Kubernetes-Based Application Hosting and Monitoring System

Introduction

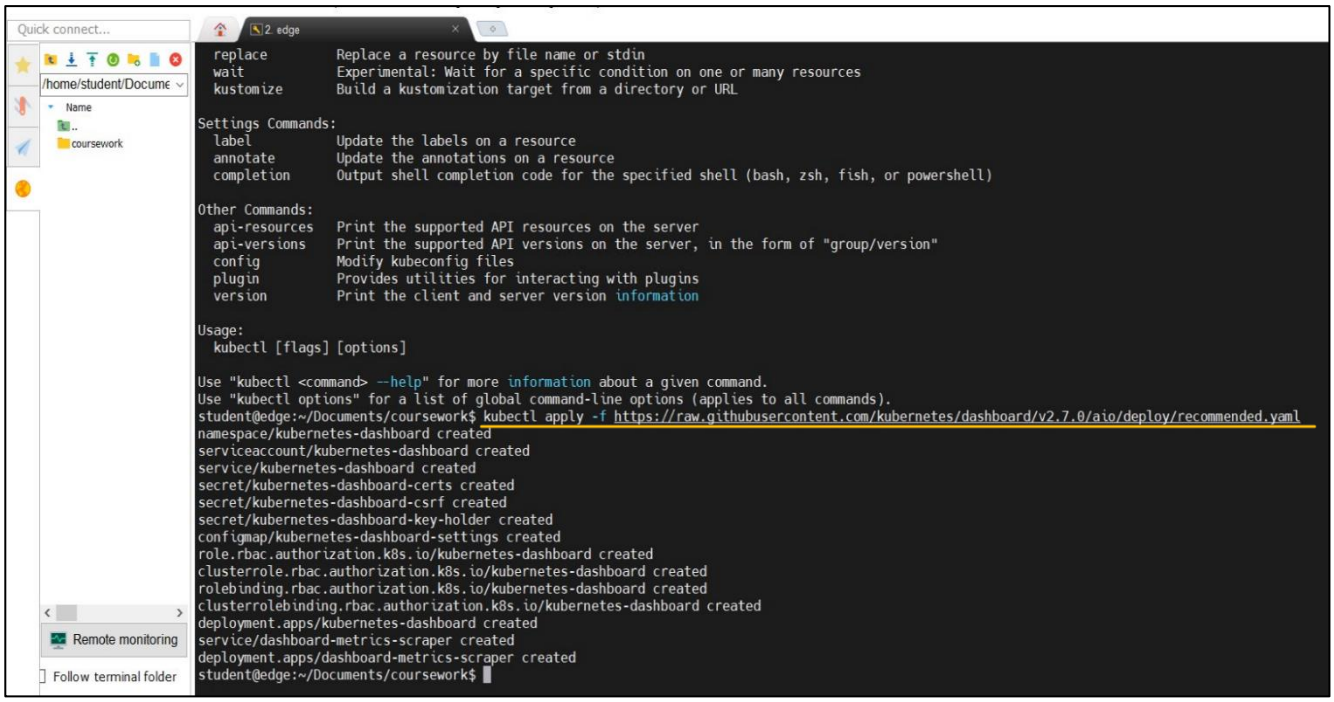
This project is based on understanding basic concepts of Kubernetes along with deploying a Kubernetes based application hosting environment. Understanding docker and learn how to build, push and pull image from docker hub. To understand about monitoring stack of Kubernetes such as Grafana, Prometheus, Metrics server and to deploy and use them.

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

1.1 Deploying Kubernetes Dashboard on edge virtual machine

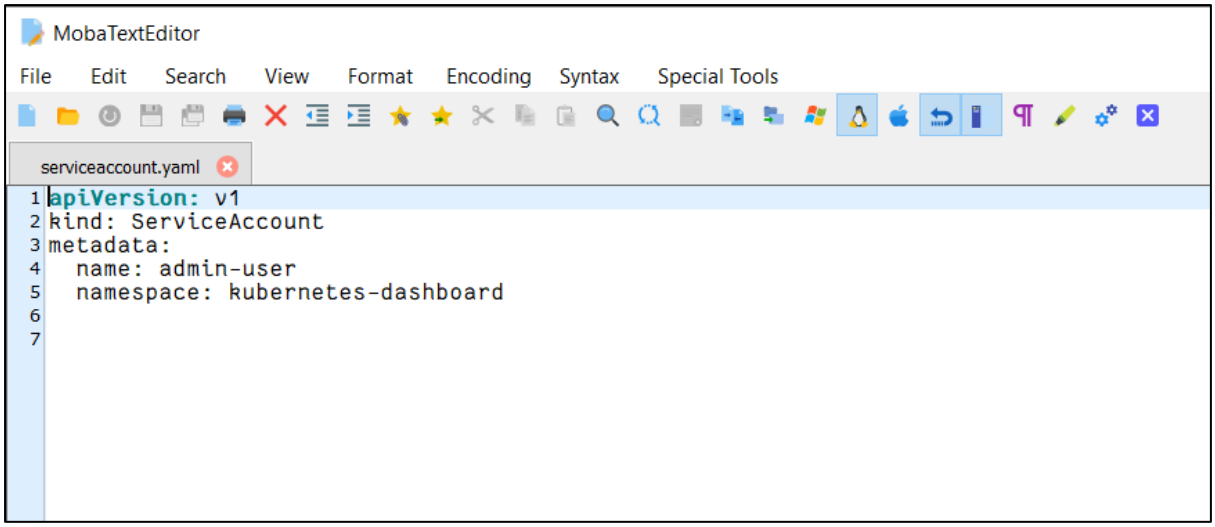
Kubernetes is basically an open-source platform which can be used to deploy containerized applications. In order to deploy Kubernetes dashboard, we run the below command into CLI of edge virtual machine.

"*kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml*"



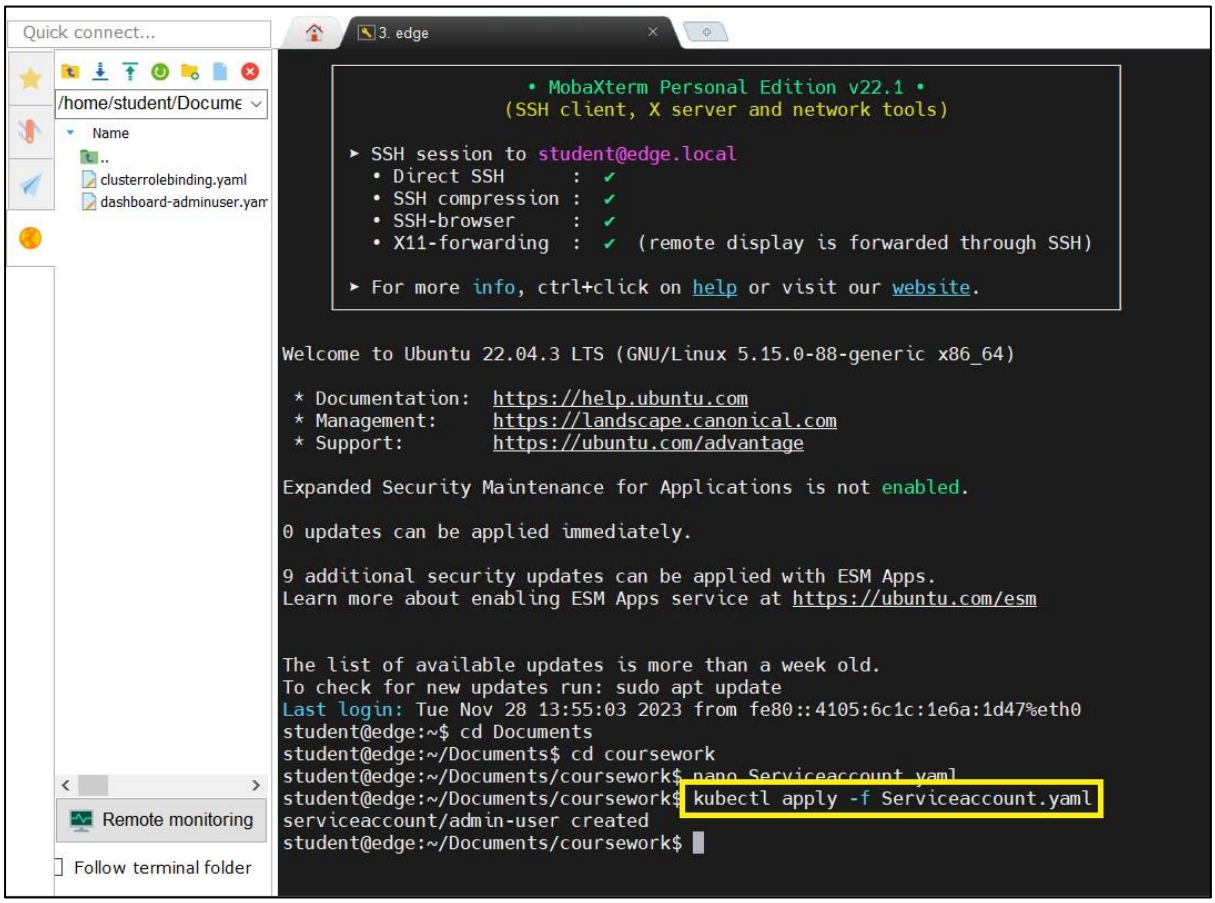
Now in order to login/access the dashboard we need to create a user using the service account mechanism of Kubernetes. Service account allows processes running in the pods to authenticate and interact with Kubernetes API.

To create a service account in Kubernetes we can specify yaml manifest file. In this file we first specify the version of Kubernetes API followed by defining the kind of Kubernetes resource being created, which is a service account. Next, we specify metadata such as name of service account which we give as admin-user and namespace in which service account will be created i.e kubernetes-dashboard.



To execute this, run the below command in CLI

"*kubectl apply -f serviceaccount.yaml*"



Next, we create a cluster role binding where we bind the cluster-admin role to the admin user service account. For this we create the yaml manifest file with the name "clusterbinding.yaml" and apply this manifest file by running the following command,

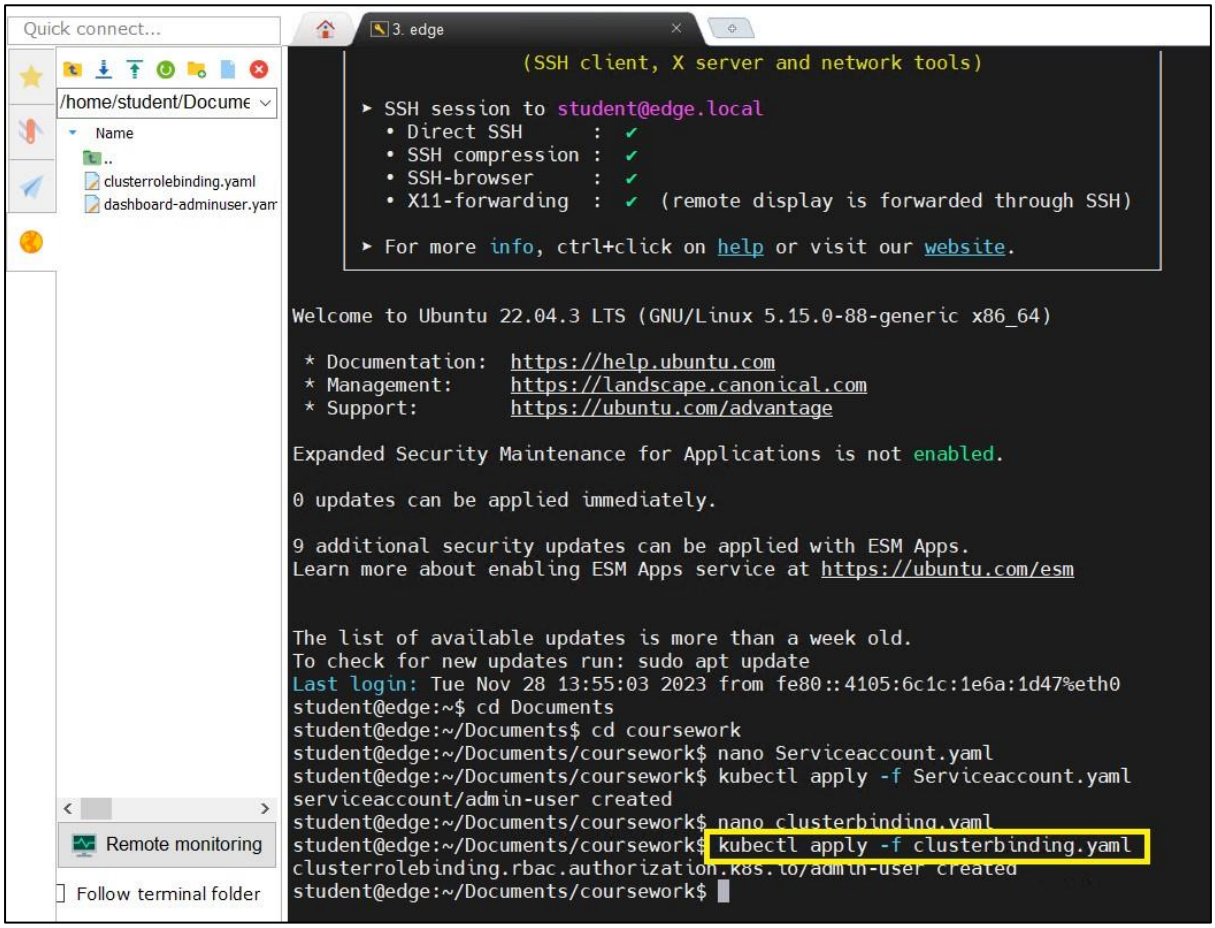
"*kubectl apply -f* clusterbinding*.yaml*"

