CSC8110 – Cloud Computing

Assessment 1 – Report - 230302626

November 25, 2023

Abstract

This project for the Cloud Computing course involves a comprehensive assessment of deploying and managing applications in Kubernetes-based environment. It requires a good research before engaging in hands-on tasks, such as deploying a Kubernetes Dashboard, a web application, setting and managing to access Grafana, developing and locally deploying a load generator for benchmarking the application that drives to monitoring stage of the container metrics. This report is consistent with parts that will detailly explain the life flow of the process step-by-step, which will be supported by the screenshots from the system.

1 Introduction

In the fast-evolving landscape of cloud computing, the integration of the Kubernetes into application development and deployment stands as a critical advancement. This project report delves into constructing a sophisticated Kubernetes-based pipeline, harnessing its power for efficient cloud application management. This involves a hands-on exploration of deploying, monitoring, and optimizing applications within a Kubernetes environment, utilizing cutting-edge tools and methodologies.

The report chronicles the journey from the foundational deployment of a Kubernetes Dashboard and a Java web application in Task 1, to the intricate setup of a monitoring stack using Grafana in Task 2. It encapsulates the practical application of Kubernetes, highlighting the synergy between container orchestration and cloud application performance. Task 3 elevates this exploration, focusing on the creation of a load generator to benchmark and analyse application resilience and scalability under varied load conditions. This task is crucial in understanding the dynamics of cloud-based applications and preparing them for real-world demands.

The culmination of this project in Task 4 involves a detailed analysis of the monitoring and benchmarking data. This task is not just about data interpretation; it's about understanding the implications of these insights for application performance and reliability in a Kubernetes environment. It emphasizes the necessity of a data-driven approach in optimizing cloud application performance and ensuring robustness in a cloud-centric world.

This report not only details the technical procedures and outcomes of each task but also reflects on the challenges encountered and the solutions devised. It provides a comprehensive view of the project, encapsulating the knowledge and skills gained in navigating the complexities of Kubernetes and cloud computing. The insights gleaned from this endeavour are invaluable, offering a profound understanding of the current and future landscape of cloud application management.

2 Journey of the Project

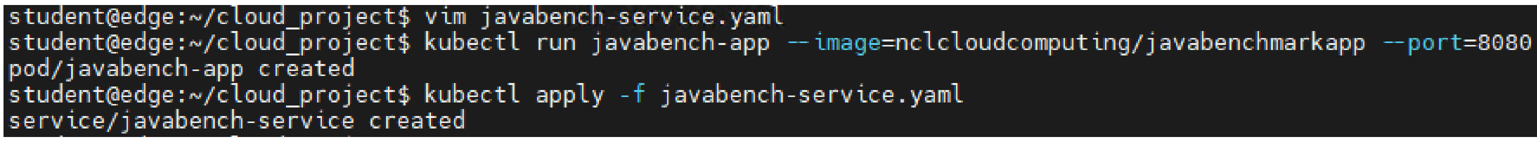
2.1 How to start IoT project

First of all, I started reading the whole project description and wrote down each main points for myself to see the big picture on the flow. As it is a quite detailed project, I used “divide and conquer” principle. I divide each task into 3 groups and defined what should I do to get to the next stage. After that I started to researching the points that I do not now deeply. As a start, I started researching the Kubernetes technology to see where, why and how it has been used in real-world scenarios. I also searched for port-forwarding methods for the project to access several dashboards from my Windows machine rather than the VM. Then the research expanded from Grafana till the load-generator application concept experiences that other guys in industry published in the web. After that try-fail-refactor cycle started for the implementation part which resulted with success at the end.

2.2 Kubernetes Dashboard and Web Application Deployment

Deployment of Kubernetes dashboard executed as it is shown in the official documentation that is why I do not want to talk about that common part. I want to emphasize the different approaches that I used for successfully completing this task. After deployment process it was time to access it, but I did not want to access dashboard inside the VM. That it why for proxy creation part I added feature to the script that will allow to get access from all hosts ***kubectl proxy --address='0.0.0.0' --accept-hosts='^*$'*** . Also I created an SSH tunnel by executing ***ssh -L 8001:localhost:8001 student@192.169.0.102*** in the terminal of the windows machine. Another different approach I used, was due to the reason that I had problem fetching the token from the secret of the user that I created for the dashboard access. As a alternative, I created a token manually for accessing using the script ***kubectl -n kubernetes-dashboard create token dashboard-admin*** in the VM terminal. After this steps I could easily access to the Kubernetes dashboard.

