CLOUD COMPUTING – CSC8110 Coursework 2023-24 Project Report

Ronil Rodrigues
MSc Cloud Computing
230112209

Table of Contents

AIM:

This coursework helps you to learn and understand the fundamentals of programming and deploying Kubernetes-based (an emerging cloud virtualisation technology) application hosting environment. By successfully completing the coursework, you will be able to gain hands-on experience in the following interrelated aspects including:

• Configuring a Kubernetes-based application hosting environment;

• Building, pushing and pulling images from the Docker Hub (global repository of software components' images maintained by the developers);

• Creating and deploying a complex web application stack consisting of multiple software components (e.g., web server, monitoring, etc.);

IMPLEMENTATION:

TASK 1: DEPLOY AND ACCESS THE KUBERNETES DASHBOARD AND A WEB APPLICATION COMPONENT

A sample Java web application component image, name "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.

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

We deploy Kubernetes Dashboard using the below command.

```
student@edge:~$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
namespace/kubernetes-dashboard unchanged
serviceaccount/kubernetes-dashboard unchanged
service/kubernetes-dashboard unchanged
secret/kubernetes-dashboard-certs unchanged
secret/kubernetes-dashboard-csrf configured
Warning: resource secrets/kubernetes-dashboard-key-holder is missing the kubectl.kubernetes.io/last-applied-configuration annotation which is required by kubectl apply.
urces created declaratively by either kubectl create --save-config or kubectl apply. The missing annotation will be patched automatically.
secret/kubernetes-dashboard-key-holder configured
configmap/kubernetes-dashboard-settings unchanged
role.rbac.authorization.k8s.io/kubernetes-dashboard unchanged
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard unchanged
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard unchanged
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard unchanged
deployment.apps/kubernetes-dashboard unchanged
service/dashboard-metrics-scraper unchanged
deployment.apps/dashboard-metrics-scraper unchanged
```

Once the Dashboard is deployed, we need to follow the below steps to gain access to the Kubernetes dashboard.

dashboard.yaml

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

In the above file we define the service account with the name admin-user in namespace Kubernetes-dashboard, we execute the service using “kubectl apply –f Kubernetes-dashboard.yaml” command.

