Module: Cloud Computing - CSC8110

Yash Gadodia – 240679307

Table of Contents

Aim:

This coursework focuses on developing a comprehensive understanding and practical expertise in configuring, deploying, and managing Kubernetes-based application hosting environments. It integrates theoretical knowledge with practical skills to prepare for real-world scenarios in cloud virtualization technologies and containerized application management. Through the successful completion of Tasks 1-4 and their subsequent demonstration, hands-on experience will be gained in:

- Configuring Kubernetes environments for application hosting.
- Building, pushing, and pulling Docker images from Docker Hub.
- Creating and deploying a multi-component web application stack on Kubernetes.
- Monitoring application stack performance using Kubernetes-native tools.

Implementation:

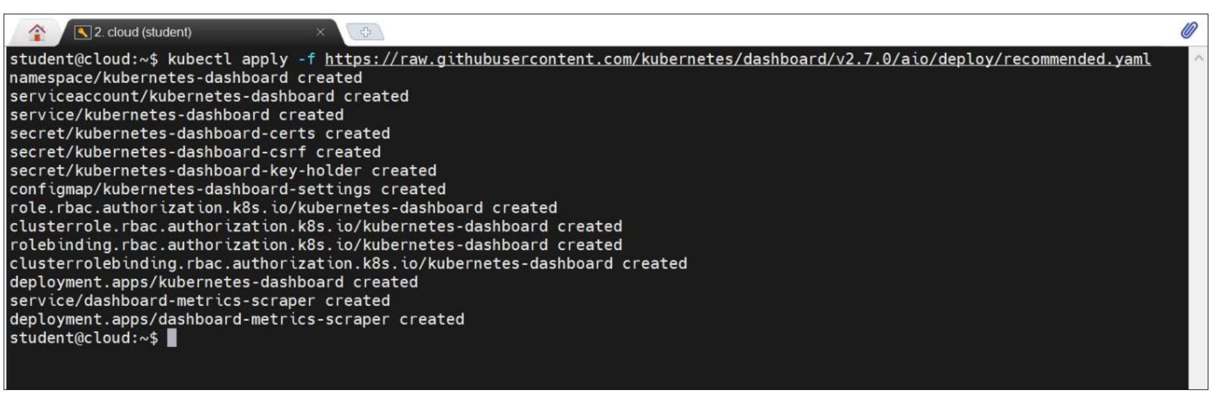
Task 1: Deploy and access the Kubernetes Dashboard and a Web Application Component

Task Objective: Understand and learn basic concepts of Kubernetes, and how to deploy applications to a Kubernetes cluster.

A sample *Java web application component* image, named "nclcloudcomputing/javabenchmarkapp", has been uploaded to the Docker Hub. The image contains a ready-to-use implementation of a web application deployed in a Tomcat server (an open-source web server). In terms of computational logic, the web application implements a prime number check on a large number. By doing so, the application can generate high CPU and memory load.

a) Deploy 'Kubernetes Dashboard' on the provided VM with CLI and access/login the Dashboard.

We use the following command to deploy the Kubernetes Dashboard.



```
student@cloud:~$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
student@cloud:~$
```

After deploying the Kubernetes Dashboard, follow these steps to access it:

First, a service account named admin-user is created in the kubernetes-dashboard namespace. This is done by applying the configuration using the kubectl apply -f kubernetes-admin-user.yaml command.

Next, a ClusterRoleBinding is set up to associate the cluster-admin ClusterRole with the admin-user ServiceAccount in the kubernetes-dashboard namespace. This ensures that the admin-user ServiceAccount has superuser privileges within the namespace. This binding is applied using the kubectl apply -f cluster_role_binding.yaml command.

cluster_role_blindin...

