kubernetes:
1. what is kubernetes?
Kubernetes is a container orchestration tool that helps manage containerized applications (like those built using Docker) in a cluster of machines. It abstracts the underlying infrastructure and provides a powerful API to manage applications with high availability, scalability, and self healing capability.

Why Kubernetes?
Modern applications are often built using microservices architecture and packaged as containers. Managing containers manually becomes difficult as the number of containers grows. Kubernetes helps by:
.Automating deployment and rollback
.Self-healing applications by restarting failed containers
.Load balancing and service discovery
.Scaling applications up or down
.Managing secrets and configurations

Core Components of Kubernetes:
.Node - A physical or virtual machine where Kubernetes runs workloads
.Pod - The smallest deployable unit. It can contain one or more containers
.Deployment - Ensures the desired number of pod replicas are running
.Service - An abstraction to expose pods to internal/external traffic
.Cluster - A set of nodes (machines) managed by Kubernetes
.Control Plane - Coordinates all cluster activities (API server, Scheduler, Controller Manager, etc.)
.Kubelet - Agent running on each node to manage containers
.Kube Proxy - Maintains network rules and enables communication to pods

Key Features:
.Self-healing - Restarts containers that fail
.Horizontal Scaling - Scale applications automatically based on load
.Load Balancing - Distributes traffic to healthy pods
.Rolling Updates- Updates applications with no downtime
.Service Discovery - Uses DNS or environment variables
.Storage Orchestration - Mounts persistent storage like EBS, NFS, etc.
.Secret & Config Management - Manages sensitive data separately from app code

How Kubernetes Works
.Developer defines a deployment YAML file that specifies:
    Container image
    Number of replicas
    Resource limits, etc.
.kubectl applies the YAML to the Kubernetes API.
.Kubernetes Scheduler places pods on suitable nodes.
.Kubelet on the node ensures containers run as specified.
.Controller Manager ensures desired state (e.g., if a pod dies, it creates a new one).
.Service exposes the application to users.

2. What is a Container?
A container is a lightweight, portable, and self-contained environment that packages everything an application needs to run — code, libraries, dependencies, configuration files — so that it can run the same way anywhere.

Why Containers Are Useful:
.Portable - Run your app on any machine (laptop, cloud, server) without changing anything.
.Lightweight - Uses fewer resources than traditional virtual machines.
.Consistent - Works the same across development, testing, and production.
.Isolated - Keeps apps separated from each other. No conflicts.
.Fast to start - Starts in seconds. Great for scaling apps quickly.

What’s Inside a Container?

App code (e.g., a Python or Java app)
Runtime (e.g., Python, Node.js)
System libraries
Config files

All of this is packaged into a container image, which can be run using a container engine like Docker or containerd.

3. What is a Pod in Kubernetes?
A Pod is the smallest and simplest unit in Kubernetes.
It’s like a wrapper around one or more containers that are tightly connected and share resources.

What Do Containers in a Pod Share?
All containers in a Pod share:
The same IP address
The same network (they can talk via localhost)
The same storage volumes
They run on the same node and are always scheduled together.

Why Use Multiple Containers in One Pod?
Sometimes, containers work as a team. For example:
One container runs a web app.
Another container logs and sends metrics.
They share data and communicate locally, so putting them in the same Pod makes sense.

Example pod.yml:
```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: app
      image: my-app:latest
    - name: sidecar
      image: my-logger:latest
```

4.What is a Node?
A Node is a physical or virtual machine (like a server or cloud instance) where Kubernetes runs your application — specifically, where Pods are scheduled and executed.

There Are Two Types of Nodes:
Master Node (Control Plane) - Makes decisions — scheduling, scaling, health checks, etc.
Worker Node - Actually runs your application Pods

Each Worker Node includes:
✅ Kubelet – Talks to the Kubernetes control plane and manages Pods on the node.
🔁 Kube-proxy – Manages networking/routing.
📦 Container runtime – Like Docker or containerd to run containers.

5.What is the Control Plane?
The Control Plane is the brain of the Kubernetes cluster.
It manages the entire cluster, decides what runs where, and keeps everything in the desired state.

The Control Plane is responsible for:
Scheduling - Decides which Node runs each Pod
Monitoring - Watches the health of Nodes and Pods
Maintaining State - Ensures the cluster matches the desired configuration

Handling Failures - Replaces failed Pods, reassigns tasks
APIs & Access - Provides the interface to communicate with the cluster (via kubectl or API)

Control Plane Components:
🧭 kube-apiserver -> Entry point for all commands (CLI, UI, API)
🧠 kube-scheduler -> Assigns Pods to Nodes based on resources
📚 kube-controller-manager -> Ensures the system stays in desired state (e.g., restarts failed Pods)
🕵 cloud-controller-manager -> Integrates with cloud provider (e.g., load balancers, volumes)
💽 etcd -> A key-value database that stores all cluster data (configuration, state)

6.What is kubectl?
kubectl (pronounced "cube control" or "kube cuddle") is the command-line tool used to interact with a Kubernetes cluster.
It allows you to talk to the Kubernetes Control Plane and manage resources like Pods, Deployments, Services, Nodes, etc.

What Can You Do with kubectl?
📋 View resources --> kubectl get pods – List all Pods
🚀 Deploy apps --> kubectl apply -f app.yaml – Deploy from a YAML file
🔄 Update apps --> kubectl set image – Change container image
❌ Delete apps --> kubectl delete pod mypod – Delete a specific Pod
📦 Inspect --> kubectl describe pod mypod – View full details of a Pod
🛠 Debug --> kubectl logs mypod – View logs
kubectl exec -it mypod -- bash --> Access inside container

7. How Does kubectl Work?
When you type a kubectl command:
.It sends a request to the kube-apiserver (part of the Control Plane).
.The API server processes it and makes changes in the cluster.
.kubectl shows the result.

You must have kubectl configured with the correct cluster credentials (Kubeconfig file- ~/.kube/config) to use it.

8. What is Namespace?
A namespace in Kubernetes is a way to logically divide cluster resources. It allows you to create isolated environments within the same Kubernetes cluster, so different teams, applications, or environments like dev, test, and prod can run independently without interfering with each other.

Each namespace has its own set of:
Pods
Services
ConfigMaps
Resource quotas and more.

This helps in:

Multi-tenant architecture (different teams sharing the same cluster)
Access control using RBAC
Resource limits specific to that namespace

9. What is a Manifest file?
A manifest file in Kubernetes is a YAML or JSON configuration file that defines the desired state of a Kubernetes object—such as a Pod, Deployment, Service, ConfigMap, etc.

It tells Kubernetes what you want to run, how you want it to run, and with what configuration.

These files describe things like:
Number of replicas
Container image to use
Resource limits
Labels and selectors
Ports to expose
Volumes to mount

10.What is kubeadm?
Kubeadm is a Kubernetes tool that helps you set up a Kubernetes cluster quickly and easily. It takes care of the core steps involved in initializing and joining nodes to a cluster.