ClusterRoleBinding.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
```

ClusterRoleBinding binds the cluster-admin ClusterRole to the admin-user ServiceAccount in the kubernetes-dashboard namespace. This configuration grants superuser access within the kubernetes-dashboard namespace to the admin-user ServiceAccount. We execute the service using “kubectl apply –f ClusterRoleBinding.yaml” command.

Secret.yaml

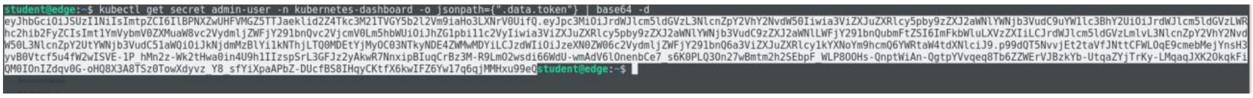
```
apiVersion: v1
kind: Secret
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
  annotations:
    kubernetes.io/service-account.name: "admin-user"
type: kubernetes.io/service-account-token
```

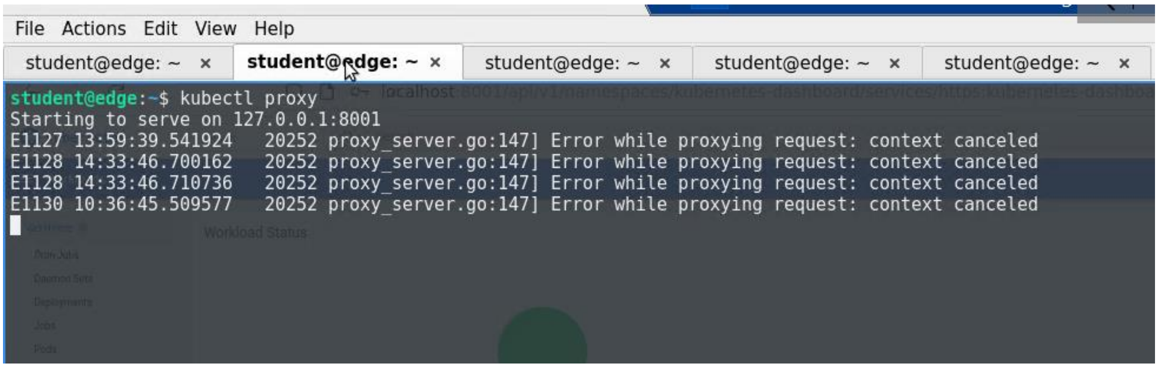
We can also create a token with the secret which bound the service account and the token will be saved in the Secret after applying the service using “kubectl apply –f secret.yaml” command.

The implementation of above yaml files can be seen in the below figure.

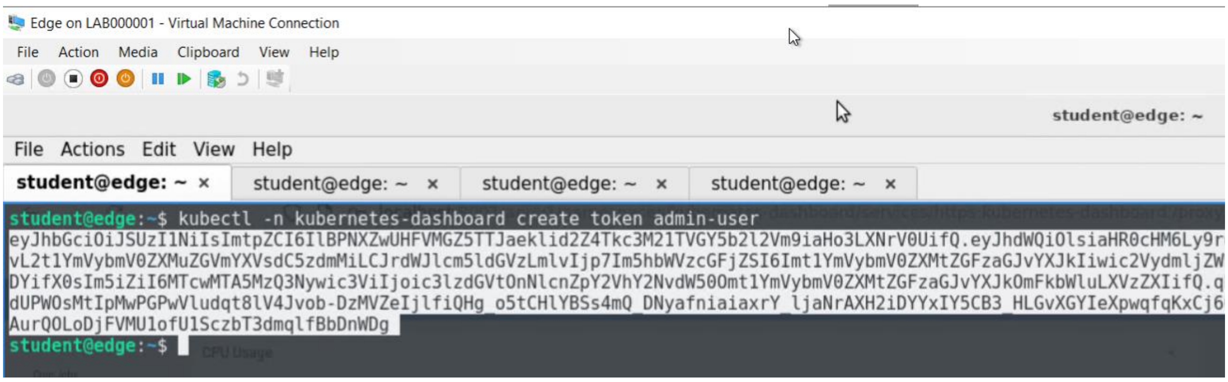
```
student@edge:~$ kubectl apply -f dashboard.yaml
serviceaccount/admin-user unchanged
