Kubernetes DevOps Handbook

h*ttps*://www.linkedin.com/in/ranjithshepur/

Preface

Kubernetes (often abbreviated as **K8s**) has become the backbone of modern cloud-native infrastructure. From startups to Fortune 500 companies, organizations rely on Kubernetes to manage containerized applications with reliability, scalability, and automation.

This handbook is designed as a **comprehensive guide** to mastering Kubernetes — starting from the basics and going deep into advanced production-grade topics. It is written for:

- **Beginners** who want to learn Kubernetes step by step.
- **Intermediate engineers** who want to strengthen their hands-on skills.
- **Experienced professionals** preparing for **real-world challenges and interviews**.

What makes this guide different?

- Covers **foundational to advanced concepts** with clarity.
- Includes **real-world production scenarios**, not just theory.
- Every topic is explained with **commands, YAML manifests, and tips**.
- Designed for **interview preparation** with "how interviewers ask" notes.
- Written in a **step-by-step learning path**, so you never get lost.

Tip: Think of Kubernetes as the "operating system for your data center." Just as Linux manages processes on a single machine, Kubernetes manages containers across a **cluster of machines**.

1 Kubernetes Basics & Architecture

Kubernetes (K8s) is an open-source container orchestration system that automates the deployment, scaling, and management of containerized applications. It follows a master–worker architecture.

Key Concepts:

- **Cluster** – A collection of nodes (machines) running Kubernetes.
- **Node** – A worker machine (VM or physical) where Pods are scheduled.
- **Pod** – The smallest deployable unit in Kubernetes (one or more containers).
- **Control Plane** – Manages the cluster, decides scheduling, scaling, and state.
- **Kubelet** – Agent running on each worker node that communicates with the API server.
- **kubectl** – Command-line tool to interact with Kubernetes clusters.

Kubernetes High-Level Architecture:

```
      +----------------------------+
      |     Kubernetes Cluster     |
      +----------------------------+
         |                   |
+----------------+    +----------------+
| Control Plane  |    |   Worker Node   |
+----------------+    +----------------+
| API Server     |    | Kubelet         |
| Scheduler      |    | Container Runtime|
| Controller Mgr |    | kube-proxy       |
| ETCD           |    | Pods             |
+----------------+    +----------------+
```

Basic Kubernetes Commands

Check Kubernetes Client and Server Version:

```
$ kubectl version --short
```

Output:

```
Client Version: v1.29.0
Server Version: v1.29.0
```

Check Cluster Information:

```
$ kubectl cluster-info
```

Output:

```
Kubernetes control plane is running at https://127.0.0.1:6443
CoreDNS is running at https://127.0.0.1:6443/api/v1/namespaces/kube-system/services/
```

List All Nodes in the Cluster:

```
$ kubectl get nodes -o wide
```

Output:

```
NAME           STATUS   ROLES    AGE   VERSION   INTERNAL-IP   EXTERNAL-IP
worker-node1   Ready    <none>   10d   v1.29.0   10.0.0.21     <none>
worker-node2   Ready    <none>   10d   v1.29.0   10.0.0.22     <none>
master-node    Ready    master   10d   v1.29.0   10.0.0.10     <none>
```

Create a Simple Pod with Nginx Image:

```
$ kubectl run mypod --image=nginx
```

Output:

```
pod/mypod created
```

Get All Pods in Default Namespace:

```
$ kubectl get pods
```

Output:

```
NAME    READY   STATUS    RESTARTS   AGE
mypod   1/1     Running   0          1m
```

Describe a Pod (Detailed Info):

```
$ kubectl describe pod mypod
```

Output (snippet):

```
Name:         mypod
Namespace:    default
Node:         worker-node1/10.0.0.21
Start Time:   Tue, 09 Sep 2025 10:10:45 IST
Containers:
  nginx:
    Image: nginx
    Port:  80/TCP
```

Pages 7–12: Kubernetes Installation

Overview

Kubernetes can be installed in different environments depending on the use case:

- Local testing and learning → **Minikube, Kind, Docker Desktop**.
- Production-ready clusters → **kubeadm, managed services (EKS, GKE, AKS)**.
- Cross-platform (Linux, Mac, Windows).

Best Practice: Always choose the simplest setup (like Minikube) for learning and move to kubeadm/managed clusters for production.

—

Installing on Linux (Ubuntu Example)

Step 1: Update system and install dependencies

```
$ sudo apt-get update && sudo apt-get upgrade -y
$ sudo apt-get install -y apt-transport-https ca-certificates curl
```

Step 2: Add Google Cloud's GPG key and repo

```
$ sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg \
  https://packages.cloud.google.com/apt/doc/apt-key.gpg

$ echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] \
https://apt.kubernetes.io/ kubernetes-xenial main" | \
sudo tee /etc/apt/sources.list.d/kubernetes.list
```

Step 3: Install kubelet, kubeadm, kubectl

```
$ sudo apt-get update
$ sudo apt-get install -y kubelet kubeadm kubectl
$ sudo apt-mark hold kubelet kubeadm kubectl
```

Output:

```
kubectl version --client
Client Version: v1.30.0
```

Interview Insight: Q: How would you set up Kubernetes on bare metal for production? A: Use `kubeadm init` to bootstrap the control plane, then join worker nodes using the join token.

—

Installing on Mac (Homebrew)

Command:

```
$ brew install kubectl
$ brew install minikube
```

```
kubectl version --client
minikube version
```

Pro Tip: On Mac, `Docker Desktop` already ships with Kubernetes (enable in settings).

—

Installing on Windows

Using Chocolatey:

```
C:\> choco install kubernetes-cli
C:\> choco install minikube
```

Enable Kubernetes in Docker Desktop (Windows 10/11): - Open Docker Desktop - Settings → Kubernetes → Enable Kubernetes

Interview Insight: Q: If you're on Windows, which installation method would you prefer for local development? A: Minikube or Docker Desktop because they abstract the VM/container setup.

—

Minikube Setup (Cross-Platform)

Start a local cluster:

```
$ minikube start --driver=docker
```

Check cluster status:

```
$ kubectl cluster-info
$ kubectl get nodes
```

```
NAME       STATUS   ROLES           AGE   VERSION
minikube   Ready    control-plane   10m   v1.30.0
```

+————+ +————+ — Minikube — —-¿ — kubectl — +————+ +————
——+ Local VM CLI to manage cluster
—

kubeadm Setup (Production-like)

Initialize Control Plane:

```
$ sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

Set kubeconfig for kubectl:

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Deploy Pod Network (Flannel):

```
$ kubectl apply -f \
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.y
```

Join Worker Node:

```
$ kubeadm join <MASTER-IP>:6443 --token <TOKEN> \
--discovery-token-ca-cert-hash sha256:<HASH>
```

Production vs Local:

- **Local:** Minikube, Kind (quick test, dev environment).
- **Production:** kubeadm, Managed Kubernetes (EKS/GKE/AKS).

Comparison: Local vs Production Kubernetes Installation

Interview Insight: Q: Which installation method would you use for a production-grade environment and why? A: For local development → Minikube/Kind. For production → kubeadm (on-prem) or managed Kubernetes like EKS/GKE/AKS depending on cloud provider.

2 Kubernetes Core Concepts

Kubernetes revolves around a set of fundamental building blocks. Mastering these is essential for working effectively in both local and production-grade clusters.

2.1 Pods

A **Pod** is the smallest deployable unit in Kubernetes. It encapsulates one or more containers, storage, network, and configuration.

Create a Pod:

```
$ kubectl run mypod --image=nginx --restart=Never
```

Output:

```
pod/mypod created
```

Pod YAML Definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

Method	**Use Case**	**Pros**	**Cons**
Minikube	Local development, learning, quick testing	Easy to install, supports multiple drivers (Docker, VirtualBox), lightweight	Not suitable for production, single-node only
kubeadm	On-premise clusters, production-like setup	Full control over cluster setup, resembles real production clusters, flexible networking choices	Complex setup, requires manual management of HA, scaling, and upgrades
EKS (AWS)	Managed Kubernetes in AWS	Highly available, integrated with AWS ecosystem (IAM, VPC, CloudWatch), automated scaling	Vendor lock-in, costs can grow, learning curve for AWS services
GKE (Google)	Managed Kubernetes in GCP	Auto-upgrades, strong monitoring and logging, deep GCP integration	Cost, less flexibility in cluster internals
AKS (Azure)	Managed Kubernetes in Azure	Integrated with Azure AD, scaling is simple, automated updates	Dependent on Azure ecosystem, region limitations

Table 1: Comparison of Local vs Production Kubernetes Installation Methods

```
    ports:
    - containerPort: 80
```

Pods are ephemeral. They are rarely created directly in production. Instead, use **Deployments** or higher controllers to manage them.

ASCII Diagram of a Pod:

```
+-----------------------------+
|          Pod: mypod         |
|   +---------------------+   |
|   |   Container: Nginx  |   |
|   |   Port: 80          |   |
|   +---------------------+   |
+-----------------------------+
```

Interviewer might ask: "Can you explain why Pods are not recommended to be created directly in production?"

2.2 ReplicaSets

ReplicaSets ensure that a specified number of Pods are running at all times.

ReplicaSet YAML Example:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myrs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.25
```

Create a ReplicaSet:

```
$ kubectl apply -f replicaset.yaml
```

ReplicaSets are typically managed by Deployments. You rarely create them manually.