MobaTextEditor

File Edit Search View Format Encoding Syntax Special Tools

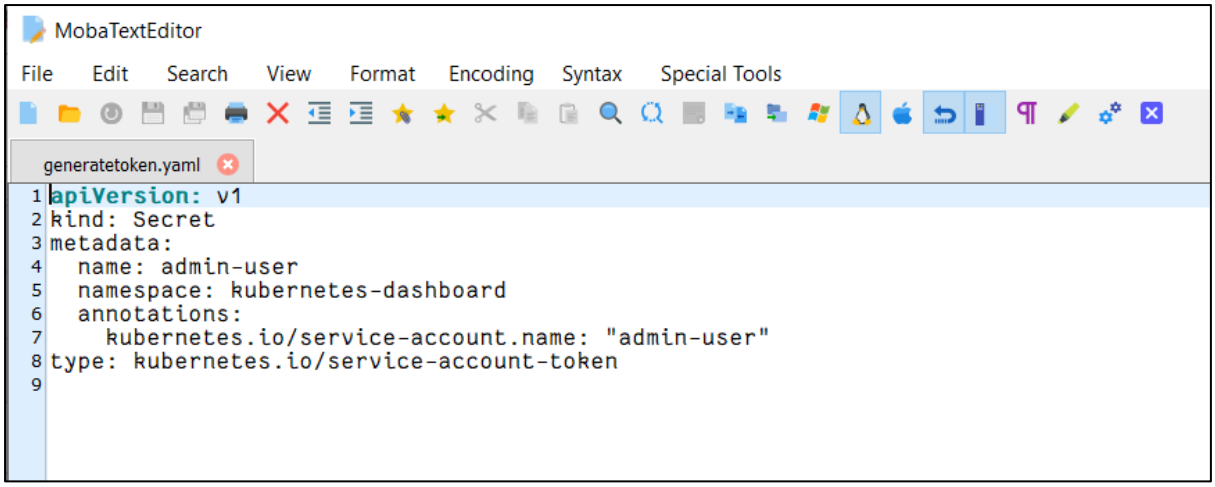
clusterbinding.yaml

```
1  apiVersion: rbac.authorization.k8s.io/v1
2  kind: ClusterRoleBinding
3  metadata:
4    name: admin-user
5  roleRef:
6    apiGroup: rbac.authorization.k8s.io
7    kind: ClusterRole
8    name: cluster-admin
9  subjects:
10 - kind: ServiceAccount
11   name: admin-user
12   namespace: kubernetes-dashboard
13
```



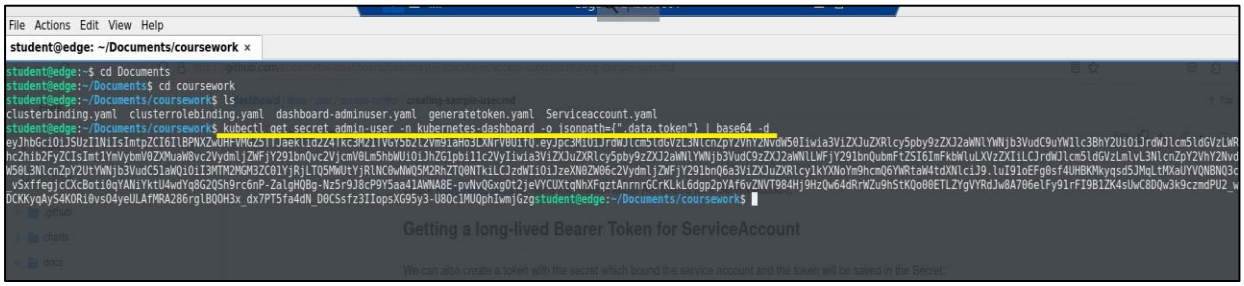
Now in order to login to the dashboard we need to generate a token. I have generated the long live bearer token which will be saved in the secret and can be used to login into the dashboard. We can also generate a temporary token to login into the dashboard however that token will keep changing every time we generate a new one. We execute the below yaml file in order to generate a token and store it in a secret. Run the following command to apply the manifest file,

"*kubectl apply -f generatetoken.yaml*"



Now run the below command to fetch that token from the secret,

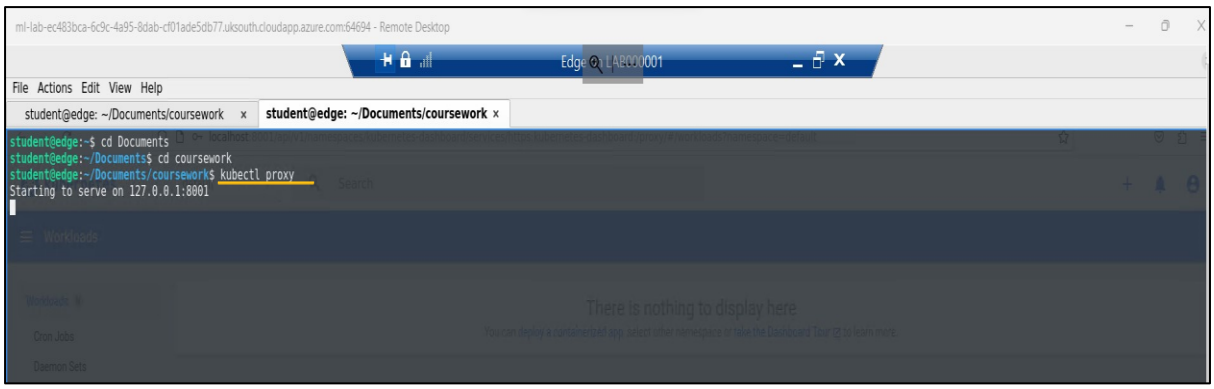
“kubectl get secret admin-user -n kubernetes-dashboard -o jsonpath={".data.token"} | base64 -d”



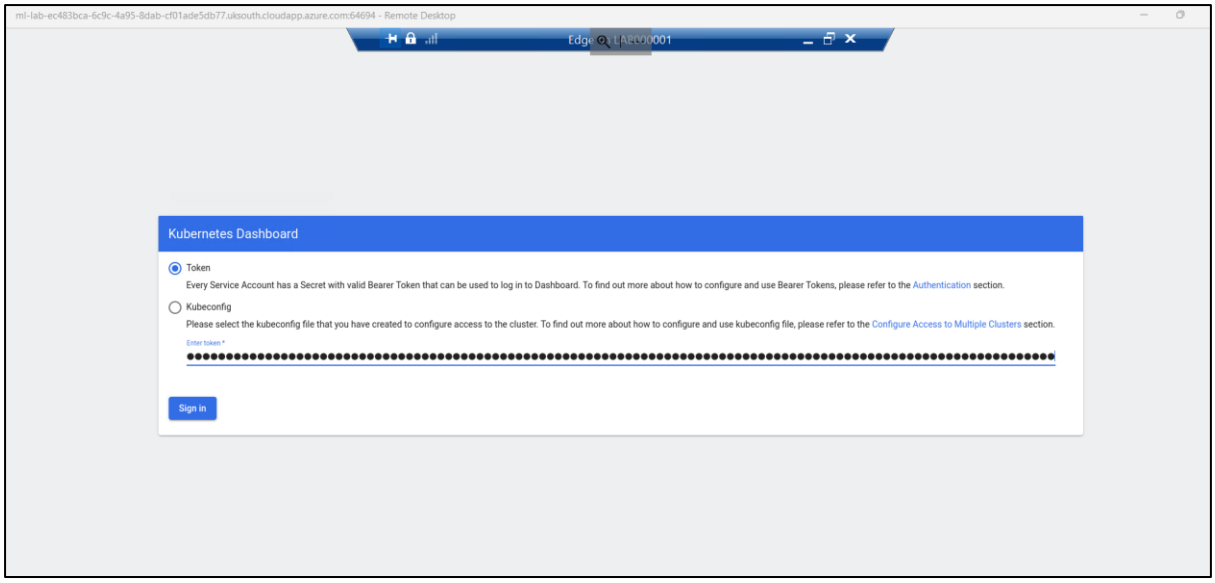
To enable access to dashboard, run the below command,

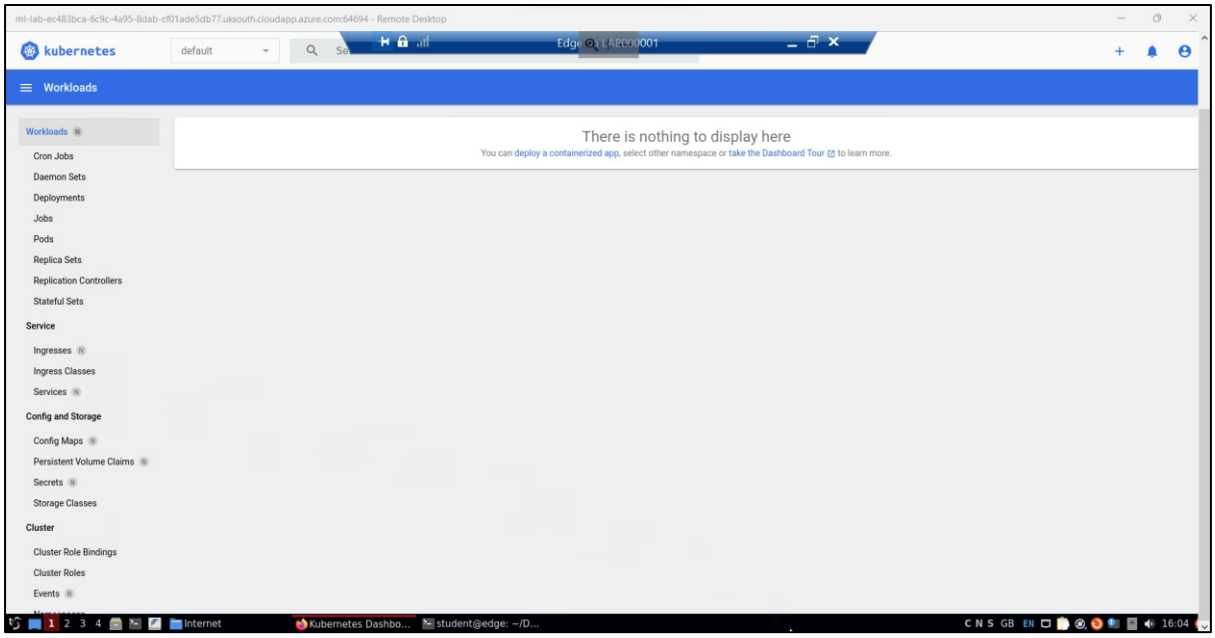
“*kubectl proxy*” – This will create a proxy server between our machine and Kubernetes API server. This proxy server allows us to access various Kubernetes service, APIs and resources exposed by Kubernetes API server.

Now the dashboard will be accessible at “http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/”.



We can now visit above URL and login to the Kubernetes dashboard using the token that was generated.





1.2 Deploying the instance of provided docker image for javabenchmarkapp

In order to deploy an image using CLI we need to use Kubernetes manifest file in yaml format which will contain the configurations of image deployment.
Firstly, we specify the version of Kubernetes API being used followed by the type of Kubernetes resource being created, which is “Deployment”. Next, we provide name for our deployment which is given as javabenchmarkapp-deployment.
Further in spec, we define 1 replica to run. In selector we define which pods this deployment manages.
Template defines the pod template that Kubernetes will use to create new pods. First part is metadata, that specifies the label for pods created using this template followed by pods specification such as containers within the pod. It contains name of the container, we give as “javabenchmarkapp-container” and docker image which is “nclcloudcomputing/javabenchmarkapp” and in port we specify the port number which application inside the container is using.

