1. Detailed response to each task and related sub-tasks

Student ID：230109586　Student Name：Boyan Li

Pre-Pequisites

Check the connectivity status of mobaXterm and edge VM： edge <username = student, password = CSC8110!>

Check the kubectl installation and the k8s cluster connection status to make sure the environment is ready.

```
kubectl version --client     # check kubectl installation
kubectl cluster-info         # check k8s cluster connection
```

```
student@edge:~$ kubectl version --client
WARNING: This version information is deprecated and will be replaced with the ou
tput from kubectl version --short.  Use --output=yaml|json to get the full versi
on.
Client Version: version.Info{Major:"1", Minor:"27", GitVersion:"v1.27.7", GitCom
mit:"07a61d861519c45ef5c89bc22dda289328f29343", GitTreeState:"clean", BuildDate:
"2023-10-18T16:09:55Z", GoVersion:"go1.20.10", Compiler:"gc", Platform:"linux/am
d64"}
Kustomize Version: v5.0.1
student@edge:~$ kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:16443
CoreDNS is running at https://127.0.0.1:16443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
student@edge:~$
```

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

KR1 · Deploy 'Kubernetes Dashboard' on the provided VM with CLI and access/login the Dashboard.

```
mkdir Config                          # create the folder Config
cd ~/Config                           # cd the folder Config
wget
https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.
yaml    # download recommended.yaml
kubectl apply -f recommonded.yaml    # deploy/update recommonded.yaml
```

```
student@edge:~/Config$ wget https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/rec
ommended.yaml
--2023-12-01 21:32:10--  https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recomm
ended.yaml
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133, 185.199.110.133, 185
.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7621 (7.4K) [text/plain]
Saving to: 'recommended.yaml'

recommended.yaml          100%[=====================================>]   7.44K  --.-KB/s    in 0s

2023-12-01 21:32:11 (75.9 MB/s) - 'recommended.yaml' saved [7621/7621]

student@edge:~/Config$
```

```
student@edge:~$ cd ~/student/Config
-bash: cd: /home/student/student/Config: No such file or directory
student@edge:~$ cd ~/Config
student@edge:~/Config$ kubectl apply -f recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
Warning: spec.template.metadata.annotations[seccomp.security.alpha.kubernetes.io
/pod]: non-functional in v1.27+; use the "seccompProfile" field instead
deployment.apps/dashboard-metrics-scraper created
student@edge:~/Config$
```

```
kubectl get pods -A                          # check pods status in all namespaces
kubectl get pods -n kubernetes-dashboard  # check pods status in namespace
"Kubernetes-dashboard"
kubectl -n kubernetes-dashboard get svc   # check services in namespace "Kubernetes-
dashboard"
kubectl proxy                                # turn on the server and access the
Dashboard UI
```

```
student@edge:~/Config$ kubectl get pods --all-namespaces
NAMESPACE              NAME                                         READY   STATUS    RESTAR
TS      AGE
kube-system            calico-kube-controllers-6c99c8747f-4t9md     1/1     Running   8 (11h
 ago)   18d
kube-system            coredns-7745f9f87f-jtk6s                     1/1     Running   8 (11h
 ago)   18d
container-registry     registry-9865b655c-7c6gq                     1/1     Running   7 (11h
 ago)   18d
kube-system            hostpath-provisioner-58694c9f4b-zc8fs        1/1     Running   8 (11h
 ago)   18d
kube-system            calico-node-zb7hj                            1/1     Running   8 (11h
 ago)   18d
kubernetes-dashboard   dashboard-metrics-scraper-764cf47594-vtxhl   1/1     Running   0
       43m
kubernetes-dashboard   kubernetes-dashboard-67dbc597b5-jl24l        1/1     Running   0
       43m
student@edge:~/Config$ kubectl get pods -n kubernetes-dashboard
NAME                                         READY   STATUS    RESTARTS   AGE
dashboard-metrics-scraper-764cf47594-vtxhl   1/1     Running   0          44m
kubernetes-dashboard-67dbc597b5-jl24l        1/1     Running   0          44m
student@edge:~/Config$ kubectl -n kubernetes-dashboard get svc
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)         AGE
kubernetes-dashboard        NodePort    10.152.183.43    <none>        443:30443/TCP   45m
dashboard-metrics-scraper   ClusterIP   10.152.183.126   <none>        8000/TCP        45m
student@edge:~/Config$
```


After run “kubectl proxy”, check the status of pods and services again.

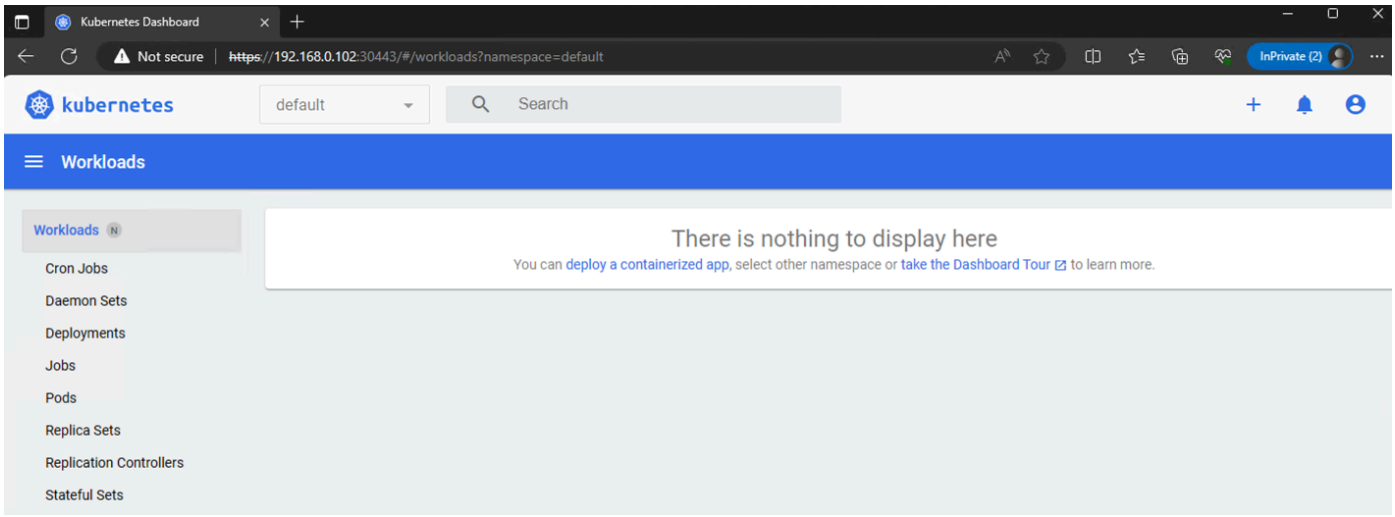
```
student@edge:~$ cd Config
student@edge:~/Config$ kubectl proxy
Starting to serve on 127.0.0.1:8001
^C
student@edge:~/Config$ kubectl get pods -n kubernetes-dashboard
NAME                                         READY   STATUS    RESTARTS   AGE
dashboard-metrics-scraper-764cf47594-vtxhl   1/1     Running   0          4h52m
kubernetes-dashboard-67dbc597b5-jl24l        1/1     Running   0          4h52m
student@edge:~/Config$ kubectl -n kubernetes-dashboard get svc
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)         AGE
kubernetes-dashboard        NodePort    10.152.183.43    <none>        443:30443/TCP   4h54m
dashboard-metrics-scraper   ClusterIP   10.152.183.126   <none>        8000/TCP        4h54m
student@edge:~/Config$
```

Use the browser in the VM which has the mobaXterm, visit K8S-dashboard：https://192.168.0.102:30443

- Need to imput token. Here's a quick way to get token:

```
microk8s config   # get token:
bjdPb2YzYmMxWFRrUHIvNHh2SlhsV3BQTjFUallHeno2ZW9tYXVid0hNODOK
```

- Visit successfully:



KR2 · Deploy an instance of the Docker image "nclcloudcomputing/javabenchmarkapp" via CLI.

Create and edit "javaapp-deploy.yaml" file in ~/Config.

