Cloud Computing Coursework

Student No.:240581505

AIM:

The aim of this assignment is to deploy, monitor, and benchmark containerized web applications using Kubernetes and related tools. It focuses on setting up a Kubernetes cluster with a web application, enabling monitoring with Prometheus and Grafana, and evaluating the performance of the application under load using a custom-built load generator. In such a way, it will provide hands-on experience in container performance testing, and resource monitoring in a cloud-native environment.

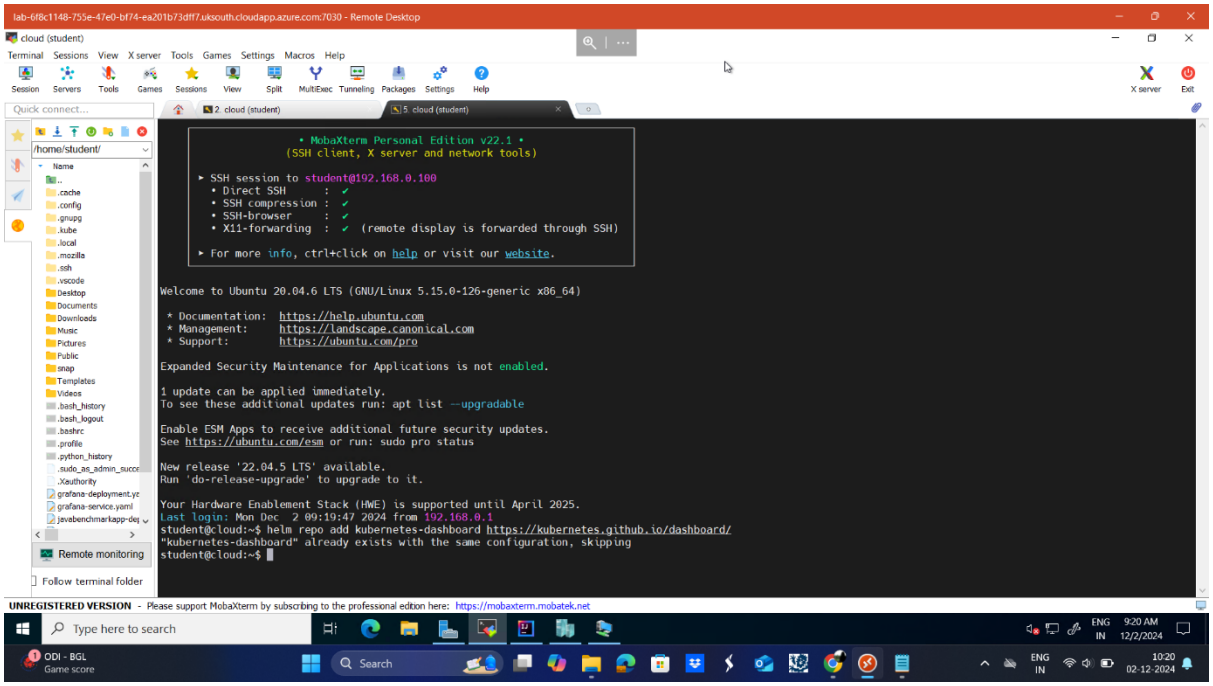
Task 1: Deploy and access the Kubernetes Dashboard and a Web Application Component

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

I deployed dashboard by adding Kubernetes dashboard repository and then add them.

Add Kubernetes-dashboard repository CLI:

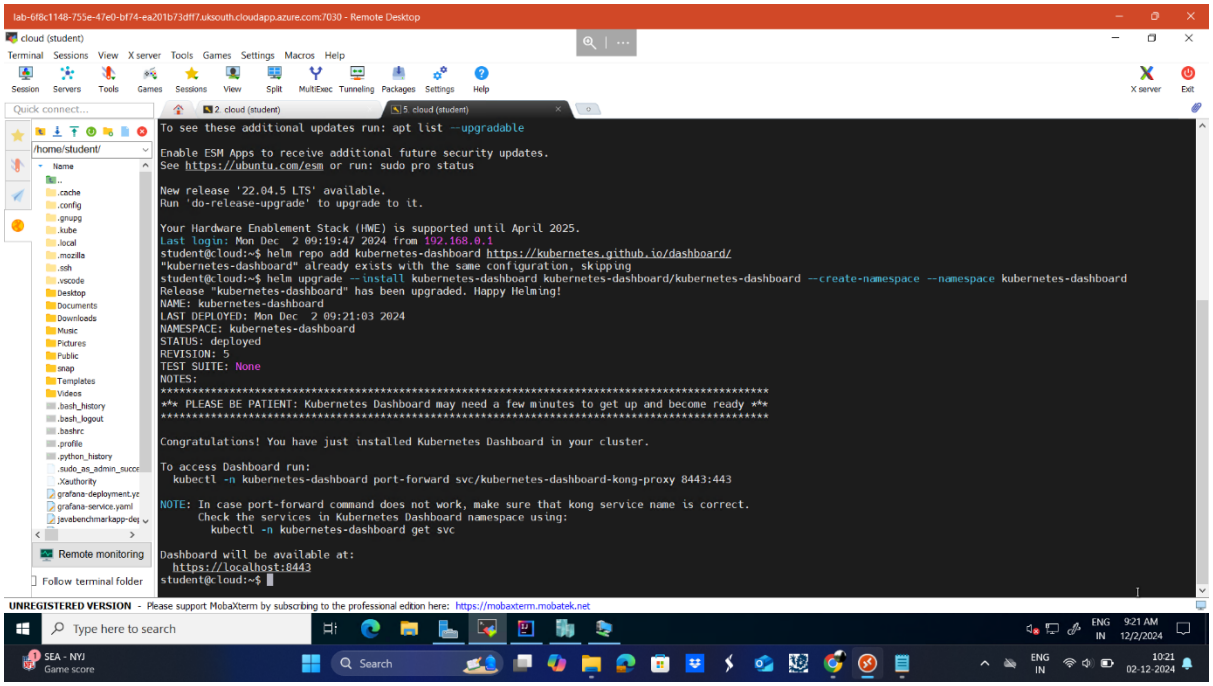
Helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/



Deploy a Helm Release named "kubernetes-dashboard" using the kubernetes-dashboard chart

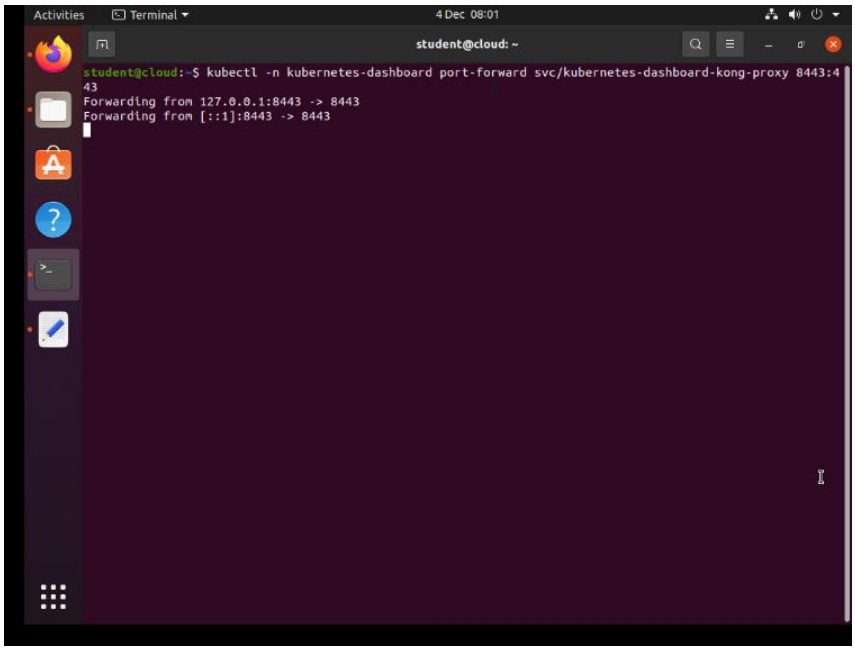
helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard --create-namespace --namespace kubernetes-dashboard

The above command will help us to install Kubernetes dashboard



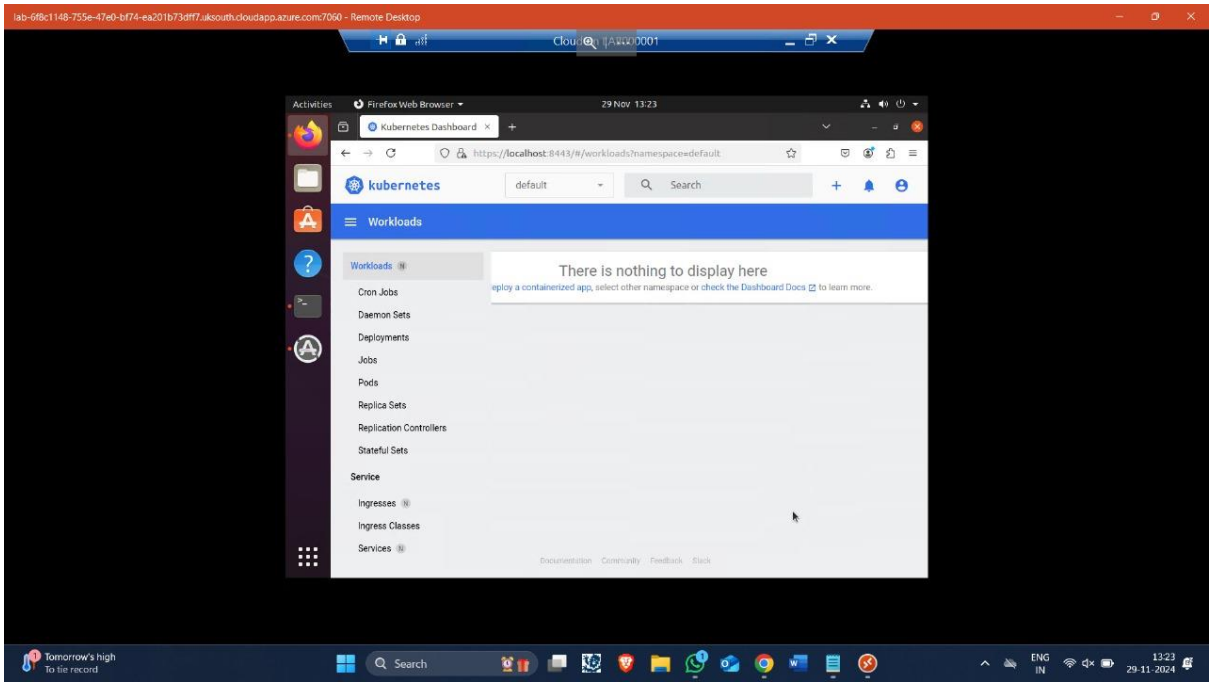
To access Dashboard, the command will be generated while we run command to install Kubernetes dashboard.

