Kubernetes Manifest File Explained (with Examples)

A Kubernetes **manifest file** (written in YAML or JSON) defines the desired state of your application in the cluster. It tells Kubernetes what to deploy, how to run it, and how to manage it.

Basic Structure of a Kubernetes Manifest

Every Kubernetes manifest has 4 key sections:

1. **apiVersion** – Which Kubernetes API to use (e.g., apps/v1, v1).
2. **kind** – The type of resource (e.g., Pod, Deployment, Service).
3. **metadata** – Name, labels, and annotations for the resource.
4. **spec** – Desired state (e.g., container image, ports, replicas).

5. **Pods** → Run a single instance of a container.
6. **Deployments** → Manage Pods (scaling, updates).
7. **Services** → Expose Pods internally/externally.
8. **ConfigMaps** → Store configuration.
9. **PV/PVC** → Manage storage.

Example 1: Simple Pod Manifest

A **Pod** is the smallest deployable unit in Kubernetes (1+ containers).

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
      ports:
        - containerPort: 80
```

Explanation:

- apiVersion: v1 → Uses the core Kubernetes API.
- kind: Pod → Defines a Pod.
- metadata → Names the Pod nginx-pod and adds a label (app: nginx).
- spec → Runs an nginx:latest container exposing port 80.

Example 2: Deployment Manifest

A **Deployment** manages Pods (handles scaling, updates, rollbacks).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

Explanation:

- replicas: 3 → Runs 3 identical Pods.
- selector.matchLabels → Ensures the Deployment manages Pods with app: nginx.
- template → Defines the Pod spec (same as Example 1).

Example 3: Service Manifest

A **Service** exposes Pods to the network (internal or external).

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

Explanation:

- selector.app: nginx → Routes traffic to Pods with app: nginx.
- port: 80 → Service listens on port 80.
- targetPort: 80 → Forwards traffic to Pod's port 80.
- type: ClusterIP → Makes the Service internal (use LoadBalancer for external access).

Example 4: ConfigMap Manifest

A **ConfigMap** stores non-sensitive configuration data.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  database_url: "mysql://db:3306"
  log_level: "debug"
```

Explanation:

- data → Key-value pairs of configuration.
- Pods can access these values as environment variables or mounted files.

Example 5: PersistentVolume (PV) & PersistentVolumeClaim (PVC)

PV = Cluster storage resource.

PVC = Request for storage by a Pod.

PersistentVolume (PV)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

PersistentVolumeClaim (PVC)

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

Explanation:

- PV defines available storage (hostPath, AWS EBS, NFS, etc.).
- PVC requests storage from PV (used by Pods via volumeMounts).

How to Apply a Manifest?

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

Key Takeaways

1. **Pods** → Run a single instance of a container.
2. **Deployments** → Manage Pods (scaling, updates).
3. **Services** → Expose Pods internally/externally.
4. **ConfigMaps** → Store configuration.
5. **PV/PVC** → Manage storage.

In Kubernetes, several **CLI (Command Line Interface) tools** are used for different purposes, such as cluster management, debugging, and interacting with resources. Below is a breakdown of the key CLI tools, their differences, and when to use them.

1. kubectl (Kubernetes Control Tool)

Primary Use: The main CLI tool for interacting with Kubernetes clusters.

Key Features:

- Deploy & manage applications (`kubectl apply`, `kubectl create`).
- Debug & inspect resources (`kubectl logs`, `kubectl describe`).
- Manage cluster state (`kubectl get`, `kubectl delete`).

Example:

```
kubectl get pods -n my-namespace
kubectl apply -f deployment.yaml
```

2. kubeadm (Kubernetes Administration Tool)

Primary Use: Bootstrapping and managing Kubernetes clusters.

Key Features:

- Initialize a control plane (`kubeadm init`).
- Join worker nodes (`kubeadm join`).
- Upgrade clusters (`kubeadm upgrade`).

Example:

```
kubeadm init --pod-network-cidr=10.244.0.0/16
kubeadm join <control-plane-ip>:<port> --token <token>
```

3. kubectx & kubens (Switching Contexts & Namespaces)

Primary Use: Quickly switch between Kubernetes clusters (`kubectx`) and namespaces (`kubens`).

Key Features:

- Avoid repetitive `--context` or `--namespace` flags.
- Useful for multi-cluster and multi-namespace workflows.

Example:

```
kubectx my-cluster  # Switch cluster
kubens my-namespace  # Switch namespace
```

4. helm (Kubernetes Package Manager)

Primary Use: Deploying and managing applications using Helm charts (pre-configured Kubernetes manifests).

Key Features:

- Install, upgrade, and roll back applications (helm install, helm upgrade).
- Share reusable application templates (Helm charts).

Example:

```
helm install nginx bitnami/nginx
helm list
```

5. k9s (Terminal UI for Kubernetes)

Primary Use: Interactive terminal-based dashboard for Kubernetes.

Key Features:

- Real-time monitoring of resources.
- Quick navigation between pods, logs, and events.

Example:

```
k9s --namespace my-namespace
```

6. kubectl debug (Debugging Pods)

Primary Use: Troubleshooting running pods by creating ephemeral debug containers.

Key Features:

- Debug without modifying the original pod.
- Supports busybox, netshoot, etc.

Example:

```
kubectl debug my-pod -it --image=busybox
```

7. kubetail (Log Aggregation)

Primary Use: Stream logs from multiple pods simultaneously.

Key Features:

- Combines logs from multiple pods (kubetail my-app).
- Similar to kubectl logs but for multiple pods.

Example:

```
kubetail -l app=nginx
```

8. stern (Multi-Pod Log Tailing)

Primary Use: Like kubetail, but more powerful for log tailing.

Key Features:

- Regex-based pod selection.
- Colorized output for better readability.

Example:

```
stern "app-.*" -n my-namespace
```

9. minikube (Local Kubernetes Cluster)

Primary Use: Running a single-node Kubernetes cluster locally.

Key Features:

- Ideal for development and testing.
- Supports add-ons (Ingress, Dashboard, etc.).

Example:

```
minikube start --driver=docker
minikube dashboard
```

10. kubefwd (Port Forwarding for Services)

Primary Use: Bulk port-forwarding Kubernetes services for local development.

Key Features:

- Forwards all services in a namespace (kubefwd svc -n my-ns).
- Simulates a local environment.

Example:

```
kubefwd services -n my-namespace
```

Comparison Table

Tool	Primary Use Case	Key Difference
kubectl	Main CLI for Kubernetes operations	Default tool for managing resources.
kubeadm	Cluster bootstrapping & management	Used by admins, not for daily app management.
kubectx/kubens	Switching clusters/namespaces	Simplifies context switching.
helm	Package management (charts)	Manages pre-packaged applications.
k9s	Interactive terminal UI	Visual alternative to kubectl.
kubectl debug	Debugging pods in real-time	Adds ephemeral debug containers.

Tool	Primary Use Case	Key Difference
kubetail/stern	Multi-pod log streaming	Better than kubectl logs for multiple pods.
minikube	Local Kubernetes development	Runs a lightweight single-node cluster.
kubefwd	Bulk port-forwarding services	Helps in local development.

When to Use Which Tool?

- **For daily Kubernetes operations** → kubectl, k9s, kubectx/kubens.
- **For cluster setup & upgrades** → kubeadm.
- **For deploying complex apps** → helm.
- **For debugging** → kubectl debug, stern.
- **For local development** → minikube, kubefwd.

Here’s a detailed comparison of **Kind (Kubernetes in Docker), Minikube, Kubeadm**, and other Kubernetes container management tools, including their use cases, pros, cons, and differences:

1. Kind (Kubernetes in Docker)

Primary Use: Lightweight Kubernetes clusters running inside Docker containers (ideal for CI/CD and development).

Key Features:

- Runs Kubernetes nodes as Docker containers (no VM overhead).
- Supports multi-node clusters (control plane + workers).
- Fast startup time (~30 seconds).

Pros:

✅ Extremely lightweight (uses Docker).
✅ Good for testing multi-node setups.
✅ Works well in CI/CD pipelines.

Cons:

❌ Limited to Docker (no other container runtimes).
❌ Not suitable for production.

Example:

```
kind create cluster --name my-cluster
```

2. Minikube

Primary Use: Local single-node Kubernetes cluster for development/testing.

Key Features:

- Runs a VM-based (or container-based) single-node cluster.
- Supports addons (Ingress, Dashboard, etc.).

Pros:

✅ Easy setup for beginners.
✅ Supports multiple drivers (Docker, VirtualBox, Hyper-V).
✅ Good for local experimentation.

**Cons:

❌ Only single-node (no true multi-node testing).
❌ Slightly heavier than Kind.

Example:

```
minikube start --driver=docker
minikube addons enable ingress
```

3. Kubeadm

Primary Use: Production-grade cluster setup (manual control over nodes).

Key Features:

- Bootstraps a Kubernetes cluster on bare metal/VMs.
- Supports HA (High Availability) setups.

  Pros:

  ✅ Used in production environments.

  ✅ Full control over cluster components (kubelet, API server, etc.).

  ✅ Supports multi-node and HA clusters.

  Cons:

  ❌ Requires manual configuration (not beginner-friendly).

  ❌ No built-in load balancer (requires external setup).

  Example:

```
kubeadm init
kubeadm join <control-plane-ip>:<port> --token <token>
```

4. MicroK8s (by Canonical)

Primary Use: Lightweight, single-command Kubernetes for edge/IoT and development.

Key Features:

- Runs as a snap package (Linux optimized).
- Includes built-in addons (DNS, Dashboard, Istio).

  Pros:

  ✅ Fast setup (`microk8s start`).

  ✅ Low resource usage.