For the deployment of provided Java application, it is mentioned that deployment should happen via the CLI that is I do not create a .yaml for this deployment. Instead, the following script has been executed for the successful result: ***kubectl run javabench-app2 --image=nclcloudcomputing/javabenchmarkapp --port=8080*** . Then the requested simple .yaml file has been created and executed using the ***kubectl apply*** command. Alternatively, it can be also done using the following script which is not used in this project but both do the same job: ***kubectl expose deployment javabench-app2 --type=NodePort --port=8080 --node-port=30000***



```
student@edge:~/cloud_project$ vim javabench-service.yaml
student@edge:~/cloud_project$ kubectl run javabench-app --image=nclcloudcomputing/javabenchmarkapp --port=8080
pod/javabench-app created
student@edge:~/cloud_project$ kubectl apply -f javabench-service.yaml
service/javabench-service created
```

Figure 1- Deployment of Java Application

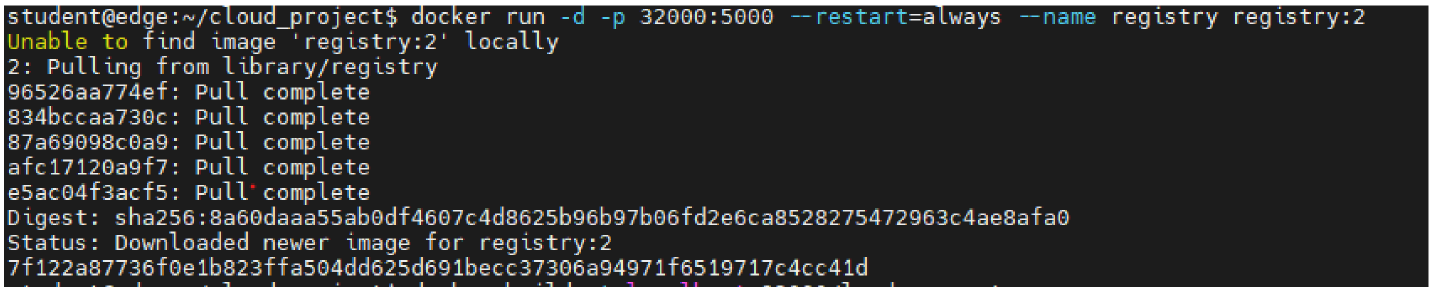
2.3 Deployment of Monitoring Stack of Kubernetes

Before starting this part of the task in was crucial to make a research about the monitoring stack of the kubernetes because it was directly related to the Task 4 which will require hands-on experience with Grafana. After gaining overall information it was time to execute and get the monitoring stack using the ***microk8s enable observability*** command in the terminal. For the next step, it was the time to configure the Grafana to allow connection from the host. The first and most common way was to edit Grafana Service using the ***kubectl -n observability edit svc kube-prom-stack-grafana*** . The point here that should be noted is the ***kube-prom-stack-grafana*** that can differ case-to-case and should be checked from the ***kubectl get svc –all-namespaces*** . Or at least it was different in my case compared to the documentation. The other was to the port-forwarding method which can be done using the ***kubectl -n observability port-forward svc/kube-prom-stack-grafana 3000:80*** script via the CLI. After successful steps, it was time to log in with the default credentials that were available in the .txt file in the VM.

