

Theory Questions

a) Kubernetes Architecture

1. Core components of a Kubernetes cluster:

- **Master Node:**
 - **kube-apiserver:** Serves the Kubernetes API and is the central point for communication between all components.
 - **etcd:** A distributed key-value store used for storing cluster configuration data and state information.
 - **kube-scheduler:** Watches for newly created pods and assigns them to nodes based on available resources and policies.
 - **kube-controller-manager:** Runs controllers that ensure the desired state of the cluster (like replication, scaling, etc.) is met.
- **Node (Worker Node):**
 - **kubelet:** Ensures that containers are running in the pods on the node and communicates with the master node.
 - **kube-proxy:** Manages network routing, ensuring that communication between services and pods is correctly routed.
 - **Container Runtime:** Software like Docker or containerd used to run containers on the node.

2. What is a pod in Kubernetes?

- A **pod** is the smallest deployable unit in Kubernetes and represents a single instance of a running process in the cluster. A pod can contain one or more containers that share the same network namespace and storage. Unlike a Docker container, which is an isolated unit, a pod can host multiple containers that need to work closely together.

b) Deployments and Services

1. Purpose of a Kubernetes Deployment:

- A **deployment** in Kubernetes manages a set of replicas of a pod. It ensures that the desired number of pods are running and allows for easy updates and rollbacks. Deployments ensure high availability by automatically creating new pods to replace failed ones and rolling out updates without downtime.

2. Types of Kubernetes services:

- **ClusterIP:** The default type of service. Exposes the service only inside the cluster. It is used for internal communication between services.
- **NodePort:** Exposes the service on a static port on each node's IP. It allows access from outside the cluster by using the node's external IP address and port.
- **LoadBalancer:** Exposes the service externally and provisions a cloud load balancer (if available). It routes traffic to the appropriate pod, offering a single point of access for external clients.

3. When to use each service type:

- **ClusterIP:** For internal communication between services inside the cluster, where no external access is needed.
- **NodePort:** When you want to expose the service externally and don't need a full-fledged load balancer (for example, for development purposes).
- **LoadBalancer:** When you want to expose the service to the outside world with a highly available load balancer, typically in cloud environments.

c) Scaling and Autoscaling

1. How Kubernetes handles scaling:

- Kubernetes supports manual and automated scaling. The **Horizontal Pod Autoscaler (HPA)** automatically adjusts the number of pods in a deployment based on observed metrics like CPU utilization or custom metrics. When the CPU usage exceeds the defined threshold (e.g., 70%), HPA scales the deployment to handle more load by adding more pods.

Practical Task

a) Create a Deployment

1. Create a Kubernetes deployment that runs 3 replicas of the web server container:

Create a file named `web-server-deployment.yaml`:

```
! web-server-deployment.yaml 1 x {} kindest_node_v1.32.0_sha256_c48c62eac5da28cdadcf560d1d8616cfa6783b58f0d94cf63ad1bf49600cb027.json
! web-server-deployment.yaml > {} spec > {} template > {} spec > [ ] containers > {} 0 > [ ] ports
io.k8s.api.apps.v1.Deployment (v1@deployment.json)
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: web-server-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: web-server
10   template:
11     metadata:
12       labels:
13         app: web-server
14     spec:
15       containers:
16         - name: web-server
17           image: uriel25x/web-server
18           ports:
19             - containerPort: 80
20
```

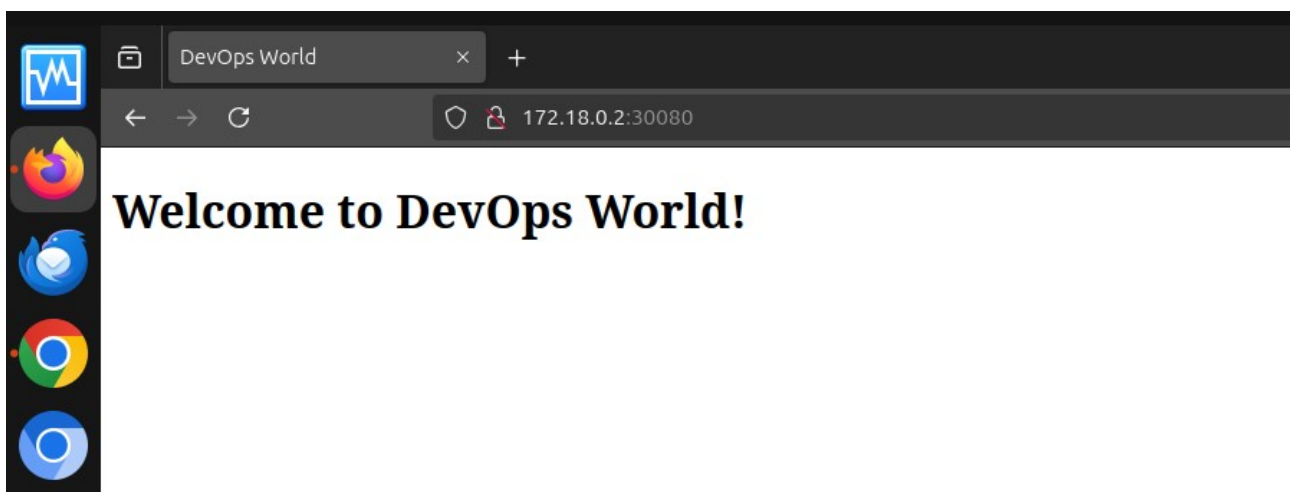
2. Create a ClusterIP service to load balance across the replicas:

Create a file named `web-server-service.yaml`:

```
! web-server-deployment.yaml 1 | web-server-nodeport.yaml x
! web-server-nodeport.yaml > apiVersion
  io.k8s.api.core.v1.Service (v1@service.json)
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: web-server-nodeport
5  spec:
6    selector:
7      app: web-server
8    ports:
9      - protocol: TCP
10        port: 80
11        targetPort: 80
12        nodePort: 30080
13    type: NodePort
14
```

3. Apply the deployment and service:

```
bash
CopyEdit
kubectl apply -f web-server-deployment.yaml
kubectl apply -f web-server-service.yaml
```



Testing load balancing:

- You can test the load balancing by sending multiple requests to the service. For example, use `curl` to send requests and check that different pods are serving them.

```
bash
CopyEdit
curl web-server-service:80
```

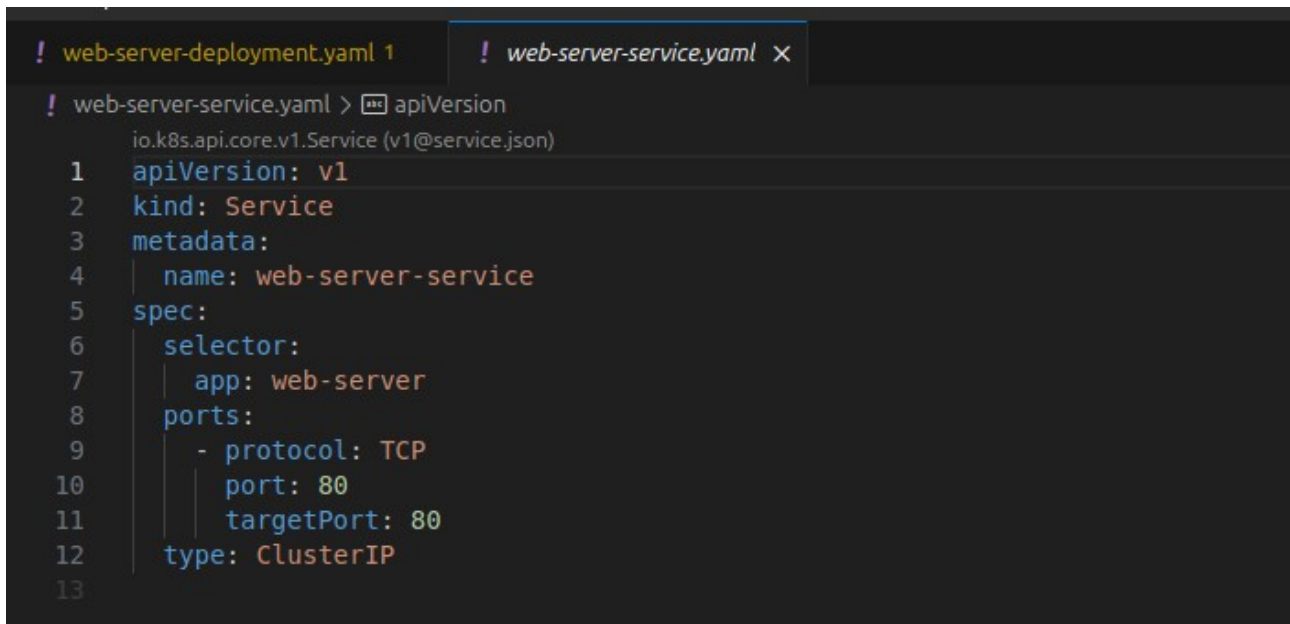
To verify, check the logs of each pod:

```
bash
CopyEdit
kubectl logs <pod-name>
```

b) Service Exposure

1. Expose the deployment via a NodePort service:

Create a file named `web-server-nodeport.yaml`:



```
! web-server-deployment.yaml 1 ! web-server-service.yaml X
! web-server-service.yaml > apiVersion
  io.k8s.api.core.v1.Service (v1@service.json)
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: web-server-service
5  spec:
6    selector:
7      app: web-server
8    ports:
9      - protocol: TCP
10      port: 80
11      targetPort: 80
12    type: ClusterIP
13
```

2. Apply the NodePort service:

```
bash
CopyEdit
kubectl apply -f web-server-nodeport.yaml
```

3. Verify the service:

- Get the external IP of any node in your cluster using `kubectl get nodes -o wide`. This will show the external IP and the port where the service is exposed.
- Access the service using a browser: `http://<node-ip>:30080`.

c) Scaling with Autoscaling

1. Set up the Horizontal Pod Autoscaler (HPA):

Run the following command to create an HPA for your deployment:

```
bash
CopyEdit
kubectl autoscale deployment web-server --cpu-percent=70 --min=3 --max=10
```

This command sets the HPA to scale the deployment when CPU usage exceeds 70%, with a minimum of 3 replicas and a maximum of 10 replicas.

2. Simulate high CPU usage:

Use `kubectl` to run a load test and increase CPU usage on a pod:

```
bash
CopyEdit
kubectl run -i --tty load-generator --image=busybox /bin/sh
stress --cpu 2 --timeout 60s
```

3. Observe scaling behavior:

To monitor the scaling and check the number of replicas, use:

```
bash
CopyEdit
kubectl get hpa
kubectl get pods
```

You should see that the number of pods increases if the CPU usage exceeds 70%.