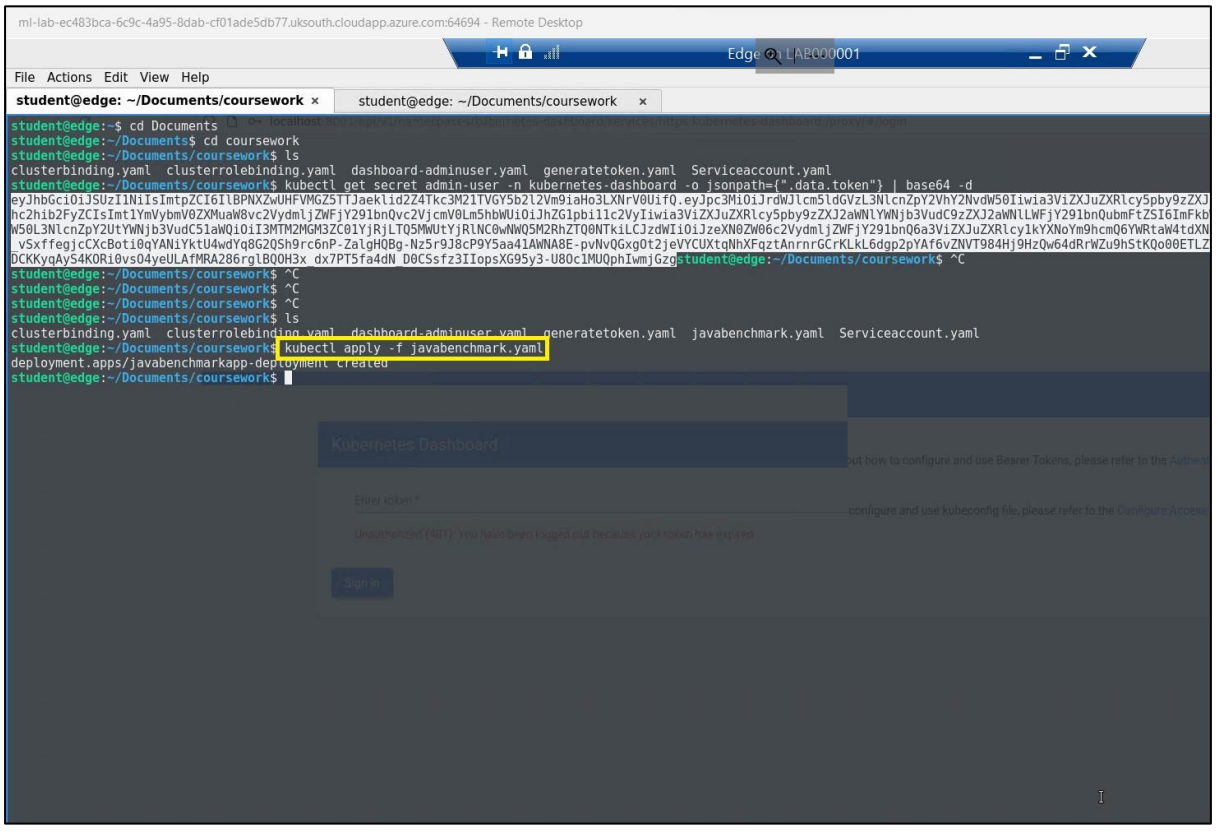
MobaTextEditor

File Edit Search View Format Encoding Syntax Special Tools

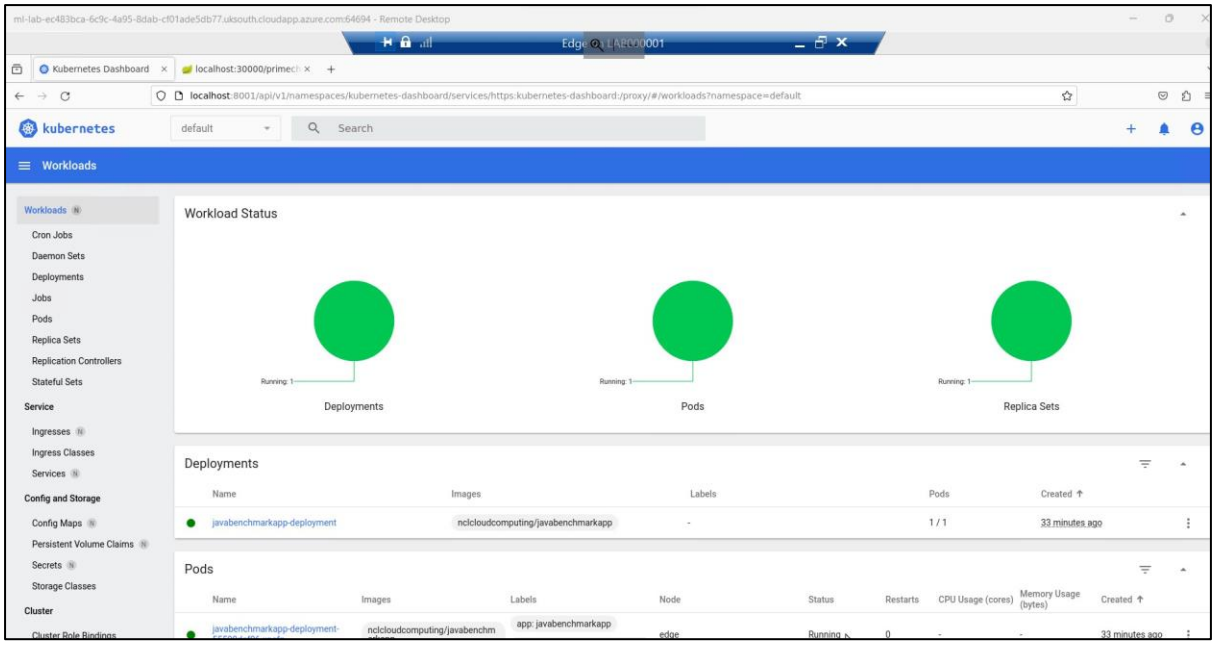
javabenchmark.yaml

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

Now we do the deployment by applying the above manifest file.



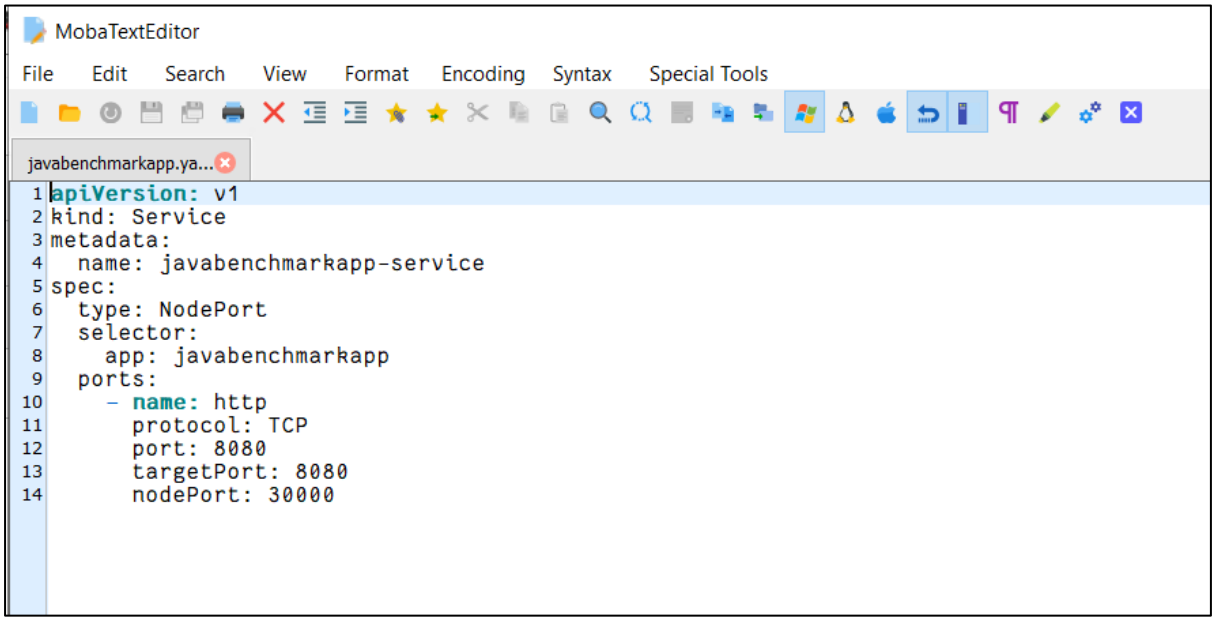
We can check our deployment on Kubernetes dashboard and ensure that pods are up and running.

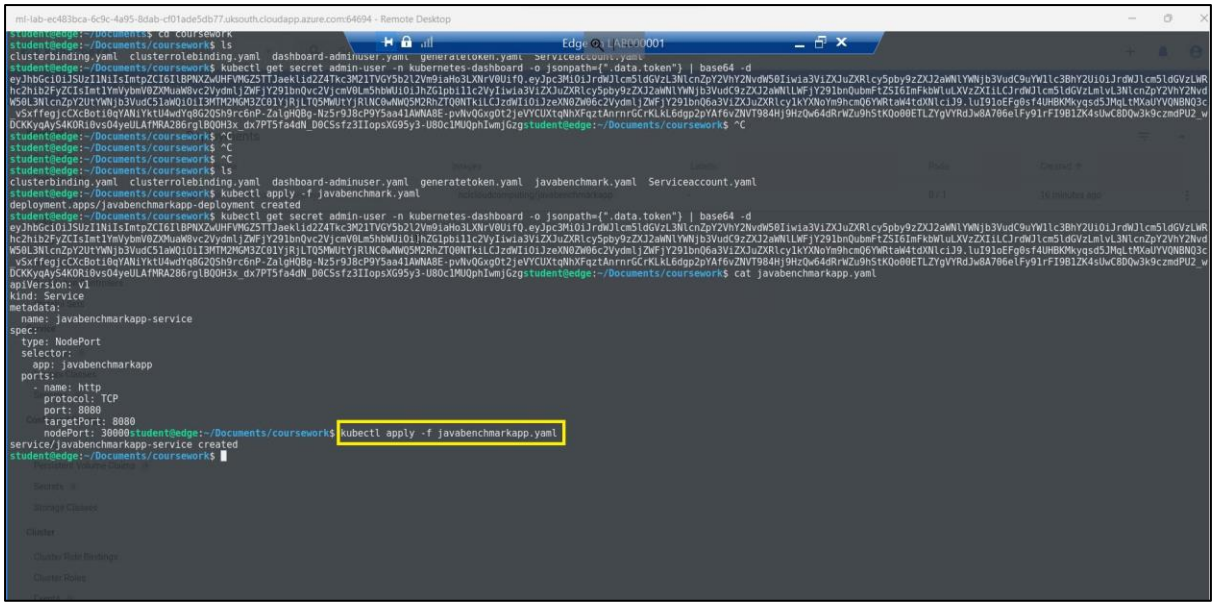


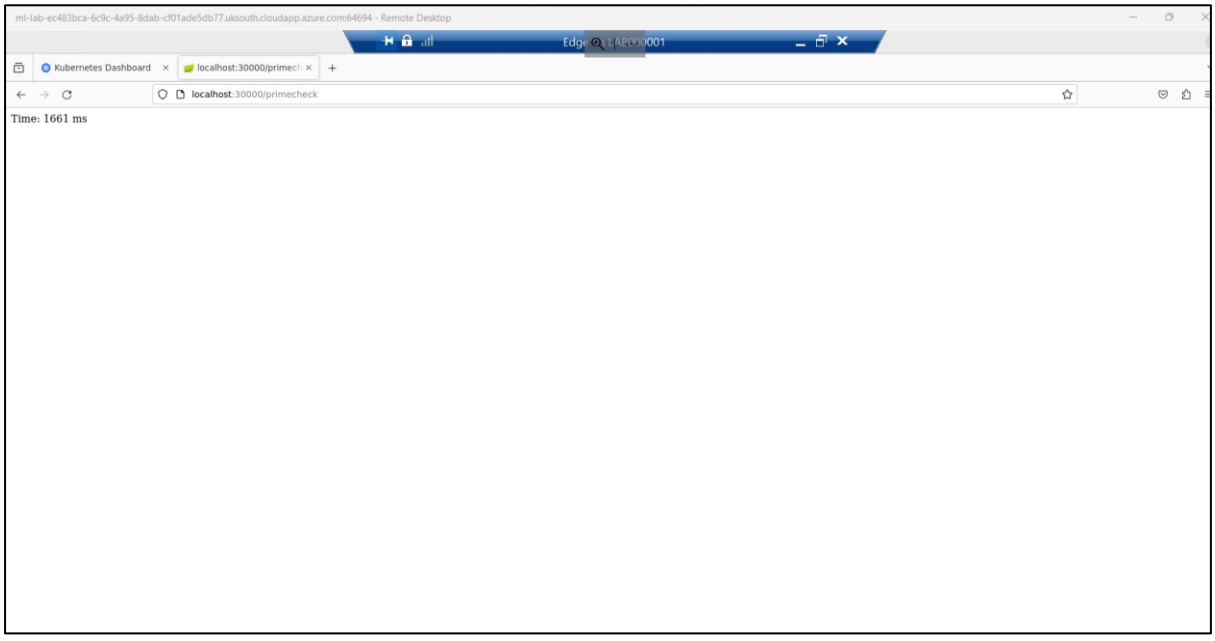
1.3 Deploying a NodePort service

We can deploy a NodePort service by specifying the configurations in a yaml file. The kind of Kubernetes resource we are deploying will be “Service”. Then in metadata we specify the name of service as “javabenchmarkapp-service”. In spec, we set the type of service as NodePort and in ports we define name as http, protocol used by the port as TCP, port 8080 exposed on the service and target port to which service will forward the traffic. Then in nodePort we provide the specific node port 30000 to use.