  ✅ Good for edge computing.

  Cons:

  ❌ Limited to Linux (best on Ubuntu).

  ❌ Less flexible than Kubeadm.

  Example:

```
microk8s start
microk8s enable dns dashboard
```

5. K3s (Lightweight Kubernetes by Rancher)

Primary Use: Ultra-light Kubernetes for edge/IoT and resource-constrained environments.

Key Features:

- Removes legacy and alpha features to reduce size.
- Bundles SQLite instead of etcd (optional).

  Pros:
  ✅ Extremely lightweight (~50MB binary).
  ✅ Works on Raspberry Pi & ARM devices.
  ✅ Single binary, easy to deploy.
  Cons:
  ❌ Not suitable for large-scale production.
  ❌ Some Kubernetes features are stripped.
  Example:

```
curl -sfL https://get.k3s.io | sh -
```

6. OpenShift (Red Hat)

Primary Use: Enterprise Kubernetes with enhanced security and CI/CD.
Key Features:

- Includes built-in monitoring, logging, and developer tools.
- Uses **CRI-O** instead of Docker.

  Pros:
  ✅ Enterprise-grade security (SELinux, RBAC).
  ✅ Integrated CI/CD (OpenShift Pipelines).
  ✅ Good for large-scale deployments.
  Cons:
  ❌ Heavyweight (requires 4+ CPU cores).
  ❌ Proprietary features (not pure Kubernetes).
  Example:

```
oc cluster up  # (Legacy, now replaced by CodeReady Containers)
```

7. Docker Desktop (Built-in Kubernetes)

Primary Use: Local Kubernetes cluster for macOS/Windows developers.
Key Features:

- One-click Kubernetes cluster.
- Uses `containerd` runtime.

  Pros:
  ✅ Simplest setup for macOS/Windows.
  ✅ Good for Docker-based workflows.
  Cons:
  ❌ Limited configurability.
  ❌ Resource-heavy (uses a VM).
  Example:

- Enable Kubernetes in Docker Desktop settings.

Comparison Table

Tool	Best For	Multi-Node?	Production?	Ease of Use	Resource Usage
Kind	CI/CD, Fast testing	✓ Yes	✗ No	★★★★	Very Low
Minikube	Local development	✗ No	✗ No	★★★★★	Medium
Kubeadm	Production clusters	✓ Yes	✓ Yes	★★	High
MicroK8s	Edge/IoT, Dev	✓ (Limited)	✗ No	★★★★	Low
K3s	Edge, Lightweight	✓ Yes	⚠ Small-scale	★★★★★	Very Low
OpenShift	Enterprise Kubernetes	✓ Yes	✓ Yes	★★	Very High
Docker Desktop	macOS/Windows Dev	✗ No	✗ No	★★★★★	High

Which One Should You Use?

- **For Local Development:**
  - Use **Minikube** (beginner-friendly) or **Kind** (Docker-based, faster).
  - **Docker Desktop** if you already use Docker.
- **For CI/CD & Fast Testing:**
  - **Kind** (best for ephemeral clusters).
- **For Edge/IoT & Lightweight Clusters:**
  - **K3s** or **MicroK8s**.
- **For Production Clusters:**
  - **Kubeadm** (manual control) or **OpenShift** (enterprise features).

Final Thoughts

- **Kind** is great for fast, disposable clusters (CI/CD).
- **Minikube** is best for beginners on a local machine.
- **Kubeadm** is for admins who need full control.
- **K3s/MicroK8s** are perfect for edge computing.
- **OpenShift** is for enterprises needing extra tooling.

All-in-One Kubernetes Manifest (Pod + Deployment + Service + ConfigMap)

Description: This manifest combines a basic Pod, Deployment, Service, and ConfigMap for a web application (Nginx).

```yaml
---
# API Version defines the Kubernetes API group and version
apiVersion: v1
# ConfigMap stores non-sensitive configuration data
kind: ConfigMap
metadata:
  name: app-config
data:
  # Key-value pairs for configuration
  APP_COLOR: "blue"
  APP_MODE: "production"
---
# Deployment manages Pods (scaling, updates, rollbacks)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  # Number of identical Pod replicas to maintain
  replicas: 2
  # Label selector to match Pods controlled by this Deployment
  selector:
    matchLabels:
      app: nginx

  # Template for creating Pods
  template:
    metadata:
      # Labels applied to Pods (must match selector)
      labels:
        app: nginx

   spec:
      # Containers running inside the Pod
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            # Container port exposed internally
            - containerPort: 80
          # Environment variables from ConfigMap
          envFrom:
            - configMapRef:
                name: app-config
---
# Service exposes Pods to the network (internal or external)
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  # Selector matches Pods to route traffic to
  selector:
    app: nginx
  ports:
    - protocol: TCP
      # Port exposed by the Service
      port: 80
      # Target port on the Pods
      targetPort: 80
  # Service type (ClusterIP, NodePort, LoadBalancer)
  type: ClusterIP
```

Line-by-Line Explanation

1. ConfigMap Section

```yaml
apiVersion: v1          # Uses the core Kubernetes API
kind: ConfigMap         # Defines a ConfigMap resource
metadata:
  name: app-config      # Name of the ConfigMap
data:                   # Key-value configuration data
  APP_COLOR: "blue"     # Example config: App theme color
  APP_MODE: "production" # Example config: Runtime mode
```

Purpose: Stores environment variables or configuration files for Pods to consume.

2. Deployment Section

```yaml
apiVersion: apps/v1     # Deployment API group
kind: Deployment        # Manages Pod lifecycle
metadata:
  name: nginx-deployment  # Name of the Deployment
spec:
  replicas: 2           # Maintains 2 identical Pods
  selector:
    matchLabels:
      app: nginx        # Targets Pods with label `app: nginx`
  template:             # Pod template
    metadata:
      labels:
        app: nginx      # Labels for Pods (must match selector)
    spec:
      containers:
        - name: nginx   # Container name
          image: nginx:latest  # Docker image
          ports:
            - containerPort: 80  # Port exposed by the container
          envFrom:      # Imports all data from ConfigMap
            - configMapRef:
                name: app-config
```

Purpose: Ensures 2 Nginx Pods are running, injects ConfigMap data as env variables.

3. Service Section

```
apiVersion: v1          # Core API for Services
kind: Service           # Exposes Pods internally/externally
metadata:
  name: nginx-service   # Service name
spec:
  selector:
    app: nginx          # Routes traffic to Pods with `app: nginx`
  ports:
    - protocol: TCP     # TCP traffic
      port: 80          # Service listens on port 80
      targetPort: 80    # Forwards to Pod's port 80
  type: ClusterIP       # Makes Service internal (default)
```

Purpose: Provides a stable IP/DNS name for Pods. Other apps can access Nginx via nginx-service:80.

Key Takeaways

1. **ConfigMap** → Centralizes configuration.
2. **Deployment** → Manages Pods (scaling/updates).
3. **Service** → Enables network access to Pods.
4. **Labels** → Connects components (app: nginx).

How to Deploy?

```
kubectl apply -f all-in-one.yaml
```

Verify:

```
kubectl get pods,deployments,services,configmaps
```

---------- Kind Types in Kubernetes Manifest YAML

In a Kubernetes manifest file, the kind field specifies the type of Kubernetes object you are defining — such as a Pod, Service, Deployment, etc.

---Common kind Types by Category

1. Workload Resources

Kind	**Purpose**
-Pod	-Basic unit of deployment (single container or multiple)
-ReplicaSet	-Ensures a specified number of pod replicas are running
-Deployment	-Manages stateless app deployment and updates
-StatefulSet	-For stateful apps (stable network ID & storage)
-DaemonSet	-Ensures one pod per node (e.g., logging agents)
-Job	-Runs pods to completion (one-time tasks)
-CronJob	-Schedules jobs to run periodically (like cron)

2. Service & Networking Resources

Kind	**Purpose**
-Service	-Exposes a set of pods as a network service
-Ingress	-Manages external access to services (HTTP/HTTPS)
-NetworkPolicy	-Controls traffic flow between pods

3. Configuration & Secrets

Kind	**Purpose**
-ConfigMap	-Stores non-sensitive configuration data
-Secret	-Stores sensitive data like passwords, tokens
-PersistentVolume	-Represents a piece of storage in the cluster
-PersistentVolumeClaim	-Requests storage from a PersistentVolume

4. Cluster & Namespace Resources

Kind	**Purpose**
-Namespace	-Isolates group of resources in the cluster
-Node	-Represents a cluster worker machine
-LimitRange	-Sets default resource requests/limits
-ResourceQuota	-Limits total resources in a namespace

5. RBAC (Access Control)

Kind	**Purpose**
-Role	-Defines permissions within a namespace
-ClusterRole	-Defines permissions cluster-wide
-RoleBinding	-Assigns a Role to a user/group in a namespace
-ClusterRoleBinding	-Assigns ClusterRole to a user/group across cluster

6. Storage & Volume

Kind	**Purpose**
-StorageClass	-Defines storage types (like fast-ssd)
-VolumeAttachment	-Binds a volume to a node (CSI driver)

7. Custom Resource Support

Kind	**Purpose**
-CustomResourceDefinition (CRD)	-Lets you define custom resource types
-Your CRD (e.g., MyApp)	-Custom object kind created from CRD

----Example Manifest

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

----Summary

Category	**Common kind Types**
-Workloads	-Pod, Deployment, StatefulSet, DaemonSet
-Networking	-Service, Ingress, NetworkPolicy
-Config & Storage	-ConfigMap, Secret, PVC, PV
-RBAC	-Role, ClusterRole, RoleBinding
-Management	-Namespace, Node, LimitRange, Quota
-Customization	-CRD, Custom Resource