It does not install Kubernetes, but it configures all the required components like:
kube-apiserver
kube-scheduler
kube-controller-manager
etcd

What does kubeadm do?
```
kubeadm init
```
Initializes the control plane (master node).

```
kubeadm join
```
Adds worker nodes to the cluster using a token.

Bootstraps TLS certificates, generates keys, and sets up cluster networking.

11.Difference between Docker and Kubernetes?
Docker and Kubernetes are related but serve different purposes in the container ecosystem.
🐳 Docker:
Docker is a containerization platform.
It helps you build, package, and run applications in lightweight, portable containers.
It focuses on the single-node lifecycle of containers: building, running, and managing them.
Think of Docker as a tool to create containers.

☸️ Kubernetes:
Kubernetes is a container orchestration platform.
It manages multiple containers across a cluster of machines.
It handles deployment, scaling, load balancing, self-healing, and rolling updates of containers.
Think of Kubernetes as the system that manages many containers (including Docker containers).

12.How to check the status of a Pod?
To check the status of a Pod, you use the kubectl get pods command. It gives you an overview of all pods and their current states.

To check in a specific namespace:
```
kubectl get pods -n <namespace-name>
```

For more detailed info:
```
kubectl describe pod <pod-name>
```

TO check logs:
```
kubectl logs <pod-name>
```

13. What are Deployments in Kubernetes?
A Deployment in Kubernetes is a controller that manages the declarative updates of your application pods. It ensures that the desired number of pod replicas are

running, and it handles rollouts, rollbacks, and updates automatically.

🚀 Key Features of a Deployment:
Creates and maintains ReplicaSets
Ensures high availability by replacing failed pods
Supports rolling updates (zero-downtime deployments)
Supports rollbacks in case of failures
Can scale pods up or down easily

You can pause, resume, or rollback a deployment using:
kubectl rollout pause deployment/my-app --> Used to pause the deployment
kubectl rollout resume deployment/my-app --> Used to resume the paused deployment
kubectl rollout undo deployment/my-app --> Used to rollback to the older stable version.

14. What is a ReplicaSet?

A ReplicaSet in Kubernetes ensures that a specified number of identical Pods are always running in the cluster.
If a pod crashes or is deleted, the ReplicaSet automatically creates a new one to maintain the desired number of replicas.

Maintain the desired state of pods by monitoring and ensuring the exact number of replicas are running at all times.

⚙️ How it works:
A ReplicaSet uses labels and selectors to identify and manage the pods it controls.
If the actual number of pods doesn't match the desired count, it adds or deletes pods accordingly.

🔁 Relation to Deployment:
You don’t usually create ReplicaSets directly.
Deployments internally use ReplicaSets to manage pods.
Deployments give you extra features like rollouts and rollbacks, which ReplicaSets alone don’t support.

Difference between Replicationcontroller and ReplicaSet?
A ReplicationController is an older Kubernetes object that ensures a specified number of identical pods are running at any time—just like a ReplicaSet.

A ReplicaSet is the next-generation version of ReplicationController. It provides the same core functionality (ensuring a stable number of pod replicas) but with enhanced features, especially around label selectors.

ReplicationController and ReplicaSet both ensure the desired number of pod replicas are running.
However, ReplicaSet is the modern replacement, supports more flexible label selectors, and is used by Deployments internally.
ReplicationController is now considered outdated and is rarely used in practice.”

15.What is service and types of it?

A Service in Kubernetes is an abstraction that defines a logical set of Pods and a policy by which to access them.
Since Pods are ephemeral (they can be created or destroyed dynamically), a Service provides a stable endpoint (IP and DNS name) to reliably communicate with those Pods.

In simple terms, a Service enables network access to a group of Pods, load balances traffic among them

Types of Services in Kubernetes?
ClusterIP:
Default type. Exposes the service internally inside the cluster.
Communication between pods inside the cluster.

NodePort:
Exposes the service on a static port on each node’s IP.
Exposes service to external traffic via <NodeIP>:<NodePort>.

LoadBalancer:
Creates an external load balancer (if supported by cloud provider) to expose service publicly.
For cloud environments to expose service externally with a cloud LB.

ExternalName:
Maps the service to an external DNS name (no proxying)
For accessing external services via a Kubernetes service name.

16.** What is ingress and diff between ingress and service?
Ingress is a Kubernetes resource that manages external HTTP and HTTPS access to services inside a cluster. It acts as a smart router or gateway that directs incoming web traffic (based on hostnames, paths, etc.) to the correct backend services.

🔷 Key points about Ingress:
It provides URL-based routing and load balancing.
Supports TLS/SSL termination (HTTPS).
Can do virtual hosting (route traffic to different services based on the hostname or URL path).
Requires an Ingress Controller to implement the actual traffic routing (like NGINX Ingress Controller, Traefik, etc.)

Service Exposes pods internally or externally with fixed IP/port
Ingress Manages external HTTP/HTTPS traffic routing to multiple services

Service routes Simple: forwards traffic to pods
Ingress routes Advanced: routes traffic based on hostname/path rules

Service: You create a Service to expose your backend pods internally or to external clients on a specific port.

Ingress: You use Ingress when you want to expose multiple services under the same IP/domain, routing traffic like:
Example:
https://myapp.com/api -> routes to api-service
https://myapp.com/web -> routes to web-service

17.What is kube-proxy?
kube-proxy is a networking component that runs on every Kubernetes node. Its primary role is to maintain network rules on nodes so that communication to and from Pods and Services works seamlessly.

🧩 Why kube-proxy is important?
It provides service discovery and load balancing inside the cluster at the network level.
Ensures that Services have a stable IP and DNS name even though Pods may change dynamically.
Runs on each node to enable efficient local routing and reduce cross-node traffic where possible.

18. How Does Service Discovery Work in Kubernetes?
Service Discovery in Kubernetes is the process that allows applications (pods) to find and communicate with other services without needing to know the exact IP addresses of the pods, which can change frequently

🔷 How it works practically:
Pods don’t connect directly to other pods; they connect to a Service.
The Service has a stable IP (ClusterIP), which kube-proxy manages and routes traffic to the correct pod endpoints.
This way, Pods remain loosely coupled and can scale up/down without breaking connections.

Your frontend pod needs to call your backend API. Instead of hardcoding pod IPs, it calls http://backend-service:80, and DNS + kube-proxy handle routing the request to any healthy backend pod.

19.What are ConfigMap and Secret in Kubernetes?
1. ConfigMap
A ConfigMap is used to store non-sensitive configuration data in key-value pairs.
It helps you externalize configuration from your container image, so you can change app behavior without rebuilding the image.

🔷 Use Cases:
Environment variables
Command-line arguments
App config files

Ex:
```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_MODE: "production"
  LOG_LEVEL: "info"
```

2. Secret
A Secret is similar to ConfigMap but is used to store sensitive data like:
Passwords
API tokens
TLS certificates
Secrets are base64-encoded and stored securely in etcd (which should be encrypted at rest).

