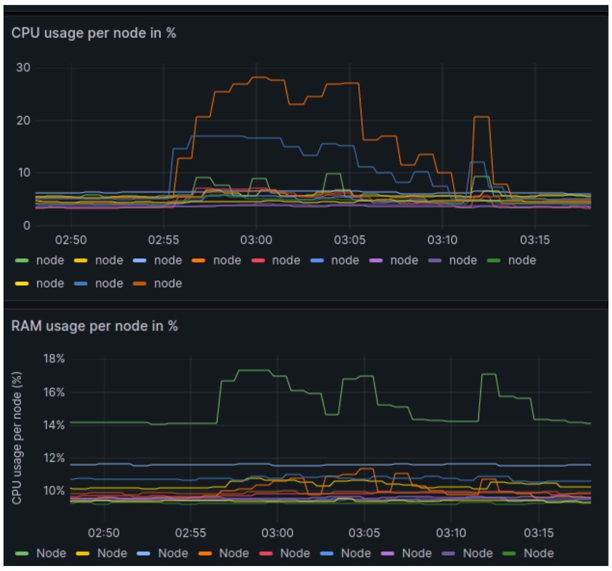
- Point of failures (which request)
- median response time
- Failure percentage

Stress test result
Here is one locust chart (each of the 3 is similar)

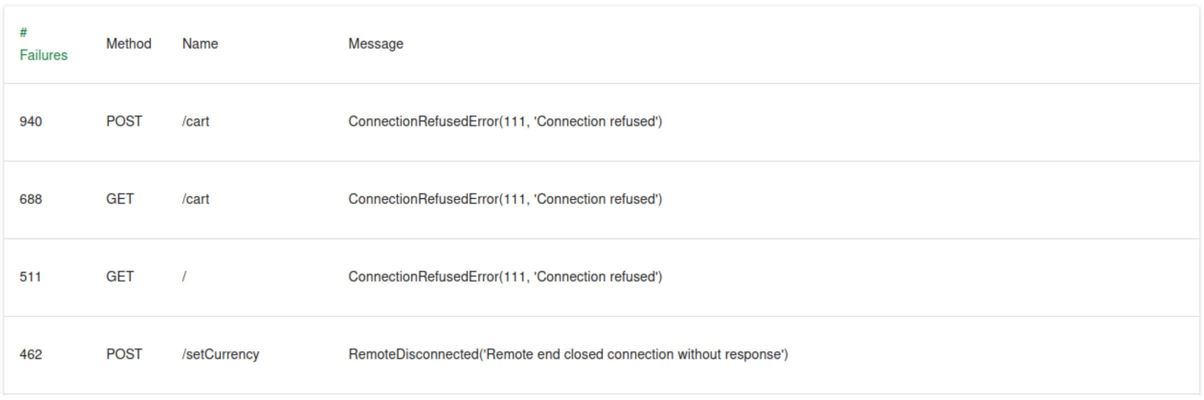


The app peaks is stable until ~ 3300 users (1100 per vm) after which the amount of failure becomes equal to the amount of request, indicating that the app is not able to deserve any requests properly.

I tried to identify the cause. We can already dismiss the ram and cpu, as shown below. The disk looks fine as well, so maybe the network buffer became a bottleneck. I couldn't identify the source with certitude. I also checked the disk I/O and they seem to not exceed 2mb/second which should be fine.