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

admin-user.yaml

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
```

The configuration of these YAML files is illustrated below.

7. cloud (student) | 8. cloud (student)

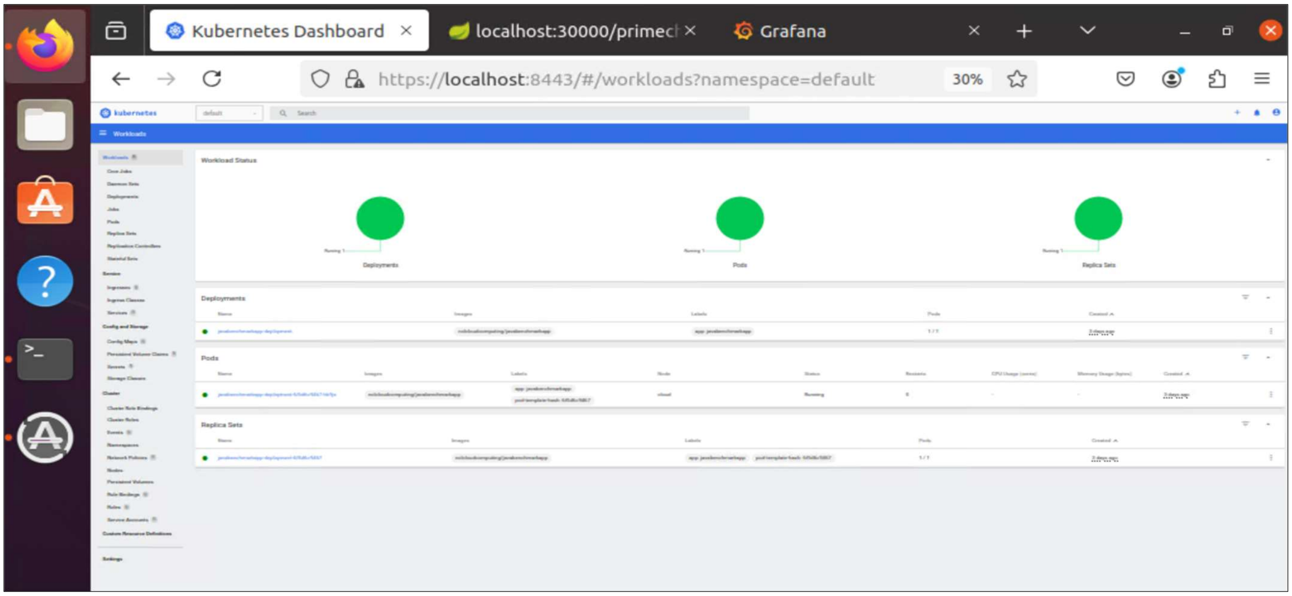
```
student@cloud:~$ kubectl apply -f admin-user.yaml
serviceaccount/admin-user unchanged
student@cloud:~$ kubectl apply -f cluster_role_blinding.yaml
clusterrolebinding.rbac.authorization.k8s.io/admin-user created
student@cloud:~$ kubectl apply -f s
service_account_token.yaml  snap/
student@cloud:~$ kubectl apply -f service_account_token.yaml
secret/admin-user created
student@cloud:~$
```

Finally, to access the Kubernetes Dashboard, a token is generated for the admin-user using the command kubectl -n kubernetes-dashboard create token admin-user

2. cloud (student)

```
student@cloud:~$ kubectl -n kubernetes-dashboard create token admin-user
eyJhbGciOiJSUzI1NiIsImtpZCI6ImtGTDhBQkRJc1o1bGs4OHhEMGptWHB5SXlkcjAzTHlGNTFqbjVLU2dRTmMifQ.eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF
1bHQuc3ZjIl0sImV4cCI6MTczMzA4NDQwMiwiaWF0IjoxNzMzMDgwODAyLCJpc3MiOiJodHRwczovL2t1YmVybmV0ZXMuZGVmYXVsdC5zdmMiLCJrdWJlcm5ldGVzLmlvIjp7I
m5hbWVzcGFjZSI6Imt1YmVybmV0ZXMtZGFzaGJvYXJkIiwic2VydmljZWFjY291bnQiOnsibmFtZSI6ImFkbWluLXVzZXIiLCJ1aWQiOiI2ZGRiMmNiZS0wZjgyLTRlOWEtOWJ
jNy1kYThiNWJhMTI1OWIifX0sIm5iZiI6MTczMzA4MDgwMiwic3ViIjoic3lzdGVtOnNlcnZpY2VhY2NvdW50Omt1YmVybmV0ZXMtZGFzaGJvYXJkOmFkbWluLXVzZXIifQ.SL
gHgtPpLUKBL3cXKrRylsRm5FLoBxvhTgi43RbTIe_cquKxi28E9z0BYJD_Ik4_df3zW9YlMoOn2aWn71MahjNByvquXoL_jCdgDBYwnh3HYGOXY-u2WONkiraDIPmLXIHRjViC
VTo0h4ZshxETjYTDOMj8t61ZsBy5Q81bbCcYh7INSk9U5gjXsQpQstMxjDh-g8p2ni1l2a1YL2z7s1ulVKrJTNQBvLpflvLfd-Xp3vE3FbhyR_GLRxrICeSQ6r43eqRBdKv-68
jnFrtm24d6yVvJFb-qd12O-KLhda9JjPAU2YHu0sG0vjI01hXpYtVT6eNmGYZHjvXcfVDnlw
```

The Kubernetes Dashboard interface is displayed below.



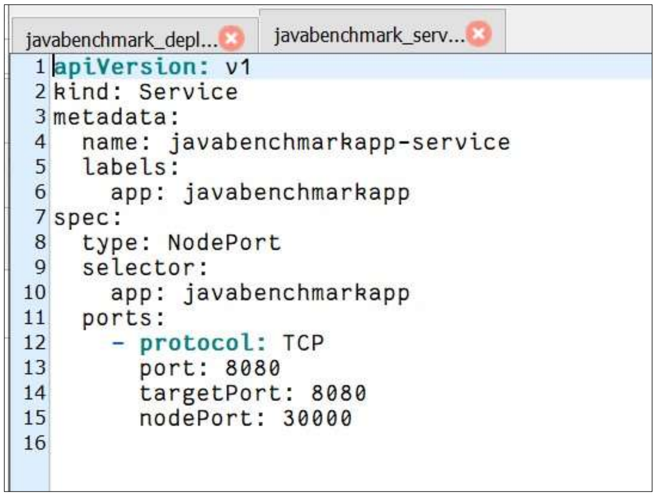
b) Deploy an instance of the Docker image "nclcloudcomputing/javabenchmarkapp" via CLI.

First, create a Kubernetes Deployment configuration in a YAML file to define how to deploy the Docker image and set up the required Kubernetes resources. To deploy the container from the nclcloudcomputing/javabenchmarkapp image in the Kubernetes cluster, run the following command: kubectl apply -f javabenchmark_deployment.yaml

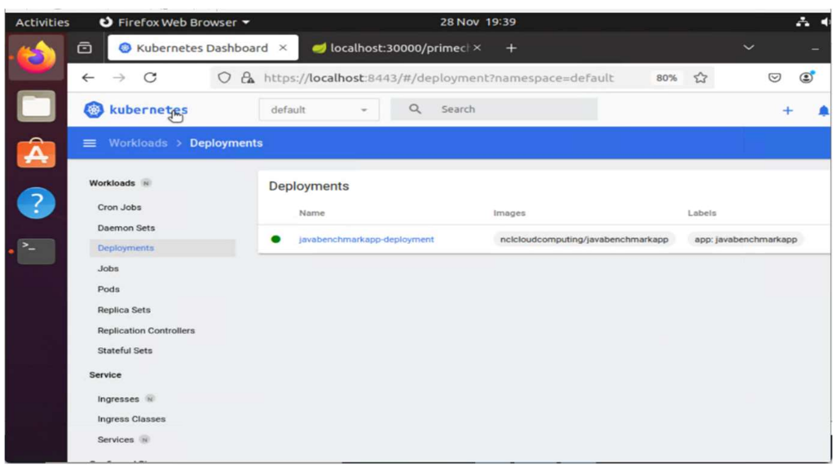
javabenchmark_depl...

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: javabenchmarkapp-deployment
  labels:
    app: javabenchmarkapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: javabenchmarkapp
  template:
    metadata:
      labels:
        app: javabenchmarkapp
    spec:
      containers:
        - name: javabenchmarkapp
          image: nclcloudcomputing/javabenchmarkapp
          ports:
            - containerPort: 8080
```

