**KUBERNETES INTERVIEW PREP**

1. **Architecture and Components**
   1. **Explain the architecture of Kubernetes.**

Kubernetes is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Its architecture is built around a master-worker model (now more commonly referred to as control plane and nodes), and it follows a declarative configuration and desired state management paradigm.

**+----------------------------+**

**| kubectl CLI |**

**+----------------------------+**

**|**

**v**

**+----------------------------+**

**| Control Plane | <-- Brain of Kubernetes**

**+----------------------------+**

**| API Server | Scheduler | Controller Manager | etcd |**

**+----------------------------+**

**|**

**v**

**+----------------------------+**

**| Nodes | <-- Workers that run workloads**

**+----------------------------+**

**| Kubelet | Kube Proxy | Container Runtime |**

**+----------------------------+**

**|**

**v**

**+----------------------------+**

**| Pods (Containers) |**

**+----------------------------+**

* + 1. **Control Plane Components**

The Control Plane manages the cluster and makes global decisions (e.g., scheduling), as well as detecting and responding to cluster events.

**📌 a. kube-apiserver**

* **Front-end of the Kubernetes control plane**
* All external and internal components interact via the API server (usually over HTTPS)
* Validates and processes REST requests
* Acts as the communication hub for all components

**📌 b. etcd**

* Key-value store for storing all cluster data
* Stores configuration data, state, secrets, and metadata
* Highly available and consistent (based on the Raft consensus algorithm)

**📌 c. kube-scheduler**

* Assigns Pods to Nodes
* Evaluates available resources and constraints
* Considers:
  + CPU and memory availability
  + Affinity/anti-affinity rules
  + Node taints and tolerations
  + Pod priorities

**📌 d. kube-controller-manager**

* Runs controller processes:
  + Node Controller: Watches node health
  + Replication Controller: Ensures desired number of pods
  + Endpoints Controller: Populates endpoints in Services
  + Job Controller, DaemonSet Controller, etc.
    1. **Node Components**

**📌 a. kubelet**

* Agent that runs on every node
* Registers the node with the API server
* Ensures containers are running in pods
* Communicates with the container runtime

**📌 b. kube-proxy**

* Maintains network rules on nodes
* Forwards traffic to the correct pod using IP tables or IPVS
* Handles networking for services

**📌 c. Container Runtime**

* Software responsible for running containers
* Kubernetes supports:
  + containerd
  + CRI-O
  + Docker (deprecated in latest versions)
  1. **Kubernetes Objects**

These are persistent entities in the Kubernetes system representing the desired state of the cluster.

* **Pod**: The smallest deployable unit; encapsulates one or more containers
* **Service**: An abstraction to expose a group of pods as a network service
* **Deployment**: Manages replica sets and rolling updates
* **ReplicaSet**: Ensures a specified number of pod replicas are running
* **StatefulSet**: Like Deployment but for stateful apps
* **DaemonSet**: Ensures a pod runs on all (or some) nodes
* **Job/CronJob**: Batch or scheduled jobs

**⚙️ How Kubernetes Works (Example Workflow)**

1. You create a Deployment using kubectl apply -f deployment.yaml
2. kubectl sends the request to the **API Server**
3. API Server validates and writes the desired state to **etcd**
4. **Controller Manager** detects a difference between desired and current state and creates a **Pod**
5. **Scheduler** assigns the Pod to a **Node**
6. **Kubelet** on that node pulls the image via **Container Runtime** and starts the container
7. **Kube Proxy** sets up networking so the pod can communicate
8. Kubernetes continuously monitors and reconciles the state
9. **Networking and Services**
   1. **How does service discovery work in Kubernetes?**

A Service in Kubernetes is an abstraction that defines a logical set of Pods (usually defined by a label selector) and a policy to access them.

Services expose Pods internally within the cluster or externally via various types (ClusterIP, NodePort, LoadBalancer, etc.).

Each Service gets:

* A **stable DNS name**
* A **stable virtual IP (ClusterIP)**
* **Port mapping** for accessing application ports

✅ **B. DNS-Based Service Discovery (Recommended and Default)**

Kubernetes clusters run a **DNS service** (usually **CoreDNS**) that automatically creates DNS records for each Service.

**🧠 How It Works:**

1. You create a Service named my-service in namespace my-namespace.
2. Kubernetes automatically creates a DNS entry:
   * my-service.my-namespace.svc.cluster.local

**🔁 DNS Resolution Flow:**

1. A pod tries to connect to my-service.
2. CoreDNS resolves the name to the Service’s **ClusterIP**.
3. kube-proxy redirects the request to one of the healthy backend Pods (round-robin).

**⚙️ 3. Role of kube-proxy in Service Discovery**

* **kube-proxy** runs on every node.
* It watches for changes to Services and Endpoints via the API server.
* It configures iptables (or IPVS) rules to forward traffic from the **Service IP/port** to the actual **Pod IP/port**.

**Service Discovery and Network Policies**

* You can restrict communication between Pods using **Network Policies**.
* Even though a Pod can discover another Pod via Service DNS, a NetworkPolicy can **block the traffic**.
  1. **What is ClusterIP vs NodePort vs LoadBalancer?**

In Kubernetes, **Services** are used to expose Pods to other parts of your cluster or to the outside world. Kubernetes offers different **Service types** depending on how and where you want your application to be accessible.

**ClusterIP (Default)** 🡪 Exposes the Service **internally within the cluster** using a **virtual IP (ClusterIP)**.

**How It Works:**

* The Service gets a virtual IP (e.g., 10.96.0.1)
* Other pods in the cluster can access it using this IP or its DNS name.

**Use Case:**

* Internal communication between services (e.g., frontend → backend)
* Ideal for microservices inside the same cluster

**🔹 Access:**

* ❌ Not accessible from outside the cluster
* ✅ Accessible from other Pods using DNS or ClusterIP

spec:

type: ClusterIP

selector:

app: my-backend

ports:

- port: 80

targetPort: 8080

**NodePort** 🡪 Exposes the Service **on a static port on each Node’s IP address**

**🔹 How It Works:**

* A port in the range **30000–32767** is allocated on all nodes
* Incoming traffic to this port is forwarded to the underlying Pods

**🔹 Use Case:**

* Useful for development or basic external access without a cloud provider
* Load balancers or reverse proxies can use NodePort to route traffic

**🔹 Access:**

* ✅ Accessible from **outside the cluster** using <NodeIP>:<NodePort>
* ✅ Also accessible inside the cluster via ClusterIP

**🔹 Example:**

**spec:**

**type: NodePort**

**selector:**

**app: my-backend**

**ports:**

**- port: 80**

**targetPort: 8080**

**nodePort: 30080 # Optional, can be auto-assigned**

**LoadBalancer🡪 Exposes the Service externally using a cloud provider’s external load balancer**

**🔹 How It Works:**

* **Only available in cloud environments (AWS, GCP, Azure, etc.)**
* **Kubernetes asks the cloud provider to create a load balancer and attach it to the Service**

**🔹 Use Case:**

* **Production workloads needing direct external access**
* **Common for APIs, web apps, etc.**

**🔹 Access:**

* **✅ Accessible from outside the cluster via an external IP or DNS [Cloud assigned]**
* **✅ Internally accessible too**

**spec:**

**type: LoadBalancer**

**selector:**

**app: my-backend**

**ports:**

**- port: 80**

**targetPort: 8080**

**🧠 When to Use What?**

* 🧩 **ClusterIP** → Internal services like databases, internal APIs
* 🌐 **NodePort** → Testing or when you have your own load balancer (e.g., nginx or HAProxy)
* ☁️ **LoadBalancer** → Production apps needing external access (on cloud platforms)
  1. **What is Ingress and how is it different from a Service?**

**What is a Kubernetes Service?**

* **Service** exposes one or more Pods internally or externally.
* It provides a **stable IP address (ClusterIP)** and DNS name for a set of Pods.
* It operates mainly at **Layer 4** (TCP/UDP).
* Types: **ClusterIP**, **NodePort**, **LoadBalancer** (which we discussed earlier).

**Key points:**

* Routes traffic to Pods.
* Service IP or NodePort is how clients reach your app.
* Can expose apps internally or externally (depending on type).

**🌐 What is a Kubernetes Ingress?**

* **Ingress** manages **external HTTP/S access** to services.
* It provides **Layer 7 (application layer) routing** capabilities.
* Acts as a **smart router** or **reverse proxy** for multiple services.

**What Ingress does:**

* Exposes HTTP and HTTPS routes from outside the cluster to services inside the cluster.
* Allows routing traffic based on:
  + Hostnames (e.g., foo.example.com, bar.example.com)
  + URL paths (e.g., /api, /web)
* Supports SSL/TLS termination.
* Supports load balancing, name-based virtual hosting, and path-based routing.

**🔍 How does Ingress work?**

* You create an **Ingress resource** that defines rules mapping hosts/paths to Services.
* You deploy an **Ingress Controller**, which is a pod (or set of pods) that implements these rules.
* The Ingress Controller usually runs a reverse proxy/load balancer (e.g., **NGINX**, **Traefik**, **Contour**).
* Incoming requests hit the Ingress Controller, which routes requests to the appropriate Service based on the rules.
  1. **How do network policies work?**

In Kubernetes, Network Policies are used to control the communication between pods and namespaces at the network level. They define rules for ingress (incoming) and egress (outgoing) traffic to/from pods, based on labels and namespaces.

By default, all pods can communicate with all other pods in a cluster. Once you apply a NetworkPolicy to a pod, only traffic allowed by that policy is permitted — all other traffic is blocked.

🔧 How Network Policies Work

A NetworkPolicy is a Kubernetes resource that:

Applies to a set of pods selected by labels (podSelector)

Specifies allowed sources (for ingress) or destinations (for egress)

Optionally filters traffic by port and protocol

Requires a network plugin (like Calico, Cilium, or Weave Net) that supports NetworkPolicy

**🧪 Example Scenario**

Let's say you have:

* A frontend pod (label: app=frontend)
* A backend pod (label: app=backend)
* You want to **allow only frontend to talk to backend on port 80**, and **block all other traffic**.
  1. Pods Deployment

apiVersion: v1

kind: Pod

metadata:

name: backend

labels:

app: backend

spec:

containers:

- name: backend

image: nginx

---

apiVersion: v1

kind: Pod

metadata:

name: frontend

labels:

app: frontend

spec:

containers:

- name: frontend

image: curlimages/curl

command: ["sleep", "3600"]

* 1. NetworkPolicy to Allow Only Frontend to Access Backend

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-frontend-to-backend

spec:

podSelector:

matchLabels:

app: backend # Applies policy to the backend pod

policyTypes:

- Ingress

ingress:

- from:

- podSelector:

matchLabels:

app: frontend # Only allow frontend pods

ports:

- protocol: TCP

port: 80 # Only on port 80

**🧠 What This Means**

* This policy **applies to the backend pod**
* It **only allows incoming TCP traffic on port 80**
* And only **from pods with label app=frontend**

🚫 All other traffic to the backend pod will be **blocked** — including from other pods or namespaces unless explicitly allowed by another policy.