Basically, what a NodePort does is, it publicly exposes a service on a fixed port number and lets us access the service from outside of our cluster.




ml-lab-ec483bca-6c9c-4a95-8dab-cf01ade5db77.uksouth.cloudapp.azure.com:64694 - Remote Desktop
clusterbinding.yaml clusterrolebinding.yaml dashboard-adminuser.yaml generatetoken.yaml Serviceaccount.yaml
student@edge:~/Documents/coursework$ kubectl get secret admin-user -n kubernetes-dashboard -o jsonpath={".data.token"} | base64 -d
student@edge:~/Documents/coursework$ ^C
student@edge:~/Documents/coursework$ ^C
student@edge:~/Documents/coursework$ ^C
student@edge:~/Documents/coursework$ ls
clusterbinding.yaml clusterrolebinding.yaml dashboard-adminuser.yaml generatetoken.yaml javabenchmark.yaml Serviceaccount.yaml
student@edge:~/Documents/coursework$ kubectl apply -f javabenchmark.yaml
deployment.apps/javabenchmarkapp-deployment created
student@edge:~/Documents/coursework$ kubectl get secret admin-user -n kubernetes-dashboard -o jsonpath={".data.token"} | base64 -d
student@edge:~/Documents/coursework$ cat javabenchmarkapp.yaml
apiVersion: v1
kind: Service
metadata:
name: javabenchmarkapp-service
spec:
type: NodePort
selector:
app: javabenchmarkapp
ports:
- name: http
protocol: TCP
port: 8080
targetPort: 8080
nodePort: 30000student@edge:~/Documents/coursework$ kubectl apply -f javabenchmarkapp.yaml
service/javabenchmarkapp-service created
student@edge:~/Documents/coursework$


ml-lab-ec483bca-6c9c-4a95-8dab-cf01ade5db77.uksouth.cloudapp.azure.com:64694 - Remote Desktop
Kubernetes Dashboard
localhost:30000/primech
localhost:30000/primecheck
Time: 1661 ms

Task 2: Deploy the monitoring stack of Kubernetes

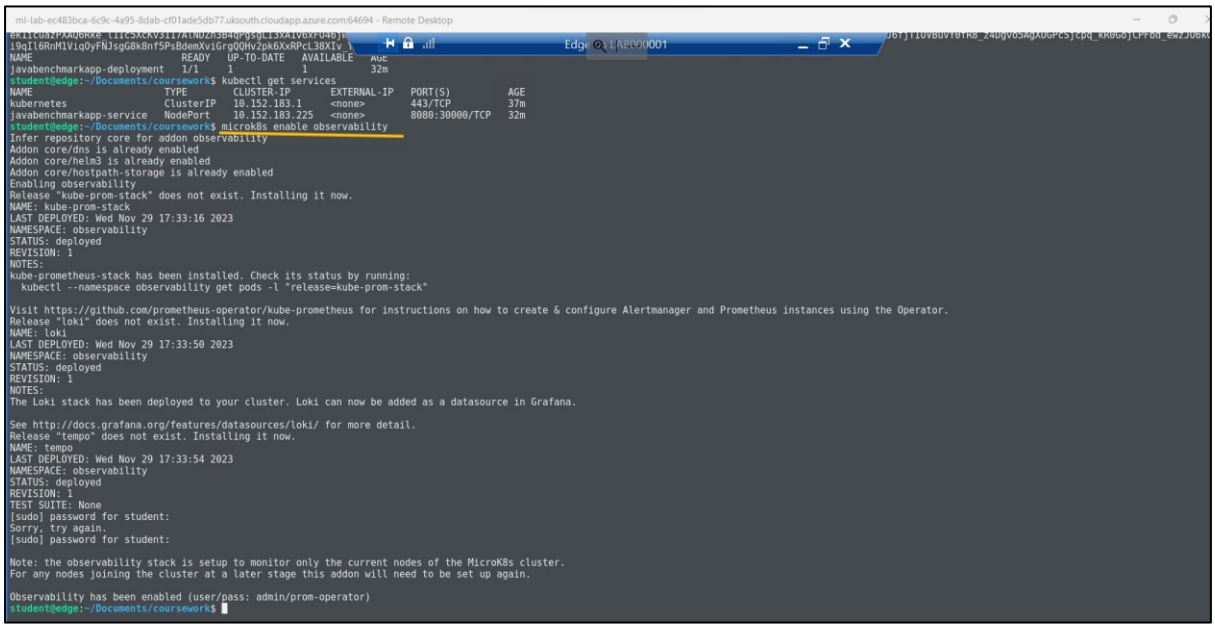
2.1 Enable observability service from microk8s addons

In order to install observability stack, we run the below command into our CLI.

"*microk8s enable observability*"

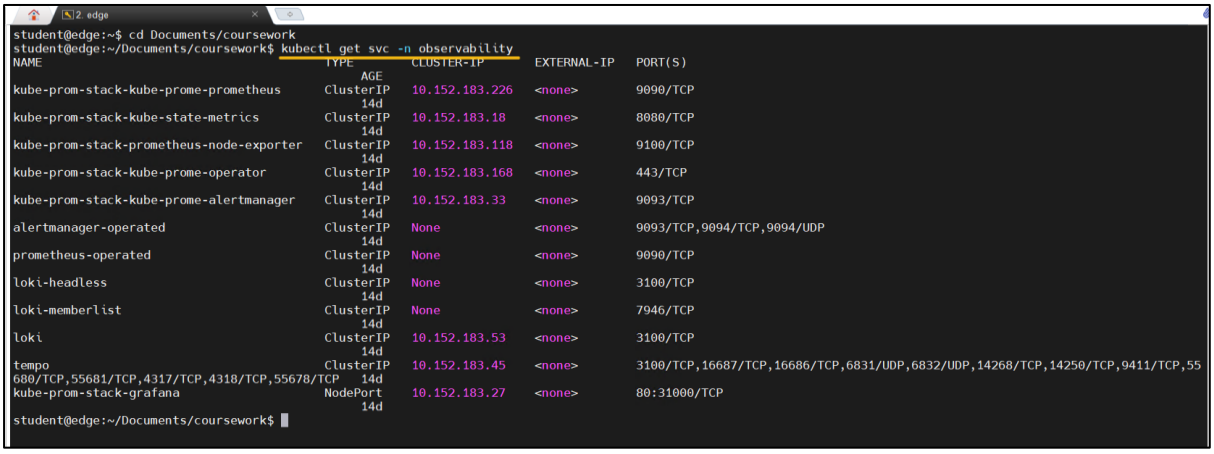
It prints out the status of different add-ons that it is enabling and installs the missing add-ons.

Running this command basically enables observability-related add-ons and functionalities. It provides us with the set of monitoring, logging, and observability tools which we can use to gain insights into the performance of Kubernetes cluster and its running applications.

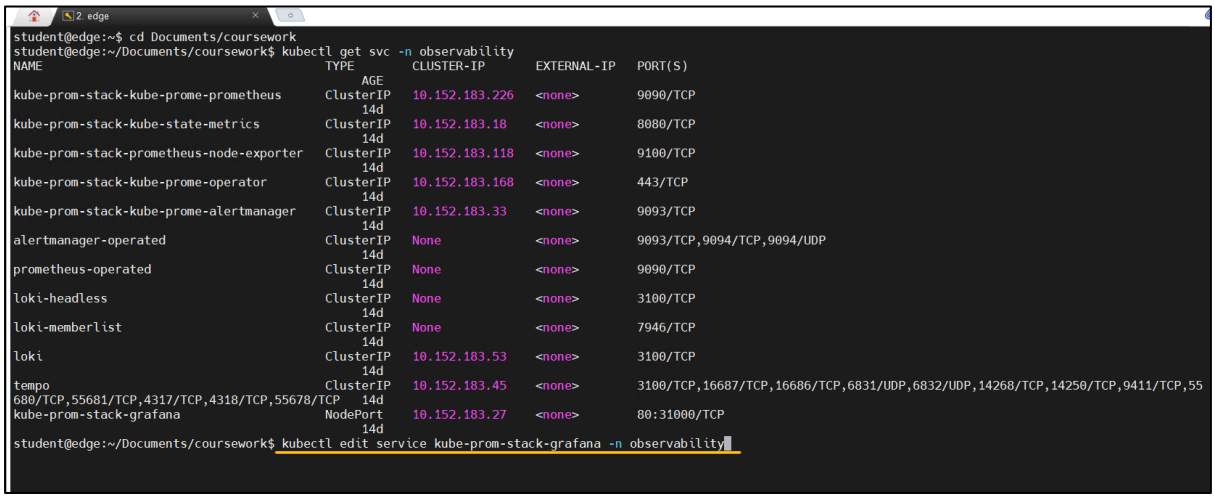


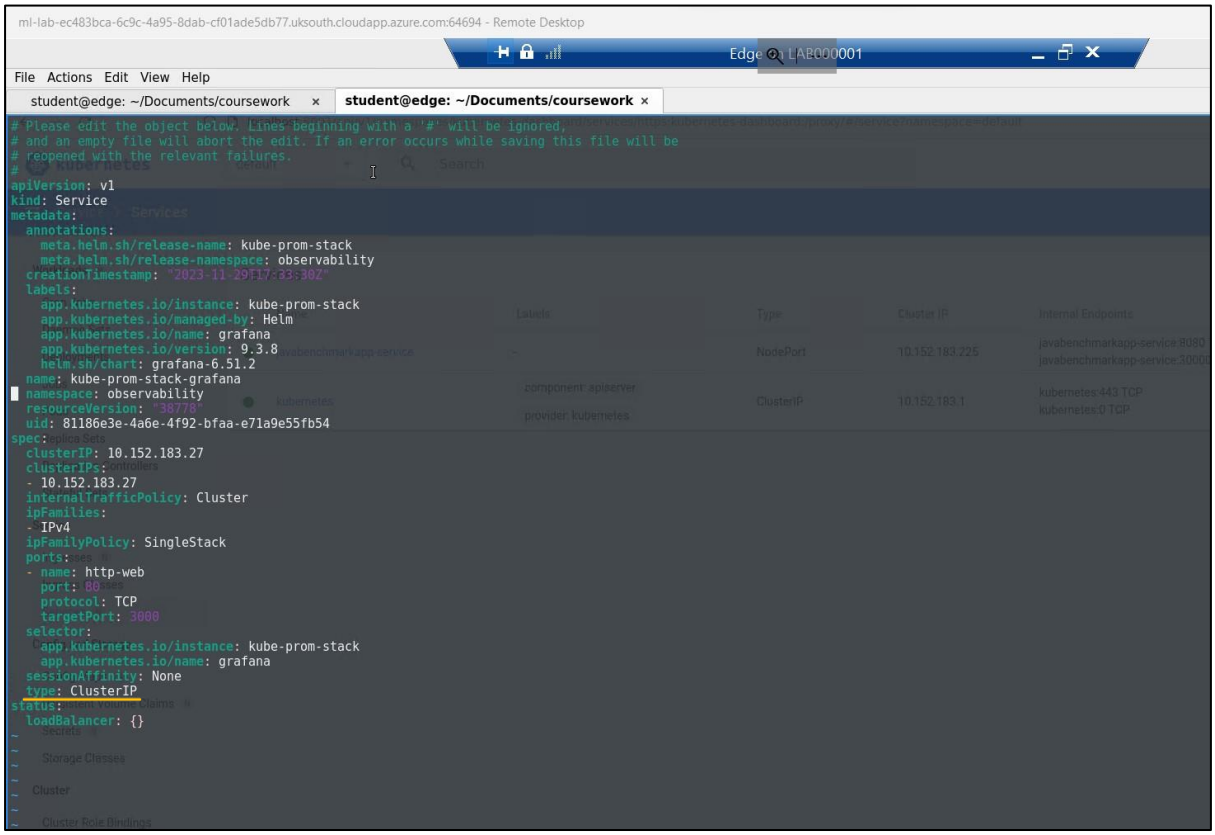
2.2 Editing the Grafana service

We will first check if the Grafana service is present by listing all the services within our namespace which is "observability". For this we run the below command into CLI,
"*kubectl get svc -n observability"*

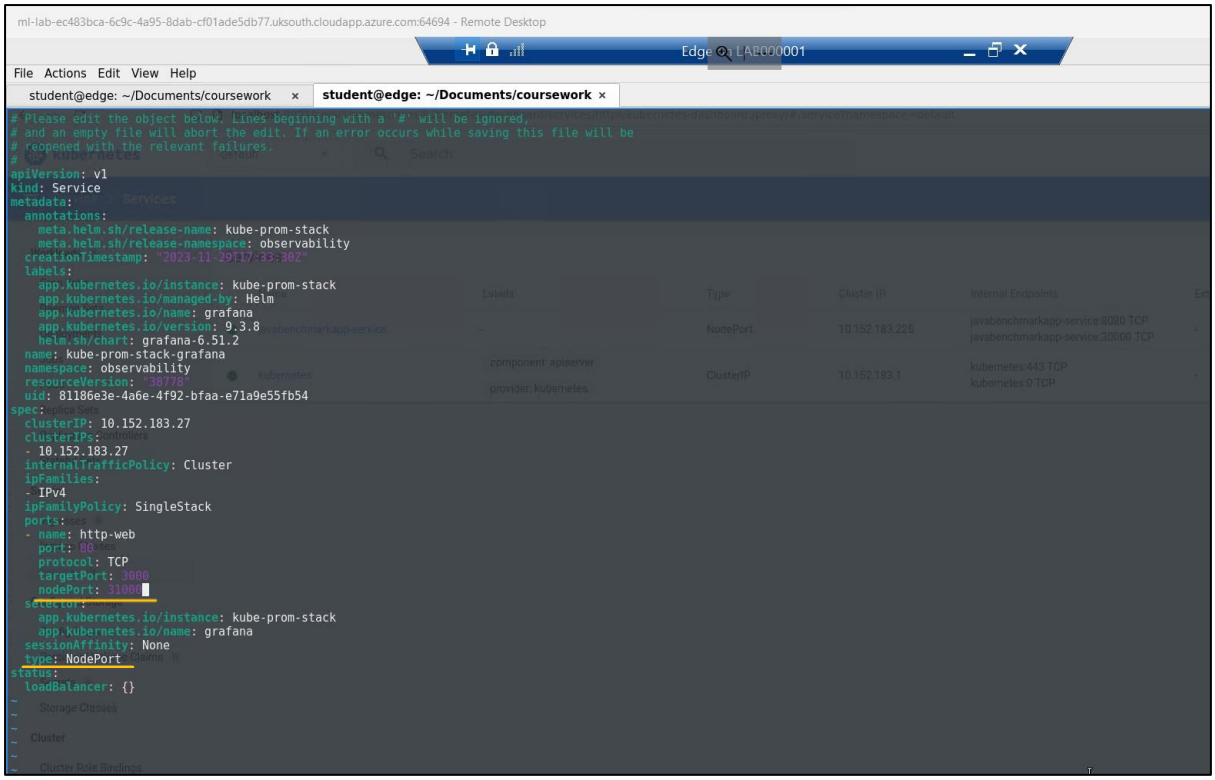


Now edit the Grafana service by executing below command,
"*kubectl edit service kube-prom-stack-grafana -n observability*"