Interviewer might ask: "What is the difference between a ReplicaSet and a Deployment?"

2.3 Deployments

A **Deployment** provides declarative updates for Pods and ReplicaSets. It is the most common way to run applications.

Deployment YAML Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: nginx:1.25
        ports:
        - containerPort: 80
```

Useful Deployment Commands:

```
$ kubectl get deployments
$ kubectl describe deployment myapp-deployment
$ kubectl rollout status deployment/myapp-deployment
$ kubectl rollout undo deployment/myapp-deployment
```

Deployments enable rolling updates and rollbacks. This is why they are preferred over direct ReplicaSet management.

2.4 Services

A **Service** exposes Pods to the network. Types include: - ClusterIP (default, internal only) - NodePort (accessible via node's IP and port) - LoadBalancer (uses cloud provider LB)

Service YAML Example:

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: NodePort
  selector:
    app: myapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30007
```

Get Service Details:

```
$ kubectl get svc
$ kubectl describe svc myapp-service
```

In production, **Ingress Controllers** (like NGINX, Traefik) are often used instead of exposing apps via NodePort directly.

2.5 ConfigMaps and Secrets

ConfigMaps store non-sensitive configuration data. **Secrets** store sensitive data (like passwords, tokens).

Create a ConfigMap:

```
$ kubectl create configmap app-config --from-literal=APP_MODE=prod
```

Create a Secret:

```
$ kubectl create secret generic db-secret \
  --from-literal=DB_USER=admin \
  --from-literal=DB_PASS=Pa55w0rd
```

Pod Using ConfigMap and Secret:

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  containers:
  - name: app
    image: nginx
    env:
    - name: APP_MODE
      valueFrom:
        configMapKeyRef:
          name: app-config
          key: APP_MODE
    - name: DB_PASS
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: DB_PASS
```

Always use **Secrets** for sensitive data, and integrate with external secret managers (Vault, AWS Secrets Manager) in production.

Interviewer might ask: "How do ConfigMaps and Secrets differ? Can Secrets be encrypted at rest?"

3 Kubernetes Networking (ClusterIP, NodePort, LoadBalancer, Ingress, CNIs)

Networking is the backbone of Kubernetes. Every Pod gets its own IP, but without Services and Ingress, Pods cannot reliably communicate or be accessed externally.

3.1 ClusterIP Service

The default Service type. Exposes Pods internally within the cluster.

ClusterIP Service Example:

```
apiVersion: v1
kind: Service
metadata:
  name: internal-svc
spec:
  type: ClusterIP
  selector:
    app: backend
  ports:
  - port: 8080
    targetPort: 8080
```

Check ClusterIP Service:

```
$ kubectl get svc internal-svc
```

ClusterIP is ideal for microservices communicating internally, but it is not accessible outside the cluster.

ASCII Diagram:

```
[ Pod A ] ---> [ ClusterIP Service ] ---> [ Pod B ]
   (10.1.1.5)       (10.96.0.12)            (10.1.1.7)
```

Interviewer might ask: "What is the default Service type in Kubernetes?"

3.2 NodePort Service

Exposes a Service on each node's IP at a static port (default range: 30000–32767).

NodePort Service Example:

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-svc
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30080
```

Access NodePort:

```
http://<NodeIP>:30080
```

NodePort is simple but not scalable. Use only for dev/test or where external load balancers are unavailable.

3.3 LoadBalancer Service

In cloud environments (AWS, GCP, Azure), Kubernetes can provision a cloud load balancer automatically.

LoadBalancer Service Example:

```
apiVersion: v1
kind: Service
metadata:
  name: lb-svc
spec:
  type: LoadBalancer
  selector:
    app: web
  ports:
  - port: 80
    targetPort: 80
```

Check External IP:

```
$ kubectl get svc lb-svc
```

LoadBalancers integrate with cloud providers (ELB, ALB, etc). In on-premise setups, you need MetalLB or similar solutions.

Interviewer might ask: “How does a LoadBalancer Service differ from a NodePort Service?”

3.4 Ingress Controller

Ingress provides external HTTP/HTTPS access, with features like path-based and host-based routing.

Ingress Example:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - host: myapp.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-svc
            port:
              number: 80
```

Apply Ingress:

```
$ kubectl apply -f ingress.yaml
```

Ingress needs an Ingress Controller (e.g., NGINX, Traefik, HAProxy). Without one, Ingress objects won't function.

ASCII Diagram:

```
   [ Internet ]
        |
   [ Ingress Controller ]
        |
   +----+-------------+
   |                  |
[ frontend-svc ]   [ backend-svc ]
```

3.5 CNI Plugins (Container Network Interface)

Kubernetes delegates networking to **CNI plugins**. They provide Pod IP assignment and routing.

Popular CNIs: - Flannel (simple overlay networking) - Calico (network policy + BGP support) - Cilium (eBPF-powered, advanced security) - Weave Net (mesh-based)

Install Calico CNI:

```
$ kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

Choice of CNI impacts cluster performance, security, and scalability. In production, Calico or Cilium is often recommended.

Interviewer might ask: "What's the role of CNI in Kubernetes, and how does it differ from kube-proxy?"

Pages 25–30: Kubernetes Storage (Volumes, PVs, PVCs, StorageClasses, CSI)

Why Storage in Kubernetes Matters?

Kubernetes workloads are often **stateful**. While containers are ephemeral, data persistence is critical for databases, message queues, and production workloads. Kubernetes provides multiple storage abstractions to decouple storage management from pods.

—

Kubernetes Volume

- A **Volume** in Kubernetes is attached to a Pod and provides data persistence as long as the Pod exists. - Once the Pod is deleted, the Volume data is lost (unless backed by a persistent storage layer).

Example: Using an emptyDir Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: app
      image: nginx
      volumeMounts:
        - mountPath: /usr/share/nginx/html
          name: html
  volumes:
    - name: html
      emptyDir: {}
```

Tip: Use `emptyDir` for temporary cache/storage. Data is deleted once the Pod is removed.

—

Persistent Volumes (PV)

- PV is a **cluster-wide resource** representing actual storage (NFS, EBS, GCE Persistent Disk, Ceph, etc.). - Created and managed by administrators.

Example: PersistentVolume YAML

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data
```

—

Persistent Volume Claims (PVC)

- PVCs are requests for storage made by **users/applications**. - Binds automatically to an available PV that matches requirements.

Example: PVC YAML

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc1
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

Bind Process: - User creates PVC - Kubernetes matches PVC with available PV - PVC is bound to PV if requirements match

—

Using PVC in Pods

Pod with PVC

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-pvc
spec:
  containers:
    - name: app
      image: nginx
      volumeMounts:
        - mountPath: "/data"
          name: mydata
  volumes:
    - name: mydata
      persistentVolumeClaim:
        claimName: pvc1
```

—

Storage Classes

- Provide **dynamic provisioning** of PVs. - Instead of admins pre-creating PVs, a **StorageClass** automatically provisions storage when PVCs are requested.

Example: StorageClass (AWS EBS)

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Delete
```

Best Practice: Always use `StorageClass` for production clusters to avoid manual PV management.

—

Container Storage Interface (CSI)

- CSI is a standard API for integrating **third-party storage providers**. - Example: Ceph, Portworx, OpenEBS, AWS EBS, Azure Disk, GCP PD.

- Allows vendors to build plugins.
- Kubernetes supports CSI drivers natively.

Example CSI Drivers: - `ebs.csi.aws.com` (AWS EBS) - `pd.csi.storage.gke.io` (GCP Persistent Disk) - `disk.csi.azure.com` (Azure Disk)

—

Comparison Table: PV vs PVC vs StorageClass

Concept	Who Creates?	Purpose
PersistentVolume (PV)	Cluster Admin	Represents actual storage (EBS, NFS, etc.)
PersistentVolumeClaim (PVC)	User/Developer	Requests storage (claims PV)
StorageClass	Cluster Admin	Defines dynamic provisioning rules

—

Interview Style Q/A

Q1. How does PVC bind to a PV? **A1.** PVC requests resources, Kubernetes checks available PVs, and binds one that satisfies the request. If none available, PVC stays in `Pending` state.

Q2. What happens if a PVC requests 10Gi but only 5Gi PVs exist? **A2.** PVC will remain unbound (`Pending`) until a matching PV is created.

Q3. When should we use StorageClass vs static PVs? **A3.** Use StorageClass for production with dynamic workloads, static PVs for fixed storage (like legacy NFS mounts).

—

ASCII Diagram: Storage Flow

```
+-------------+     +-------------+     +-------------+
|   PVC       | --> |   PV        | --> |   Storage   |
| (Developer) |     | (Admin)     |     | (EBS/NFS)   |
+-------------+     +-------------+     +-------------+
```

—

Best Practices for Storage in Production:

- Use **StorageClasses** with dynamic provisioning.
- Set **reclaimPolicy** to `Retain` for critical data.
- Always encrypt volumes (EBS, GCP PD, Azure Disk).
- Monitor PVC states with `kubectl get pvc`.

Pages 31–36: Kubernetes Workloads (ReplicaSets, StatefulSets, DaemonSets, Jobs, CronJobs)

Why Workloads Matter?

