Kubernetes Mastery Guide: Complete Technical Interview Companion

1. What is Kubernetes?

Definition:
Kubernetes (K8s) is an open-source container orchestration platform that automates deployment, scaling, and management of containerized applications.

Explanation:
It was originally developed by Google and is now maintained by CNCF. Kubernetes manages containers across multiple hosts and provides mechanisms for service discovery, load balancing, rolling updates, and self-healing of failed applications.

Architecture Overview:

- **Control Plane:** API Server, Scheduler, Controller Manager, etcd.
- **Worker Nodes:** kubelet, kube-proxy, and container runtime (like Docker or containerd).

Example Commands:

```
kubectl get nodes
kubectl get pods -A
kubectl describe node <node-name>
```

Real-world Use Case:
Used by companies like Netflix and Spotify to deploy microservices architectures that auto-scale based on user traffic.

---

2. What is a Pod in Kubernetes?

Definition:
A Pod is the smallest deployable unit in Kubernetes that represents a single instance of a running process.

Explanation:
A Pod can contain one or more tightly coupled containers that share the same network namespace, storage volumes, and IP address. If a Pod fails, Kubernetes can restart or reschedule it automatically.

Architecture Overview:

- Each Pod runs on a node.
- Managed by higher-level controllers such as Deployments or StatefulSets.

Example Commands:

```
kubectl get pods
kubectl describe pod <pod-name>
kubectl delete pod <pod-name>
```

Real-world Use Case:
Pods are used to run microservices like a REST API server or a frontend web application.

---

3. What is a Deployment in Kubernetes?

Definition:
A Deployment ensures that a specified number of Pods are running at all times with the correct configuration and image version.

Explanation:
Deployments provide declarative updates. If a Pod crashes, a new one is automatically created. Rolling updates and rollbacks allow smooth upgrades without downtime.

Architecture Overview:

- **Deployment → ReplicaSet → Pods**
- Rolling update and rollback mechanisms are built-in.

Example Commands:

```
kubectl create deployment nginx --image=nginx
kubectl scale deployment nginx --replicas=5
kubectl rollout status deployment/nginx
```

Real-world Use Case:
Used for maintaining high availability in production services during version upgrades.

---

4. What is a Service in Kubernetes?

Definition:
A Service defines a stable networking endpoint to access a set of Pods.

Explanation:
Since Pods are temporary and can change IPs, Services provide a consistent way to access them using a DNS name or IP. Services also handle load balancing among Pods.

Architecture Overview:

- **Types:** ClusterIP, NodePort, LoadBalancer, ExternalName.
- **Managed by:** kube-proxy using iptables or IPVS.

Example Commands:

```
kubectl expose deployment nginx --type=NodePort --port=80
kubectl get svc
kubectl describe svc nginx
```

Real-world Use Case:
Used to expose APIs or web applications to other microservices or external users.

---

5. What is a Namespace in Kubernetes?

Definition:
Namespaces provide logical separation within a single cluster for managing resources independently.

Explanation:
They are useful in multi-environment clusters (like dev, staging, prod). Each namespace can have its own resource quotas and access policies.

Architecture Overview:

- Default namespaces: `default`, `kube-system`, `kube-public`, `kube-node-lease`.

Example Commands:

```
kubectl get namespaces
kubectl create namespace test
kubectl config set-context --current --namespace=test
```

Real-world Use Case:
Organizations use namespaces to isolate team workloads and limit resource usage via quotas.

---

6. What is etcd in Kubernetes?

Definition:
etcd is a distributed key-value store that stores all cluster configuration data and state information.

Explanation:
Kubernetes uses etcd as its backing store for storing configurations, secrets, cluster state, and node information. It ensures data consistency and availability across nodes.

Architecture Overview:

- Runs on control plane nodes.
- Provides data consistency using the Raft consensus algorithm.

Example Commands:

```
kubectl get componentstatus
ETCDCTL_API=3 etcdctl get /registry/pods/ --prefix
```

Real-world Use Case:
Used to maintain cluster state — for example, which Pods should be running on which nodes.

---

7. What is a ReplicaSet?

Definition:
A ReplicaSet ensures that a specified number of Pod replicas are running at any given time.

Explanation:
It automatically adds or removes Pods to maintain the desired replica count. Deployments manage ReplicaSets under the hood.