Next, define a Kubernetes Service in a YAML file to expose the application to external traffic using a NodePort. To make the application accessible externally, run: kubectl apply -f javabenchmark_service.yaml

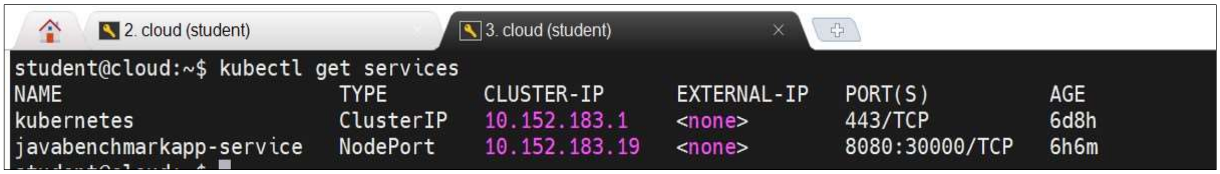


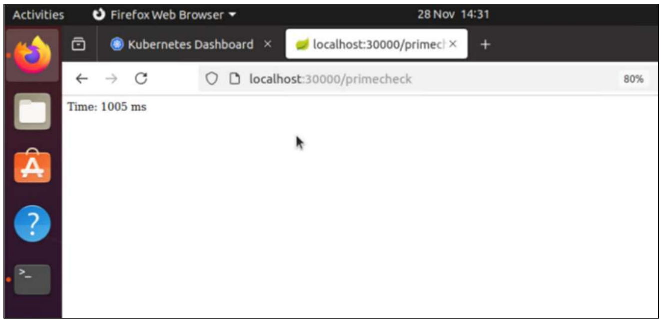
The command kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443 is used to forward traffic from a local port (8443) to the Kubernetes service port (443), enabling access to the Kubernetes Dashboard



c) Deploy a NodePort service so that the web app is accessible via http://localhost:30000/primecheck.

A service named javabenchmarkapp-service was created using the command provided below. In the configuration file, the service type is set to NodePort to make the application accessible outside the cluster.





d) You cannot use the dashboard addon from microk8s.

Task 2: Deploy the monitoring stack of Kubernetes

Task Objective: Understand and learn how to deploy a monitoring stack of Kubernetes consisting of Prometheus, metrics server, Grafana.