Workloads in Kubernetes define **how applications run** on clusters. Pods are the smallest unit, but workloads manage Pods at scale by controlling:

- Replication (scaling horizontally)
- Scheduling across nodes
- Persistent identity/state
- Batch processing
- Cluster-wide tasks

—

ReplicaSet (RS)

- Ensures a specified number of Pod replicas are running. - Replacement for older `ReplicationController`. - Typically used indirectly via `Deployment`.

ReplicaSet Example

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```

Command: Check RS

```
$ kubectl get rs
```

Tip: Always prefer `Deployment` over raw ReplicaSet for easier rollouts/rollbacks.

—

StatefulSet

- Manages Pods that require **stable network identity, persistent storage, and ordered deployment/termination**. - Used for **databases** (MySQL, Cassandra, Kafka, etc.).

StatefulSet Example

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: "mysql"
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:5.7
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-persistent-storage
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Observation: - Pods are named `mysql-0`, `mysql-1`, `mysql-2`. - Each Pod has its own PVC.

Best Practice: Use StatefulSets for applications that cannot tolerate data loss and require unique identities.

—

DaemonSet

- Ensures that **one Pod runs on every node**. - Common for logging, monitoring, and networking agents (Fluentd, Prometheus Node Exporter, CNI).

DaemonSet Example

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-logger
spec:
  selector:
    matchLabels:
      app: logger
  template:
    metadata:
      labels:
        app: logger
    spec:
      containers:
      - name: fluentd
        image: fluent/fluentd
```

Command: Check DaemonSet

```
$ kubectl get ds -A
```

Use Case: Perfect for cluster-wide services like log collectors, monitoring agents, or CNIs.

—

Job

- Runs Pods to **completion**. - Retries failed Pods until success or retry limit. - Used for batch workloads (data processing, backup jobs).

Job Example

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-job
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

Check Job Status:

```
$ kubectl get jobs
```

—

CronJob

- Runs Jobs on a **schedule**, like Linux cron. - Perfect for backups, scheduled reports, or periodic tasks.

CronJob Example

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello-cron
spec:
  schedule: "*/5 * * * *"   # every 5 minutes
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from Kubernetes
          restartPolicy: OnFailure
```

Command: Check CronJob

```
$ kubectl get cronjobs
```

—

Comparison Table of Workloads

Workload	**Use Case**	**Key Features**
ReplicaSet	Maintain desired number of Pods	Self-healing, scaling, but no rollout/rollback
StatefulSet	Databases, Kafka, Zookeeper	Stable identity, ordered start/stop, persistent storage
DaemonSet	Logging, Monitoring, CNIs	One Pod per node, cluster-wide service
Job	Batch tasks	Runs to completion, retries on failure
CronJob	Scheduled tasks	Periodic execution (like cron)

—

Interview Style Q/A

Q1. How is a StatefulSet different from a ReplicaSet? **A1.** StatefulSets give each Pod a stable identity and persistent storage, while ReplicaSets treat all Pods as identical and replaceable.

Q2. Why use DaemonSets instead of Deployments? **A2.** DaemonSets guarantee one Pod per node (node agents), Deployments may schedule Pods unevenly across nodes.

Q3. What happens if a CronJob is scheduled every minute but takes 2 minutes to run? **A3.** Overlapping Jobs may run unless `concurrencyPolicy` is set to `Forbid`.

—

ASCII Diagram: Workload Controllers

```
                  +---------------------+
                  |   Workload API      |
                  +----------+----------+
                             |
       ----------------------------------------------
       |          |             |          |        |
 ReplicaSet  StatefulSet   DaemonSet   Job     CronJob
       |             |            |         |        |
     Pods         Pods w/ID     Pods/node   Pods   Scheduled Jobs
```

Best Practices:

- Use Deployments (built on ReplicaSets) for stateless apps.
- Use StatefulSets only when apps need stable identity + storage.
- Monitor DaemonSets carefully to ensure agents run on all nodes.
- For CronJobs, set `concurrencyPolicy` and resource limits.

Pages 37–42: Kubernetes Config & Security (RBAC, Service Accounts, Network Policies, Pod Security, Secrets Encryption)

Why Security Matters in Kubernetes?

Kubernetes clusters often run multi-tenant workloads across shared infrastructure. Without proper security:

- Pods may access resources they shouldn't.
- Unauthorized users could escalate privileges.
- Secrets may leak in plaintext.
- Network traffic could be sniffed or manipulated.

—

RBAC (Role-Based Access Control)

- Controls **who can do what** in the cluster. - Managed via `Role`, `ClusterRole`, `RoleBinding`, `ClusterRoleBinding`.

Create a Role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: dev
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

Bind Role to a User:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: dev
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Check RBAC Permissions:

```
$ kubectl auth can-i get pods --as alice -n dev
```

Best Practice: Follow the principle of *least privilege*. Grant only required permissions.

—

Service Accounts

- Pods authenticate to the API Server via ServiceAccounts. - By default, each Pod gets a ServiceAccount token mounted.

Create ServiceAccount:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-bot
  namespace: ci
```

Assign Role to ServiceAccount:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: sa-binding
  namespace: ci
subjects:
- kind: ServiceAccount
  name: build-bot
  namespace: ci
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Tip: Use dedicated ServiceAccounts per workload. Rotate tokens frequently.

—

Network Policies

- Define how Pods communicate with each other and external systems. - By default, all traffic is allowed (no isolation).

Restrict Pod-to-Pod Traffic:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
  namespace: prod
spec:
  podSelector:
    matchLabels:
      app: database
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: backend
```

Result: Only Pods with label `app=backend` can connect to Pods with `app=database`.

Best Practice: Start with a default deny policy, then explicitly allow only required traffic.

—

Pod Security (Pod Security Admission / PSA)

- Replaces PodSecurityPolicy (deprecated). - Three modes:

- **Privileged** – unrestricted.
- **Baseline** – prevents known privilege escalations.
- **Restricted** – strictest, enforcing least privilege.

Example Namespace with Restricted Policy:

```
apiVersion: v1
kind: Namespace
metadata:
  name: restricted-ns
  labels:
    pod-security.kubernetes.io/enforce: "restricted"
    pod-security.kubernetes.io/audit: "baseline"
    pod-security.kubernetes.io/warn: "baseline"
```

Tip: Apply PSA labels at namespace level for easier enforcement.

—

Secrets Encryption

- By default, Secrets are stored in **etcd** in plaintext. - Encrypt Secrets at rest using KMS providers.

Enable Encryption at Rest:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
- resources:
  - secrets
  providers:
  - kms:
      name: aws-kms
      endpoint: unix:///var/run/kmsplugin/socket
  - identity: {}
```

Check Secrets:

```
$ kubectl get secrets
```

Best Practice: Always enable encryption at rest. Limit etcd access to admins only.

—

ASCII Diagram: Security Layers

```
              +----------------------+
              |   Kubernetes API     |
              +----------------------+
                     | AuthN/AuthZ
---------------------------------------------------
| RBAC | Service Accounts | Network Policies    |
---------------------------------------------------
            | Pod Security Admission |
---------------------------------------------------
          | Secrets Encryption (etcd) |
```

—

Interview Style Q/A

Q1. How would you prevent a Pod from accessing all Secrets in the namespace? **A1.** Use dedicated ServiceAccounts with limited RBAC roles. Avoid mounting default ServiceAccount tokens.

Q2. What's the difference between Role and ClusterRole? **A2.** Role is namespace-scoped, ClusterRole is cluster-wide.

Q3. How do you enforce network isolation between teams in the same cluster? **A3.** Use NetworkPolicies with namespace + label selectors to restrict cross-namespace communication.

Q4. Where are Kubernetes Secrets stored, and how do you secure them? **A4.** Secrets are stored in etcd. Secure them using encryption at rest + RBAC + etcd TLS.

—

Key Takeaways:

- RBAC controls API access (*who can do what*).
- ServiceAccounts enable secure Pod-to-API communication.
- NetworkPolicies restrict Pod-to-Pod traffic.
- Pod Security Admission enforces workload isolation.
- Secrets must be encrypted at rest and in transit.

Pages 43–48: Kubernetes Scaling & High Availability

Scaling and HA (High Availability) are critical for running production-grade Kubernetes clusters. This section covers Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA), Cluster Autoscaler, and multi-master HA setups.

Horizontal Pod Autoscaler (HPA)

HPA automatically scales the number of pods in a Deployment, ReplicaSet, or StatefulSet based on observed CPU/memory utilization or custom metrics.

Enable Metrics Server (required for HPA):

```
$ kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest
```

Create a Deployment:

```
$ kubectl create deployment nginx --image=nginx --replicas=1
```

Expose the Deployment:

```
$ kubectl expose deployment nginx --port=80 --type=ClusterIP
```

Create an HPA (scale between 1 and 10 pods at 50% CPU):

```
$ kubectl autoscale deployment nginx --cpu-percent=50 --min=1 --max=10
```

Check HPA status:

```
$ kubectl get hpa
NAME    REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
nginx   Deployment/nginx   25%/50%   1         10        2          5m
```

Best Practice: Always run a load generator to test HPA, e.g., `kubectl run -it load-generator --image=busybox /bin/sh`

Vertical Pod Autoscaler (VPA)

VPA adjusts CPU/memory *requests and limits* for pods automatically.

Install VPA components:

```
$ git clone https://github.com/kubernetes/autoscaler.git
$ cd autoscaler/vertical-pod-autoscaler
$ kubectl apply -f deploy/
```

Create VPA resource:

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: nginx-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: nginx
  updatePolicy:
    updateMode: "Auto"
```

Real-world insight: HPA & VPA can conflict (HPA scales pods, VPA modifies pod requests). Use them carefully together!