Architecture Overview:

- **Controller:** Monitors Pods and maintains replica count.
- **Relation:** ReplicaSet ← Deployment ← Pods.

Example Commands:

```
kubectl get replicasets
kubectl scale rs my-app-rs --replicas=4
```

Real-world Use Case:
Used for maintaining consistent availability for stateless applications.

---

8. What are Labels and Selectors?

Definition:
Labels are key-value pairs attached to Kubernetes objects. Selectors are used to filter or identify these objects based on labels.

Explanation:
They help organize, group, and query resources efficiently. For example, a Service can select Pods with a specific label to route traffic to them.

Architecture Overview:

- **Labels:** Attached metadata.
- **Selectors:** Match labels to associate objects.

Example Commands:

```
kubectl get pods --show-labels
kubectl get pods -l app=nginx
kubectl label pod nginx-pod env=prod
```

Real-world Use Case:
Used to route traffic to specific environments like “dev” or “prod” Pods.

---

9. What is a ConfigMap?

Definition:
A ConfigMap is used to store non-confidential configuration data as key-value pairs.

Explanation:
ConfigMaps allow you to decouple configuration details from the application code. They can be mounted as environment variables or files.

Architecture Overview:

- Stored in etcd.

- Used by Pods through environment variables or mounted volumes.

Example Commands:

```
kubectl create configmap app-config
--from-literal=APP_MODE=production
kubectl get configmaps
kubectl describe configmap app-config
```

Real-world Use Case:
Used to manage environment-specific configuration without modifying container images.

---

10. What is a Secret in Kubernetes?

Definition:
A Secret is used to store sensitive information like passwords, API keys, and certificates.

Explanation:
Secrets are base64 encoded and stored in etcd. They can be mounted as volumes or exposed as environment variables to Pods securely.

Architecture Overview:

- **Types:** generic, docker-registry, TLS.
- **Security:** Can be encrypted at rest in etcd.

Example Commands:

```
kubectl create secret generic db-secret
--from-literal=username=admin --from-literal=password=pass123
kubectl get secrets
kubectl describe secret db-secret
```

Real-world Use Case:
Used for securely providing credentials to applications like connecting to a database or API gateway.

11. What is a DaemonSet in Kubernetes?

Definition:
A DaemonSet ensures that a specific Pod runs on all or selected nodes in the Kubernetes cluster.

Explanation:
DaemonSets are used to deploy system-level services such as log collectors, monitoring agents, or networking components on every node. When new nodes are added, Kubernetes automatically schedules the DaemonSet Pods on them.

Architecture Overview:

- Managed by the Kubernetes control plane.
- Ensures one Pod copy per node (or per subset of nodes using selectors).

Example Commands:

```
kubectl get daemonsets

kubectl describe daemonset kube-proxy -n kube-system

kubectl delete daemonset <name>
```

Real-world Use Case:
Used for deploying **Fluentd**, **Prometheus Node Exporter**, or **CNI plugins** like Calico or Weave Net across all nodes.

---

12. What is a StatefulSet?

Definition:
A StatefulSet manages the deployment and scaling of Pods that require stable, unique network identifiers and persistent storage.

Explanation:
Unlike Deployments, StatefulSets maintain sticky identities for each Pod (pod-0, pod-1, etc.), which makes them ideal for stateful applications like databases. Pods in StatefulSets are created and terminated in an ordered sequence.

Architecture Overview:

- **Components:** Headless Service, PersistentVolumeClaims, and StatefulSet.

- Maintains ordered Pod startup and termination.

Example Commands:

```
kubectl get statefulsets

kubectl describe statefulset mysql

kubectl delete statefulset mysql
```

Real-world Use Case:
Used to deploy stateful workloads like **MySQL**, **MongoDB**, or **Kafka** that require consistent storage and stable DNS.

---

13. What is a Job in Kubernetes?

Definition:
A Job creates one or more Pods that run to completion and then terminate.

Explanation:
Jobs are used for tasks that need to be executed once (e.g., backups, data migrations). Kubernetes ensures that the Pods complete successfully even if some nodes fail.

Architecture Overview:

- Runs Pods until the specified completions are reached.
- Can run in parallel or sequentially based on configuration.