* 1. How do you expose a service to external traffic?

**✅ 1. NodePort**

* **How it works:** Opens a specific port on all nodes in your cluster, and forwards traffic to the service.
* **Use case:** Good for simple setups or testing.

**✅ 2. LoadBalancer**

* **How it works:** Provisions an external load balancer (like AWS ELB, GCP LB) and exposes the service to the internet.
* **Use case:** Recommended for production on cloud platforms.

**✅ 3. Ingress**

* **How it works:** Routes external HTTP/HTTPS traffic to services based on host/path rules. Requires an Ingress Controller (e.g., NGINX, Traefik).
* **Use case:** Best for managing multiple HTTP routes with a single external IP.

**✅ 4. Port Forwarding (kubectl port-forward)**

* **Temporary access** from your local machine for testing. Not meant for production.

kubectl port-forward service/my-service 8080:80

Access via: <http://localhost:8080>

* 1. **Pod Networking**

**3.1** Flat / Native Kubernetes Networking (Pod-to-Pod via CNI)

* Every pod gets its **own unique IP**.
* All pods can reach each other without NAT, across nodes.
* Achieved using CNI plugins (Calico, Flannel, Weave, Cilium).

+---------+ +---------+

| Pod A | <----> | Pod B | (Same Node)

| 10.0.1.2| | 10.0.1.3|

+---------+ +---------+

Node1 (10.0.1.0/24 subnet)

|

-----------------

|

Node2 (10.0.2.0/24 subnet)

+---------+ +---------+

| Pod C | <----> | Pod D |

| 10.0.2.2| | 10.0.2.3|

+---------+ +---------+

✔ Pod-to-Pod across nodes without NAT

3.2 **Overlay Network**

* Pods communicate through an **encapsulated network** (VXLAN, IP-in-IP).
* Used by **Flannel (VXLAN mode)**, Weave Net, etc.
* Hides underlying infrastructure differences.

Pod A (10.244.1.2) --VXLAN--> Node1 Overlay

|

Encapsulated traffic

|

Node2 Overlay --VXLAN--> Pod B (10.244.2.3)

✔ Works across clouds / mixed infra

❌ Slight performance overhead due to encapsulation

**3.3 Routing-Based Network**

* **No encapsulation. Instead, routes are programmed into nodes.**
* **Calico (BGP or IP-in-IP) works this way.**
* **Each node knows how to reach every pod subnet.**

Node1 Pod CIDR: 10.244.1.0/24

Node2 Pod CIDR: 10.244.2.0/24

Pod A (10.244.1.2) ---[Routing Table]---> Pod B (10.244.2.3)

✔ Faster (no VXLAN overhead)

❌ Requires routing setup (BGP, IP pools, etc.)

**3.4 Host-Network Pods**

* Pods share the **host network namespace**.
* No separate pod IP; they use the node’s IP directly

Pod A (--net=host) → Uses Node1 IP: 192.168.1.10

Pod B (--net=host) → Uses Node1 IP: 192.168.1.10

✔ Low latency, useful for system daemons (DNS, monitoring)

❌ No pod isolation, IP conflicts possible

* 1. **Storage and Volumes**

**4.1 What is a Volume in Kubernetes?**

In **Kubernetes**, a **Volume** is a way to **persist and share data** between containers in a Pod, and sometimes even across Pod restarts depending on the volume type.

**📦 What is a Volume (Conceptually)?**

* A **volume** in Kubernetes is just a **directory** that is accessible to containers in a Pod.
* It's **defined at the Pod level** and **mounted into containers**.
* The data in the volume **persists** for the **lifetime of the Pod**, unless you're using persistent volumes (see below).

**🔍 Why Do We Need Volumes?**

By default, **containers are ephemeral** — their filesystem is **wiped when the container crashes or restarts**. This is where Volumes come in:

* ✅ Persist data across container restarts (in a Pod)
* ✅ Share data between multiple containers in the same Pod
* ✅ Mount external storage systems (e.g., NFS, cloud disks)

**4.2 What is the difference between emptyDir, hostPath, and persistentVolume?**

**📦 1. emptyDir**

| **Feature** | **Details** |
| --- | --- |
| **What it is** | A temporary, empty directory created **when a Pod starts**. |
| **Data persistence** | **Data is lost** when the Pod is deleted or restarted. |
| **Use case** | Temporary storage or sharing data between containers in a Pod (e.g., cache, scratch space). |
| **Where it's stored** | On the **node’s disk**, in a temporary location. |

volumes:

- name: temp-storage

emptyDir: {}

**🖥️ 2. hostPath**

| **Feature** | **Details** |
| --- | --- |
| **What it is** | Mounts a file or directory from the **host node's filesystem** into the Pod. |
| **Data persistence** | Data persists as long as it's on the **host** (node). If the Pod moves to another node, the data is **not available**. |
| **Use case** | Accessing node-local files, logging to a shared location, or using tools installed on the host. |
| **Security concern** | **Can be dangerous**, because it exposes the host filesystem to containers. Use with care. |

volumes:

- name: host-volume

hostPath:

path: /data/host

type: Directory

**💾 3. persistentVolume (used via PVC)**

| **Feature** | **Details** |
| --- | --- |
| **What it is** | An abstraction for **external, durable storage** (cloud disks, NFS, iSCSI, etc.). |
| **Data persistence** | **Persists independently** of Pod or even Node lifecycles. |
| **Use case** | Databases, user uploads, any app that needs **durable storage**. |
| **Where it's stored** | Backed by cloud storage (e.g., AWS EBS, GCE PD), NFS, or other storage providers. |

volumes:

- name: my-storage

persistentVolumeClaim:

claimName: my-pvc

4.3 What is a PersistentVolume (PV) and PersistentVolumeClaim (PVC)?

A **PersistentVolume (PV)** is a **storage resource** in the cluster. It represents a piece of **real storage** — from cloud providers (like AWS EBS, GCP PD), NFS, iSCSI, or even a local disk — that has been provisioned by an admin or dynamically via a **StorageClass**.

📥 What is a PersistentVolumeClaim (PVC)?

A PersistentVolumeClaim (PVC) is a request for storage by a user (typically from a Pod). It specifies things like:

* How much storage is needed
* What access mode (read/write)
* Which storage class (optional)

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: my-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 5Gi

**🔄 How PV & PVC Work Together**

1. **Admin or dynamic provisioner** creates a **PV**.
2. An app (Pod) creates a **PVC** requesting storage.
3. Kubernetes **matches the PVC with a PV** that satisfies the request.
4. PVC is **bound** to that PV.
5. Pod uses the **PVC** to mount the storage.

| **Mode** | **Description** |
| --- | --- |
| ReadWriteOnce | One node can mount the volume for read/write |
| ReadOnlyMany | Multiple nodes can mount, read-only |
| ReadWriteMany | Multiple nodes can mount, read/write |

**4.4 How does dynamic volume provisioning work?**

**Dynamic volume provisioning** in Kubernetes allows the cluster to **automatically create storage volumes on demand** — no need for a cluster admin to pre-create PersistentVolumes (PVs).

**🧠 Why Use Dynamic Provisioning?**

* Without dynamic provisioning:
  + An admin must **manually create a PV** before an app can use it.
* With dynamic provisioning:
  + Kubernetes **automatically provisions a PV** when a **PVC is created**, using a StorageClass.

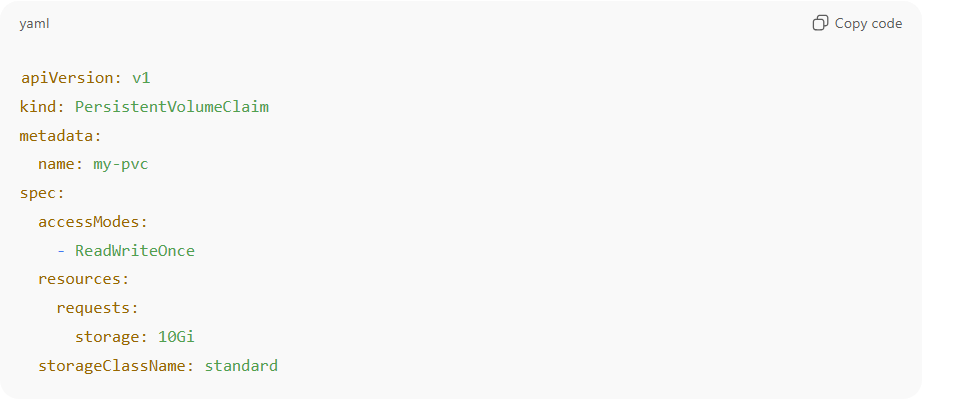
🔁 How It Works (Step-by-Step)

* PVC is created by a user (usually via a deployment or a pod).
* The PVC references a StorageClass.
* The StorageClass defines:
* Provisioner (e.g., AWS EBS, GCE PD, NFS)
* Parameters like disk type, size, etc.
* The Kubernetes controller:
* Talks to the cloud provider or storage backend
* **Creates a PersistentVolume**
* Binds the new PV to the PVC automatically
* The PVC is now ready to be used by the Pod.

A screenshot of a computer

AI-generated content may be incorrect.

📥 Example: PVC Using That StorageClass



**4.5 How does volumeBindingMode: WaitForFirstConsumer affect volume provisioning and Pod scheduling?**

volumeBindingMode: WaitForFirstConsumer delays the provisioning and binding of a PersistentVolume (PV) until a Pod that uses the PVC is scheduled. This is particularly useful in multi-zone or multi-availability-zone environments. It ensures that the volume is created in the **same zone as the Pod**, which prevents cross-zone mount failures and avoids unnecessary data transfer costs.

For example, in AWS, if EBS volumes are created **before** a Pod is scheduled, they might end up in a different zone than the node, and can't be attached. With WaitForFirstConsumer, Kubernetes waits until a Pod is scheduled to a specific node, then provisions the volume in that node's zone.

This setting is common in **dynamic provisioning** scenarios, especially with cloud CSI drivers.

**4.6 How would you troubleshoot a PVC stuck in Pending state?**

First, I would describe the PVC using kubectl describe pvc <name> to look for events. Common causes include:

No available PVs that match the requested size, access mode, or StorageClass.