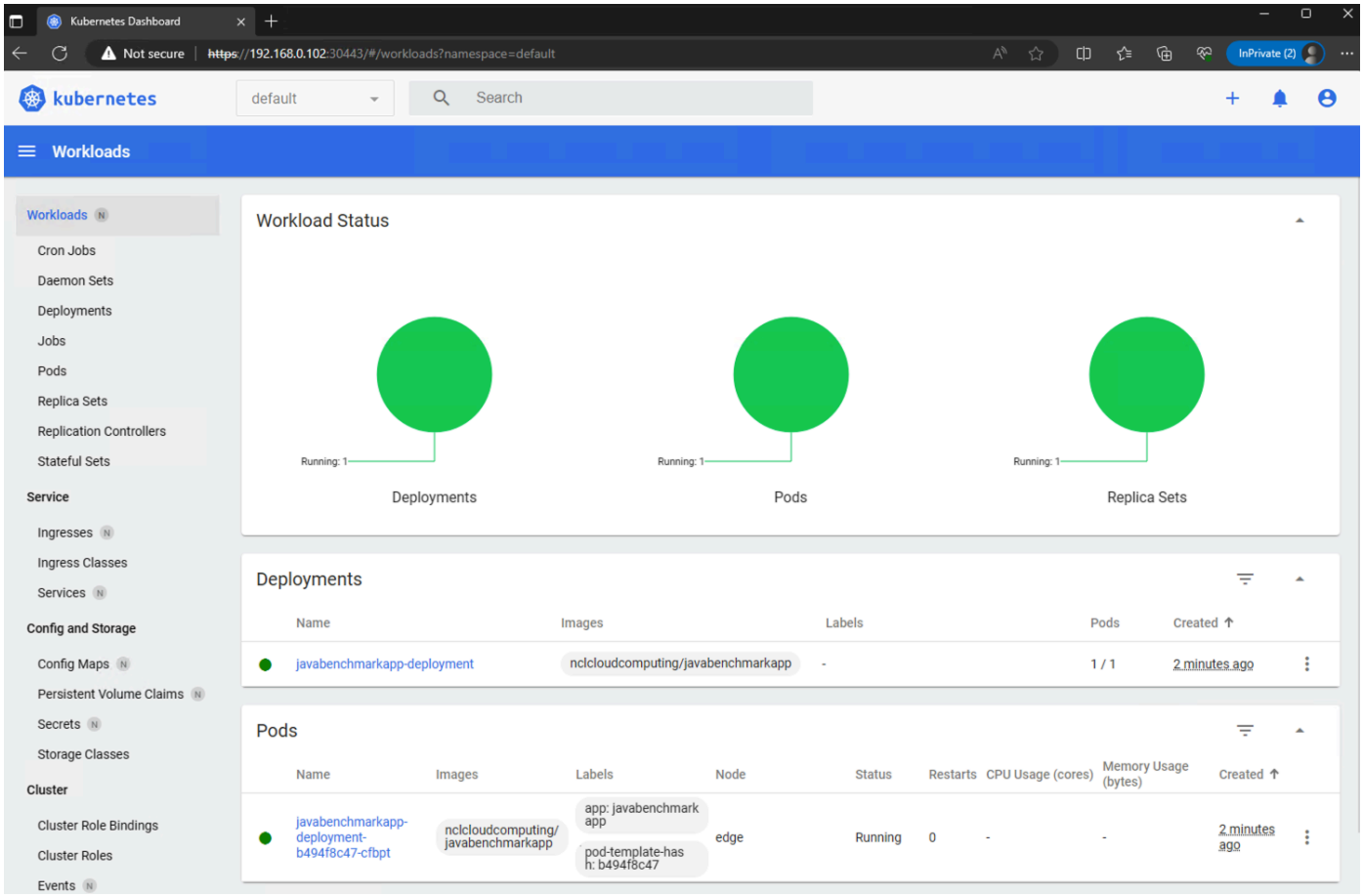
```
touch javaapp-deploy.yaml    # create javaapp-deploy.yaml
nano javaapp-deploy.yaml     # modify javaapp-deploy.yaml
```

```
# javaapp-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: javabenchmarkapp-deployment
spec:
  selector:
    matchLabels:
      app: javabenchmarkapp
  template:
    metadata:
      labels:
        app: javabenchmarkapp
    spec:
      containers:
      - name: javabenchmarkapp-container
        image: nclcloudcomputing/javabenchmarkapp
        resources:
          limits:
            memory: "512Mi"
```

```
            cpu: "500m"
        ports:
        - containerPort: 8080
```

```
kubectl apply -f javaapp-deploy.yaml    # run/update javaapp-deploy.yaml
```

Deploy successfully.



KR3 · Deploy a NodePort service so that the web app is accessible via http://localhost:30000/primecheck. The container uses port 8080 internally.

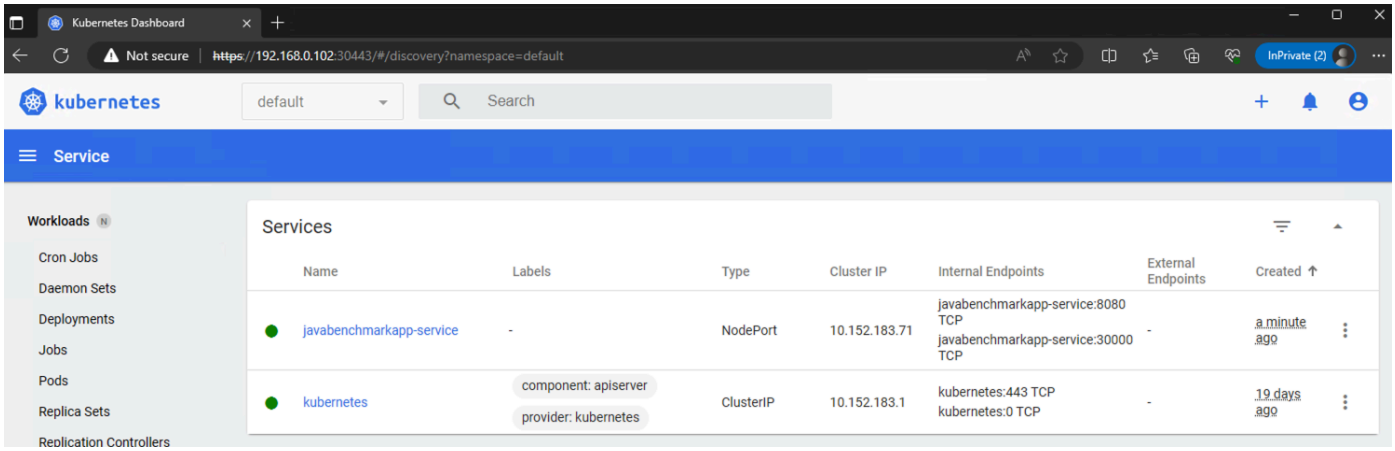
Create and edit "javaapp-svc.yaml" file in ~/Config.

```
touch javaapp-svc.yaml    # create javaapp-svc.yaml
nano javaapp-svc.yaml   # modify javaapp-svc.yaml
```

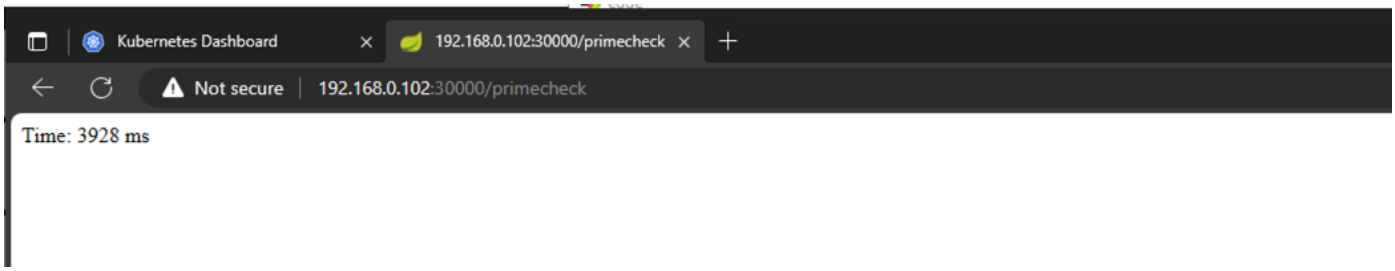
```
# javaapp-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: javabenchmarkapp-service
spec:
  type: NodePort
  selector:
    app: javabenchmarkapp
  ports:
  - port: 8080
    targetPort: 8080
    nodePort: 30000
```

```
kubectl apply -f javaapp-svc.yaml    # run/update javaapp-svc.yaml
```

Deploy service successfully.



the web app is accessible via http://192.168.0.102:30000/primecheck



Object 2: Deploy the monitoring stack of Kubernetes

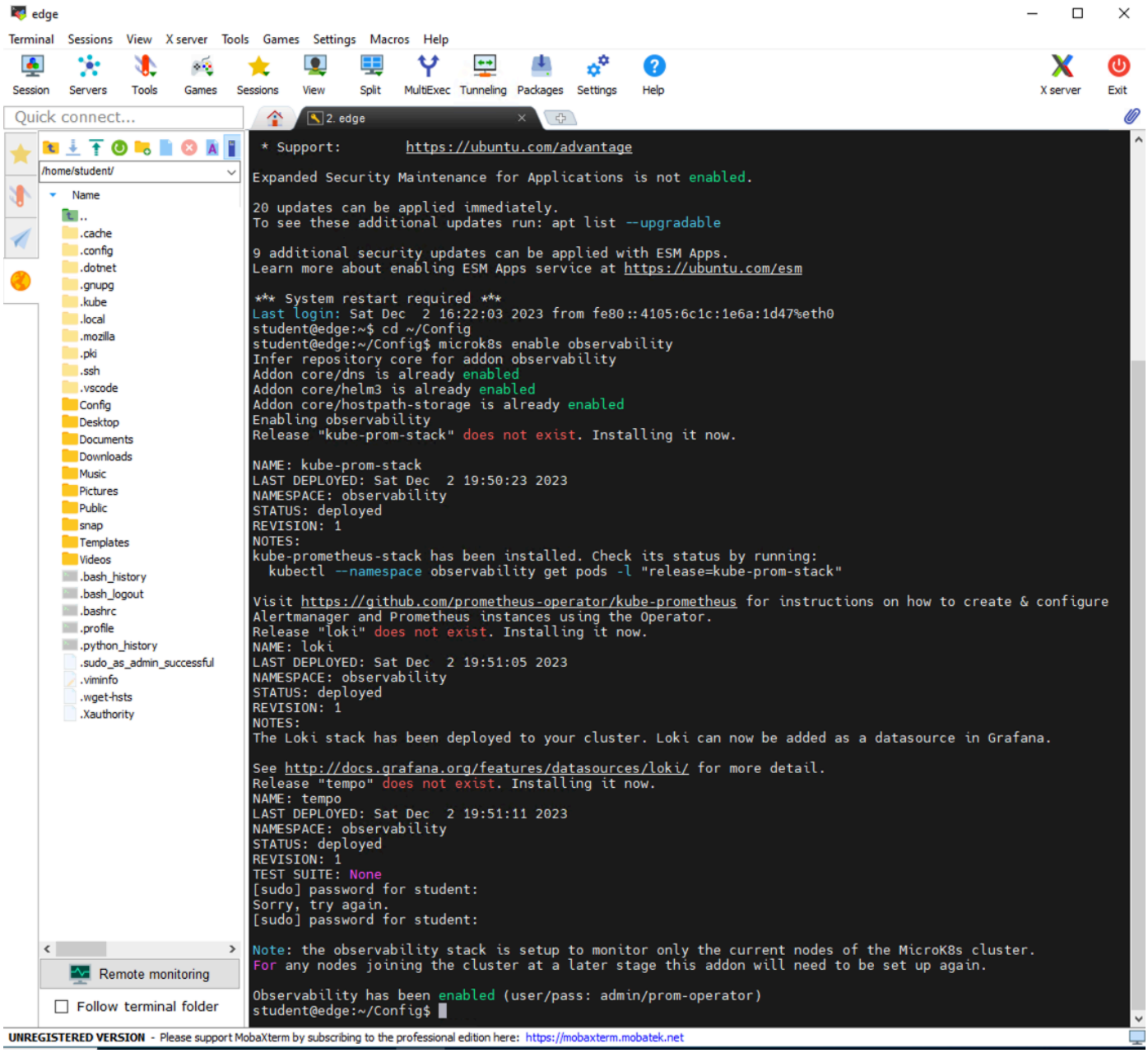
KR1 · Enable observability service from microk8s addons.

```
# command line
microk8s enable observability
```

"microk8s": is interface of the MicroK8s tool for managing and operating MicroK8s clusters.