-------Kubernetes Kind Types in Manifest YAML Files

In Kubernetes, the kind field in a YAML manifest specifies the type of resource you're defining. Here's a comprehensive list of common kind types you'll encounter:

---Core Workload Resources

1. Pod - The smallest deployable unit in Kubernetes

2.Deployment - Manages ReplicaSets and provides declarative updates

3. ReplicaSet - Ensures a specified number of pod replicas are running

4. StatefulSet - Manages stateful applications with stable network identities

5. DaemonSet - Ensures all (or some) nodes run a copy of a pod

6. Job - Creates one or more pods to complete a task

7. CronJob - Runs Jobs on a time-based schedule

----Service & Networking Resources

8. Service - Exposes an application running on pods as a network service

9. Ingress - Manages external access to services

10. IngressClass - Defines a class of Ingress controller

-----Configuration & Storage Resources

11. ConfigMap - Stores non-confidential configuration data
12. Secret - Stores sensitive information
13. PersistentVolume (PV) - Storage resource in the cluster
14. PersistentVolumeClaim (PVC) - Request for storage
15. torageClass - Defines a class of storage

----Cluster Management Resources

16. Namespace - Logical partitioning of a cluster
17. Node - A worker machine in Kubernetes
18. CustomResourceDefinition (CRD) - Extends the Kubernetes API

----Security Resources

19. ServiceAccount - Provides an identity for processes in pods
20. Role - Defines permissions within a namespace
21. ClusterRole - Defines cluster-wide permissions
22. RoleBinding - Grants permissions to a Role
23. ClusterRoleBinding - Grants cluster-wide permissions

----Other Important Kinds

24. HorizontalPodAutoscaler (HPA) - Automatically scales workloads
25. PodDisruptionBudget - Limits disruptions during maintenance
26. NetworkPolicy - Specifies how pods communicate
27. LimitRange - Constraints resource limits per pod/container
28. ResourceQuota - Limits resource usage per namespace

----Example of a complete Deployment manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Feature	Pod	Deployment	StatefulSet	DaemonSet
Manages Pods?	No	Yes	Yes	Yes
Pod Identity	Random	Random	Stable	One per Node
Storage	Ephemeral	Ephemeral	Persistent	Depends
Rolling Updates	Manual	Supported	Ordered	Partial
Use Case	Basic	Stateless apps	Databases	Node-level agents

Feature	Service	Ingress	NetworkPolicy
Layer	L4 (TCP/UDP)	L7 (HTTP/HTTPS)	Network Security
Purpose	Stable access to Pods	Route external traffic to Services	Control Pod communication
Exposes to	Internal/External Clients	External users (via HTTP/HTTPS)	No exposure; restricts access
Type of Rule	Port-based routing	Path/host-based routing	Allow/Deny ingress/egress
Controller Needed	No	Yes (Ingress Controller)	No, but needs compatible CNI plugin

Feature	ConfigMap	Secret	PV (PersistentVolume)	PVC (PersistentVolumeClaim)
Stores	Config data	Sensitive data	Actual disk/storage	Request for storage
Data type	Plain text	Base64-encoded	Storage resource	Storage request
Used by	Developers	Developers	Admin or Dynamic provisioner	Developers
Security	Not secure	More secure (base64 + RBAC)	Depends on backend	Depends on bound PV
Mounted into Pod?	Yes	Yes	Indirectly via PVC	Yes

Feature	Role	ClusterRole	RoleBinding
Scope	Single namespace	All namespaces (or none)	One namespace
Targets	Namespaced resources	Cluster + namespaced	Binds Role/ClusterRole to subject
Used for	Local permissions	Global/admin permissions	Assigning access in namespace
Can bind to	Users/Groups/ServiceAccounts	Users/Groups/ServiceAccounts	Users/Groups/ServiceAccounts

Feature	Namespace	Node	LimitRange	ResourceQuota
Type	Logical grouping	Physical/Virtual machine	Per-container resource limit	Per-namespace resource cap
Scope	Cluster-wide object	Cluster-wide object	Namespace	Namespace
Purpose	Isolate resources/apps	Run workloads (pods)	Set default/min/max resource	Limit total usage per NS
Controls	Pods, services, etc.	Pod scheduling	CPU/memory per pod/container	Total CPU/memory/pods/etc.

Feature	CRD (CustomResourceDefinition)	Custom Resource
What is it?	Definition of a new resource	An instance of that resource
Purpose	Extends Kubernetes API	Stores custom config/data
Created by?	Cluster Admin	Developer/User
Required First?	Yes	Only after CRD is defined
Comparable to	A class or data model	An object created from that model

Feature	Pod	Deployment	ReplicaSet	DaemonSet	StatefulSet
Purpose	Smallest unit, runs 1+ containers	Manages stateless apps with rolling updates	Ensures desired Pod replicas are running	Runs **1 Pod per node** (cluster-wide)	Manages stateful apps with stable identity
Scaling	Single instance (no scaling)	Yes (via ReplicaSet)	Yes (maintains replica count)	Auto-scales with nodes	Yes (with ordered scaling)
Self-Healing	No (dies if Pod crashes)	Yes (recreates Pods via ReplicaSet)	Yes (maintains replica count)	Yes (recreates Pods on node failures)	Yes (maintains Pod identity)
Updates	Manual replacement	Supports rolling updates & rollbacks	Manual updates only	Rolling updates supported	Ordered, rolling updates
Storage	Ephemeral (unless manually configured)	Typically stateless	Typically stateless	Can use host volumes	**Persistent storage** (stable PVCs)
Networking	Ephemeral IP & hostname	Dynamic IP & hostname	Dynamic IP & hostname	Dynamic IP & hostname	**Stable hostname** & network identity
Ordering	N/A	No ordering	No ordering	No ordering	**Ordered** deployment/scaling
Node Assignment	Can run anywhere	Runs on any node	Runs on any node	**Runs on every node** (or subset)	Runs on any node
Use Cases	Debugging, testing	Web servers, stateless microservices	Rarely used directly (managed by Deployments)	Log collectors, monitoring agents	Databases (MySQL, MongoDB)
Example Command	kubectl run nginx --image=nginx	kubectl create deploy nginx --image=nginx	kubectl create rs nginx --image=nginx --replicas=3	kubectl apply -f daemonset.yaml	kubectl apply -f statefulset.yaml

Key Differences Summary:

1. **Pod** → Single instance, no management.
2. **Deployment** → Manages ReplicaSets for stateless apps.
3. **ReplicaSet** → Ensures Pod replicas run (used by Deployments).
4. **DaemonSet** → One Pod per node (for system services).
5. **StatefulSet** → For stateful apps with stable identity & storage.

When to Use What?

- Use **Pods** for testing/debugging.
- Use **Deployments** for stateless apps (common default).
- Use **DaemonSets** for node-level services (e.g., logging).
- Use **StatefulSets** for databases/clustered apps.

Analogy:

- **Pod** = A single process.
- **Deployment** = A manager ensuring enough workers (Pods) are always up.
- **DaemonSet** = A mandatory service on every machine (like antivirus).
- **StatefulSet** = A database server with persistent data.

Feature	**Service**	**Ingress**	**NetworkPolicy**
Purpose	Exposes Pods internally or externally via a stable IP/DNS	Manages external HTTP/HTTPS access to Services (Layer 7 routing)	Controls traffic flow between Pods (firewall-like rules)
Layer (OSI Model)	Layer 4 (TCP/UDP)	Layer 7 (HTTP/HTTPS)	Layer 3/4 (IP, Port, Protocol)
Traffic Direction	Internal (ClusterIP) or External (NodePort/LoadBalancer)	External (via Ingress Controller)	Both ingress (inbound) & egress (outbound)
Load Balancing	Yes (round-robin across Pods)	Yes (via Ingress Controller)	No (only traffic filtering)
Use Cases	Connecting apps within a cluster, exposing apps via LoadBalancer	Host-based routing, TLS termination, path-based routing	Security, isolating Pods, restricting cross-namespace traffic
Example Types	ClusterIP, NodePort, LoadBalancer	Nginx, Traefik, AWS ALB	allow-from-namespace, deny-all
Dependencies	Works standalone (kube-proxy)	Requires an **Ingress Controller** (e.g., Nginx, Istio)	Requires a **CNI plugin** (e.g., Calico, Cilium)
Example Config	yaml kind: Service spec: type: ClusterIP ports: - port: 80 	yaml kind: Ingress spec: rules: - host: app.com http: paths: - path: / backend: serviceName: app 	yaml kind: NetworkPolicy spec: podSelector: matchLabels: {role: db} ingress: - from: - podSelector: matchLabels: {role: api}

Key Differences Summary:

1. **Service** → Internal/external connectivity (IP/port-based).
2. **Ingress** → External HTTP routing (needs an Ingress Controller).
3. **NetworkPolicy** → Security/firewall for Pod-to-Pod traffic (needs CNI support).

When to Use What?

- Use **Service** for basic load balancing within the cluster.
- Use **Ingress** for advanced HTTP routing (e.g., multiple domains/paths).
- Use **NetworkPolicy** to enforce security (e.g., "only frontend can talk to backend").