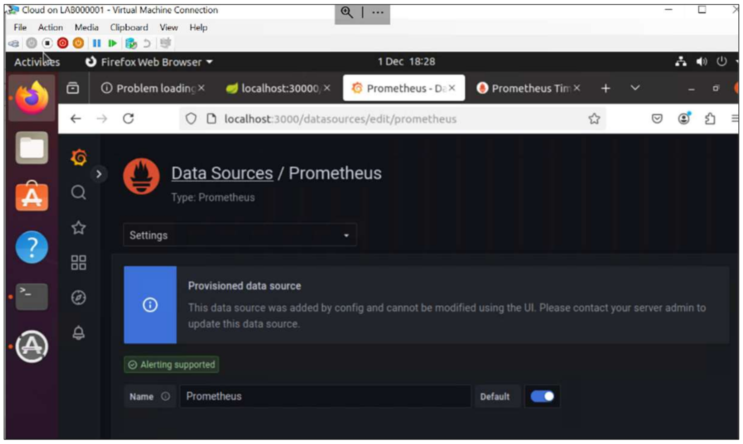
MicroK8s provides an easy way to enable observability features, such as Prometheus, Grafana, and the metrics server. To start with, I enabled the necessary MicroK8s addons for Prometheus and Grafana using the following command: microk8s enable observability

```
2. cloud (student)
student@cloud:~$ microk8s enable observability
Infer repository core for addon observability
Addon core/dns is already enabled
Addon core/helm3 is already enabled
Addon core/hostpath-storage is already enabled
Enabling observability
Release "kube-prom-stack" does not exist. Installing it now.
NAME: kube-prom-stack
LAST DEPLOYED: Fri Nov 29 13:34:37 2024
NAMESPACE: observability
STATUS: deployed
REVISION: 1
NOTES:
kube-prometheus-stack has been installed. Check its status by running:
  kubectl --namespace observability get pods -l "release=kube-prom-stack"

Visit https://github.com/prometheus-operator/kube-prometheus for instructions on how to create & configure Alertmanager and
 Prometheus instances using the Operator.
Release "loki" does not exist. Installing it now.
NAME: loki
LAST DEPLOYED: Fri Nov 29 13:35:09 2024
NAMESPACE: observability
STATUS: deployed
REVISION: 1
NOTES:
The Loki stack has been deployed to your cluster. Loki can now be added as a datasource in Grafana.

See http://docs.grafana.org/features/datasources/loki/ for more detail.
Release "tempo" does not exist. Installing it now.
NAME: tempo
LAST DEPLOYED: Fri Nov 29 13:35:15 2024
NAMESPACE: observability
STATUS: deployed
```

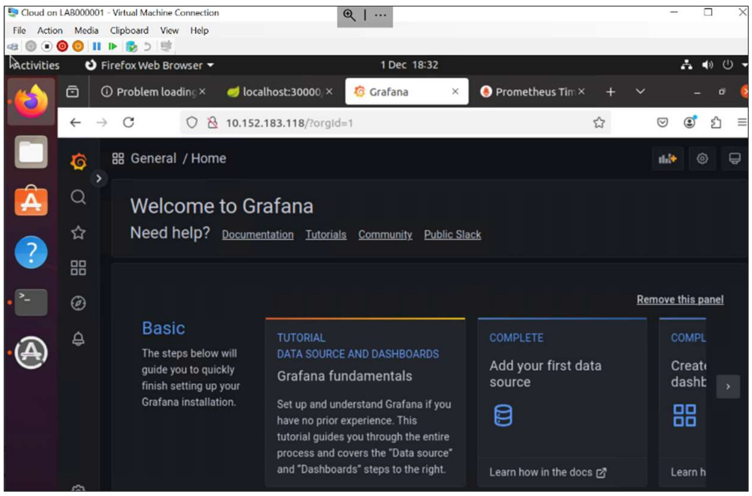
```
student@cloud:~$ microk8s kubectl get pods -n observability
NAME                                                 READY   STATUS    RESTARTS       AGE
kube-prom-stack-kube-prome-operator-64ffd55b77-kgvgv 1/1     Running   0              6m12s
kube-prom-stack-prometheus-node-exporter-96lpd       1/1     Running   0              6m13s
kube-prom-stack-kube-state-metrics-6c586bf4c8-nvjhz  1/1     Running   0              6m12s
kube-prom-stack-grafana-6c47f548d6-kj5k4             3/3     Running   0              6m13s
alertmanager-kube-prom-stack-kube-prome-alertmanager-0 2/2   Running   1 (5m7s ago)   5m53s
loki-promtail-25cv4                                  1/1     Running   0              5m54s
tempo-0                                              2/2     Running   0              5m49s
prometheus-kube-prom-stack-kube-prome-prometheus-0   2/2     Running   0              5m53s
loki-0                                               1/1     Running   0              5m54s
```

Explanation of Task2: By default, the Grafana and Prometheus services are not directly accessible from the host system. To make these services reachable, I used the kubectl port-forward command to expose them to specific ports on the local machine. For Prometheus, I forwarded port 9090 by running the command: microk8s kubectl port-forward -n observability service/kube-prom-stack-kube-prome-prometheus 9090:9090. Similarly, for Grafana, I forwarded port 3000 with the following command: microk8s kubectl port-forward -n observability service/kube-prom-stack-grafana 3000:80. I attach the snip of all the outputs below.

