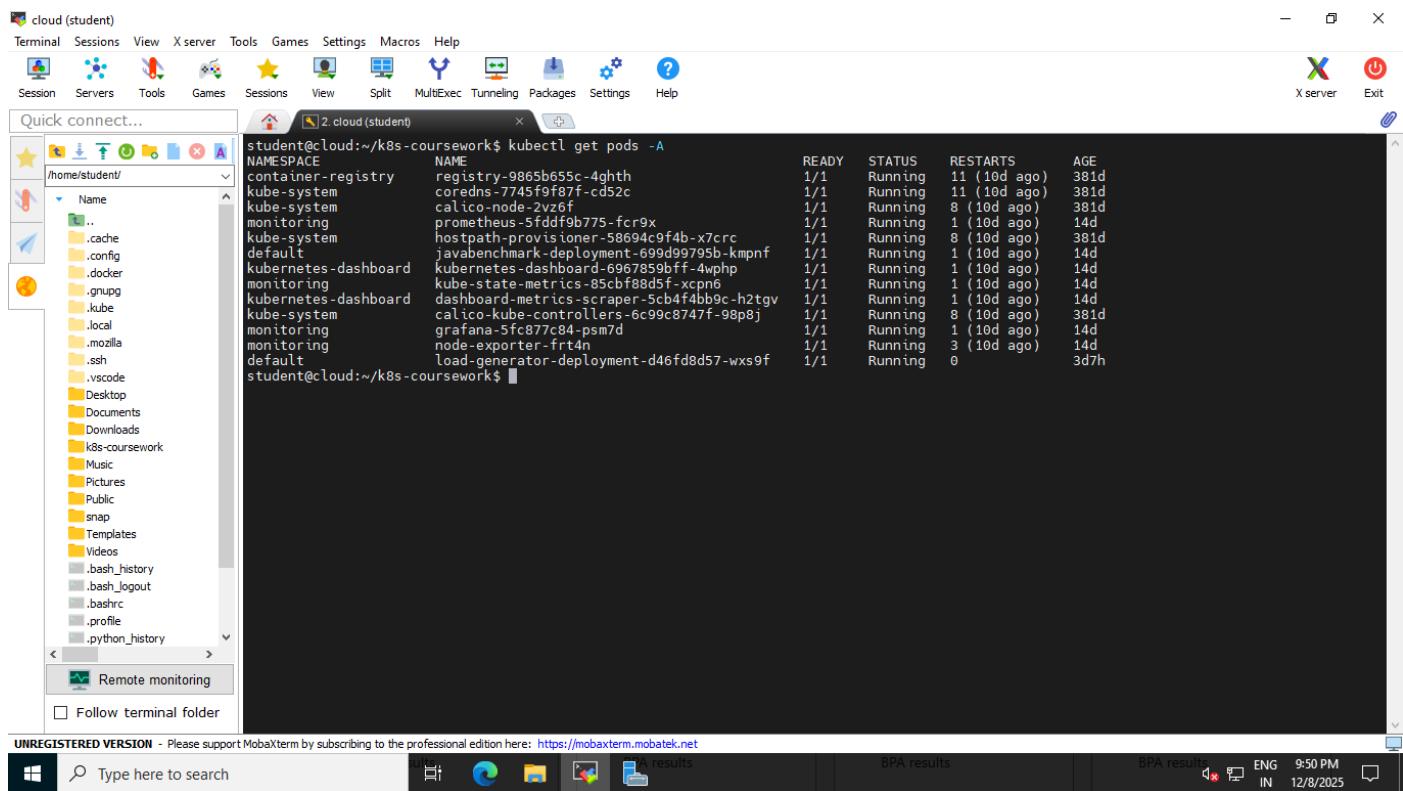


CSC8110-Cloud Computing

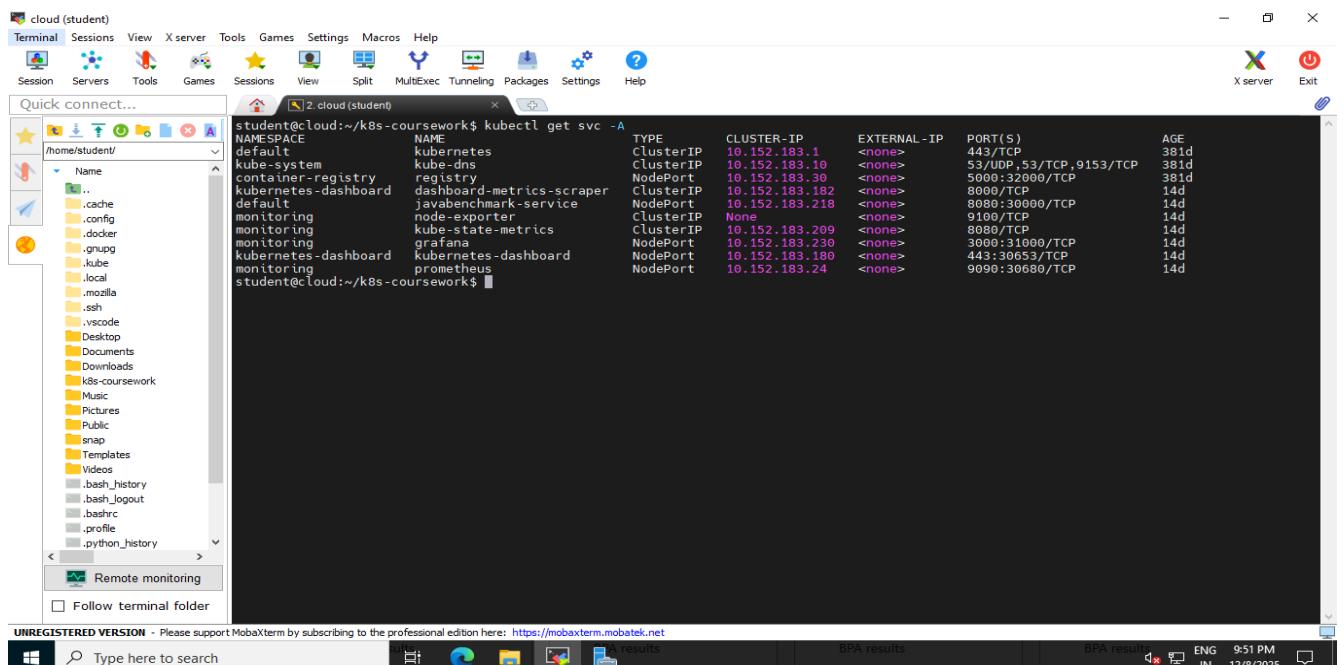
Student No: 250501717

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.



```
student@cloud:~/k8s-coursework$ kubectl get pods -A
  NAME                               READY   STATUS    RESTARTS   AGE
container-registry      registry-9865b655c-4gtht   1/1     Running   11 (10d ago)   381d
coredns-7745f9f87f-cd52c   coredns-7745f9f87f-cd52c   1/1     Running   11 (10d ago)   381d
calico-node-2vz6f          calico-node-2vz6f   1/1     Running   8 (10d ago)   381d
prometheus-5fddfb9b775-fcr9x  prometheus-5fddfb9b775-fcr9x   1/1     Running   1 (10d ago)   14d
hostpath-provisioner-58694c9f4b-x7crc  hostpath-provisioner-58694c9f4b-x7crc   1/1     Running   8 (10d ago)   381d
javabenchmark-deployment-699d9795b-kmpnf  javabenchmark-deployment-699d9795b-kmpnf   1/1     Running   1 (10d ago)   14d
kubernetes-dashboard       kubernetes-dashboard-6967859bf-4wphp   1/1     Running   1 (10d ago)   14d
kube-state-metrics-85cbf88d5f-xcpn6   kube-state-metrics-85cbf88d5f-xcpn6   1/1     Running   1 (10d ago)   14d
dashboard-metrics-scraper-5cb4f4bb9c-h2tgv  dashboard-metrics-scraper-5cb4f4bb9c-h2tgv   1/1     Running   1 (10d ago)   14d
calico-kube-controllers-6c99c8747f-98p8j  calico-kube-controllers-6c99c8747f-98p8j   1/1     Running   8 (10d ago)   381d
grafana-5fc877c84-psm7d   grafana-5fc877c84-psm7d   1/1     Running   1 (10d ago)   14d