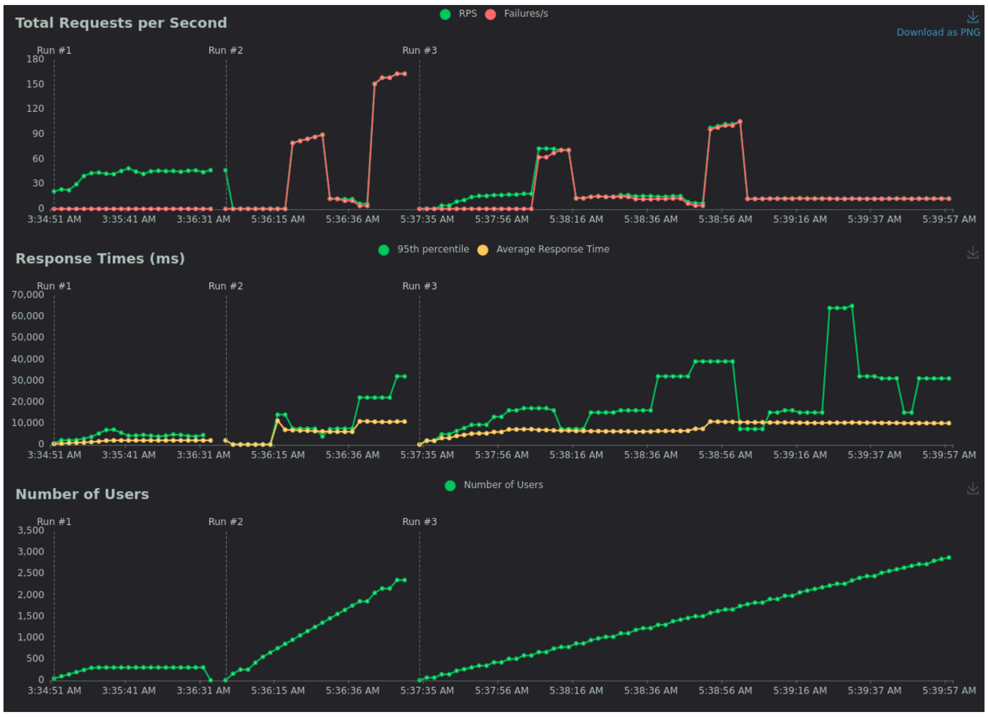


On the locust side, almost all errors occur after a certains threshold of the user indicating that this is due to the inability to scale rather than actuall app bugs (indeed, it does not scale yet since its not on autopilot). Here are the most common ones which confirm this idea (Almost all errors are “Connection refused”, obviously, when the servers ceases to answer..). This is also the reason we will be looking at the median latency and not the average

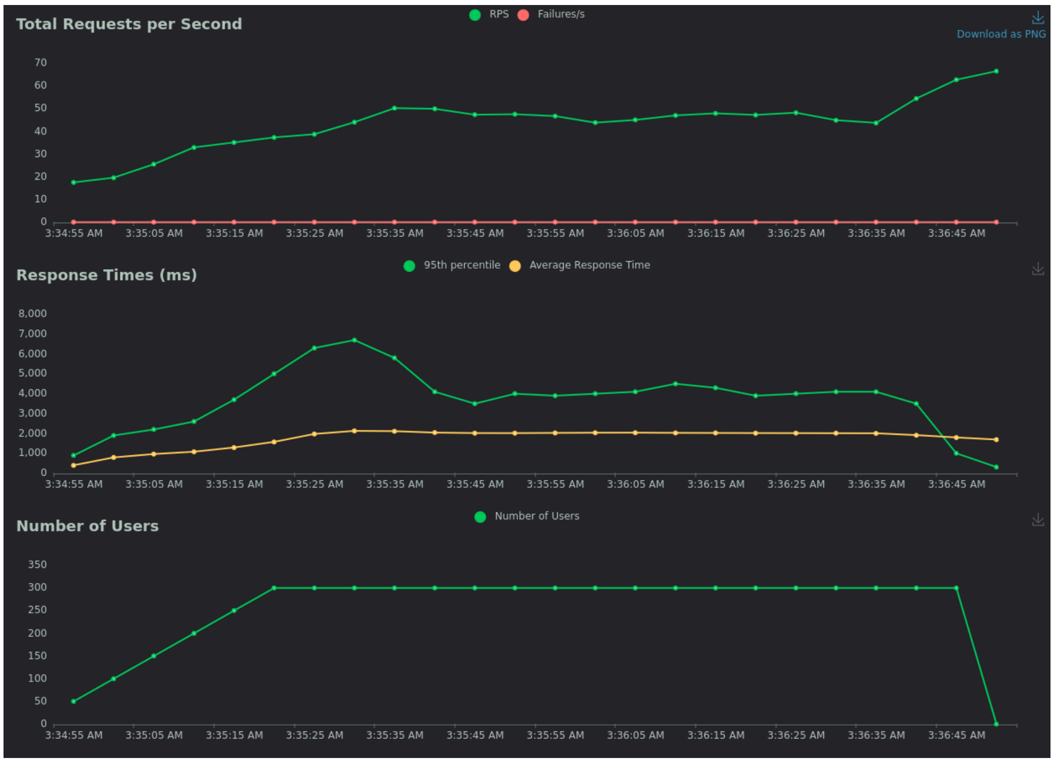


# Failures	Method	Name	Message
940	POST	/cart	ConnectionRefusedError(111, 'Connection refused')
688	GET	/cart	ConnectionRefusedError(111, 'Connection refused')
511	GET	/	ConnectionRefusedError(111, 'Connection refused')
462	POST	/setCurrency	RemoteDisconnected('Remote end closed connection without response')