Feature	**ConfigMap**	**Secret**	**PersistentVolume (PV)**	**PersistentVolumeClaim (PVC)**
Purpose	Stores **non-sensitive** config data (e.g., env vars, files)	Stores **sensitive** data (e.g., passwords, tokens)	Represents **physical storage** (disk, NFS, etc.) in the cluster	Requests storage from a **PV** (like a "ticket" for storage)
Data Type	Plaintext (key-value pairs or files)	Base64-encoded (or Kubernetes-managed encryption)	Raw storage (block/file/object)	Abstracted storage request
Use Cases	Configuration files, environment variables	Database credentials, TLS certificates	Long-term storage for databases, logs	Dynamic provisioning for Pods
Accessibility	Mounted as volumes or env vars in Pods	Mounted as volumes or env vars (hidden in kubectl describe)	Bound to a PVC, then used by Pods	Pods reference PVCs in volumes
Lifecycle	Tied to namespace, deleted manually	Tied to namespace, can be encrypted	Cluster-wide resource, manually or dynamically provisioned	Namespace-scoped, binds to PV
Example YAML	yaml kind: ConfigMap data: app.conf: | key=value 	yaml kind: Secret type: Opaque data: password: <base64>	yaml kind: PersistentVolume spec: capacity: {storage: 10Gi} accessModes: [ReadWriteOnce] 	yaml kind: PersistentVolumeClaim spec: resources: requests: {storage: 5Gi}
Security	Not encrypted (avoid secrets!)	Encrypted at rest (optional: with **KMS**)	Depends on backend (e.g., encrypted EBS)	Inherits PV security
Dynamic Provisioning	No (static data)	No (static data)	Yes (via **StorageClass**)	Yes (auto-binds to PV)

Key Differences Summary:

1. **ConfigMap** → Non-sensitive configs (e.g., app.properties).
2. **Secret** → Sensitive data (e.g., passwords, TLS certs).
3. **PV** → Actual storage (like a "disk").
4. **PVC** → Pod’s request for storage (like a "claim ticket").

When to Use What?

- Use **ConfigMap** for environment variables or config files.
- Use **Secret** for credentials/certs (always over ConfigMap for secrets!).
- Use **PV/PVC** for persistent data (e.g., databases).

Analogy:

- **ConfigMap** = Notice board (public info).
- **Secret** = Locked safe (private info).
- **PV** = Hard drive.
- **PVC** = Request to use the hard drive.

Feature	Role	RoleBinding	ClusterRole	ClusterRoleBinding
Scope	**Namespace-scoped** (applies to a single namespace)	**Namespace-scoped** (grants permissions within a namespace)	**Cluster-scoped** (applies to the entire cluster)	**Cluster-scoped** (grants permissions across all namespaces)
Purpose	Defines permissions (rules) for resources **within a namespace** (e.g., Pods, Services)	Binds a **Role** or **ClusterRole** to users/groups/ServiceAccounts **in a namespace**	Defines cluster-wide permissions (e.g., nodes, PVs) or reusable rules across namespaces	Binds a **ClusterRole** to users/groups/ServiceAccounts **cluster-wide**
Example Use Cases	Grant read-only access to Pods in dev namespace	Allow a user to create Pods in dev namespace	Grant admin access to cluster-level resources (e.g., Nodes, StorageClasses)	Give a ServiceAccount full access to all namespaces
YAML Example	yaml kind: Role rules: - apiGroups: [""] resources: ["pods"] verbs: ["get", "list"] 	yaml kind: RoleBinding subjects: - kind: User name: alice roleRef: kind: Role name: pod-reader 	yaml kind: ClusterRole rules: - apiGroups: [""] resources: ["nodes"] verbs: ["get", "list"] 	yaml kind: ClusterRoleBinding subjects: - kind: Group name: admins roleRef: kind: ClusterRole name: cluster-admin
Binding Flexibility	Can only be referenced by **RoleBinding** in the same namespace	Can reference a **Role** (namespace-scoped) or **ClusterRole** (but applies only to the bound namespace)	Can be referenced by **RoleBinding** (namespace-scoped) or **ClusterRoleBinding** (cluster-scoped)	Can only reference a **ClusterRole**
Common Roles	pod-reader, namespace-admin	Binds users to predefined Roles	cluster-admin, view, edit	Binds groups/users to ClusterRoles

Key Differences Summary:

1. **Role** → Namespace-specific permissions.
2. **RoleBinding** → Grants a **Role/ClusterRole** to subjects **in a namespace**.
3. **ClusterRole** → Cluster-wide or reusable permissions.
4. **ClusterRoleBinding** → Grants a **ClusterRole** to subjects **across all namespaces**.

When to Use What?

- Use **Role + RoleBinding** for team-specific access (e.g., dev namespace).
- Use **ClusterRole + ClusterRoleBinding** for cluster admins (e.g., kube-system access).
- Use **ClusterRole + RoleBinding** to define reusable roles (e.g., pod-reader in multiple namespaces).

Analogy:

- **Role** = Department-specific permissions (e.g., "HR folder access").
- **RoleBinding** = Assigning HR access to an employee.
- **ClusterRole** = Company-wide permissions (e.g., "CEO access").
- **ClusterRoleBinding** = Granting CEO access to an executive.

Feature	Namespace	Node	LimitRange	ResourceQuota
Purpose	Logical isolation of resources (virtual cluster inside a physical cluster)	A worker machine (physical/virtual) that runs Pods	Enforces **min/max resource limits** per Pod/Container in a namespace	Restricts **total resource usage** per namespace
Scope	Logical grouping (e.g., dev, prod)	Physical/virtual machine (part of the cluster)	Applies to **Pods/Containers** within a namespace	Applies to **all resources** in a namespace
Managed By	Kubernetes API (kubectl create ns)	Kubelet, Container Runtime (Docker/containerd)	Kubernetes API (enforced by kube-apiserver)	Kubernetes API (enforced by kube-apiserver)
Key Functions	- Resource segregation	- Runs Pods	- Sets default CPU/memory requests/limits	- Limits total CPU/memory/storage
	- Access control (RBAC)	- Provides CPU/memory/storage	- Prevents resource hogging	- Limits object counts (Pods, PVCs)
Example Use Case	Separate dev and prod environments	A VM running Pods in a cluster	Ensure no Pod requests >2 CPU cores	Prevent a namespace from using >16 CPU cores total
Example YAML	yaml kind: Namespace metadata: name: dev 	(Nodes are usually managed by cloud providers or admins)	yaml kind: LimitRange spec: limits: - max: {cpu: "2"} 	yaml kind: ResourceQuota spec: hard: cpu: "16"
Dependencies	Works standalone	Requires kubelet, container runtime	Requires a namespace	Requires a namespace
Cluster Impact	Affects resource organization	Affects cluster capacity/scaling	Affects Pod scheduling/resource fairness	Affects namespace-level resource allocation

Key Differences Summary:

1. **Namespace** → Virtual cluster for isolation (logical boundary).
2. **Node** → Worker machine that runs Pods (physical boundary).
3. **LimitRange** → Guards against **per-Pod** resource abuse.
4. **ResourceQuota** → Guards against **namespace-wide** resource exhaustion.

When to Use What?

- Use **Namespaces** for multi-tenancy (e.g., teams, environments).
- Use **Nodes** to scale cluster capacity (add/remove workers).
- Use **LimitRange** to enforce Pod/Container resource constraints.
- Use **ResourceQuota** to cap total namespace resources.

Analogy:

- **Namespace** = Apartment building (separate units).
- **Node** = The land the building sits on.
- **LimitRange** = Rules for each apartment (e.g., "no loud music after 10 PM").
- **ResourceQuota** = Building-wide limits (e.g., "total water usage ≤1000L/day").

------ Kind Types in Kubernetes Manifest YAML

1. Workloads (Pods, Deployments, Jobs, etc.)

Kind	**Description**
-Pod	Smallest deployable unit (1+ containers)
-Deployment	Manages stateless app replicas (rolling updates)
-ReplicaSet	Ensures a stable set of Pod replicas (used by Deployments)
-StatefulSet	Manages stateful apps (stable network IDs, persistent storage)
-DaemonSet	Ensures all/nodes run a copy of a Pod (e.g., logging agents)
-Job	Runs a task to completion (batch jobs)
-CronJob	Runs Jobs on a schedule
-ReplicationController	-Legacy (use Deployment/ReplicaSet instead)

2. Service & Networking

Kind	**Description**
-Service	-Exposes Pods as a network service (ClusterIP,NodePort,LoadBalancer)
-Ingress	Manages external HTTP/S routing to Services
-IngressClass	Defines Ingress controller types
-NetworkPolicy	Controls Pod-to-Pod traffic (firewall rules)
-EndpointSlice	Tracks Service endpoints (scalable alternative to Endpoints)

3. Storage

Kind	**Description**
-PersistentVolume (PV)	Cluster-wide storage resource
-PersistentVolumeClaim (PVC)	-User request for storage (binds to PV)
-StorageClass	Defines dynamic PV provisioning (e.g., SSD, HDD)
-VolumeAttachment	Links a PV to a Node

4. Configuration & Secrets

Kind	**Description**
-ConfigMap	Stores non-sensitive configs (env vars, files)
-Secret	Stores sensitive data (passwords, tokens)
-ResourceQuota	Limits resource usage per namespace
-LimitRange	Sets default min/max resources for Pods

5. Cluster Management

Kind	**Description**
-Namespace	Isolates resources (logical clusters)
-Node	Worker machine in the cluster
-CustomResourceDefinition (CRD)	-Extends Kubernetes API with custom resources
-RuntimeClass	-Selects container runtime (e.g., containerd, gVisor)

6. Autoscaling

Kind	**Description**
-HorizontalPodAutoscaler (HPA)	-Scales Pods based on CPU/memory/custom metrics
-VerticalPodAutoscaler (VPA)	Adjusts Pod CPU/memory requests/limits
-ClusterAutoscaler	Scales nodes (cloud providers only)

7. Security

Kind	**Description**
-ServiceAccount	Identity for Pods to access the API
-Role / ClusterRole	-Defines permissions (namespaced/cluster-wide)
-RoleBinding / ClusterRoleBinding	Grants roles to users/ServiceAccounts
-PodSecurityPolicy (PSP)	-Deprecated (use Pod Security Admission instead)

8. Monitoring & Metrics

Kind	**Description**
-PodDisruptionBudget (PDB)	-Ensures min available Pods during disruptions
-MetricsServer	Aggregates cluster resource metrics (used by HPA)

9. Batch & Execution

Kind	**Description**
-CronJob	Scheduled Jobs
-Job	One-time task execution

10. Custom Resources (CRDs)

Kind	**Description**
*-(Custom)	User-defined resources (e.g., Knative Services, Argo Workflows)

11. Others (Less Common)

Kind	**Description**
-Lease	Used for node heartbeats (internal)
-PriorityClass	Defines Pod scheduling priority
-VolumeSnapshot	Point-in-time copy of a volume

------Installing Kubernetes Dashboard on Kind (Kubernetes in Docker)

1) Install Docker, Kind CLI, Kubectl.
2) Install Kubernetes Dashboard.
3) Create Admin User & Bind Cluster Role (Service Account).
4) Create Access Token for Admin-user.
5) Access Dashboard –
   a) Port-Forward (Recommended) or kubectl proxy.
   b) Expose via NodePort (Edit Service) (For External Access).