Example Commands:

```
kubectl create job data-backup --image=busybox -- /bin/sh -c "echo
Backup Complete"

kubectl get jobs

kubectl describe job data-backup
```

Real-world Use Case:
Used for database backups, cleanup scripts, or one-time batch processes.

---

14. What is a CronJob in Kubernetes?

Definition:
A CronJob schedules Jobs to run periodically based on a cron expression.

Explanation:
It automates recurring tasks such as data backups, sending reports, or log cleanup. Kubernetes ensures Jobs are created at the specified intervals and handles retries if a Job fails.

Architecture Overview:

- Uses **cron syntax** like `"0 2 * * *"` for scheduling.
- Relies on the Job controller to execute workloads.

Example Commands:

```
kubectl create cronjob db-backup --image=busybox --schedule="0 2 * * *" -- /bin/sh -c "echo Backup done"

kubectl get cronjobs

kubectl describe cronjob db-backup
```

Real-world Use Case:
Used to schedule nightly backups, periodic data exports, and maintenance scripts.

---

15. What is Ingress in Kubernetes?

Definition:
Ingress is an API object that manages external access to Services within a cluster, typically via HTTP and HTTPS.

Explanation:
Ingress allows you to define rules for routing external traffic to internal Services based on URLs or hostnames. It can also handle SSL termination and load balancing.

Architecture Overview:

- Requires an **Ingress Controller** (e.g., NGINX, Traefik).
- **Ingress Resource:** Defines routing rules and TLS configurations.

Example Commands:

```
kubectl get ingress

kubectl describe ingress web-ingress
```

Real-world Use Case:
Used for managing routes like:

- `api.myapp.com` → Backend API
- `web.myapp.com` → Frontend Web Service

---

16. What is a Node in Kubernetes?

Definition:
A Node is a worker machine (physical or virtual) where Kubernetes runs Pods.

Explanation:
Each Node runs components like `kubelet`, `kube-proxy`, and the container runtime. The Control Plane schedules workloads onto Nodes based on resource availability.

Architecture Overview:

- **Master Node:** Runs control plane components.
- **Worker Node:** Executes Pods and application workloads.

Example Commands:

```
kubectl get nodes

kubectl describe node <node-name>

kubectl cordon <node-name>  # Mark node unschedulable

kubectl drain <node-name>   # Safely evict pods
```

Real-world Use Case:
Nodes can scale up or down dynamically in cloud environments like AWS EKS or GKE.

---

17. What is a Controller in Kubernetes?

Definition:
A Controller is a control loop that monitors the cluster's state and makes changes to move it toward the desired state.

Explanation:
Controllers like Deployment, ReplicaSet, and StatefulSet continuously compare the desired state (from manifests) with the actual state and perform corrective actions automatically.

Architecture Overview:

- Runs in the Controller Manager.
- Common Controllers: Node Controller, Deployment Controller, Endpoint Controller.

Example Commands:

```
kubectl get deployments

kubectl describe deployment nginx
```

Real-world Use Case:
Ensures that an application always has the specified number of Pods running even after failures.

---

18. What is kubelet in Kubernetes?

Definition:
The kubelet is an agent running on each node that ensures containers are running in a Pod as expected.

Explanation:
It watches for Pod specifications through the API Server and manages the container runtime to maintain those Pods. If a Pod fails, kubelet restarts it automatically.

Architecture Overview:

- Communicates with the API Server.
- Works with container runtimes like containerd or CRI-O.

Example Commands:

```
systemctl status kubelet

journalctl -u kubelet
```

Real-world Use Case:
Responsible for maintaining the health of node-level workloads and ensuring containers adhere to their Pod specifications.

---

19. What is kube-proxy in Kubernetes?

Definition:
kube-proxy is a network component that maintains network rules on each node to route traffic to the appropriate Pods.

Explanation:
It watches the Kubernetes API for Service and Endpoint changes and updates the local routing tables (using iptables or IPVS) to ensure correct packet forwarding.

Architecture Overview:

- Works in **userspace**, **iptables**, or **IPVS** mode.
- Runs on each node alongside kubelet.

Example Commands:

```
kubectl get pods -n kube-system | grep kube-proxy

kubectl logs -n kube-system <kube-proxy-pod>
```

Real-world Use Case:
Used to balance traffic between Pods for internal Services efficiently.