To access Dashboard CLI : kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443

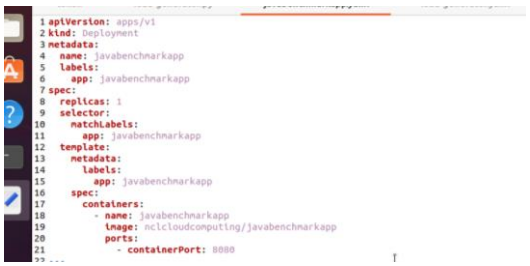


https://localhost:8443:8443

go to the browser in vm and run the above url then you will enter the Kubernetes dashboard.



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



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: javabenchmarkapp
  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
```

Explanation

1.apiVersion: apps/v1

This is the API version used when defining this resource.

apps/v1 is the stable API version for the Kubernetes Deployments.

2. kind: Deployment

The kind field specifies the type of Kubernetes resource described - in this case, a Deployment that ensures the described number of pod replicas are running and up to date.

3. metadata

name: javabenchmarkapp

This string refers to the name given to the Deployment. It is by this name the resource will be known to Kubernetes cluster.

labels

Labels are key-value pairs attached to the resource for identification and grouping.

Here, the Deployment is labeled with name: javabenchmarkapp, which can be used for selectors or grouping resources.

4. spec

The spec field contains the desired state of the Deployment.

4.1. replicas: 1

Specifies the number of pod replicas to run.

In this case, it ensures one pod of javabenchmarkapp is always running.

4.2. selector

matchLabels

Used to match pods to this Deployment based on their labels.

Pod having the label app: javabenchmarkapp will be managed by this Deployment.

4.3. template

Define the template for the pods that the Deployment manages.

5. Pod Template (template)

The template section defines the desired configuration for the pods created by this Deployment.

5.1. metadata

labels

Add a label app: javabenchmarkapp to each pod created by this Deployment.

This label has to match the matchLabels in the selector to ensure proper management.

5.2. spec

Defines the pod's configuration, including its containers. 6. containers Lists the containers that constitute a pod.

6.1. name: javabenchmarkapp The name of the container. This is useful to identify and manage the container.

6.2. image: nclcloudcomputing/javabenchmarkapp Specifies which container image to use. Here, the image is pulled from a container registry. nclcloudcomputing/javabenchmarkapp

6.3. ports containerPort: 8080 Exposes port 8080 on the container to the outside world for communication. This must match the port used by the application inside the container.

CLI for deploying the file:

kubectl apply -f javabenchmarkapp.yaml

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