student@edge:~$ kubectl apply -f ClusterRoleBinding.yaml
clusterrolebinding.rbac.authorization.k8s.io/admin-user unchanged
student@edge:~$ kubectl apply -f secret.yaml
error: the path "secret.yaml" does not exist
student@edge:~$ kubectl apply -f Secret.yaml
secret/admin-user unchanged
```

“kubectl get secret admin-user -n kubernetes-dashboard -o jsonpath={".data.token"} | base64 –d” command can be used to get the token which is saved in the Secret.





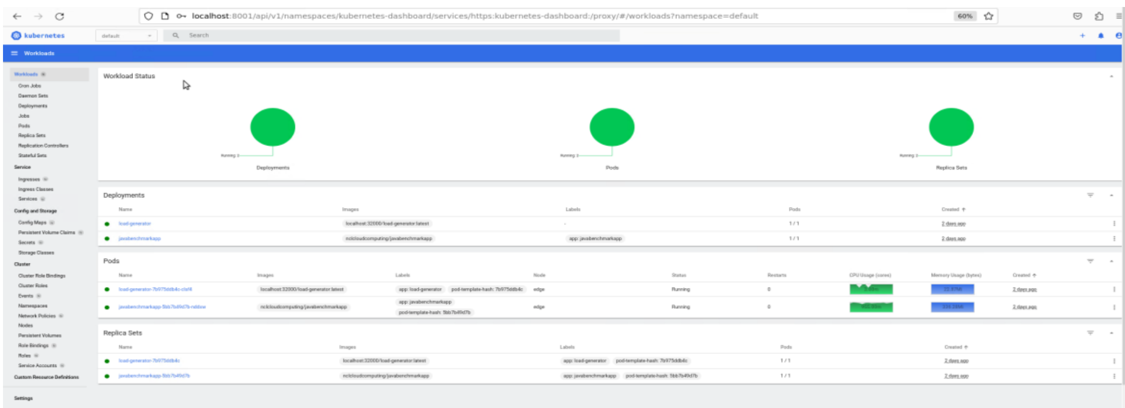
We enable the access to Kubernetes dashboard using “kubectl proxy” command



Alternatively we can create token to access Kubernetes dashboard using below command.

“kubectl –n Kubernetes-dashboard create token admin-user”

Kubernetes Dashboard interface is shown below.

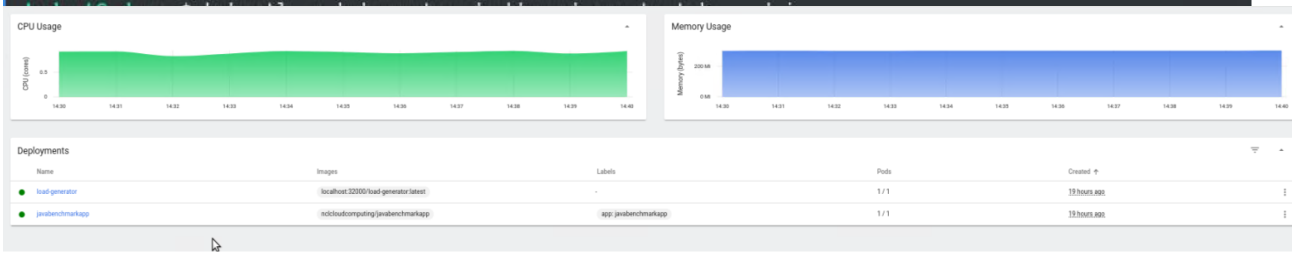


DEPLOY AN INSTANCE OF THE DOCKER IMAGE "NCLCLOUDCOMPUTING/JAVABENCHMARKAPP" VIA CLI.

We deploy docker image nclcloudcomputing/javabenchmarkapp using the command "kubectl create deployment javabenchmarkapp --image=nclcloudcomputing/javbenchmarkapp"

Where javabenchamarkapp is the deployment name.