------------pod-defination.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
      app: myapp
      type: fornt-end
spec:
  containers:
    - name: nginx-container
      image: nginx
```

```
# kubectl create -f pod-defination.yml
# kubectl get pod
# kubectl describe pod nginx
# kubectl get pods -o wide
```

------------repilcaset.yml

```
apiVersion: apps/v1
kind: Replicaset
metadata:
   name: forntend
   labels:
     app: my-app
     tier: forntend
spec:
   replicas: 3
   selector
     matchLabels:
       tier: forntend
   template:
     meatadata:
       labels:
         tier: forntend
      spec:
         containers:
           - name: nginx container
             image: nginx
```

```
# kubectl apply -f repilcaset.yml
# kubectl get rs
# kubectl describe name
```

-------------deployment-definition.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
   name: myapp-deployment
   labels:
     app: my-app
     tier: forntend
spec:
   template:
     metadata:
       name: myapp-pod
       labels:
          app: myapp
          type: fornt-end
      spec:
        containers:
        - name: nginx-container
          image: nginx:latest
replicas: 3
selector:
  matchLabels:
    type: fornt-end
```

```
kubectl apply -f deployment-defination.yml
kubectl get deployments
kubectl rollout status deployment/myapp-deployment
kubectl rollout history deployments
```

--------------service-defination.yml

```
apiVersion: v1
kind: Service
metadata:
   name: myapp-service
spec:
    type: NodePort
    ports:
      - targetPort: 80
        port: 80
        nodePort: 30009
```

----------service-defination-cluster.yml

```
apiVersion: v1
kind: Service
metadata:
   name: back-end
spec:
    type: ClusterIP
    ports:
      - targetPort: 80
        port: 80
```

------------Secret.yml

```
apiVersion: v1
Kind: Secret
metadata:
  name: wordpress-admin-password
data:
  key: Clasm$up=0lknsdfkn2878723

#kubectl apply -f Secret.yml
```

-----------volume.yml

```
apiVersion: v1
Kind: PersistentVolume
metadata: pv001
  spec:
    capacity:
      storage: 20GB
    volumeMode: Filesystem
    accessModes:
      - ReadWriteOnce
```

----------ClusterIP & NodePort.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  type: NodePort
  selector:
    app: web

  ports:
    - protocol: TCP
      port: 80                    # ClusterIP port
      targetPort: 80              # Pod port
      nodePort: 31000             # Optional (default: random)
```

----------ReplicationController.yml

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-rc
spec:
  replicas: 3                 # Maintain 3 identical pods
  selector:
    app: nginx               # Must match pod template labels
  template:                  # Pod template
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14
        ports:
        - containerPort: 80
```

------------ReplicaSet.yml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: modern-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx

OR    matchExpressions:
    - {key: app, operator: In, values: [nginx,web]}
    - {key: env, operator: NotIn, values: [dev]}

  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.19
```

------------Deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3                      # Number of pod replicas
  revisionHistoryLimit: 3          # Keeps last 3 revisions
  selector:                          # Pod selection criteria
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate            # Default (others: Recreate)
    rollingUpdate:
      maxSurge: 1                  # Pods created above desired count
      maxUnavailable: 0           # Pods that can be unavailable
  template:                        # Pod template
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.19
        ports:
        - containerPort: 80
```

1. Create a Deployment

```
kubectl create deployment nginx --image=nginx:1.19
# OR
kubectl apply -f deployment.yaml
```

2. Get Deployments

```
kubectl get deployments
kubectl get deploy  # Short form
```

3. View Deployment Details

```
kubectl describe deployment nginx-deployment
```

4. Scale a Deployment

```
kubectl scale deployment nginx-deployment --replicas=5
```

5. Update a Deployment

```
# Method 1: Update image
kubectl set image deployment/nginx-deployment nginx=nginx:1.20

# Method 2: Edit live config
kubectl edit deployment nginx-deployment
```

6. Rollout Management

```
# Check rollout status
kubectl rollout status deployment/nginx-deployment

# Pause a rollout
kubectl rollout pause deployment/nginx-deployment

# Resume a rollout
kubectl rollout resume deployment/nginx-deployment
```

7. Rollback a Deployment

```
# View rollout history
kubectl rollout history deployment/nginx-deployment

# Rollback to previous version
kubectl rollout undo deployment/nginx-deployment

# Rollback to specific revision
kubectl rollout undo deployment/nginx-deployment --to-revision=2
```

8. Delete a Deployment

```
kubectl delete deployment nginx-deployment

# Delete without waiting
kubectl delete deployment nginx-deployment –now
```

--------Rollout trigger in kubernetes

In Kubernetes, a rollout is triggered whenever there's a change to the Pod template specification (spec.template) in your Deployment YAML. Here's exactly what triggers rollouts and where to make changes:

-----What Triggers a Rollout?

A rollout is initiated when any of these fields in spec.template are modified:

1. Container Image Changes

```
spec:
  template:
    spec:
      containers:
      - name: nginx
        image: nginx:1.23          # ← Changing this triggers rollout
```

2. Environment Variables

```
env:
- name: APP_ENV
  value: "production"              # ← Changing this triggers rollout
```

3. Labels/Annotations (if used in selectors)

```
metadata:
  labels:
    app: frontend       # ← Changing this may trigger rollout
```

4. Resource Limits/Requests

```
resources:
  limits:
    cpu: "1"            # ← Changing this triggers rollout
```

5. Volume/ConfigMap/Secret Changes

```
volumes:
- name: config
  configMap:
    name: app-config               # ← Changing this triggers rollout
```

------What Does NOT Trigger a Rollout?

These changes won't trigger a new rollout:
1. Scaling replicas (replicas: 5)
2. Modifying rollout strategy parameters (maxSurge, maxUnavailable)
3. Changing resource limits not in the Pod template

---------How to Trigger a Rollout (3 Methods)

1. Edit the YAML File

```
kubectl edit deployment/my-deployment
# Make changes to spec.template → Save triggers rollout
```

2. Apply Updated YAML

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

3. Imperative Command

```
kubectl set image deployment/my-deployment nginx=nginx:1.24
```

--------Rollout Process Flow

1. New ReplicaSet Created

Kubernetes creates a new ReplicaSet with the updated template.

2. Scaling

New pods are created (controlled by maxSurge)
Old pods are terminated (controlled by maxUnavailable)

3. Completion

When all new pods are Ready, old ReplicaSet is scaled to 0.

-------Key Fields That Control Rollouts

```
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%              # How many extra pods can be created
      maxUnavailable: 25%        # How many pods can be unavailable
  minReadySeconds: 10            # Wait time before considering pod "ready"
  revisionHistoryLimit: 3        # How many old ReplicaSets to keep
```

----Verification Commands

```
# Check rollout status                        kubectl rollout status deployment/my-deployment
# View rollout history                        kubectl rollout history deployment/my-deployment
# See ReplicaSets (shows old and new versions)          kubectl get rs
```

-------Namespace

In Kubernetes, a namespace is a virtual cluster or a logical division within a physical Kubernetes cluster. It helps organize and isolate resources among multiple users, teams, or projects while sharing the same underlying infrastructure.

----Key Features of Namespaces:

1. Resource Isolation
-Namespaces provide a scope for resource names (e.g., Pods, Services, Deployments), allowing the same resource name to be used in different namespaces without conflict.
-Resources in one namespace are isolated from those in another unless explicitly allowed (e.g., via network policies).

2. Access Control (RBAC)
-Role-Based Access Control (RBAC) can be applied at the namespace level, restricting users or service accounts to specific namespaces.

3. Resource Quotas
-Administrators can enforce ResourceQuotas to limit CPU, memory, and storage usage per namespace.

4. Logical Grouping
-Namespaces help organize resources (e.g., dev, prod, monitoring) for better management.

-----Default Namespaces in Kubernetes

When you create a Kubernetes cluster, some default namespaces are automatically created:

1. default → Where resources are created if no namespace is specified.
2. kube-system → For Kubernetes system components (e.g., kube-proxy, CoreDNS).
3. kube-public → Stores publicly accessible resources (rarely used).
4. kube-node-lease → For node lease objects (used for node heartbeats).

----Common Use Cases

--Multi-Tenancy → Separate environments (e.g., dev, staging, prod).
--Team Isolation → Different teams (e.g., team-a, team-b) can work independently.
--Specialized Workloads → Dedicated namespaces for monitoring, logging, or CI/CD.