Ex:
```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: dXNlcm5hbWU=     # base64 encoded "username"
  password: cGFzc3dvcmQ=     # base64 encoded "password"
```

How to use in pod
```
envFrom:
  - configMapRef:
      name: app-config
  - secretRef:
      name: db-secret
```

20. What is a DaemonSet?
A DaemonSet is a Kubernetes controller that ensures a specific pod runs on every node (or selected nodes) in the cluster.
Think of it as: “one pod per node” — automatically deployed and managed by Kubernetes.

🔷 Use Cases for DaemonSet:
Log collection agents like Fluentd or Filebeat
Monitoring agents like Prometheus Node Exporter
Security or intrusion detection tools
Storage drivers or node-specific utilities

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

if you want every node to collect logs, you can deploy Fluentd as a DaemonSet so that it runs automatically on all worker nodes without manual scheduling.

21.What is a StatefulSet?
A StatefulSet is a Kubernetes controller used to manage stateful applications—apps that require:
Stable network identity
Stable, persistent storage
Ordered deployment, scaling, and termination

Unlike Deployments or ReplicaSets (which are used for stateless apps), StatefulSet ensures that each pod is uniquely identified and retains its state across restarts

🔷 Key Features of StatefulSet:

Stable Hostnames	Each pod gets a fixed DNS name like pod-0, pod-1, etc.
Stable Storage	Uses PersistentVolumeClaims (PVCs) that stick to the pod's identity
Ordered Operations	Pods are created, scaled, and deleted in order (pod-0 → pod-1...)
Unique Identity	Each pod has a predictable name and storage, even after rescheduling

EX:
```
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-pv
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-pv
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```

If you're deploying a database like MySQL, Cassandra, or Kafka, you need each instance (pod) to retain its identity and storage—even if it crashes or reschedules.
StatefulSet ensures this, whereas a Deployment would treat all pods as replaceable.

🔁 Difference: StatefulSet vs Deployment

Pod Identity	All pods are identical	Each pod has a unique, stable name
Storage	Shared or ephemeral	Persistent per pod
Start/Stop	Unordered	Ordered (0 → N)
Use Case	Web servers, APIs	Databases, Kafka, etc.

22.What is a Job in Kubernetes?
A Job in Kubernetes is a controller that is used to run a one-time task or batch process and ensures that it completes successfully.
Unlike Deployments or StatefulSets (which run continuously), a Job runs until the task is complete—whether that's processing data, performing a backup, or running a script—and then it exits.

🔷 Key Features:
Runs Pods to completion
Restarts failed Pods if needed
Tracks completion: ensures the specified number of successful completions
Used for batch processing, cron jobs, data migration, etc.

Ex:
```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-job
spec:
  template:
    spec:
      containers:
      - name: hello
        image: busybox
        command: ["echo", "Hello from Kubernetes Job"]
      restartPolicy: Never
```

🔁 How It Works:
You define what task to run in a pod.
Kubernetes schedules the pod.
If the pod fails, it retries it until it succeeds (based on backoffLimit).
Once completed, the job is marked "Succeeded".

Suppose you want to run a database migration script or a file cleanup process. A Job ensures the task runs once, even if the pod fails and needs to retry.

Job → For one-time execution
Parallel Job → Run multiple pods in parallel (e.g. 10 files to process)

CronJob → Run jobs on a schedule (like a Linux cron task)

23.What is a CronJob?
A CronJob in Kubernetes is used to run Jobs on a scheduled basis, similar to a Linux cron job.
It allows you to automate recurring tasks like backups, report generation, data cleanup, or periodic API calls.

🔷 Key Features of CronJob:

Feature	Description
Schedules Jobs	Uses cron format like `"0 2 * * *"`
Creates a Job	At each scheduled time, it creates a **Job** resource
Automatic Cleanup	Can manage history of successful and failed jobs
One-time or recurring	Supports repeating tasks at fixed intervals

Database backups every night at 2 AM
Sending weekly emails or reports
Cleaning up old files from storage

Ex:
```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: db-backup
spec:
  schedule: "0 2 * * *"  # Every day at 2 AM
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: alpine
            command: ["sh", "-c", "echo Performing backup && sleep 30"]
          restartPolicy: OnFailure
```

24.How to expose a Pod to the Internet?
To expose a Pod to the internet, we don’t expose the Pod directly. Instead, we follow a standard Kubernetes pattern:
✅ Expose the Pod using a Service, and then
✅ Make the Service externally accessible using one of the following methods:
1. Using a Service of type NodePort
Exposes the Pod on a static port across all nodes.
You can access it via <NodeIP>:<NodePort>

```
kubectl expose pod my-pod --type=NodePort --port=80
kubectl get service
```

2. Using a Service of type LoadBalancer (Cloud only)
In cloud environments (e.g., AWS, GCP, Azure), this automatically provisions an external load balancer.
Best for production setups.
Ex:
```
apiVersion: v1
kind: Service
metadata:
  name: my-app
```

```
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

3. Using Ingress (for HTTP/HTTPS Traffic)
Best for web apps and domain-based routing
You expose your service using an Ingress resource and an Ingress Controller
Offers TLS, routing, path-based access, etc.

Let’s say your pod runs a web app. You can:
Create a Deployment for the pod
Expose it via a Service (type LoadBalancer)
Optionally configure an Ingress to access it via a domain like app.company.com

25.How to update a Deployment without downtime?
Kubernetes supports rolling updates out of the box using the Deployment controller.
You can update a Deployment (e.g., change the container image, environment variables, or configuration), and Kubernetes will replace old Pods with new ones gradually, ensuring zero downtime.

🔧 Steps to Perform a Zero-Downtime Update:
Modify the Deployment (e.g., change the image tag in YAML)
Apply the updated Deployment:

```
kubectl apply -f deployment.yaml
```

OR update directly:

```
kubectl set image deployment/my-app my-container=nginx:1.25
```

Kubernetes performs a Rolling Update:
Gradually creates new pods
Waits for them to become ready
Then deletes the old pods
Ensures some pods are always available

📌 Key Configuration Parameters in Deployment:
In your Deployment YAML, these fields control rollout behavior:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1           # How many extra pods to run during update
    maxUnavailable: 0     # Ensure no downtime (keep all pods available)
```

Suppose your app is running with image v1. You push v2 to Docker Hub.
You update the Deployment with kubectl set image.
Kubernetes gradually replaces the pods with new ones running v2, keeping the app available the whole time.

26. How to rollback a Deployment?
Check rollout Status:

```
kubectl rollout status deployment/my-app
```

You can roll back a Deployment in Kubernetes using:

```
kubectl rollout undo deployment/my-app
```

This command reverts the Deployment to the previous revision, ensuring your application goes back to the last known working state.

🔁 Example:
Let’s say you updated your Deployment with a wrong image:

```
kubectl set image deployment/my-app my-container=myapp:v2
```

But something went wrong (e.g., app crashing), you can rollback:
```
kubectl rollout undo deployment/my-app
```

Roll back to a specific revision:
First, check history:
```
kubectl rollout history deployment/my-app
```

Then rollback to a specific revision:
```
kubectl rollout undo deployment/my-app --to-revision=2
```

You pushed a new version of your app, but users report errors.
You immediately run kubectl rollout undo to bring your app back to its previous healthy version—without downtime.

27.How to define resource requests and limits for a Pod?
In Kubernetes, resource requests and limits are defined in the Pod spec to control how much CPU and memory (RAM) a container can use.
This helps the scheduler assign the pod to a suitable node, and prevents a single pod from consuming all resources.

🧩 Key Terms:

Term	Meaning
Request	The **minimum** amount of CPU/memory the container needs to run. Kubernetes uses this to **schedule the pod**.
Limit	The **maximum** amount of CPU/memory the container can use. It **cannot exceed** this.

Example YAML:
```yaml
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"
```

Units:
Memory: Mi (mebibytes), Gi (gibibytes)
CPU: 1 = 1 vCPU/core, 500m = 0.5 core

Defining requests and limits prevents overcommitting and protects other pods from being affected by noisy neighbors.
It also helps autoscalers make better decisions based on resource usage.

28.What is a Volume in Kubernetes?
A Volume in Kubernetes is a storage abstraction that allows data to be stored and shared by containers in a pod — and most importantly, it persists beyond container restarts.

🔷 Why Volumes Are Needed:

By default, data inside a container is lost if the container restarts.
Volumes solve this by providing a stable storage layer attached to the pod.
They can also be shared across containers within the same pod.

📦 Types of Volumes:

Volume Type	Description
`emptyDir`	Temporary storage — wiped when pod dies
`hostPath`	Uses a directory from the node’s filesystem
`configMap` / `secret`	Used to inject config data or secrets
`persistentVolumeClaim (PVC)`	Dynamic external storage from a PersistentVolume
`nfs`, `awsElasticBlockStore`, `azureDisk`, `gcePersistentDisk`	Cloud/provider-backed storage

EX:
```
apiVersion: v1
kind: Pod
metadata:
  name: volume-demo
spec:
  containers:
  - name: app
    image: busybox
    command: ["sh", "-c", "echo Hello > /data/file.txt && sleep 3600"]
    volumeMounts:
    - mountPath: /data
      name: temp-storage
  volumes:
  - name: temp-storage
    emptyDir: {}
```

This gives the container a writable /data directory that lasts as long as the pod is running.

If you're running a database, you’ll mount a volume using a PersistentVolumeClaim so the data is stored even if the pod is rescheduled to a different node.

Summary:
Volumes enable data persistence beyond container lifecycles.
Used for temporary storage, configuration injection, or permanent storage (with PVC).
Essential for stateful applications and sharing data between containers.

27.What is a PersistentVolume (PV) and PersistentVolumeClaim (PVC) and Storageclass?
In Kubernetes, PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs) are used to provision and manage durable storage for pods.
They allow you to decouple storage from the pod lifecycle, so data is not lost when a pod is deleted or restarted.

📦 PersistentVolume (PV)
A pre-provisioned piece of storage in the cluster.
Managed by the cluster admin.
Represents a physical or cloud-based disk (e.g., EBS on AWS, GCE disk, NFS).

Ex:
```
apiVersion: v1
```

```
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data
```

📩 PersistentVolumeClaim (PVC)
A request for storage by a user/pod.
You don’t request a specific PV — you ask for a type and size of storage.
Kubernetes matches the PVC with a suitable PV.

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

How They Work Together:
Admin creates a PersistentVolume (PV).
Developer creates a PersistentVolumeClaim (PVC).
Kubernetes binds the PVC to a suitable PV.
You mount the PVC into a pod like this:
```
volumeMounts:
  - mountPath: "/data"
    name: my-storage

volumes:
  - name: my-storage
    persistentVolumeClaim:
      claimName: my-pvc
```

In real-world setups, we often use StorageClasses to provision PVs dynamically when a PVC is created — no need to pre-create PVs manually.

Think of a PV as a storage room in a building, and a PVC as someone saying:
“I need a storage room of 2GiB that I can write to.”
Kubernetes finds a room (PV) that matches and assigns it to them.

StorageClass:
A StorageClass in Kubernetes defines how storage should be dynamically provisioned.
It allows users to automatically create PersistentVolumes (PVs) when they create PersistentVolumeClaims (PVCs), without needing pre-provisioned storage.

📦 Why StorageClass is Useful:
Eliminates the need for manually creating PVs.
Allows setting different types of storage (e.g., SSD, HDD, slow, fast).
Tells Kubernetes which storage plugin/driver to use (like AWS EBS, GCE PersistentDisk, NFS, etc.).
Supports parameters like disk type, encryption, replication, etc.

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

```
kind: StorageClass
metadata:
  name: fast-storage
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  encrypted: "true"
  fsType: ext4
```

📥 Example PVC using this StorageClass:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: fast-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

➡️ When this PVC is created:
Kubernetes uses the fast-storage StorageClass.
It provisions a new EBS volume of 5Gi automatically.
Binds the new PV to the PVC.

You’re deploying a PostgreSQL database. You define a PVC that requests 100Gi of SSD storage using a fast StorageClass. Kubernetes automatically provisions and mounts the right disk behind the scenes. No manual PVs required.

28. What are Labels and Selectors in Kubernetes?
In Kubernetes, labels are key-value pairs attached to objects like Pods, Services, Deployments, etc.
They are used to identify, organize, and select groups of objects.
Selectors are queries used to filter and match objects based on their labels.

```
labels:
  app: my-app
  environment: production
  version: v1
```

🔍 Selectors – What are they?
Used by controllers (like Services, Deployments, ReplicaSets) to select a group of Pods based on labels.
Two types:
Equality-based (e.g., app=my-app)
Set-based (e.g., environment in (dev, test))
Ex:

```
selector:
  app: my-app
  environment: production
```

📦 Real-World Use Case:
Let’s say you have 3 versions of your app (v1, v2, v3).
You can label each pod with version: vX and create different services or deployments that select specific versions via label selectors.

29.What is the Role of Annotations in Kubernetes?
In Kubernetes, annotations are key-value pairs like labels, but they are used to store non-identifying metadata about objects — information that tools or systems use, but not meant for selecting or filtering objects like labels.

🔍 Key Differences: Label vs Annotation

Feature	**Label**	**Annotation**
Purpose	Identify, filter, group objects	Store non-identifying metadata
Used by	Selectors, controllers	Tools, controllers, monitoring systems
Size Limit	Small (for quick lookups)	Larger values allowed (e.g., long text, JSON)
Examples	`app=nginx`, `env=prod`	`kubectl.kubernetes.io/last-applied-configuration`

