Example questions will be like this:

*Create a new service account named "sa" in the development namespace. Create a cluster role called "pod-reader," having permission to get pod and list pods. "sa" should be able to get pod and list pods.*

So how would you tackle a question like this?

First, you need to create a namespace called "development" and create a service called "sa" in the development workspace:

kubectl create namespace development

kubecrl create serviceaccount sa -n development

kubectl create clusterrole pod-reader --verb=get,list,watch --resource=pods

kubectl create clusterrolebinding pod-reader --clusterrole=pod-reader --serviceaccount=development:sa

You can test if the sa is allowed to read pods using the below command:

kubectl auth can-i list pods --development target --as system:serviceaccount:development:sa

**How to install and configure a Kubernetes cluster using kubeadm**

Kubeadm automates the installation and configuration of Kubernetes components like Control Manager, API server, and KubeDNS.

If you have time, I highly recommend building a Kubernetes cluster using the [Kubernetes the Hard Way guide](https://github.com/kelseyhightower/kubernetes-the-hard-way) designed by Kelsey Hightower.

If you don't have time to go through the complete guide, from the exam point of view just study certification location and Kubernetes Config Path.

**How to Upgrade Your Kubernetes Cluster Version**

It's very likely you will get this question, as it is specifically mentioned in the exam syllabus.

Below are steps to upgrade the cluster version from 1.22.21 to 1.22.22. You might be also asked to upgrade Kubelet and Kube proxy versions too.

* Check the current version of cluster, kubeadm, and kubelet:

kubectl get nodes -o wide

kubeadm version

kubectl version

* Upgrade the control plane nodes first:

apt-get update && apt-get install -y kubeadm=1.2.22-00

* Verify the upgrade plan. Use the below command to see if the cluster can be upgraded:

kubeadm upgrade plan

* Apply the upgraded version:

sudo kubeadm upgrade apply v1.22.0

Once the command finishes, you should be able to see "upgrade/successful SUCCESS! Your cluster was upgraded to "v1.22.0". Enjoy!"

* Prepare the node for maintenance by marking it unschedulable and evicting the workloads:

kubectl drain node01 --ignore-daemonsets

* Next upgrade the kubelet and kubectl:

apt-get update && apt-get install -y kubelet=1.22.0-00 kubectl=1.22.0-00

* At last restart the kubelet and check if the desired version was upgraded:

sudo systemctl daemon-reload

sudo systemctl restart kubelet

kubectl get nodes -o wide

kubeadm version

kubectl version

* Bring the node back online by marking it schedulable:

kubectl uncordon node01

**How to Backup and Restore an ETCD Cluster**

ETCD is a consistent, distributed key-value store that provides a reliable way to store data that a distributed system or cluster of machines needs to be accessed.

Kubernetes uses etcd to keep all its config and data. You can think of it as a database of Kubernetes. When you run "kubectl get pods", the results are fetched from etcd. In the exam the certification name and path are provided

* Login to the master node and run the below command to backup the etcd:

etcdctl snapshot save /tmp/etcd-backup.db --cacert /etc/kubernetes/pki/etcd/ca.crt --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key

* Test your backup file:

ETCDCTL\_API=3 etcdctl --write-out=table snapshot status snapshotdb

* Restore etcd from a backup file:

ETCDCTL\_API=3 etcdctl snapshot restore tmp/etcd-backup.db --data-dir /var/lib/etcd-backup --cacert /etc/kubernetes/pki/etcd/ca.crt --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key

**Workloads & Scheduling Module**

In this section you will get questions about deploying a Kubernetes application, creating daemonsets, scaling the application, configuring health checks, multi-container pods, and using config maps and secrets in a pod.

**How to deploy an application and expose the app using a service**

Example questions for deploying an app and creating a service might look like this:

*Create a deployment as follows:  
Name: nginx  
Exposed via a service nginx using CluserIP, Ensure that the service & pod are accessible within the cluster*

* Manifesto file for creating deployment:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

Run kubectl get deployments to check if the Deployment was created. If the deployment is sucessfull "Ready" should show 3/3. Ready displays how many replicas of the application are available to your users.

If you need to expose your application outside the cluster or inside the cluster, you need to create a service. There are different options available to expose your application as per your needs.

* ClusterIP: Exposes application within cluster. For example exposing a database to a backend application.
* NodePort: Exposes application outside the cluster using node ip. For exmaple exposing your frontend application to the outside world.
* LoadBalancer: Exposes application outside the cluster using a load balancer.