"enable": This is a MicroK8s command that enables or activates a specific function or plug-in. In this case, enable is used to enable functionality related to observability.

"observability": This is the name given to a set of observability capabilities that typically include tools and components for monitoring and diagnosing Kubernetes clusters and applications.



: Grafana Login (username = admin, password = prom-operator)

```
kubectl get svc -A | grep grafana             # check grafana services
kubectl get pods -n observability             # check pods in 'observability'
namespace

# get information about the service named "kube-prom-stack-grafana" from the
namespace named "observability".
kubectl get svc kube-prom-stack-grafana -n observability -o yaml > ~/Config/grafana-
svc-output.yaml
# run/update grafana-svc-output.yaml
kubectl apply -f grafana-svc-output.yaml
```

```
student@edge:~/Config$ kubectl get svc -A|grep grafana
observability          kube-prom-stack-grafana                          ClusterIP   10.152.183.100   <no
ne>        80/TCP
                         18m
student@edge:~/Config$ kubectl get pods -n observability
NAME                                                     READY   STATUS    RESTARTS   AGE
kube-prom-stack-kube-prome-operator-64ffd55b77-msrg2     1/1     Running   0          19m
tempo-0                                                  2/2     Running   0          18m
alertmanager-kube-prom-stack-kube-prome-alertmanager-0   2/2     Running   0          19m
kube-prom-stack-prometheus-node-exporter-x8rnm           1/1     Running   0          19m
kube-prom-stack-grafana-6c47f548d6-fq8fp                 3/3     Running   0          19m
kube-prom-stack-kube-state-metrics-6c586bf4c8-2vpzt      1/1     Running   0          19m
loki-promtail-2r7b4                                      1/1     Running   0          18m
prometheus-kube-prom-stack-kube-prome-prometheus-0       2/2     Running   0          19m
loki-0                                                   1/1     Running   0          18m
student@edge:~/Config$ kubectl get svc kube-prom-stack-grafana -n observability -o yaml > ~/Config/grafana-svc-output.yaml
student@edge:~/Config$
```

KR2 · Edit the Grafana service to allow access from the host.

Edit grafana-svc-output.yaml:

1. add: nodePort: 30001
2. modify: type: NodePort

```
# modify grafana-svc-output.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Service","metadata":{"annotations":
{"meta.helm.sh/release-name":"kube-prom-stack","meta.helm.sh/release-
namespace":"observability"},"creationTimestamp":"2023-12-02T19:50:40Z","labels":
{"app.kubernetes.io/instance":"kube-prom-stack","app.kubernetes.io/managed-
by":"Helm","app.kubernetes.io/name":"grafana","app.kubernetes.io/version":"9.3.8","he
lm.sh/chart":"grafana-6.51.2"},"name":"kube-prom-stack-
grafana","namespace":"observability","resourceVersion":"80294","uid":"3ecbc728-906e-
4394-9d54-6d7c5f88e12b"},"spec":{"clusterIP":"10.152.183.100","clusterIPs":
["10.152.183.100"],"internalTrafficPolicy":"Cluster","ipFamilies":
["IPv4"],"ipFamilyPolicy":"SingleStack","ports":[{"name":"http-
web","nodePort":30001,"port":80,"protocol":"TCP","targetPort":3000}],"selector":
{"app.kubernetes.io/instance":"kube-prom-
stack","app.kubernetes.io/name":"grafana"},"sessionAffinity":"None","type":"NodePort"
},"status":{"loadBalancer":{}}}
    meta.helm.sh/release-name: kube-prom-stack
    meta.helm.sh/release-namespace: observability
  creationTimestamp: "2023-12-02T19:50:40Z"
  labels:
```

```
    app.kubernetes.io/instance: kube-prom-stack
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: grafana
    app.kubernetes.io/version: 9.3.8
    helm.sh/chart: grafana-6.51.2
  name: kube-prom-stack-grafana
  namespace: observability
  resourceVersion: "83430"
  uid: 3ecbc728-906e-4394-9d54-6d7c5f88e12b
spec:
  clusterIP: 10.152.183.100
  clusterIPs:
  - 10.152.183.100
  externalTrafficPolicy: Cluster
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: http-web
    nodePort: 30001
    port: 80
    protocol: TCP
    targetPort: 3000
    nodePort: 30001
  selector:
    app.kubernetes.io/instance: kube-prom-stack
    app.kubernetes.io/name: grafana
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

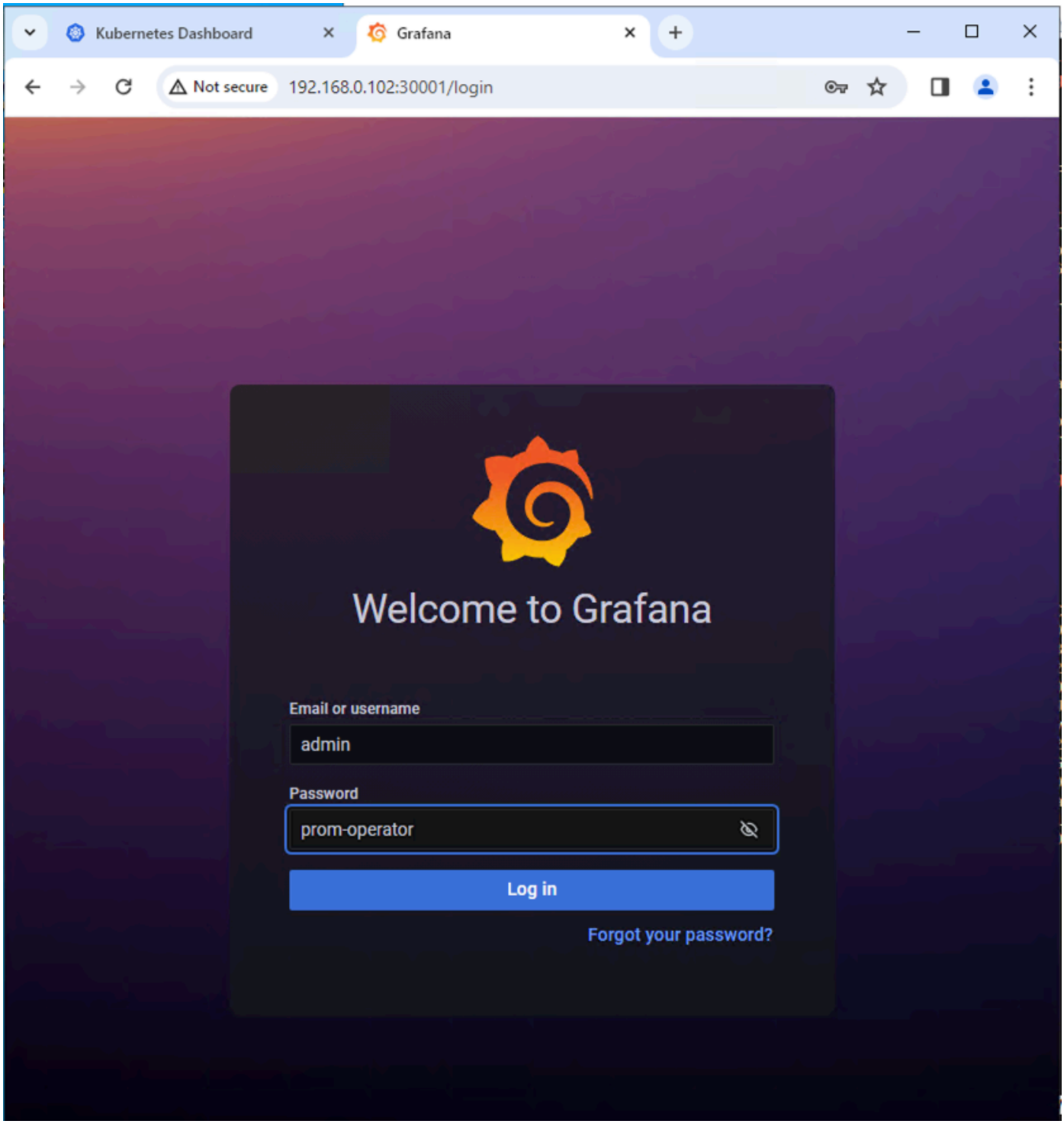
Update grafana-svc-output.yaml, and check grafana service status again:

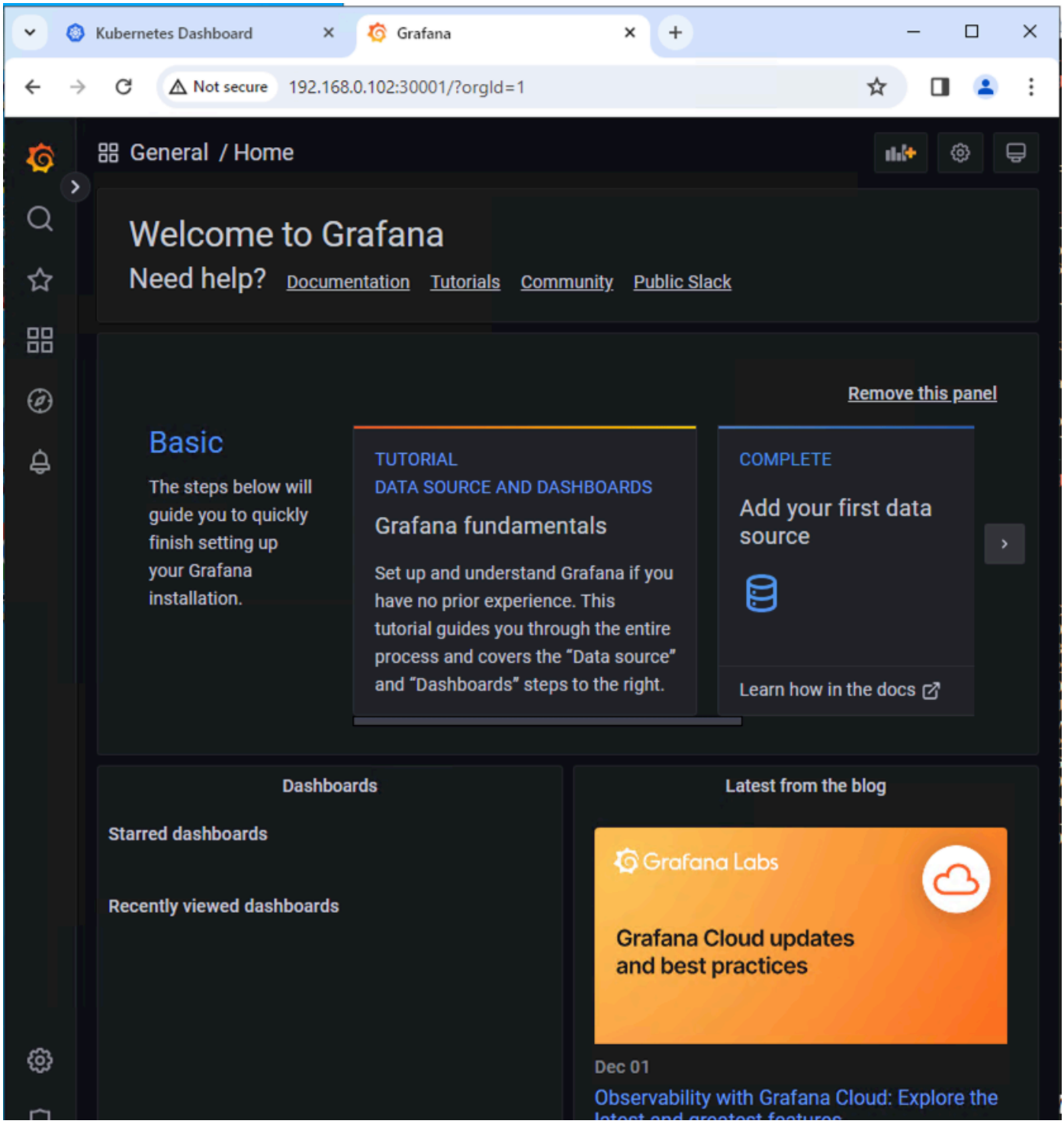
```
kubectl apply -f grafana-svc-output.yaml    # run/update grafana-svc-output.yaml
kubectl get svc -A | grep grafana           # check grafana service status again
```