node-exporter-frt4n        node-exporter-frt4n   1/1     Running   3 (10d ago)   14d
load-generator-deployment-d46fd8d57-wxs9f  load-generator-deployment-d46fd8d57-wxs9f   1/1     Running   0           3d7h
student@cloud:~/k8s-coursework$
```

The above screenshot shows all the pods.



```
student@cloud:~/k8s-coursework$ kubectl get svc -A
  NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
default   LoadBalancer   10.152.183.1   <none>           443/TCP      381d
kube-dns  ClusterIP   10.152.183.10   <none>           53/UDP,53/TCP,9153/TCP   381d
coredns-7745f9f87f-cd52c   ClusterIP   10.152.183.30   <none>           5000:32000/TCP   381d
registry-9865b655c-4gtht   NodePort    10.152.183.30   <none>           8000/TCP      14d
dashboard-metrics-scraper-5cb4f4bb9c-h2tgv  ClusterIP   10.152.183.182  <none>           8080:30000/TCP   14d
javabenchmark-service      NodePort    10.152.183.218  <none>           8080:30000/TCP   14d
node-exporter-frt4n        ClusterIP   None           <none>           9100/TCP      14d
kube-state-metrics-85cbf88d5f-xcpn6   ClusterIP   10.152.183.209  <none>           8080/TCP      14d
grafana-5fc877c84-psm7d   NodePort    10.152.183.230  <none>           3000:31000/TCP   14d
kubernetes-dashboard       NodePort    10.152.183.180  <none>           443:30653/TCP   14d
prometheus-5fddfb9b775-fcr9x  NodePort    10.152.183.24   <none>           9090:30680/TCP   14d
student@cloud:~/k8s-coursework$
```