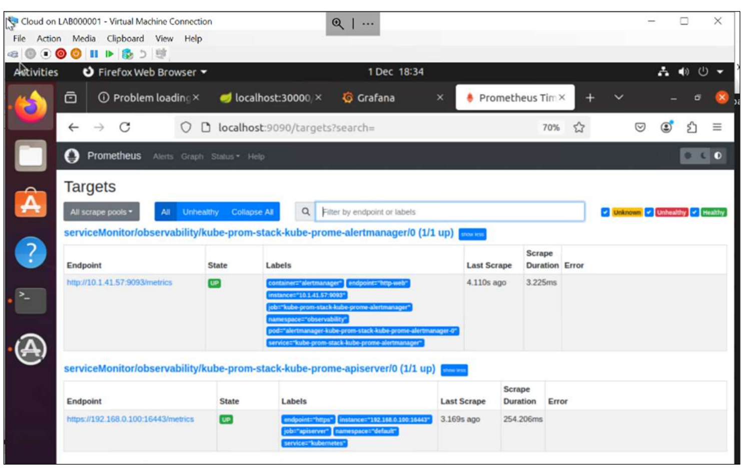
a) Grafana must reach Prometheus as a data source



b) Grafana must be reachable from the host



c) Prometheus must reach each instance of the metric server to collect metrics



d) The metric server must be deployed as a Daemon Set

```
student@cloud:~$ microk8s kubectl get deployments,daemonsets -A | grep -E "kube-state-metrics|node-exporter"
observability          deployment.apps/kube-prom-stack-kube-state-metrics     1/1     1            1            2d5h
observability   daemonset.apps/kube-prom-stack-prometheus-node-exporter   1         1         1         1            1           <none>
          2d5h
```

Task 3: Load Generator

Task Objective: Understand the logic of the load generator of benchmarking web applications and the process of deploying your own application of the cluster. Additionally, understand how to build and push a Docker image from scratch.

a) Write a load generator with the following specifications
   a. Accepts two configurable values either via a config file or environment variables. target (The address for the load generation) and frequency (Request per second)
   b. Generate web request to the target at the specified frequency
   c. Collect 2 types of metrics. Average response time and accumulated number of failures
   d. Request should timeout if it takes more than 10 seconds. Counted as failures
   e. Test results need to be printed to the console
   f. There are no requirements in programming language

* load_generator.py

```python
import requests
import time
import os

class LoadGenerator:
    def __init__(self, target=None, frequency=None):
        self.target = target or os.environ.get("target", "http://192.168.0.100:30000/primecheck")
        self.frequency = frequency or float(os.environ.get("Frequency", 10.0))
        self.failures = 0
        self.total_response_time = 0
        self.total_requests = 0

    def perform_request(self): """HTTP GET request to the target."""
        try:
            response = requests.get(self.target, timeout=10)
            response.raise_for_status()
            return response
        except requests.exceptions.RequestException as e:
            print(f"Request failed: {e}")
            self.failures += 1
            return None

    def calculate_response_time(self, start_time): """Calculate and return the response time."""
        end_time = time.time()
        return end_time - start_time

    def update_metrics(self, response_time): """Update the metrics for total response time and total requests."""
        self.total_response_time += response_time
        self.total_requests += 1

    def print_metrics(self): """Print the collected metrics."""
        if self.total_requests > 0:
            avg_response_time = self.total_response_time / self.total_requests
            print(f"Average Response Time: {avg_response_time:.2f} seconds")
            print(f"Total Failures: {self.failures}")
            print(f"Total Requests: {self.total_requests}")

    def generate_load(self): """Generate load by repeatedly performing requests."""
        while True:
            start_time = time.time()
            response = self.perform_request()

            if response:
                response_time = self.calculate_response_time(start_time)
                self.update_metrics(response_time)
                print(f"Request successful. Response Time: {response_time:.2f} seconds")

            time.sleep(1 / self.frequency)
            self.print_metrics()

if __name__ == "__main__":
    load_generator = LoadGenerator()
    load_generator.generate_load()
```

This Python script functions as a load generator that sends HTTP GET requests to a specified target URL at a user-defined frequency. The target URL and request frequency can be configured through environment variables, with fallback default values if the variables are not set. The script ensures requests are sent at the desired rate by using time.sleep (1 / frequency). It monitors two main metrics: the average response time, which is calculated by dividing the total response time by the number of successful requests, and the total failures, which increases each time a request times out or encounters an error. To manage timeouts, the requests.get() method is set with a 10-second timeout, and any request that exceeds this limit is treated as a failure. At regular intervals, the script outputs test results to the console, including the average response time, the number of failures, and the total number of requests, allowing for real-time performance monitoring of the target system during the load test.

b) After programming, pack the program as a standalone Docker image and push it to the local registry at port 32000. Name the image as *load-generator*.