```
student@edge:~$ kubectl create deployment javabenchmarkapp --image=nclcloudcomputing/javabenchmarkapp
deployment.apps/javabenchmarkapp created
```



DEPLOY A NODEPORT SERVICE SO THAT THE WEB APP IS ACCESSIBLE VIA HTTP://LOCALHOST:30000/PRIMECHECK. THE CONTAINER USES PORT 8080 INTERNALLY.

javabenchmarkapp-se...

```yaml
apiVersion: v1
kind: Service
metadata:
  name: javabenchmarkapp-service
spec:
  type:  NodePort
  selector:
    app:  javabenchmarkapp
  ports:
    - protocol:  TCP
      port: 8080
      targetPort: 8080
      nodePort: 30000
```

We created a service javabenchmarkapp-service by applying the below given command. In the above file we pass type as NodePort to expose the application service outside the cluster.

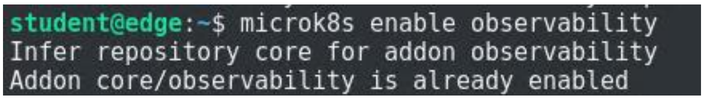
```
student@edge:~$ kubectl apply -f javabenchmarkapp-service.yaml
service/javabenchmarkapp-service created
student@edge:~$ kubectl get services
NAME                       TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
kubernetes                 ClusterIP   10.152.183.1     <none>        443/TCP          14d
javabenchmarkapp-service   NodePort    10.152.183.110   <none>        8080:30000/TCP   13s
```

TASK 2: DEPLOY THE MONITORING STACK OF KUBERNETES

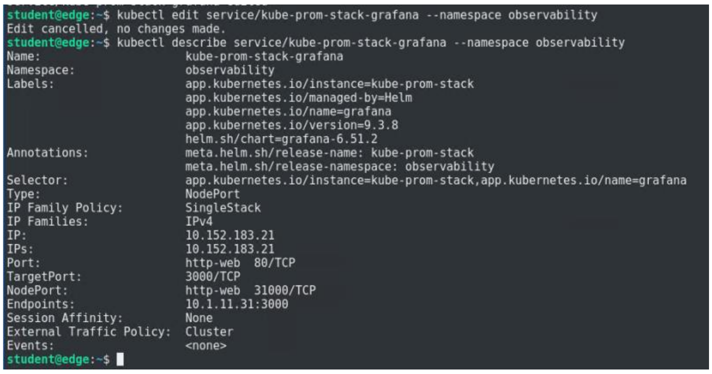
Task Objective: Understand and learn how to deploy a monitoring stack of Kubernetes consisting of Prometheus, metrics server, Grafana.

ENABLE OBSERVABILITY SERVICE FROM MICROK8S ADDONS.

Microk8s is a lightweight Kubernetes distributor which is used to get metrics and for monitoring purpose. The below command is used to enable the observability related add Ons for microk8s. The Observability feature includes addons like Grafana, Prometheus.



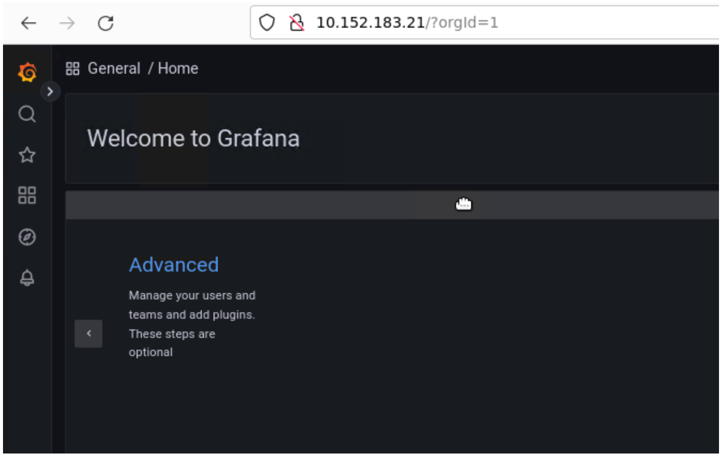
EDIT THE GRAFANA SERVICE TO ALLOW ACCESS FROM THE HOST.



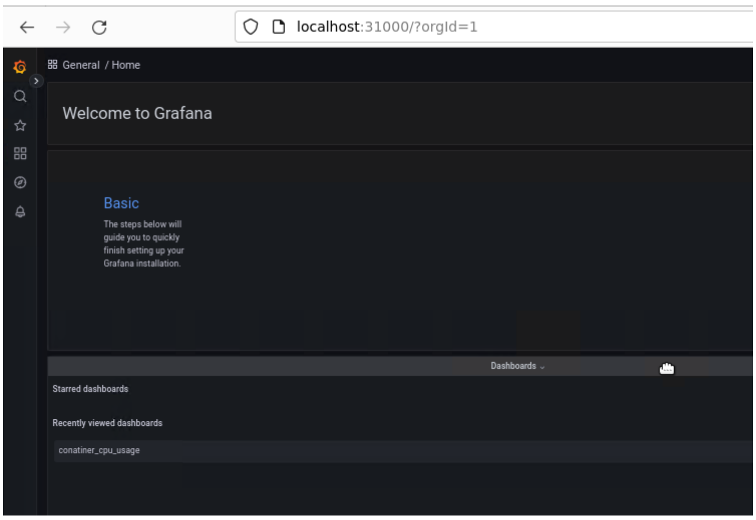
We edit the given Grafana monitoring service, where we change the type of service to NodePort to expose and allow access from the host. We use the command "kubectl edit service/Kuber-prom-stack-grafana –namespace observability" to edit the service.

LOG IN TO THE GRAFANA DASHBOARD.

Accessing Grafana service through IP.



Accessing Grafana monitoring service from the host port 31000.



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.

1. 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):
        self.target = os.environ.get("target", "http://10.152.183.110:8080/primecheck")
        self.frequency = float(os.environ.get("frequency", 10.0))
        self.failures = 0
        self.total_response_time = 0
        self.total_requests = 0

    def generate_load(self):
        while True:
            start_time = time.time()
            try:
                response = requests.get(self.target, timeout=10)
                response.raise_for_status()
            except requests.exceptions.RequestException as e:
                print(f"Request failed: {e}")
                self.failures += 1
            else:
                end_time = time.time()
                response_time = end_time - start_time
                self.total_response_time += response_time
                self.total_requests += 1
                print(f"Request successful. Response Time: {response_time:.2f} seconds")

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

    def print_metrics(self):
        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}\n")

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

The above load-generator code was created to benchmark our javabenchmarkapp. In the first part of the code, we accept target URL and frequency as input from environment variables, which is further defined in the task4 yaml file. If in case the code is unable to get inputs from our yaml file it would take predefined inputs as defined in the code. In this code we run and infinite loop of sending requests to our javabenchmarkapp to benchmark it. The code sends invoking request at an interval frequency of 10.0 as defined in the code. We record the response time for each request and also the average response time for all the requests invoked. We use .2f in print statement to restrict how floating point numbers are displayed in response time.

2.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