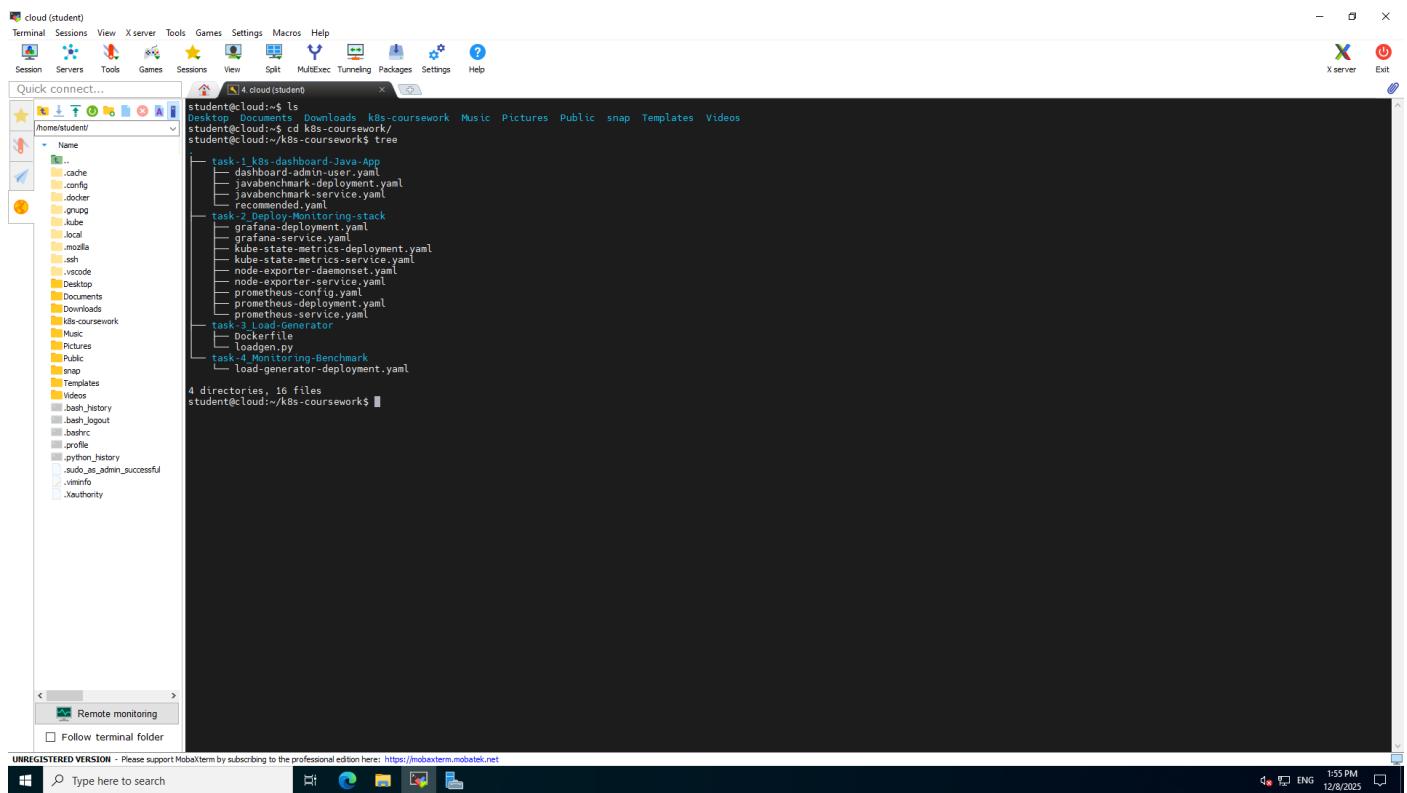
The above screenshot shows all the service pods.