If using dynamic provisioning, I'd check that the PVC has a valid storageClassName.

I would then check the StorageClass using kubectl get storageclass to make sure it's present and has the right provisioner.

For cloud volumes, I'd also check the CSI controller or cloud provider logs (kube-controller-manager, csi-provisioner) for provisioning errors like permission issues or API limits.

In one past case, the IAM role of the CSI driver didn't have the right EBS permissions, so volume creation failed silently.

**4.7 What are the pros and cons of dynamic provisioning vs static PersistentVolumes?**

**Dynamic provisioning** is great for scalability and automation. It allows storage to be created on demand based on PVCs, reducing manual effort and mistakes. It works well in cloud environments where infrastructure is programmable (e.g., AWS EBS, GCP PD).

**Static provisioning**, on the other hand, gives admins more control. You can pre-define storage with specific performance characteristics or for regulatory compliance.

That said, static provisioning is hard to scale in large environments and increases ops overhead. I would only use static PVs in edge cases — for example, pre-configured NFS shares or legacy systems — and use dynamic provisioning with StorageClasses for most workloads.

**4.8 Pod stuck in ContainerCreating because PVC is Pending**

Check the PVC → PV → StorageClass binding flow, then CSI/controller logs and cloud provider events. Most common causes: no matching PV, dynamic provisioning failed (CSI error), or topology mismatch.

**Step-by-step debug & commands:**

1. kubectl describe pod <pod> — see event saying PVC pending.
2. kubectl get pvc / kubectl describe pvc <pvc> — check status, events (provisioner errors).
3. kubectl get pv — is there a matching PV? Check storageClassName, accessModes, capacity.
4. kubectl get sc — verify provisioner, volumeBindingMode (WaitForFirstConsumer matters).
5. Check CSI/controller pods: kubectl -n kube-system get pods | grep -i csi and kubectl -n kube-system logs <csi-controller-pod>.
6. If cloud-provider, check cloud API (eg. AWS EBS provisioning errors, IAM permissions).
7. If WaitForFirstConsumer, verify scheduler decisions (node selectors, affinity) blocking binding.

**Remediation:**

* Fix StorageClass or create a PV with matching attributes.
* Resolve CSI errors (reconfigure credentials, RBAC, fix driver CrashLoopBackOff).
* If topology mismatch, either use a regional storage class or schedule Pod to same zone.

**4.9 Volume mount errors inside a running Pod (Read-only file system / Permission denied)**

**Short answer:**  
Check volume mode (block vs filesystem), mount options, Linux permissions/UIDs, and Pod securityContext (fsGroup, runAsUser), then node mount state.

**Step-by-step debug & commands:**

1. kubectl describe pod <pod> → events for mount errors.
2. Exec into the Pod (if possible): kubectl exec -it <pod> -- /bin/sh and inspect mount point: mount | grep <mountpath> and ls -l <mountpath>.
3. Check PV & PVC: accessMode, volumeMode (Filesystem vs Block).
4. Check Pod spec securityContext and container securityContext. fsGroup ensures group ownership for mounted volumes.
5. On the node: ssh node → lsblk, mount, dmesg | tail and journalctl -u kubelet for mount/permission errors.

**Remediation:**

* If ReadOnly due to mount option, update StorageClass (or explicit mountOptions).
* If permission issue, set fsGroup in Pod or change on-disk ownership (chown) on the PV (careful on shared volumes).
* If volumeMode is block but you expect filesystem — fix PVC/Pod spec

**4.10 Data not persisting after Pod restart**

Short answer:  
Verify it's not emptyDir (ephemeral) and that PVC actually binds to a persistent PV; **check reclaim policy and workload type** (Deployment vs StatefulSet).

Debug & commands:

1. Inspect Pod: ensure volume is referencing a PVC, not emptyDir. kubectl get pod -o yaml <pod>
2. kubectl get pvc,pv — confirm PVC is Bound to PV.
3. kubectl describe pv <pv> — check persistentVolumeReclaimPolicy and underlying storage.
4. If StatefulSet, check volumeClaimTemplates usage.

Remediation:

* Replace ephemeral emptyDir with PVC backed by persistent storage.
* If PV is recreated and Retain/Delete policy removed data, adjust policy or backup/restore.
* Use StatefulSet for stable identity + stable PVCs if application requires.

**4.11 Stuck VolumeAttachment objects**

**🔎 Why VolumeAttachment Gets Stuck**

1. Node is Unavailable

* If a node went NotReady or was terminated unexpectedly, Kubernetes cannot detach the volume cleanly.
* Cloud providers (AWS, Azure, GCP) sometimes don’t detach disks quickly.

1. CSI Driver Crash / Bug

* If the CSI driver (node plugin or controller) is in CrashLoopBackOff, it can’t process attach/detach requests.

1. **Finalizer Not Removed**

* **VolumeAttachment objects have a finalizer:**

****

* **If the CSI external-attacher sidecar fails, the finalizer never gets removed → object stuck.**

1. **Stale Node Reference**

* If a node was deleted but the VolumeAttachment still references it, Kubernetes can’t progress.

**🛠️ How to Troubleshoot**

1. List stuck VolumeAttachment objects

kubectl get volumeattachments

2. Describe one to see details

kubectl describe volumeattachment <va-name>

Look for:

* attached: false
* attachmentMetadata
* Events like: timed out waiting for attach or driver not found.

3. Check CSI Controller Logs

The csi-attacher (sidecar in controller Deployment) manages attach/detach:

kubectl -n kube-system logs deploy/csi-controller -c csi-attacher

**🛠️ Fixes**

**A. If caused by a dead node**

* Remove the node object so K8s can clean up:

kubectl delete node <node-name>

Then Kubernetes should retry detaching and clean up VolumeAttachment

**B. If finalizer is stuck**

Patch the object to remove the finalizer manually:

kubectl patch volumeattachment <va-name> -p '{"metadata":{"finalizers":[]}}' --type=merge

⚠️ Be careful — this forces Kubernetes to “forget” about the attachment, you might need to manually detach volume at cloud level.

**C. If CSI driver is broken**

* Fix/restart the CSI driver (see your earlier CrashLoopBackOff scenario).
* Once the CSI attacher comes back, it usually removes finalizers automatically.

1. **Kubernetes – Configuration and Secrets**
   1. **What are ConfigMaps and Secrets?**

ConfigMaps and Secrets are API objects that let you decouple configuration and sensitive information from your containerized applications, so you don’t have to hardcode them in your images.

🔹 ConfigMaps

* Purpose: Store non-sensitive configuration data (key-value pairs, config files, command-line arguments, environment variables).
* Use cases:
  + Store app settings (e.g., APP\_MODE=production).
  + Provide entire config files (like nginx.conf).
* Data type: Plaintext (not encrypted).
* Ways to consume:
  + As environment variables in a Pod.
  + As command-line arguments.
  + As files in a mounted volume.

A white background with a black and white flag

AI-generated content may be incorrect.

**🔹 Secrets**

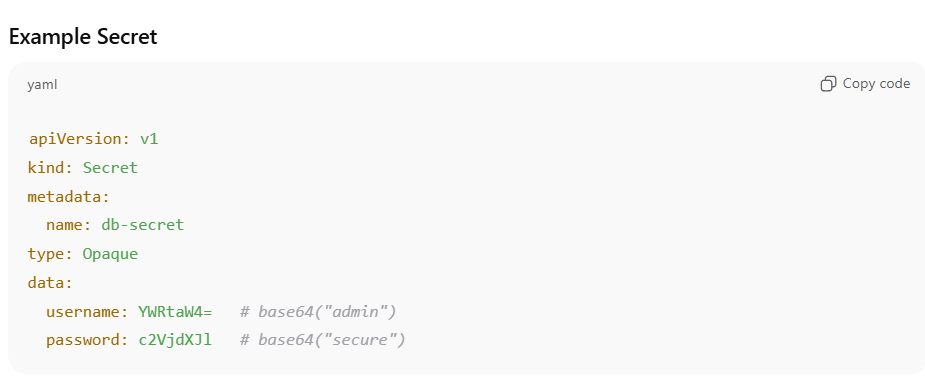
* **Purpose**: Store **sensitive data** such as passwords, API tokens, SSH keys, TLS certificates.
* **Use cases**:
  + Database credentials.
  + Docker registry credentials.
  + Certificates and private keys.
* **Data type**: Base64-encoded (not true encryption by default, but can be encrypted at rest using KMS or etcd encryption).
* **Ways to consume**:
  + As environment variables.
  + As files in a mounted volume.

A white background with black text

AI-generated content may be incorrect.

* 1. How do you mount a Secret as an environment variable or volume?

🔹 1. Mounting a Secret as Environment Variables



**A screenshot of a computer

AI-generated content may be incorrect.**

**A white rectangular object with a black border

AI-generated content may be incorrect.**

**🔹 2. Mounting a Secret as a Volume**

**A screenshot of a computer code

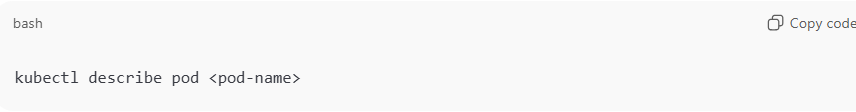
AI-generated content may be incorrect.**

**A close-up of a computer screen

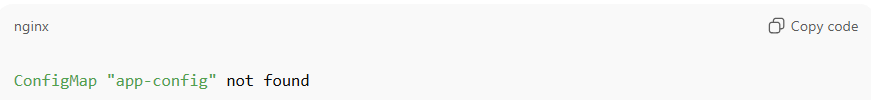
AI-generated content may be incorrect.**

****

* 1. **A Pod is stuck in CrashLoopBackOff because it can’t load a ConfigMap. How do you debug and fix it?**
* **First, check if the Pod references a ConfigMap that exists:**

****

**Look for events like:**

****

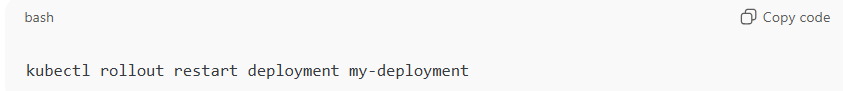
* If the ConfigMap doesn’t exist → create it or fix the Pod spec.
* If the ConfigMap exists → check if the key used in env or volumeMount matches exactly (case-sensitive).
* If the ConfigMap was recently updated, and Pod uses immutable ConfigMap (--immutable=true) → you must recreate the Pod to pick up changes. Restart will not work.
  1. If you update a ConfigMap or Secret, will Pods automatically get the new values? How would you force Pods to reload them?