2.4 Load Generator Implementation

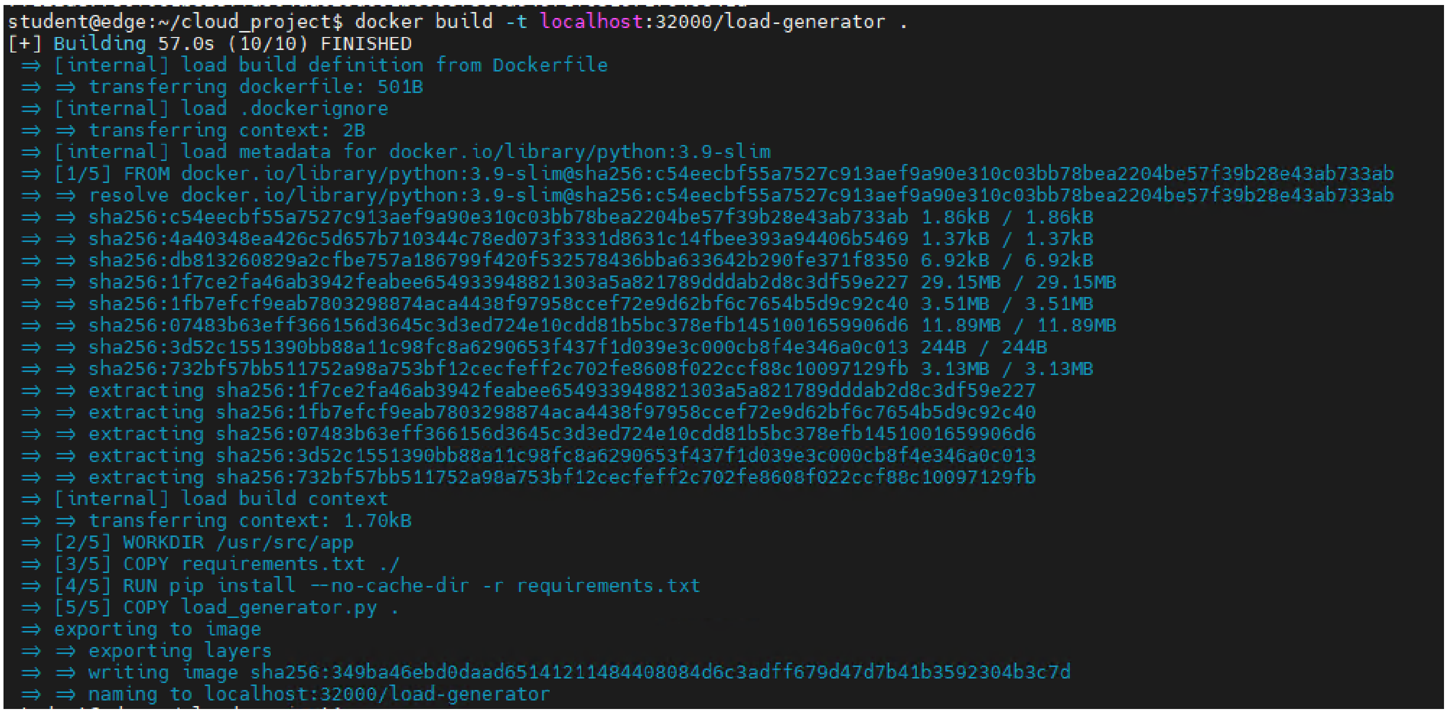
For this section of the project we were asked to program a load generator and push it to local registry at port **32000**. The target url and the request frequency are requested to be assigned as environment variable and as a output it was crucial to see the average response time and cumulative failures a output metrics. I used Python because in the IoT module we were also requested to work with **requests** library, that is why I was familiar with it. The response time can be also taken from the **/primecheck** but I think that under a lot load retrieving can be a problem. For this reason average time logic is done as a mathematical expression result within the program.

Later we were requested to pack the program as a standalone Docker image and push It to the local registry at port 32000. In the following figures you will see the completed of the process of the requirement, code snippet of the load generator, output of the program, docker images, active pods.



