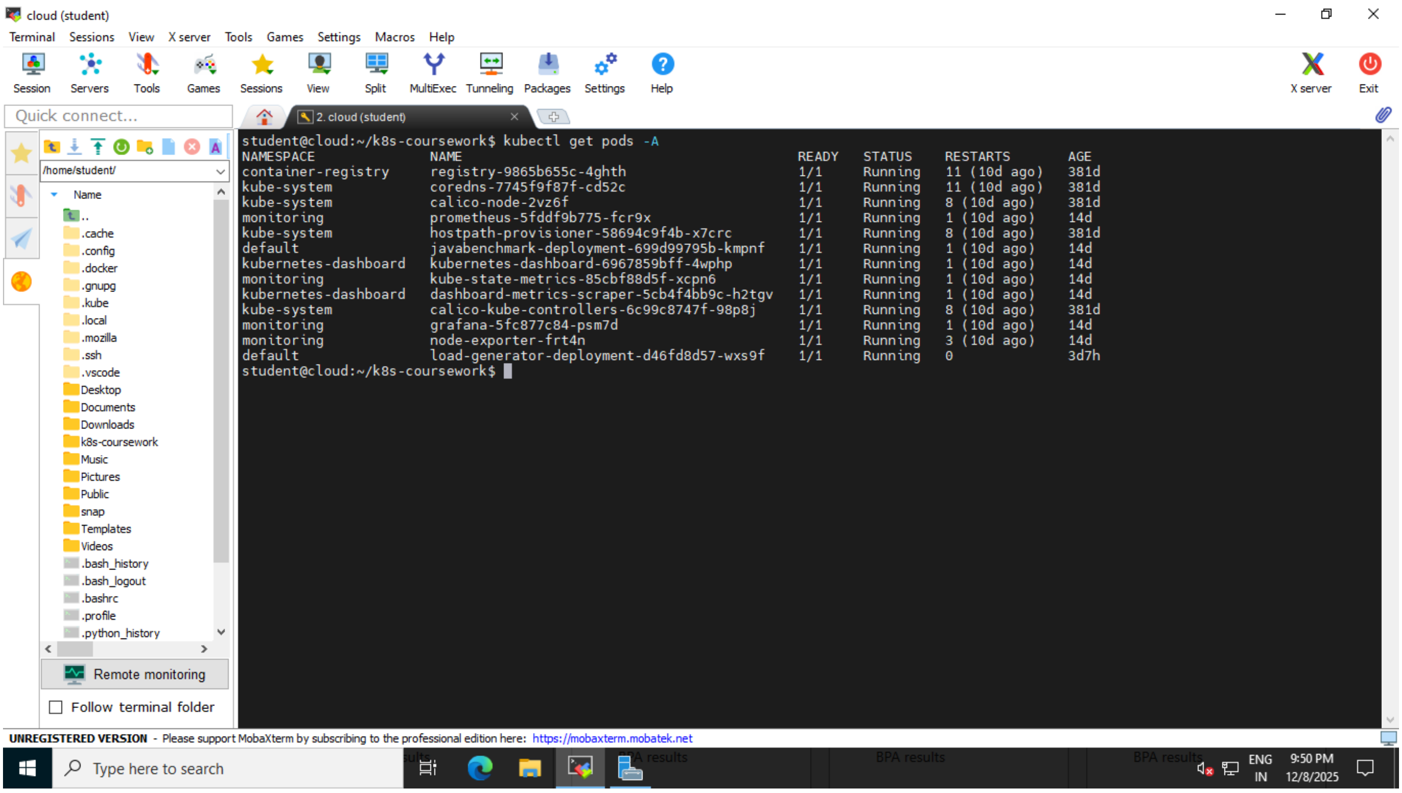
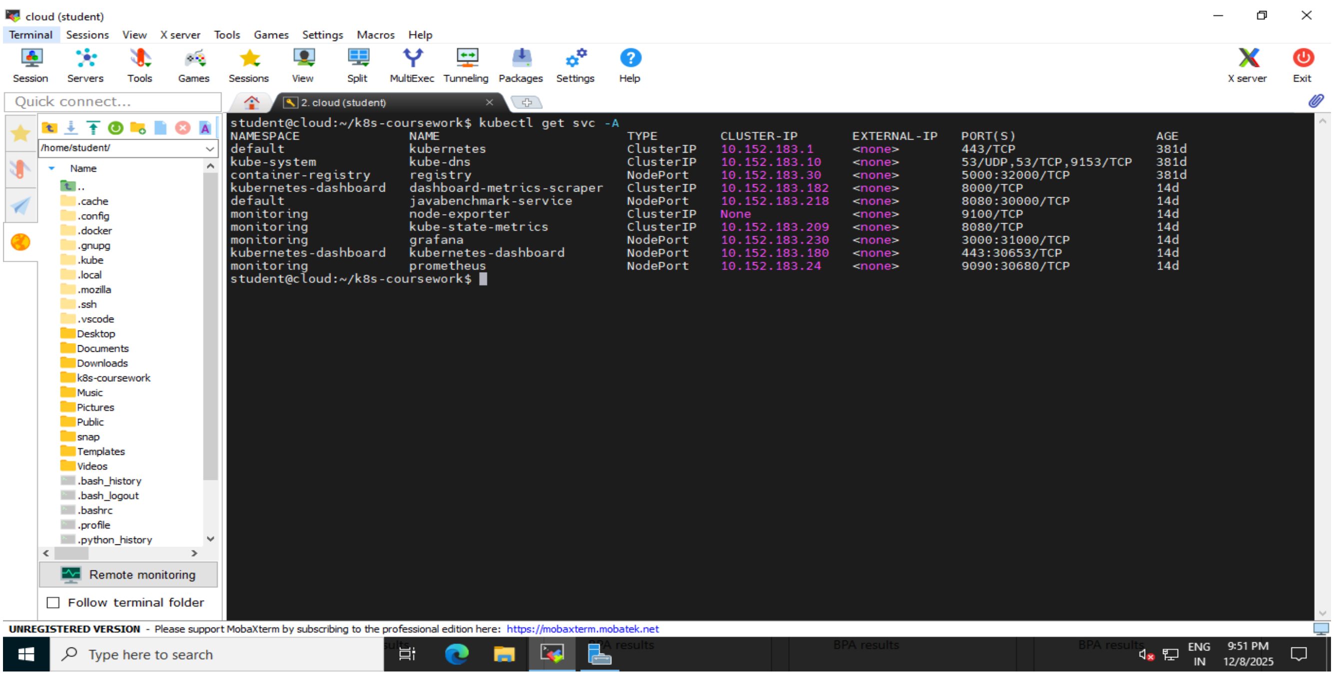
CSC8110-Cloud Computing

Student No: 250501717

Objective of the Coursework: The objective of this coursework is to gain practical experience in deploying and managing containerised applications using Kubernetes. It also aims to develop skills in building Docker images, generating application load, and monitoring system performance using Prometheus and Grafana.