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:



A screenshot of a terminal window titled "cloud (student)". The terminal shows the following command and its output:

```
student@cloud:~$ ls
student@cloud:~$ cd k8s-coursework/
student@cloud:~/k8s-coursework$ tree
.
+-- task-1-k8s-dashboard-Java-app
|   |-- dashboard-admin-user.yaml
|   |-- javabenchmark-deployment.yaml
|   |-- javabenchmark-service.yaml
|   `-- recommended.yaml
+
+-- task-2-k8s-monitoring-stack
|   |-- grafana-deployment.yaml
|   |-- grafana-service.yaml
|   |-- kube-state-metrics-deployment.yaml
|   |-- kube-state-metrics-service.yaml
|   |-- node-exporter-daemonset.yaml
|   |-- node-exporter-service.yaml
|   |-- prometheus-config.yaml
|   |-- prometheus-deployment.yaml
|   `-- prometheus-service.yaml
+
+-- task-3-load-generator
|   |-- Dockerfile
|   `-- loadgen.py
+
+-- task-4-Monitoring-Benchmark
|   |-- load-generator-deployment.yaml
+
4 directories, 16 files
student@cloud:~/k8s-coursework$
```

The terminal window is part of a desktop environment with various icons and a menu bar at the top. The status bar at the bottom shows "UNREGISTERED VERSION - Please support Mobaxterm by subscribing to the professional edition here: <https://mobaxterm.mobatek.net>".

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

1. Deployment 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 screenshot shows the Kubernetes Dashboard interface. On the left, a sidebar lists various resources: Cron Jobs, Daemon Sets, Deployments, Jobs, Pods, Replica Sets, Replication Controllers, Stateful Sets, Service, Ingresses, Ingress Classes, Services, Config and Storage, Config Maps, Persistent Volume Claims, Secrets, Storage Classes, Cluster, Cluster Role Bindings, Cluster Roles, Events, Namespaces, Network Policies, Nodes, Persistent Volumes, Role Bindings, Roles, Service Accounts, and Custom Resource Definitions. The main area displays the 'Workload Status' section with three green circular icons representing Deployments (Running 2), Pods (Running 2), and Replica Sets (Running 4). Below this, the 'Deployments' section lists two entries:

Name	Images	Labels	Pods	Created
load-generator-deployment	localhost:32000/load-generator:latest	app:load-generator	1/1	10 days ago
javabenchmark-deployment	nceloudcomputing/javabenchmarkapp	app:javabenchmarkapp	1/1	14 days ago

The 'Pods' section shows two running pods corresponding to the deployments:

Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created
load-generator-deployment-d46fd8d57-wxx9f	localhost:32000/load-generator:latest	app:load-generator pod-template-hash: d46fd8d57	cloud	Running	0	-	-	2 days ago
javabenchmark-deployment-699d99795b-kmpnf	nceloudcomputing/javabenchmarkapp	app:javabenchmarkapp pod-template-hash: 699d99795b	cloud	Running	1	-	-	14 days ago

The 'Replica Sets' section lists two entries:

Name	Images	Labels	Pods	Created
load-generator-deployment-d46fd8d57	localhost:32000/load-generator:latest	app:load-generator pod-template-hash: d46fd8d57	1/1	10 days ago
load-generator-deployment-69f7dff57f	127.0.0.1:32000/load-generator:latest	app:load-generator pod-template-hash: 69f7dff57f	0/0	10 days ago

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 screenshot shows a Microsoft Edge browser window. The address bar indicates the URL is 192.168.0.100:30000/primecheck. A restore pages dialog box is visible in the top right corner, stating "Restore pages" and "Microsoft Edge closed while you had some pages open." The main content area of the browser is blank, showing only the header and navigation bar.

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-state 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. Then 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-5fdf9b775-fcr9x   1/1     Running   1 (10d ago)  14d
kube-state-metrics-85cbf88df-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 screenshot shows the Kubernetes Dashboard interface. On the left, a sidebar lists various resources: Workloads (Cron Jobs, Daemon Sets, Deployments, Jobs, Pods, Replica Sets), Replication Controllers, Stateful Sets, Service (Ingresses, Ingress Classes, Services), Config and Storage (Config Maps, Persistent Volume Claims, Secrets, Storage Classes), Cluster (Cluster Role Bindings, Cluster Roles, Events, Namespaces), Network Policies, and Nodes. The main area is titled "Workload Status" and displays four green circular icons representing different workload types: Daemon Sets (1 running), Deployments (3 running), Pods (4 running), and Replica Sets (3 running). Below this, sections for Daemon Sets, Deployments, and Pods are shown as tables. The Deployments table lists three entries: grafana (grafana/grafana:11.0, 1/1 pods, 14 days ago), prometheus (prom/prometheus:v2.51.2, 1/1 pods, 14 days ago), and kube-state-metrics (registry.k8s.io/kube-state-metrics/kube-state-metrics:v2.13.0, 1/1 pods, 14 days ago). The Pods table lists two entries: grafana-5fc877c84-psm7d (grafana/grafana:11.0, pod-template-hash: 5fc877c84, cloud node, 1 restart, 14 days ago) and prometheus-5fdd9b775-fcr9x (prom/prometheus:v2.51.2, pod-template-hash: 5fdd9b775, cloud node, 1 restart, 14 days ago).