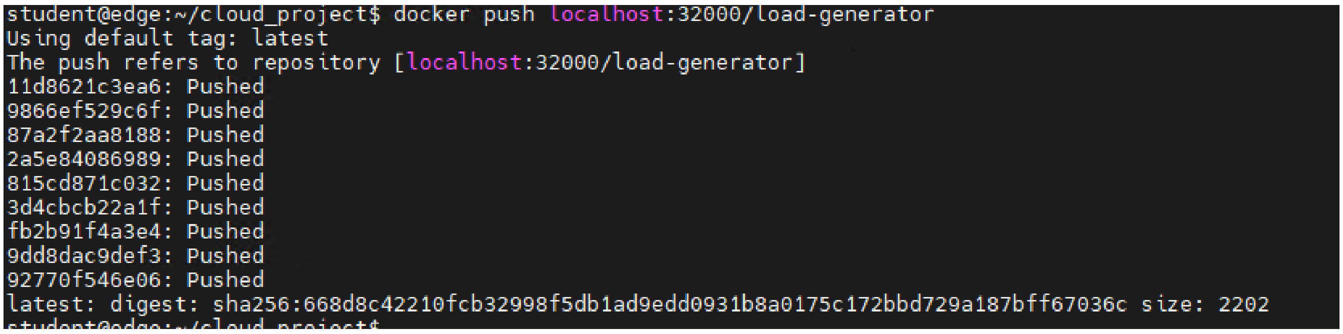
```
student@edge:~/cloud_project$ docker run -d -p 32000:5000 --restart=always --name registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry
96526aa774ef: Pull complete
834bccaa730c: Pull complete
87a69098c0a9: Pull complete
afc17120a9f7: Pull complete
e5ac04f3acf5: Pull complete
Digest: sha256:8a60daaa55ab0df4607c4d8625b96b97b06fd2e6ca8528275472963c4ae8afa0
Status: Downloaded newer image for registry:2
7f122a87736f0e1b823ffa504dd625d691becc37306a94971f6519717c4cc41d
```

Figure 2 - Local Registry



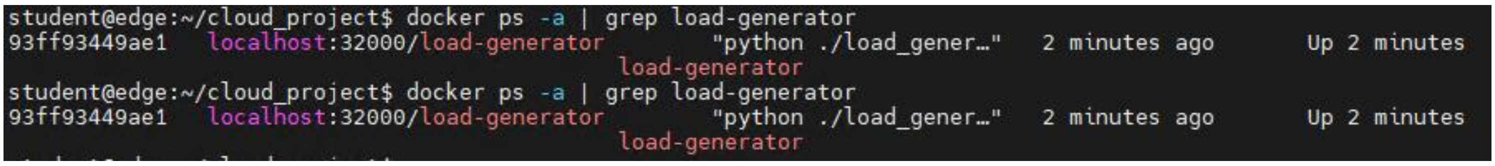
```
student@edge:~/cloud_project$ docker build -t localhost:32000/load-generator .
[+] Building 57.0s (10/10) FINISHED
 ⇒ [internal] load build definition from Dockerfile
 ⇒ ⇒ transferring dockerfile: 501B
 ⇒ [internal] load .dockerignore
 ⇒ ⇒ transferring context: 2B
 ⇒ [internal] load metadata for docker.io/library/python:3.9-slim
 ⇒ [1/5] FROM docker.io/library/python:3.9-slim@sha256:c54eecbf55a7527c913aef9a90e310c03bb78bea2204be57f39b28e43ab733ab
 ⇒ ⇒ resolve docker.io/library/python:3.9-slim@sha256:c54eecbf55a7527c913aef9a90e310c03bb78bea2204be57f39b28e43ab733ab
 ⇒ ⇒ sha256:c54eecbf55a7527c913aef9a90e310c03bb78bea2204be57f39b28e43ab733ab 1.86kB / 1.86kB
 ⇒ ⇒ sha256:4a40348ea426c5d657b710344c78ed073f3331d8631c14fbee393a94406b5469 1.37kB / 1.37kB
 ⇒ ⇒ sha256:db813260829a2cfbe757a186799f420f532578436bba633642b290fe371f8350 6.92kB / 6.92kB
 ⇒ ⇒ sha256:1f7ce2fa46ab3942feabee654933948821303a5a821789dddab2d8c3df59e227 29.15MB / 29.15MB
 ⇒ ⇒ sha256:1fb7efcf9eab7803298874aca4438f97958ccef72e9d62bf6c7654b5d9c92c40 3.51MB / 3.51MB
 ⇒ ⇒ sha256:07483b63eff366156d3645c3d3ed724e10cdd81b5bc378efb1451001659906d6 11.89MB / 11.89MB
 ⇒ ⇒ sha256:3d52c1551390bb88a11c98fc8a6290653f437f1d039e3c000cb8f4e346a0c013 244B / 244B
 ⇒ ⇒ sha256:732bf57bb511752a98a753bf12cecfeff2c702fe8608f022ccf88c10097129fb 3.13MB / 3.13MB
 ⇒ ⇒ extracting sha256:1f7ce2fa46ab3942feabee654933948821303a5a821789dddab2d8c3df59e227
 ⇒ ⇒ extracting sha256:1fb7efcf9eab7803298874aca4438f97958ccef72e9d62bf6c7654b5d9c92c40
 ⇒ ⇒ extracting sha256:07483b63eff366156d3645c3d3ed724e10cdd81b5bc378efb1451001659906d6
 ⇒ ⇒ extracting sha256:3d52c1551390bb88a11c98fc8a6290653f437f1d039e3c000cb8f4e346a0c013
 ⇒ ⇒ extracting sha256:732bf57bb511752a98a753bf12cecfeff2c702fe8608f022ccf88c10097129fb
 ⇒ [internal] load build context
 ⇒ ⇒ transferring context: 1.70kB
 ⇒ [2/5] WORKDIR /usr/src/app
 ⇒ [3/5] COPY requirements.txt ./
 ⇒ [4/5] RUN pip install --no-cache-dir -r requirements.txt
 ⇒ [5/5] COPY load_generator.py .
 ⇒ exporting to image
 ⇒ ⇒ exporting layers
 ⇒ ⇒ writing image sha256:349ba46ebd0daad65141211484408084d6c3adff679d47d7b41b3592304b3c7d
 ⇒ ⇒ naming to localhost:32000/load-generator
```