---

20. What is a Persistent Volume (PV) and Persistent Volume Claim (PVC)?

Definition:
A Persistent Volume (PV) is a piece of storage provisioned in a cluster, and a Persistent Volume Claim (PVC) is a request for storage by a user.

Explanation:
PVs and PVCs decouple storage from Pods. When a Pod needs storage, it references a PVC, and Kubernetes binds it to an available PV automatically.

Architecture Overview:

- **PV:** Physical storage resource.
- **PVC:** Logical request for storage.
- **StorageClass:** Defines dynamic provisioning of volumes.

Example Commands:

```
kubectl get pv

kubectl get pvc

kubectl describe pvc <pvc-name>
```

Real-world Use Case:
Used to persist data for databases, file storage, and caching systems like Redis or PostgreSQL running in Kubernetes.

---

21. What is a StorageClass in Kubernetes?

Definition:
A StorageClass defines *how* a storage volume is dynamically provisioned in Kubernetes.

Explanation:
Instead of manually creating PersistentVolumes (PVs), StorageClasses allow Kubernetes to automatically create storage based on user requests (PVCs). Different classes can represent SSD, HDD, encrypted volumes, or cloud storage like AWS EBS or GCP Persistent Disks.

Architecture Overview:

- StorageClass → Provisioner → PV creation

- Common provisioners:
  - `kubernetes.io/aws-ebs`
  - `kubernetes.io/gce-pd`
  - `kubernetes.io/no-provisioner`

Example Commands:

```
kubectl get storageclass

kubectl describe storageclass standard
```

Example YAML:

```
apiVersion: storage.k8s.io/v1

kind: StorageClass

provisioner: kubernetes.io/aws-ebs

parameters:

  type: gp2
```

Real-world Use Case:
Used to provision cloud storage automatically when PVCs are created — essential for dynamic scaling of databases.

---

22. What are Probes in Kubernetes (Liveness, Readiness, Startup)?

Definition:
Probes are health checks used by kubelet to monitor the state of containers.

Explanation:

- **Liveness Probe:** Checks if the container is alive. If it fails, Kubernetes restarts the container.
- **Readiness Probe:** Checks if container is ready to serve traffic. If not, it is removed from the Service load balancer.
- **Startup Probe:** Ensures application startup completes before other probes run — used for slow-start containers.

Architecture Overview:
Probes use:

- `httpGet`
- `tcpSocket`
- `exec` commands

Example YAML:

```yaml
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
```

Real-world Use Case:
Ensures stable rolling updates and prevents traffic from hitting services that aren't ready.

---

23. What are Taints and Tolerations?

Definition:
Taints restrict Pods from being scheduled on certain nodes, while Tolerations allow Pods to run on tainted nodes.

Explanation:
Taints are applied to nodes. If a Pod does not tolerate a taint, it won't be scheduled on that node. This enables dedicated nodes for special workloads, GPU workloads, or maintenance isolation.

Architecture Overview:

- **Taints:** Added at node level
- **Tolerations:** Added inside Pod spec

Example Commands:

```
kubectl taint nodes node1 dedicated=highcpu:NoSchedule

kubectl describe node node1
```

Example YAML:

```
tolerations:

- key: "dedicated"

  operator: "Equal"

  value: "highcpu"

  effect: "NoSchedule"
```

Real-world Use Case:
Used to isolate workloads like secure apps, GPU pods, or maintenance tasks.

---

24. What is Node Affinity?

Definition:
Node Affinity allows you to schedule Pods on specific nodes based on labels.

Explanation:
It improves scheduling decisions by allowing preferred (soft) or required (hard) placement rules. It replaces the older `nodeSelector` functionality with more flexibility.

Architecture Overview:

- `requiredDuringSchedulingIgnoredDuringExecution`
- `preferredDuringSchedulingIgnoredDuringExecution`

Example YAML:

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

Real-world Use Case:
Used to run high-performance workloads on SSD-backed nodes.

---

25. What is Pod Affinity and Anti-Affinity?

Definition:
Pod Affinity schedules Pods *near* other Pods.
Pod Anti-Affinity schedules Pods *away* from other Pods.

Explanation:
Affinity improves performance when Pods should run close together (low-latency communication).
Anti-Affinity improves reliability by spreading Pods across nodes.