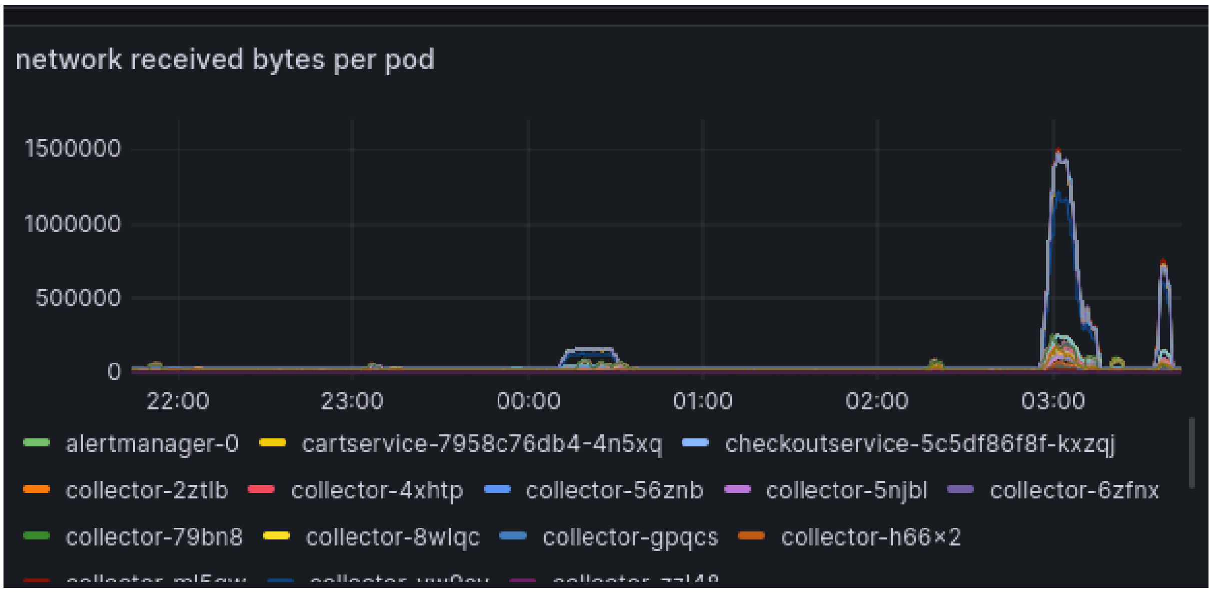
I reran the experiment going even further (run 3, up to 7500 users, but with 20 new users per second and per vm) and the same pattern appears. Again, after some threshold around 3000 thousand users, we almost have only failures.