Figure 3 - Docker Build



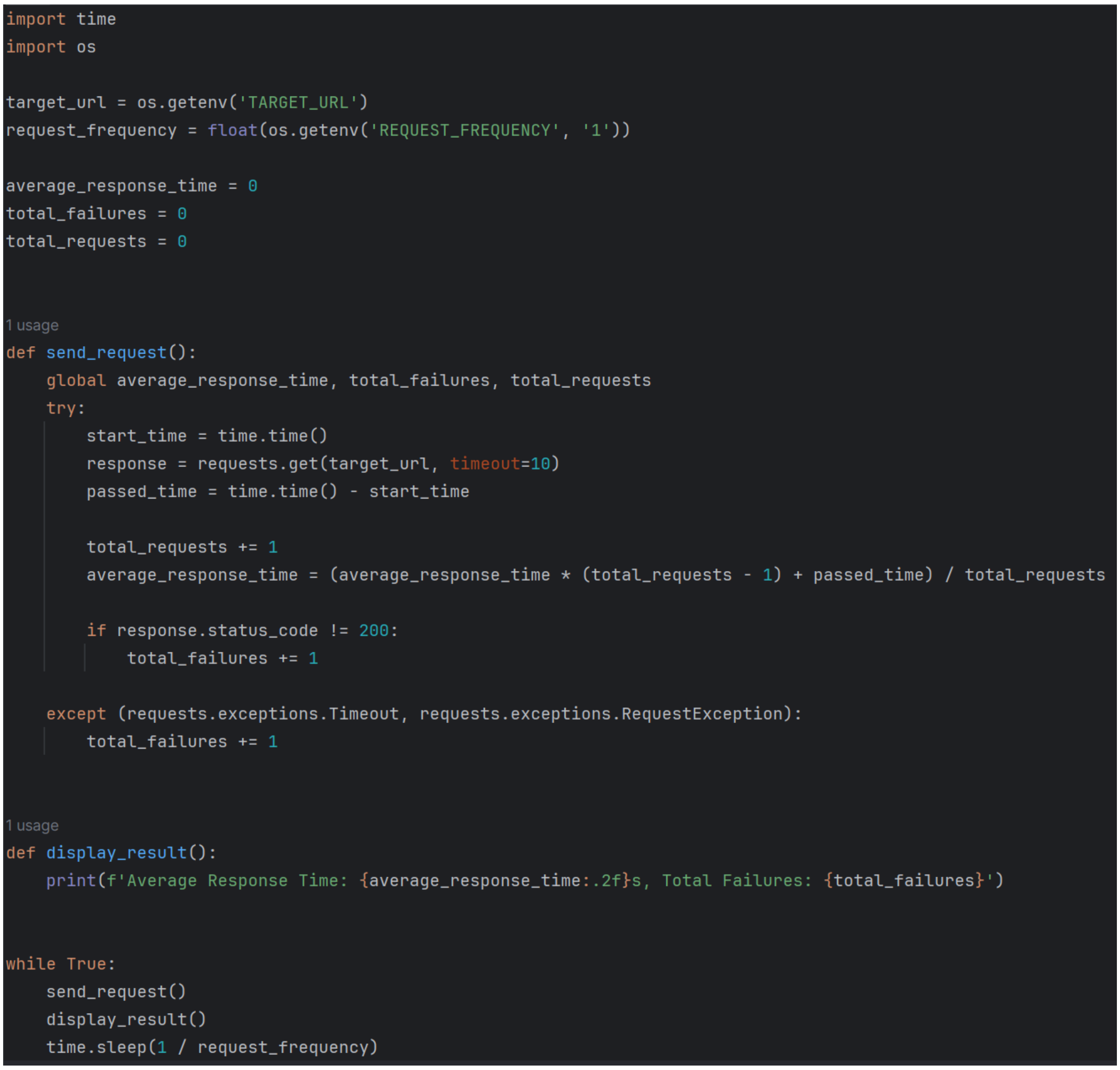
```
student@edge:~/cloud_project$ docker push localhost:32000/load-generator
Using default tag: latest
The push refers to repository [localhost:32000/load-generator]
11d8621c3ea6: Pushed
9866ef529c6f: Pushed
87a2f2aa8188: Pushed
2a5e84086989: Pushed
815cd871c032: Pushed
3d4cbcb22a1f: Pushed
fb2b91f4a3e4: Pushed
9dd8dac9def3: Pushed
92770f546e06: Pushed
latest: digest: sha256:668d8c42210fcb32998f5db1ad9edd0931b8a0175c172bbd729a187bff67036c size: 2202
```