By default:

* **Environment variables** → changes **do not update** in running Pods. You must restart Pods.
* **Mounted volumes** → for ConfigMaps/Secrets (not immutable), the update propagates (default ~1 minute refresh). There is no need to restart .

To force reload:

* Rollout restart:



* Use checksum annotations in Pod template (common pattern in Helm charts):

A computer screen shot of a computer

AI-generated content may be incorrect.

* 1. Your Secret is mounted as a volume, but inside the container the files are empty. What could be the issue?

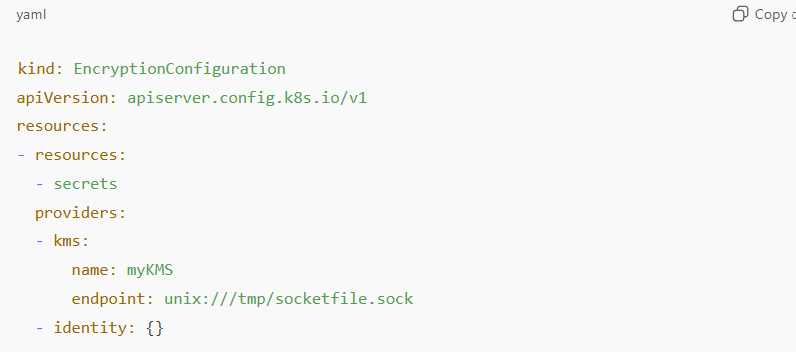
Possible causes:

* The Secret exists but has **empty data** (values were not base64 encoded correctly or encoded from an empty string).
  + Check with:

A white background with black text

AI-generated content may be incorrect.

* Wrong key is referenced in Pod manifest.
* Pod’s service account does not have **RBAC permission** to access Secrets in that namespace.
* If using immutable: true → check if the Secret was re-created instead of updated.
  1. **How do you secure Secrets in etcd and ensure they are not exposed accidentally?**
* **By default, Secrets are stored base64-encoded in etcd → not secure.**
* **Security best practices:**
* **Enable encryption at rest in EncryptionConfiguration:**

****

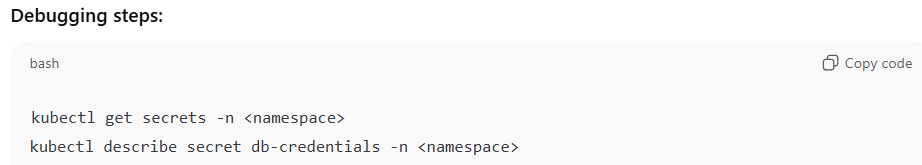
* Use RBAC to restrict access:
* Don’t give developers get secrets permission.
* Use external Secret stores (e.g., HashiCorp Vault, AWS Secrets Manager, Azure Key Vault) via CSI drivers.
* Avoid mounting Secrets as environment variables (can show up in kubectl describe pod output). Prefer volumes.

**Key takeaway:** Base64 ≠ encryption; you must use KMS/etcd encryption + RBAC + external secret managers.

* 1. **A Pod is not starting because of this error: secret "db-credentials" not found. But you see the Secret exists. Why could this happen?**

Possible causes:

1. Secret is in the **wrong namespace** (Secrets are namespace-scoped).
   * Pod and Secret must be in the same namespace.
   * Fix: recreate the Secret in the correct namespace.
2. Secret type mismatch (e.g., Pod expects kubernetes.io/dockerconfigjson for pulling images, but Secret type is Opaque).
3. The Secret name in the Pod spec has a typo.

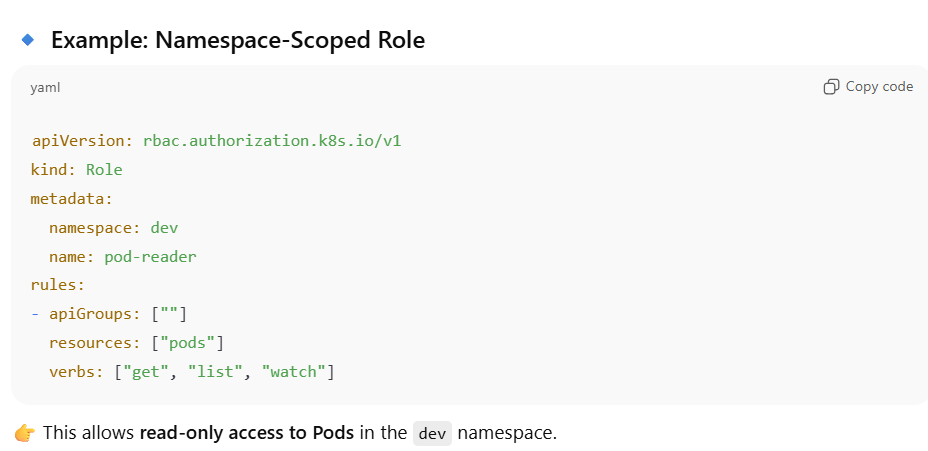


1. **Security**
   1. **What is Role-Based Access Control (RBAC) in Kubernetes?**

**RBAC is a security mechanism in Kubernetes that controls who (users, groups, service accounts) can perform what actions on which resources in the cluster.**

🔹 Key Concepts in RBAC

1. Subjects (the "who")
   * User → human or external identity.
   * Group → collection of users.
   * ServiceAccount → identity for Pods/services inside the cluster.
2. Resources (the "what")
   * Pods, ConfigMaps, Secrets, Deployments, etc.
   * Non-resource URLs (like /healthz).
3. Verbs (the "action")
   * get, list, watch, create, update, delete, etc.
4. Roles & ClusterRoles
   * Role → grants permissions within a namespace.
   * ClusterRole → grants permissions cluster-wide (or across all namespaces).
5. RoleBinding & ClusterRoleBinding
   * RoleBinding → attaches a Role to a subject in a namespace.
   * ClusterRoleBinding → attaches a ClusterRole to a subject cluster-wide.



A screenshot of a computer

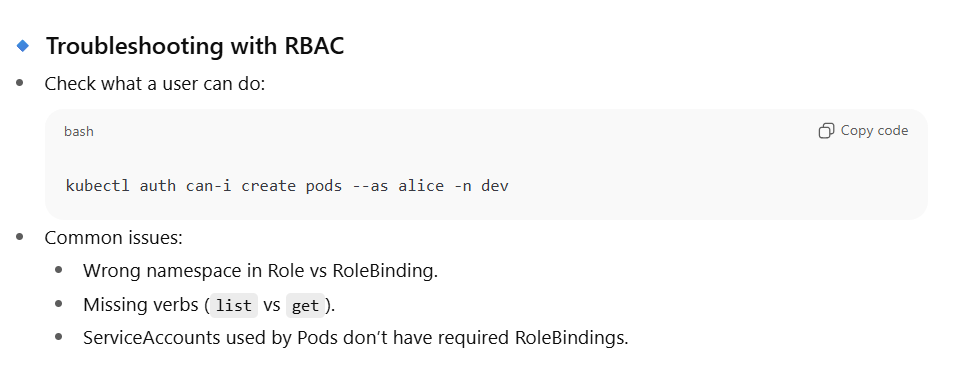
AI-generated content may be incorrect.

A white background with black text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



* 1. How do you secure communication in Kubernetes?

Securing communication in Kubernetes is a **multi-layered** topic, because traffic flows between:

* **Pods ↔ Pods**
* **Pods ↔ Services**
* **Users ↔ API server**
* **API server ↔ etcd**
* **Cluster ↔ external clients**

🔹 Ways to Secure Communication in Kubernetes

* 1. **API Server Communication (Control Plane Security)**
* **TLS Everywhere:**
* The Kubernetes API server only serves HTTPS (443), secured with TLS.
* Communication between API server and kubelet/kube-proxy/controller-manager is also TLS-secured.
* **Authentication**:
* X.509 client certificates, Bearer tokens, OIDC, or ServiceAccounts.
* **Authorization**:
* Enforce RBAC (Role-Based Access Control) to control what users/services can do.
* **Encryption at Rest**:
* Encrypt Secrets and sensitive data stored in etcd using KMS or AES.
  1. **Pod-to-Pod / Pod-to-Service Communication (Data Plane Security)**
* mTLS (Mutual TLS):
* Use a service mesh (e.g., Istio, Linkerd, Consul) to enforce pod-to-pod encryption and identity verification.
* Network Policies:
* Restrict traffic between Pods/namespaces (default is allow-all).
* Example: Only allow frontend to talk to backend

**A screenshot of a computer

AI-generated content may be incorrect.**

1. Ingress/Egress Security (External Communication)

* **TLS Termination at Ingress Controller**:
* Store TLS certificates in Kubernetes Secrets.
* Configure Ingress to use TLS:



* **Mutual TLS for Clients**: Require client certs for APIs (use NGINX/Envoy/Istio).
* **Egress Control**: Use NetworkPolicy or service mesh to control outbound traffic to external services.

1. ServiceAccount & Identity Security

* Each Pod should use a **dedicated ServiceAccount**.
* Use **RBAC** to give minimal permissions.
* In cloud environments:
* Use IAM Roles for ServiceAccounts (IRSA in AWS, Workload Identity in GCP, Azure AD Workload Identity).

1. **ETCD Communication Security**

* etcd stores all cluster state (including Secrets).
* Secure with:
* TLS certificates for etcd peer and client connections.
* Encryption at rest.
* Strict firewall rules to allow only API server access.

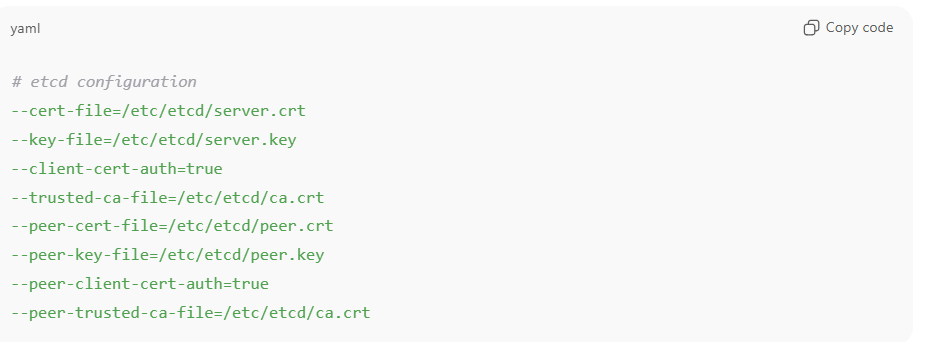
**1. TLS Certificates for etcd Communication**

etcd has two types of communication:

1. **Peer communication** (etcd ↔ etcd nodes)
2. **Client communication** (API server ↔ etcd)

**How to enable TLS:**

* **Generate certificates** using cfssl, openssl, or AWS Certificate Manager.
  + peer.crt/key → for etcd nodes to talk to each other
  + server.crt/key → for API server to talk to etcd
  + ca.crt → trusted CA for verifying certs
* **Configure etcd**:



* **Verify**: Use etcdctl with TLS flags to test:

A screenshot of a computer

AI-generated content may be incorrect.