-------Working with Namespaces

1. List Namespaces

```
kubectl get namespaces
# or
kubectl get ns
```

2. Create a Namespace

```
kubectl create namespace my-namespace
```

Or via a YAML file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
```

Apply it:

```
# kubectl apply -f my-namespace.yaml
```

3. Set a Default Namespace

To avoid specifying --namespace every time:

# kubectl config set-context --current --namespace=my-namespace

4. Deploy a Resource in a Namespace

```
# kubectl apply -f my-pod.yaml --namespace=my-namespace
```

Or specify in the YAML:

```
metadata:
  name: my-pod
  namespace: my-namespace
```

5. Delete a Namespace

```
# kubectl delete namespace my-namespace
```

--------Resource Quotas in Kubernetes

A ResourceQuota in Kubernetes is a mechanism to limit the total resource consumption within a namespace. It ensures that no single team or application can monopolize cluster resources, promoting fair usage and cost control.

---Key Features of ResourceQuotas

1. Limit Compute Resources (CPU, Memory)
2. Limit Storage Resources (PersistentVolumeClaims, requests.storage)
3. Limit Object Counts (Pods, Deployments, Services, etc.)
4. Prevent Resource Exhaustion (Avoid "Noisy Neighbor" problems)
5. Enforce Fair Sharing in Multi-Tenant Clusters

------Types of Resource Quotas

1. Compute Resource Quotas

Limits CPU and memory usage for:
-Requests (requests.cpu, requests.memory)
-Limits (limits.cpu, limits.memory)

2. Storage Resource Quotas

Limits storage requests:
-PersistentVolumeClaims (persistentvolumeclaims)
-Total Storage Requests (requests.storage)

3. Object Count Quotas

Limits the number of Kubernetes objects:
-Pods (pods)
-Services (services)
-Deployments (deployments.apps)
-Secrets (secrets)
-ConfigMaps (configmaps)

-------Example ResourceQuota YAML

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-a-quota
  namespace: team-a
spec:
  hard:
    # Compute Resources
    requests.cpu: "10"
    requests.memory: 20Gi
    limits.cpu: "20"
    limits.memory: 40Gi

    # Storage Resources
    requests.storage: 100Gi
    persistentvolumeclaims: "5"

    # Object Counts
    pods: "50"
    services: "10"
    secrets: "20"
    configmaps: "20"
```

---------LimitRange in Kubernetes

A LimitRange is a Kubernetes resource that enforces minimum, maximum, and default resource constraints (CPU, memory, storage) for Pods, Containers, and PersistentVolumeClaims within a namespace. It works alongside ResourceQuotas to ensure efficient and fair resource usage.

----Key Features of LimitRange

1. Sets Default CPU/Memory Requests & Limits → If a Pod doesn't specify resources, LimitRange provides defaults.

2. Enforces Min/Max Constraints → Prevents users from requesting too little or too much.

3. Applies to:

Containers (CPU, memory)
Pods (Total CPU/memory across all containers)
PersistentVolumeClaims (PVCs) (Storage requests)

--------Example LimitRange YAML

```
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
  namespace: team-a
spec:
  limits:
    - type: Container
      default:                       # Default limits if not specified
        cpu: "500m"
        memory: "512Mi"
      defaultRequest:                # Default requests if not specified
        cpu: "100m"
        memory: "256Mi"
      min:                           # Minimum allowed
        cpu: "50m"
        memory: "64Mi"
      max:                           # Maximum allowed
        cpu: "2"
        memory: "4Gi"
    - type: Pod
      max:                           # Total limits for all containers in a Pod
        cpu: "4"
        memory: "8Gi"
    - type: PersistentVolumeClaim
      min:                           # Min storage request
        storage: "1Gi"
      max:                           # Max storage request
        storage: "10Gi"
```

1. Create the LimitRange # kubectl apply -f limit-range.yaml -n team-a

2. Check Enforced Limits # kubectl describe limitrange app-limits -n team-a

---------ConfigMap

A ConfigMap in Kubernetes is used to store configuration data (key-value pairs) separately from application code. It allows you to inject dynamic configuration into your pods without changing the container image.

----Use Cases

1. Store application settings, environment variables
2. Externalize config files (YAML, INI, properties, etc.)
3. Share config across multiple pods
4. Keep secrets separate (use Secrets for sensitive data)

----Types of Data You Can Store

1. Simple key-value pairs
2. Entire files or multi-line text
3. Command-line arguments or environment configs

-----◆ 1. Create a ConfigMap
◆ a. From YAML

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  namespace: default
data:
  APP_ENV: "production"
  LOG_LEVEL: "info"
```

◆ b. From command line

```
kubectl create configmap app-config \
  --from-literal=APP_ENV=production \
  --from-literal=LOG_LEVEL=info
```

◆ c. From a file

```
 # kubectl create configmap app-config --from-file=config.properties
```

-----◆ 2. Using ConfigMap in a Pod

a. As Environment Variables

```
envFrom:
  - configMapRef:
      name: app-config
```

b. As Specific Env Vars

```
env:
  - name: APP_ENV
    valueFrom:
      configMapKeyRef:
        name: app-config
        key: APP_ENV
```

c. As a Mounted Volume

```
volumes:
  - name: config-volume
    configMap:
      name: app-config
containers:
  - name: app
    volumeMounts:
      - name: config-volume
        mountPath: /etc/config
```

------Example: Pod with ConfigMap as Env

```
apiVersion: v1
kind: Pod
metadata:
  name: config-demo
spec:
  containers:
    - name: myapp
      image: nginx
      envFrom:
        - configMapRef:
            name: app-config
```

--View and Debug

```
 # kubectl get configmap
 # kubectl describe configmap app-config
 # kubectl get pod config-demo -o yaml
```

----ConfigMap Limitations

Limitation	**Value**
1. Max size per ConfigMap	1 MB
2. Not for sensitive data	Use Secret instead
3. Static after creation	Pods must be restarted to pick up changes (unless re-mounted)

----Summary

Feature	**Description**
1. Purpose	Store non-sensitive config outside container
2. Data format	Key-value pairs, files, environment settings
3. Usage methods	Env vars, volume mounts, command args
4. Best for	Dynamic app config, decoupled from code/image

-------Secrets in Kubernetes

A Secret in Kubernetes is an object that stores sensitive data (like passwords, API keys, TLS certificates) in an encrypted form. It provides a more secure way to manage confidential information compared to ConfigMaps.

--Key Features of Secrets

1. Secure Storage
-Secrets are base64-encoded (not encrypted by default, but can be encrypted at rest in etcd).
-Kubernetes 1.13+ supports encryption at rest for Secrets.

2. Types of Secrets
-Generic (Opaque) – Default type for arbitrary key-value pairs (e.g., DB_PASSWORD).
-TLS (kubernetes.io/tls) – Stores TLS certificates (tls.crt, tls.key).
-Docker Registry (kubernetes.io/dockerconfigjson) – Stores credentials for pulling images from private registries.

3. Namespace-Scoped
-Secrets exist within a namespace (like ConfigMaps).

4. Immutable Option (Kubernetes 1.21+)
-Prevents accidental modifications.

-------When to Use Secrets?
✅ Database passwords
✅ API tokens
✅ TLS certificates
✅ Private Docker registry credentials
❌ Highly sensitive data → Consider external secret managers (Vault, AWS Secrets Manager).

------Creating a Secret

1. From Literal Values (CLI)

```
 # kubectl create secret generic db-secret \
  --from-literal=DB_USER=admin \
  --from-literal=DB_PASSWORD='S3cr3t!'
```

2. From a File

```
 # kubectl create secret generic tls-secret \
  --from-file=tls.crt=./cert.pem \
  --from-file=tls.key=./key.pem
```

3. From YAML (Base64-encoded)

-First, encode the data:

```
echo -n "admin" | base64        # Yields "YWRtaW4="
echo -n "S3cr3t!" | base64      # Yields "UzNjcjN0IQ=="
```

-Then define the Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque                          # Default type
data:
  DB_USER: YWRtaW4=                   # base64-encoded "admin"
  DB_PASSWORD: UzNjcjN0IQ==           # base64-encoded "S3cr3t!"
```

Apply it:

```
 # kubectl apply -f secret.yaml
```

-----Using Secrets in Pods

1. As Environment Variables

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: app
      image: my-app
      env:
        - name: DB_USER
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: DB_USER
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: DB_PASSWORD
```

2. As a Volume (Mounted Files)

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: app
      image: nginx
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secrets
          readOnly: true
  volumes:
    - name: secret-volume
      secret:
        secretName: db-secret
```

-------Managing Secrets

--List Secrets

```
# kubectl get secrets -n my-namespace
```

--View Secret Details

```
# kubectl describe secret db-secret -n my-namespace
```

--Decode a Secret

```
# kubectl get secret db-secret -o jsonpath='{.data.DB_PASSWORD}' | base64 --decode
```