Cluster Autoscaler

Cluster Autoscaler scales the number of nodes in your cluster based on pod scheduling needs.

Install Cluster Autoscaler on AWS EKS:

```
$ kubectl apply -f \
https://github.com/kubernetes/autoscaler/releases/download/cluster-autoscaler-chart/
```

Check logs:

```
$ kubectl -n kube-system logs -f deployment/cluster-autoscaler
```

Interview Note: Q: How does Cluster Autoscaler decide to add/remove nodes? A: It checks for unschedulable pods and underutilized nodes. It integrates with cloud APIs (AWS, GCP, Azure).

High Availability (HA) Setup

Why HA? To avoid single points of failure, Kubernetes production clusters run multiple masters (control plane nodes).

```
      +----------------------+
      |    Load Balancer     |
      +----------+-----------+
                 |
    +------------+------------+
    |     |            |      |
+---+---+   +---+---+   +---+---+
|Master1|   |Master2|   |Master3|
+---+---+   +---+---+   +---+---+
    |            |          |
    +------------+----------+
                 |
          +------+------+
          |   Worker    |
          +-------------+
```

Initialize HA cluster (example with kubeadm):

```
$ kubeadm init --control-plane-endpoint "LOADBALANCER:6443" \
  --upload-certs
```

Join additional control-plane nodes:

```
$ kubeadm join LOADBALANCER:6443 --token <token> \
  --discovery-token-ca-cert-hash sha256:<hash> \
  --control-plane --certificate-key <key>
```

Best Practice: Always configure an external etcd cluster with odd members (3/5) for fault tolerance.

Local vs Production Scaling

Feature	Local (Minikube/Kind)	Production (Cloud/Kubeadm)
HPA Testing	Yes, using metrics-server	Yes, integrates with Prometheus/custom metrics
VPA	Limited setup	Full support
Cluster Autoscaler	Not available	Fully supported with cloud APIs
HA Setup	Single control-plane	Multi-master with load balancer

Interview Tip: *Q: What's the difference between HPA and Cluster Autoscaler?* A: HPA scales **pods**, Cluster Autoscaler scales **nodes**.

Pages 49–54: Kubernetes Monitoring & Logging

Monitoring and logging are essential pillars of Kubernetes observability. They help ensure application health, troubleshoot failures, and optimize resource usage.

Prometheus Monitoring

Prometheus is the de facto standard for metrics in Kubernetes.

Install Prometheus via Helm:

```
$ helm repo add prometheus-community https://prometheus-community.github.io/helm-cha
$ helm repo update
$ helm install prometheus prometheus-community/kube-prometheus-stack
```

Access Prometheus UI:

```
$ kubectl port-forward svc/prometheus-operated 9090:9090 -n default
```

Best Practice: Always scrape Kubernetes components (kubelet, API server, etcd) for cluster-wide insights.

Interview Note: Q: How does Prometheus discover targets in Kubernetes? A: It uses **service discovery** via Kubernetes API to dynamically discover pods/endpoints.

Grafana Visualization

Grafana is used to visualize Prometheus metrics and create dashboards.

Access Grafana:

```
$ kubectl port-forward svc/prometheus-grafana 3000:80 -n default
```

Default Credentials:

```
Username: admin
Password: prom-operator
```

Best Practice: Import Kubernetes dashboards from Grafana Labs (e.g., ID: 315) for ready-to-use monitoring.

```
+-------------------+      +-------------+
|  Prometheus       | ---> |   Grafana   |
| (Metrics store)   |      | Dashboards  |
+-------------------+      +-------------+
```

EFK Stack (Elasticsearch, Fluentd, Kibana)

EFK is commonly used for log aggregation and visualization.

Deploy EFK Stack:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/efk/main/elasticsear
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/efk/main/fluentd.yam
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/efk/main/kibana.yaml
```

Access Kibana UI:

```
$ kubectl port-forward svc/kibana 5601:5601
```

Real-world Insight: Fluentd collects logs from nodes and pods → Elasticsearch indexes logs → Kibana queries logs.

Interview Note: Q: Why Fluentd over Filebeat? A: Fluentd is more extensible (plugins), while Filebeat is lightweight. Choice depends on use case.

Loki + Grafana for Logging

Loki is a log aggregation system designed by Grafana Labs. Unlike Elasticsearch, it stores metadata efficiently and queries logs via labels.

Install Loki via Helm:

```
$ helm repo add grafana https://grafana.github.io/helm-charts
$ helm install loki grafana/loki-stack
```

Query logs in Grafana (Loki Data Source):

```
{app="nginx"} |= "error"
```

```
+------------+       +---------+       +----------+
|  Pods      |  -->  | Loki    |  -->  | Grafana |
| (Logs)     |       | Store   |       | Explore |
+------------+       +---------+       +----------+
```

Best Practice: Use Loki when logs are high-volume but you need cheap storage with fast search by labels.

Comparison: EFK vs Loki

Feature	**EFK (Elasticsearch)**	**Loki**
Storage	Heavy (indexes logs fully)	Lightweight (indexes metadata only)
Query	Full-text search	Label-based search
Performance	High resource usage	Optimized for cost and scale
Best Use Case	Complex log analysis	Lightweight, cheap log storage

Local vs Production Monitoring & Logging

Feature	**Local (Minikube/Kind)**	**Production (EKS/GKE/AKS)**
Prometheus	Runs inside cluster	Integrates with managed Prometheus/Grafana
Grafana	Manual port-forwarding	Exposed via Ingress/LoadBalancer
Logs	kubectl logs	Centralized logging (EFK/Loki)
HA Setup	Single-node components	Multi-node Elasticsearch / Loki clusters

Interview Style Notes

- Q: How would you troubleshoot a pod crash using logs? A: First use `kubectl logs <pod>`, then check central logging system (EFK/Loki) for history.
- Q: Difference between monitoring and logging? A: Monitoring = metrics (quantitative), Logging = events (qualitative).

- Q: Why do we need both Prometheus and Grafana? A: Prometheus stores metrics, Grafana visualizes them.

Key Takeaway: Prometheus + Grafana = Metrics/Monitoring. EFK/Loki + Grafana = Logs/Observability. Together they provide full observability.

Pages 55–60: Kubernetes CI/CD & GitOps

Continuous Integration and Continuous Delivery (CI/CD) combined with GitOps practices ensures reliable and automated application deployments on Kubernetes.

Jenkins + Kubernetes CI/CD

Jenkins can integrate with Kubernetes for dynamic build agents and automated deployments.

Install Jenkins on Kubernetes (Helm):

```
$ helm repo add jenkins https://charts.jenkins.io
$ helm install jenkins jenkins/jenkins
```

Expose Jenkins:

```
$ kubectl port-forward svc/jenkins 8080:8080
```

Default Credentials:

```
Username: admin
Password: <kubectl get secret jenkins ...>
```

Best Practice: Use Jenkins Kubernetes Plugin to launch ephemeral pods as agents instead of static build nodes.

Interview Note: Q: How can Jenkins dynamically scale builds in Kubernetes? A: Jenkins spawns ephemeral pods (agents) inside the cluster for each build job.

ArgoCD for GitOps

ArgoCD is a declarative, GitOps continuous delivery tool for Kubernetes.

Install ArgoCD:

```
$ kubectl create namespace argocd
$ kubectl apply -n argocd -f \
https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

Access ArgoCD UI:

```
$ kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Deploy App via GitOps:

```
$ argocd app create nginx-app \
--repo https://github.com/myorg/nginx-k8s.git \
--path manifests \
--dest-server https://kubernetes.default.svc \
--dest-namespace default
```

Best Practice: Treat Git as the single source of truth. All changes must go via pull requests.