```
metadata:
  name: my-pod
  labels:
    app: my-app
  annotations:
    description: "This pod runs the main application"
    backup-timestamp: "2025-07-05T10:00:00Z"
```

30. Explain the Lifecycle of a Pod in Kubernetes?
A Pod's lifecycle in Kubernetes goes through a series of phases and states — from scheduling to running to termination.
Kubernetes manages this flow automatically based on the Pod's configuration, health checks, and events in the cluster.

🔧 Detailed Pod Lifecycle Flow:
Pod is Created:
Via kubectl apply, CI/CD pipeline, or Deployment controller.
Status: Pending

Scheduler Assigns Node:
Scheduler finds a suitable node based on resources and constraints.
Pod is scheduled to that node.

Container Runtime Starts Containers:
Docker, containerd, or CRI-O pulls the image and starts the containers.
Status: Running

Health Checks:
Kubernetes uses liveness and readiness probes to monitor container health.
If readinessProbe fails → traffic is not sent to the pod.
If livenessProbe fails → container may be restarted.

Pod Completion:
For batch jobs: containers finish successfully → Status: Succeeded
If containers exit with error → Status: Failed

Pod Deletion:
On deletion, preStop hook runs (if configured).
Grace period is honored (default: 30s).
Kubernetes cleans up resources (volumes, IPs, etc.).

🔌 Hooks and Events in Lifecycle:

Hook	When It Triggers
`postStart`	After container starts
`preStop`	Before container is terminated

“We use readiness probes to ensure the pod doesn’t receive traffic before it’s ready, and liveness probes to auto-restart unhealthy containers. For jobs, we monitor the pod’s status to check if it reached Succeeded or Failed.”

31.How to Debug a Failed Pod in Kubernetes?

To debug a failed pod, I follow a systematic approach using kubectl commands to check the pod's status, logs, events, and container behavior.
Here's how I do it:
🔍 1. Check Pod Status:

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

Status: CrashLoopBackOff, Error, ImagePullBackOff, etc.
Look at the Events section in describe — it shows scheduling failures, image issues, etc.

📜 2. View Pod Logs:

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

If the pod has multiple containers:

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

For failed pods (exited too fast):

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

🛠 3. Exec Into the Pod (If Still Running):

```
kubectl exec -it <pod-name> -- /bin/sh
```

Useful to inspect file paths, configs, network tools like ping, curl, etc.

🔄 4. Check Liveness/Readiness Probes:

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

Misconfigured probes can cause healthy containers to restart repeatedly.

▫ 5. Check Image and Pull Issues:

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

Errors like ImagePullBackOff or ErrImagePull usually mean:
Wrong image name
Image not found in registry
Bad credentials for private registry

"We once had a pod failing with CrashLoopBackOff.
Logs showed the app was failing due to a missing config file.
We fixed it by mounting a ConfigMap and added readiness probes to prevent early traffic routing."

32. What is CRI in Kubernetes?
CRI stands for Container Runtime Interface.
It is a standard API used by the Kubelet to communicate with container runtimes like containerd, CRI-O, or Docker (via dockershim).
This allows Kubernetes to abstract away the container runtime, making it pluggable and flexible.

🔧 Why CRI Was Introduced:
In early Kubernetes versions, Docker was hard-coded.
To make Kubernetes more flexible and support other runtimes, CRI was introduced.
Now, Kubernetes doesn’t depend on Docker — it talks to any CRI-compliant runtime.

Kubelet sends pod/container instructions to CRI.
CRI runtime handles image pulling, container creation, logging, etc.

Docker is not CRI-compliant directly, which is why Kubernetes removed dockershim in version 1.24+.
Now, most clusters use containerd or CRI-O underneath

33.How to Secure a Kubernetes Cluster?
Securing a Kubernetes cluster involves protecting the control plane, worker nodes, network, API access, and workloads.
I follow a layered approach — often called "defense in depth."

🔐 1. Secure the API Server:
Enable Role-Based Access Control (RBAC):
Define fine-grained permissions for users and service accounts.

```
kubectl create role ...
kubectl create rolebinding ...
```

Enable Authentication & Authorization:
Use OIDC, LDAP, or certificates for secure access.

Use TLS everywhere:
Ensure all API server communication is encrypted.

Audit Logging:
Enable audit-log-path in the API server to log access and changes.

🛡️ 2. Use Role-Based Access Control (RBAC) :
Give least privilege to users and service accounts.
Avoid using cluster-admin unless absolutely necessary.
Use namespace-specific roles when possible.

🔒 3. Isolate Workloads with Namespaces and Network Policies:
Use namespaces to separate environments (dev/test/prod).
Apply NetworkPolicies to control pod-to-pod traffic.

```
kind: NetworkPolicy
...
```

🧼 4. Secure Images and Containers:
Use images from trusted registries only.
Scan images for vulnerabilities (with tools like Trivy, Aqua, Anchore).
Use read-only root filesystems, drop capabilities, and avoid running as root:

```
securityContext:
  runAsNonRoot: true
  readOnlyRootFilesystem: true
```

🔐 5. Use Pod Security Policies or Pod Security Admission (PSA):
Restrict dangerous pod behaviors (privileged mode, hostPath mounts, etc.)
Use Pod Security Admission (default since K8s v1.25):

```
labels:
  pod-security.kubernetes.io/enforce: restricted
```

🧱 6. Enable Node Security
Use minimal OS (like Flatcar or Bottlerocket).
Regularly patch nodes and Kubernetes binaries.
Use firewall rules and disable unused ports.

📦 7. Use Secrets and ConfigMaps Securely
Store secrets in Kubernetes Secrets, not in environment variables or images.
Use encryption at rest for secrets (via kube-apiserver settings).

34.What is Kubernetes Federation?
Kubernetes Federation is a feature that allows you to manage multiple Kubernetes clusters as if they were a single cluster.
It enables multi-cluster management, high availability, disaster recovery, and global traffic distribution across geographically distributed clusters.

🧠 Why Use Federation?
✅ High Availability – Spread workloads across clusters in different regions.
🌍 Disaster Recovery – If one cluster fails, workloads can be rebalanced to another.
📡 Latency Optimization – Route users to the closest cluster.
🔄 Centralized Control – Deploy apps across all clusters from a single control plane.

35. How to handle stateful applications in Kubernetes?
In Kubernetes, stateful applications are handled using StatefulSets, PersistentVolumes (PVs), and PersistentVolumeClaims (PVCs).
These components ensure stable storage, consistent network identity, and ordered startup/shutdown, which are essential for managing state.

🔍 Key Components to Handle Stateful Apps:
🧱 1. StatefulSet
A controller that manages stateful pods.
Ensures:
Stable network identity (e.g., mysql-0, mysql-1…)
Persistent storage (unique volume for each pod)
Ordered deployment, scaling, and termination

📦 2. PersistentVolume (PV) and PersistentVolumeClaim (PVC)
PV: actual storage resource in the cluster.
PVC: request for storage made by the pod.
StatefulSets automatically create one PVC per pod, ensuring data persistence even if a pod is deleted or rescheduled.

🌐 3. Headless Services for Network Identity
Used with StatefulSets to allow DNS-based stable naming.