The above screenshot shows all the pods.



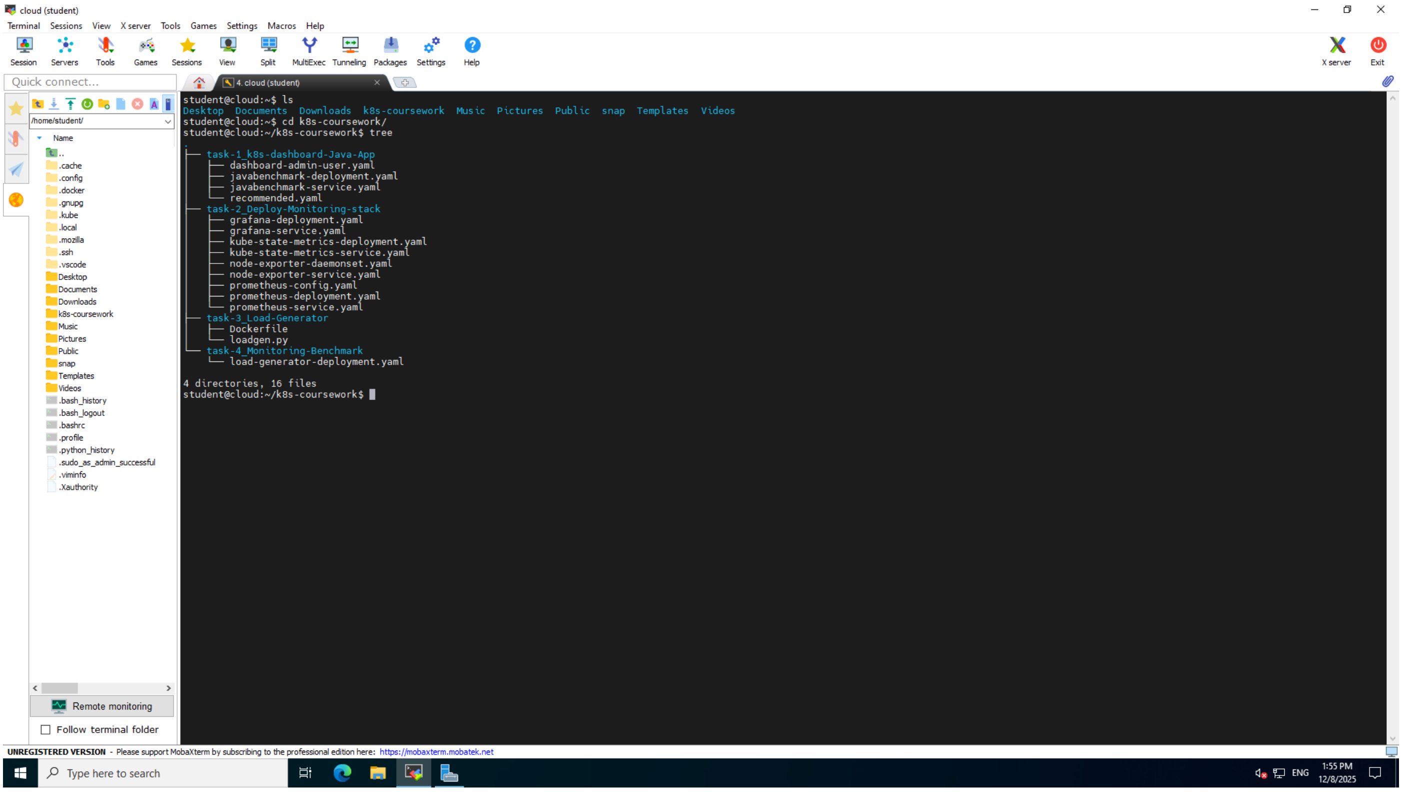
The above screenshot shows all the service pods.

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

Objective:

The objective is to deploy and access the Kubernetes Dashboard, deploy a Java-based web application inside the Kubernetes cluster, and expose the application using a NodePort service so that it could be accessed from the host machine.

Implementation:



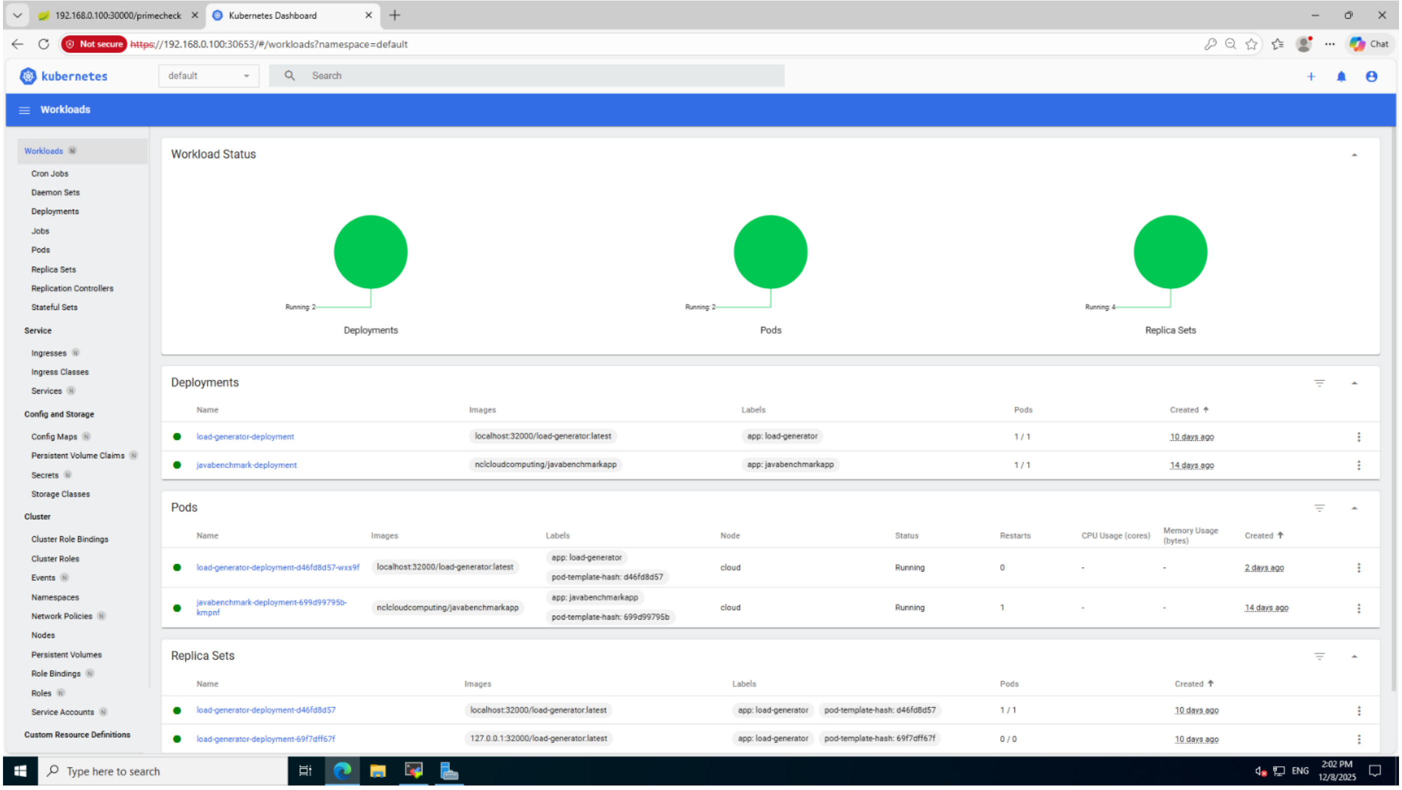
The above snippet shows the Organised Directory Structure of the coursework, where each task is separated into its own directory.