How about an example of exposing your application using ClusterIP (within the cluster). You can create a service using the below manifesto file:

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

selector:

app: nginx

type: ClusterIP

ports:

- protocol: TCP

port: 80

targetPort: 8080

You can use "kubectl get service" to see the IP address of the service.

Here's an example of exposing your application using NodePort (outside the cluster). You can create a service using the below manifesto file:

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

selector:

app: nginx

type: NodePort

ports:

- protocol: TCP

port: 80

targetPort: 8080

You can use "kubectl get service" to see the IP address of the node.

Another sample question will be like this:

*Schedule the pod on a node labeled with distype=ssd*

Here you can use node-selector like this:

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

env: test

spec:

containers:

- name: nginx

image: nginx

nodeSelector:

disktype: ssd

**How to scale and update the deployments**

If you need to scale the deployment after creating it, you can use the below command.

kubectl scale deployment/nginx-deployment --replicas=6

You can update the image of the existing deployment using the below command:

kubectl set image deployment/nginx-deployment nginx=nginx:1.8

**How to configure healthcecks for your application**

Once your application is deployed, you need to make sure that the app is running successfully. If an application crashes, you need to know how you can kill the container and bring in the new one.

Health checks help to achieve this use case. There the three different types of health checks you can perform:

* Readiness Probe: Kubernates uses readiness probes to know when a container is ready to start accepting traffic.
* Liveness Probe: Kubernates uses liveness probes to check when to restart a container. Once the application is deployed succesfully, if it crashes in between, a liveness probe will detect and restart the application.
* Startup Probe: Kubernates uses startup probes to know when a container application has started.

Example of configuring a livenss probe:

kubectl apply -f https://k8s.io/examples/pods/probe/exec-liveness.yaml

Example of configuring a readines probe:

apiVersion: v1

kind: Pod

metadata:

labels:

test: liveness

name: liveness-exec

spec:

containers:

- name: liveness

image: k8s.gcr.io/busybox

args:

- /bin/sh

- -c

- touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600

livenessProbe:

exec:

command:

- cat

- /tmp/healthy

initialDelaySeconds: 5

periodSeconds: 5

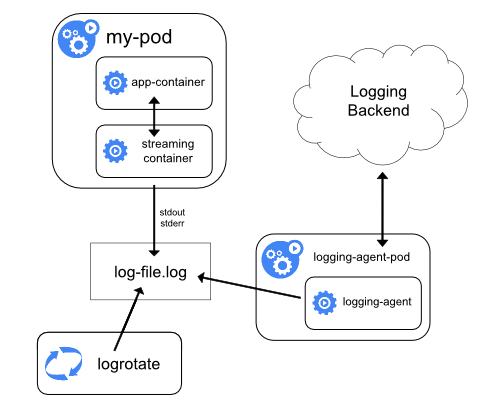
**MultiContainer pod/sidecar containers**

The primary purpose of a multi-container pod is to support a co-located helper container for the main program.

The standard logging method for containerized applications is writing to standard output and standard error streams.

There might be use cases where you also need to access these logs after a container crashes. For example, your NGINX designed for serving the web page is not suitable for shipping the logs to a centralized logging solution.

You can set up a sidecar container that specialises in log shipping. The sidecar container is designed as a logging agent, which is configured to pick up logs from an application container.



Example questions about this topic will be like this:

*Create a Pod with the main container NGINX, which outputs logs to shared volume, and configure the sidecar container to stream those logs. Verify both containers are running.*

apiVersion: v1

kind: Pod

metadata:

name: nginx-server

spec:

volumes:

- name: shared-logs

emptyDir: {}

containers:

- name: nginx

image: nginx

volumeMounts:

- name: shared-logs

mountPath: /var/log/nginx

- name: sidecar-container

image: busybox

command: ["sh","-c","while true; do cat /var/log/nginx/access.log /var/log/nginx/error.log; sleep 30; done"]

volumeMounts:

- name: shared-logs

mountPath: /var/log/nginx

**How to configure a pod to use a ConfigMap**

ConfigMaps store data in key-value format. A possible usecase of ConfigMaps is keeping application code and configuration separate.

ConfigMaps are designed to store non-confidential data such as environment variables or properties of a game or application. If you want to store sensitive data, use secrets.

ConfigMaps help create separate config files for each environment (development, staging, prod).