```
student@edge:~$ docker run -d -p 32000:5000 --restart=always --name registry registry:2
docker: Error response from daemon: Conflict. The container name "/registry" is already in use by container "eaa05bb13c1cf3463707bb9a63ed5505a464f7e77f0a52bdabd85331285cc365". You have to remove (or rena
t container to be able to reuse that name.
See 'docker run --help'.
student@edge:~$ docker build -t load-generator .
[+] Building 18.1s (9/9) FINISHED                                                                  docker:
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 163B
 => [internal] load metadata for docker.io/library/python:3
 => [1/4] FROM docker.io/library/python:3@sha256:31ceea009f42df76371a8fb94fa191f988a25847a228dbeac35b6f8d2518a6ef
 => [internal] load build context
 => => transferring context: 1.58kB
 => CACHED [2/4] WORKDIR /app
 => [3/4] COPY load_generator.py .
 => [4/4] RUN pip install requests
 => exporting to image
 => => exporting layers
 => => writing image sha256:40eba16cb58d1ddddb7914e52c8384e06e9add884490cdcd870b3123e1699481
 => => naming to docker.io/library/load-generator
student@edge:~$ docker ps -a
CONTAINER ID   IMAGE        COMMAND                  CREATED      STATUS       PORTS                                         NAMES
eaa05bb13c1c   registry:2   "/entrypoint.sh /etc…"   4 days ago   Up 3 hours   0.0.0.0:32000->5000/tcp, :::32000->5000/tcp   registry
```

Firstly we run docker registry on our host at port 32000 using the docker run command. In the next step we build a docker image of load-generator using the below docker file and running "docker build –t load-generator . " command.

Dockerfile

```
FROM python:3

WORKDIR /app

COPY load_generator.py .

RUN pip install requests

CMD ["python", "load_generator.py"]

```

Once we have created the docker image, we tag our image and deploy the same image to our docker registry. To tag the image we use "docker tag load-generator localhost:32000/load-generator" command. We push the image to docker registry using docker push command.

```
student@edge:~$ docker images
REPOSITORY                        TAG       IMAGE ID       CREATED          SIZE
load-generator                    latest    40eba16cb58d   38 seconds ago   1.03GB
localhost:32000/load-generator    latest    c8c67fcbf020   4 days ago       1.03GB
registry                          2         ff1857193a0b   5 weeks ago      25.4MB
student@edge:~$ docker image rm c8c6
Untagged: localhost:32000/load-generator:latest
Untagged: localhost:32000/load-generator@sha256:098aded9a68b43fe142470063d315cefff56584038ae6663f4b2e9b3880e5861
Deleted: sha256:c8c67fcbf0209fc108ea13f591cf3f6de876afd236320e61ea7e744166720941
student@edge:~$ docker tag load-generator localhost:32000/load-generator
student@edge:~$ docker push localhost:32000/load-generator
Using default tag: latest
The push refers to repository [localhost:32000/load-generator]
9b567ab688cd: Pushed
1dc53eb1ad48: Pushed
99aacc8cdfe0: Layer already exists
8a90026ddd52: Layer already exists
767b9eb445be: Layer already exists
e47f7ad06b01: Layer already exists
2b4cf8a5bd5e: Layer already exists
80bd043d4663: Layer already exists
30f5cd833236: Layer already exists
7c32e0608151: Layer already exists
7cea17427f83: Layer already exists
latest: digest: sha256:cf280eeee8304344f4f31cb44ad0539c934c9511abb3dfd8662b0eb8384b75c9 size: 2631
student@edge:~$ kubectl apply -f load-generator.yaml
deployment.apps/load-generator created
```

TASK 4: MONITOR BENCHMARKING RESULTS

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

DEPLOY LOAD-GENERATOR SERVICE CREATED IN TASK 3.

MobaTextEditor

File Edit Search View Format Encoding Syntax Special Tools

load-generator.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: load-generator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: load-generator
  template:
    metadata:
      labels:
        app: load-generator
    spec:
      containers:
      - name: load-generator
        image: localhost:32000/load-generator:latest
        env:
        - name: target
          value: "http://10.152.183.110:8080/primecheck"
        - name: frequency
          value: "10.0"
        command: ["python", "load_generator.py"]
```