```
+----------+       +----------+       +--------------+
|   Git    | --->  | ArgoCD   | --->  | Kubernetes   |
|  (Repo)  |       | (Sync)   |       | (Cluster)    |
+----------+       +----------+       +--------------+
```

Interview Note: Q: Difference between ArgoCD and Helm? A: Helm manages packaging/deployment. ArgoCD automates deployments by syncing Git state with cluster state.

FluxCD (GitOps Alternative)

FluxCD is another GitOps operator for Kubernetes.

Install FluxCD:

```
$ curl -s https://fluxcd.io/install.sh | sudo bash
$ flux check --pre
```

Bootstrap Flux with GitHub Repo:

```
$ flux bootstrap github \
  --owner=myorg \
  --repository=k8s-gitops \
  --branch=main \
  --path=./clusters/my-cluster
```

Best Practice: FluxCD excels at multi-tenant GitOps setups with fine-grained RBAC.

Interview Note: Q: Why GitOps instead of manual kubectl apply? A: GitOps ensures consistency, version control, and rollback capability by using Git as the truth source.

CI/CD Workflow in Kubernetes

```
          +---------+
          |  Devs   |
          +---------+
               |
               v
         +-----------+
         |   Git     |
         +-----------+
               |
       +-------+-------+
       | Jenkins CI    |
       +---------------+
               |
        Build -> Test -> Push Image
               |
               v
       +---------------+
       | ArgoCD/FluxCD |
       +---------------+
               |
               v
         +-----------+
         | Kubernetes|
+-----------+
```

Local vs Production CI/CD & GitOps

Aspect	Local (Minikube/Kind)	Production (EKS/GKE/AKS)
CI/CD Runner	Jenkins pod in local cluster	Jenkins + cloud runners (scalable)
GitOps Tool	ArgoCD/Flux in dev mode	HA ArgoCD/Flux with SSO, RBAC
Image Registry	Local Docker daemon	Cloud registries (ECR, GCR, ACR)
Secrets Mgmt	Kubernetes Secrets	Vault / KMS integrated

Interview Style Notes

- Q: How does GitOps improve security? A: Prevents direct cluster access; all changes go via Git.
- Q: Jenkins vs ArgoCD — do they compete? A: No, Jenkins = CI, ArgoCD = CD. They complement each other.
- Q: How to roll back deployment in GitOps? A: Revert commit in Git; ArgoCD auto-syncs rollback.

Key Takeaway: Jenkins = Build + Test (CI). ArgoCD/FluxCD = Deploy + Sync (CD). Together = End-to-end GitOps CI/CD on Kubernetes.

Pages 61–66: Kubernetes Advanced Scheduling

Kubernetes scheduling decides **which Pod runs on which Node**. Advanced scheduling ensures workloads are placed optimally considering performance, HA, and business policies.

NodeSelectors

The simplest way to constrain Pods to nodes using labels.

Label a Node:

```
$ kubectl label nodes worker1 disktype=ssd
```

Pod with NodeSelector:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-ssd
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    disktype: ssd
```

Best Practice: Use NodeSelectors for simple constraints. For advanced rules, use `affinity`.

Taints & Tolerations

Taints repel pods; tolerations let pods run on tainted nodes.

Apply a Taint:

```
$ kubectl taint nodes worker1 dedicated=ml:NoSchedule
```

Pod with Toleration:

```
apiVersion: v1
kind: Pod
metadata:
  name: ml-pod
spec:
  tolerations:
  - key: "dedicated"
    operator: "Equal"
    value: "ml"
    effect: "NoSchedule"
  containers:
  - name: ml
    image: tensorflow/tensorflow
```

Real-World: Taints are used to dedicate GPU nodes for ML workloads, ensuring regular apps don't get scheduled there.

Affinity & Anti-Affinity

Affinity allows advanced matching; Anti-affinity avoids co-location.

Pod with Node Affinity:

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
```

Pod with Pod Anti-Affinity:

```
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - nginx
        topologyKey: "kubernetes.io/hostname"
```

Best Practice: Pod Anti-Affinity ensures replicas of the same app do not land on the same node — improves HA.

Topology Spread Constraints

Ensures pods are evenly distributed across zones, regions, or nodes.

Pod with Topology Spread:

```
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: "topology.kubernetes.io/zone"
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        app: web
```

```
+---------+---------+---------+
| Zone A  | Zone B  | Zone C  |
|   Pod1  |   Pod2  |   Pod3  |
+---------+---------+---------+
```

Real-World: Used to ensure even distribution across availability zones in multi-AZ clusters.

Preemption & Priorities

High-priority pods can evict lower-priority pods.

Create PriorityClass:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 100000
globalDefault: false
description: "Critical pods"
```

Use Priority in Pod:

```
spec:
  priorityClassName: high-priority
```

Best Practice: Assign higher priority to system-critical workloads like DNS or monitoring agents.

Interview Style Questions

- Q: What's the difference between NodeSelector and NodeAffinity? A: NodeSelector is simple exact match; NodeAffinity allows complex rules with operators.
- Q: How to dedicate nodes for a specific workload type? A: Use `taints` on nodes and matching `tolerations` in pods.
- Q: How do Topology Spread Constraints help in HA? A: They distribute replicas evenly across zones/nodes to avoid single-point failures.
- Q: What happens if a high-priority pod cannot be scheduled? A: It can preempt (evict) lower-priority pods to free resources.

Pages 67–72: Kubernetes Multi-Tenancy & Namespaces

Concept Overview

Multi-tenancy in Kubernetes allows multiple teams, applications, or environments to securely share the same cluster. This is usually achieved by isolating workloads with **Namespaces**, applying resource constraints, and enforcing security boundaries.

Namespaces are the foundation of multi-tenancy. Think of them like separate "folders" inside a cluster, each containing its own resources (Pods, Services, ConfigMaps, etc.).

Namespaces

Create a new Namespace:

```
$ kubectl create namespace dev-team
```

Output:

```
namespace/dev-team created
```

List all Namespaces:

```
$ kubectl get namespaces
```

```
NAME            STATUS   AGE
default         Active   10d
kube-system     Active   10d
kube-public     Active   10d
dev-team        Active   2m
```

Resource Quotas

Resource quotas enforce limits on CPU, memory, and object counts (e.g., Pods, Services) per namespace.

Example ResourceQuota:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-team-quota
  namespace: dev-team
spec:
  hard:
    requests.cpu: "2"
    requests.memory: 2Gi
    limits.cpu: "4"
    limits.memory: 4Gi
    pods: "10"
```

Apply ResourceQuota:

```
$ kubectl apply -f resource-quota.yaml
```

Quotas prevent a single team from consuming all cluster resources.

LimitRanges

`LimitRange` defines default and max/min resource requests and limits per Pod/Container.

Example LimitRange:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: dev-limits
  namespace: dev-team
```

```
spec:
  limits:
  - default:
      cpu: 500m
      memory: 512Mi
    defaultRequest:
      cpu: 200m
      memory: 256Mi
    type: Container
```

Use LimitRanges with ResourceQuotas to enforce fairness between teams.

Multi-Cluster Management

Large organizations often require multiple clusters (e.g., for different regions, compliance, or isolation). Common solutions:

- **Cluster Federation (KubeFed)** – sync resources across clusters.
- **Service Mesh (Istio/Linkerd)** – enable cross-cluster communication.
- **Multi-Cluster Tools:** Rancher, Anthos, OpenShift ACM, ArgoCD.

ASCII: Multi-Tenant Cluster with Namespaces +———————- Kubernetes Cluster — ————————+ — — — [Namespace: dev-team] [Namespace: qa-team] — — Pods, Services, PVCs Pods, Services, PVCs — — — — [Namespace: prod-team] — — Apps, DBs, Configs — +————————————————————+

Interview Notes

- Q: How do you isolate teams in the same cluster? **A:** By creating namespaces, applying ResourceQuotas, LimitRanges, and RBAC.
- Q: What is the difference between ResourceQuota and LimitRange? **A:** ResourceQuota applies at the namespace level; LimitRange applies at the container/pod level.
- Q: How would you manage multiple Kubernetes clusters across regions? **A:** Using federation or multi-cluster management tools like Rancher, Anthos, or ArgoCD.

Best Practices

- Always assign workloads to namespaces (avoid using `default`).
- Combine RBAC with namespaces for strong security isolation.
- Apply quotas and limits early to avoid noisy neighbor problems.
- Use multi-cluster only if compliance/regional isolation demands it.

Key Takeaway: Advanced scheduling in Kubernetes = Fine-grained workload placement, higher reliability, and optimal resource usage.

Pages 73–78: Kubernetes Upgrades & Maintenance

Concept Overview

Kubernetes clusters evolve rapidly, with frequent releases that introduce new features, bug fixes, and security patches. Proper upgrade and maintenance strategies are critical to ensuring cluster stability and availability.

Golden Rule: Always test upgrades in a staging/pre-production cluster before applying them to production.

Kubernetes Versioning

Kubernetes follows a **semantic versioning** pattern: `MAJOR.MINOR.PATCH`.

- **Patch** (e.g., 1.28.3 → 1.28.4): bug fixes, no new features.
- **Minor** (e.g., 1.27.x → 1.28.x): new features, may deprecate old APIs.
- **Major** (rare): breaking changes.

Check Cluster Version:

```
$ kubectl version --short
```

Output:

```
Client Version: v1.28.3
Server Version: v1.27.5
```

ETCD Backup & Restore

ETCD is the key-value store that holds all cluster state. Backups are critical before upgrades.

Backup ETCD (example static pod setup):

```
$ ETCDCTL_API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  snapshot save /backup/etcd-snapshot.db
```

Restore ETCD Snapshot:

```
$ ETCDCTL_API=3 etcdctl snapshot restore /backup/etcd-snapshot.db \
  --data-dir /var/lib/etcd-from-backup
```

Always back up ETCD before any upgrade or critical maintenance task.

Node Upgrades & Draining

Nodes must be upgraded carefully to avoid workload disruption.