```
student@edge:~/Config$ kubectl apply -f grafana-svc-output.yaml
Warning: resource services/kube-prom-stack-grafana is missing the kubectl.kubernetes.io/last-applied-configuration annotat
ion which is required by kubectl apply. kubectl apply should only be used on resources created declaratively by either kub
ectl create --save-config or kubectl apply. The missing annotation will be patched automatically.
service/kube-prom-stack-grafana configured
student@edge:~/Config$ kubectl get svc -A | grep grafana
observability          kube-prom-stack-grafana                          NodePort    10.152.183.100   <none>        80:
30001/TCP
 26m
student@edge:~/Config$
```

KR3 · Log in to the Grafana dashboard.

Grafana is using 30001 port:





Object 3: Load Generator

KR1 · A load generator with the following specifications

(a) Accepts two configurable values either via a config file or environment variables. target (The address for the load generation) and frequency (Request per second)

(b) Generate web request to the target at the specified frequency

(c) Collect 2 types of metrics. Average response time and accumulated number of failures

(d) Request should timeout if it takes more than 10 seconds. Counted as failures

(e) Test results need to be printed to the console

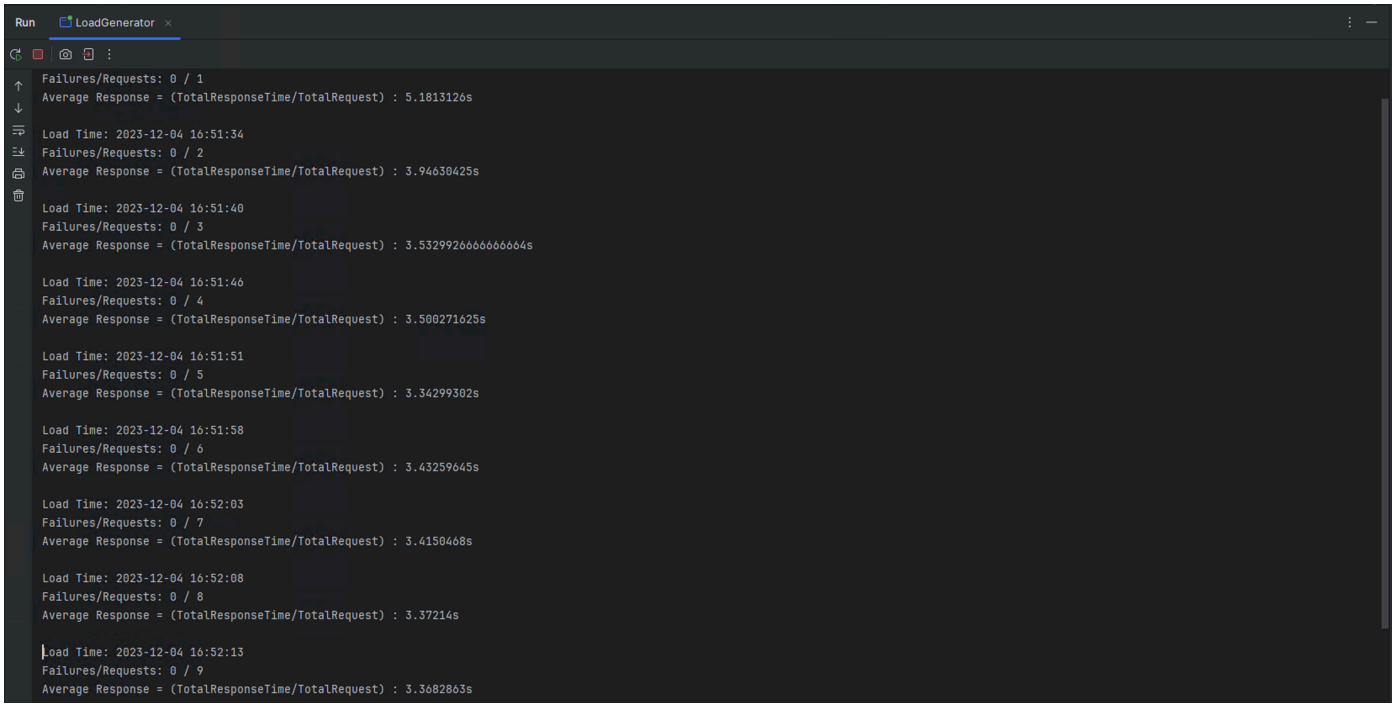
(f) There are no requirements in programming language

```java
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.concurrent.TimeUnit;

/**
 * LoadGenerator class to simulate network requests for load testing.
 * This class is designed to generate HTTP requests to a specified target URL at a
specified frequency,
 * and track the number of total requests, failures, and the total response time.
 */
public class LoadGenerator {

    private final String target;
    private final int frequency;
    private int totalRequests;
    private int totalFailures;
    private double totalResponseTime;

    /**
     * Constructor to initialize LoadGenerator with target URL and frequency.
     *
     * @param target The URL to which the load requests are sent.
     * @param frequency The number of requests per second.
     */
    public LoadGenerator(String target, int frequency) {
        this.target = target;
        this.frequency = frequency;
        this.totalRequests = 0;
        this.totalFailures = 0;
        this.totalResponseTime = 0;
    }
```

```
    /**
     * Makes a single HTTP GET request to the target URL.
     * Records the response time and updates the total number of requests and
failures.
     */
    public void makeRequest() {
        long startTime = System.nanoTime();
        try {
            HttpURLConnection connection = (HttpURLConnection) new
URL(target).openConnection();
            connection.setConnectTimeout(10000);
            connection.setReadTimeout(10000);
            connection.setRequestMethod("GET");
            connection.connect();

            int status = connection.getResponseCode();
            if (status != HttpURLConnection.HTTP_OK) {
                totalFailures++;
            }

            connection.disconnect();
        } catch (IOException e) {
            totalFailures++;
        } finally {
            long endTime = System.nanoTime();
            totalResponseTime += (endTime - startTime) / 1e9;
        }
    }

    /**
     * Runs the load generator in a continuous loop.
     * This method generates load according to the specified frequency and prints the
load information.
     */
    public void run() {
        while (true) {
            long startEpoch = System.nanoTime();
```

```
            for (int i = 0; i < frequency; i++) {
                makeRequest();
                totalRequests++;
            }

            long endEpoch = System.nanoTime();
            double elapsedSeconds = (endEpoch - startEpoch) / 1e9;

            DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
            String formattedDate = dtf.format(LocalDateTime.now());
            System.out.println("Load Time: " + formattedDate);
            System.out.println("Failures/Requests: " + totalFailures + " / " +
totalRequests);
            System.out.println("Average Response = (TotalResponseTime/TotalRequest) :
" + (totalResponseTime / totalRequests) + "s\n");

            try {
                TimeUnit.SECONDS.sleep(Math.max(0, 5 - (long) elapsedSeconds));
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.out.println("The load generator was interrupted!");
                break;
            }
        }
    }