**2. Encryption at Rest**

Even if network traffic is encrypted, Secrets and other sensitive data in etcd must also be encrypted on disk.

How to enable encryption:

* Kubernetes supports encryption providers configured in EncryptionConfiguration file (usually /etc/kubernetes/encryption-config.yaml):



* **Steps:**
  1. Create an AES key (base64-encoded)
  2. Reference it in the encryption config
  3. Update API server manifest with --encryption-provider-config=/etc/kubernetes/encryption-config.yaml
  4. Restart API server to pick up changes
  5. Optionally, **re-encrypt existing Secrets** by using a small controller or re-apply them

🔹 In **EKS**, encryption at rest can be enabled with **KMS keys** when creating the cluster:

A close-up of a computer code

AI-generated content may be incorrect.

1. Strict Firewall Rules (Network Isolation)

 **etcd should never be directly exposed** to the Internet.

 Only **API server nodes** should talk to etcd.

**How to implement network security:**

* **Self-managed Kubernetes:**
  + Use firewall rules (iptables/security groups/VPC rules)
  + Only allow port 2379 (client) and 2380 (peer) for authorized hosts
* **EKS (managed control plane):**
  + AWS manages control plane network
  + Nodes only communicate via private ENIs to API server
  + etcd is not publicly exposed
* **Optional extra step:**
  + Use VPC **private subnets** for worker nodes and control plane endpoints
  + Restrict access via **Security Groups and Network ACLs**

**🔹 4. Best Practices**

1. **Rotate certificates** regularly (especially client certs)
2. **Rotate KMS keys** for encrypted Secrets
3. **Use RBAC and IAM** to restrict API server access
4. **Audit access** with control plane logs
5. **Backup etcd** regularly (encrypted S3, preferably using etcdctl snapshot save with TLS)
6. For high security, consider **Etcd TLS mutual authentication + PodNetworkIsolation + KMS encryption**
   1. **Why service account is needed in Kubernetes and how does it work ?**

**In Kubernetes, Service Accounts (SAs) are identities for Pods.**

1. **Authentication for Pods**

* When a Pod wants to interact with the Kubernetes API (e.g., read ConfigMaps, write to Secrets, watch resources), it needs an identity.
* Service Accounts provide that identity automatically.
* Each Pod can be associated with a Service Account → Kubernetes mounts a token (JWT) inside the Pod (/var/run/secrets/kubernetes.io/serviceaccount/token) which is used to authenticate to the API server.

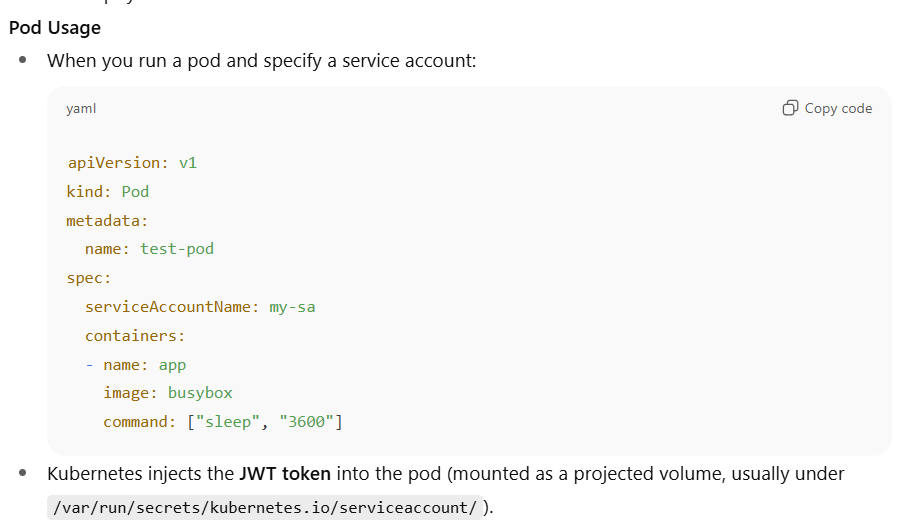
**2. Fine-Grained Permissions (RBAC)**

* Kubernetes uses **RBAC (Role-Based Access Control)** to define **what a Service Account can do**.
* Example:
  + metrics-server Pod needs to read node stats → bind its Service Account to a Role with get/list/watch permissions on nodes/metrics.
  + Your application Pod may need **only read access to a specific ConfigMap**.

This ensures **least privilege access** (security principle).

1. **Isolation Between Workloads**

* Each app / Pod can use a **different Service Account**.
* Prevents apps from accidentally (or maliciously) accessing resources they don’t need.
* Example:
  + A “frontend” app should not have permissions to delete Pods in another namespace.
  + With Service Accounts + RBAC, you isolate them.



* 1. **What are pod security contexts?**

A **security context** in Kubernetes defines **privilege and access control settings** for a pod or container.

It controls **how processes inside containers run**, including:

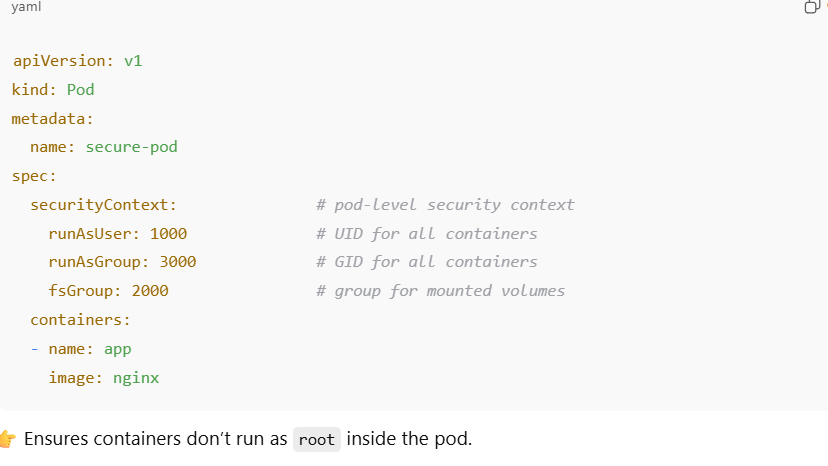
* User/group IDs
* Linux capabilities
* Privileged mode
* Filesystem access
* SELinux/AppArmor profiles
* Seccomp profiles

**🔹 Levels**

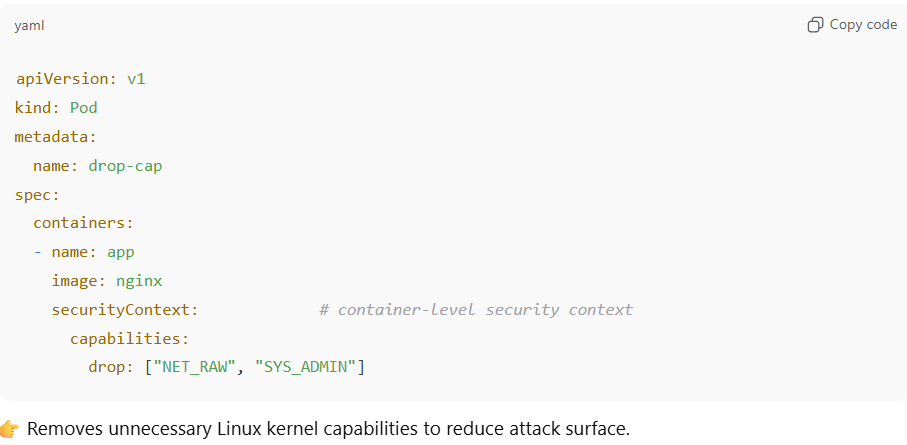
* **Pod-level security context** → applies to **all containers in the pod**.
* **Container-level security context** → overrides pod-level settings for that specific container.

**🔹 Examples**

**1. Run as non-root**

****

1. **Drop Linux capabilities**

****

1. **Read-only root filesystem**

**A white background with text

AI-generated content may be incorrect.**

1. **Deployments and Management**
   1. **Difference between Replica Set, Deployment, StatefulSet and DaemonSet**

1. ReplicaSet

* Purpose: Ensures a specified number of identical Pods should be running at all times.
* Key Points:
  + If a Pod crashes or is deleted, the ReplicaSet will create a replacement.
  + Works on a set of identical, stateless Pods.
  + Rarely used directly — most people use Deployment, which manages ReplicaSets automatically.
* Use Case: Keeping a fixed number of stateless app Pods running.

**2. Deployment**

* **Purpose**: Higher-level controller that manages **ReplicaSets** (and hence Pods).
* **Key Points**:
  + Supports **rolling updates** and **rollbacks** (important for versioned deployments).
  + Creates/updates ReplicaSets for each version of your app.
  + Most commonly used resource for deploying stateless applications.
* **Use Case**: Deploying and updating stateless workloads (e.g., web servers, APIs).

**3. StatefulSet**

* **Purpose**: Manages Pods that need **stable network identity and persistent storage**.
* **Key Points**:
  + Pods are **not interchangeable** — each has a fixed identity (pod-0, pod-1, etc.).
  + Used with **PersistentVolumes** so data is preserved if Pods restart.
  + Pods are created and deleted **in order** (sequential scaling).
  + Useful for workloads that require **sticky identities and storage**.
* **Use Case**: Databases, Kafka, Zookeeper, Redis Cluster, Elasticsearch.