Explanation

1. apiVersion: v1

Defines the API version to use for this resource.

v1 is the stable API version for Services in Kubernetes.

2. kind: Service

Kubernetes resource type being defined: a Service.

Services in Kubernetes expose one or more running Pods to internally or externally bound network traffic and provides load balancing.

3. metadata

Information about the Service:

name: javabenchmarkapp-service

The name of the Service. It will identify this Service in the cluster.

4. spec

The spec section defines the desired state of the Service.

4.1. type: NodePort

NodePort exposes the Service on a static port on each worker node in the cluster.

This allows external traffic to access the application.

4.2. selector

app: javabenchmarkapp

This specifies the label of the pods that the Service targets.

Any pod with the label app: javabenchmarkapp will be exposed by this Service.

4.3. ports

The ports section defines how the Service maps traffic to the targeted pods.

protocol: TCP

Specifies the protocol used by the Service; in this case, TCP.

port: 8080

The port exposed by the Service that other services/pods in the cluster can use to access this Service.

targetPort: 8080

The port on the pod where the application is running. It routes traffic to this port inside the container.

nodePort: 30000

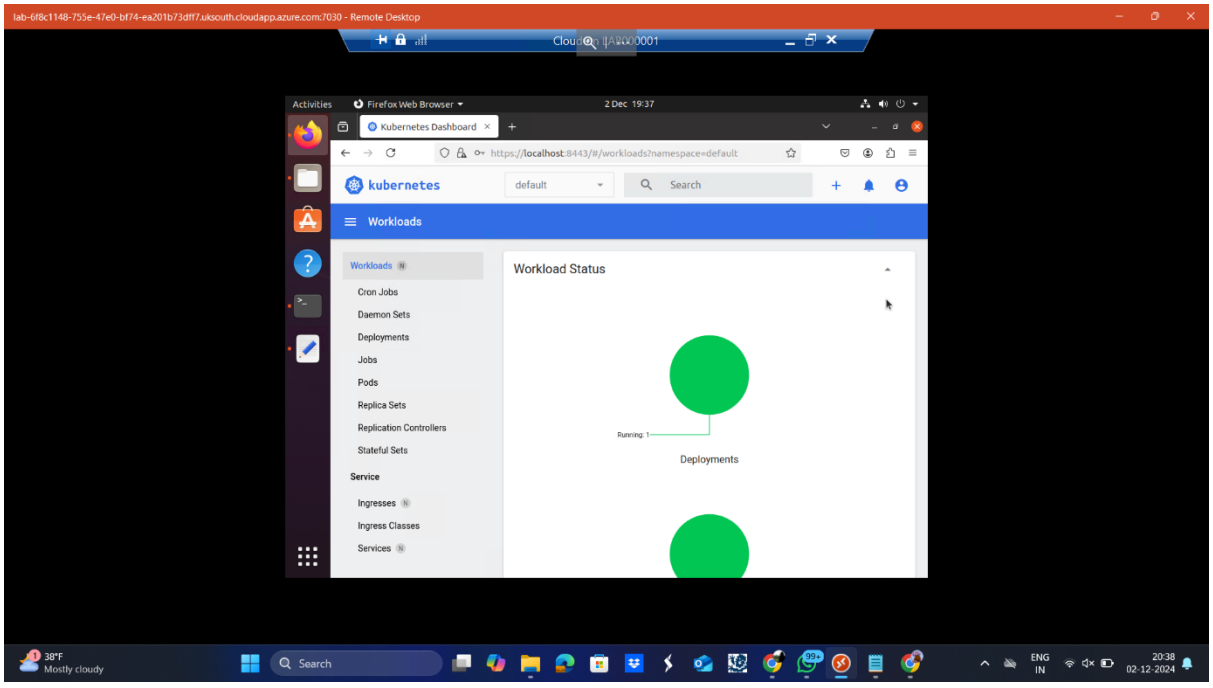
The port exposed on each worker node in the cluster to direct external traffic to the Service.

This allows users outside the cluster to access the application at http://<NodeIP>:30000

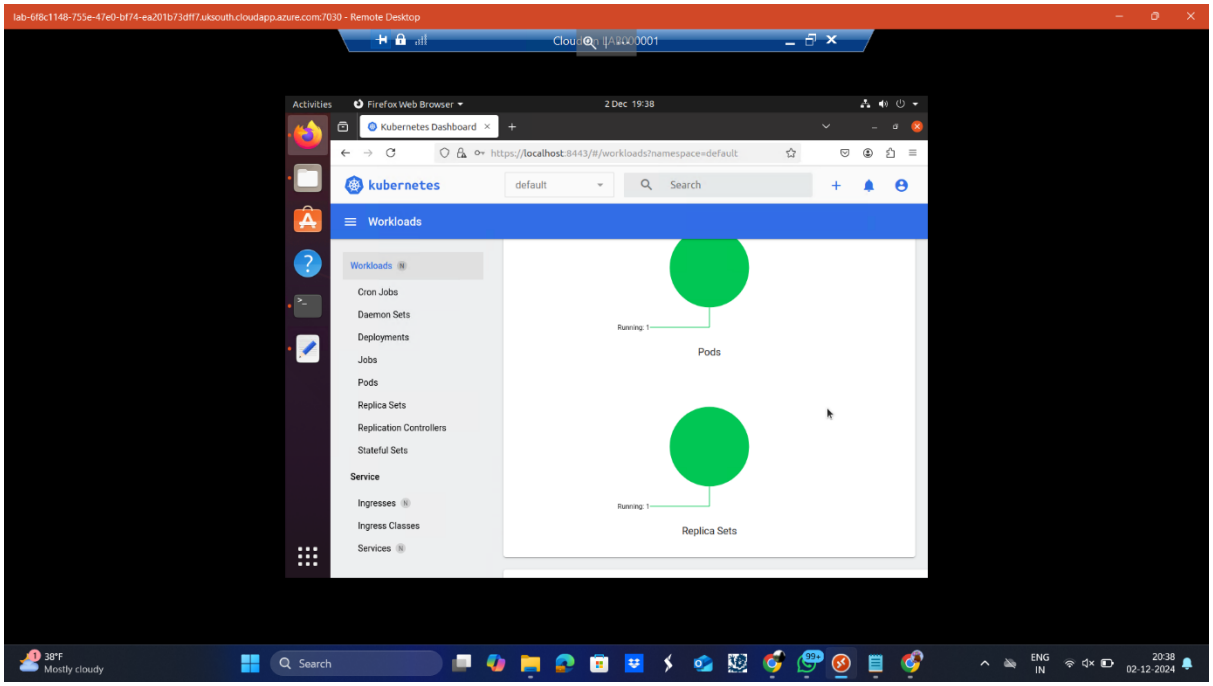
CLI for deploying the file:

kubectl apply -f javabenchmarkappservice.yaml

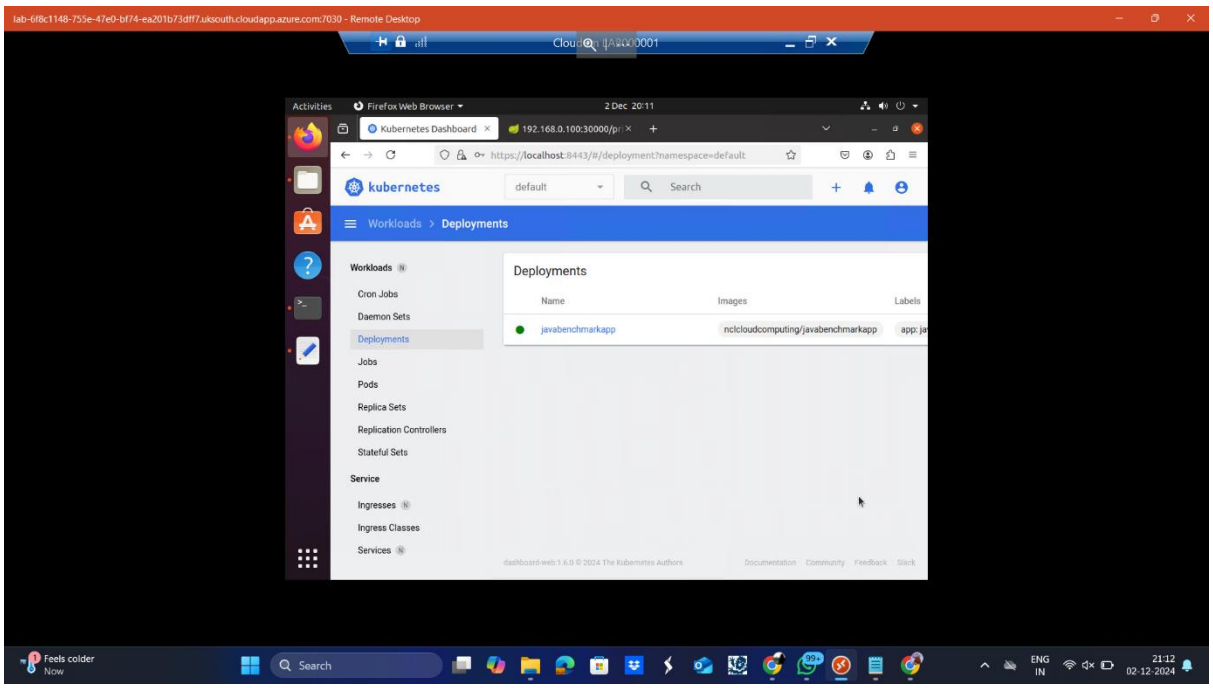
Now both the files are applied so now we can view our deployment and service in Kubernetes dashboard. The screenshots are attached below:



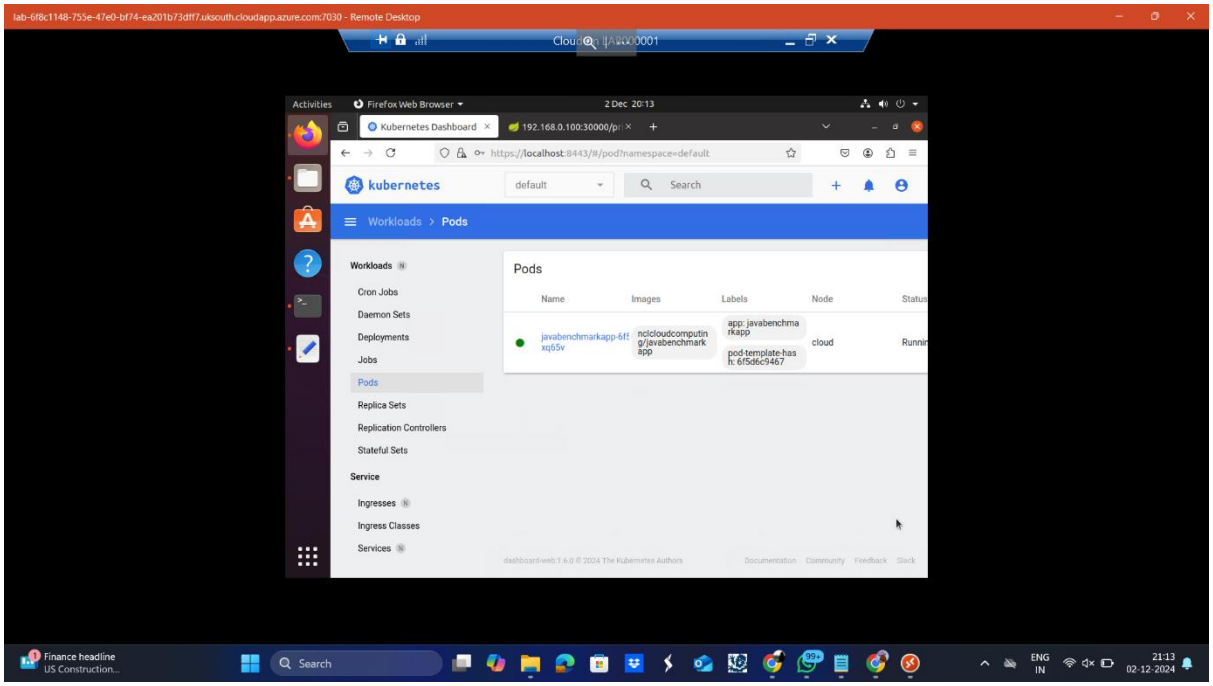
Kubernetes Dashboard Workloads Overview The overview of the workloads in the default namespace. It indicates that 1 deployment is successfully running.



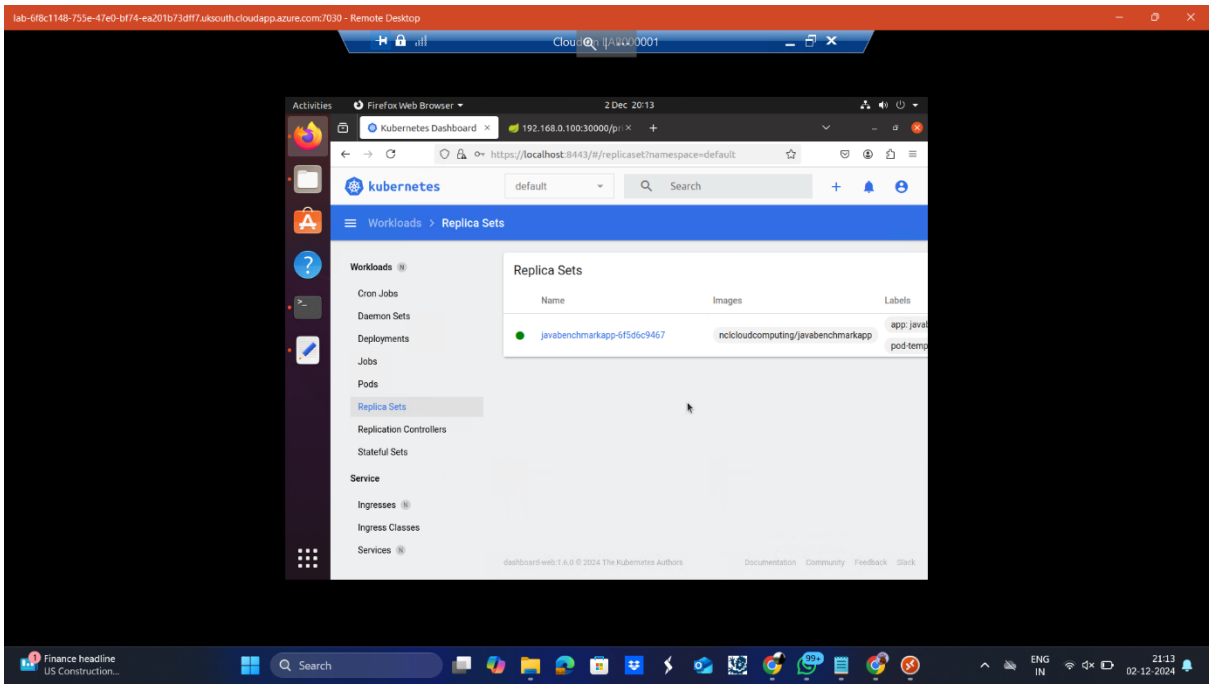
This above screenshot gives a breakdown of the running workloads, including:
Pods: The actual application instances.
Replica Sets: The controller ensuring the desired number of pods is maintained.



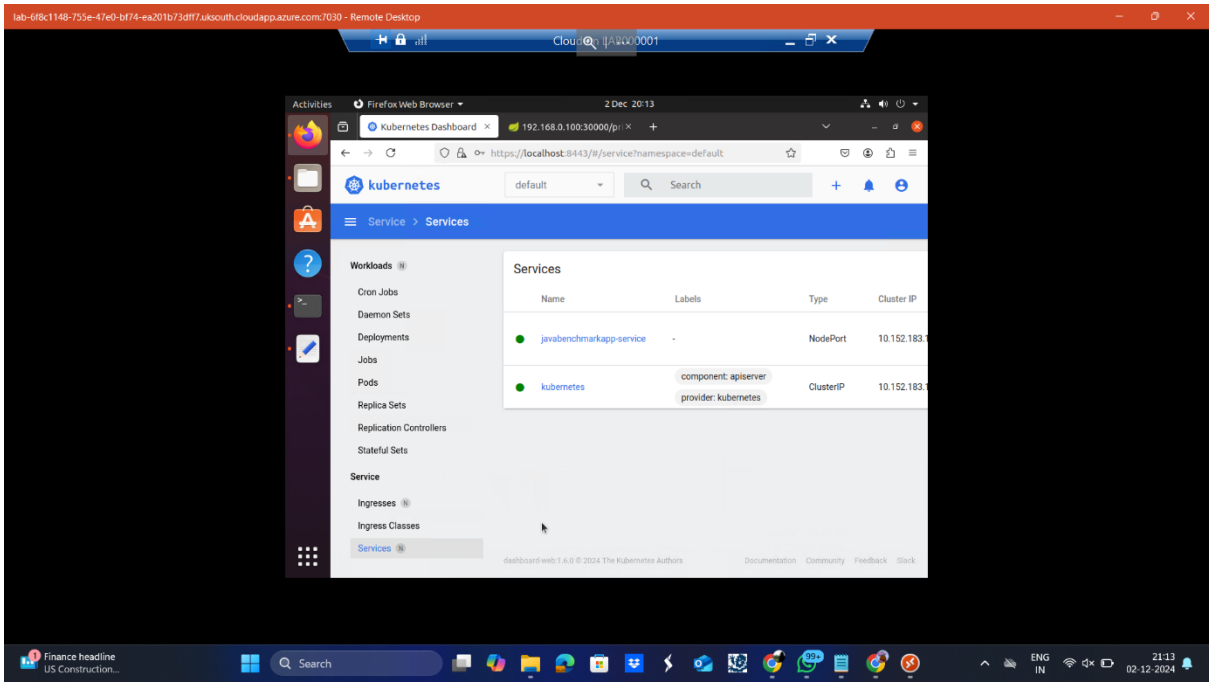
The above Deployment Details focuses on the deployment javabenchmarkapp.
Confirms that the image nclcloudcomputing/javabenchmarkapp is in use.



The above Pod Details lists the pod associated with the javabenchmarkapp deployment.
Indicates:
The pod is running successfully.
The image used and the associated labels.