You can create ConfigMaps from files, directories, and literal values. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

Example questions will be like this:

*Create a ConfigMap called cfg-data with values var1=val1,var2=val2 and create a busybox pod with volume config-volume, which reads data from this ConfigMap cfg-volume and put it on the path  
/etc/cfg*

kubectl create configmap cfg-data --from-literal=key1=val1 --from-literal=key2=val2 --from-literal=key3=val3

kubectl create -f https://github.com/nitheesh86/cka/blob/main/deployments-services/configmap.yml

**How to configure a pod to use secrets**

Secrets in Kubernetes can be used to store sensitive data such as passwords and tokens. Secrets are similar to ConfigMaps but are specifically designed to hold sensitive data. Pods can use secrets as an environment variable or as files in a volume.

* Example questions about secrets will be like this:

*Create a secret named "db-secret" in namespace database. The secret should contain db\_user=root and pass=1234. Mount it as ready only into the pod named "mysql-db" as an enviournment variable.*

kubectl create namespace database

kubectl -n secret create secret generic db-secret --from-literal=username=db\_user --from-literal=db\_pass=1234 -n database

https://github.com/nitheesh86/cka/blob/main/deployments-services/mysql-secret.yml

**Services and Networking Module**

In this section you will get questions about creating networking polices, creating ingress controllers, and exposing apps through services (already covered in above)

**How to creating networking policies**

In Kubernetes, by default, communication between all pods is allowed. If you need to isolate pods, you need to apply a network policy.

Example questions about netowrking policies will be like this:

Allow traffic from production namespace only:

kind: NetworkPolicy

apiVersion: networking.k8s.io/v1

metadata:

name: allow-traffic-from-namespace

spec:

podSelector:

matchLabels:

ingress:

- from:

- namespaceSelector:

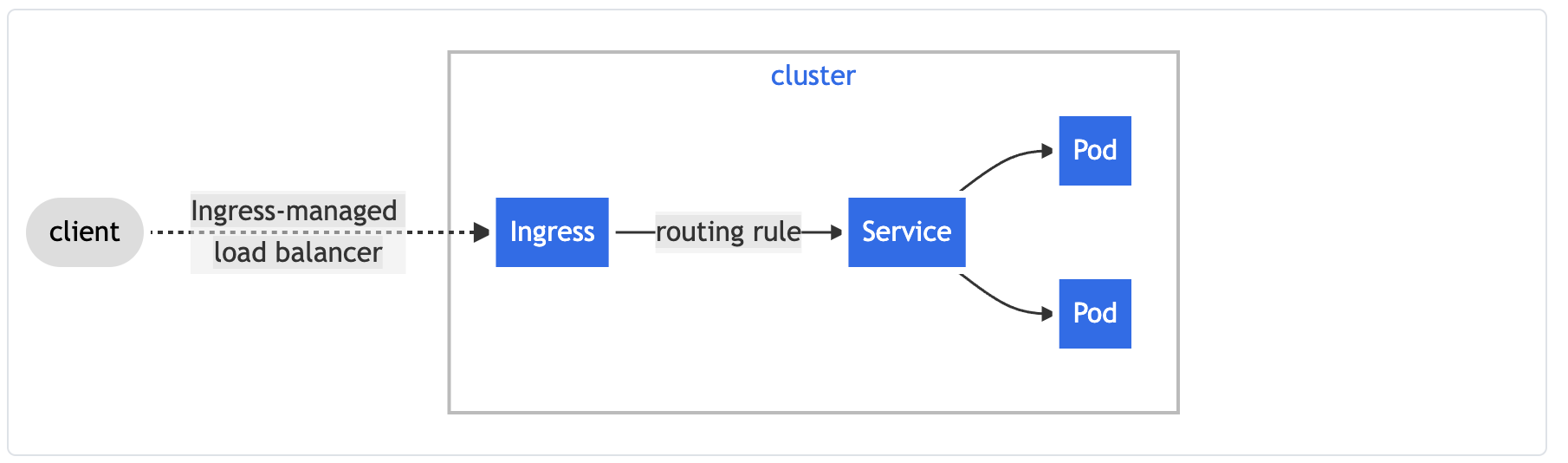
matchLabels:

purpose: production

This policy will allow traffic to all pods from the production namespace.

**How to create an ingress resource**

An ingress controller is a type of load balancer. It accepts traffic from outside the cluster and loads the balance to pods. We can also configure rules like redirections.