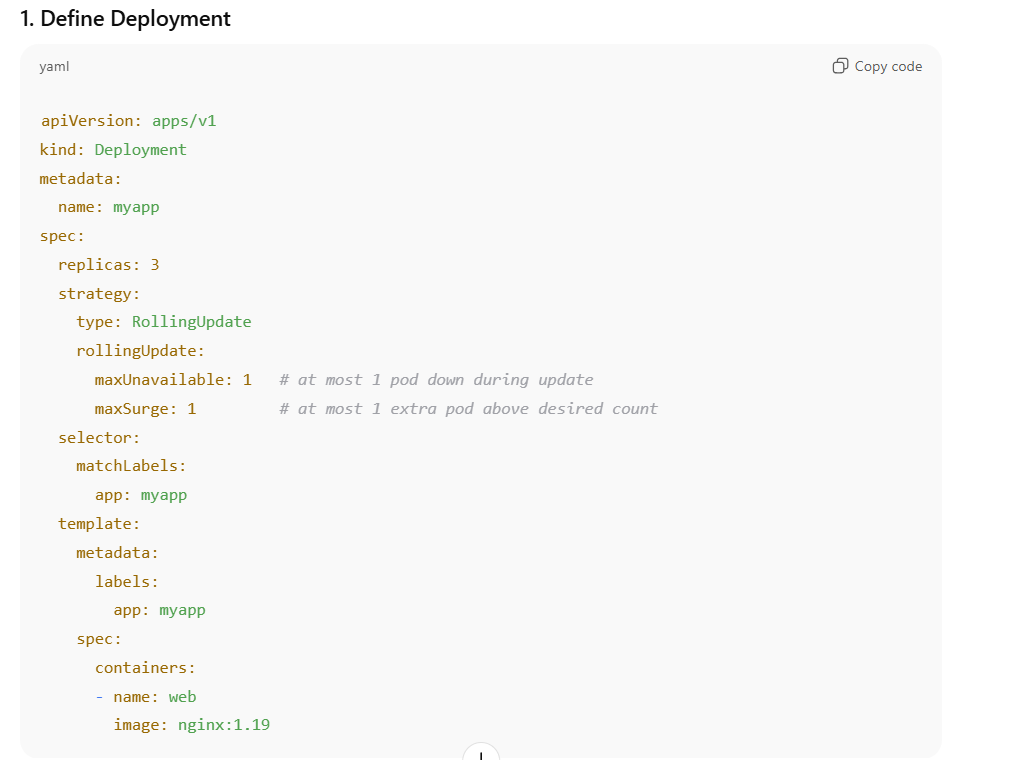
**4. DaemonSet**

* **Purpose**: Ensures **one Pod runs on every (or selected) node** in the cluster.
* **Key Points**:
  + Automatically schedules Pods on **all current and new nodes**.
  + No matter how many nodes you add, each gets exactly one Pod.
  + Typically used for **infrastructure or monitoring agents**.
* **Use Case**: Logging (Fluentd, Filebeat), Monitoring (Prometheus node exporter, Datadog agent), Networking plugins (CNI, kube-proxy).
  1. **How do you perform rolling updates in Kubernetes?**
* A rolling update is the default deployment strategy in Kubernetes.
* It updates pods gradually by creating new pods with the updated spec and terminating old ones.
* This ensures zero downtime if configured correctly.

🔹 How Rolling Updates Work

* Managed by a Deployment object.
* The Deployment controller:
  1. Scales up new ReplicaSet (with updated spec).
  2. Scales down old ReplicaSet.
  3. Balances this until the desired state is reached.

🔹 **Performing a Rolling Update**



A screenshot of a computer

AI-generated content may be incorrect.

A close-up of a logo

AI-generated content may be incorrect.

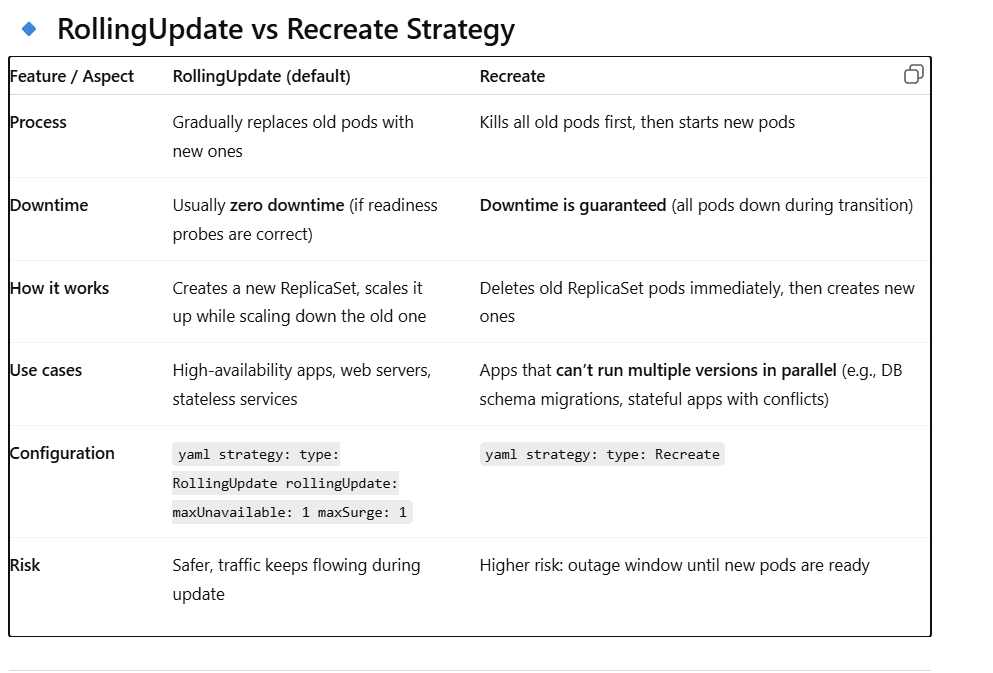
**🔹 Rollbacks**

If something goes wrong:

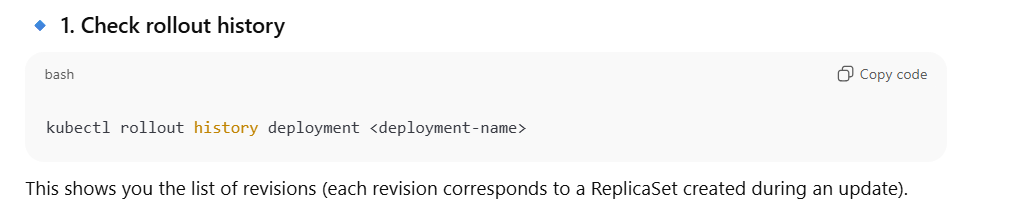
A close-up of a logo

AI-generated content may be incorrect.

* 1. What is the difference between rolling update and recreate strategy?



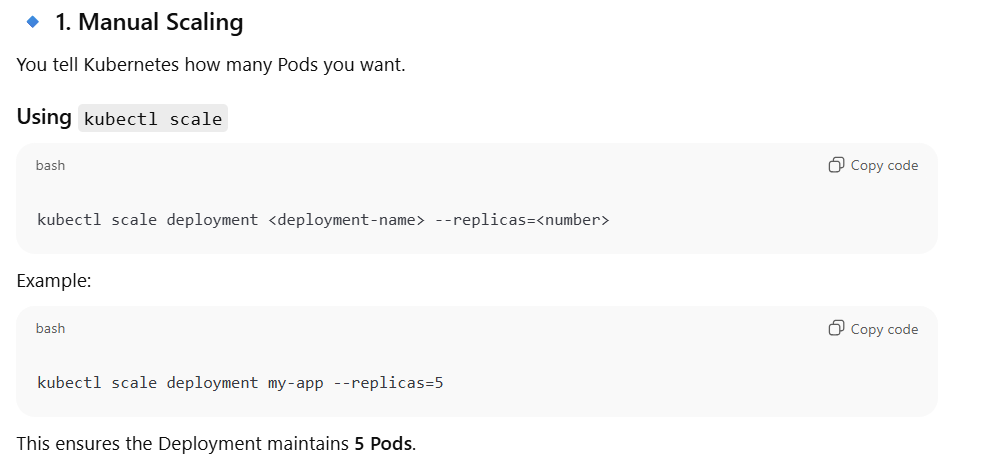
* 1. How do you rollback a deployment to a specific version?



A screenshot of a computer program

AI-generated content may be incorrect.

* 1. How do you scale a deployment?



A screenshot of a computer

AI-generated content may be incorrect.

* 1. What is the difference between **readiness** and **liveness** probes?

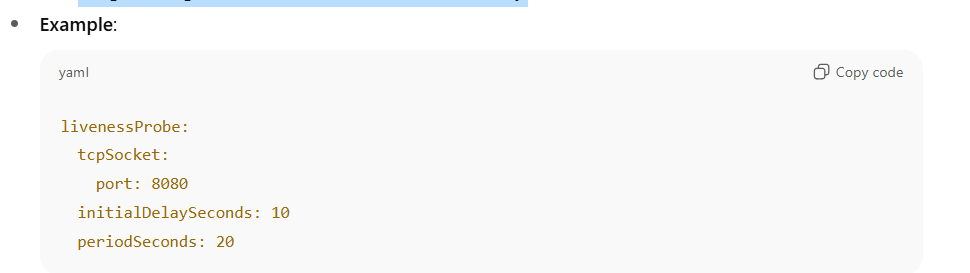
**🔹 Readiness Probe**

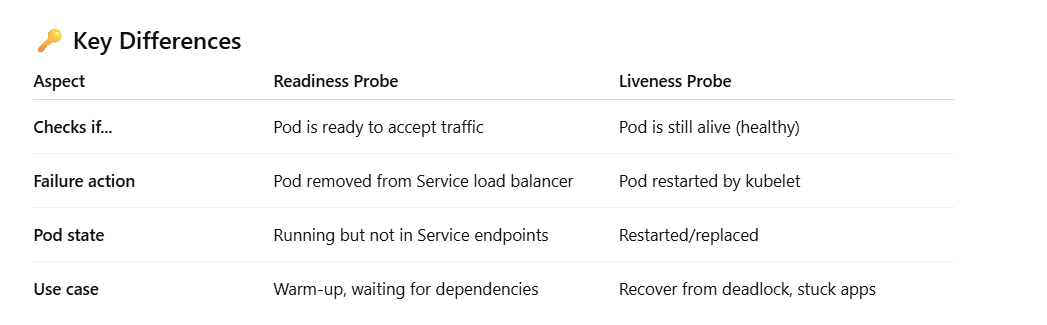
* **Purpose**: Tells Kubernetes **when a Pod is ready to serve traffic**.
* **Behavior**:
  + If the probe fails → Pod is marked **“Not Ready”**.
  + Kubernetes removes it from the **Service Endpoints** (traffic won’t be sent to it).
  + The Pod still runs, but doesn’t receive requests.
* **Use Case**:
  + App needs some **warm-up/startup time**.
  + Database connections or caches must be ready before serving traffic.



**Liveness Probe**

* **Purpose**: Tells Kubernetes **if a Pod is still alive** (healthy).
* **Behavior**:
  + If the probe fails → Kubernetes **kills the Pod** and restarts it.
  + Ensures your app can recover from deadlocks, memory leaks, or hangs.
* **Use Case**:
  + App gets stuck and stops responding.
  + Long-running services that need restarts if unhealthy.