```
spec:
  clusterIP: None
```

This gives pods predictable DNS names like:
mysql-0.mysql.default.svc.cluster.local

“For stateful applications, I use StatefulSets with PersistentVolumeClaims and headless services to provide consistent storage and identity. This ensures that data isn't lost when pods restart or reschedule. I also take care of backups, readiness probes, and graceful shutdowns.”

36. What is CRD?
A Custom Resource Definition (CRD) is a way to extend Kubernetes by defining your own custom resources, just like built-in ones such as Pods or Deployments.
With a CRD, you can create new types of objects that behave like native Kubernetes resources.

📦 What is a Custom Resource?
A Custom Resource is an object defined by the user, made available through a CRD.
Once registered, you can manage it using kubectl, like:

```
kubectl get myresources
kubectl apply -f myresource.yaml
```

“CRDs allow Kubernetes to be extended with new resource types. We used CRDs to define custom objects, and our controller would watch and act on them using Kubernetes-native patterns. It gave us flexibility to implement domain-specific logic cleanly.”

37. Explain Kubernetes Operators?
A Kubernetes Operator is a method of automating the management of complex, stateful applications on Kubernetes using Custom Resource Definitions (CRDs) and controllers.

It captures human operational knowledge (like install, upgrade, backup, failover) into software that runs inside the cluster.

🎯 What Problems Do Operators Solve?
Operators automate tasks like:
Deploying and scaling complex apps
Performing upgrades safely
Taking scheduled backups
Restoring failed services
Managing HA and failover

🧠 Example: PostgreSQL Operator
Let’s say you want to run PostgreSQL in Kubernetes and:
Ensure only 1 primary exists
Automatically scale replicas
Take backups to S3
Handle failover
With a PostgreSQL Operator, you define a PostgresCluster CR, and the Operator does the rest — installing, configuring, monitoring, healing, and upgrading PostgreSQL automatically.

⚙️ How It Works (Workflow)
You define a CRD like DatabaseCluster.
The Operator runs a controller watching that CRD.
When you kubectl apply -f my-db.yaml, the Operator reacts to that custom resource.
It performs all the steps an SRE would normally do — automatically.

38. What are Admission Controllers?
“Admission Controllers are Kubernetes plugins that intercept API requests before they are persisted. They help enforce security, governance, and consistency. We used both validating and mutating webhooks to enforce policies like resource limits and inject default configurations.”

🧠 Popular Built-in Admission Controllers:

Controller	Purpose
`NamespaceLifecycle`	Prevents deletion of default namespaces
`LimitRanger`	Enforces resource limits
`ServiceAccount`	Auto-adds service accounts
`PodSecurityPolicy`	(Deprecated) Enforces security settings
`ValidatingAdmissionWebhook`	Integrate with custom webhooks for validation
`MutatingAdmissionWebhook`	Integrate with webhooks to modify requests

39. What is the use of etcd in Kubernetes?
etcd is a distributed, consistent key-value store used by Kubernetes to store all cluster state and configuration data.
It is the single source of truth for Kubernetes — everything from nodes, pods, secrets, and config maps to the current state of deployments is stored in etcd.

🧠 Why etcd Is Important:
Stores Kubernetes objects (Pods, Deployments, Services, etc.)
Ensures high availability and data consistency
Powers the reconciliation loop (controllers compare desired vs actual state)
Critical for cluster recovery and backup

⚙️ How It Works in Kubernetes:
You kubectl apply -f pod.yaml
Kubernetes API server validates the request
The API server writes the Pod spec to etcd
The scheduler and controller-manager watch etcd and take action

40. What is a Helm chart?

A Helm Chart is a collection of YAML templates and metadata that defines a Kubernetes application, including all of its components like Deployments, Services, ConfigMaps, etc.
Helm uses these charts to package, deploy, version, and manage applications on a Kubernetes cluster — just like apt or yum does for Linux packages.

📦 What a Helm Chart Contains:
A typical Helm chart has the following structure:

```
mychart/
├── Chart.yaml        # Metadata about the chart (name, version, etc.)
├── values.yaml       # Default values for templates (like variables)
├── templates/        # Kubernetes manifest templates
│   ├── deployment.yaml
│   ├── service.yaml
│   └── _helpers.tpl  # Reusable snippets
```

🧠 Why Use Helm Charts?
✅ Templating – Reuse YAML templates with different values
✅ Versioning – Track versions of your app like Git tags
✅ Dependency Management – Include other charts as dependencies
✅ Rollbacks – Revert to previous releases easily
✅ Simplifies CI/CD – Automate app deployments with versioned packages

⚙️ Common Helm Commands:

Command	Purpose
`helm install myapp ./mychart`	Deploys the app using the chart
`helm upgrade myapp ./mychart`	Updates the app with changes
`helm rollback myapp 1`	Rolls back to a previous release
`helm uninstall myapp`	Deletes the release
`helm repo add` / `helm search`	Use remote chart repositories

41.How does Kubernetes handle logging and monitoring?
Kubernetes doesn't handle logging and monitoring directly, but it provides the infrastructure and extensibility to integrate third-party tools for logging, metrics, and tracing.
In production, we usually integrate tools like ELK, Prometheus, Grafana, Loki, etc., for full observability.

🔍 Logging in Kubernetes

📦 Where Are Logs Stored?
Kubernetes does not store logs — containers write logs to stdout/stderr
Logs are stored on the node's file system under /var/log/containers/ or captured by log drivers

🧰 Common Logging Tools:

Tool	Purpose
Fluentd	Log collector and forwarder
Logstash	Log processing
Elasticsearch	Log storage and indexing
Kibana	Visualization of logs
Loki	Lightweight logging by Grafana
EFK / ELK	Common stacks for centralized logging

🧪 How It Works:
Fluentd (or another agent) runs as a DaemonSet on all nodes.
It reads logs from /var/log/containers/.
Sends logs to Elasticsearch or Loki.
Use Kibana or Grafana to view/search logs.

📈 Monitoring in Kubernetes:

Tool	Purpose

Prometheus	Collect metrics from nodes, pods, apps
Grafana	Visualize metrics dashboards
Kube State Metrics	Provides info on the health/state of cluster resources
Node Exporter	OS-level metrics
Alertmanager	Sends alerts via email, Slack, etc.

🔧 How It Works:
Prometheus scrapes metrics from:
      Kubernetes API
      Node exporter
      Kube state metrics
      Application endpoints (/metrics)
Stores the data in time-series format.
Grafana connects to Prometheus to display custom dashboards.
Alertmanager triggers notifications when thresholds are breached.

42. What tools are commonly used for monitoring Kubernetes?*
In Kubernetes, monitoring involves tracking the health and performance of the cluster, nodes, pods, and applications.
The most commonly used tools include Prometheus, Grafana, Kube State Metrics, Node Exporter, Alertmanager, and tools like Datadog, New Relic, and Dynatrace for enterprise use.

43. What is Taint and Toleration?
Taints and Tolerations work together to control which Pods can be scheduled on which nodes.
A Taint is applied to a node to repel Pods that do not tolerate the taint.
A Toleration is applied to a Pod to allow it to be scheduled on nodes with matching taints.

🔍 How They Work Together:
When a node is tainted, it tells the scheduler:
“Don’t schedule any Pod here unless it explicitly tolerates this taint.”
Pods with matching tolerations can be scheduled on those tainted nodes.
This mechanism helps dedicate nodes for special purposes, such as:
Running critical workloads
Isolating noisy neighbors
Reserving nodes for maintenance or special hardware

44. What is an Affinity and Anti-Affinity?
Affinity and Anti-Affinity are rules that influence the Kubernetes Scheduler to prefer or avoid placing Pods on certain nodes or near other Pods, based on labels and topology.
Affinity expresses a preference or requirement to schedule Pods close to other Pods or nodes with specific labels.
Anti-Affinity expresses a preference or requirement to schedule Pods away from other Pods or nodes with specific labels.

🔍 Types of Affinity:

Type	Description
Node Affinity	Schedule Pods on nodes that have specific labels (like node selectors but more expressive)

Pod Affinity	Schedule Pods on the same node or close to other Pods with specified labels
Pod Anti-Affinity	Schedule Pods away from other Pods with specified labels to avoid co-location

🧩 Example: Pod Anti-Affinity
To avoid placing two replicas of the same app on the same node (for high availability):

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

🧩 Example: Node Affinity
To schedule Pods only on nodes labeled disktype=ssd:

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

45.** How does Horizontal Pod Autoscaler work? (or) How you magage autoscaling?
The Horizontal Pod Autoscaler (HPA) automatically adjusts the number of pod replicas in a deployment, replica set, or stateful set based on observed CPU utilization, memory usage, or custom metrics.
It helps maintain the desired performance by scaling pods out (adding more) when demand increases and scaling in (reducing pods) when demand decreases.

🔧 How HPA Works:
Metrics Collection:
HPA periodically queries the Metrics Server (or custom metrics API) to get current resource usage (e.g., CPU, memory).

Decision Making:
It compares the observed metrics with the target metrics defined in the HPA spec.

Scaling Action:
If metrics exceed or fall below the target thresholds, HPA updates the .spec.replicas field of the workload to increase or decrease the number of pods.

⚙️ Example HPA YAML:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: myapp-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: myapp
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 60
```