The above Replica Set Details outlines the replica set for javabenchmarkapp.
Verifies the replica set is managing the pod with an image
matching the expectation set.



The above screenshot shows Service Overview
This shows the services in the default namespace.
Lists:
javabenchmarkapp-service: A NodePort service that exposes the application
ClusterIP: Internal Kubernetes service for communication between services.

Task 2: Deploy the monitoring stack of Kubernetes

1.Grafana must reach Prometheus as a data source

The Kube Prometheus Stack deploys Prometheus and Grafana as part of the same Helm chart. Grafana is configured with Prometheus as a default data source. To do this we need to set up Kube-Prometheus-Stack in a Kubernetes cluster using Helm.

Step 1: Add Helm Repository: The prometheus-community Helm repository is added to the local Helm client.

Command: helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

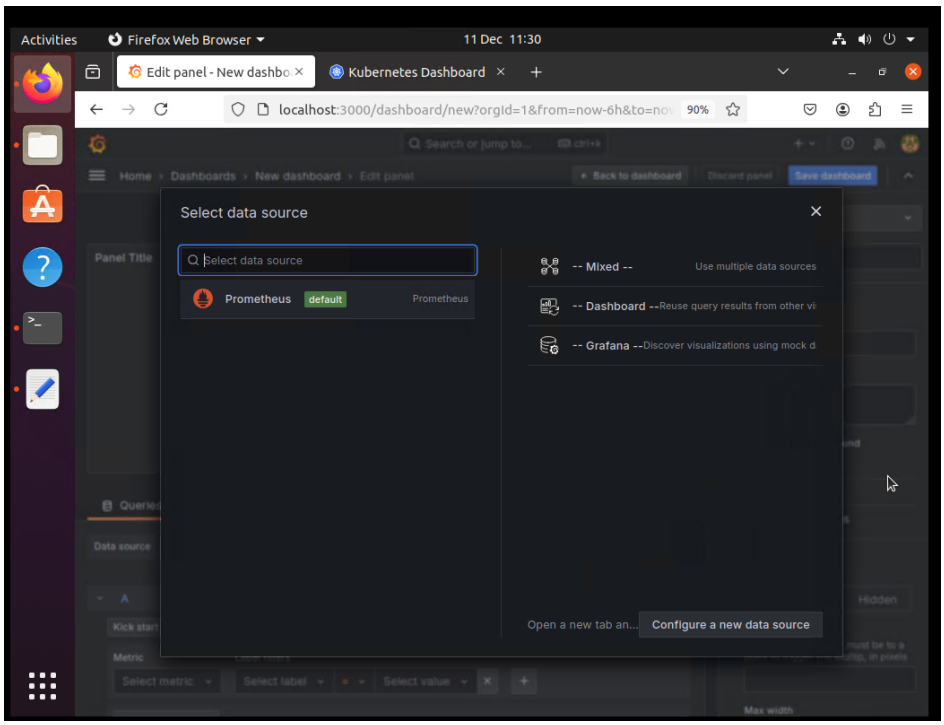
Step 2: Update Repositories: Updates the local index of Helm charts to ensure the latest versions from the added repository are available.

Command: helm repo update

Step 3: Install Kube-Prometheus-Stack: Installs the kube-prometheus-stack Helm chart from the prometheus-community repository into the monitoring namespace. The namespace is created if it doesn't already exist (--create-namespace).

Command: helm install kube-prometheus-stack prometheus-community/kube-prometheus-stack --namespace monitoring --create-namespace

Therefore Grafana is reaching Prometheus as a data source. It is shown in below screenshot.



Below screenshot shows all the 3 commands running

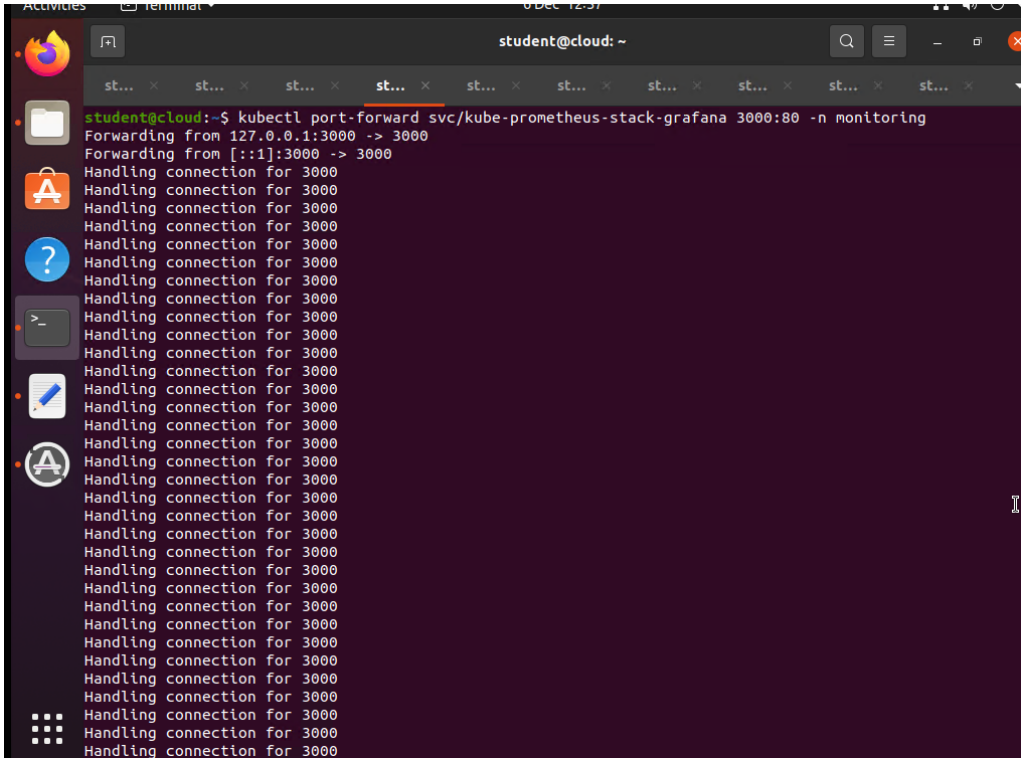
```
student@cloud:~$ helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
"prometheus-community" has been added to your repositories
student@cloud:~$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "kubernetes-dashboard" chart repository
...Successfully got an update from the "prometheus-community" chart repository
Update Complete. ⎈Happy Helming!⎈
student@cloud:~$ helm install kube-prometheus-stack prometheus-community/kube-prometheus-stack --namespa
ce monitoring --create -namespace
Error: unknown flag: --create
student@cloud:~$ helm install kube-prometheus-stack prometheus-community/kube-prometheus-stack --namespa
ce monitoring --create-namespace
NAME: kube-prometheus-stack
LAST DEPLOYED: Thu Dec  5 17:20:03 2024
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
NOTES:
kube-prometheus-stack has been installed. Check its status by running:
  kubectl --namespace monitoring get pods -l "release=kube-prometheus-stack"