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 screenshot shows the Prometheus web interface. At the top, the URL is 192.168.0.100:30680/graph. The interface includes a search bar, a query editor with a dropdown menu, and several configuration checkboxes. The main area displays a table of metrics results. The table has columns for metric name, value, and status. The results show three active instances: up{instance="localhost:9090", job="prometheus"} (value 1), up{instance="kube-state-metrics.monitoring.svc.cluster.local:8080", job="kube-state-metrics"} (value 1), and up{instance="192.168.0.100:9100", job="node-exporter"} (value 1). The bottom of the screen shows the Windows taskbar with the browser icon and system status.

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 screenshot shows the Grafana interface for managing data sources. On the left, a sidebar navigation includes Home, Starred, Dashboards, Snapshots, Library panels, Public dashboards, Explore, Alerting, Connections, Data sources (which is selected), and Administration. The main content area is titled "prometheus" and is identified as a "Type: Prometheus" source. It has tabs for Settings (selected) and Dashboards. The "Name" field is set to "prometheus", and the "Default" toggle is on. A note at the top states: "Before you can use the Prometheus data source, you must configure it below or in the config file. For detailed instructions, [view the documentation](#)". Below this, there are sections for Connection (Prometheus server URL: http://192.168.0.100:30680) and Authentication (Authentication methods: Choose an authentication method to access the data source). At the bottom of the main content area, there is a standard Windows taskbar with icons for File Explorer, Task View, Start, and Search.

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 screenshot shows the "Node Exporter Full - Dashboards" page in Grafana. The top navigation bar includes Home, Data sources (selected), Dashboards, Explore, Alerting, Connections, Add new connection, Data sources, and Administration. The main content area displays a dashboard titled "Node Exporter Full". It features several data visualization components: a "Quick CPU / Mem / Disk" section with five gauge charts (Pressure, CPU Busy, Sys Load, RAM Used, SWAP Used) and three summary cards (CPU Cores: 1, RAM Total: 8 GiB, SWAP Total: 2 GiB); a "Basic CPU / Mem / Net / Disk" section with two line graphs (CPU Basic and Memory Basic) showing usage over time; a "Network Traffic Basic" section with a line graph showing network traffic in Mb/s; and a "Disk Space Used Basic" section with a line graph showing disk space usage in percent. The bottom of the screen shows a Windows taskbar with icons for File Explorer, Task View, Start, and Search.

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:

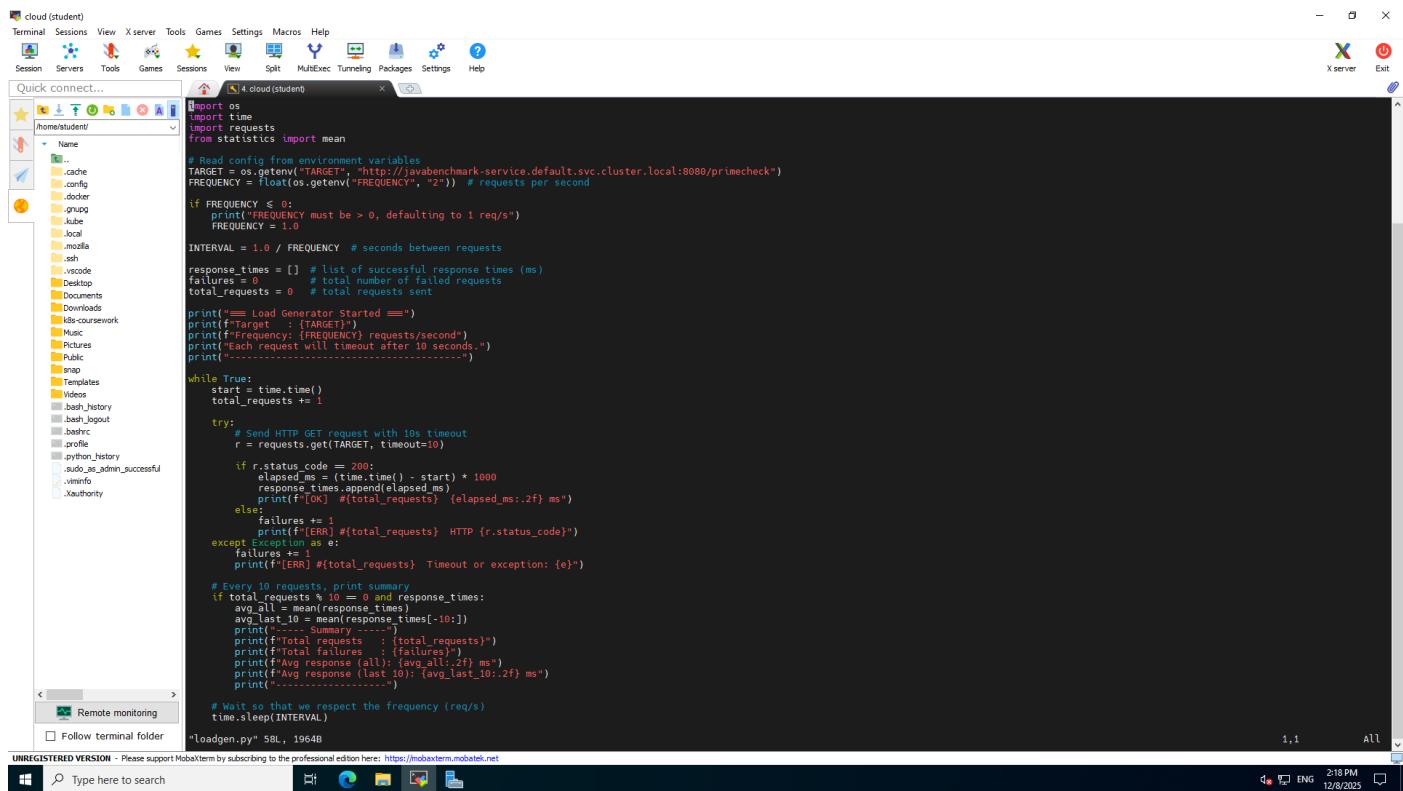
The objective of this task is to create a load generator using Python and Docker. This load generator continuously sends requests to the web application to create CPU and memory load. This helps us test how the application and monitoring system behave under load.