* How to create an ingress using the NGINX ingress controller:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: nginx-ngress

annotations:

nginx.ingress.kubernetes.io/rewrite-target: /

spec:

rules:

- http:

paths:

- path: /example

pathType: Prefix

backend:

service:

name: nginx-service

port:

number: 80

**Storage Module**

This section is all about creating persistence volume, persistence volume claims, and mounting into to a pod. It's helpful to study a lot about persistence and persistence volume.

A PersistentVolume (PV) is a storage in the cluster that has been provisioned by a stroage administrator or dynamically provisioned using Storage Classes like AWSElasticBlockStore, AzureDisk, and so on.

A PersistentVolumeClaim (PVC) is a request for storage by a user or Pod.

Example questions will be like this:

*Create an NGINX Pod and mount index.html as PersistentVolume.*

* ssh into the node and create a /mnt/data directory and then create an index.html file:

sudo mkdir /mnt/data

sudo sh -c "echo 'Hello from Kubernetes storage' > /mnt/data/index.htm

* Create PersistentVolume and PersistentVolume Claim:

apiVersion: v1

kind: PersistentVolume

metadata:

name: task-pv-volume

labels:

type: local

spec:

storageClassName: manual

capacity:

storage: 10Gi

accessModes:

- ReadWriteOnce

hostPath:

path: "/mnt/data"

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: task-pv-claim

spec:

storageClassName: manual

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 3Gi

* Now, the config pod uses the PersistentVolume Claim

apiVersion: v1

kind: Pod

metadata:

name: task-pv-pod

spec:

volumes:

- name: task-pv-storage

persistentVolumeClaim:

claimName: task-pv-claim

containers:

- name: task-pv-container

image: nginx

ports:

- containerPort: 80

name: "http-server"

volumeMounts:

- mountPath: "/usr/share/nginx/html"

name: task-pv-storage

Also, in the exam, you might be asked to expand PersistentVolume. Some storage classes support resizing the volume—for example, AWS-EBS, GCE-PD, Azure Disk, Azure File, Glusterfs.

If the storage class is not enabled, you need to set it to "allowVolumeExpansion: true."

Get the storage class name you want to expand with "kubectl get storage classes". Then edit the YAML file.

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: standard

parameters:

type: pd-standard

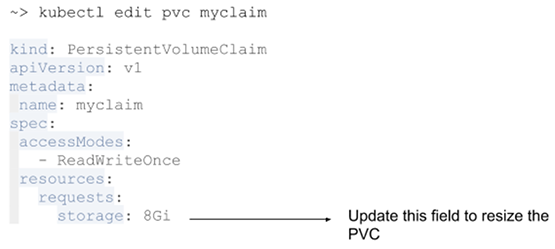
provisioner: kubernetes.io/gce-pd

allowVolumeExpansion: true

reclaimPolicy: Delete

* Then edit PVC to request more space:

kubectl edit pvc myclaim and update request parameter



* Once PVC is updated, you need to replace the pod to change to effect. You can check the new size by "kubectl get pvc. myclaim".

Kubespray is an Ansible-based Kubernetes provisioner. It helps us to set up a production-grade, highly available, and highly scalable Kubernetes cluster.

* Prerequisites

Hardware Prerequisites

4 Nodes: Virtual/Physical Machines

Memory: 2GB

CPU: 1 Core

Hard disk: 20GB available

Software Pre-Requisites

* On All Nodes

Ubuntu 16.04 OS

Python

SSH Server

Privileged user

* On Ansible Control Node

Ansible version 2.4 or greater

Jinja

* Networking Pre-Requisites

Internet access to download docker images and install software.

IPv4 Forwarding should be enabled.

The firewall should allow ssh access as well as ports required by Kubernetes. Internally open all the ports between nodes.

* Ingress controller such as Nginx, and Trafeik needs to be deployed before creating ingress resources.
* On GCE, the ingress controller runs on the master. On all other installations, it needs to be deployed, either as a deployment, or a daemonset. In addition, a service needs to be created for ingress.
* Daemonset will run ingress on each node. The deployment will just create a highly available setup, which can then be exposed on specific nodes using ExternalIPs configuration in the service.

**Steps involved to complete this task:**

1. Create an Ingress Controller
2. Setting up Named-Based Routing for Vote App
3. Adding Local DNS
4. Adding HTTP Authentication with Annotations
5. Creating htpasswd spec as Secret