First, we start a Docker registry on our host, running on port 32000, using the command: docker run -d -p 32000:5000 --name registry.

```
student@cloud:~$ docker run -d -p 32000:5000 --name registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry
dc0decf4841d: Pull complete
6cb0aa443e23: Pull complete
813676e291ef: Pull complete
dc2fb7dcec61: Pull complete
916205650bfe: Pull complete
Digest: sha256:543dade69668e02e5768d7ea2b0aa4fae6aa7384c9a5a8dbecc2be5136079ddb
Status: Downloaded newer image for registry:2
e3e530ac9ecddc12e787d8e66119095f83755da257e8be3400b9fd3a96c7e249
student@cloud:~$ docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED          STATUS         PORTS
       NAMES
e3e530ac9ecd   registry:2   "/entrypoint.sh /etc…"   12 seconds ago   Up 9 seconds   0.0.0.0:32000->5000/tcp, :::32000->500
0/tcp   registry
```

Next, we build the load-generator Docker image with the following command: docker build -t load-generator . .

dockerfile

```
# Use the official Python 3.9 slim image as the base image
FROM python:3

# Set the working directory inside the container
WORKDIR /app

# Copy the Python script into the container
COPY load_generator.py .

# Install the required Python library
RUN pip install requests

EXPOSE 30000

# Set the default command to execute the load generator script
CMD ["python", "load_generator.py"]

```

```
student@cloud:~$ docker build -t load-generator .
Sending build context to Docker daemon  240.3MB
Step 1/5 : FROM python:3
 ---> c41ea8273365
Step 2/5 : WORKDIR /app
 ---> Running in 3620ff109dc6
Removing intermediate container 3620ff109dc6
 ---> d8bef7fa704a
Step 3/5 : COPY load_generator.py .
 ---> a5ab1abda056
Step 4/5 : RUN pip install requests
 ---> Running in dd964cd14f68
Collecting requests
  Downloading requests-2.32.3-py3-none-any.whl.metadata (4.6 kB)
Collecting charset-normalizer<4,>=2 (from requests)
  Downloading charset_normalizer-3.4.0-cp313-cp313-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (34 kB)
Collecting idna<4,>=2.5 (from requests)
  Downloading idna-3.10-py3-none-any.whl.metadata (10 kB)
Collecting urllib3<3,>=1.21.1 (from requests)
  Downloading urllib3-2.2.3-py3-none-any.whl.metadata (6.5 kB)
Collecting certifi>=2017.4.17 (from requests)
  Downloading certifi-2024.8.30-py3-none-any.whl.metadata (2.2 kB)
Downloading requests-2.32.3-py3-none-any.whl (64 kB)
Downloading certifi-2024.8.30-py3-none-any.whl (167 kB)
Downloading charset_normalizer-3.4.0-cp313-cp313-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (144 kB)
Downloading idna-3.10-py3-none-any.whl (70 kB)
Downloading urllib3-2.2.3-py3-none-any.whl (126 kB)
Installing collected packages: urllib3, idna, charset-normalizer, certifi, requests
Successfully installed certifi-2024.8.30 charset-normalizer-3.4.0 idna-3.10 requests-2.32.3 urllib3-2.2.3
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package ma
nager, possibly rendering your system unusable.It is recommended to use a virtual environment instead: https://pip.pypa.io/wa
rnings/venv. Use the --root-user-action option if you know what you are doing and want to suppress this warning.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: pip install --upgrade pip
Removing intermediate container dd964cd14f68
 ---> fbe94d1f7fec
Step 5/5 : CMD ["python", "load_generator.py"]
```

After building the image, we can see load-generator listed in the output of the docker images command.

```
student@cloud:~$ docker images
REPOSITORY       TAG       IMAGE ID       CREATED          SIZE
load-generator   latest    9c5568a1b457   44 seconds ago   1.03GB
python           3         c41ea8273365   6 weeks ago      1.02GB
registry         2         c18a86d35e98   14 months ago    25.4MB
```