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.



A screenshot of the MobaXterm terminal window titled "cloud (student)". The window shows a Python script named "loadgen.py". The script imports os, time, requests, and statistics. It reads environment variables TARGET and FREQUENCY. It defines an INTERVAL as 1.0 / FREQUENCY. It initializes response_times, failures, and total_requests. It prints a start message and the target URL. It enters a while loop where it starts a timer, sends a GET request to the target with a 10-second timeout, and increments the total_requests counter. It then checks the status code: if 200, it appends the elapsed time to response_times; if an exception occurs, it increments failures. After 10 requests, it calculates the average response time for all requests and the last 10 requests, and prints these metrics along with the total requests and failures. Finally, it waits for the specified interval before the next iteration. The bottom of the terminal shows the command "loadgen.py" being run with parameters 5BL and 1964B.

```
#!/usr/bin/python3
# Read config from environment variables
TARGET = os.getenv("TARGET", "http://javabenchmark-service.default.svc.cluster.local:8088/primecheck")
FREQUENCY = float(os.getenv("FREQUENCY", "2")) # requests per second

if FREQUENCY <= 0:
    print("FREQUENCY must be > 0, defaulting to 1 req/s")
    FREQUENCY = 1.0

INTERVAL = 1.0 / FREQUENCY # seconds between requests