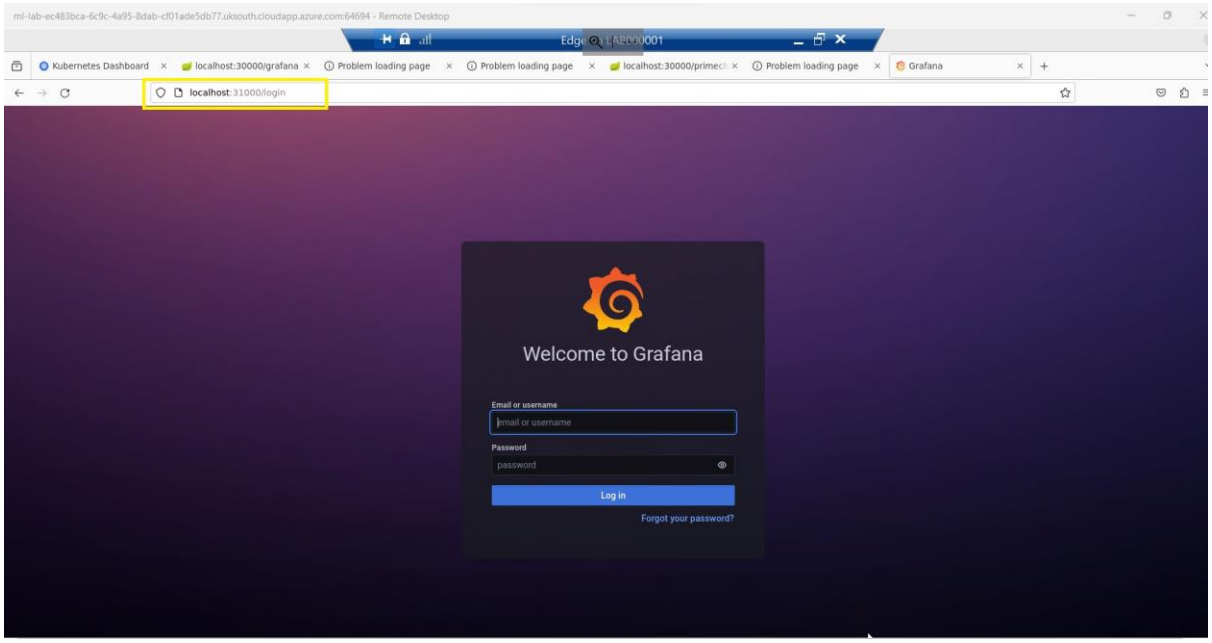


Here we change the type from ClusterIP to NodePort and then we define the port 31000 for nodePort.

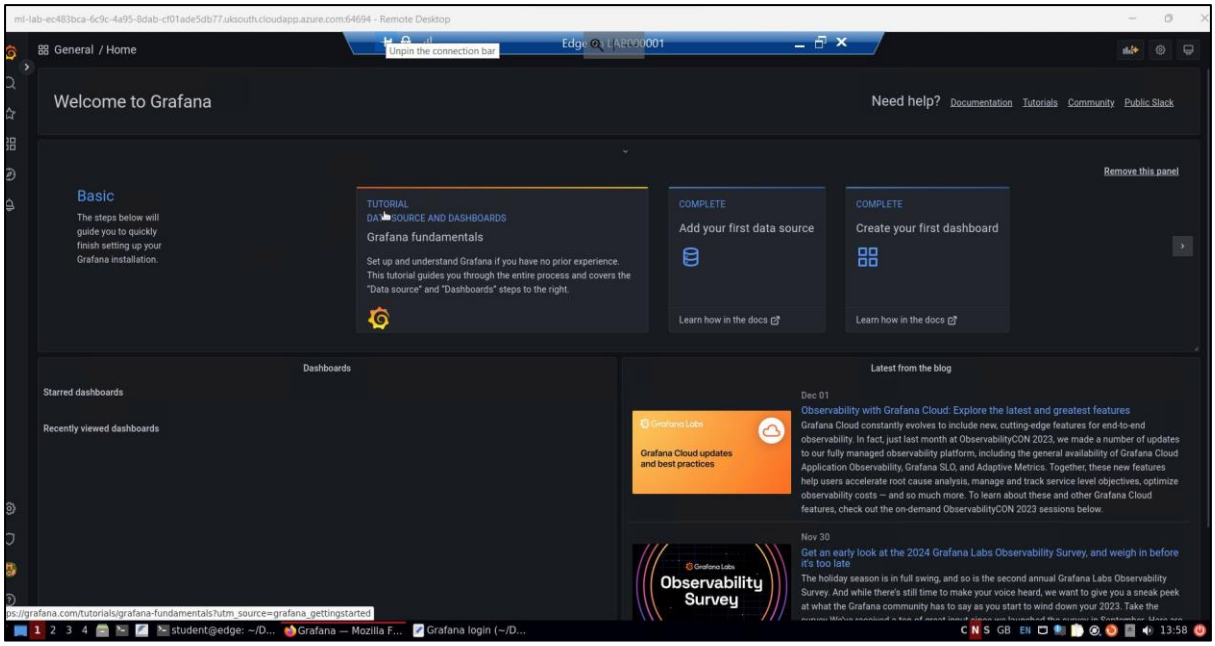


2.3 Log in to Grafana dashboard

After editing the Grafana service, now we can access it from localhost and port number 31000 that we have specified while editing the service.



Login into the Grafana dashboard by using the credentials provided in the text document present on edge virtual machine.



Task 3: Creating load generator and building its docker image

3.1 Writing a load generator

loadgenerator.py

```python
import requests
import time
import os


class LoadGenerator:
    def __init__(self, target, frequency):
        self.target = target
        self.frequency = frequency
        self.noofrequests = 0
        self.nooffailures = 0
        self.totalresponsetime = 0


    def print_metrics(self):
        avgresponsetime = self.totalresponsetime / self.noofrequests if self.noofrequests > 0 else 0
        print(f"Metrics after {self.noofrequests} requests:")
        print(f"Average Response Time: {avgresponsetime:.4f} seconds")
        print(f"Accumulated Number of Failures: {self.nooffailures}\n")

    def generate_load(self):
        while True:
            starttime = time.time()
            try:
                response = requests.get(self.target, timeout=10)
                if response.status_code != 200:
                    self.nooffailures = self.nooffailures + 1
            except requests.exceptions.Timeout:
                self.nooffailures = self.nooffailures + 1

            endtime = time.time()
            self.noofrequests = self.noofrequests + 1
            self.totalresponsetime += endtime - starttime

            time.sleep(1 / self.frequency)

            if self.noofrequests % 10 == 0:
                self.print_metrics()
```

```
41
42
43 if __name__ == "__main__":
44     target = os.getenv("TARGET_ADDRESS", "http://10.152.183.225:8080/primecheck")
45     frequency = float(os.getenv("REQUESTS_PER_SECOND", 10))
46
47     load_generator = LoadGenerator(target, frequency)
48     load_generator.generate_load()
```

The above code for load generation is written in python programming language. Initially we import all the required libraries such as requests, time and os. The code consists of a main method and LoadGenerator class. In the main method we specify two variables which are namely target and frequency. In "target" we will basically store the URL on which we will be generating the load and set the frequency value to 10 requests per second. Next, we are creating the object of LoadGenerator class by passing target and frequency and then calling generate_load() function of LoadGenerator class.

In generate_load function we have used a never ending while loop which will keep sending requests to our benchmark app. If response for the sent request is not successful i.e status code is not 200 then it will be counted as a failure. We have also specified a timeout of 10 seconds which means if a particular request is taking more than 10 seconds then it will timeout and be treated as a failure. We collect total number of failures and average response time. Next, we call the print_metrics function to output the average response time and total number of accumulated failures.

3.2 Building the image and pushing it to local registry

We have to first write a docker file to build the image of our load generator python code.

Dockerfile

```
1 FROM python:3.9
2 WORKDIR /app
3 COPY loadgenerator.py .
4 Run pip install requests
5 CMD ["python", "loadgenerator.py"]
```

First, we specify the base image of python 3.9 and then set the working directory to /app. Now we copy our loadgenerator python code to the working directory and specify the command that needs to execute after image starts running.

Now we will build the image and name it as "load-generator" by using the below command,

"*docker build -t load-generator .*"

```
student@edge:~/Documents/courseworks$ docker build -t load-generator .
[+] Building 13.2s (9/9) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 153B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/python:3.9
 => [1/4] FROM docker.io/library/python:3.9@sha256:b2e47a7eca3178e4ce6c095d3a2d1cd05bfa616efe7f2047c95fffe159e00166
 => [internal] load build context
 => => transferring context: 1.72kB
 => CACHED [2/4] WORKDIR /app
 => [3/4] COPY loadgenerator.py .
 => [4/4] RUN pip install requests
 => exporting to image
 => => exporting layers
 => => writing image sha256:de6b8740f3f6127bff8efd8251e815ab5ba59a9c4819bd00feb57adc3ca79eaa
 => => naming to docker.io/library/load-generator
```

Now we need to push this image to the local registry at port 32000. We will pull the registry image by executing the below command.

"*docker run -d -p 32000:5000 --restart=always --name registry registry:2*"

```
student@edge:~/Documents/coursework$ docker run -d -p 32000:5000 --restart=always --name registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry
c926b61bad3b: Pull complete
5501dced60f8: Pull complete
e875fe5e6b9c: Pull complete
21f4bf2f86f9: Pull complete
98513cca25bb: Pull complete
Digest: sha256:0a182cb82c93939407967d6d71d6caf11dcef0e5689c6afe2d60518e3b34ab86
Status: Downloaded newer image for registry:2
4d85846a7fb3b16fb48eeff5b0fc8e41d99a244f95df4b91a3fa7ca37dbe3aad
student@edge:~/Documents/coursework$ docker tag load-generator localhost:32000/load-generator
```

Now we will tag the image and push it to the local registry at port 32000

```
student@edge:~/Documents/coursework$ docker build -t load-generator .
[+] Building 13.2s (9/9) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 153B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/python:3.9
 => [1/4] FROM docker.io/library/python:3.9@sha256:b2e47a7eca3178e4ce6c095d3a2d1cd05bfa616efe7f2047c95fffe159e00166
 => [internal] load build context
 => => transferring context: 1.72kB
 => CACHED [2/4] WORKDIR /app
 => [3/4] COPY loadgenerator.py
 => [4/4] RUN pip install requests
 => exporting to image
 => => exporting layers
 => => writing image sha256:de6b8740f3f6127bff8efd8251e815ab5ba59a9c4819bd00feb57adc3ca79eaa
 => => naming to docker.io/library/load-generator
student@edge:~/Documents/coursework$ docker tag load-generator localhost:32000/load-generator
student@edge:~/Documents/coursework$ docker push localhost:32000/load-generator
Using default tag: latest
The push refers to repository [localhost:32000/load-generator]
90f00dde2ecc: Pushed
08ccd560c8d8: Pushed
81eaa271543c: Layer already exists
d5ad3ac69862: Layer already exists
4929be171cab: Layer already exists
f021e1878a27: Layer already exists
a04a14a911a5: Layer already exists
80bd043d4663: Layer already exists
30f5cd833236: Layer already exists
7c32e0608151: Layer already exists
7cea17427f83: Layer already exists
latest: digest: sha256:c5df4bbdd30839a5b90deba8a50887e9264894d2170a7e343d263600b15dae51 size: 2631
```

Task 4: Monitor benchmarking results

4.1 To deploy load-generator service that we created in task 3

We can deploy load-generator service using Kubernetes manifest file in yaml format. We will specify all the required configurations in this file. First, we specify Kubernetes API version and specify “Deployment” as the kind of Kubernetes resource being created. In metadata field we name our deployment as “generateload” and specify 1 replica set to be created. Next is template, which is basically used by Kubernetes while creating new pods. In this we specify name as “generateload” followed by container specification such as name, image that we built and pushed to local registry in task3, container port 8080 and environment variables which are target and frequency. We name this file as “loadgeneratorservice.yaml”.