Once the image is created, we tag it and push it to our Docker registry. To tag the image, we use the command: docker tag load-generator localhost:32000/load-generator. Finally, we push the image to the registry using: docker push localhost:32000/load-generator.

```
student@cloud:~$ docker tag load-generator localhost:32000/load-generator
student@cloud:~$ docker push localhost:32000/load-generator
Using default tag: latest
The push refers to repository [localhost:32000/load-generator]
ed2757717041: Pushed
953b03b0ea16: Pushed
20da8de59e62: Pushed
1e8b53dc33fa: Layer already exists
94878487f485: Layer already exists
20a1c75bf72e: Layer already exists
96d99c63b722: Layer already exists
00547dd240c4: Layer already exists
b6ca42156b9f: Layer already exists
24b5ce0f1e07: Layer already exists
latest: digest: sha256:7690962e920fd37a4df8367b3c450bcfa4041afa2a6b94fc6b2a2a95c9b3d530 size: 2420
```

After pushing, the docker images command will show both the load-generator and localhost:32000/load-generator images.

```
student@cloud:~$ docker images
REPOSITORY                        TAG       IMAGE ID       CREATED          SIZE
load-generator                    latest    9c5568a1b457   4 minutes ago    1.03GB
localhost:32000/load-generator    latest    9c5568a1b457   4 minutes ago    1.03GB
python                            3         c41ea8273365   6 weeks ago      1.02GB
registry                          2         c18a86d35e98   14 months ago    25.4MB
```

Task 4: Monitor benchmarking results

Task Objective: To learn and understand how to monitor container metrics.

a) Deploy *load-generator* service created in Task 3.

* load-generator.yaml

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: load-generator
spec:
  replicas: 1
  selector:
    matchLabels:
      name: load-generator
  template:
    metadata:
      name: testpod1
      labels:
        name: load-generator
    spec:
      containers:
        - name: c01
          image: localhost:32000/load-generator
          ports:
            - containerPort: 30000

---
kind: Service
apiVersion: v1
metadata:
  name: demoservice
spec:
  ports:
    - port: 30000
      targetPort: 30000
  selector:
    name: load-generator
  type: NodePort

```

```
student@cloud:~$ kubectl apply -f load-generator-deployment.yaml
deployment.apps/load-generator created
```

Above YAML file configures a Kubernetes Deployment and Service to deploy and expose a load generator application. The Deployment defines a single replica of a pod, labeled Load-generator, which runs a container named c01 with the image localhost:32000/load-generator. The container listens on port 30000. The Service, configured as a NodePort, exposes the pods on port 30000, enabling external access to the load generator. It links to the pods using the label selector name: Load-generator.

We deploy the load generator by configuring environment variable values that the load-generator code uses to repeatedly send requests to our JavaBenchmarkApp. The figure below shows the logs generated after deploying the service.