response_times = [] # list of successful response times (ms)
failures = 0 # total number of failed requests
total_requests = 0 # total requests sent

print("== Load Generator Started ==")
print(f"Target : {TARGET}")
print(f"Frequency: {FREQUENCY} requests/second")
print("With request will timeout after 10 seconds.")
print("-----")

while True:
    start = time.time()
    total_requests += 1

    try:
        # Send HTTP GET request with 10s timeout
        r = requests.get(TARGET, timeout=10)

        if r.status_code == 200:
            elapsed_ms = (time.time() - start) * 1000
            response_times.append(elapsed_ms)
            print(f"[OK] #{total_requests} {elapsed_ms:.2f} ms")
        else:
            failures += 1
            print(f"[ERR] #{total_requests} HTTP {r.status_code}")
    except Exception as e:
        failures += 1
        print(f"[ERR] #{total_requests} Timeout or exception: {e}")

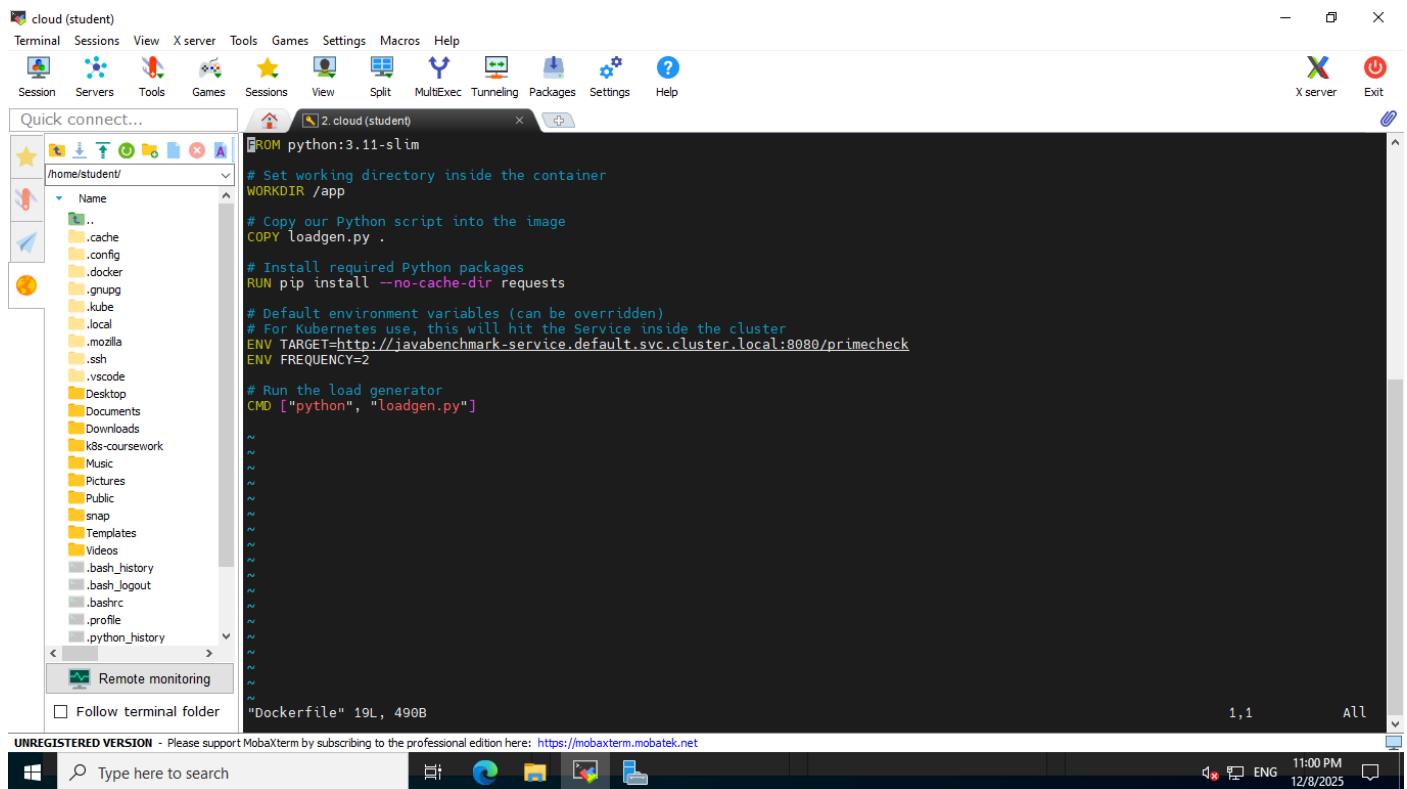
    # Every 10 requests, print summary
    if total_requests % 10 == 0 and response_times:
        avg_all = mean(response_times)
        avg_last_10 = mean(response_times[-10:])
        print("-----")
        print(f"Total requests : {total_requests}")
        print(f"Total failures : {failures}")
        print(f"Avg response (all): {avg_all:.2f} ms")
        print(f"Avg response (last 10): {avg_last_10:.2f} ms")
        print("-----")

    # Wait so that we respect the frequency (req/s)
    time.sleep(INTERVAL)

#loadgen.py 5BL, 1964B
```