    /**
     * Main method to start the LoadGenerator.
     */
    public static void main(String[] args) {
        String targetAddress = "http://192.168.0.102:30000/primecheck";  //
javabenchmarkapp
        int requestFrequency = 10;                                      // requests per
second
        LoadGenerator generator = new LoadGenerator(targetAddress, requestFrequency);
        generator.run();
    }
}
```

LoadGenerator.java local running result：



KR2 · 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.

File structure

```
load-generator
  ├── Dockerfile
  └── LoadGenerator.java
```

```
# Dockerfile
# Use an official OpenJDK image as a parent image (Already installed through
mobaXterm)
FROM openjdk:21-jdk

# Set the working directory to /app
WORKDIR /app

# Copy the Java source code into the container at /app
COPY . /app

# Compile the Java code
RUN javac LoadGenerator.java
```

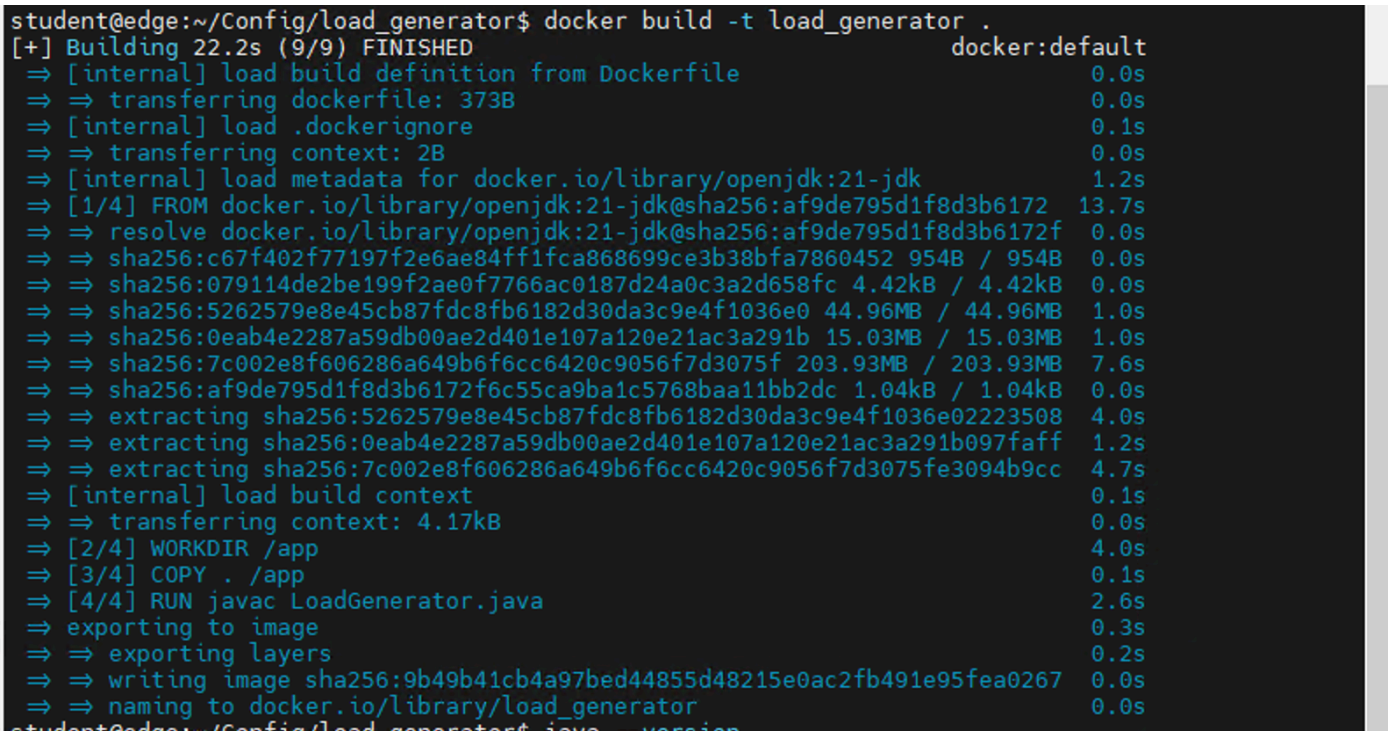
```
# Run the application when the container lauches
CMD ["java", "LoadGenerator"]
```

Deploy Docker image

```
cd load-generator                    # enter load-generator folder
docker build -t load_generator .     # use this Dockerfile to build the image "load-generator"
docker run --rm -it load_generator   # run load_generator container and delete it after exist
```

The third ctl is to check the running staus:

- `--rm`: This is an option to automatically delete the container when it exits. This ensures that the container doesn't leave anything behind after it exits.
- `-it`: '-i' means to keep the container's standard input open, and '-t' means to allocate a pseudo-TTY for interacting with the container. This allows to have interactive command-line sessions inside the container.



Check if the load_generator image is running inside the container, which is a one-time operation, so the container is set to be deleted after exist:

```
student@edge:~/Config/load_generator$ docker run --rm -it load_generator
Load Time: 2023-12-04 19:36:49
Failures/Requests: 0 / 1
Average Response = (TotalResponseTime/TotalRequest) : 2.107322877s

Load Time: 2023-12-04 19:36:54
Failures/Requests: 0 / 2
Average Response = (TotalResponseTime/TotalRequest) : 2.038240689s

Load Time: 2023-12-04 19:37:01
Failures/Requests: 0 / 3
Average Response = (TotalResponseTime/TotalRequest) : 2.1172002780000003s

Load Time: 2023-12-04 19:37:06
Failures/Requests: 0 / 4
Average Response = (TotalResponseTime/TotalRequest) : 2.1185342190000003s

Load Time: 2023-12-04 19:37:11
Failures/Requests: 0 / 5
Average Response = (TotalResponseTime/TotalRequest) : 2.0938254432s

Load Time: 2023-12-04 19:37:17
Failures/Requests: 0 / 6
Average Response = (TotalResponseTime/TotalRequest) : 2.09236029116666666s

Load Time: 2023-12-04 19:37:22
Failures/Requests: 0 / 7
Average Response = (TotalResponseTime/TotalRequest) : 2.08090478271428575s

Load Time: 2023-12-04 19:37:27
Failures/Requests: 0 / 8
Average Response = (TotalResponseTime/TotalRequest) : 2.080657552125s

Load Time: 2023-12-04 19:37:32
Failures/Requests: 0 / 9
Average Response = (TotalResponseTime/TotalRequest) : 2.0834539448888889s

Load Time: 2023-12-04 19:37:37
Failures/Requests: 0 / 10
Average Response = (TotalResponseTime/TotalRequest) : 2.0681089589s

Load Time: 2023-12-04 19:37:43
Failures/Requests: 0 / 11
```

Push it to the local registry at port 32000 and name the image as load-generator

```
docker tag load-generator:latest localhost:32000/load-generator:latest     # docker
tag
docker push localhost:32000/load-generator:latest                          # docker push
# make sure the image completes push successfully
curl -X GET http://localhost:32000/v2/_catalog
```

Tag： Mark the load-generator:latest image as localhost:32000/load-generator:latest for future reference via that tag.

Push： Push the previously marked load-generator:latest image to localhost:32000 in the local docker repository to store the image and make it accessible.

```
student@edge:~/Config/load_generator$ docker tag load_generator localhost:32000/load_generator
student@edge:~/Config/load_generator$ docker push localhost:32000/load_generator
Using default tag: latest
The push refers to repository [localhost:32000/load_generator]
26934fcc3707: Pushed
c06915435cf8: Pushed
929882fa2ead: Pushed
10359c5dc4ba: Pushed
601b48657e0c: Pushed
b42107e74152: Pushed
latest: digest: sha256:6de25a23f883ac883d90d448809e876945a24906f9100850e500d36cea9fff50 size: 15
76
student@edge:~/Config/load_generator$ curl -X GET http://localhost:32000/v2/_catalog
{"repositories":["load_generator"]}
student@edge:~/Config/load_generator$
```

restart docker and check "docker images" again：

```
sudo systemctl restart docker    # restart docker
docker images                    # check "docker images" status again
```

Pushed and pulled successfully.

```
student@edge:~/Config/load_generator$ sudo systemctl restart docker
[sudo] password for student:
student@edge:~/Config/load_generator$ docker images
REPOSITORY                          TAG       IMAGE ID       CREATED         SIZE
load_generator                      latest    9b49b41cb4a9   2 hours ago     504MB
localhost:32000/load_generator      latest    9b49b41cb4a9   2 hours ago     504MB
student@edge:~/Config/load_generator$
```

Object 4: Monitor benchmarking results

Hints : In the Grafana panel, can specify metric: container_cpu_usage_seconds_total/ container_memory_usage_bytes

KR1 · Deploy load-generator service created before

Create and edit "loadgenerator.yaml" file in ~/Config.