Drain a Node before upgrade:

```
$ kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data
```

Uncordon Node after upgrade:

```
$ kubectl uncordon <node-name>
```

ASCII: Node Upgrade Workflow +————————————-+ — 1. Drain node — — 2. Upgrade kubelet/kubectl — — 3. Restart node services — — 4. Uncordon node — +————————————-+

Cluster Upgrade with kubeadm

Upgrade kubeadm tool:

```
$ sudo apt-get update && sudo apt-get install -y kubeadm=1.28.x-00
```

Plan upgrade:

```
$ sudo kubeadm upgrade plan
```

Apply upgrade:

```
$ sudo kubeadm upgrade apply v1.28.3
```

Master nodes should be upgraded one by one to maintain quorum.

Patching & Maintenance

- **Security Patches:** Regularly update kubelet, kubeadm, and kubectl binaries.
- **API Deprecation:** Check for deprecated APIs using:

Check API Deprecations:

```
$ kubectl get --raw / | jq .
```

Run `kubectl api-resources` and `kubectl deprecations` (with plugins) to detect outdated APIs.

Interview Notes

- Q: How do you safely upgrade a Kubernetes cluster? **A:** Back up ETCD, drain nodes, upgrade control plane components, upgrade worker nodes, validate cluster health.
- Q: How do you prevent downtime during node upgrades? **A:** Use `kubectl drain` to gracefully evict pods, leverage pod disruption budgets (PDBs).
- Q: What's the difference between patch, minor, and major upgrades? **A:** Patch = bug fixes, Minor = features/deprecations, Major = breaking changes.

Best Practices

- Always perform ETCD backup before upgrades.
- Automate upgrades using CI/CD pipelines where possible.
- Use PodDisruptionBudgets (PDBs) to protect workloads.
- Keep cluster components and CNI plugins aligned with the same version range.

Pages 79–84: Kubernetes Troubleshooting (Pods, Nodes, Networking, CrashLoopBackOff, Events)

Concept Overview

Troubleshooting in Kubernetes involves analyzing pod logs, node states, networking paths, and cluster events to identify the root cause of failures. This section provides structured steps, common commands, and real-world troubleshooting practices.

Golden Rule: Always start from the Pod, move outwards to the Node, Networking, and then Cluster components.

Pod Troubleshooting

Pods are the smallest deployable unit in Kubernetes. Most issues occur here.

Check Pod Status:

```
$ kubectl get pods -n <namespace>
```

Describe Pod (detailed events):

```
$ kubectl describe pod <pod-name> -n <namespace>
```

Check Pod Logs:

```
$ kubectl logs <pod-name> -n <namespace>
```

Stream Logs:

```
$ kubectl logs -f <pod-name> -c <container-name>
```

Common Pod Issues:

- **ImagePullBackOff** → Check image registry access.
- **CrashLoopBackOff** → Application misconfiguration or missing dependency.
- **Pending** → Scheduler cannot place pod (resource shortage or taints).

Node Troubleshooting

Nodes run workloads. If they are unhealthy, pods fail to schedule.

Check Node Status:

```
$ kubectl get nodes
```

Detailed Node Info:

```
$ kubectl describe node <node-name>
```

Check Kubelet Logs:

```
$ journalctl -u kubelet -f
```

Node Troubleshooting Flow: +——————-+ — Node Not Ready — +———+— ——+ — Check Kubelet logs — Verify CNI plugins — Inspect disk/memory

Networking Troubleshooting

Networking issues cause pod-to-pod or external communication failures.

Check Service Endpoints:

```
$ kubectl get endpoints <service-name>
```

Test Pod Connectivity (exec into pod):

```
$ kubectl exec -it <pod-name> -- curl http://<service>:<port>
```

Check DNS Resolution:

```
$ kubectl exec -it <pod-name> -- nslookup <service>
```

If DNS fails, check CoreDNS pods: `kubectl get pods -n kube-system -l k8s-app=kube-dns`

CrashLoopBackOff

A common pod state when containers fail repeatedly.

Check Last Logs of Failed Pod:

```
$ kubectl logs --previous <pod-name>
```

Describe Pod for Events:

```
$ kubectl describe pod <pod-name>
```

Possible Causes:

- Application bug.
- Wrong command/entrypoint in Docker image.
- Missing ConfigMap/Secret.
- Resource limits too strict (CPU throttling).

Events Troubleshooting

Events give cluster-wide context of issues.

Check Recent Events:

```
$ kubectl get events --sort-by=.metadata.creationTimestamp
```

Filter Events by Namespace:

```
$ kubectl get events -n <namespace>
```

Events are stored only for a short time (default 1h). For long-term analysis, integrate with monitoring/logging tools (EFK, Loki).

Interview Notes

- Q: Your pod is stuck in `CrashLoopBackOff`. How will you debug? **A:** Check pod logs, describe pod for events, verify ConfigMaps/Secrets, ensure resource requests are adequate.
- Q: A service is not reachable inside the cluster. How do you approach? **A:** Check endpoints, test pod connectivity with curl, validate DNS resolution, inspect network policies.
- Q: What do you check when a node shows `NotReady`? **A:** Inspect kubelet logs, check CNI plugins, ensure enough CPU/memory, verify etcd and control plane connectivity.

Best Practices

- Automate log and event collection (EFK, Loki).
- Use readiness and liveness probes for self-healing.
- Apply PodDisruptionBudgets (PDBs) to avoid cascading failures.
- Document troubleshooting runbooks for the team.

Pages 85–90: Kubernetes Security & Compliance (Secrets Mgmt, Policies, Auditing, OPA/Gatekeeper, Kyverno)

Concept Overview

Kubernetes security ensures applications and clusters are protected from misconfigurations, vulnerabilities, and unauthorized access. Compliance involves enforcing organizational policies, auditing changes, and verifying cluster integrity.

Secrets Management

Kubernetes Secrets store sensitive data like passwords, tokens, and certificates.

Create a Secret:

```
$ kubectl create secret generic db-secret \
  --from-literal=username=admin \
  --from-literal=password=Passw0rd!
```

Decode Secret:

```
$ kubectl get secret db-secret -o yaml | \
  grep password | awk '{print $2}' | base64 --decode
```

By default, Secrets are only base64-encoded, not encrypted. Enable **Secrets Encryption at Rest** in production clusters.

Pod using a Secret:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-demo
spec:
  containers:
  - name: app
    image: nginx
    env:
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: password
```

Policies & RBAC

Role-Based Access Control (RBAC) defines permissions for users, groups, and service accounts.

Role Example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: dev
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

RoleBinding Example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: dev
subjects:
- kind: User
  name: alice
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Principle of Least Privilege (PoLP): Assign the **minimum required permissions** for each user or service account.

Auditing

Audit logs track requests to the Kubernetes API server.

Check Audit Events:

```
$ kubectl get events --sort-by=.metadata.creationTimestamp
```

Audit Policy Example:

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: Metadata
  resources:
  - group: ""
    resources: ["secrets"]
```

Forward audit logs to SIEM systems for real-time analysis and compliance reports.

OPA Gatekeeper

OPA (Open Policy Agent) with Gatekeeper enforces custom policies.

Disallow Privileged Pods (ConstraintTemplate):

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8sprivileged
spec:
  crd:
    spec:
      names:
        kind: K8sPrivileged
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sprivileged
        violation[{"msg": msg}] {
          input.review.object.spec.containers[_].securityContext.privileged == true
          msg := "Privileged containers are not allowed"
        }
```

OPA Gatekeeper = Policy as Code. Great for enforcing organization-wide rules (e.g., all pods must have resource limits).

Kyverno

Kyverno is a Kubernetes-native policy engine, easier to use than OPA (YAML-based).

Example Kyverno Policy: Enforce Image Registry

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: enforce-image-registry
spec:
  validationFailureAction: enforce
  rules:
  - name: check-registry
    match:
      resources:
        kinds:
        - Pod
    validate:
      message: "Only images from myregistry.com are allowed"
      pattern:
        spec:
          containers:
          - image: "myregistry.com/*"
```

Kyverno is more human-friendly for YAML-based security rules, while OPA is more powerful with Rego language.

Interview Notes

- Q: How do you secure Secrets in Kubernetes? **A:** Enable encryption at rest, restrict RBAC access, use external secret stores like Vault/SealedSecrets.

- Q: What's the difference between OPA Gatekeeper and Kyverno? **A:** Gatekeeper uses Rego (flexible but complex), Kyverno uses YAML policies (easier for teams).
- Q: How do you enforce compliance in Kubernetes clusters? **A:** Apply policies (OPA/Kyverno), enable audit logging, integrate with SIEM, and use CIS Benchmarks.

Best Practices

- Always enable RBAC and disable legacy ABAC.
- Use `PodSecurity` or `PSA` admission for pod-level restrictions.
- Enable audit logs for API calls.
- Integrate policy engines (OPA/Kyverno) for compliance automation.
- Rotate secrets frequently and use external secret managers (HashiCorp Vault, AWS Secrets Manager).

Pages 91–95: Kubernetes Case Studies / Real Production Scenarios

Case Study 1: Outage Due to Misconfigured Liveness Probe

Scenario: A production app kept restarting due to a misconfigured liveness probe. The probe was hitting a non-existent endpoint.