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.



```

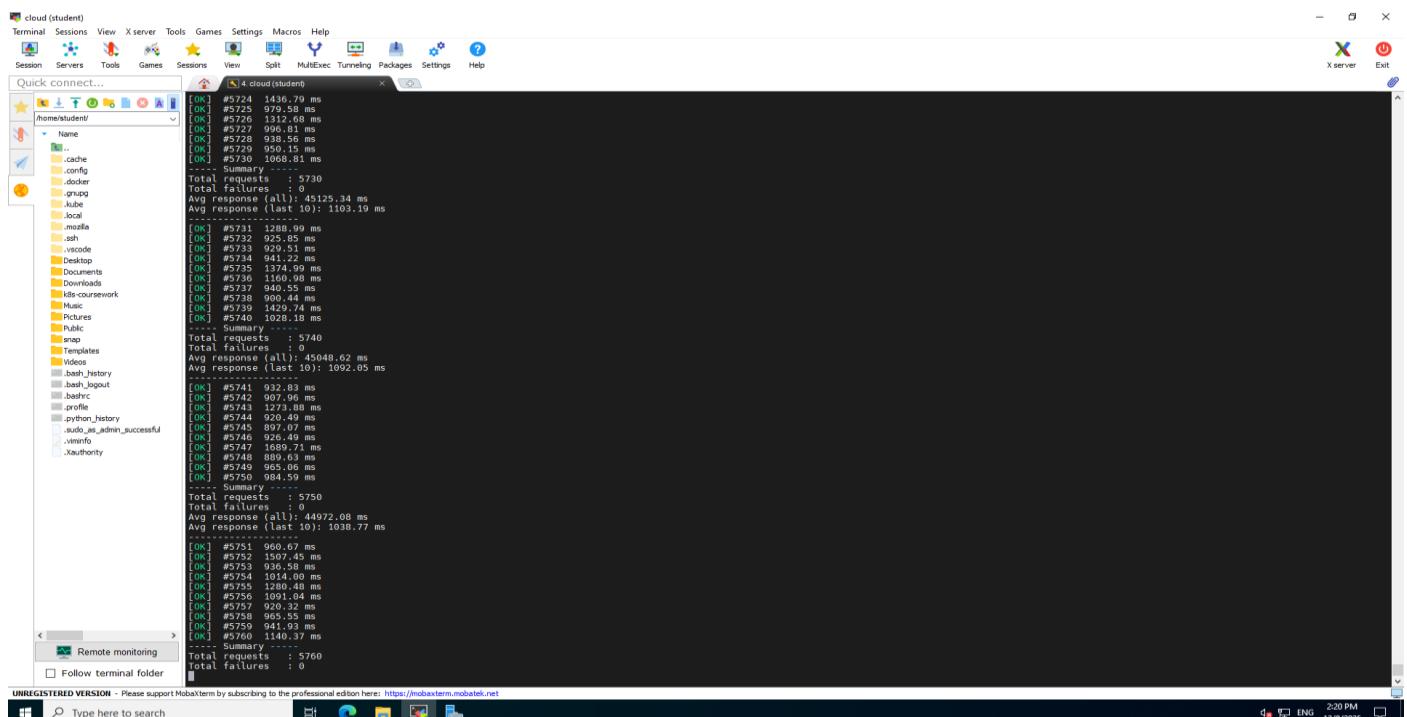
FROM python:3.11-slim
# Set working directory inside the container
WORKDIR /app
# Copy our Python script into the image
COPY loadgen.py .
# Install required Python packages
RUN pip install --no-cache-dir requests
# Default environment variables (can be overridden)
# For Kubernetes use, this will hit the Service inside the cluster
ENV TARGET=http://javabenchmark-service.default.svc.cluster.local:8080/primecheck
ENV FREQUENCY=2
# Run the load generator
CMD ["python", "loadgen.py"]

```

"Dockerfile" 19L, 490B

UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: <https://mobaxterm.mobatek.net>

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.



```

[OK] #5724 1436.79 ms
[OK] #5725 979.58 ms
[OK] #5726 1000.00 ms
[OK] #5727 996.01 ms
[OK] #5728 938.56 ms
[OK] #5729 1020.15 ms
[OK] #5730 1008.81 ms
---- Summary ----
Total requests : 5730
Total failures : 0
Avg response (all): 45125.34 ms
Avg response (last 10): 1103.19 ms

[OK] #5731 1288.99 ms
[OK] #5732 925.85 ms
[OK] #5733 929.31 ms
[OK] #5734 922.73 ms
[OK] #5735 1374.99 ms
[OK] #5736 1160.98 ms
[OK] #5737 900.38 ms
[OK] #5738 900.44 ms
[OK] #5739 1429.74 ms
[OK] #5740 1028.18 ms
---- Summary ----
Total requests : 5740
Total failures : 0
Avg response (all): 45048.62 ms
Avg response (last 10): 1092.05 ms

[OK] #5741 932.83 ms
[OK] #5742 907.96 ms
[OK] #5743 1273.88 ms
[OK] #5744 920.49 ms
[OK] #5745 921.30 ms
[OK] #5746 926.49 ms
[OK] #5747 1689.71 ms
[OK] #5748 921.30 ms
[OK] #5749 965.06 ms
[OK] #5750 984.59 ms
---- Summary ----
Total requests : 5750
Total failures : 0
Avg response (all): 44972.08 ms
Avg response (last 10): 1038.77 ms

[OK] #5751 960.67 ms
[OK] #5752 1007.40 ms
[OK] #5753 938.58 ms
[OK] #5754 1014.00 ms
[OK] #5755 1280.48 ms
[OK] #5756 920.32 ms
[OK] #5757 920.32 ms
[OK] #5758 965.55 ms
[OK] #5759 1111.93 ms
[OK] #5760 1111.93 ms
---- Summary ----
Total requests : 5760
Total failures : 0

```

UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: <https://mobaxterm.mobatek.net>

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 stop 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.