```
touch loadgenerator.yaml     # create loadgenerator.yaml
nano loadgenerator.yaml      # modify loadgenerator.yaml
```

```
# loadgenerator.yaml
 apiVersion: apps/v1
 kind: Deployment
 metadata:
   name: load-generator-deployment
 spec:
   selector:
     matchLabels:
```

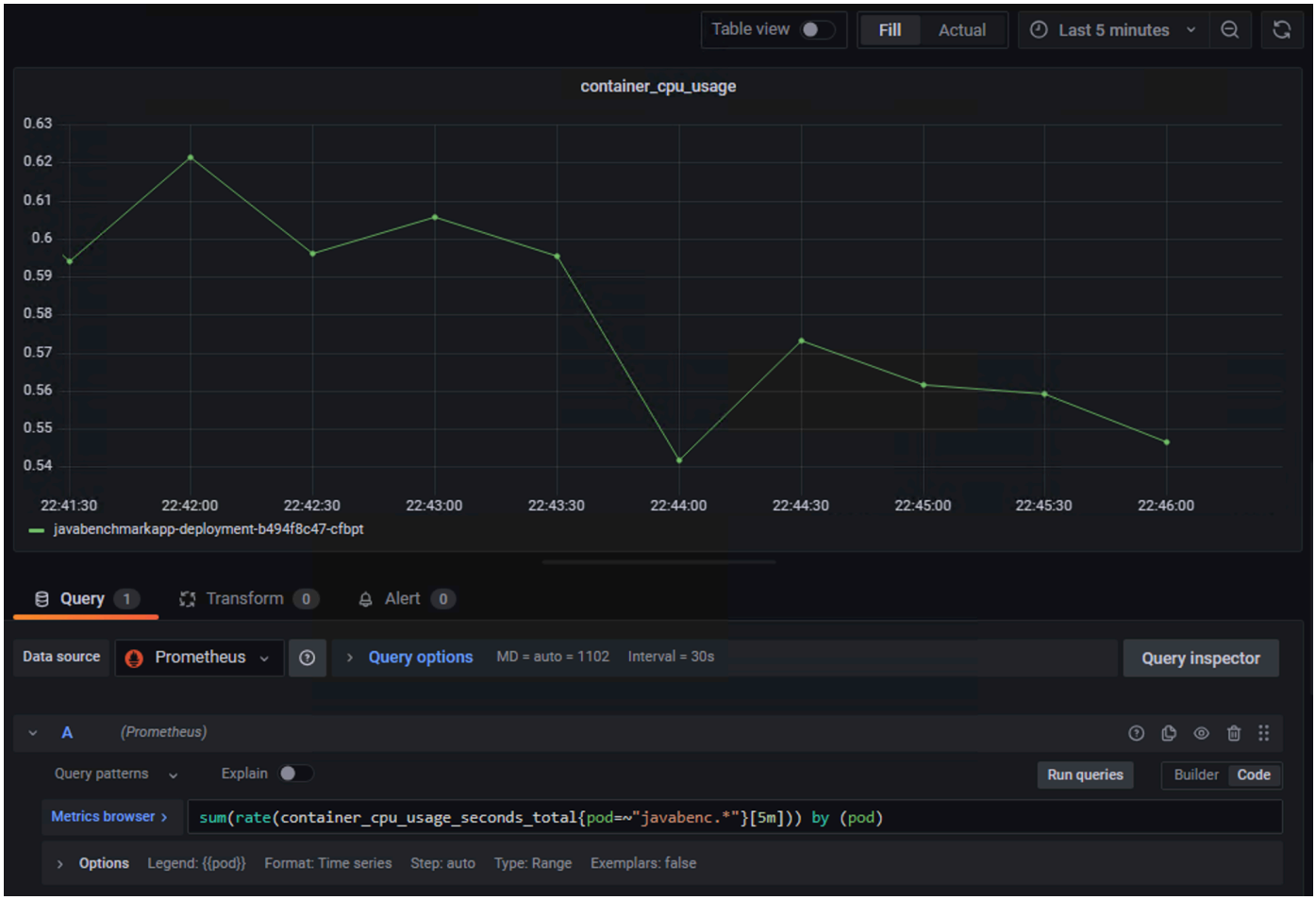
```
      app: load-generator
  template:
    metadata:
      labels:
        app: load-generator
    spec:
      containers:
      - name: load-generator
        image: localhost:32000/load_generator
        ports:
        - containerPort: 8081
```

```
kubectl apply -f loadgenerator.yaml   # run/update loadgenerator.yaml
```

Deploy load-generator service successfully.

KR2+3 · 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+Screenshot the two panels)

· panel1: container_cpu_usage




```
# Prometheus query
sum(rate(container_cpu_usage_seconds_total{pod=~"javabenc.*"}[5m])) by (pod)
```

This is a Prometheus query that calculates the sum of CPU usage for a specific condition and groups the results by the label of the container (in this case, the pod label). Here‘s the explaination of each parts of this query:

- sum(): This is an aggregate function that sums the results of expressions in parentheses.
- rate(): Calculates the rate at which the CPU is used
- container_cpu_usage_seconds_total{pod=~"javabenc.*"}: *This is a metric selector that selects time series data with the container_cpu_usage_seconds_total metric name and a pod label, where the value of the pod label matches the regular expression javabenc.*. This will select the time series data for all containers whose pod label starts with "javabenc".
- [5m]: Specifies the time range to consider. Here, it selected the data points of the last 5 minutes to calculate the rate of CPU usage.
- by (pod): This is a grouping operation that groups the results by different values of the pod label in order to display the sum of CPU usage for each different pod in the result.

Putting it all together, the purpose of this query is to calculate the sum of CPU usage for all containers that match the criteria (pod labels start with "javabenc") over the past 5 minutes, and to get the sum of CPU usage for each pod, grouped by each distinct pod value.

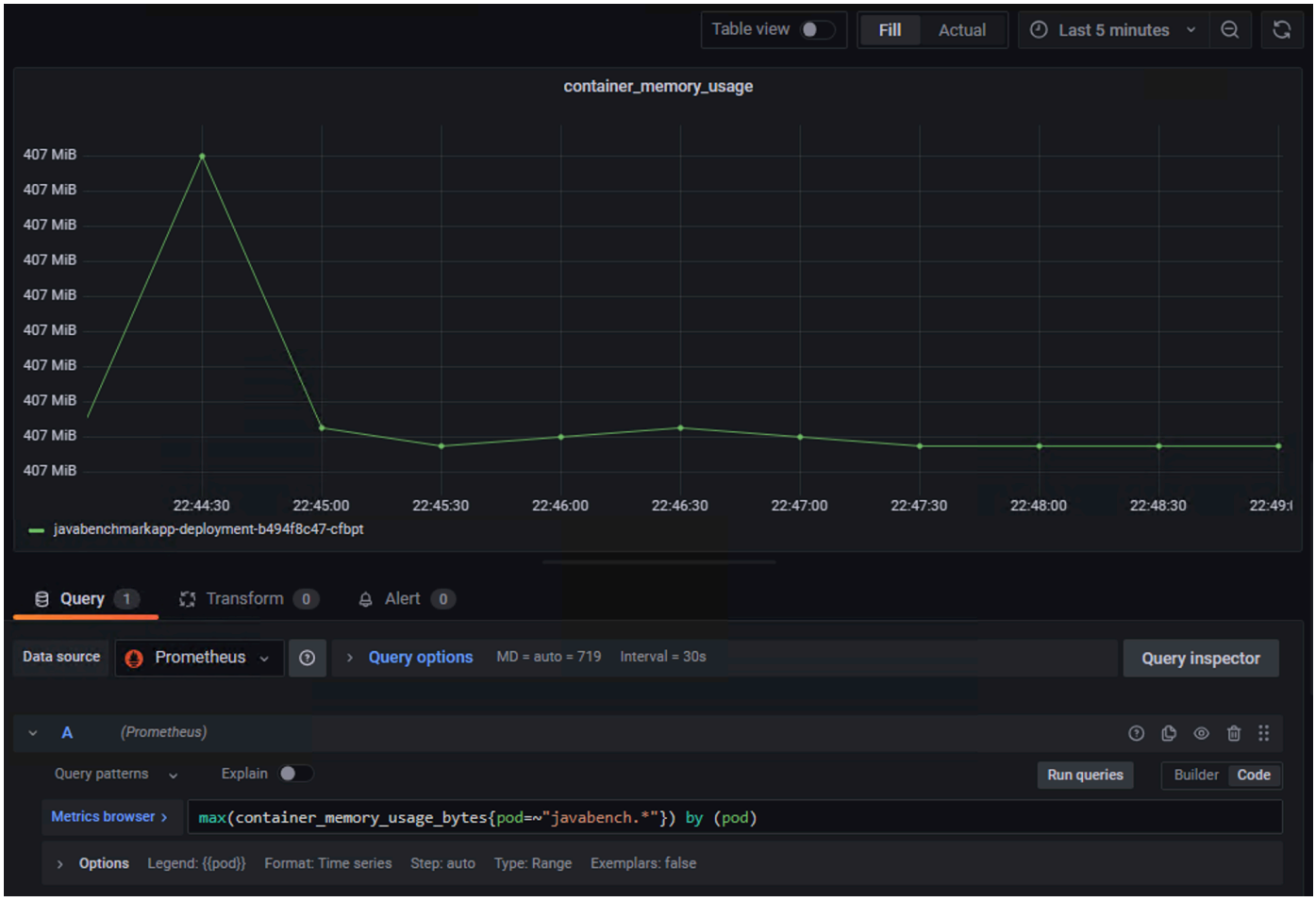
： Choose to use rate() vs irate() , in the Prometheus query：

```
sum(irate(container_cpu_usage_seconds_total{pod=~"javabenc.*"}[5m])) by (pod)
```

- rate() : This computes the average rate over a given time horizon and is more suitable for smooth long-term trend analysis because it considers more data points over a longer period of time, thus reducing the impact of transient jitter.
- irate() is good for observing rates of change over short periods of time, such as sudden spikes or drops.

Here we looking at cpu usage over a certain period of time(5m), so I chose to use rate() here.

· panel2: container_memory_usage