Visit https://github.com/prometheus-operator/kube-prometheus for instructions on how to create & configu
re Alertmanager and Prometheus instances using the Operator.
student@cloud:~$ kubectl get pods -n monitoring
NAME                                                     READY   STATUS            RESTARTS   AGE
kube-prometheus-stack-prometheus-node-exporter-lrb8m     1/1     Running           0          34s
kube-prometheus-stack-operator-5c68fdcd6d-9mx9s          1/1     Running           0          34s
kube-prometheus-stack-kube-state-metrics-cbb545d5b-9dq9z 1/1     Running           0          34s
prometheus-kube-prometheus-stack-prometheus-0            0/2     PodInitializing   0          26s
kube-prometheus-stack-grafana-7444c98c6f-dszsp           2/3     Running           0          34s
alertmanager-kube-prometheus-stack-alertmanager-0        2/2     Running           0          27s
student@cloud:~$
```

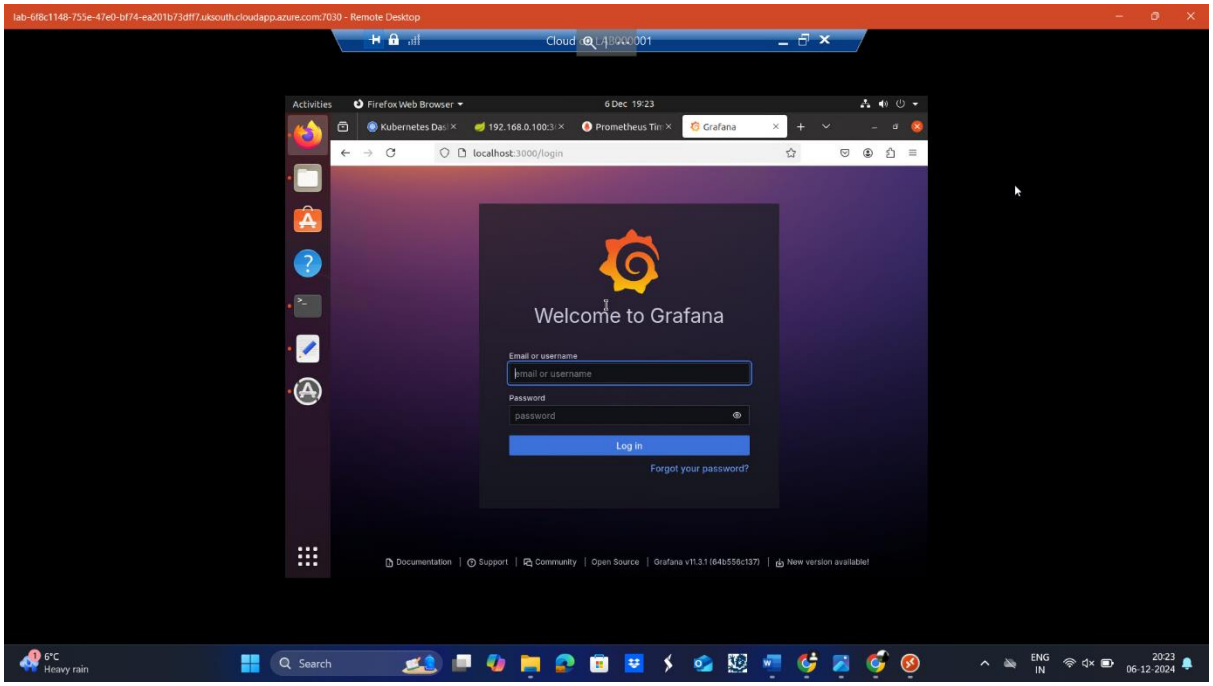
2.Grafana must be reachable from the host

As we installed Kube Prometheus stack, Grafana is deployed to local Helm client.

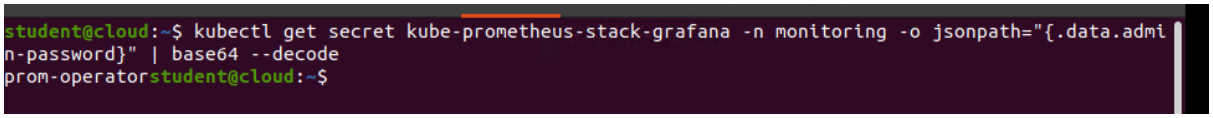
Now to access the Grafana the command we need to provide is



Now Grafana has been deployed and I am opening them with the localhost url

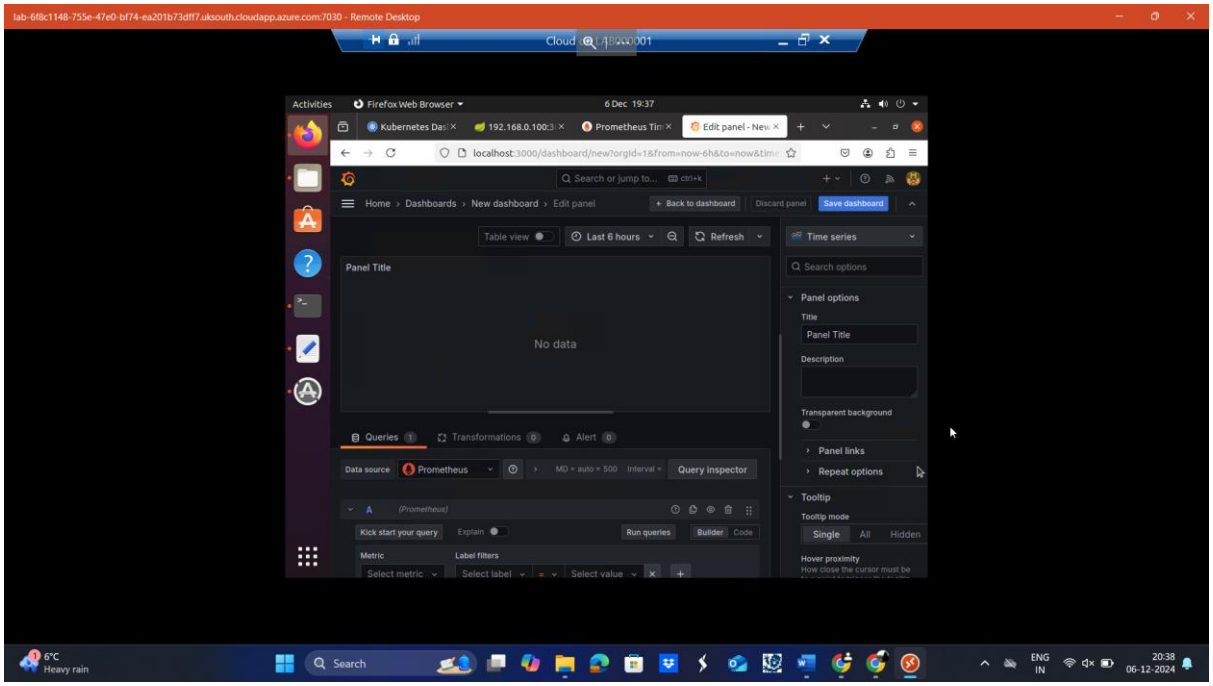


Now to generate password, the command I use is

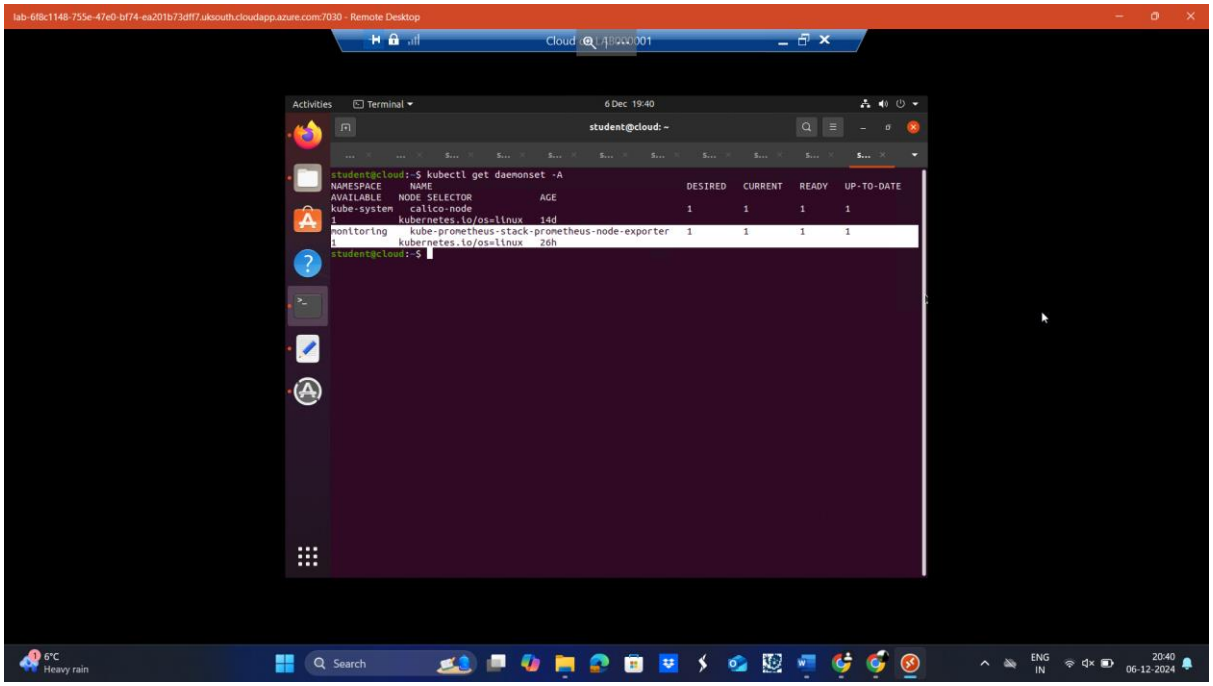


Log in with the username as admin and password as prom-operator (which we generated)

3.Prometheus must reach each instance of the metric server to collect metrics

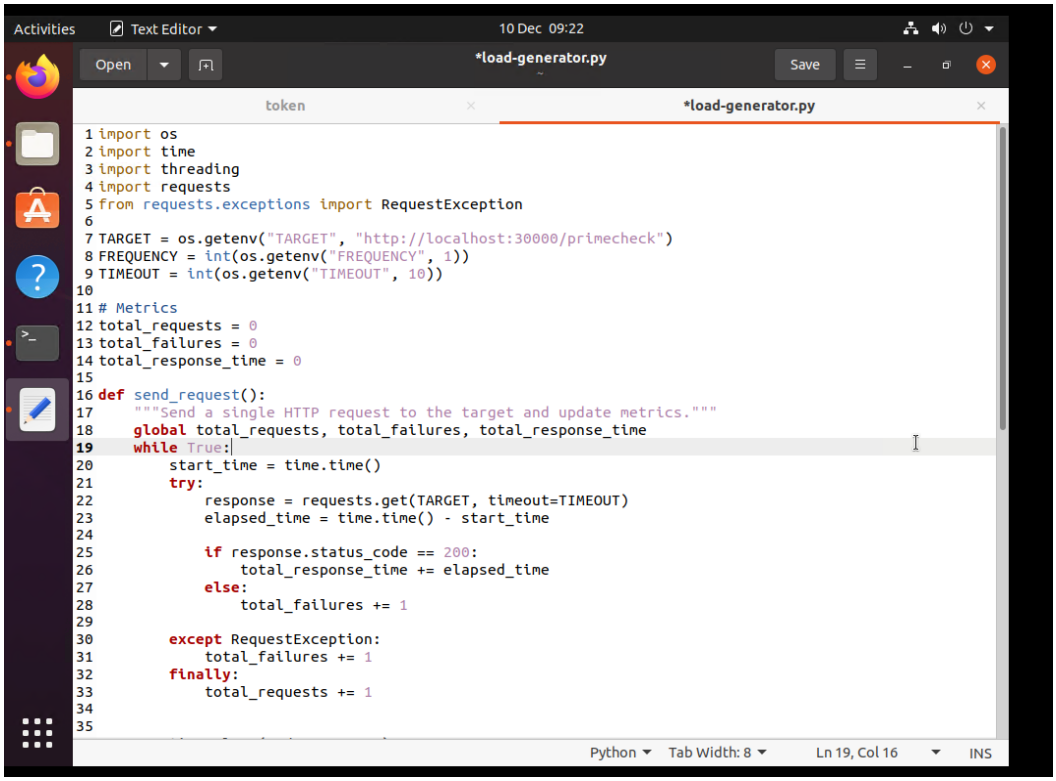


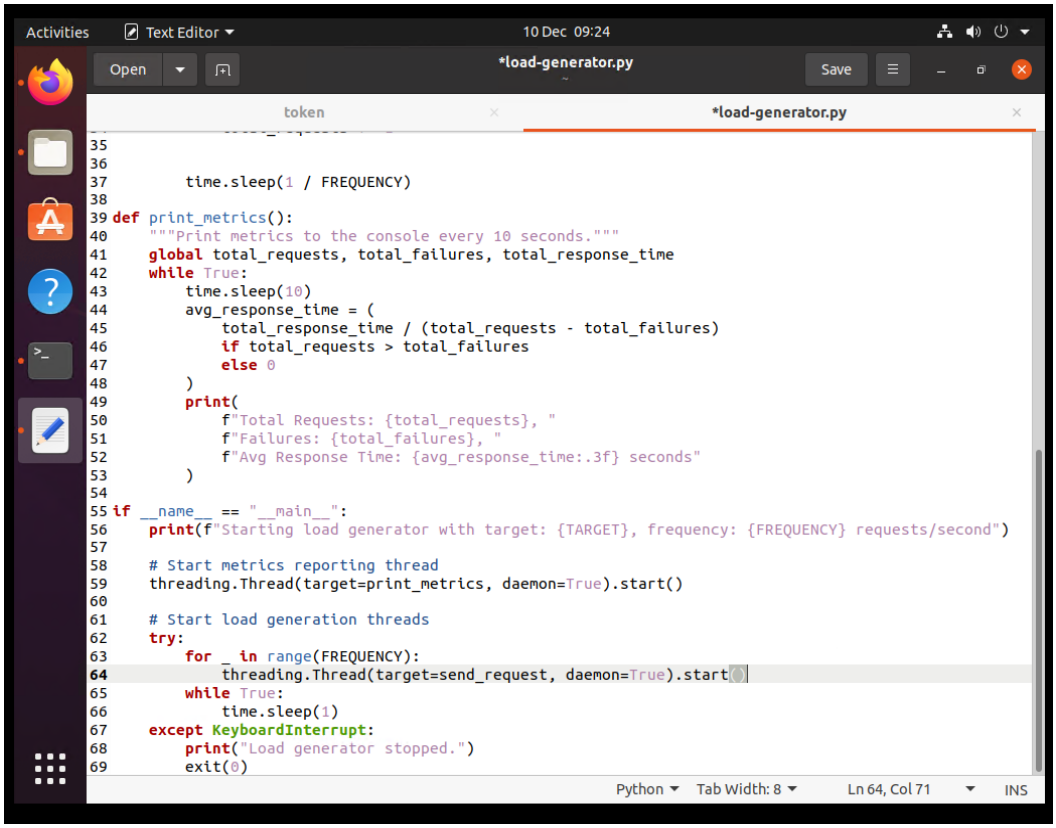
4.The metric server must be deployed as a DaemonSet: As we installed Kube Prometheus Stack, damen set will be installed automatically from repo.



Task 3: Load Generator

1.Load generator python code





Code Explanation:

The Python code here sends many HTTP requests to a given TARGET web application for the purpose of load testing. The requests per second can be manipulated with the FREQUENCY variable to allow through only a certain number of requests per second.

It uses a TIMEOUT variable for limiting how long each request is allowed to wait without response before being counted as a failure.

It picks up key metrics in real-time, including:

- Total requests sent.
- Number of failed requests.
- Average response time for successful requests.

The send_request function:

Sends HTTP requests to the target.

Measures response times for successful requests.

Updates the metrics for total requests, failures, and response time.

The print_metrics function:

Runs in a separate thread.

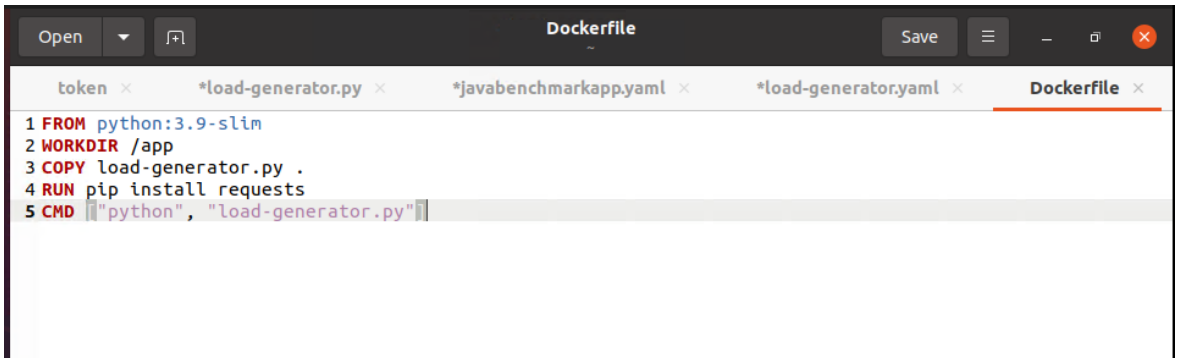
Displays metrics - total requests, failures, and average response time, every 10 seconds to monitor performance.

The main program:

Initializes the threads for metrics reporting and request generation.

Launches multiple threads to send requests concurrently in order to achieve the desired rate.

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*.



Dockerfile explanation:

FROM python:3.9-slim:

The base image for the container will be a minimal Python 3.9 image, optimized for smaller size.

WORKDIR /app:

Creates and sets /app as the working directory where all subsequent operations will take place.

COPY load-generator.py .:

Copies the load-generator.py script from your local system into the /app directory of the container.

RUN pip install requests:

Installs the requests Python library in the container. The load-generator.py script requires this.

CMD ["python", "load-generator.py"]:

Sets the default command to run when the container starts: run the load-generator.py script with Python.

Now the contsiner is crested to check the log enter the following command:

kubectl logs -f load-generaor-7dfsdc47b7bsrm7

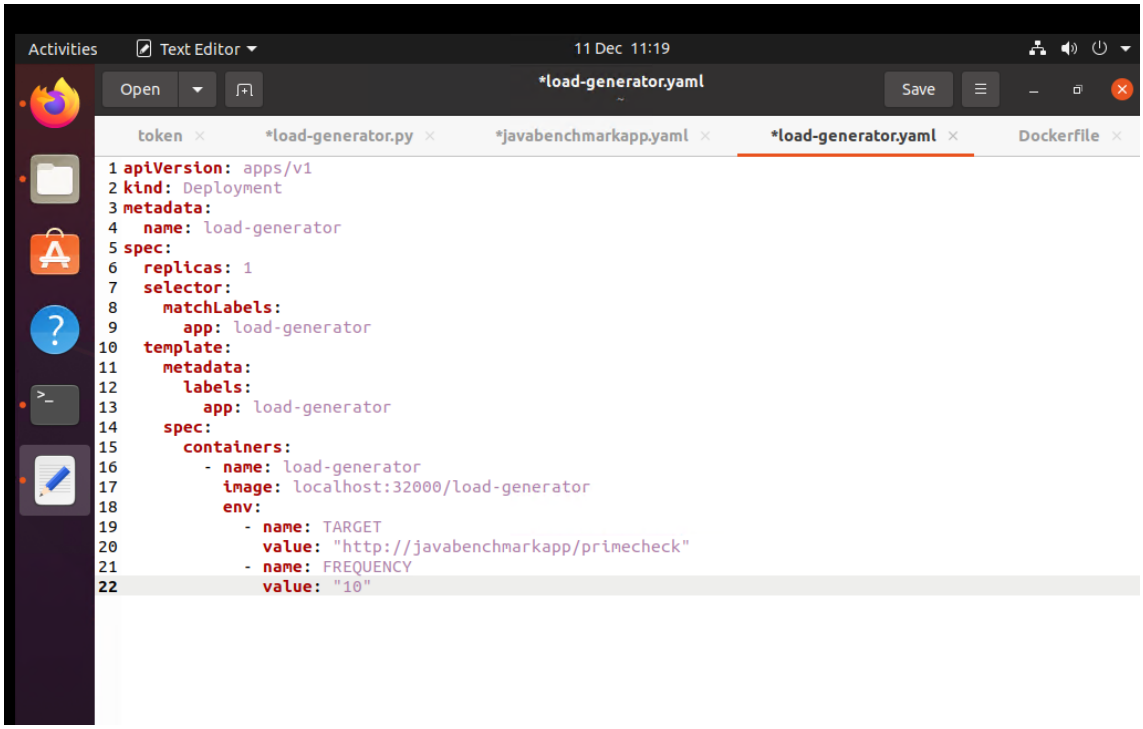
After running the above command, you see the data that is been containerised.

Screenshot has been added below.