1.Deployement of the Kubernetes Dashboard:

A Kubernetes Dashboard has been set up with the aim of monitoring and managing the resources in a graphical form. I then went on to download a YAML file for a Kubernetes Dashboard from the official website and set it up in the cluster, which led to all necessary components such as a dashboard pod, services, and security rules being set up.

To provide secure access for using the dashboard, a special admin user has been created, and all permission has been granted using a ClusterRoleBinding. With this, all workloads in the cluster were accessible as well as controllable using the dashboard.

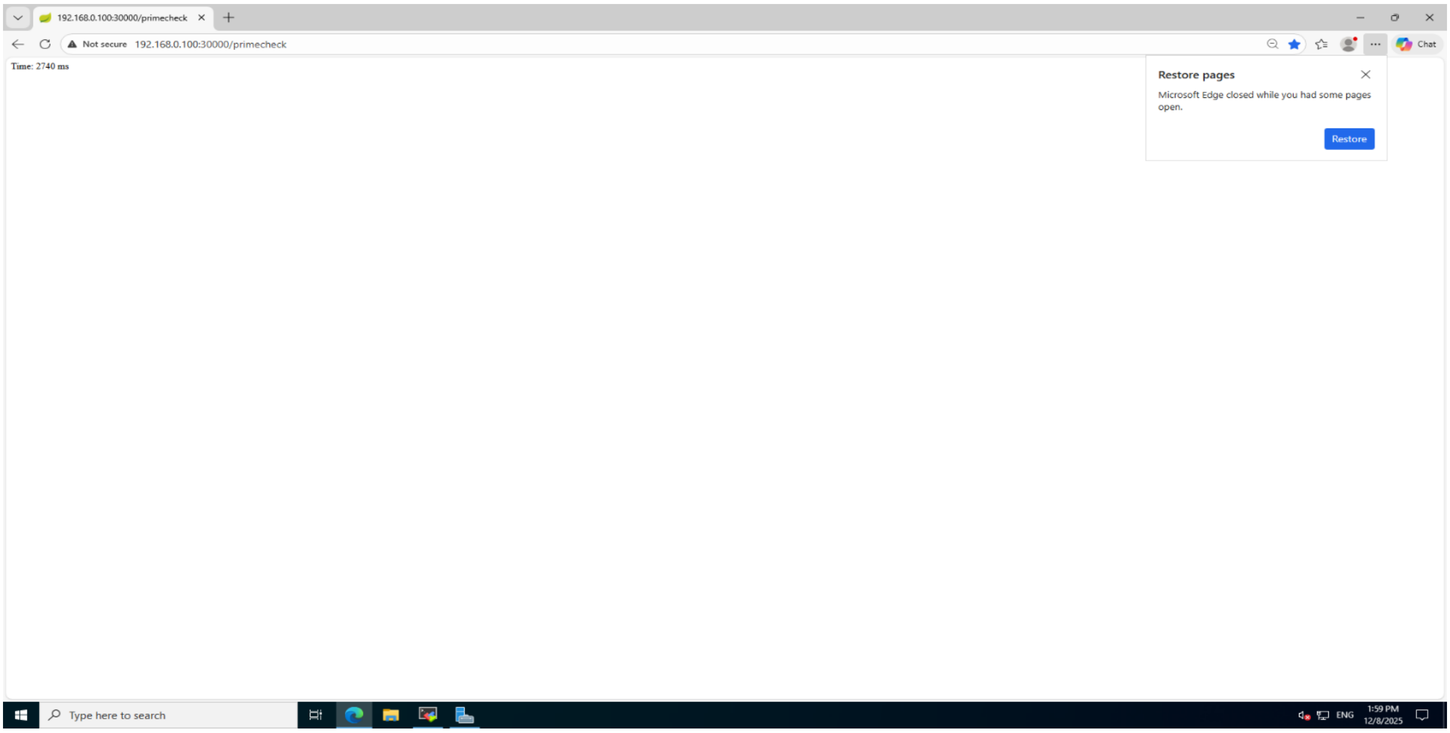
Then, the Dashboard service was exported using NodePort to be able to access it from a web browser with help of the IP address and port number of the virtual machine. Next, login to dashboard with a token which has been created.