* 1. **How does Kubernetes ensure zero downtime during a Deployment rollout, and under what conditions could downtime still occur?**

Kubernetes uses **rolling updates** by default:

* Controlled by maxUnavailable (Pods allowed to be unavailable) and maxSurge (extra Pods allowed temporarily).
* Example: With replicas=4, maxUnavailable=1, maxSurge=1, Kubernetes ensures at least **3 Pods are always available**.

**Downtime can still occur if**:

* Readiness probes are not configured → traffic is routed to unready Pods.
* Bad app image/config → new Pods crash immediately, causing cascading failures.
* Network or node issues during rollout.

**Best practices**: Always configure **readiness probes**, use maxUnavailable=0, and implement

* 1. How do you handle database schema changes during zero-downtime Deployments?
* Databases are **stateful**, so rolling updates can break compatibility.
* Strategies:
* **Expand-Contract Pattern**:
  1. First deploy changes that are backward-compatible (add new fields, keep old ones).
  2. Update the app to use new schema.
  3. Remove old fields later.
* Use **migrations as init containers** before app Pods start.
* Or, manage schema with **separate migration jobs**.
* **Challenge**: A Deployment only guarantees app-level zero downtime, not DB schema-level.
  1. **How do you update a secret or config in a Deployment without downtime?**
* If using **mounted volumes** → Pods won’t pick up updates automatically → restart required.
* If using **env vars from ConfigMap/Secret** → restart required too.
* Zero downtime approach:
* Update ConfigMap/Secret
* Trigger a **rolling restart** of Deployment

A close-up of a logo

AI-generated content may be incorrect.

* With proper readinessProbe, only ready Pods serve traffic during restart.
  1. **cheatsheet for command.**



1. **Pod Placements( scheduling, affinity, taints, topology)**
   1. How does the Kubernetes scheduler decide where to place a Pod?

When a **Pod** is created without a node assignment, it enters the Pending state. The **Kubernetes Scheduler** is responsible for finding the “best” node for that Pod.

The scheduling decision happens in **two main phases**: **Filtering** and **Scoring**.

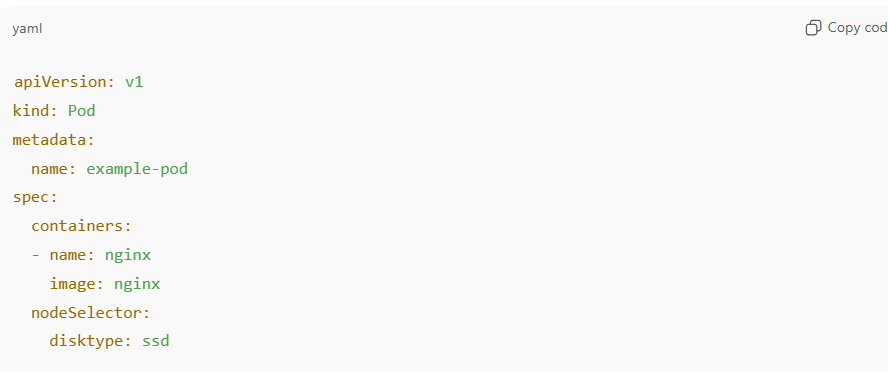
**1️. Filtering Phase (a.k.a. Predicates)**

The scheduler first **filters out nodes that cannot run the Pod**.  
It checks constraints such as:

* **NodeSelector / NodeAffinity**
* If the Pod has a nodeSelector or nodeAffinity, only matching nodes are considered
  1. **NodeSelector (Simple Node Selection)**

**✅ How it works:**

* NodeSelector is **key-value based**.
* Pod will **only** be scheduled on nodes with matching labels.
* Very simple, but not flexible.



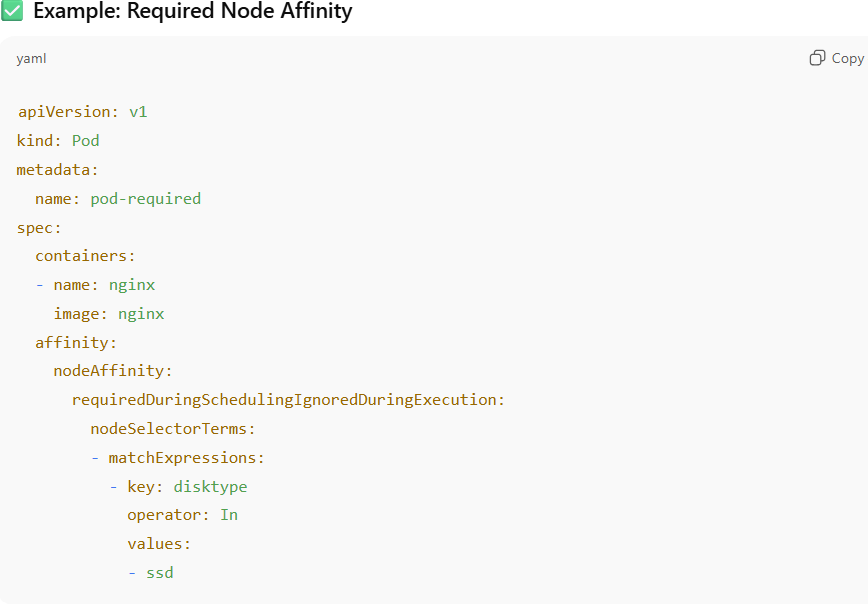
* disktype: ssd → Pod will only be scheduled on nodes that have the label disktype=ssd.

**Limitations:**

* Only **AND logic** is supported (all labels must match).
* No complex conditions (like OR, NOT, or weight-based preferences).
  1. **NodeAffinity (Advanced Node Selection)**

NodeAffinity is more flexible than NodeSelector.  
It has two types:

* + 1. **RequiredDuringSchedulingIgnoredDuringExecution**
* **Hard requirement**: Pod **must** be scheduled on nodes that match these rules
* Equivalent to NodeSelector but more expressive.
  + 1. **PreferredDuringSchedulingIgnoredDuringExecution**
* **Soft preference**: Scheduler tries to place Pod on matching nodes, but can place elsewhere if necessary.



Explanation:

* Pod will **only** schedule on nodes with disktype=ssd.
* operator can be:
  + In, NotIn, Exists, DoesNotExist, Gt, Lt.

A screenshot of a computer

AI-generated content may be incorrect.

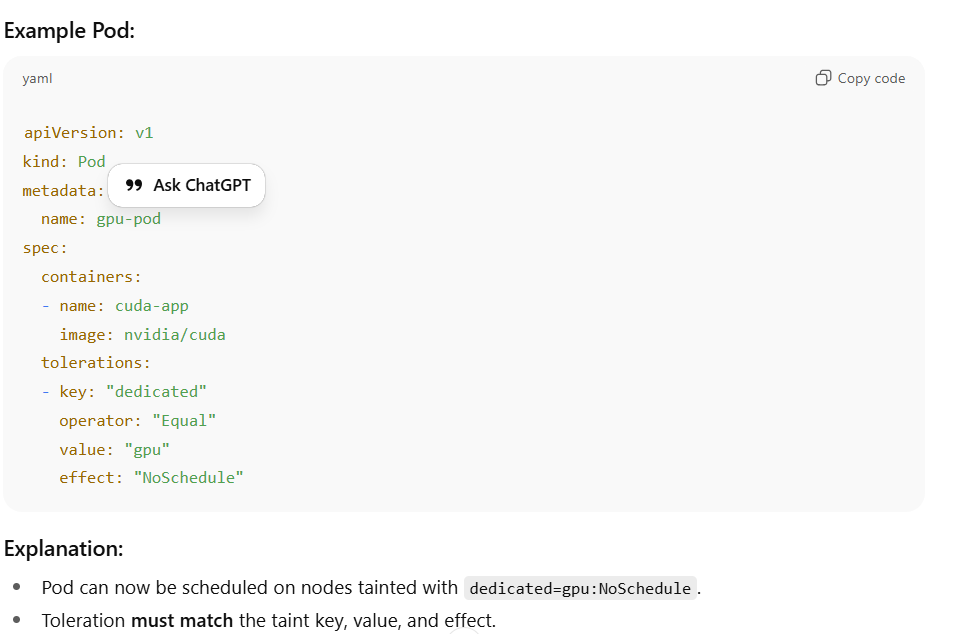
* **Taints and Tolerations**
* Nodes with taints are excluded unless the Pod has tolerations that “neutralize” them
  1. **Why We Need Taints and Tolerations**
* Kubernetes scheduler tries to schedule Pods **anywhere by default**.
* Sometimes, you want **certain nodes to be “special” or restricted** so that only specific Pods can run on them.
* **Taints** are applied to nodes to **repel Pods**, and **Tolerations** are applied to Pods to **allow them to be scheduled on tainted nodes**.
  1. Taints (Applied to Nodes)

kubectl taint nodes <node-name> key=value:effect

A screenshot of a computer

AI-generated content may be incorrect.

* 1. Tolerations (Applied to Pods)



* Resource Requests
* A node must have enough CPU, memory, and extended resources (like GPUs) available to satisfy the Pod’s requests.
* Node Unschedulable
* If a node is marked unschedulable (like a cordoned node), it’s ignored.

**Cordoning** a node means **marking it as unschedulable**.

* + 1. **How it works:**
* The node remains **ready and operational**, and **existing Pods keep running**
* No **new Pods** are scheduled on this node.
* Existing Pods are **not affected**; only future scheduling is blocked.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Draining a Node
* Draining is related but **more aggressive**: it **evicts existing Pods** before maintenance.
* This **safely moves workloads** elsewhere, preparing the node for maintenance



* Pod Topology Spread Constraints
* **Pod Topology Spread Constraints** ensure Pods are **evenly distributed across failure domains** like nodes, zones, or regions.
  1. What Problem Does It Solve?
* If one node or zone fails, too many replicas may go down together.
* It reduces availability and fault tolerance.
  1. How It Works?

You define rules in your Pod spec that tell the scheduler:

* **Where to spread Pods** → topology domain (like zone, hostname).
* **How many Pods can be in each domain** → max skew.
* **Which Pods to compare against** → via label selectors.
* **What to do if spreading isn’t possible** → DoNotSchedule or ScheduleAnyway.

A screen shot of a computer

AI-generated content may be incorrect.