```
student@cloud:~$ kubectl logs -f load-generator-7df5dc47b7-b5rm7
Starting load generator with target: http://javabenchmarkapp/primecheck, frequency: 10 requests/second
Total Requests: 901, Failures: 901, Avg Response Time: 0.000 seconds
Total Requests: 1856, Failures: 1856, Avg Response Time: 0.000 seconds
Total Requests: 2801, Failures: 2801, Avg Response Time: 0.000 seconds
Total Requests: 3752, Failures: 3752, Avg Response Time: 0.000 seconds
Total Requests: 4690, Failures: 4690, Avg Response Time: 0.000 seconds
Total Requests: 5643, Failures: 5643, Avg Response Time: 0.000 seconds
Total Requests: 6585, Failures: 6585, Avg Response Time: 0.000 seconds
Total Requests: 7534, Failures: 7534, Avg Response Time: 0.000 seconds
Total Requests: 8481, Failures: 8481, Avg Response Time: 0.000 seconds
Total Requests: 9432, Failures: 9432, Avg Response Time: 0.000 seconds
Total Requests: 10383, Failures: 10383, Avg Response Time: 0.000 seconds
Total Requests: 11328, Failures: 11328, Avg Response Time: 0.000 seconds
Total Requests: 12281, Failures: 12281, Avg Response Time: 0.000 seconds
```

Task 4: Monitor benchmarking results

1. **Deploy *load-generator* service created in Task 3.**



Load-generator.yaml explanation:

apiVersion: apps/v1:

Specifies the API version for the Kubernetes Deployment resource.

kind: Deployment:

Declares that this configuration is for a Kubernetes Deployment.

metadata:

Contains metadata information about the Deployment resource.

name: load-generator:

Sets the name of the Deployment to load-generator.

spec:

Defines the desired state and specifications of the Deployment.

replicas: 1

Specifies that one pod instance (replica) of the load-generator should run.

selector:

Defines how the Deployment selects the pods it manages.

matchLabels:

Define which labels are required to consider a pod managed by this Deployment. app: load-generator

Match the label app=load-generator to select and manage the pod. template:

Describes the template to use for creating the pods. metadata:

Metadata about the created pod. labels:

Apply labels to the pod. app: load-generator

Sets the label app=load-generator for the pod. spec:

The specification of the pod and its containers. containers:

Defines the containers that will run inside the pod.

- name: load-generator

Names the container load-generator.

image: localhost:32000/load-generator

Specifies the container image to use, pulled from the local registry.

env:

Declares environment variables to be passed to the container.

- name: TARGET

Defines the TARGET environment variable for the container.

value: "http://javabenchmarkapp/primecheck"

Sets the value of TARGET to the specified web application URL.

- name: FREQUENCY

Defines the FREQUENCY environment variable for the container.

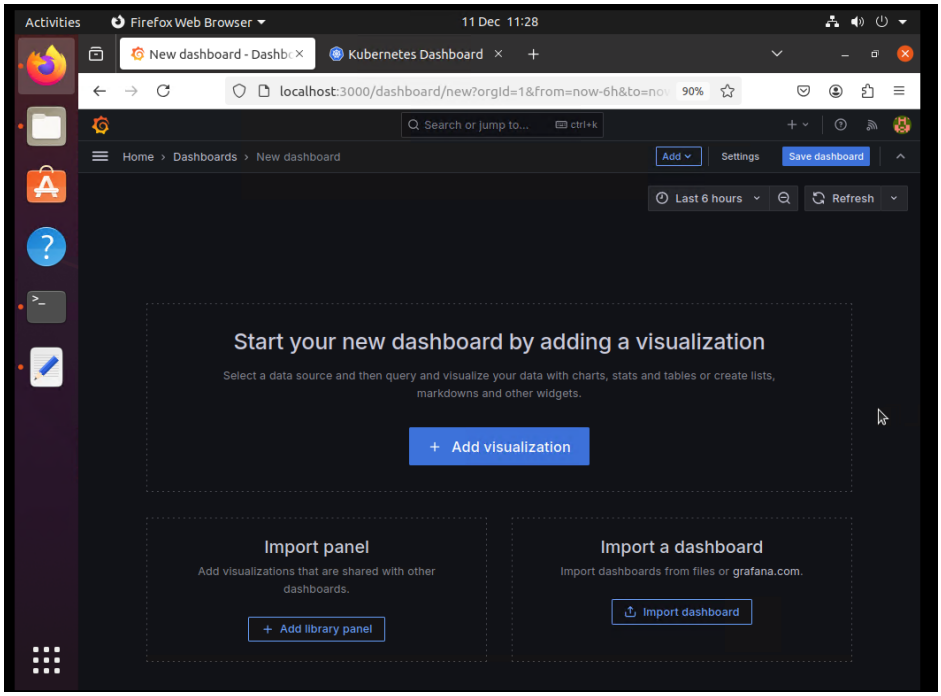
value: "10"

Sets the value of FREQUENCY to 10 requests per second.

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

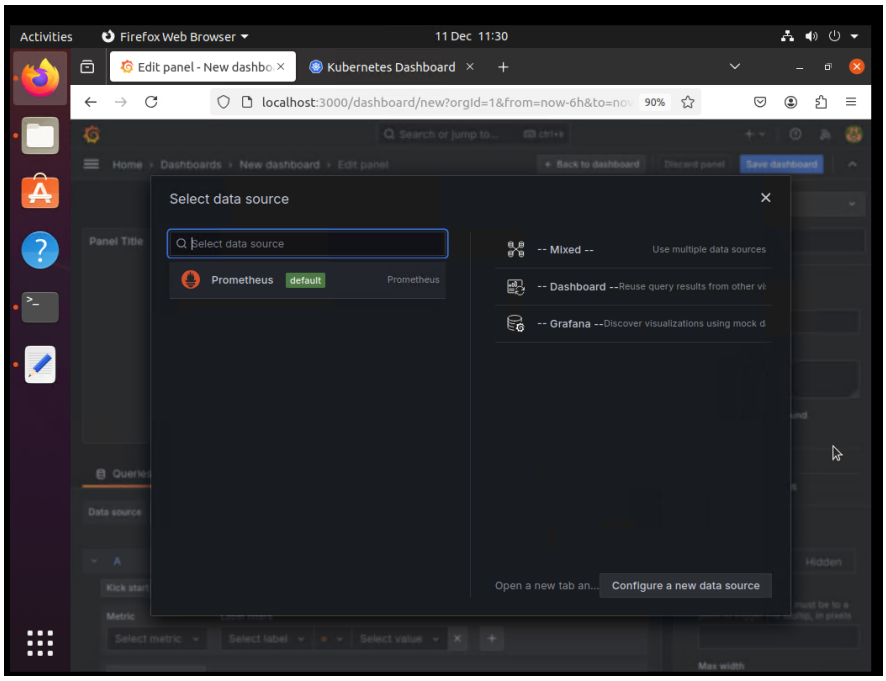
Step 1: Create a New Dashboard

- Navigate to the Grafana home page.
- Locate the "+" symbol in the top-right corner of the page.
- Click on the "+" symbol, and from the dropdown, select "New Dashboard".
- A new screen will appear where you can configure your dashboard, as shown in the screenshot.



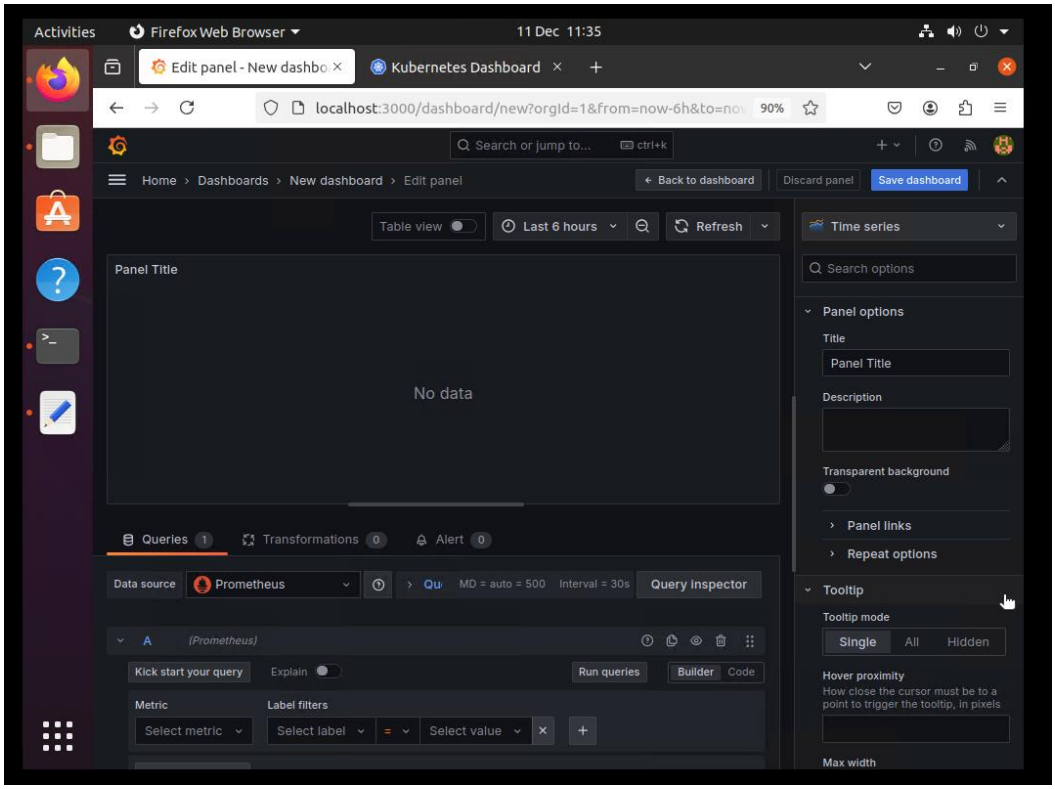