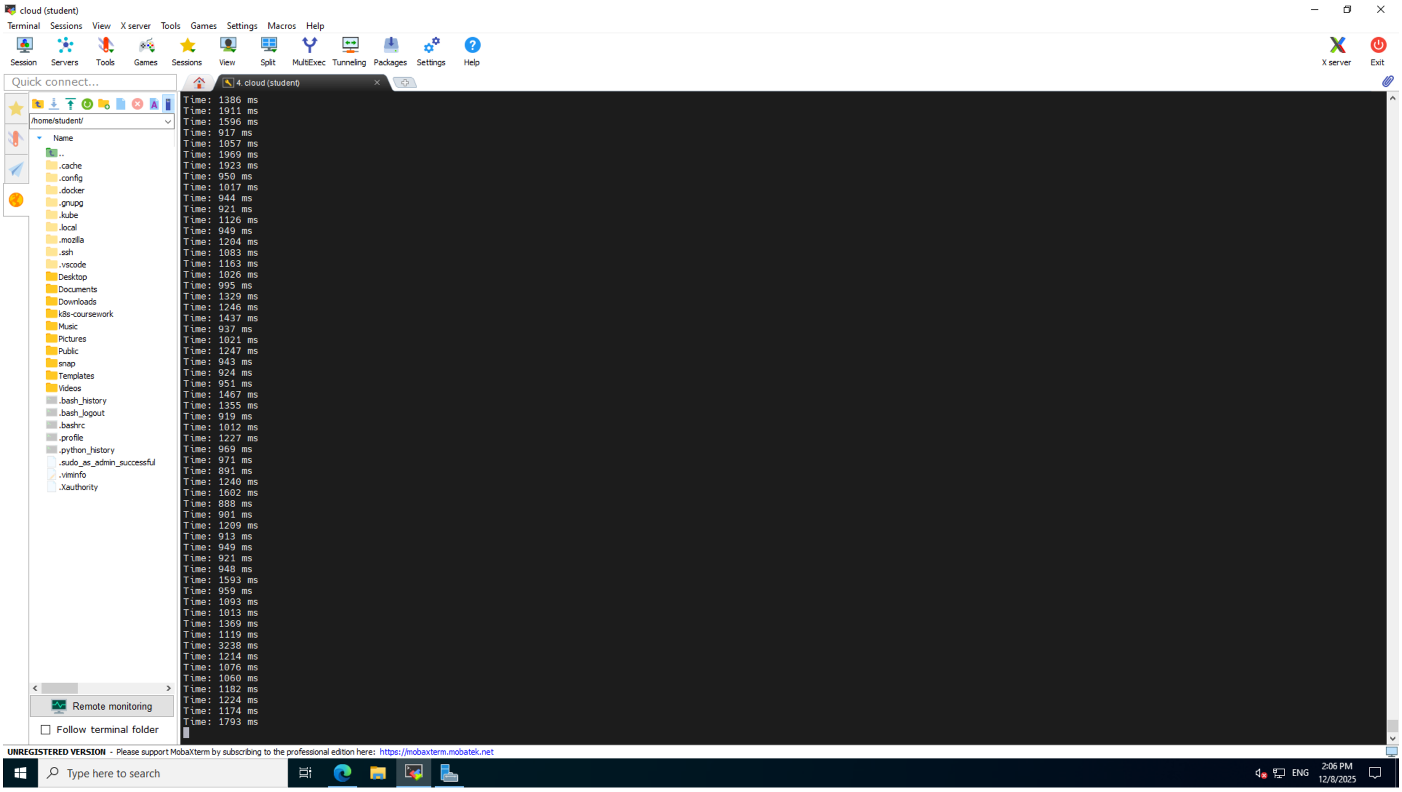


The above snippet shows that the Kubernetes dashboard was deployed successfully and shows all the required things.

2. Deployment of the Java Benchmark Application:

After deploying the kubernetes dashboard, I deployed the Java benchmark application into the cluster. I used a deployment file to create a running pod for the application. This application checks large prime numbers and creates CPU load. Then, I created a service to expose this application outside the cluster. I used the NodePort type so that I could open the application in the browser using the VM IP and port number.





The above screenshot shows the live execution logs. The response time for each request is displayed in milliseconds.

Task 2: Deploy the monitoring stack of Kubernetes

Objective :

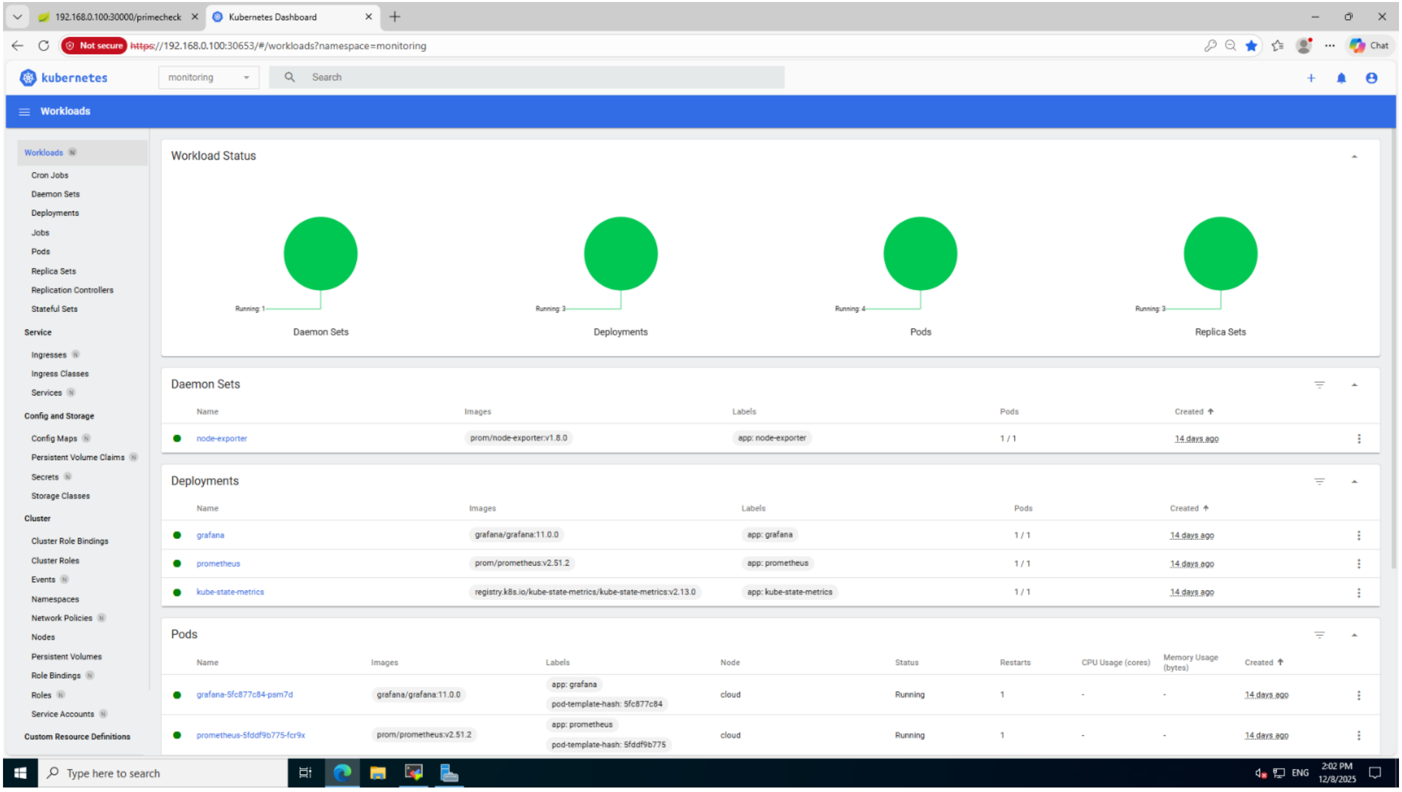
The objective of this task is to deploy a complete monitoring system in the Kubernetes using Prometheus, kube-sate metrics, Grafana and node exporter. This is used to collect, store and visualize the performance and health of the cluster.