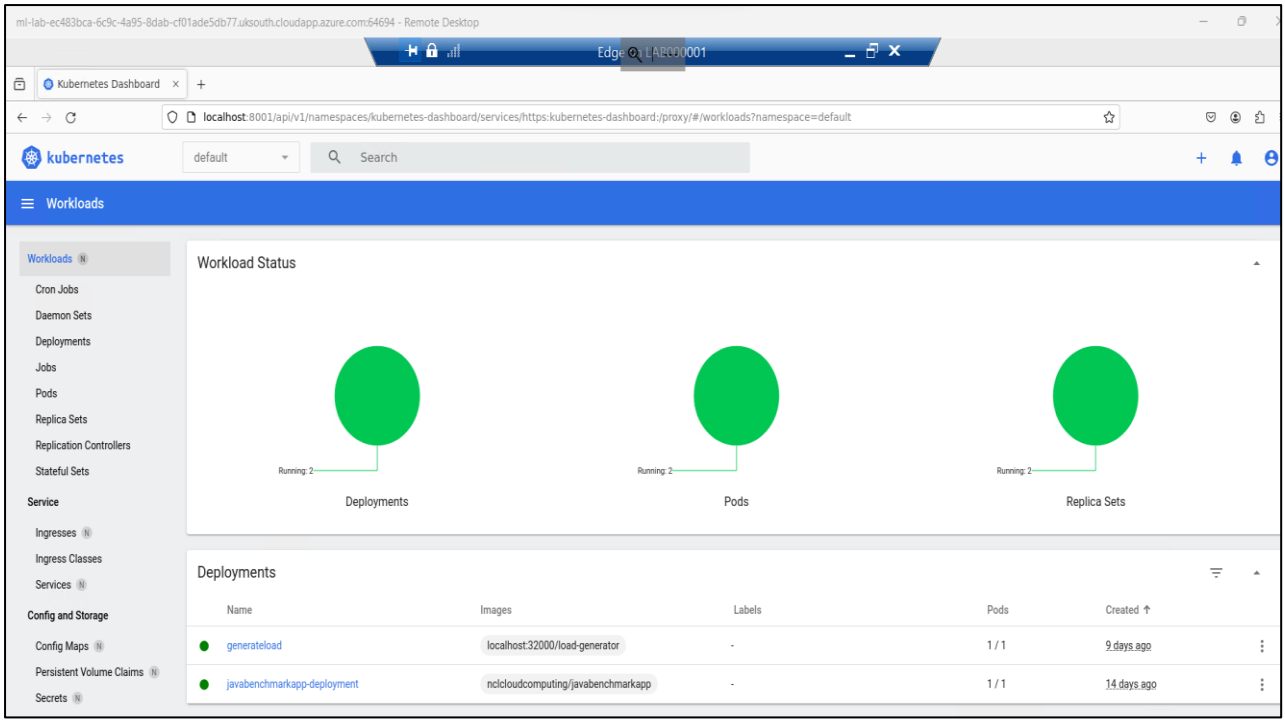
loadgeneratorservice....

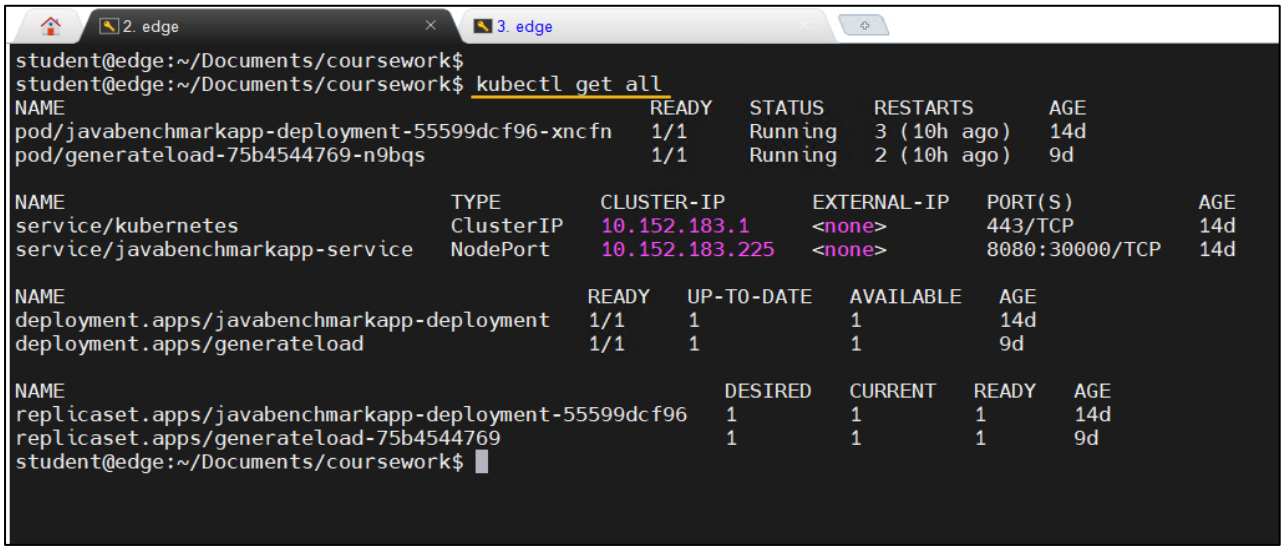
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: generateload
spec:
  replicas: 1
  selector:
    matchLabels:
      app: generateload
  template:
    metadata:
      labels:
        app: generateload
    spec:
      containers:
        - name: generateload
          image: localhost:32000/load-generator
          ports:
           - containerPort: 8080
          env:
          - name: target
            value: "http://10.152.183.225:8080/primecheck"
          - name: frequency
            value: "10.0"
          command: ["python", "loadgenerator.py"]
```

Now we can do the deployment by using following command,

“kubectl apply -f loadgeneratorservice.yaml”

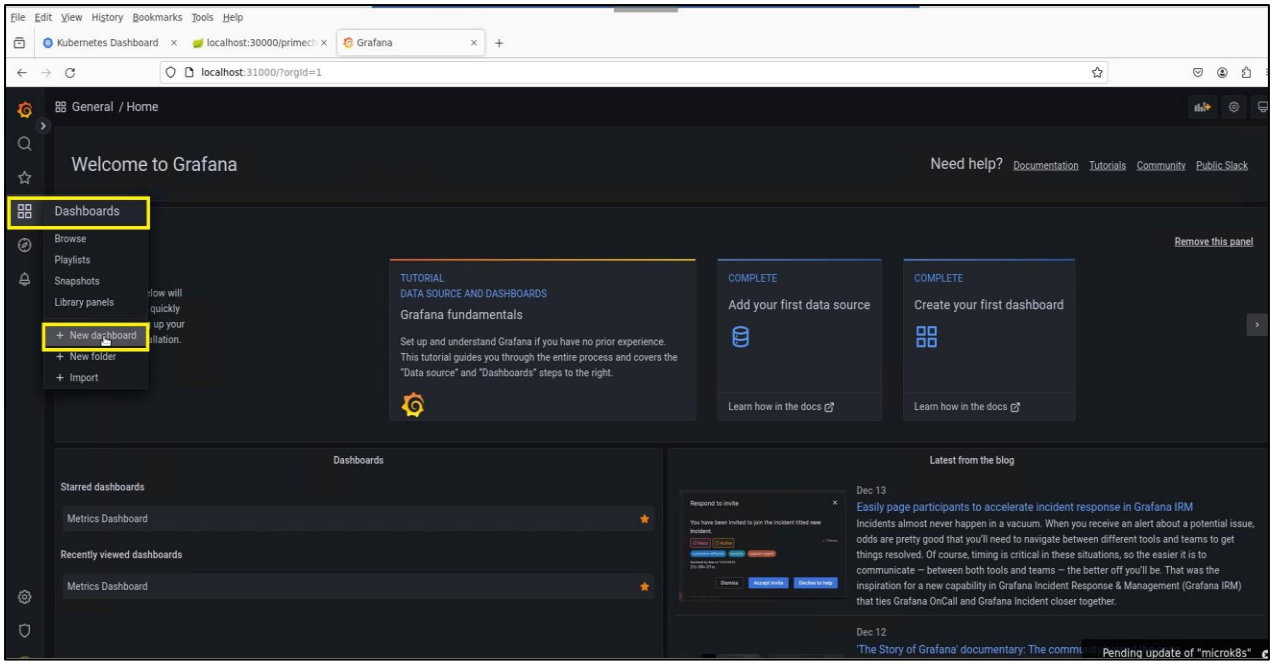
```
student@edge:~/Documents/coursework$ kubectl get all
NAME                                             READY   STATUS    RESTARTS      AGE
pod/javabenchmarkapp-deployment-55599dcf96-xncfn   1/1     Running   2 (18m ago)   4d22h

NAME                                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/kubernetes                  ClusterIP   10.152.183.1     <none>        443/TCP          4d22h
service/javabenchmarkapp-service    NodePort    10.152.183.225   <none>        8080:30000/TCP   4d22h

NAME                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/javabenchmarkapp-deployment   1/1     1            1           4d22h