Step 2: Add a Visualization

- On the new dashboard page, click on the "Add Visualization" button.
- This action opens the data source selection page.
- From the available options, choose "Prometheus" as the data source.



Step 3: Configure the Panel

- Once you select Prometheus, you will be redirected to the panel configuration screen.
- In this screen:
    - You can add metrics and specify queries.
    - Customize the visualization type (e.g., graph, time series).
- Enter the relevant metrics and labels based on your YAML file for the javabenchmarkapp.



Step 4 :

Enter Metric Details:

In the Metric field, enter: container_cpu_usage_seconds_total

In the Label filters section:

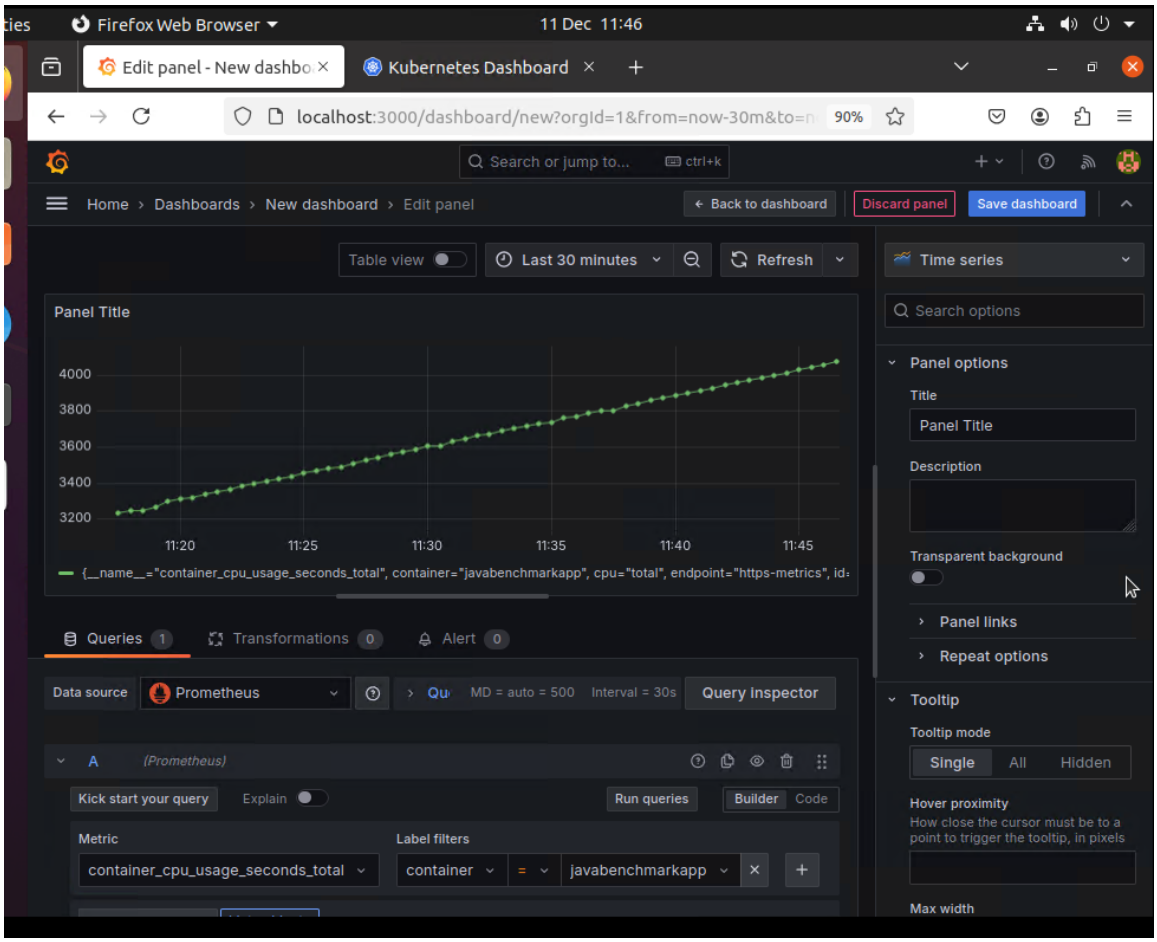
- Set Label to: container
- Set Value to: javabenchmarkapp

Run the Query:

- Click "Run queries" to execute the enerate the graph.

Expected Outcome

- The graph will plot the CPU usage of the javabenchmarkapp container over time.
- You'll see the time on the X-axis and CPU usage (in seconds) on the Y-axis, similar to the attached screenshot.



Enter Query Details, in the Metric field, enter: container_memory_usage_bytes

In the Label filters section:

- Label: container
- Value: javabenchmarkapp

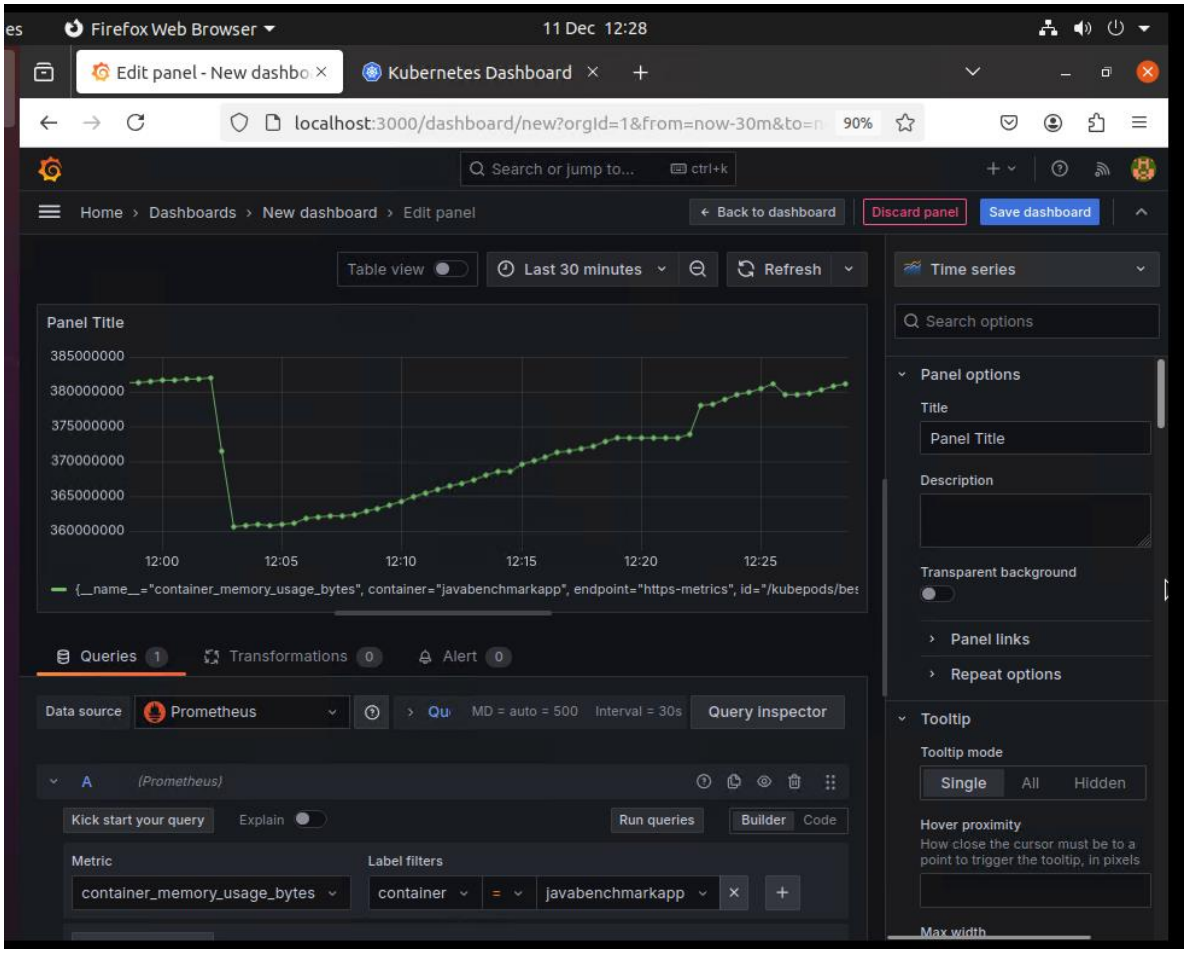
Run and Save:

2. Click "Run queries" to verify the graph is working as expected.

Expected Outcome

- A graph plotting the container_memory_usage_bytes metric for the javabenchmarkapp container.

- The X-axis will represent time, while the Y-axis will show memory usage in bytes, MB, or GB, depending on the unit you set.



Analysis:

The analysis of the coursework highlights the effective deployment, monitoring, and benchmarking of a containerized web application using Kubernetes, Prometheus, and Grafana. The deployment demonstrated the ability to manage containerized workloads through YAML configurations and Kubernetes Services, ensuring scalability and accessibility. Monitoring tasks successfully visualized real-time CPU and memory usage metrics, providing insights into the application's performance and resource utilization. The benchmarking phase revealed the application’s behavior under load, showcasing trends in request handling, failures, and resource consumption. The results indicate that the application performed reliably within the defined parameters, and the use of monitoring tools allowed for identifying bottlenecks and optimizing resource allocation. Overall, the coursework demonstrates a practical understanding of cloud-native application management and highlights the importance of observability in ensuring system reliability and scalability.

CONCLUSION:

This assignment will successfully demonstrate how to deploy and manage containerized applications on Kubernetes. The assignment shows key aspects of container orchestration and application benchmarking, such as deploying and accessing the Kubernetes dashboard, setting up monitoring tools like Prometheus and Grafana, and creating a custom load generator. Real-time monitoring of application metrics, such as CPU and memory usage, using Grafana, further demonstrates the effectiveness of Kubernetes in resource management. These results highlight the importance of performance testing and monitoring to be done for scalable and reliable cloud-based applications.