Implementation:

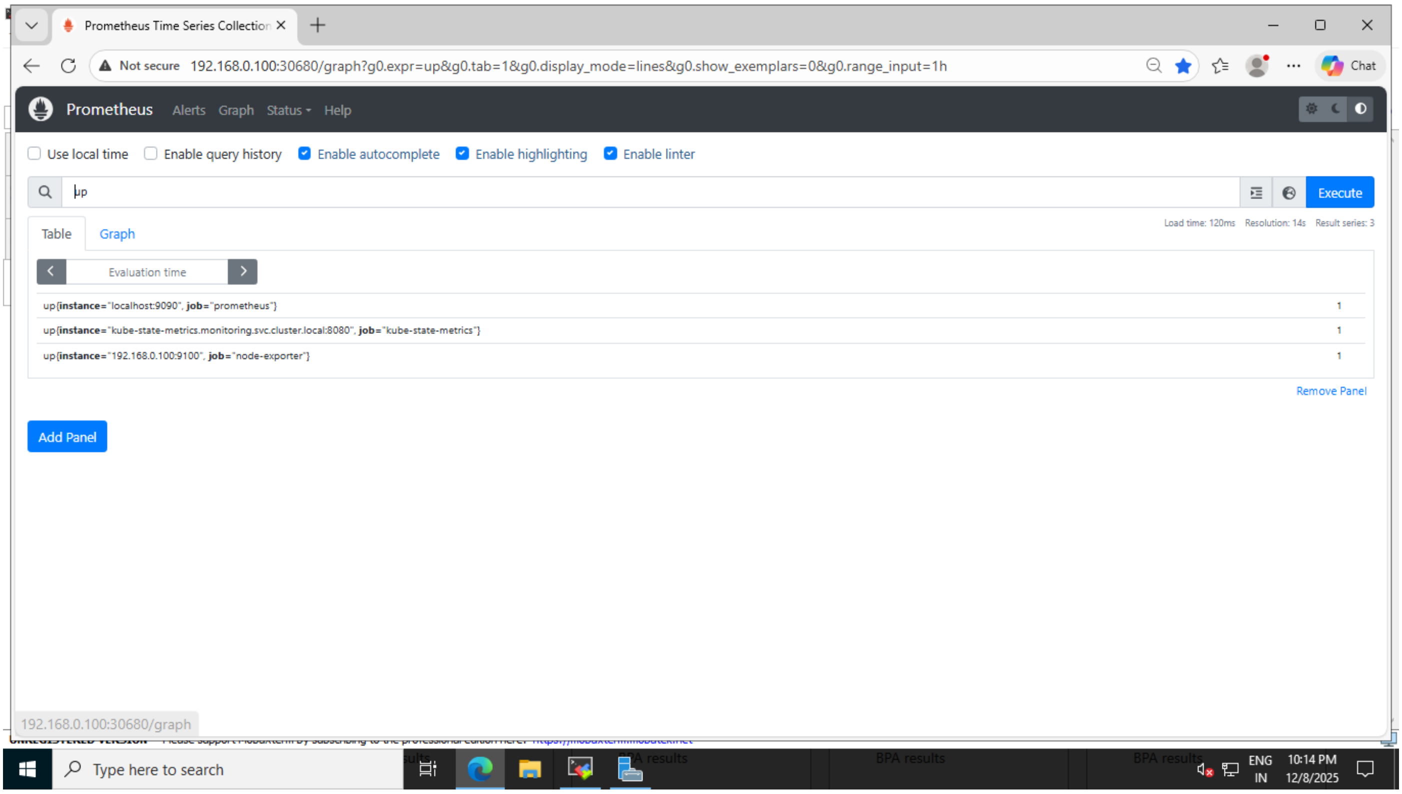
In this first I created a separate namespace named monitoring to keep all monitoring components in one place and also organized. The I deployed Node Exporter as a DaemonSet so that it will collect all the CPU and Memory data.

```
student@cloud:~/k8s-coursework$ kubectl get pods -n monitoring
NAME                                 READY   STATUS    RESTARTS      AGE
prometheus-5fddf9b775-fcr9x          1/1     Running   1 (10d ago)   14d
kube-state-metrics-85cbf88d5f-xcpn6  1/1     Running   1 (10d ago)   14d
grafana-5fc877c84-psm7d              1/1     Running   1 (10d ago)   14d
node-exporter-frt4n                  1/1     Running   3 (10d ago)   14d
student@cloud:~/k8s-coursework$
```

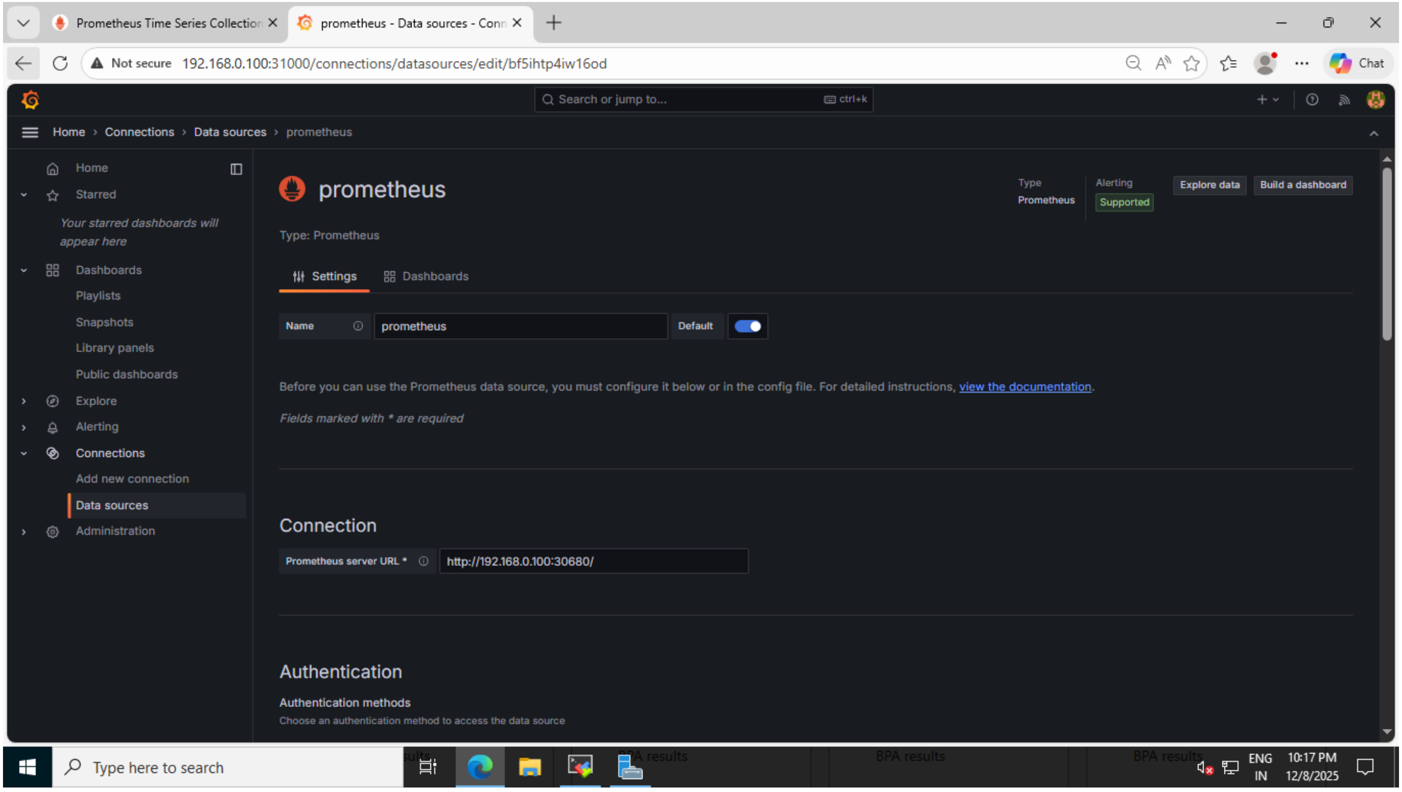
After that I deployed kube-state-metrics to collect all the information about pods and deployment. Then I deployed Prometheus to collect all the metrics from Node Exporter and kube-state-metrics. After deploying all monitoring components, Grafana was connected to Prometheus as its data source. Because of which Grafana was able read and display the metrics collected by Prometheus. Once the connection was successful, live system data such as CPU and memory usage was visible on the Grafana interface.