```
student@cloud:~$ kubectl logs load-generator-6556685d9d-pkpgp
Request successful. Response Time: 1.03 seconds
Average Response Time: 1.03 seconds
Total Failures: 0
Total Requests: 1
Request successful. Response Time: 0.86 seconds
Average Response Time: 0.94 seconds
Total Failures: 0
Total Requests: 2
Request successful. Response Time: 0.84 seconds
Average Response Time: 0.91 seconds
Total Failures: 0
Total Requests: 3
Request successful. Response Time: 0.92 seconds
Average Response Time: 0.91 seconds
Total Failures: 0
Total Requests: 4
Request successful. Response Time: 0.98 seconds
Average Response Time: 0.93 seconds
Total Failures: 0
Total Requests: 5
Request successful. Response Time: 0.97 seconds
Average Response Time: 0.93 seconds
Total Failures: 0
Total Requests: 6
Request successful. Response Time: 0.88 seconds
Average Response Time: 0.93 seconds
Total Failures: 0
Total Requests: 7
Request successful. Response Time: 0.86 seconds
Average Response Time: 0.92 seconds
Total Failures: 0
Total Requests: 8
Request successful. Response Time: 0.90 seconds
Average Response Time: 0.92 seconds
Total Failures: 0
Total Requests: 9
Request successful. Response Time: 0.91 seconds
```

b) During the benchmarking, create a new dashboard on Grafana and add 2 new panels which should contain queries of CPU/memory usage of the web application

I created a new dashboard on Grafana and added two new panels displaying queries for CPU and memory usage of the web application, which you can see in the figure below.

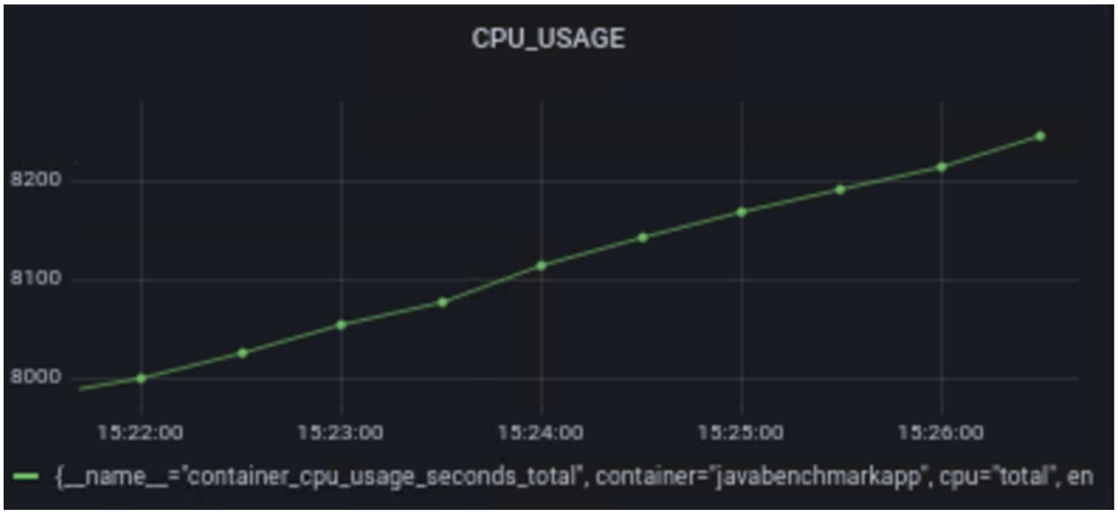
c) Screenshot the two panels

i) Cpu_Usage

The image below presents the metrics for container_cpu_usage_seconds_total, which illustrates a steady rise in CPU usage over time. The graph reflects an increase in CPU utilization as the web application starts handling requests. CPU usage only decreases when there is a request failure or an issue with the service. However, in this case, since there are no request failures, the graph shows a consistent upward trend.

Server Metric: container_cpu_usage_seconds_total
Label: container > javabenchmarkapp

Snapshot:

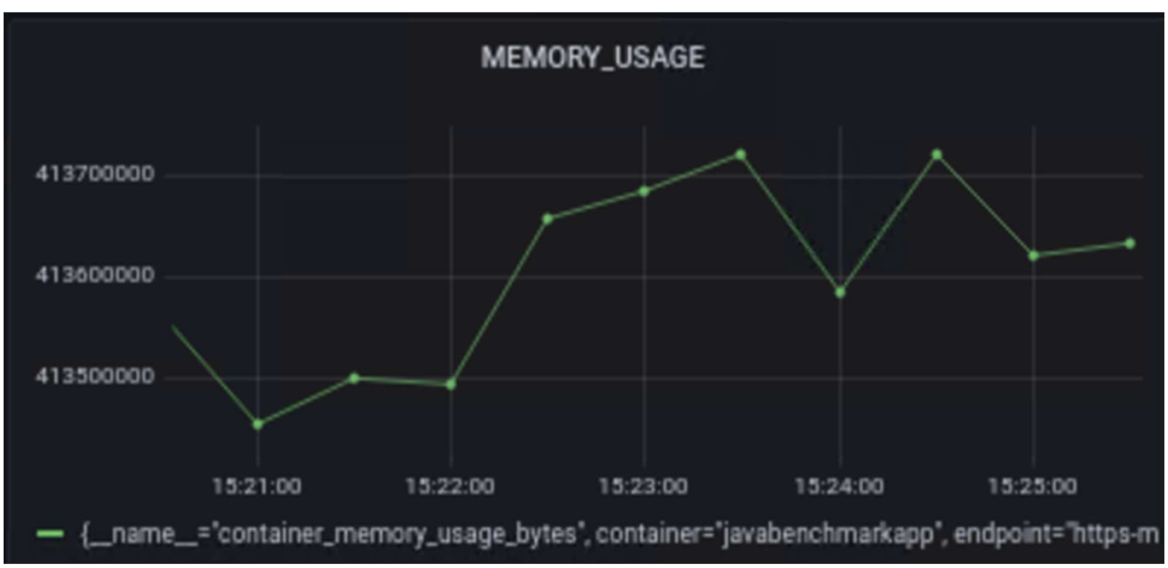


ii) Memory_usage

The graph below displays the container_memory_usage_bytes metric, illustrating a non-linear pattern of memory consumption over time. Memory usage is influenced by the behaviour of the web application. When the javabenchmarkapp service is down, the graph shows a decline in memory usage, while an increase in memory usage occurs when the service is operational.

Server Metric: container_memory_usage_bytes
Label: container > javabenchmarkapp

Snapshot:



Conclusion:

Throughout this coursework, we gained hands-on experience in deploying, managing, and monitoring applications within a Kubernetes environment. We also developed a load generation tool for benchmarking, which provided valuable insights into application performance and resource usage, helping to identify areas for optimization and scalability. By utilizing Grafana, we created a comprehensive dashboard to visualize CPU and memory usage, revealing a linear relationship between increased load and resource consumption. This exercise has enhanced our understanding of the impact of load on application performance and the importance of monitoring tools in optimizing cloud-based applications. Furthermore, it emphasized the need for effective scaling strategies to handle varying levels of demand.

References:

a) Kubernetes: Kubernetes Documentation and Access the Kubernetes Cluster Web UI Dashboard.
b) Stackoverflow: Stack Overflow, for various doubts related to implementation and errors