Architecture Overview:

- Based on labels of other Pods
- Hard (required) or soft (preferred) rules

Example YAML:

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

Real-world Use Case:
Used to spread replicas of a service across multiple nodes to avoid single-node failure.

---

26. What is a ServiceAccount in Kubernetes?

Definition:
A ServiceAccount provides an identity for Pods to interact with the Kubernetes API.

Explanation:
Every Pod runs with a default ServiceAccount unless overridden. It provides API credentials, mounted automatically inside Pods, and is used for RBAC-controlled access.

Architecture Overview:

- Generates tokens stored inside Secrets
- Attached automatically to Pods

Example Commands:

```
kubectl get serviceaccounts
kubectl describe serviceaccount default
```

Example YAML:

```
serviceAccountName: my-sa
```

Real-world Use Case:
Used for applications that need to query Kubernetes API, like monitoring tools or operators.

---

27. What is RBAC (Role-Based Access Control)?

Definition:
RBAC is a Kubernetes security mechanism that restricts user and Pod access using roles and permissions.

Explanation:
It uses Roles, ClusterRoles, and bindings to grant fine-grained permissions. RBAC ensures only authorized users and workloads can modify or view cluster resources.

Architecture Overview:

- **Role**: namespace-level permissions
- **ClusterRole**: cluster-wide permissions
- **RoleBinding / ClusterRoleBinding**: assigns roles to users or ServiceAccounts

Example Commands:

```
kubectl get roles

kubectl create rolebinding <name> --role=<role> --user=<user>
```

Example YAML:

```
kind: Role

rules:

- apiGroups: [""]
```

```yaml
  resources: ["pods"]

  verbs: ["get", "list"]
```

Real-world Use Case:
Used to enforce least-privilege principles in production clusters.

---

28. What is a Network Policy?

Definition:
A NetworkPolicy controls how Pods communicate with each other and with external networks.

Explanation:
By default, all Pods can communicate freely. NetworkPolicies restrict ingress/egress traffic based on labels, namespaces, or IP ranges—similar to firewall rules at Pod level.

Architecture Overview:
Requires CNI plugins such as:

- Calico
- Cilium
- Weave Net

Example YAML:

```yaml
apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

spec:

  podSelector:

    matchLabels:

      app: db

  ingress:

  - from:
    - podSelector:
        matchLabels:
          app: backend
```

Real-world Use Case:
Used to isolate database Pods so only backend Pods can access them.

---

29. What is a Horizontal Pod Autoscaler (HPA)?

Definition:
HPA automatically scales Pods in a Deployment or ReplicaSet based on resource metrics.

Explanation:
It retrieves CPU/memory usage via Metrics Server and adjusts the number of Pods to handle real-time traffic changes.

Architecture Overview:

- Requires Metrics Server
- Supports CPU, memory, and custom metrics

Example Commands:

```
kubectl autoscale deployment web --cpu-percent=70 --min=2 --max=10
kubectl get hpa
```

Example YAML:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
spec:
  minReplicas: 2
  maxReplicas: 10
```

Real-world Use Case:
Used for auto-scaling stateless services like web APIs during high traffic.

---

30. What is a Vertical Pod Autoscaler (VPA)?

Definition:
VPA adjusts CPU and memory requests/limits of Pods automatically.

Explanation:
While HPA scales the *number* of Pods, VPA scales the *resources* allocated to each Pod. It analyzes usage and recommends or applies updated resource values.

Architecture Overview:
Components:

- Recommender
- Updater
- Admission Plugin

Example Command:

```
kubectl get vpa
```

Real-world Use Case:
Used for workloads with varying memory/CPU needs, such as machine learning apps or data processors.

Used for auto-scaling stateless services like web APIs during high traffic.

---

30. What is a Vertical Pod Autoscaler (VPA)?

Definition:
VPA adjusts CPU and memory requests/limits of Pods automatically.

Explanation:
While HPA scales the *number* of Pods, VPA scales the *resources* allocated to each Pod. It analyzes usage and recommends or applies updated resource values.

Architecture Overview:
Components:

- Recommender
- Updater
- Admission Plugin

Example Command:

```
kubectl get vpa
```

Real-world Use Case:
Used for workloads with varying memory/CPU needs, such as machine learning apps or data processors.

---

31. What is Cluster Autoscaler?

Definition:
Cluster Autoscaler automatically adjusts the number of worker nodes in a Kubernetes cluster based on workload demands.

Explanation:
If Pods fail to schedule due to insufficient resources, Cluster Autoscaler adds new nodes. If nodes remain underutilized for too long, it removes them. It is widely used in cloud-managed Kubernetes like EKS, AKS, and GKE.

Architecture Overview:

- Works with cloud provider APIs.
- Automatically scales node pools based on pending pods.

Example Commands:
Cluster Autoscaler is usually deployed via Helm:

```
helm repo add autoscaler https://kubernetes.github.io/autoscaler

helm install cluster-autoscaler autoscaler/cluster-autoscaler
```

Real-world Use Case:
Used to reduce cloud costs by scaling nodes up during peak load and scaling down during idle periods.

---

32. What is a ResourceQuota?

Definition:
A ResourceQuota is a policy that limits the total compute resources (CPU, memory, storage) that a Namespace can use.

Explanation:
It prevents one team or application from consuming all cluster resources. It works alongside LimitRange to control Pod-level resource usage.

Architecture Overview:
Applied at Namespace level via the API server.

Example YAML:

```
apiVersion: v1

kind: ResourceQuota

spec:

  hard:

    cpu: "10"

    memory: "20Gi"

    pods: "15"
```

Example Commands:

```
kubectl get resourcequota

kubectl describe resourcequota <name>
```

Real-world Use Case:
Used in multi-team clusters to control cost and prevent runaway resource usage.

---

33. What is a LimitRange?

Definition:
A LimitRange sets default, maximum, and minimum CPU/memory requests and limits per Pod or container in a Namespace.

Explanation:
It helps enforce resource fairness and prevent users from requesting too many or too few resources.

Architecture Overview:

- Enforced by the Admission Controller.
- Works with ResourceQuota.

Example YAML:

```
apiVersion: v1

kind: LimitRange

spec:

  limits:

  - type: Container

    default:

      cpu: 500m

      memory: 512Mi

    max:

      cpu: 1

      memory: 1Gi
```

Example Commands:

```
kubectl get limitrange

kubectl describe limitrange <name>
```

Real-world Use Case:
Used in production namespaces to prevent Pods from accidentally consuming excessive resources.

---

34. What is Helm in Kubernetes?

Definition:
Helm is the package manager for Kubernetes, used to install, manage, and upgrade applications using “charts”.

Explanation:
Helm charts package all Kubernetes manifests needed for an application. Helm supports versioning, rollbacks, templating, and shared configuration values.

Architecture Overview:

- **Charts:** Application templates
- **Repositories:** Store charts
- **Releases:** Application instances installed into clusters

Example Commands:

```
helm repo add bitnami https://charts.bitnami.com/bitnami

helm install myapp bitnami/nginx

helm upgrade myapp bitnami/nginx

helm uninstall myapp
```

Real-world Use Case:
Used to deploy complex apps like Prometheus, Grafana, Jenkins, Elasticsearch with a single command.

---

35. What is kubeadm?

Definition:
kubeadm is a Kubernetes tool that bootstraps and configures production-grade Kubernetes clusters.

Explanation:
It simplifies installing Control Plane components, setting up worker nodes, generating certificates, and managing cluster bootstrap configuration.

Architecture Overview:

- `kubeadm init` → sets up master
- `kubeadm join` → adds worker nodes
- Creates certificates and default configuration files

Example Commands:

```
kubeadm init

kubeadm join <master-ip>:6443 --token <token>
```

Real-world Use Case:
Used to create highly available on-premise or cloud clusters manually.

---

36. What is Minikube?

Definition:
Minikube is a lightweight tool for running a single-node Kubernetes cluster locally on your laptop.

Explanation:
Perfect for development, testing, and learning Kubernetes. It supports addons like Ingress, Dashboard, DNS, and storage drivers.

Architecture Overview:
Runs Kubernetes inside:

- Docker
- VirtualBox
- Hyper-V

- KVM

Example Commands:

```
minikube start

minikube dashboard

minikube addons enable ingress
```

Real-world Use Case:
Used by developers to test applications locally before deploying to real clusters.

---

37. What is a Sidecar Container?

Definition:
A Sidecar container is a secondary container in a Pod that enhances or supports the main application container.

Explanation:
Sidecars run alongside the main container and add features like logging, proxying, monitoring, or configuration syncing.

Common Sidecar Patterns:

- Logging sidecar
- Envoy proxy sidecar (service mesh)
- File sync sidecar

Example YAML:

```
containers:

- name: app

  image: app:latest

- name: log-agent

  image: fluentd
```

Real-world Use Case:
Used in service meshes (Istio) where every Pod has an Envoy proxy sidecar.

---

38. What is a Multi-container Pod?

Definition:
A Multi-container Pod is a Pod containing more than one container sharing the same network and storage.

Explanation:
Containers communicate via `localhost` and share volumes. Kubernetes ensures they start and stop together.

Common Multi-container Patterns:

- Sidecar
- Ambassador
- Adapter

Example Commands:

```
kubectl get pods

kubectl describe pod <multi-pod>
```

Example YAML:

```
containers:

- name: app

  image: myapp

- name: helper

  image: busybox
```

Real-world Use Case:
Used when auxiliary tasks (monitoring, proxying) need to run beside the main app.

---

39. What is an Admission Controller?

Definition:
Admission Controllers intercept API requests before they are stored in etcd, enforcing rules and security policies.

Explanation:
They can mutate or validate incoming resource requests. Examples include PodSecurityPolicy, NamespaceLifecycle, and ResourceQuota controllers.

Architecture Overview:
Two types:

- **MutatingAdmissionWebhook**
- **ValidatingAdmissionWebhook**

Example Commands:

```
kubectl api-resources | grep admission
```

Real-world Use Case:
Used to enforce organizational policies like “all Pods must have resource limits”.

---

40. What is kube-scheduler?

Definition:
kube-scheduler is the component that assigns Pods to nodes based on constraints and available resources.

Explanation:
It evaluates available nodes and selects the most suitable one using scheduling algorithms that consider CPU, memory, taints, affinity, and more.

Architecture Overview:
Scheduling steps:

1. Filtering (nodes that can run the Pod)
2. Scoring (best-fit node)
3. Binding (assigning Pod)

Example Commands:

```
kubectl get pods -n kube-system | grep scheduler
```

Real-world Use Case:
Ensures optimal workload placement for high performance and resource efficiency.

---

41. What is a NodePort Service in Kubernetes?

Definition:
NodePort is a Service type that exposes an application externally by opening a specific port on every worker node.

Explanation:
NodePort allows users to access a Service from outside the cluster using `NodeIP:NodePort`. Kubernetes routes this request to the correct Pod through kube-proxy. NodePorts are allocated in the range **30000–32767** by default.

Architecture Overview:

- Client → NodeIP → NodePort → kube-proxy → Pod
- Works on all nodes, even if Pods run on only some nodes

Example Commands:

```
kubectl expose deployment nginx --type=NodePort --port=80

kubectl get svc
```

Real-world Use Case:
Used in on-premise or bare-metal clusters where LoadBalancer services are not available.

---

42. What is a ClusterIP Service in Kubernetes?

Definition:
ClusterIP is the default Service type that exposes a service internally within the cluster.

Explanation:
ClusterIP assigns an internal virtual IP that allows communication between Pods and services inside the cluster. It cannot be accessed externally.

Architecture Overview:

- Internal load balancing using kube-proxy (iptables or IPVS)
- Service discovery via DNS: `<service-name>.<namespace>.svc.cluster.local`

Example Commands:

```
kubectl get svc

kubectl describe svc backend
```

Real-world Use Case:
Used for internal microservice communication (e.g., frontend → backend API).

---

43. What is a LoadBalancer Service in Kubernetes?

Definition:
LoadBalancer is a service type that exposes an application externally using a cloud provider's load balancer.

Explanation:
This is the most common service type in cloud environments (EKS, AKS, GKE). It automatically provisions a load balancer that forwards traffic to NodePorts, which then route to Pods.

Architecture Overview:
Client → Cloud LoadBalancer → NodePort → kube-proxy → Pod

Example Commands:

```
kubectl expose deployment api --type=LoadBalancer --port=80
kubectl get svc
```

Real-world Use Case:
Used to expose production APIs over the internet with automatic health checks and load balancing.

---

44. What is `kubectl get` used for?

Definition:
`kubectl get` retrieves information about Kubernetes resources.

Explanation:
It supports listing resources across namespaces, displaying wide output, showing labels, and even JSON/YAML output formatting.

Useful Flags:

- `-A` → all namespaces
- `-o wide` → detailed info
- `--show-labels` → show object labels

Example Commands:

```
kubectl get pods

kubectl get nodes -o wide

kubectl get svc -A
```

Real-world Use Case:
Used daily by DevOps engineers for cluster inspection.

---

45. What is `kubectl describe` used for?

Definition:
`kubectl describe` shows detailed information about a specific Kubernetes resource.

Explanation:
It displays events, status, annotations, and configuration useful for debugging. It is essential when troubleshooting issues like Pod crashes or scheduling failures.

Example Commands:

```
kubectl describe pod frontend

kubectl describe deployment backend

kubectl describe node node-1
```

Real-world Use Case:
Used to investigate Pod failures, image pull errors, scheduling issues, or resource limits.

---

46. What is `kubectl logs` used for?

Definition:
`kubectl logs` retrieves logs from containers running inside Pods.

Explanation:
It helps developers and DevOps engineers troubleshoot application errors, crashes, and behavior. Logs can be streamed in real time using the `-f` flag.

Example Commands:

```
kubectl logs mypod

kubectl logs -f mypod

kubectl logs mypod -c sidecar-container
```

Real-world Use Case:
Used to monitor microservices or troubleshoot failures during deployments.

---

47. What is `kubectl exec` used for?

Definition:
`kubectl exec` executes a command inside a running container.

Explanation:
This is used for debugging applications, checking file systems, or running shell commands inside Pods.

Example Commands:

```
kubectl exec -it db-pod -- /bin/bash

kubectl exec mypod -- ls /app
```

Real-world Use Case:
Used by engineers to inspect application logs, fix temporary issues, or test commands inside running containers.

---

48. What is a ServiceAccount in Kubernetes?

Definition:
A ServiceAccount provides an identity for Pods to interact with the Kubernetes API securely.

Explanation:
ServiceAccounts generate tokens stored in Secrets and mounted inside Pods. RoleBindings or ClusterRoleBindings assign RBAC permissions to these accounts.

Architecture Overview:

- Token injected automatically into Pods
- Used by controllers, operators, and backend apps

Example Commands:

```
kubectl get serviceaccounts

kubectl describe sa default
```

Example YAML:

```
serviceAccountName: backend-sa
```

Real-world Use Case:
Used by CI/CD pipelines, monitoring tools, and operators to authenticate with Kubernetes.

---

49. What is a NetworkPolicy in Kubernetes?

Definition:
A NetworkPolicy restricts which Pods can communicate with each other or with external networks.

Explanation:
By default, all Pods can talk to all Pods. NetworkPolicies act like firewalls — allowing or blocking traffic based on labels, namespaces, or IP blocks.

Architecture Overview:
Implemented by CNI plugins like Calico, Cilium, Weave Net, Kube-router.

Example YAML:

```yaml
apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

spec:

  podSelector:

    matchLabels:

      role: db

  ingress:

  - from:

    - podSelector:

        matchLabels:

          role: backend
```

Real-world Use Case:
Used to isolate sensitive Pods like databases from public-facing applications.

---

50. What is the difference between a Deployment and a StatefulSet?

Definition:
Both are Kubernetes workload controllers but used for different types of applications.

Explanation:

- **Deployment:**
  - Used for stateless applications (web servers, APIs).
  - Pods are identical and interchangeable.
  - Supports rolling updates and scaling.
- **StatefulSet:**
  - Used for stateful applications (databases, message queues).
  - Each Pod has a stable hostname & persistent storage.
  - Ordered deployment and termination.

Comparison Table:

Feature	Deployment	StatefulSet
Pod Identity	Random	Ordered (pod-0, pod-1)
Storage	Ephemeral	Persistent (PVC per Pod)
Use Case	Stateless apps	Databases, Zookeeper
Scaling Behavior	Flexible	Ordered

Example Commands:

```
kubectl get deployments

kubectl get statefulsets
```

Real-world Use Case:
Deployments → web servers, API microservices
StatefulSets → MongoDB, Cassandra, Kafka clusters