--Make a Secret Immutable

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
immutable: true           # Prevents changes
data:
  TOKEN: <base64-encoded-data>
```

-----Best Practices

1. Avoid Hardcoding in YAML → Use kubectl create secret or tools like Sealed Secrets.
2. Limit Access → Use RBAC to restrict who can read Secrets.
3. Rotate Secrets → Change passwords/keys periodically (no built-in rotation in Kubernetes).
4. Prefer External Secret Managers → For highly sensitive data, use HashiCorp Vault or AWS Secrets Manager.
5. Use TLS Secrets for HTTPS → Automatically manages certs with Ingress controllers.

----Secrets vs. ConfigMaps

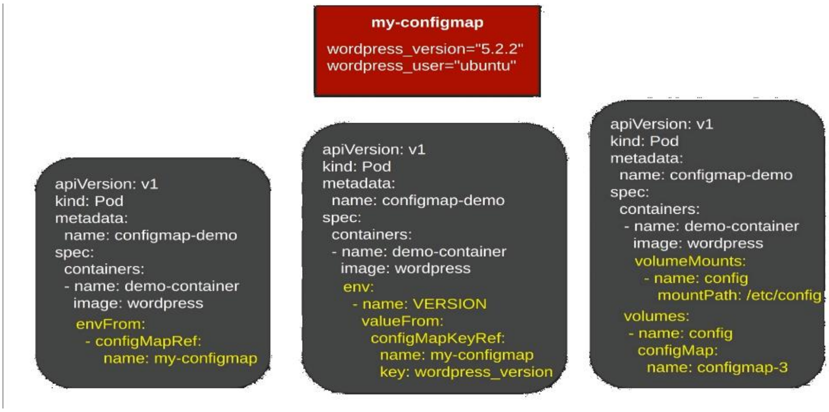
Feature	**Secret**	**ConfigMap**
1. Data Type	Sensitive (base64-encoded)	Non-sensitive (plain text)
2. Use Case	Passwords, tokens, TLS certs	App configs, environment vars
3. Security	Encrypted in etcd (optional)	Stored as plain text

To create a ConfigMap from a literal value, use the following command:

```
# kubectl create configmap configmap-1 --from-literal=name=first-configmap
# kubectl create configmap configmap-2 --from-literal=name=second-configmap --from-literal=color=blue
```

To create a ConfigMap from a file, use the following command:

```
# kubectl create configmap configmap-3 --from-file=data-file
```



```
my-configmap
wordpress_version="5.2.2"
wordpress_user="ubuntu"
```

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo
spec:
  containers:
  - name: demo-container
    image: wordpress
    envFrom:
      - configMapRef:
          name: my-configmap
```

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo
spec:
  containers:
  - name: demo-container
    image: wordpress
    env:
      - name: VERSION
        valueFrom:
          configMapKeyRef:
            name: my-configmap
            key: wordpress_version
```

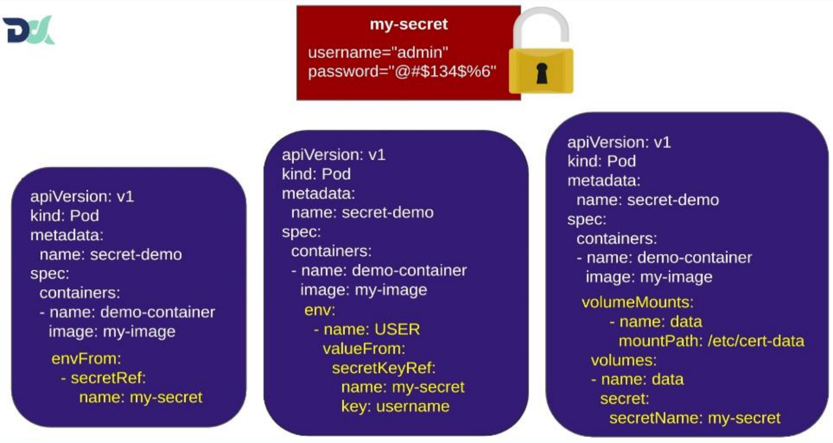
```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo
spec:
  containers:
  - name: demo-container
    image: wordpress
    volumeMounts:
      - name: config
        mountPath: /etc/config
  volumes:
   - name: config
     configMap:
       name: configmap-3
```

Generic Secret-- # kubectl create secret generic db-secret --from-literal=username=dbuser --from-literal=password=Y4nys7f11

Docker-registry Secret--# kubectl create secret docker-registry docker-secret --docker-email=example@gmail.com --docker-username=dev --docker-password=pass1234 --docker-server=my-registry.example:5000

TLS Secret-- # kubectl create secret tls my-tls-secret --cert=/root/data/serverca.crt --key=/root/data/servercakey.pem

```
# kubectl apply -f <file-name.yaml>  # kubectl get pods  #  kubectl get secret     # kubectl describe secret <secret-name>     # kubectl exec -it <pod-name> -- printenv  # kubectl exec -it <pod-name> -- bash   # kubectl get pod <pod-name> -o yaml     # echo "<data>" | base64 –d
```



```
my-secret
username="admin"
password="@#$134$%6"
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-demo
spec:
  containers:
  - name: demo-container
    image: my-image
    envFrom:
      - secretRef:
          name: my-secret
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-demo
spec:
  containers:
  - name: demo-container
    image: my-image
    env:
      - name: USER
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: username
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-demo
spec:
  containers:
  - name: demo-container
    image: my-image
    volumeMounts:
        - name: data
          mountPath: /etc/cert-data
    volumes:
    - name: data
      secret:
        secretName: my-secret
```

------Volume in Kubernetes

In Kubernetes, a volume is a directory that is accessible to containers in a pod, used to store and share data. Unlike container storage (which is ephemeral), volumes help persist data, share data between containers, and support various storage backends.

---Why Do We Use Volumes?

-Container storage is temporary — deleted on restart.
-Volumes solve this by:
- -Persisting data across restarts
- -Sharing data between containers
- -Mounting configs/secrets into containers

---Common Volume Types in Kubernetes

Volume Type	Description
1. emptyDir	Temporary storage created when pod starts; deleted when pod ends
2. hostPath	Mounts a file/directory from the host node into the pod
3. configMap	Mounts non-sensitive config data from a ConfigMap
4. secret	Mounts sensitive data (passwords, tokens) from a Secret
5. persistentVolumeClaim (PVC)	Used for long-term persistent storage like cloud disks
6. nfs	Mounts an NFS (network file system) share
7. projected	Combines secrets, configMaps, and more into one volume

---Summary

Volume Type	Persistence	Use Case
- emptyDir	✕ No	Scratch space, temp files
- hostPath	✕ No	Local testing, logs
- configMap	✕ No	Configuration files
- secret	✕ No	Passwords, tokens
- PVC	✓ Yes	Databases, important application data

◆ What are Volumes in Kubernetes?

- By default, containers in Pods are **ephemeral** (data is lost when a pod restarts).
- **Volumes** solve this by providing storage that survives pod restarts and can be shared between containers.
- Volumes can be **ephemeral** (temporary) or **persistent** (backed by PVC, cloud, NFS, etc.).

◆ Types of Volumes in Kubernetes with YAML Snippets

1. emptyDir

- Temporary storage, deleted when pod stops.

```
volumes:
  - name: cache-vol
    emptyDir: {}
```

2. hostPath

- Mounts a file/directory from the **host node** into the pod.

```
volumes:
  - name: hostpath-vol
    hostPath:
      path: /data/logs
      type: Directory
```

3. ConfigMap Volume

- Mounts key-value pairs from a **ConfigMap** as files.

```
volumes:
  - name: config-vol
    configMap:
      name: app-config
```

---

4. Secret Volume

- Mounts **sensitive data** (passwords, tokens) from a Secret.

```
volumes:
  - name: secret-vol
    secret:
      secretName: db-secret
```

---

5. PersistentVolumeClaim (PVC)

- Most common in production → connects pod to **PersistentVolume** (EBS, NFS, AzureDisk, etc.).

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

---

6. NFS (Network File System)

- Shared storage between pods.

```
volumes:
  - name: nfs-vol
    nfs:
      server: 10.0.0.50
      path: "/exported/path"
```

---

7. AWS EBS Volume

- Attaches an existing **AWS Elastic Block Store** volume.

```
volumes:
  - name: ebs-vol
    awsElasticBlockStore:
      volumeID: vol-0abcd1234ef567890
      fsType: ext4
```

---

8. GCP Persistent Disk

- Attaches a **Google Persistent Disk**.

```
volumes:
  - name: gcp-pd
    gcePersistentDisk:
      pdName: my-disk
      fsType: ext4
```

---

9. Azure Disk

- Attaches an **Azure Managed Disk**.

```
volumes:
  - name: azure-disk
    azureDisk:
      diskName: myDisk
      diskURI: /subscriptions/.../providers/Microsoft.Compute/disks/myDisk
```

---

10. CSI Volumes (Container Storage Interface)

- Standardized method to use 3rd-party/cloud storage (e.g., EBS CSI, Ceph, Portworx).

```
volumes:
  - name: csi-vol
    csi:
      driver: ebs.csi.aws.com
      volumeHandle: vol-0abcd1234ef567890
```

◆ Interview-ready Summary

☞ *"Volumes in Kubernetes provide storage to pods. Types include ephemeral ones like emptyDir and hostPath, config-driven ones like ConfigMap and Secret, and persistent ones like PVC, NFS, cloud provider disks (AWS EBS, GCP PD, Azure Disk), and CSI volumes. In production, we usually use PVCs with dynamic provisioning via StorageClasses."*

-------Persistent Volumes (PV) and Persistent Volume Claims (PVC) in Kubernetes

In Kubernetes, Persistent Volume (PV) and Persistent Volume Claim (PVC) are used to manage storage independently of the lifecycle of Pods.

1. Persistent Volume (PV)
-A Persistent Volume is a piece of storage in the cluster that has been provisioned by an administrator or dynamically using Storage Classes.
-Key Points:
    -It is a resource in the cluster, like CPU or RAM.
    -Created independently of a Pod.
    -Backed by physical storage (like EBS, NFS, GCE Disk, etc.).
    -Defined by a YAML manifest.

2. Persistent Volume Claim (PVC)
-A Persistent Volume Claim is a request for storage by a user or a Pod.
-Key Points:
    -Users create a PVC to request a specific amount and access mode of storage.
    -Kubernetes binds the PVC to an available PV that matches the request.
    -Once bound, the PVC can be mounted to a Pod as a volume.

---How PV & PVC Work Together
1. Admin creates a PV (or it is dynamically created).
2. User creates a PVC.
3. Kubernetes matches the PVC to an available PV.
4. Pod uses the PVC in its volume section.
5. PVCs must match PVs in size, access mode, and StorageClass.

----Kubernetes volumes enable:

1. Ephemeral storage → For temporary data (emptyDir).
2. Persistent storage → For databases (PV/PVC).
3. Configuration injection → Via configMap/secret.

◆ What is a Persistent Volume (PV)?

- A **Persistent Volume (PV)** is a piece of storage in the cluster.
- It is provisioned by an administrator (static) or dynamically through a **StorageClass**.
- PV is a cluster resource, just like CPU or memory, but for storage.
- PV can be backed by **local disk, NFS, AWS EBS, Azure Disk, GCP PD, CSI drivers**, etc.

✅ Example:

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

◆ What is a Persistent Volume Claim (PVC)?

- A **Persistent Volume Claim (PVC)** is a request for storage by a pod.
- PVCs specify **size** and **access mode** (e.g., ReadWriteOnce, ReadWriteMany).
- Kubernetes automatically **binds PVC to a matching PV**.
- If using dynamic provisioning, PVC triggers a new PV creation from a **StorageClass**.

✅ Example:

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

◆ Pod Using PVC