🧠 Important Points:
HPA requires Metrics Server to be installed for CPU/memory metrics.
Supports custom metrics and external metrics (e.g., queue length).
Works with Deployments, ReplicaSets, StatefulSets, and more.
HPA periodically (default every 15 seconds) checks and adjusts replicas.
Scaling decisions respect min and max replicas to avoid thrashing.

46.** What is Vertical Pod Autoscaler?
Vertical Pod Autoscaler (VPA) automatically adjusts the CPU and memory requests/limits of a Pod based on its actual usage over time.
This ensures that Pods get the right amount of resources — not too little (which can cause OOM kills) or too much (which wastes resources).

📦 Why Use VPA?
Prevents under-provisioning (app crashes or slow performance)
Avoids over-provisioning (wasted CPU/memory)
Useful for non-distributed workloads like batch jobs or stateful apps
Complements HPA in some scenarios, but both cannot manage the same resource simultaneously

🔍 How VPA Works:
Recommender:
Continuously analyzes historical usage metrics of Pods.

Updater:
Deletes and restarts Pods with updated resource requests/limits (optional depending on mode).

Admission Controller:
Applies recommendations at Pod creation time for new Pods.

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

47. ** What is Cluster Autoscaler?
Cluster Autoscaler (CA) automatically adds or removes nodes in a Kubernetes cluster based on the scheduling needs of Pods.
It scales out by adding nodes when Pods are pending due to insufficient resources, and scales in by removing underutilized nodes if workloads no longer need them.

🔁 How It Works:
Watches for unschedulable Pods (e.g., due to lack of CPU/RAM).
If needed, it asks the cloud provider (AWS/GCP/Azure) to scale out the node

group (e.g., via Auto Scaling Group in AWS).
It also continuously looks for underutilized nodes (running no or very few Pods) and drains and deletes them if they are safe to remove.
It respects Pod disruption budgets, taints/tolerations, and node group constraints.

⚙️ Cluster Autoscaler Architecture (Cloud-native)
On AWS (EKS): integrates with Auto Scaling Groups (ASGs)
On GCP (GKE): works with Managed Instance Groups
On Azure (AKS): works with VM Scale Sets

🧠 Key Features:

Feature	Description
Auto scale out	Adds nodes when Pods are pending
Auto scale in	Removes underutilized nodes to save cost
Pod disruption-aware	Doesn’t evict critical Pods without replacement
Cloud provider integration	Communicates directly with cloud APIs to change cluster size
Node groups awareness	Can work across multiple node groups with custom taints, labels

48.** Explain how Kubernetes Networking works?
Kubernetes networking is designed to make communication between Pods, Services, and external systems seamless and scalable.
Each Pod gets its own IP address, and Kubernetes ensures that:
All Pods can talk to all other Pods across Nodes (Pod-to-Pod)
Pods can talk to Services (Pod-to-Service)
Services can be exposed outside the cluster (Service-to-External)

🔑 Core Networking Principles in Kubernetes:
Flat network: Every Pod has a unique IP and can communicate with every other Pod without NAT.
No need to manually assign ports or IPs for Pods or Services.
Kubelet and kube-proxy help implement networking logic on each node.

🔁 Key Components in Kubernetes Networking:

Component	Role
Pod Network	Each Pod gets an IP from a **cluster-wide CIDR**; implemented by a CNI plugin
Service	Abstraction over a set of Pods; gets a **stable virtual IP** (ClusterIP)
kube-proxy	Maintains network rules on each node for directing traffic to the right Pods
DNS	Internal DNS resolution for services (e.g., `myservice.default.svc.cluster.local`)
Ingress	Manages external access (HTTP/HTTPS) to internal Services
Network Policies	Define **firewall-like rules** to restrict traffic between Pods

📦 How Pods Communicate:
Each Pod gets its own network namespace and IP address.
Pods talk to each other directly via IP.
The underlying CNI (Container Network Interface) plugin (like Calico, Flannel,

Cilium) sets up networking across nodes.

49.** How does RBAC work in Kubernetes?
RBAC (Role-Based Access Control) in Kubernetes is a mechanism that controls who can do what within the cluster.
It uses roles, bindings, and subjects to grant or restrict access to Kubernetes resources.