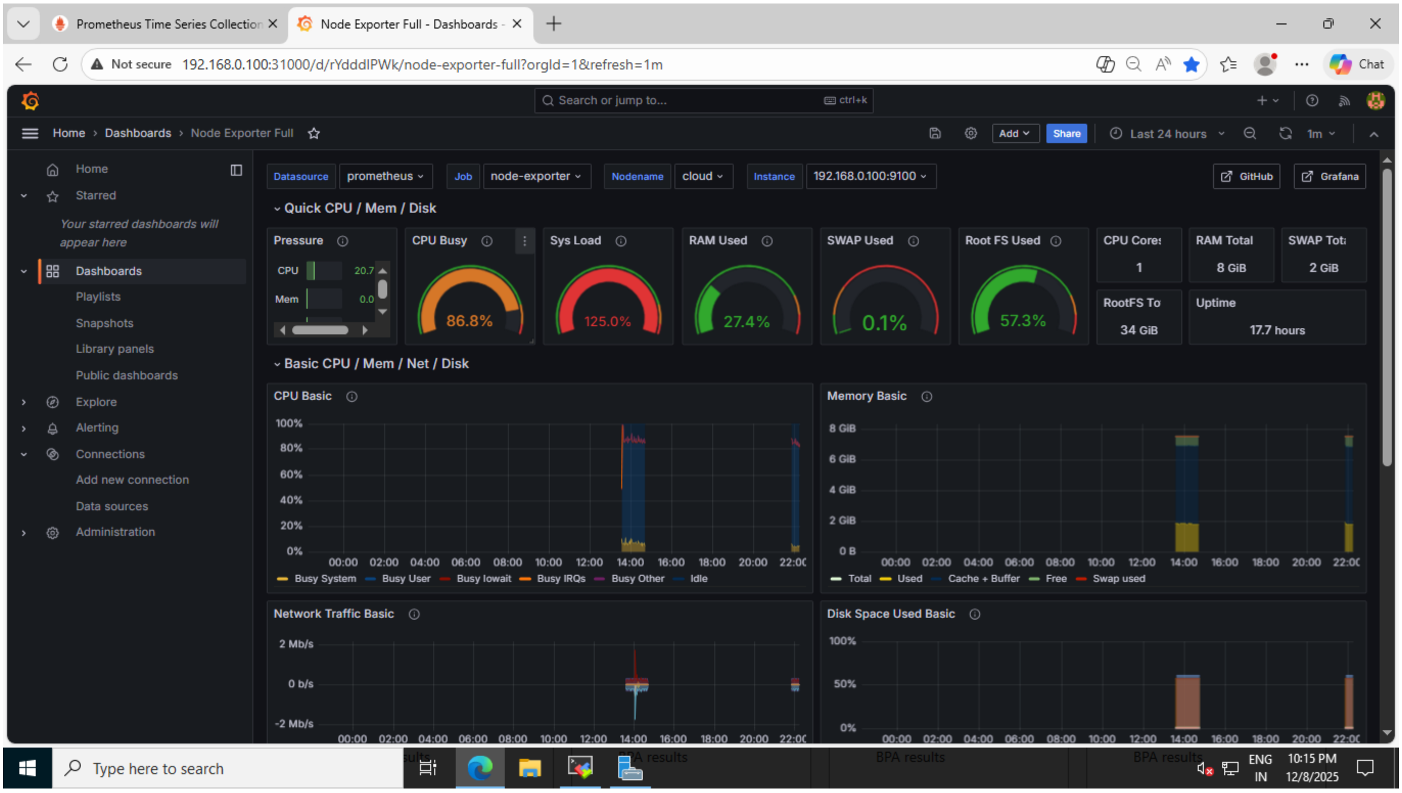
The above screenshot shows the monitoring workloads running inside the kubernetes dashboard. It consists of Node Exporter as a DaemonSet and Grafana, Prometheus, and kube-state-metrics as Deployments. All pods are in the running state, which confirms that the monitoring stack is working properly.



The above screenshot shows the Prometheus web interface running in the browser. The **up** query confirms Prometheus, Node Exporter, and kube-state-metrics are all active and reachable. This means Prometheus is successfully collecting data from all the monitoring component.



The above screenshot shows Prometheus added as a data source in Grafana. The Prometheus server URL is configured so Grafana can fetch monitoring data from it. The connection is successful that means Grafana can now display metrics from Prometheus.



The above screenshot shows the Node Exporter dashboard in Grafana. It displays live system metrics such as CPU usage, memory usage, disk usage, and network traffic of the node. The graphs change in real time as the load on the system increases or decreases.

Task 3: Load Generator

Objective:

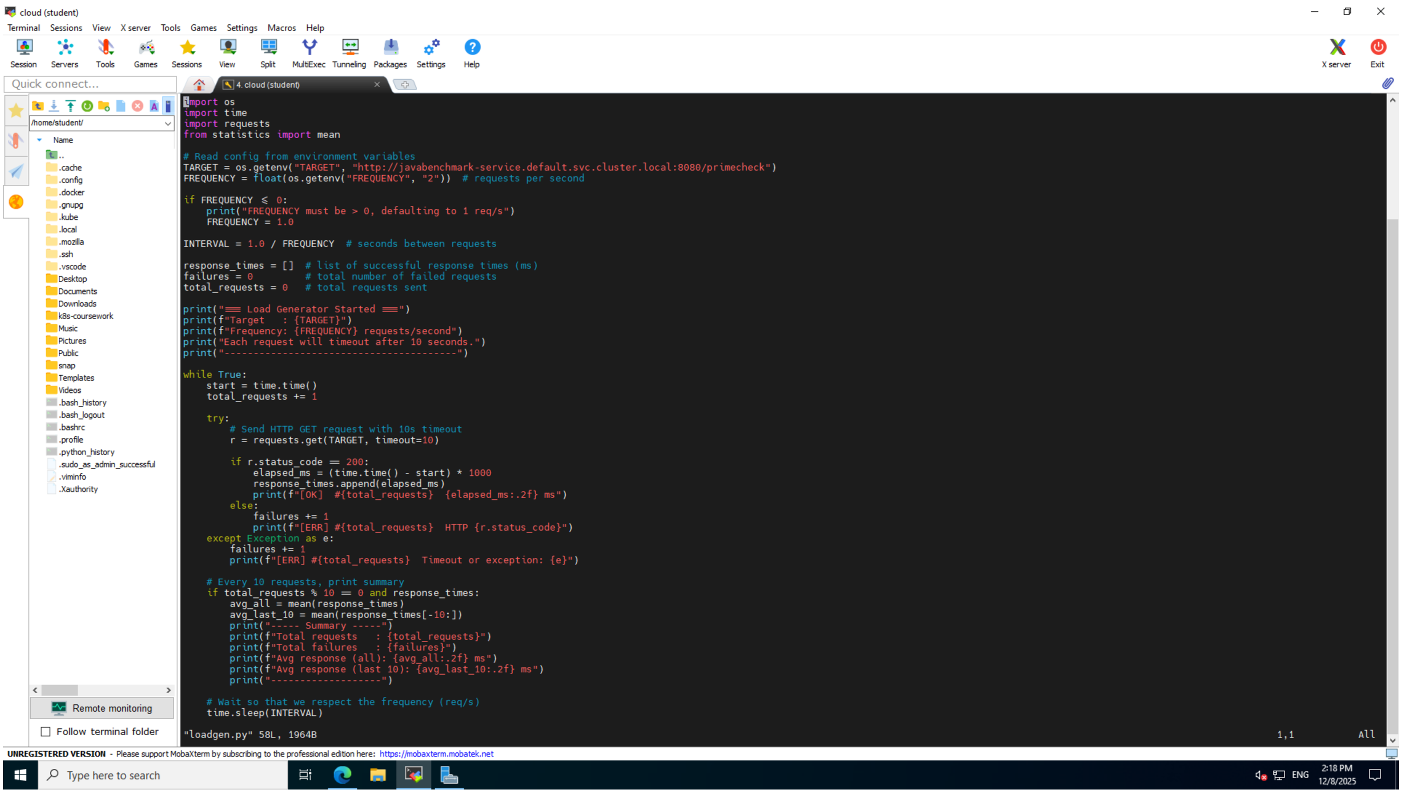
The objective of this task is to create a load generator using Python and Docker. This load generator continuously sends requests to the web application to create CPU and memory load. This helps us test how the application and monitoring system behave under load.

Implementation:

1. Python Load Generator

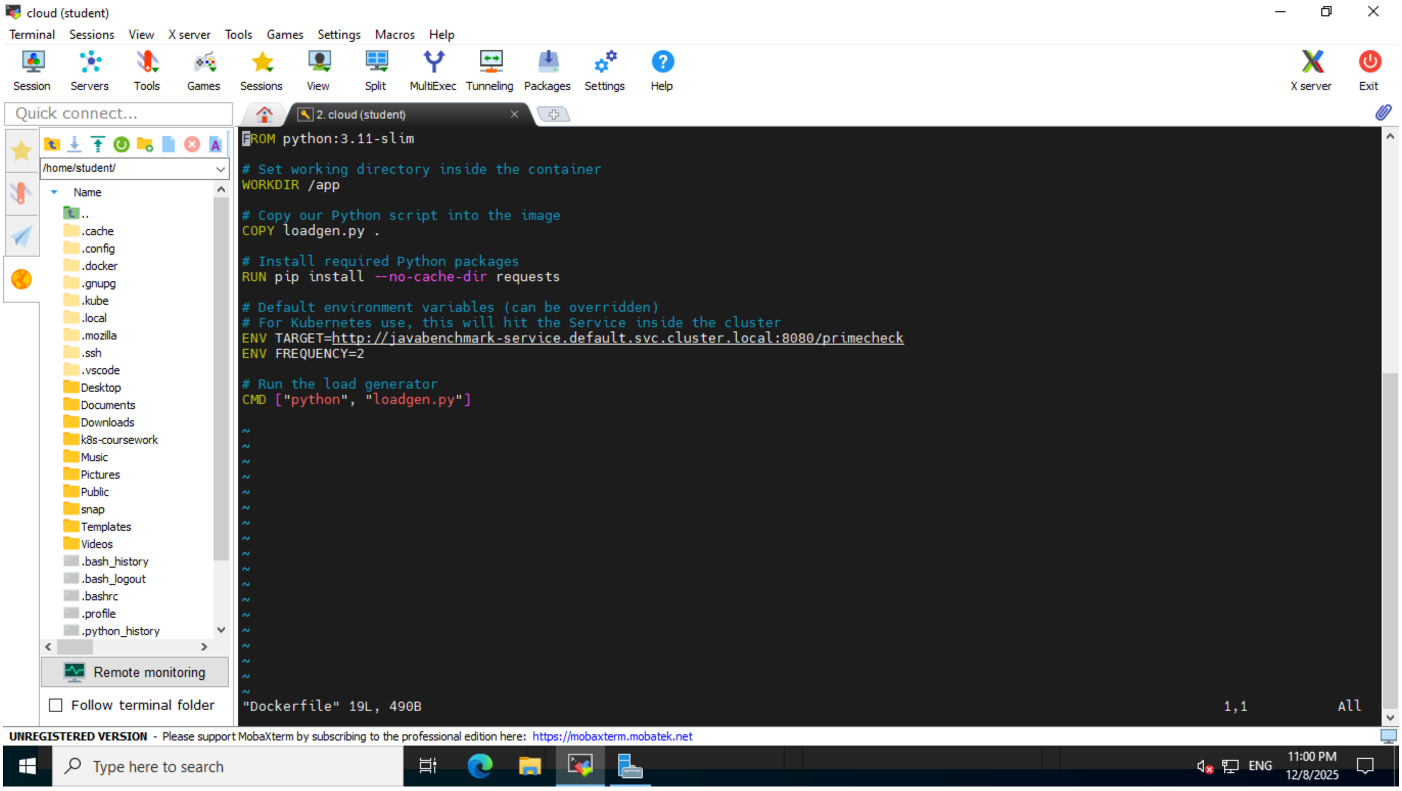
I wrote a Python script that works as a load generator for the Java benchmark application. The script reads two values from environment variables: TARGET which is the URL of the /primecheck endpoint and FREQUENCY which control how many requests are sent per second. Inside an infinite loop, the script sends HTTP requests to the target URL measures how long each request takes, and checks whether it was successful.