Figure 4 - Docker Push



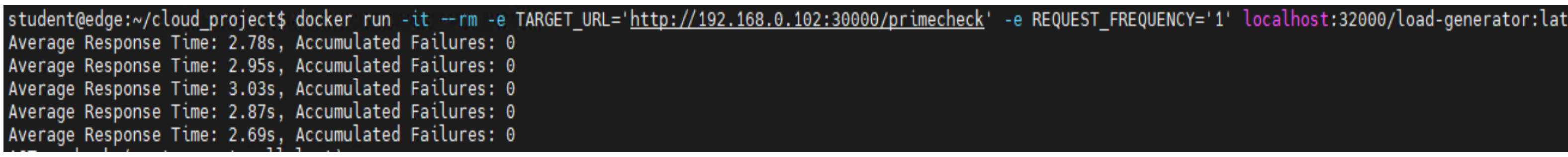
```
student@edge:~/cloud_project$ docker ps -a | grep load-generator
93ff93449ae1   localhost:32000/load-generator   "python ./load_gener…"   2 minutes ago      Up 2 minutes
                                      load-generator
student@edge:~/cloud_project$ docker ps -a | grep load-generator
93ff93449ae1   localhost:32000/load-generator   "python ./load_gener…"   2 minutes ago      Up 2 minutes
                                      load-generator
```

Figure 5 - "docker ps" command output



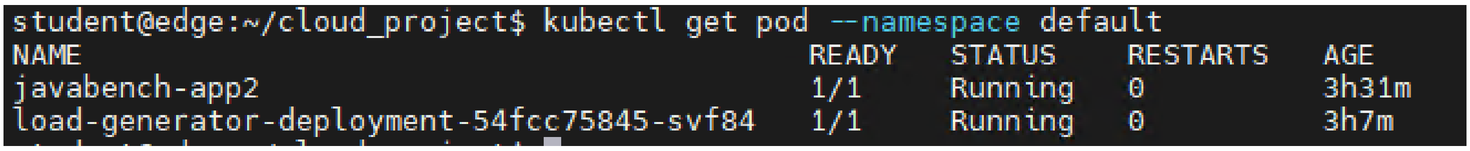
```python
import time
import os

target_url = os.getenv('TARGET_URL')
request_frequency = float(os.getenv('REQUEST_FREQUENCY', '1'))

average_response_time = 0
total_failures = 0
total_requests = 0


1 usage
def send_request():
    global average_response_time, total_failures, total_requests
    try:
        start_time = time.time()
        response = requests.get(target_url, timeout=10)
        passed_time = time.time() - start_time

        total_requests += 1
        average_response_time = (average_response_time * (total_requests - 1) + passed_time) / total_requests

        if response.status_code != 200:
            total_failures += 1

    except (requests.exceptions.Timeout, requests.exceptions.RequestException):
        total_failures += 1


1 usage
def display_result():
    print(f'Average Response Time: {average_response_time:.2f}s, Total Failures: {total_failures}')


while True:
    send_request()
    display_result()
    time.sleep(1 / request_frequency)
```