Spike test

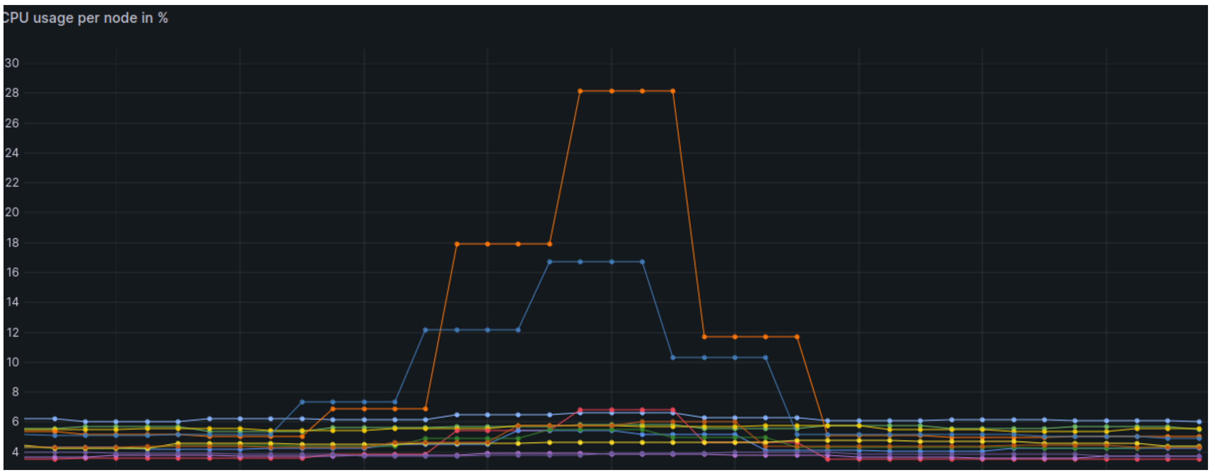


This time as expected, under 1000 users, there is no failure. The response time is too high, with a median of 3600ms. It increases along with the number of users. Using grafana we observe that none of the pods reach max usage of the cpu, nor RAM. I also checked the network inputs to see if it could potentially be a bottleneck, but it appear that for this second spike corresponding to the second test, it caps at ~1.5MB/s which should be handled (e2-standard-2 instance have up to 4Gb/s). This strengthen the idea that the latency and eventually the failures are dues to network buffer overflows



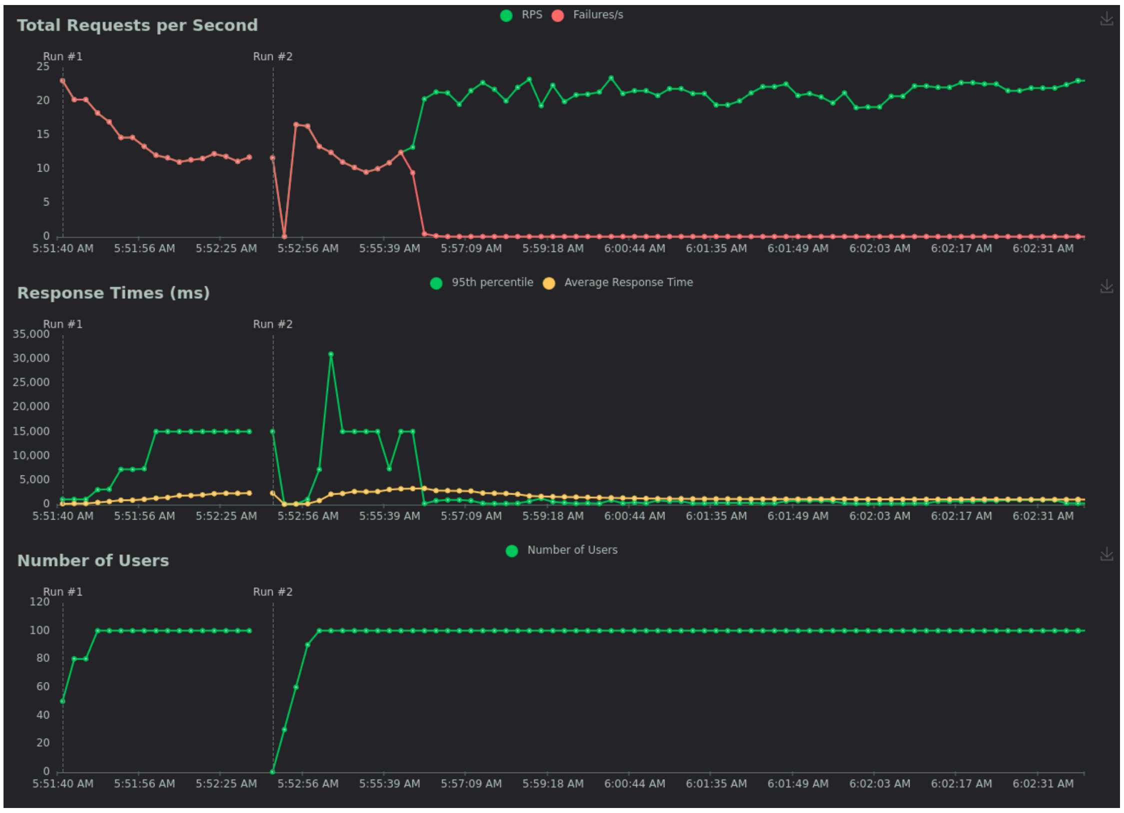
The latency is probably high only because we have a maximum of 4 nodes, therefore the network buffer could be easily overwhelmed.

As for the load balancing here is the results in grafana.



Endurance test

For this test I'm running 100 user per vm (10/sec) for 10 minutes and observe a failure rate of 0% with a median latency of 55ms



Note that the failures at the beginning are due to the fact that I was just stopping the stress test.