🔐 Key Concepts in Kubernetes RBAC:

Component	Description
Role	A set of permissions (rules) for resources within a **namespace**
ClusterRole	A set of permissions at the **cluster level** or across all namespaces
RoleBinding	Grants a Role to a user/group/service account **within a namespace**
ClusterRoleBinding	Grants a ClusterRole to a user/group/service account **cluster-wide**
Subject	The entity (user, group, or service account) receiving the permission

🧩 RBAC Example:
Let's say you want to give read-only access to Pods in the dev namespace:
1. 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"]
```

2. Bind the Role to a user:
```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: dev
subjects:
- kind: User
  name: prasanth@example.com
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

🔍 ClusterRole Example:
To allow someone to view nodes or create namespaces:
```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cluster-admin-lite
rules:
- apiGroups: [""]
  resources: ["nodes", "namespaces"]
  verbs: ["get", "list", "create"]
```

Use Cases:

Give developers access only to their own namespace
Allow CI/CD service accounts to deploy workloads
Grant read-only access to monitoring teams
Restrict sensitive operations like deleting resources

50. What is a ServiceAccount?
A ServiceAccount in Kubernetes is an account used by Pods to interact with the Kubernetes API server.
It's different from a normal user — it's for processes inside the cluster, not for humans.
Every Pod runs under a ServiceAccount identity, and that identity is used for authentication and authorization.

📦 Use Cases:
Give a Pod permission to list secrets or configmaps
Allow CI/CD tools (like ArgoCD or Jenkins) to talk to the cluster
Control access between microservices
Secure access to cloud resources via IAM roles (e.g., IRSA in EKS)

51. What is a sidecar container?
A sidecar container is a secondary container that runs in the same Pod as the main application container and helps extend or support its functionality.
Since all containers in a Pod share the same network and storage, sidecars can easily communicate and collaborate with the main app.

🔧 How Sidecars Work:
Sidecar containers run alongside the main container inside the same Pod.
They share resources like:
Network namespace (so they can talk over localhost)
Volumes (for shared data)
This allows sidecars to observe, modify, or enhance the primary container’s behavior.

💡 Service Mesh Example (Istio):
In Istio, an Envoy sidecar is automatically injected into each Pod. It intercepts all incoming/outgoing traffic and enables features like:
mTLS encryption
Traffic routing
Observability (telemetry)
Retries and circuit breaking

52. What is service mesh?
A Service Mesh is an infrastructure layer that handles secure, reliable, and observable communication between services in a microservices architecture — without changing the application code.

It does this by injecting sidecar proxies into Pods, which intercept all network traffic to and from the service and enforce policies like mTLS, retries, rate limits, observability, etc

🧩 How It Works (Basic Flow):
A sidecar proxy (like Envoy) is injected into each Pod.
All traffic to/from the app goes through the proxy.
A control plane configures these proxies centrally (e.g., routes, policies).

```
[Service A] <---> [Sidecar A] <=====> [Sidecar B] <---> [Service B]
```

🌐 Most Common Service Meshes:
```
| Service Mesh                 | Built On         | Best Known For
|
| --------------------------- | ---------------- |
---------------------------------------------------------------------- |
| **Istio**                    | Envoy            | Most powerful and popular,
rich features, used in large-scale systems |
```

Linkerd	Rust-based proxy	Simpler, lightweight, fast setup
Consul	Envoy	From HashiCorp, integrates with Vault and Terraform
AWS App Mesh	Envoy	Managed mesh for EKS/Fargate
Open Service Mesh (OSM)	Envoy	Lightweight, supported by Microsoft, SMI-compliant

53. How to deploy a Kubernetes cluster on AWS/GCP/Azure?
You can deploy Kubernetes clusters on AWS, GCP, and Azure using their managed services or by setting up self-managed clusters.
Managed Kubernetes services simplify cluster creation, management, and scaling by handling control plane tasks for you.

🔷 1. AWS – Using Amazon EKS (Elastic Kubernetes Service):
Steps:
Create an EKS cluster using AWS Management Console, AWS CLI, or eksctl (a popular CLI tool).
Configure IAM roles and policies for the EKS control plane and worker nodes.
Set up VPC networking (private subnets, security groups).
Launch worker nodes (EC2 instances or managed node groups).
Configure kubectl to connect to the cluster (aws eks update-kubeconfig).
Deploy applications or add addons like CoreDNS, VPC CNI, Metrics Server.
Popular Tools:
eksctl – simple CLI tool to create EKS clusters quickly.
AWS CLI and CloudFormation for infrastructure as code.

🔷 3. Azure – Using Azure Kubernetes Service (AKS):
Steps:
Create an AKS cluster using Azure Portal, Azure CLI (az aks create), or ARM templates.
Configure networking (Azure CNI or Kubenet), node pools, and RBAC.
Set up Azure Active Directory (AAD) integration for authentication.
Connect kubectl (az aks get-credentials).
Deploy applications and enable addons like monitoring and scaling.
Popular Tools:
Azure CLI.
Azure Resource Manager (ARM) templates.
Terraform.

54. What are best practices for CI/CD with Kubernetes?
CI/CD with Kubernetes focuses on automating build, test, and deployment of containerized applications to Kubernetes clusters, ensuring fast, reliable, and repeatable releases.

🔑 Best Practices for CI/CD with Kubernetes:
1. Use Container Images & Registries
Build container images in CI pipelines using Docker or Buildah.
Push images to a trusted container registry (Docker Hub, ECR, GCR, ACR).
Tag images uniquely with commit SHA or semantic versioning.
Use image scanning tools to detect vulnerabilities before deployment.

2. Declarative Manifests & Infrastructure as Code
Store Kubernetes manifests (YAML files) in version control (Git).
Use Helm charts or Kustomize to templatize and manage manifests.
Keep infrastructure as code for cluster resources (ingress, configmaps, secrets).
Review changes through Git workflows (merge requests, pull requests).

3. Use GitOps Practices
Use Git as the single source of truth for app and infra manifests.
Deploy using tools like ArgoCD or Flux that sync Git state to the cluster automatically.

Rollback easily by reverting Git commits.

4. Automated Testing
Run unit and integration tests in CI before building images.
Perform smoke tests or canary tests post-deployment.
Use Kubernetes namespaces or ephemeral clusters for isolated testing.

5. Deploy Strategies
Implement rolling updates to avoid downtime.
Use canary or blue-green deployments for safer releases.
Automate rollback on failures using Kubernetes Deployment features.

6. Secrets and Configuration Management
Manage secrets securely using Kubernetes Secrets or external tools like Vault.
Avoid committing secrets to Git.
Use ConfigMaps for non-sensitive config data.

7. Monitoring & Alerting
Integrate monitoring tools (Prometheus, Grafana) to track app health.
Setup alerts on deployment failures or abnormal metrics.
Collect logs using centralized logging solutions (ELK, Fluentd).

55.How do you switch between Kubernetes contexts?
You can switch between Kubernetes contexts using the "kubectl config use-context" command. Each context defines a cluster, user, and namespace combination.
This helps manage multiple environments like dev, staging, and production safely from a single machine.

To see list of contexts:
```
kubectl config get-contexts
```

🔁 How to Switch Contexts:
```
kubectl config use-context prod-context
```

📌 How to View Current Context:
```
kubectl config current-context
```