Faulty Probe Configuration:

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 5
```

Best Practice: Always test health endpoints manually before configuring probes. Consider separate endpoints for `/healthz` (liveness) and `/readyz` (readiness).

Resolution: Changed probe to correct path `/actuator/healthz`.

Lesson Learned: Misconfigured probes cause unnecessary restarts and downtime.

Interview Angle: *"What happens if liveness probes are misconfigured in production?"*

—

Case Study 2: Rollback During Bad Deployment

Scenario: A faulty app release caused 500 errors in production. Quick rollback was required.

Rollback Command:

```
$ kubectl rollout undo deployment web-app --to-revision=3
```

Best Practice: Use `kubectl rollout history` to check revisions before rolling back.

```
$ kubectl rollout history deployment web-app
```

Resolution: Rolled back to stable revision, service restored within minutes.
Lesson Learned: Always keep previous stable deployments for fast rollback.
Interview Angle: *"How do you rollback a faulty deployment in Kubernetes?"*

—

Case Study 3: Node Resource Exhaustion

Scenario: Pods went into `Pending` due to no available CPU/memory on worker nodes.

Check Node Resources:

```
$ kubectl describe node worker-1
```

Observed:

```
Allocatable:
  cpu: 2
  memory: 8Gi
Pods: 110/110
Conditions: MemoryPressure=True
```

Resolution: - Added more nodes using cluster autoscaler. - Optimized pod resource requests/limits.

Best Practice: Never overcommit memory in production. Use requests/limits wisely.

Interview Angle: *"What steps do you take if pods remain in Pending state due to lack of resources?"*

—

Case Study 4: Debugging CI/CD Pipeline Failures

Scenario: ArgoCD showed "Sync Failed" due to missing ConfigMap in GitOps repo.

ArgoCD Error:

```
Error: resource ConfigMap "app-config" not found
```

Resolution: - Identified missing YAML in Git repo. - Added ConfigMap manifest and committed. - Triggered ArgoCD sync again.

Best Practice: Ensure manifests for all dependencies are in Git. Use `kubectl diff` or ArgoCD dry-run mode before syncing.

Interview Angle: *"How do you troubleshoot GitOps deployment failures?"*

—

Case Study 5: Network Outage Due to Misconfigured NetworkPolicy

Scenario: A team applied a restrictive NetworkPolicy blocking intra-service communication.

Faulty Policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

Impact: All ingress/egress blocked, microservices unable to talk to each other.
Resolution: Re-applied policy allowing required traffic between services.

Best Practice: Always test NetworkPolicies in staging before production. Start with `default-deny` and explicitly allow traffic.

Interview Angle: *"What issues can occur if NetworkPolicies are misconfigured?"* —

Summary

- Kubernetes outages often arise from misconfigurations (probes, policies, resources).
- Rollback strategies are crucial for minimizing downtime.
- CI/CD pipeline debugging often involves missing manifests or mis-synced GitOps repos.
- Always balance automation with manual checks in production.

Key Takeaway: *"Kubernetes is powerful, but small misconfigurations in YAML or probes can bring down entire production clusters. Real-world resilience comes from automation, monitoring, and rollback readiness."*

Kubernetes Best Practices & Future Trends

As Kubernetes matures, its use cases are expanding from simple container orchestration to powering global-scale, multi-cloud, and AI-driven platforms. This section highlights key production-ready practices and upcoming trends every DevOps professional must know.

1. Resource Management & Efficiency

Always set **resource requests** and **limits** to prevent noisy-neighbor problems in multi-tenant clusters.

Pod with Resource Limits:

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-demo
spec:
  containers:
  - name: app
    image: nginx
    resources:
      requests:
        memory: "128Mi"
        cpu: "250m"
      limits:
        memory: "256Mi"
        cpu: "500m"
```

Best Practice: Use `LimitRanges` + `ResourceQuotas` to enforce policies across namespaces.

2. GitOps as the New Norm

Git is becoming the single source of truth for infrastructure.

Use ArgoCD or FluxCD to continuously reconcile cluster state with Git.

[Git Repo] — ArgoCD/Flux — [Kubernetes Cluster]
Why? Auditable, version-controlled deployments with automatic rollbacks.

3. Multi-Cluster & Hybrid Cloud

Adopt **KubeFed**, **ClusterAPI**, or service meshes like **Istio/Linkerd** for multi-cluster federation.

Use Case: High availability across AWS (EKS) + GCP (GKE).
[Cluster A: AWS EKS] —— — Federation — [Cluster B: GCP GKE] ——
Production Insight: Plan for latency, DNS, and policy consistency across regions.

4. Security-First Approach

Shift-left security: integrate vulnerability scanning (Trivy, Anchore) + policy enforcement (OPA/Gatekeeper, Kyverno).

Scan Images with Trivy:

```
$ trivy image nginx:latest
```

Best Practice: Encrypt secrets at rest, use RBAC minimally, enable audit logging.

5. Serverless on Kubernetes (Knative, KEDA)

Knative: Event-driven serverless platform on top of Kubernetes. **KEDA:** Kubernetes Event-Driven Autoscaler for fine-grained scaling.

Event —¿ KEDA —¿ Scale Pods

Why? Pay-per-use, burst scaling, no idle costs.

6. AI/ML Workloads on Kubernetes

Use **Kubeflow** for ML pipelines, **NVIDIA GPU operators** for GPU scheduling.

Schedule Pod with GPU:

```
resources:
  limits:
    nvidia.com/gpu: 1
```

Future Insight: Kubernetes is becoming the standard platform for ML training + inference.

7. Future Trends in Kubernetes

- **KubeEdge:** Extending Kubernetes to edge/IoT devices.
- **WebAssembly (WASM):** Running ultra-light workloads inside K8s pods.
- **Green Kubernetes:** Optimizing workloads for energy efficiency & sustainability.
- **Policy as Code:** Stronger compliance integration.

8. Interview-Style Questions (Forward-Looking)

- **Q:** How would you secure Kubernetes workloads in a multi-tenant environment?
- **Q:** What's the role of GitOps in enterprise Kubernetes adoption?
- **Q:** How can Kubernetes support serverless and AI/ML workloads?
- **Q:** What trends do you see shaping Kubernetes in the next 5 years?
- **Q:** How would you design a hybrid multi-cloud Kubernetes deployment?

—

Kubernetes is no longer just container orchestration—it is the backbone of cloud-native infrastructure.

Top 25 Kubernetes Real-Time Interview Questions

These are the most frequently asked real-world Kubernetes interview questions, written exactly as interviewers ask them in live interviews.

1. Can you explain the difference between a Pod and a Deployment? In what situations would you use one over the other?
2. How do you troubleshoot a Pod stuck in `CrashLoopBackOff`? Walk me through your step-by-step approach.
3. What is the role of `etcd` in Kubernetes? How do you back it up and restore it in case of cluster failure?
4. Can you explain the difference between ClusterIP, NodePort, and LoadBalancer services? Which one is used in production?
5. How would you expose a Kubernetes application to the outside world? Which option is better — Ingress or LoadBalancer — and why?
6. Walk me through the process of upgrading a Kubernetes cluster in production. What risks do you consider?
7. How does the Horizontal Pod Autoscaler (HPA) work internally? Can you give an example of scaling an app based on CPU and custom metrics?
8. How do you ensure multi-tenancy and resource isolation in Kubernetes clusters?

9. Can you explain the role of the kube-scheduler? How do you influence scheduling decisions using taints, tolerations, and affinity rules?
10. How do you secure secrets in Kubernetes? Would you use plain Secrets, KMS, or external secret managers like HashiCorp Vault?
11. What's the difference between ConfigMaps and Secrets? Can you mount them into Pods?
12. How would you debug networking issues inside a Kubernetes cluster when two Pods cannot communicate?
13. What's the difference between StatefulSet and Deployment? Can you give a real-world scenario where you'd use StatefulSet?
14. What are Kubernetes DaemonSets, and can you give an example of when you deployed one in production?
15. How does the Kubernetes control plane maintain the desired state of the cluster?
16. What's the role of kube-proxy in Kubernetes networking?
17. Have you worked with CSI drivers in Kubernetes? Can you explain how Persistent Volumes and Persistent Volume Claims work in production?
18. How do you monitor Kubernetes clusters in production? Which stack do you prefer (Prometheus, Grafana, Loki, EFK)?
19. Have you implemented GitOps with ArgoCD or FluxCD? How does it improve CI/CD pipelines compared to traditional Jenkins pipelines?
20. How do you manage Kubernetes cluster upgrades in a zero-downtime way?
21. In production, how do you handle resource limits and quotas? Have you faced "OOMKilled" issues, and how did you resolve them?
22. How do you enforce security and compliance in Kubernetes? Have you used OPA/Gatekeeper or Kyverno?
23. Can you explain how Kubernetes handles service discovery internally?
24. What's the difference between Kubernetes in cloud providers (EKS, AKS, GKE) vs. on-prem kubeadm clusters? Which one have you worked with?
25. Let's say a production app is running slow inside Kubernetes. How would you troubleshoot — from Pods → Nodes → Cluster → Infrastructure?