Conclusion on performances

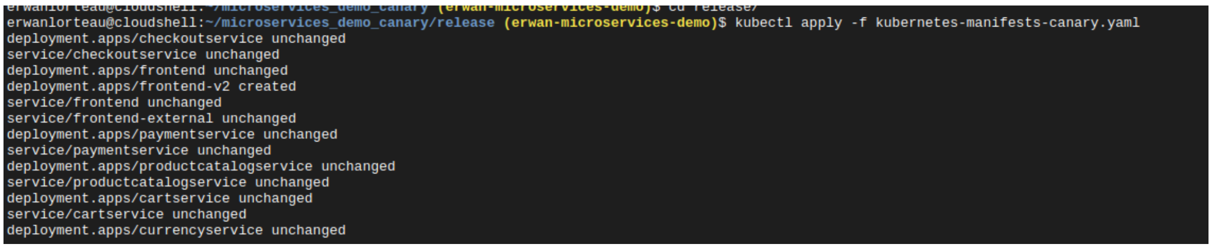
To conclude, after a lot of other experience and observations the app works properly until ~ 1000's of users, and the median average increases with the amount of users. It indicates some kind of network buffer overflower. After ~ 3000 users, the app cannot serve requests anymore. The ressource usage (disk, cpu, memory), are enough to handle even more users, but the latency migth indicate a network buffer overflow that lead to a lot of failure before reaching the cpu capacity.

Additionnal test that could be added
Unfortunatly I would have liked to progressively decrease the amount of users to see if the cluster was recovering but locust does not provide the option

Canary releases

I modified the title of the header (Online shop → Online shop v2) in the frontend component. The very first thing I did is to modify the code, and then rebuild the docker image. I registered the image on the google cloud registery. Then, I modified the deployment yaml file.

I copied the entire deployment section of the frontend service, and changed the source image to the docker image I've just discussed. I deployed it.

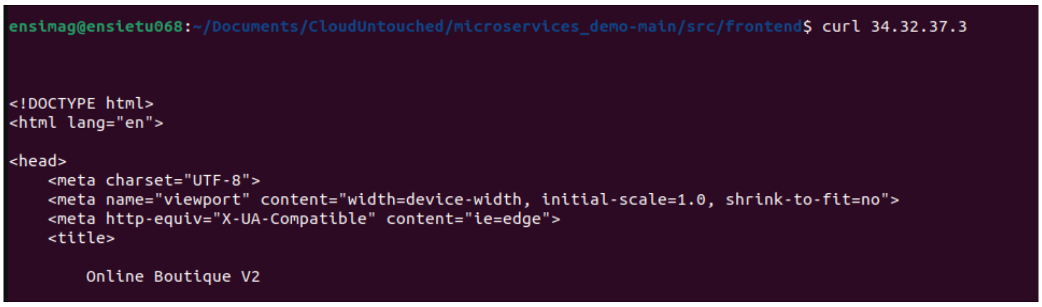


```
erwanlorteau@cloudshell:~/microservices_demo_canary (erwan-microservices-demo)$ cd release/
erwanlorteau@cloudshell:~/microservices_demo_canary/release (erwan-microservices-demo)$ kubectl apply -f kubernetes-manifests-canary.yaml
deployment.apps/checkoutservice unchanged
service/checkoutservice unchanged
deployment.apps/frontend unchanged
deployment.apps/frontend-v2 created
service/frontend unchanged
service/frontend-external unchanged
deployment.apps/paymentservice unchanged
service/paymentservice unchanged
deployment.apps/productcatalogservice unchanged
service/productcatalogservice unchanged
deployment.apps/cartservice unchanged
service/cartservice unchanged
deployment.apps/currencyservice unchanged
```

It should be enough to ensure a 50-50 splitting but to ensure 25-75, I had to use istio.

I installed istio, allows injections inside the defaults namespaced, and modified the yaml config file to add a virtual service. The virtual service allows to specify a splitting for the routing rule : when accessing the frontend service, it can goes to v1 or v2, with a 75-25 splitting. the corresponding yaml file are available in /release folder (rule/virtual)

I tested the code using curl (unfortunatly web browsers uses cache hes which make it hard to test) and obtained around 25% of the time the desired service with an altered title (online boutique v2 instead of online Boutique) as shown below. However beware, I'm not sure it actually change anything on the browser UI since i tested it directly on the console.



```
ensimag@ensietu068:~/Documents/CloudUntouched/microservices_demo-main/src/frontend$ curl 34.32.37.3



<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0, shrink-to-fit=no">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>

        Online Boutique V2
```