NAME                                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/javabenchmarkapp-deployment-55599dcf96   1         1         1       4d22h
student@edge:~/Documents/coursework$ kubectl apply -f loadgeneratorservice.yaml
deployment.apps/generateload created
```



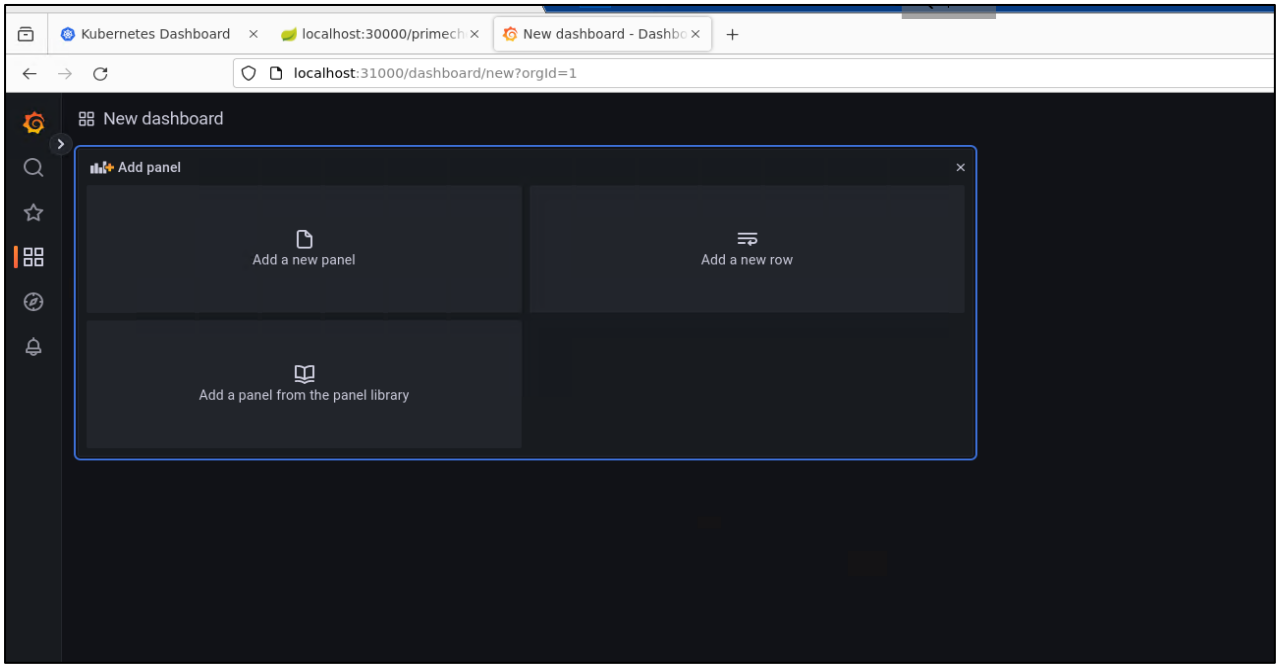


4.2 Creating dashboard and adding panels in Grafana

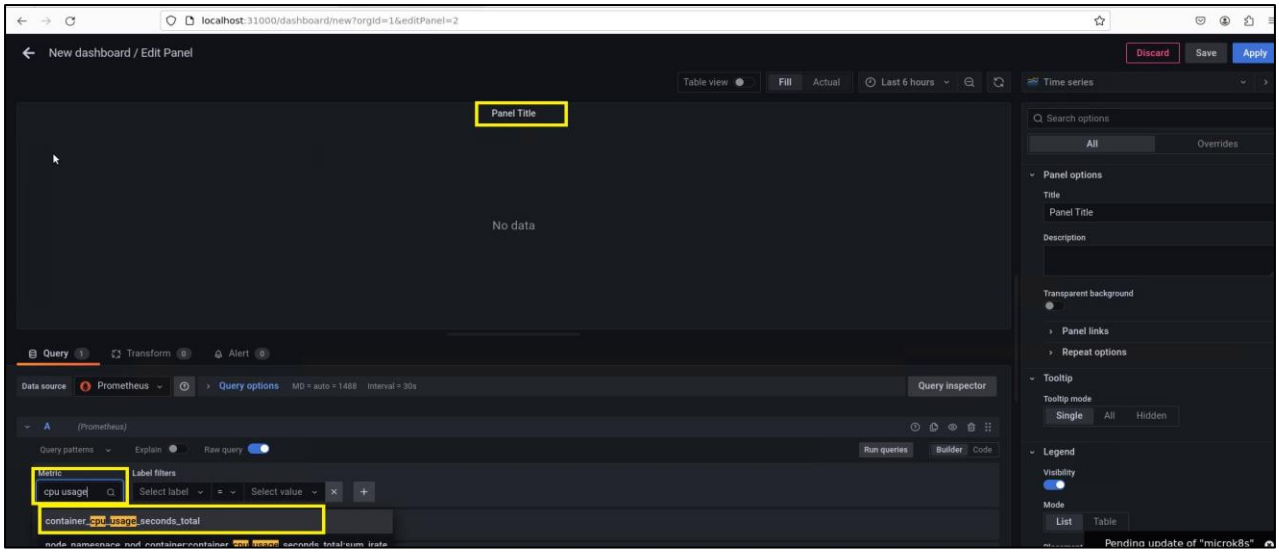
Login into Grafana and create a new dashboard.

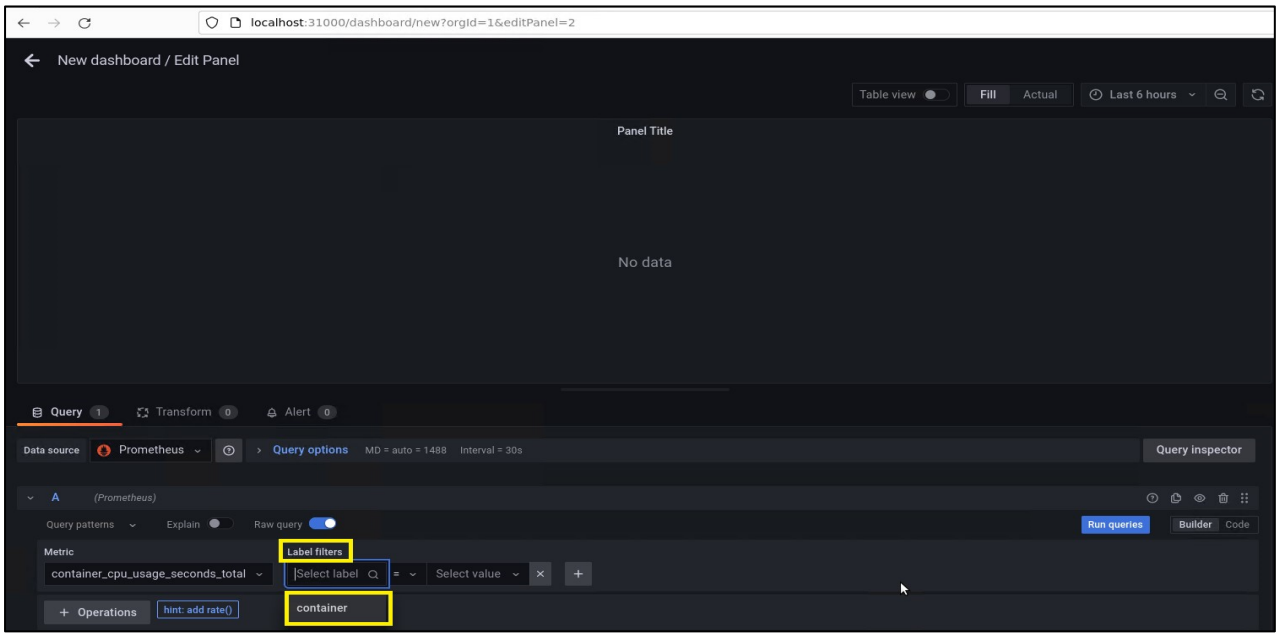


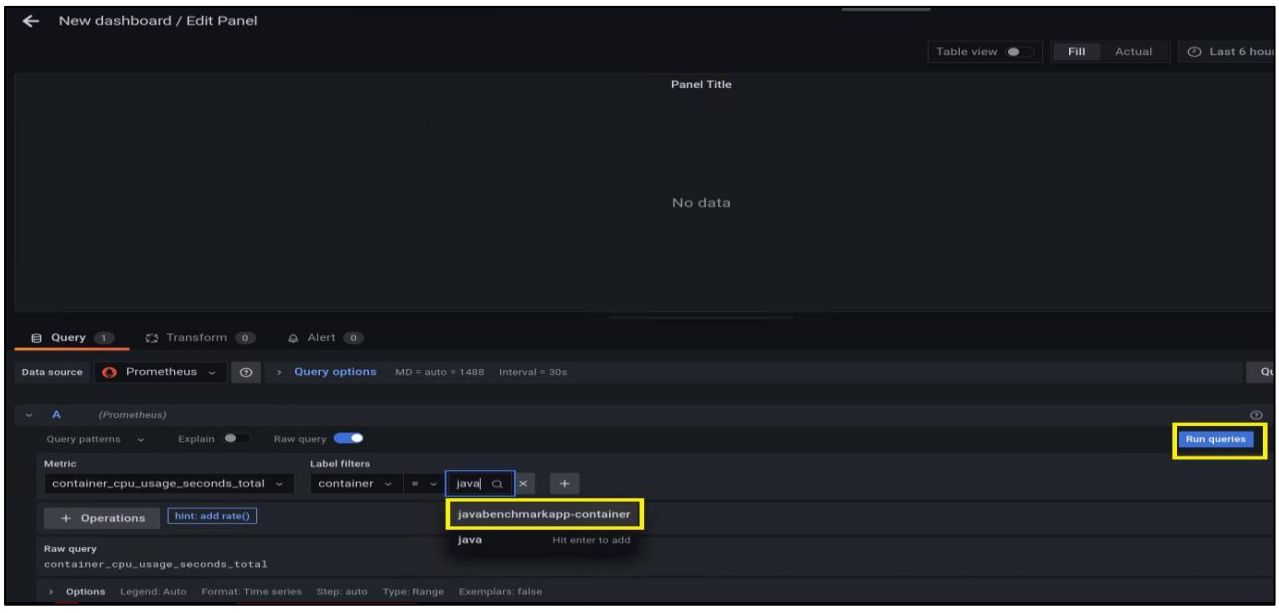
Click on Add a new panel



Give the title to the panel as “cpu_usage” and select the metric “container_cpu_usage_seconds_total". Then in Label filters select “container” and select the value as “javabenchmarkapp-container“ and click on run query.

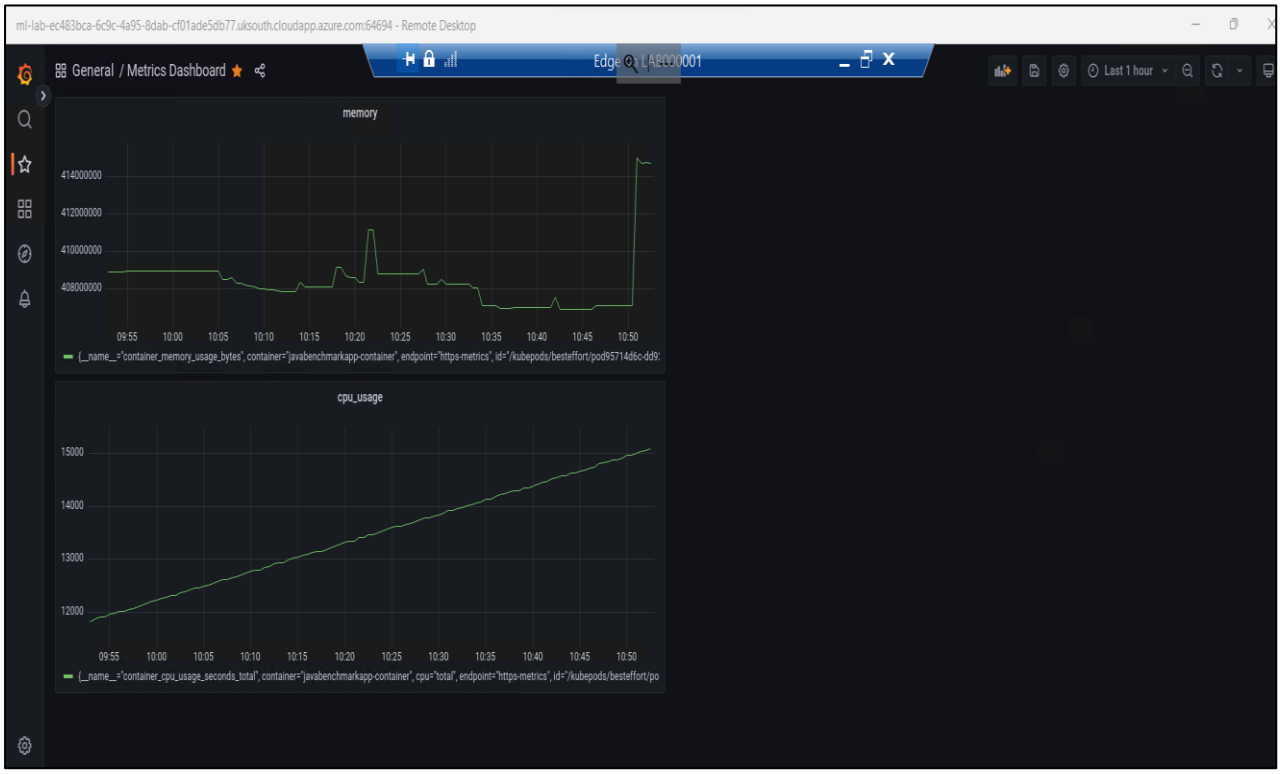

localhost:31000/dashboard/new?orgId=1&editPanel=2
New dashboard / Edit Panel
Discard
Save
Apply
Table view
Fill
Actual
Last 6 hours
Time series
Panel Title
No data
Search options
All
Overrides
Panel options
Title
Panel Title
Description
Transparent background
Panel links
Repeat options
Tooltip
Tooltip mode
Single
All
Hidden
Legend
Visibility
Mode
List
Table
Query
Transform
Alert
Data source
Prometheus
Query options
MD = auto = 1488
Interval = 30s
Query inspector
A
(Prometheus)
Query patterns
Explain
Raw query
Run queries
Builder
Code
Metric
Label filters
cpu usage
Select label
Select value
container_cpu_usage_seconds_total
Pending update of "microk8s"


localhost:31000/dashboard/new?orgId=1&editPanel=2
New dashboard / Edit Panel
Table view
Fill
Actual
Last 6 hours
Panel Title
No data
Query
Transform
Alert
Data source
Prometheus
Query options
MD = auto = 1488
Interval = 30s
Query inspector
A
(Prometheus)
Query patterns
Explain
Raw query
Run queries
Builder
Code
Metric
Label filters
container_cpu_usage_seconds_total
Select label
Select value
Operations
hint: add rate()
container

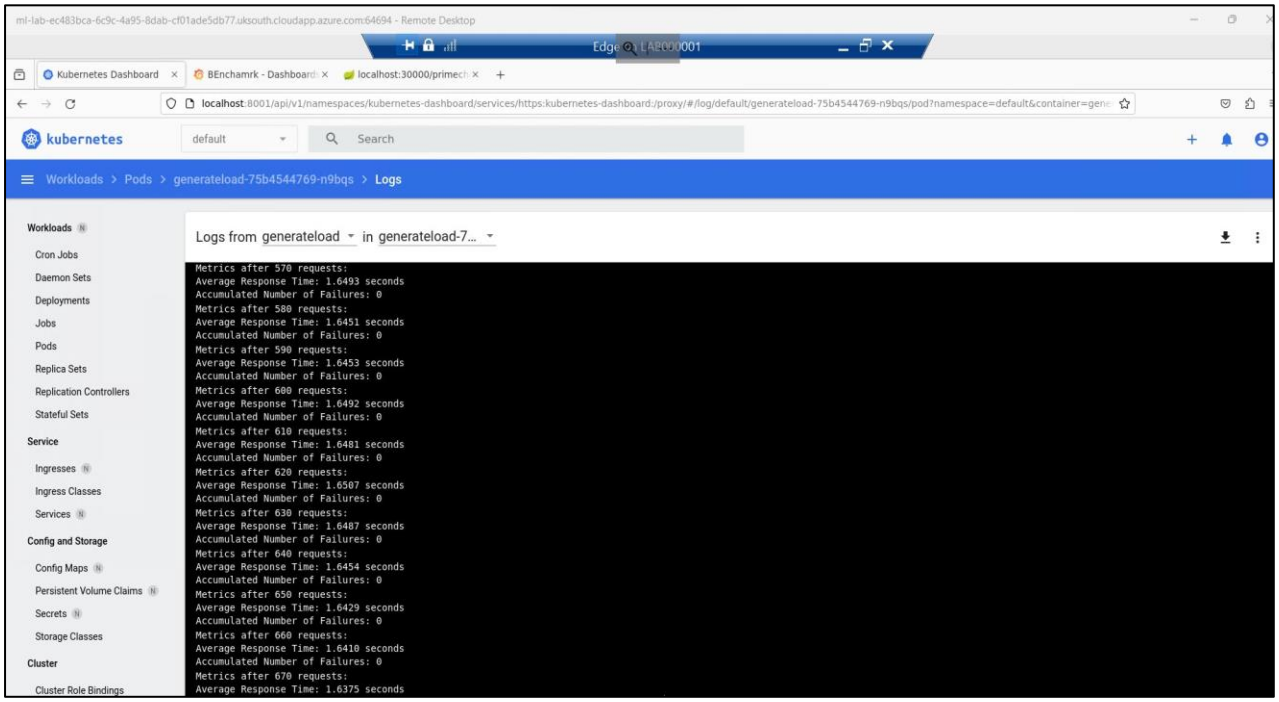

New dashboard / Edit Panel
Table view
Fill
Actual
Panel Title
No data
Query
Transform
Alert
Data source
Prometheus
Query options
MD = auto = 1488
Interval = 30s
A
(Prometheus)
Query patterns
Explain
Raw query
Run queries