Kubernetes Mega Command Cheat Sheet

- **$ kubectl version** *Check client and server version of Kubernetes.*
- **$ kubectl cluster-info** *Show cluster master and service endpoints.*
- **$ kubectl get nodes** *List all worker and master nodes.*
- **$ kubectl get pods** *List Pods in default namespace.*
- **$ kubectl get pods -A** *List Pods across all namespaces.*
- **$ kubectl describe pod ¡pod¿** *Detailed Pod information.*
- **$ kubectl logs ¡pod¿** *View logs of a Pod.*
- **$ kubectl logs -f ¡pod¿** *Stream Pod logs.*
- **$ kubectl exec -it ¡pod¿ – /bin/sh** *Open shell inside Pod.*
- **$ kubectl create -f pod.yaml** *Create resource from YAML.*
- **$ kubectl delete pod ¡pod¿** *Delete a Pod by name.*
- **$ kubectl get all** *List all resources in namespace.*
- **$ kubectl explain pod** *Show API documentation for Pods.*
- **$ kubectl get namespaces** *List namespaces.*
- **$ kubectl config current-context** *Show current context.*
- **$ kubectl get deployments** *List all Deployments.*
- **$ kubectl describe deployment ¡name¿** *Details of a Deployment.*
- **$ kubectl rollout status deployment/¡name¿** *Check rollout status.*
- **$ kubectl rollout undo deployment/¡name¿** *Rollback Deployment.*
- **$ kubectl scale deployment ¡name¿ –replicas=3** *Scale Deployment.*
- **$ kubectl expose pod ¡pod¿ –type=NodePort –port=80** *Expose Pod via Service.*
- **$ kubectl get services** *List Services.*
- **$ kubectl describe svc ¡name¿** *Show Service details.*

- **$ kubectl port-forward ¡pod¿ 8080:80** *Forward local port to Pod.*
- **$ kubectl cp ¡pod¿:/path /local/path** *Copy files from Pod.*
- **$ kubectl top pods** *CPU and memory usage of Pods.*
- **$ kubectl top nodes** *CPU and memory usage of Nodes.*
- **$ kubectl get events –sort-by=.metadata.creationTimestamp** *Events sorted by time.*
- **$ kubectl cordon ¡node¿** *Mark node unschedulable.*
- **$ kubectl uncordon ¡node¿** *Mark node schedulable again.*
- **$ kubectl drain ¡node¿ –ignore-daemonsets** *Safely evict Pods for maintenance.*
- **$ kubectl taint nodes ¡node¿ key=value:NoSchedule** *Prevent Pods unless tolerated.*
- **$ kubectl label pod ¡pod¿ env=prod** *Add label to a Pod.*
- **$ kubectl annotate pod ¡pod¿ team=devops** *Add annotation to Pod.*
- **$ kubectl api-resources** *List all API resource types.*
- **$ kubectl api-versions** *List supported API versions.*
- **$ kubectl edit deployment ¡name¿** *Edit live Deployment.*
- **$ kubectl proxy** *Start proxy to API server.*
- **$ kubectl apply -f manifests/ –prune** *Apply and remove old resources.*
- **$ kubectl get configmap** *List ConfigMaps.*
- **$ kubectl create configmap app-config –from-literal=key=value** *Create ConfigMap.*
- **$ kubectl get secrets** *List Secrets.*
- **$ kubectl create secret generic my-secret –from-literal=pwd=123** *Create Secret.*
- **$ kubectl auth can-i create pods** *Check RBAC permissions.*
- **$ kubectl get serviceaccounts** *List Service Accounts.*

- **$ kubectl describe clusterrolebinding ¡name¿** *Show RBAC bindings.*
- **$ kubectl get networkpolicy** *List Network Policies.*
- **$ kubectl get ingress** *List Ingress resources.*
- **$ kubectl describe ingress ¡name¿** *Inspect Ingress rules.*
- **$ kubectl get pv** *List PersistentVolumes.*
- **$ kubectl get pvc** *List PersistentVolumeClaims.*
- **$ kubectl describe pvc ¡name¿** *PVC details.*
- **$ kubectl get sc** *List StorageClasses.*
- **$ kubectl describe sc ¡name¿** *Inspect StorageClass.*
- **$ kubectl get volumeattachments** *Check volume bindings.*
- **$ kubectl delete pvc ¡name¿** *Delete PVC safely.*
- **$ kubeadm init** *Initialize Kubernetes master.*
- **$ kubeadm join ¡IP¿:6443 –token ¡token¿ –discovery-token-ca-cert-hash sha256:¡hash¿** *Join worker node.*
- **$ kubeadm reset** *Reset cluster config.*
- **$ kubeadm upgrade plan** *Check upgrade plan.*
- **$ kubeadm upgrade apply v1.29.0** *Upgrade cluster version.*
- **$ kubeadm token list** *List bootstrap tokens.*
- **$ kubeadm certs check-expiration** *Check certificate expiry.*
- **$ helm version** *Check Helm version.*
- **$ helm repo add stable https://charts.helm.sh/stable** *Add Helm repo.*
- **$ helm repo update** *Update chart repo index.*
- **$ helm search repo nginx** *Search charts in repo.*
- **$ helm install my-nginx stable/nginx** *Install chart.*
- **$ helm list** *List Helm releases.*
- **$ helm upgrade my-nginx stable/nginx** *Upgrade release.*

• **$ helm rollback my-nginx 1**	*Rollback release to revision 1.*
• **$ helm uninstall my-nginx**	*Uninstall release.*
• **$ helm template mychart/**	*Render chart templates locally.*
• **$ helm get values my-nginx**	*View values of release.*
• **$ helm history my-nginx**	*Release history.*
• **$ kubectl describe node ¡node¿**	*Inspect node conditions.*
• **$ kubectl get pod ¡pod¿ -o yaml**	*YAML definition of Pod.*
• **$ kubectl get svc ¡svc¿ -o wide**	*Show Service details.*
• **$ kubectl get endpoints ¡svc¿**	*Show Service endpoints.*
• **$ kubectl debug ¡pod¿ –image=busybox**	*Attach debug container.*
• **$ kubectl describe events**	*Inspect recent events.*
• **$ kubectl logs ¡pod¿ –previous**	*View logs of crashed Pod.*
• **$ kubectl get csr**	*List certificate requests.*
• **$ kubectl get componentstatuses**	*Cluster health check.*
• **$ kubectl cluster-info dump**	*Full cluster debug dump.*

Kubernetes Advanced Cheat Sheet (CRDs, Operators, Plugins, Monitoring)

Custom Resource Definitions (CRDs) & Operators

- **$ kubectl get crd**
  List all installed Custom Resource Definitions.

- **$ kubectl describe crd ¡crd-name¿**
  Show detailed information about a specific CRD.

- **$ kubectl get ¡custom-resource¿**
  View instances of a custom resource type (e.g., `kubectl get prometheuses`).

- **$ kubectl apply -f operator.yaml**
  Install an Operator from a YAML manifest.

- **$ kubectl get pods -n ¡operator-namespace¿**
  Verify Operator pods are running in the cluster.
- **$ kubectl logs ¡operator-pod¿ -n ¡namespace¿**
  Check logs of an Operator pod for debugging.
- **$ kubectl delete crd ¡crd-name¿**
  Remove a CRD from the cluster (careful: deletes associated resources).

Kubectl Plugins (krew, ctx, ns)

- **$ kubectl krew install ¡plugin¿**
  Install a kubectl plugin using krew (e.g., `kubectl krew install ctx`).
- **$ kubectl krew list**
  List all installed plugins.
- **$ kubectl ctx**
  Quickly switch between Kubernetes contexts.
- **$ kubectl ns**
  Quickly switch between Kubernetes namespaces.
- **$ kubectl plugin list**
  Display all available kubectl plugins.
- **$ kubectl whoami**
  (via plugin) Show current user identity.
- **$ kubectl df-pv**
  Check PVC usage (via plugin).

Monitoring & Metrics

- **$ kubectl top nodes**
  Show CPU and memory usage per node (requires metrics-server).
- **$ kubectl top pods –all-namespaces**
  Show CPU and memory usage for all pods.
- **$ kubectl describe pod ¡pod¿**
  View resource requests, limits, and usage of a pod.
- **$ kubectl get –raw /apis/metrics.k8s.io/**
  Direct API call to metrics-server.

- **Prometheus Query:** `rate(container_cpu_usage_seconds_total[5m])`
  Get CPU usage rate per container in Prometheus.
- **Prometheus Query:** `sum by (namespace)(container_memory_working_set_bytes)`

  Monitor memory usage per namespace.
- **Prometheus Query:** `up`
  Check which Prometheus targets are up.
- **Grafana Tip:**
  Import dashboards for Kubernetes clusters using Prometheus/Loki data sources.

Captain’s Log

Our expedition into Kubernetes concludes here, but your journey as a cluster commander has just begun.
Keep deploying, keep scaling, keep mastering the cloud.

Thank You for reading this Kubernetes Handbook!

Author: Sainath Shivaji Mitalakar
Edition: 2025

”A smooth sea never made a skilled sailor.”
– Franklin D. Roosevelt

Key Takeaways:

- Understand the core components: Pods, Nodes, Services, and Controllers.
- Embrace Infrastructure as Code with Helm and GitOps.
- Prioritize observability: use Prometheus, Grafana, and logging stacks.
- Secure your clusters with RBAC, Network Policies, and Secrets Management.
- Stay updated—Kubernetes evolves fast; so should you.

Call-to-Action:

Let’s connect on LinkedIn—share your Kubernetes war stories, ask questions, and grow together as cloud-native professionals!
Sainath Shivaji Mitalakar