For every request the script records the response time in milliseconds and keeps a running count of the total number of requests and the number of failures. If a request takes more than 10 seconds or returns an error it is counted as a failure. The script prints live statistics to the terminal including average response time and total failures to show how the system is behaving while the load is running.

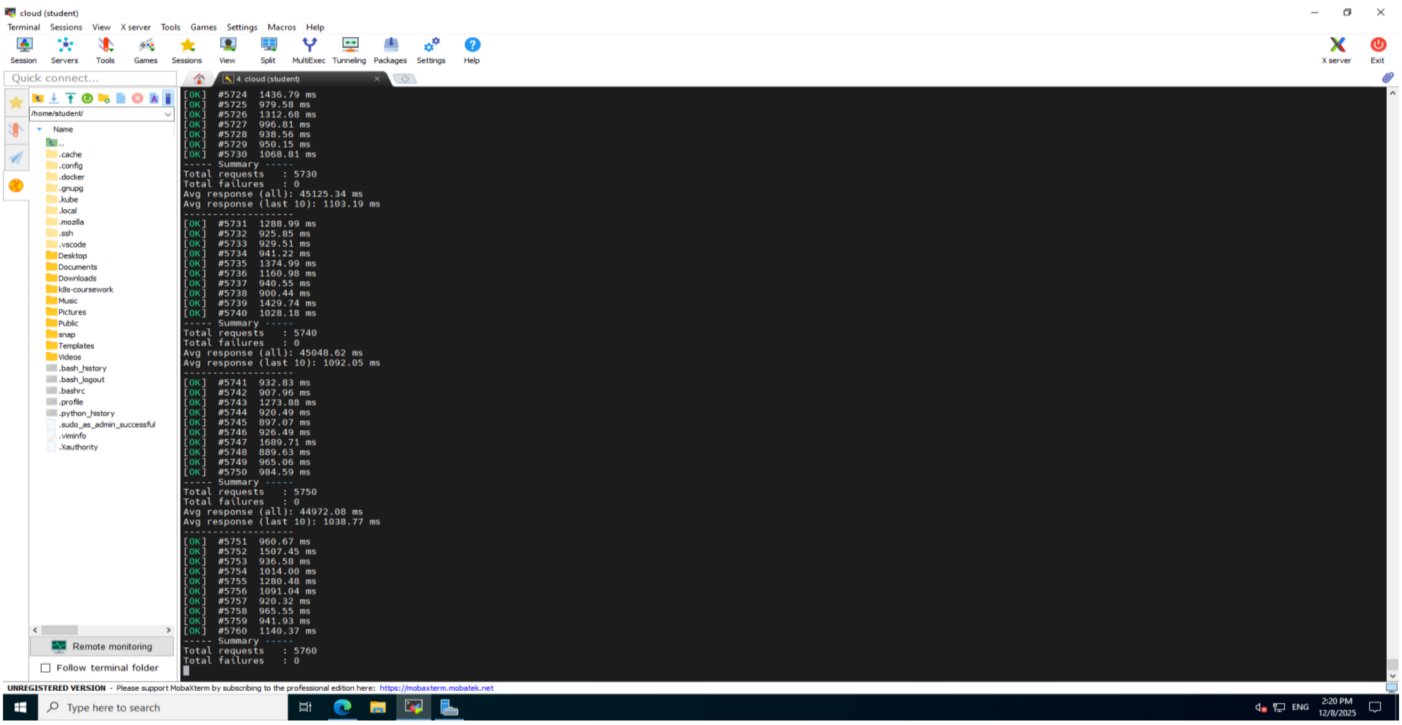


This Python script is used as a load generator for the Java benchmark application. It continuously sends HTTP requests to the primecheck endpoint and measures response time and failures. The load frequency and target URL is controlled using environment variables.

To run the load generator inside Kubernetes I created a docker file. This docker file starts with a lightweight python base image, copies the python script into the container and then install the required libraries(requests). Then sets the default command to run the python script when the container starts. I used the Docker file to convert the script into a Docker image.



Using this Docker image I packaged the load generator and all its dependencies into a single container. This image was then pushed to the local registry so that Kubernetes could pull it and create pod.



These are the live logs of the load generator pod running the Python script. It shows continuous requests being sent to the application along with their response times and success status. This load is later used in task 4 to observe performance change in Prometheus and Grafana.

Task 4: Monitoring Benchmarking Results.

Objective:

The objective is to monitor the performance of the system while load is running on the application. Using Prometheus and Grafana the CPU, memory and system usage are observed in real time. This shows how the load affects the Kubernetes cluster.

Implementation:

In this task, a load generator was deployed in Kubernetes to create continuous traffic on the Java benchmark application. After starting the load generator pod, the system load increased, and this change was visible in Grafana through CPU, memory, and network graphs. Prometheus collected the metrics from node-exporter and kube-state-metrics, and Grafana displayed these values in real time. Then we made the load to stopped and start again to clearly observe the difference in system behaviour.

```
student@cloud:~/k8s-coursework$ kubectl get pods -n default
NAME                                      READY   STATUS    RESTARTS      AGE
javabenchmark-deployment-699d99795b-kmpnf 1/1     Running   1 (10d ago)   14d
load-generator-deployment-d46fd8d57-wxs9f 1/1     Running   0             3d9h
student@cloud:~/k8s-coursework$
```

The above screenshot shows that both the Java benchmark application pod and the load generator pod are running successfully in the default namespace. The load generator is actively sending requests to the application to create CPU load.

```
student@cloud:~/k8s-coursework$ docker images
REPOSITORY                        TAG      IMAGE ID       CREATED       SIZE
127.0.0.1:32000/load-generator    latest   6905aa0d8655   11 days ago   138MB
load-generator                    latest   6905aa0d8655   11 days ago   138MB
localhost:32000/load-generator    latest   6905aa0d8655   11 days ago   138MB
student@cloud:~/k8s-coursework$
```

The above screenshot shows the docker image of the load generator that was built and stored locally. The same image is also tagged and pushed to the local container registry for Kubernetes to use. This confirms that the load generator image is ready for deployment.

The above graph shows the memory usage of the Java Benchmark app over time. When the load generator is running the memory usage increases and when the load is reduced, the memory usage becomes stable again. This confirms that the application is under real workload.

The above graph shows the CPU usage of the Java Benchmark application over time. When the load generator is active, the CPU usage increases, and when the load is stopped, the CPU usage decreases.

Conclusion:

In this coursework, a Java benchmark application was deployed on a Kubernetes cluster and then monitored using a comprehensive monitoring stack. Prometheus, Node Exporter, kube-state-metrics, and Grafana were employed for metric gathering and visualization. A load generator helped simulate a practical CPU and memory load, which was then observable in Grafana. This coursework has been valuable as it gave insight into how Kubernetes apps are deployed, monitored, and load-tested in a practical setup.