```
# Prometheus query
max(container_memory_usage_bytes{pod=~"javabench.*"}) by (pod)
```

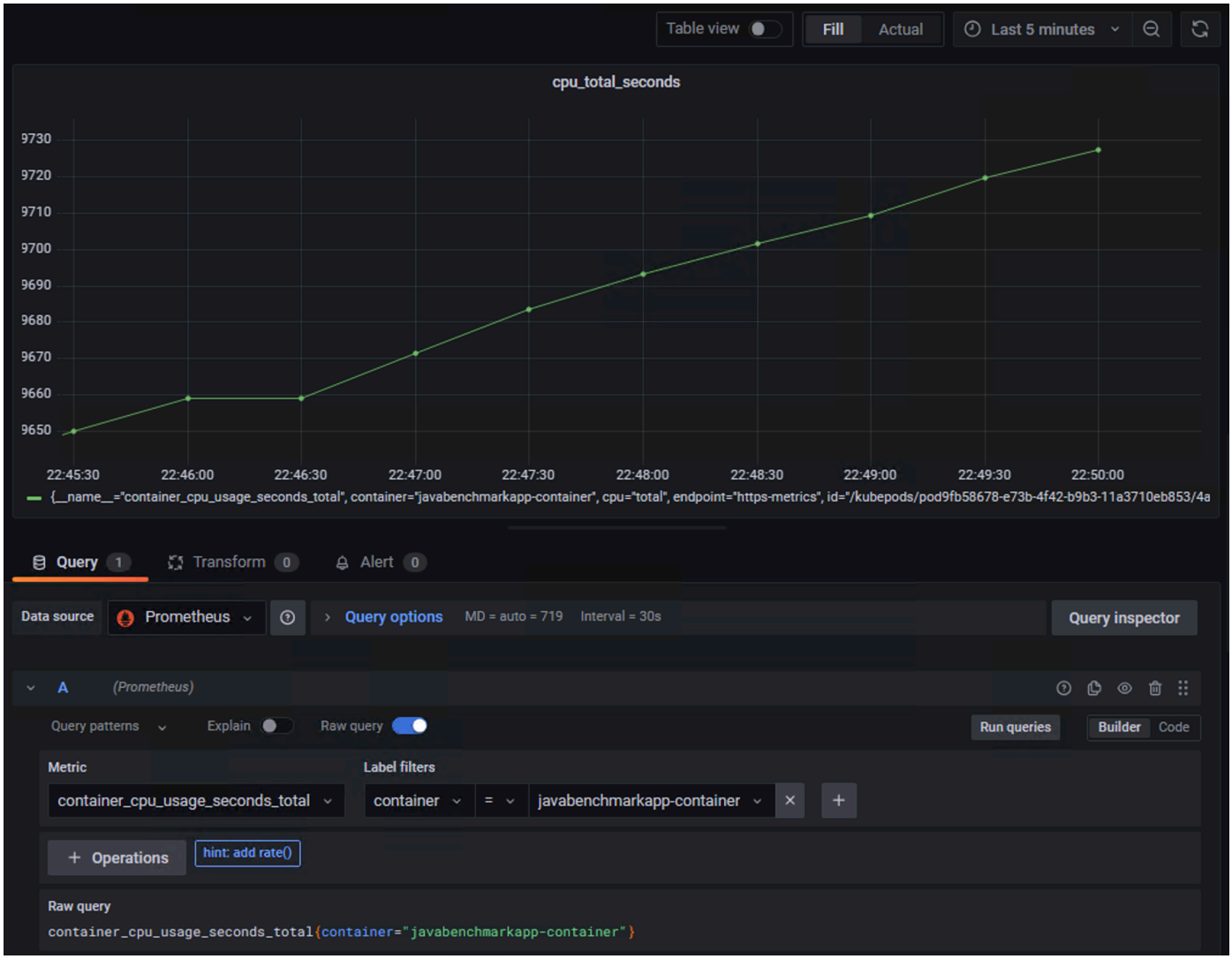
This Prometheus query is used to find the maximum memory usage for each container in a set of containers that meet certain criteria. Here‘s the explaination of each parts of this query:

- max() : This is an aggregation function that finds the maximum value in the result of the expression in parentheses.
- container_memory_usage_bytes{pod=~"javabench.*"}: This is a metric selector that selects time series data that meets certain criteria. Specifically, it selects time series data with the name of the container_memory_usage_bytes metric, where the value of the pod label matches the regular expression javabench.*. This will select all time series data whose pod label starts with "javabench", which typically indicates the memory usage of a container for a Java application.
- by (pod): This is a grouping operation that groups the results by different values of the pod label in order to show the maximum memory usage for each different pod in the result.

The goal of this query is to find the maximum memory usage for each container in a set of containers that satisfy a condition, grouped by each distinct pod value, to obtain the maximum memory usage for the containers in each pod. This is useful for monitoring and analyzing the memory usage of a Java application container. The result is a time series dataset with the maximum memory usage for each pod.

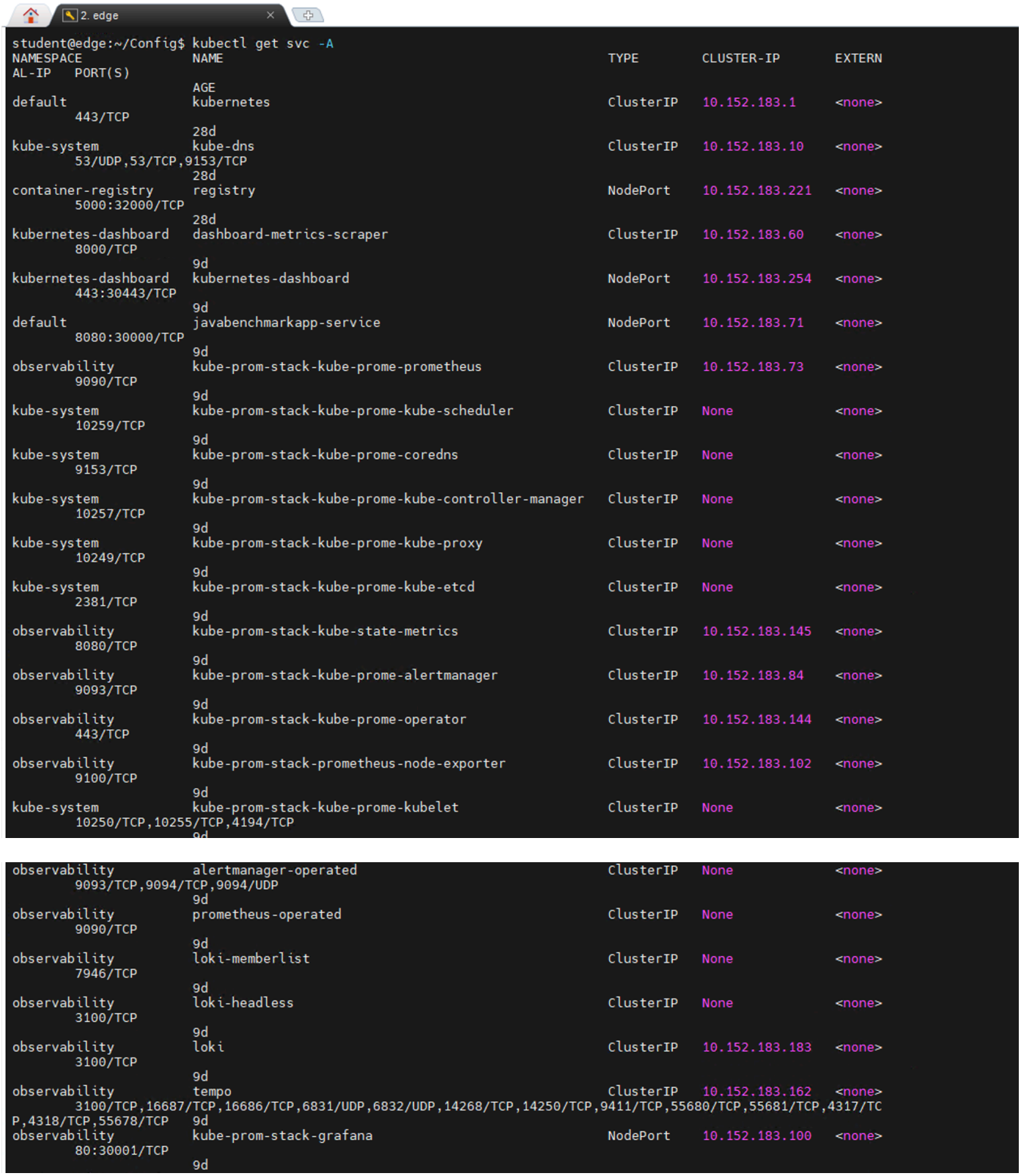
· *panel3: cpu_total_seconds

This panel monitors the cpu usage time (unit: second) , which must be an increasing number. In panel1 on the basis of cpu_total_seconds, calculates the cpu utilization. And I just put screenshots of "cpu_total_seconds" here.



2. Screenshots of running services in Kubernetes

Display a list of all services across all namespaces:

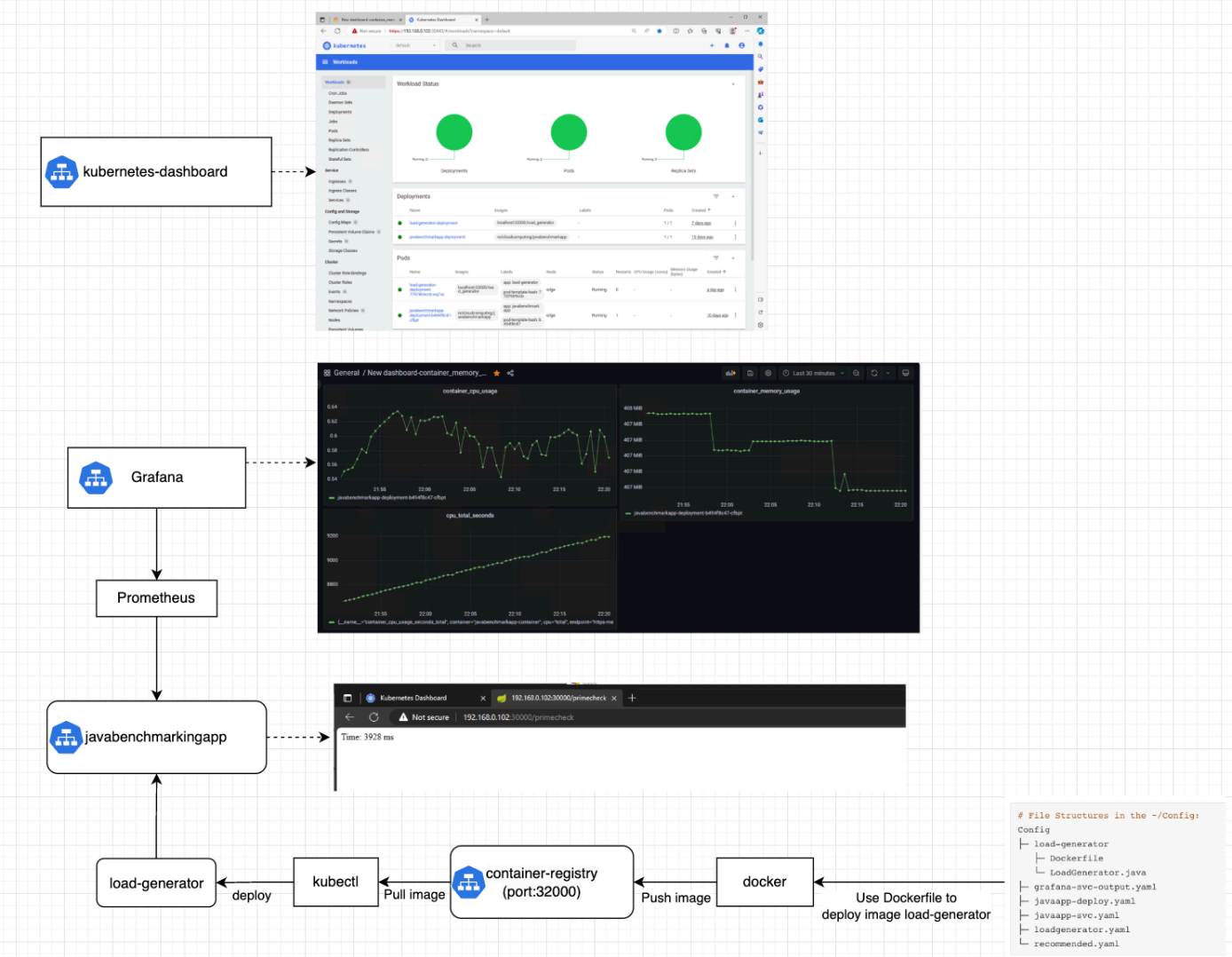


Here are some key Kubernetes services in the list (in the above figure):

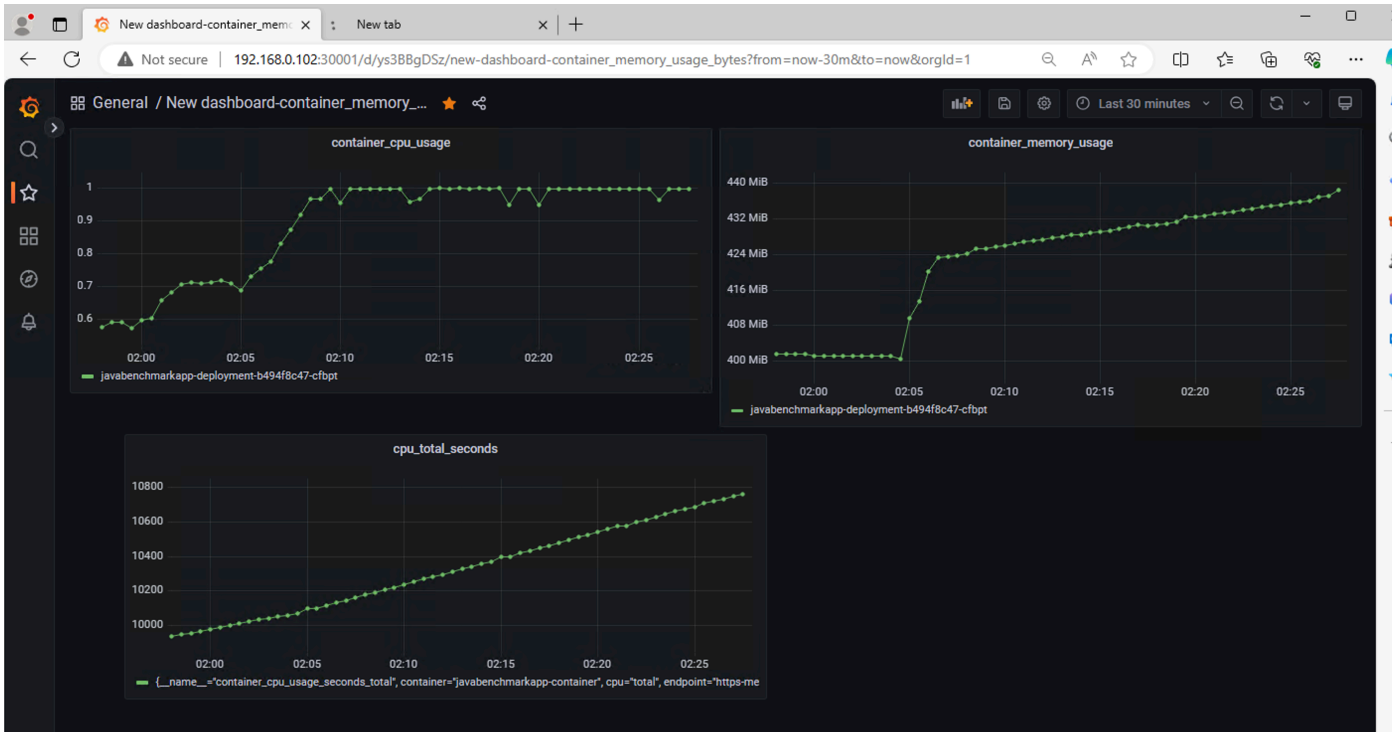
- kubernetes (default namespace)： A default service that provides an entry point for API access.
- registry (Container-registry namespace)： Possibly a private Docker registry service for storing and managing the container images.

- Kubernetes-dashboard (Kubernetes-Dashboard namespace)： Kubernetes' official web user interface.
- javabenchmarkapp-service (default namespace)： the Java application service that uses NodePort to serve.
- kube-prom-stack-kube-prome-prometheus (observability namespace)： Prometheus monitoring tool that collects and stores metric data for the cluster.
- kube-prom-stack-grafana (observability namespace)： Grafana service for visualizing monitoring data.

3. Plots of Benchmarking results



- **Deployment Part**: Docker is used to build the container of the application. Deploy the containers to the cluster via Kubernetes. Kubernetes takes care of managing the container lifecycle, load balancing, auto-scaling, and more.
- **Monitoring Part**: Prometheus integrates with the Kubernetes cluster to collect performance metrics about containers and the entire cluster. Prometheus periodically fetches metric data from configured targets.
- **Data Visualization Part**: Grafana connects to Prometheus, fetches data from it, and visualizes this data.



To test the Javabenchmarkapp load, the number of pods replicates in load-gnerator was adjusted from 1 to 20 at around 02:05. Here is the trend of the data compared with the analysis graph:

1. **CPU Utilization (top left panel)** : At around 02:05, there is a significant increase in CPU utilization. This indicates that increasing the number of copies of the load-generator pods increases the demand for CPU resources because more computing power is required to process more requests. Since is the CPU utilization of the container, we can see that it fluctuates between 0.6 and 1, with 1 representing 100% CPU usage. After increasing the load, the CPU utilization quickly approaches 1, which means that the container's computing resources are fully utilized.