* maxSkew
* Defines imbalance tolerance.
* Example: maxSkew: 1 → the number of Pods in any two zones can’t differ by more than 1.
* topologyKey
* kubernetes.io/hostname → spread across nodes.
* topology.kubernetes.io/zone → spread across zones.
* topology.kubernetes.io/region → spread across regions.
* whenUnsatisfiable
* DoNotSchedule: If spreading rule can’t be met, the Pod won’t be scheduled.
* ScheduleAnyway: Pod will still be scheduled even if imbalance happens (best effort).
* labelSelector
* Defines which Pods to count when calculating skew. Usually matches Pods of the same Deployment/Service.
* **PodDisruptionBudgets / InterPodAffinity**
  1. PodDisruptionBudget (PDB)
* Ensures a **minimum number of Pods remain available** during **voluntary disruptions** (like node maintenance, draining, or upgrades).
* Protects applications from **downtime caused by cluster operations**.
* Does **not** protect against **failures** (like node crashes)

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. **Inter-Pod Affinity / Anti-Affinity**

**Purpose**

* Affinity: Schedule Pods close to or on the same nodes as other Pods.
* Anti-Affinity: Schedule Pods away from other Pods to improve availability.

**Useful for:**

* Performance optimization (e.g., colocating Pods for low-latency communication).
* High availability (e.g., spreading replicas across nodes/zones).



**Explanation:**

* Scheduler **must** place this Pod on a node that already runs a Pod with app=frontend.
* topologyKey defines the domain to consider:
  + kubernetes.io/hostname → same node
  + topology.kubernetes.io/zone → same zone
* requiredDuringSchedulingIgnoredDuringExecution → **hard requirement**
* preferredDuringSchedulingIgnoredDuringExecution → **soft preference** (scheduler tries but doesn’t enforce).

****

1. **How Affinity / Anti-Affinity Works with Topology**

* The scheduler calculates a **score for each node** based on matching rules.
* Hard rules = node is excluded if not satisfied.
* Soft rules = node is preferred but not required.

1. **Combining PDBs with Pod Anti-Affinity**

* Anti-Affinity ensures Pods are **spread across nodes**.
* PDB ensures **minimum Pods remain running** during maintenance.
* Together:
* Cluster can tolerate node failures or updates without downtime.
* Example: 3 replicas, anti-affinity across nodes, PDB minAvailable: 2.

At the end of filtering, Kubernetes has a **set of feasible nodes**.

**2. Scoring Phase (a.k.a. Priorities)**

Next, the scheduler scores each feasible node to pick the most suitable one.  
Scoring functions assign points (0–100), and then the scheduler picks the node with the highest score.

Scoring factors include:

* Least Requested Priority
  + Prefers nodes with more free CPU/memory.
* Balanced Resource Allocation
  + Prefers nodes where resource usage is balanced (e.g., not CPU-heavy but memory-idle).
* Node Affinity Priority
  + If multiple nodes match affinity rules, some nodes may get higher scores.
* Image Locality Priority
  + Prefers nodes that already have the container image cached (faster startup).
* Topology Spread Priority
  + Balances replicas across zones/racks/nodes for HA.
* Custom Plugins
  + You can write custom scheduler plugins for business-specific logic.

**3.Binding Phase**

After scoring, the scheduler selects the highest-scoring node and sends a Bind API call to the API server, updating the Pod’s .spec.nodeName.

At this point, the Kubelet on that node picks up the Pod and starts pulling images + running containers

**📊 Visual Summary**

Pod Created

│

▼

Scheduler Watches for Pending Pods

│

▼

Filtering (Constraints)

- NodeSelector / Affinity

- Taints / Tolerations

- Resource Requests

- Topology Rules

│

▼

Scoring (Priorities)

- Resource balance

- Image locality

- Spread constraints

- Custom plugins

│

▼

Select Best Node

│

▼

Bind Pod → NodeName

│

▼

Kubelet runs containers

* 1. **What’s the difference between NoSchedule, PreferNoSchedule, and NoExecute taint effects?**

**🌟 Taints and Their Effects**

**1️. NoSchedule**

* **Meaning: Pods that do not tolerate this taint will not be scheduled on the node.**
* **Already running Pods are unaffected — they continue running, even if they don’t tolerate the taint.**

**✅ Use case:  
Reserve a node for special workloads (e.g., GPU nodes, infra-only nodes).**

**A screenshot of a computer

AI-generated content may be incorrect.**

**2️. PreferNoSchedule**

* **Meaning**: Scheduler will **try to avoid placing pods** that don’t tolerate the taint, but it’s a *soft rule*.
* If no better node is available, the Pod **may still be scheduled** onto the tainted node.

**✅ Use case:**Suggest to the scheduler to avoid certain nodes (e.g., noisy neighbors, less powerful machines), but still allow scheduling if cluster is constrained.

**kubectl taint nodes node2 environment=test:PreferNoSchedule**

**3️. NoExecute**

* **Meaning**: Strongest effect.
* **New Pods** that don’t tolerate the taint are **not scheduled**.
* **Existing Pods** that don’t tolerate the taint are **evicted immediately** (or after the tolerationSeconds grace period).

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. **Monitoring, Logging & Debugging**
   1. How do you debug a failing pod?

* **Check Pod Status**

**A screenshot of a message

AI-generated content may be incorrect.**

* **Describe the Pod** 
  + **Check Resource Issues**
  + **Check Scheduling (if Pod is Pending)**

**A screenshot of a computer

AI-generated content may be incorrect.**

* **If OOMKilled → increase memory limits.**
* **If throttling → adjust CPU requests/limits.**

**A screenshot of a computer

AI-generated content may be incorrect.**

* **Check Logs**

****

* **Exec into Pod (if running but unhealthy)**

**A close-up of a white background

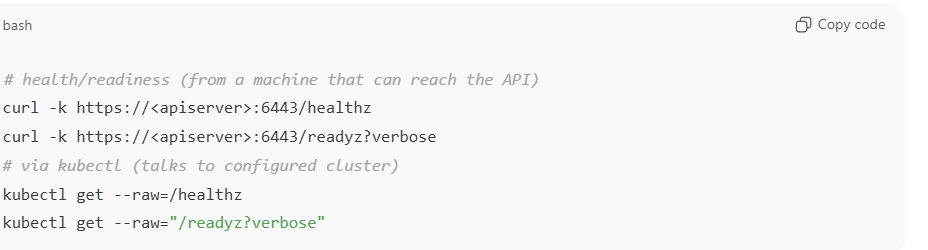
AI-generated content may be incorrect.**

* **Events and Cluster Health**

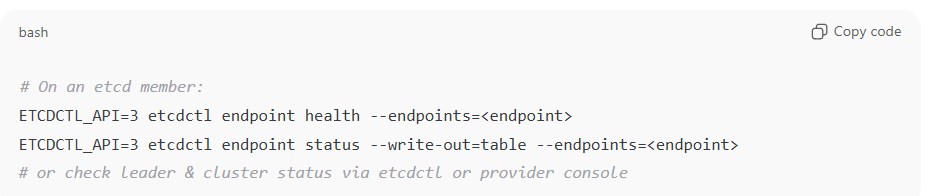
**A screenshot of a computer

AI-generated content may be incorrect.**

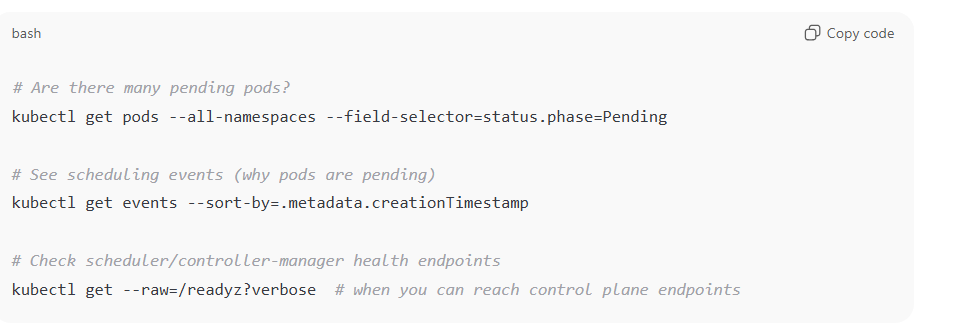
* 1. **How do you monitor Kubernetes cluster health?**
* **Control Plane Components health**
* **API Server**
* **Is it reachable, responsive, not overloaded?**
* **Check /healthz and /metrics endpoints.**
* Latency (how long API requests take)
* Error rate (5xx responses)
* Request drops / in-flight requests (throttling/overload)
* Authentication/authorization latency (slow webhook or OPA calls)



* ETCD
* Monitor DB size, latency, disk I/O, leader elections.



* Scheduler & Controller Manager
* Scheduler: scheduling latency and failed scheduling attempts (backlog of Pending pods).
* Controller Manager: processing of workqueues (are controllers reconciling objects?), controller restarts, leader election.



* Practical checks & dashboards
* **Prometheus + Grafana**: deploy kube-state-metrics, node-exporter, kube-apiserver/kube-scheduler/kube-controller-manager scrapers (or use Prometheus Operator ServiceMonitor). Use standard dashboards (Kubernetes Control Plane, etcd Overview, API Server Overview).
* **kubectl**:



* **Logs**: check control plane component logs:
*  If control plane runs as pods (kubeadm): kubectl -n kube-system logs -l component=kube-apiserver (or kubectl logs -n kube-system pod/<kube-apiserver-pod>).
*  On static-pod setups: check /var/log/containers or systemd unit logs on master nodes.
*  For managed clouds (EKS/GKE/AKS) use cloud provider diagnostics/console and provider metrics.
* **Node Health**
* **Nodes should be in Ready state**

**kubectl get nodes**

* **Monitor:**
* **CPU, memory, disk usage.**
* **Kubelet status.**
* **Node pressure conditions: MemoryPressure, DiskPressure, PIDPressure.**
* **Pod & Workload Health**
* **Pods should be Running and passing liveness/readiness probes.**
* **Monitor CrashLoopBackOff or ImagePullBackOff.**
* **Ensure desired vs available replicas match in Deployments.**

**A white background with a black and white flag

AI-generated content may be incorrect.**

* **Cluster Resource Utilization**
* **Watch requests vs limits vs actual usage for CPU/memory.**
* **Detect unschedulable Pods (Pending state).**
* **How: kubectl top nodes/pods, Metrics Server, Prometheus.**
* **Networking & Service Health**
* **CoreDNS → monitor query errors, latency.**
* **Service & Ingress reachability.**
* **Network policies (ensure they aren’t blocking critical traffic).**
* **Storage Health**
* **PVs/PVCs bound and accessible.**
* **Volume mount errors.**
* **Disk usage nearing capacity.**
* **CSI driver metrics, node storage exporters.**