Metric
Label filters
container_cpu_usage_seconds_total
container
java
Operations
hint: add rate()
javabenchmarkapp-container
java
Hit enter to add
Raw query
container_cpu_usage_seconds_total
Options
Legend: Auto
Format: Time series
Step: auto
Type: Range
Exemplars: false

Now we need to add one more panel for memory usage. Follow the same steps as above by giving title to the panel as “memory” then in metrics select “container_memory_usage_bytes”, in label filters select “container” and select value as “javabenchmarkapp-container” and click on run query.

Our dashboard will now contain two panels with metrics for total CPU usage in seconds and memory usage in bytes.



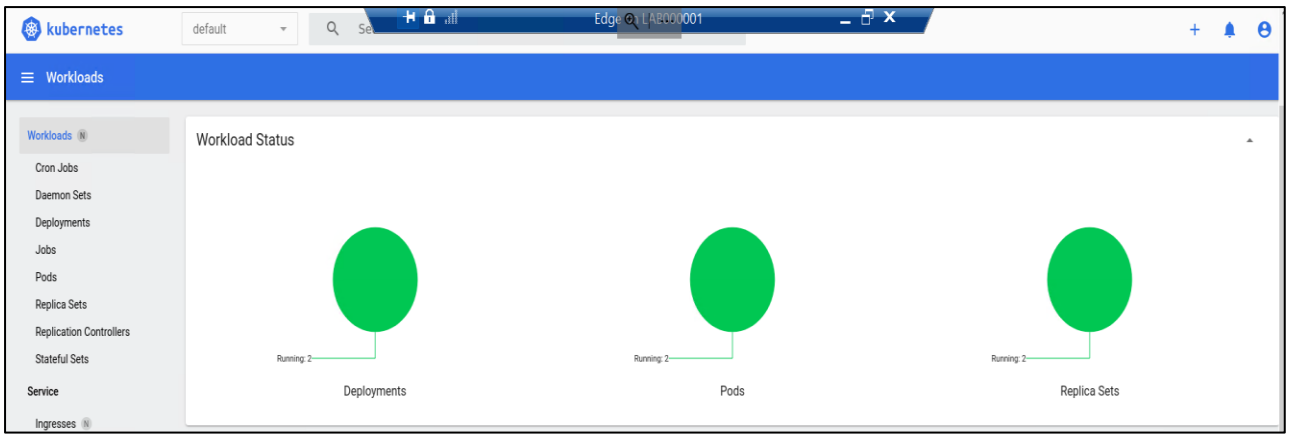
We can also see the logs of our “generateload” pod as we print the metrics such as “Average response time” and “Accumulated number of failures” in our load generator python code.



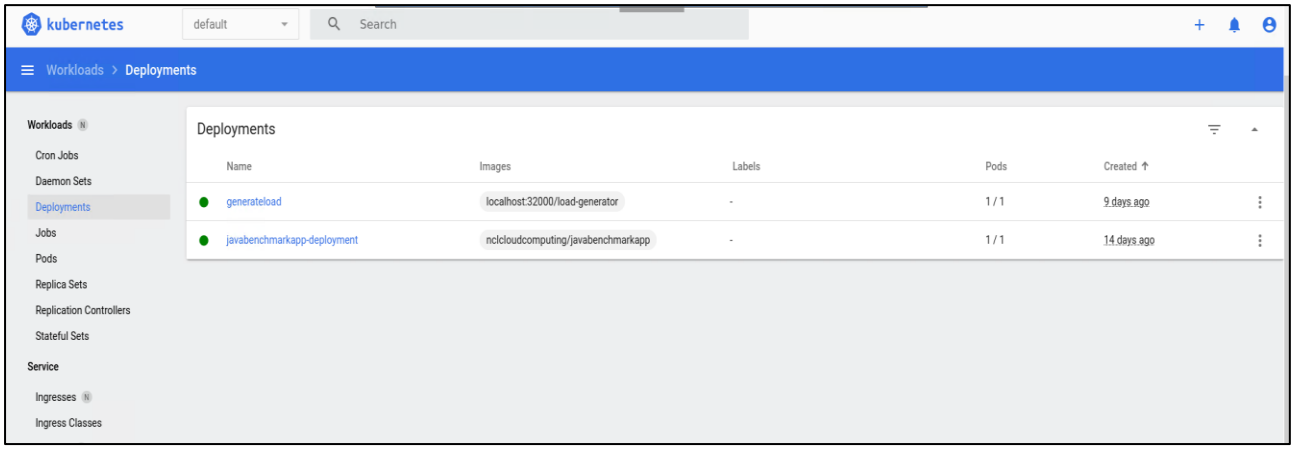
Task 5

Screenshot of running services in Kubernetes

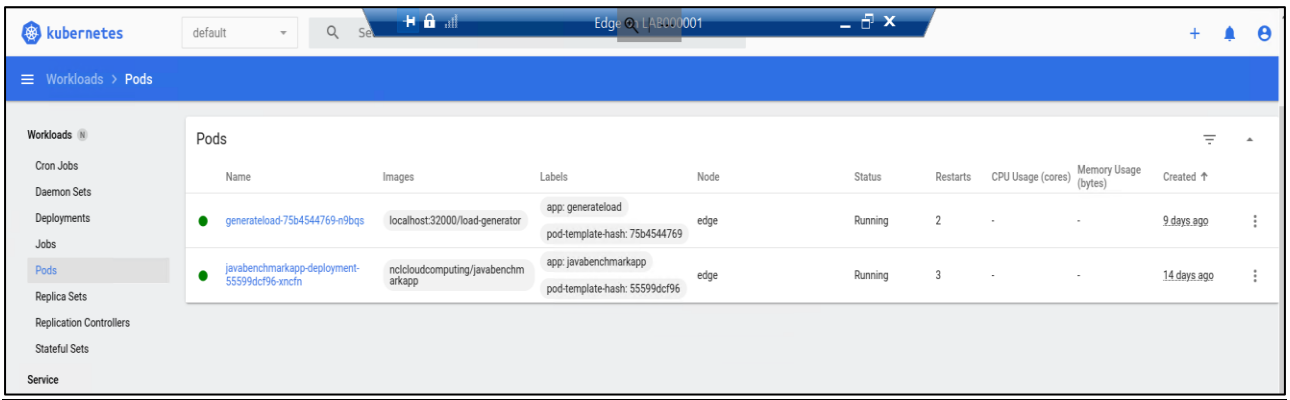
Workload Status



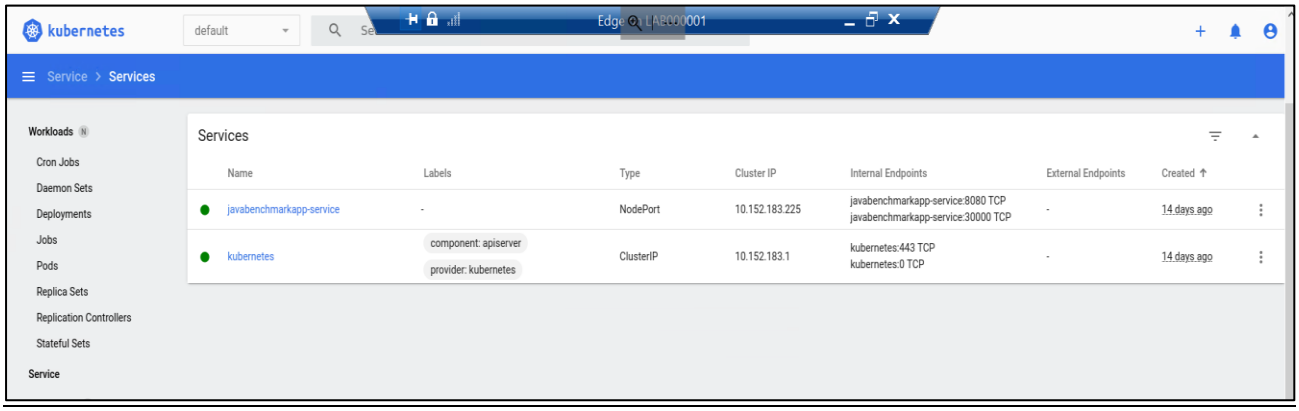
Deployments

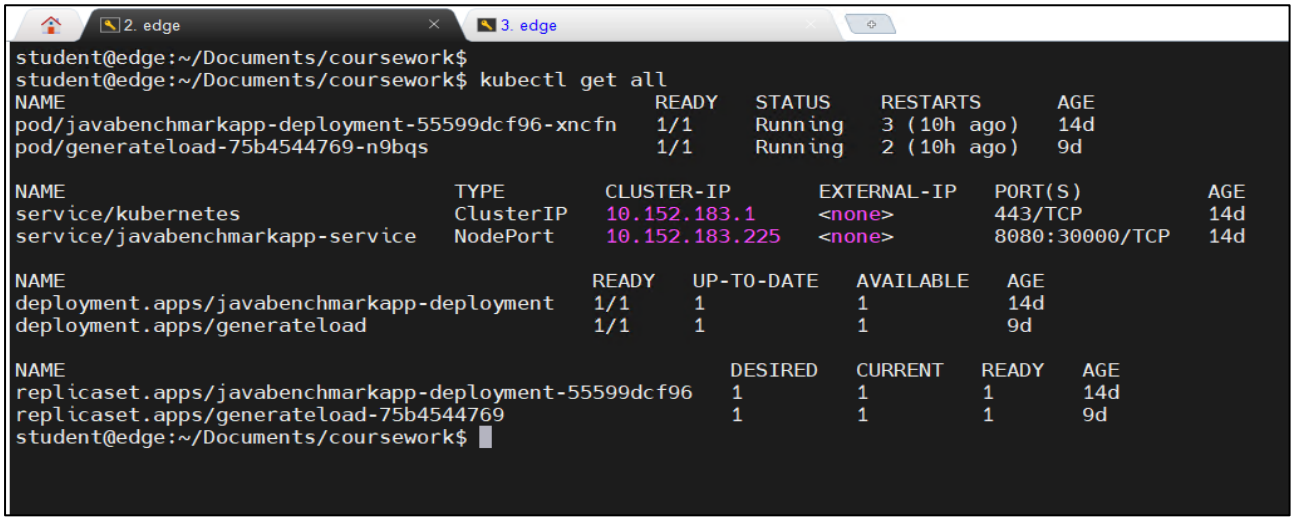


Pods



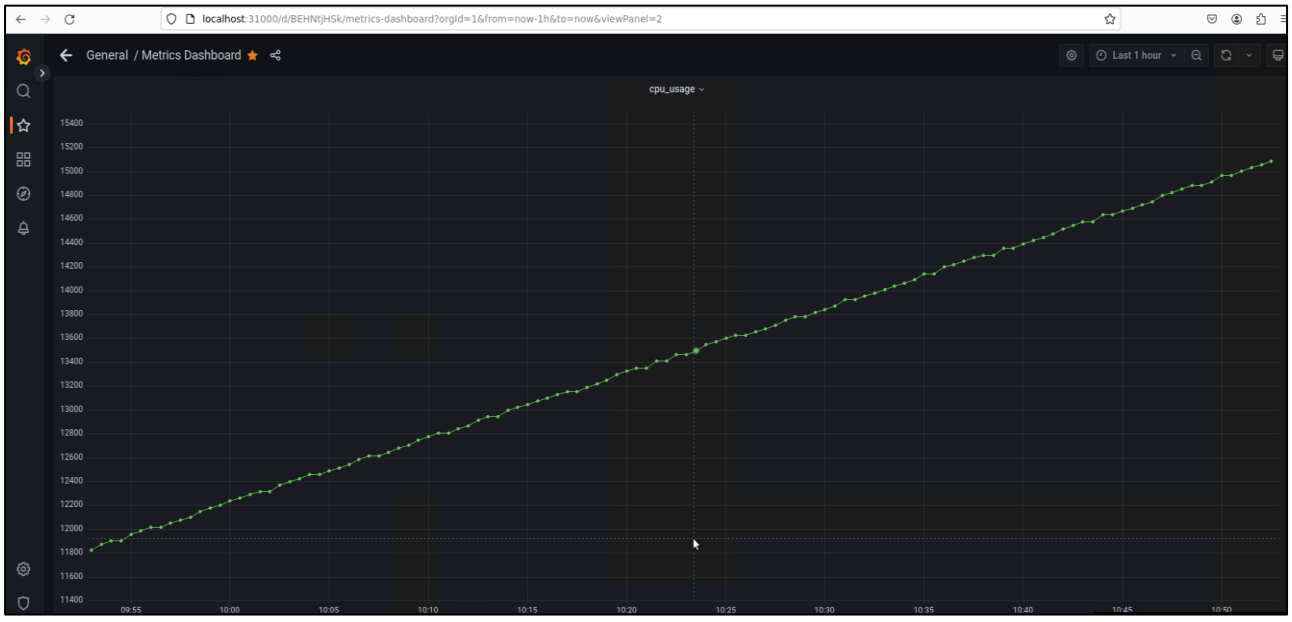
Services





Plots of benchmarking results

1. container_cpu_usage_seconds_total



2. container_memory_usage_bytes

Results and Conclusion

Talking about our first plot which is “container_cpu_usage_seconds_total“. Our load generator python code is continuously sending requests to the javabenchmarkapp. This javabenchmarkapp basically checks weather a number is prime or not on a very large number which in turn leads to generation of high load on the CPU. As a result of which we can see in the graph that CPU usage seconds keeps on increasing.

Our next plot is about “container_memory_usage_bytes”. The memory consumption by the container keeps fluctuating and is in the range of 400 to 420 megabytes. To conclude we have deployed the Kubernetes dashboard and javabenchmarkapp docker image on Kubernetes. Then with the help of our load generator python code’s deployment we are simulating the load and benchmarking the systems performance using the monitoring stack of Kubernetes and Grafana.