2. **Memory Usage (top right panel)** : Memory usage also shows a sharp rise around 02:05. This indicates that processing more concurrent requests leads to more memory consumption. The memory usage remained relatively stable until the load increased, but then increased rapidly, possibly indicating that the Java application increased its memory

footprint as it processed more load.

3. **Total CPU Usage (bottom panel) :** Total CPU usage is a metric that always cumulatively increases, increasing at a steady rate over time.

Conclusion :

- Increasing the number of copies of load-Generator pods significantly increases the resource usage of the Java application, especially CPU utilization and memory usage, which reflects the direct impact of the increased load.
- An increase in CPU utilization may indicate that the container's resource limit is being approached, which may result in a decrease in processing speed or an increase in request latency.
  -Sharp increases in memory usage require attention, as this can lead to memory overflow errors, especially in memory-limited environments.
- If this resource usage trend continues, you may need to consider scaling resources, optimizing application performance, or placing further limits on concurrent requests to the load-generator.

4. Discussion of the results and related conclusions

The coursework tasks undertaken involved a series of steps in deploying, accessing, and monitoring applications within a Kubernetes environment, using various tools and services. Here are some brief conclusion of each stage.

1. **Check connection and environment Settings** : The initial steps to make sure the environment is ready include checking the mobaXterm connection and Kubernetes cluster connection using 'kubectl'.
2. **Deployment of Kubernetes dashboard and Javabenchmarkapp Web application** : The deployment of Kubernetes dashboard and Java Web application was successfully performed using command line commands. The Kubernetes dashboard provides a user-friendly interface for managing a Kubernetes cluster.

3. **Monitoring Stack deployment** : Activation of Observability services in microk8s, enabling visualization of Kubernetes clusters with Prometheus, metrics server, and Grafana.
4. **Web Application Accessibility and service deployment** : The deployment of NodePort services makes Java Web applications accessible from the outside. This step demonstrates how the Kubernetes service exposes the application to external traffic.
5. **Load Generator Implementation:** Developed in Java and containerized with Docker, the Load Generator plays a vital role in simulating web requests to Java applications. This tool is essential for testing the responsiveness and resilience of an application under load.
6. **Monitoring** :Grafana's integration with Prometheus provides a comprehensive view of application performance, including CPU and memory usage. Creating specific panels in Grafana allows real-time monitoring and analysis of web applications under load. In the "3. Plots of Benchmarking results" section of this document, in order to control the load, the load is increased by increasing the number of pods replicates of the load-generator, and the CPU utilization and memory usage are monitored in Grafana in real time. The changes are obvious in the figure.

Overall observation: Docker for containerization, Kubernetes for orchestration, Prometheus for metric collection, and Grafana for visualization, the combination of these three creates a robust environment for deploying, monitoring, and analyzing web applications. The successful execution of these tasks demonstrates the effectiveness of these tools in managing and monitoring containerized applications in Kubernetes environments.