We deploy our load-generator by setting environment variables values that would be used by our load-generator code to continuously send requests to our JavaBenchmarkApp.

```
student@edge:~$ kubectl logs -f load-generator-65dccc8955-hqwxg
Request successful. Response Time: 1.64 seconds
Average Response Time: 1.64 seconds
Total Failures: 0
Total Requests: 1

Request successful. Response Time: 1.70 seconds
Average Response Time: 1.67 seconds
Total Failures: 0
Total Requests: 2

Request successful. Response Time: 1.88 seconds
Average Response Time: 1.74 seconds
Total Failures: 0
Total Requests: 3

Request successful. Response Time: 2.17 seconds
Average Response Time: 1.85 seconds
Total Failures: 0
Total Requests: 4

Request successful. Response Time: 2.74 seconds
Average Response Time: 2.03 seconds
Total Failures: 0
Total Requests: 5

Request successful. Response Time: 2.72 seconds
Average Response Time: 2.14 seconds
Total Failures: 0
Total Requests: 6

Request successful. Response Time: 2.64 seconds
Average Response Time: 2.21 seconds
Total Failures: 0
Total Requests: 7

Request successful. Response Time: 1.54 seconds
Average Response Time: 2.13 seconds
Total Failures: 0
Total Requests: 8
```

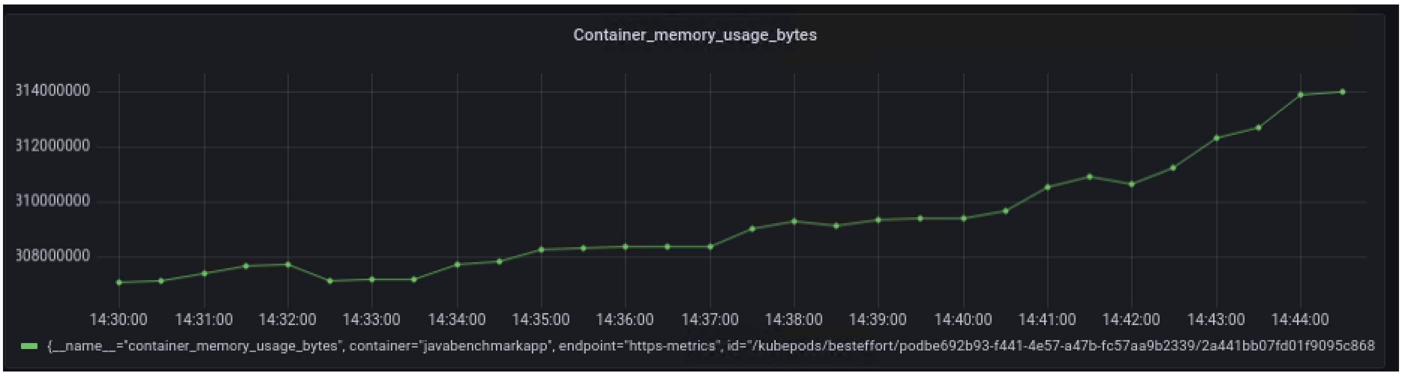
The above figure represents logs obtained after deploying the service.

2. 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.

We created a new dashboard and queried the required metrics as shown below.

3. SCREENSHOT THE TWO PANELS

The above image shows the metrics obtained for Container_cpu_usage_Seconds_total, which shows a linear graph that depicts the CPU usage to be increasing with time. The graph shows the usage of CPU once the webapp starts receiving requests, the cpu decline only when the requests fails and it encounters an issue with the service but in our case there is no failure in request, so the graph shows a linear graph.



The above graph shows the Container_memory_usage_bytes metrics obtained which shows non-linear trend of memory usage with time. The Memory usage depends upon the nature of webapp. When the Javabenchmarkapp service is down, the graph shows a negative curve and vice versa when the Javabenchmarkapp is up and running.

CONCLUSION:

Through the above coursework, we were able to obtain practical experience with the deployment, management, and monitoring of Kubernetes applications. We also learned how to generate load for benchmarking, which produces insightful data that can be used to optimize application scaling and enhance overall performance. Using Grafana, we built a dashboard with panels to show the CPU and memory utilization of containers. We discovered that when the load on our webapp increased, the graph displayed a linear curve.

REFERENCES:

1. https://stackoverflow.com/ : For various doubts related to implementation and errors.
2. https://kubernetes.io/docs/ : Kubernetes documentation