Figure 6 - Load Generator Code



```
student@edge:~/cloud_project$ docker run -it --rm -e TARGET_URL='http://192.168.0.102:30000/primecheck' -e REQUEST_FREQUENCY='1' localhost:32000/load-generator:lat
Average Response Time: 2.78s, Accumulated Failures: 0
Average Response Time: 2.95s, Accumulated Failures: 0
Average Response Time: 3.03s, Accumulated Failures: 0
Average Response Time: 2.87s, Accumulated Failures: 0
Average Response Time: 2.69s, Accumulated Failures: 0
```

Figure 7 - Output Example of Load Generator



```
student@edge:~/cloud_project$ kubectl get pod --namespace default
NAME                                      READY   STATUS    RESTARTS   AGE
javabench-app2                            1/1     Running   0          3h31m
load-generator-deployment-54fcc75845-svf84   1/1     Running   0          3h7m
```

Figure 8 - Running Pods

2.5 Monitoring CPU and Memory Usage with Grafana

In this section we were requested to create 2 Grafana panels that will display the CPU and Memory usage while our Java application in under the synthetic load generated by our load-generator application. It should be mentioned that our VM has 4-core CPU and 8 GB of RAM. Both graphs are created using the ***container_cpu_usage_seconds_total*** and ***container_memory_usage_bytes*** with a recommended prefix function for more detailed analysis.

From both graphs we can definitely observe that targeted VM in under serios load. As it is clearly shown in the **CPU Usage** graph, after starting the synthetic load, more than 80% of the CPU resource is used. Same also can be stated for the RAM usage referencing to the **Memory Usage** graph. The initial increase in both graph is instant because of the implemented logic in our load balancer application.

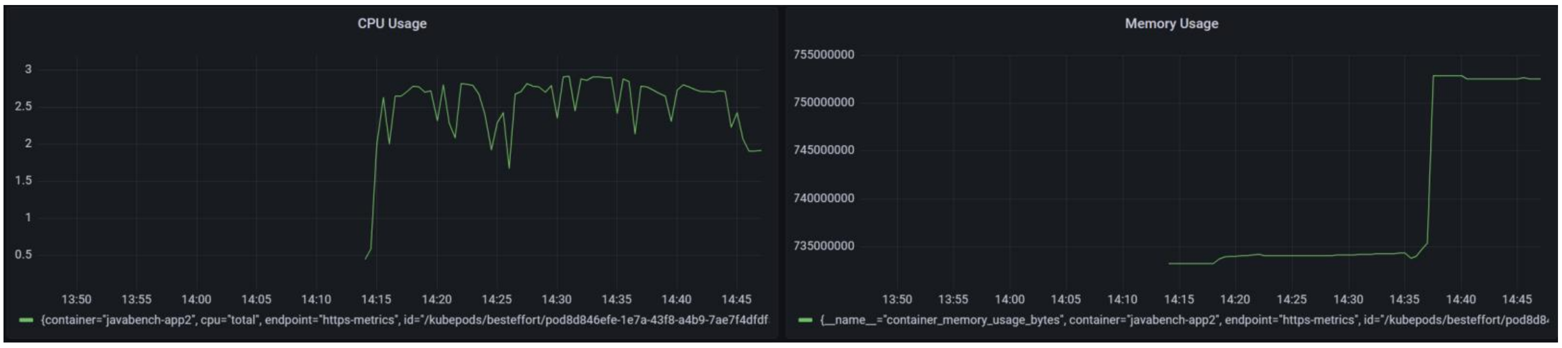


Figure 9 - Grafana Panels

3 Conclusion

As a conclusion, I want to state that all above explanations are made in the assumption that all required dependencies are installed or pulled properly. Unshowed parts of the project are same as it stated in the official documentations and no analytical approach are done there. The main intension of this report is mainly about to emphasize the cases in which different approaches are implemented for the successful result. Nevertheless, I tried to put screenshots that will show the execution flow of the project. Final results of the system are same as it intended to be. The system answers all the requirements in the way that it was provided in the assessment documentation.