```
apiVersion: v1
kind: Pod
metadata:
  name: pvc-pod
spec:
  containers:
    - name: app
      image: nginx
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: storage
  volumes:
    - name: storage
      persistentVolumeClaim:
        claimName: my-pvc
```

◆ **Key Concepts**

- **PV** = Actual storage (supply).
- **PVC** = Request for storage (demand).
- **Binding** = PVC is automatically mapped to a suitable PV.
- **StorageClass** = Defines how dynamic PVs are created (e.g., AWS gp3 EBS).
- **Access Modes**:
  - ReadWriteOnce → Single node read/write.
  - ReadOnlyMany → Multiple nodes read-only.
  - ReadWriteMany → Multiple nodes read/write.
- **Reclaim Policies**:
  - Retain → Data kept even if PVC is deleted.
  - Delete → Volume deleted with PVC.
  - Recycle → Deprecated (basic wipe).

◆ **Interview-ready Answer**

☞ *"In Kubernetes, a Persistent Volume (PV) is the actual storage resource, while a Persistent Volume Claim (PVC) is a request for storage by pods. PVCs are bound to PVs based on size and access modes. For automation, StorageClasses are used to dynamically provision PVs. This separation allows developers to request storage without knowing underlying storage details, and admins to manage storage independently."*

👉 Let's now cover the **real-world production way** → using **StorageClass with dynamic provisioning** so you don't need to manually create PersistentVolumes (PVs).

1). Dynamic Provisioning with AWS EBS (gp3)

Step 1: Create a StorageClass

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gp3-sc
provisioner: ebs.csi.aws.com
parameters:
  type: gp3                    # gp2, gp3, io1 supported
  fsType: ext4
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

✅ Explanation:

- provisioner: ebs.csi.aws.com → Uses AWS EBS CSI driver.
- type: gp3 → Storage type (can be gp2, gp3, io1 etc).
- WaitForFirstConsumer → PV created only when a pod is scheduled, ensuring it's in the right AZ.

Step 2: Create a PersistentVolumeClaim (PVC)

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-dynamic-pvc
spec:
  accessModes:
    - ReadWriteOnce           # EBS can only be mounted to one node at a time
  resources:
    requests:
      storage: 10Gi
  storageClassName: gp3-sc
```

✅ Explanation:

- PVC automatically provisions an **EBS volume** when bound.
- No need to manually define a PV.

Step 3: Pod Using the PVC

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ebs-dynamic-pod
spec:
  containers:
    - name: app
      image: nginx
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: ebs-storage
  volumes:
    - name: ebs-storage
      persistentVolumeClaim:
        claimName: ebs-dynamic-pvc
```

◆ **Interview-ready Summary**

☞ *"In Kubernetes, we can either create PVs manually and bind them to PVCs, or use StorageClasses for dynamic provisioning. In AWS, the most common approach is to use an EBS CSI driver with a StorageClass (like gp3). Then PVCs automatically create and manage the lifecycle of EBS volumes. This ensures automation, scalability, and consistency in production."*

-----Installing and Configuring AWS EBS CSI Driver for Kubernetes Cluster with Dynamic Provisioning of EBS Volumes

This guide provides the steps to install and configure the AWS EBS CSI Driver on a Kubernetes cluster to enable the use of Amazon Elastic Block Store (EBS) volumes as persistent volumes.

-Prerequisites

- A running Kubernetes cluster.
- A Kubernetes version of 1.20 or greater.
- An AWS account with access to create an IAM user and obtain an access key and secret key.

--Installing Helm

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications.

- Run the following commands to install Helm

```
        # curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
        # chmod 700 get_helm.sh
        # ./get_helm.sh
```

--Installing AWS EBS CSI Driver

- Create a secret to store your AWS access key and secret key using the following command:

```
        # kubectl create secret generic aws-secret \
  --namespace kube-system \
  --from-literal "key_id=${AWS_ACCESS_KEY_ID}" \
  --from-literal "access_key=${AWS_SECRET_ACCESS_KEY}"
```

- Add the AWS EBS CSI Driver Helm chart repository:

```
        # helm repo add aws-ebs-csi-driver https://kubernetes-
sigs.github.io/aws-ebs-csi-driver
        # helm repo update
```

- Deploy the AWS EBS CSI Driver using the following command:

```
        # helm upgrade --install aws-ebs-csi-driver \
  --namespace kube-system \
        # aws-ebs-csi-driver/aws-ebs-csi-driver
```

- Verify that the driver has been deployed and the pods are running:

```
        # kubectl get pods -n kube-system -l
app.kubernetes.io/name=aws-ebs-csi-driver
```

--Provisioning EBS Volumes

- Create a storageclass.yaml file and apply the storageclass.yaml file using the following command:

```
kubectl apply -f storageclass.yaml
```

- Create a pvc.yaml file and apply the pvc.yaml file using the following command:

```
kubectl apply -f pvc.yaml
```

- Create a pod.yaml file and apply the pod.yaml file using the following command:

```
kubectl apply -f pod.yaml
```

- Verify that the EBS volume has been provisioned and attached to the pod:

```
        # kubectl exec -it app -- df -h /data
```

The output of the above command should show the mounted EBS volume and its available disk space.

--Conclusion

In this guide, we have shown you how to install and configure the AWS EBS CSI Driver on your Kubernetes cluster and how to use it for dynamic provisioning of EBS volumes.

2). Dynamic Provisioning with NFS in Kubernetes

Step 1: Install the NFS Subdir External Provisioner

This is the controller that will create PVs dynamically inside your NFS server path.

```
  # helm repo add nfs-subdir-external-provisioner https://kubernetes-sigs.github.io/nfs-subdir-external-provisioner/
  # helm install nfs-provisioner nfs-subdir-external-provisioner/nfs-subdir-external-provisioner \
 --set nfs.server=<NFS_SERVER_IP> \
 --set nfs.path=/exported/path
```

- Replace <NFS_SERVER_IP> with your NFS server's IP.
- Replace /exported/path with the NFS shared directory path.

Step 2: Create StorageClass for NFS

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-storage
provisioner: k8s-sigs.io/nfs-subdir-external-provisioner
parameters:
  archiveOnDelete: "true"   # keeps data even after PVC is deleted
reclaimPolicy: Delete       # can be Delete or Retain
```

Step 3: Create a PersistentVolumeClaim (PVC)

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  storageClassName: nfs-storage
  accessModes:
    - ReadWriteMany   # key feature of NFS (multi-node support)
  resources:
    requests:
      storage: 2Gi
```

Step 4: Use PVC in a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-test-pod
spec:
  containers:
    - name: app
      image: busybox
      command: [ "sleep", "3600" ]
      volumeMounts:
        - name: nfs-volume
          mountPath: /data
  volumes:
    - name: nfs-volume
      persistentVolumeClaim:
        claimName: nfs-pvc
```

◆ **Key Points for Interview**

- **NFS allows ReadWriteMany (RWX)** → multiple pods on multiple nodes can share the same data.
- **Dynamic provisioning** → no need to manually create PVs. The provisioner auto-creates subdirectories in NFS for each PVC.
- **Use cases** → logs, CMS (WordPress/Joomla), ML shared datasets, or any app requiring shared storage.

---

3). Dynamic Provisioning with AWS EFS in Kubernetes

Step 1: Install the AWS EFS CSI Driver

```
# kubectl apply -k "github.com/kubernetes-sigs/aws-efs-csi-driver/deploy/kubernetes/overlays/stable/ecr/?ref=release-1.5"
```

This installs the CSI driver in your cluster.

Step 2: Create an EFS File System in AWS

- Go to **AWS Console** → **EFS** → **Create file system**.
- Note down the **File System ID** (e.g., fs-12345678).
- Ensure EFS mount targets exist in all subnets used by your worker nodes.

Step 3: Create an EFS Access Point

-In EFS console → Access Points → Create → choose the file system.
-This gives you a stable mount path (e.g., /data).

Note the **Access Point ID** (e.g., fsap-12345678).

Step 4: Create a StorageClass for EFS

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: efs-sc
provisioner: efs.csi.aws.com
parameters:
  provisioningMode: efs-ap
  fileSystemId: fs-12345678   # Replace with your File System ID
  directoryPerms: "700"
  gidRangeStart: "1000"
  gidRangeEnd: "2000"
  basePath: "/dynamic_provisioning"
reclaimPolicy: Retain
volumeBindingMode: Immediate
```

Step 5: Create a PersistentVolumeClaim (PVC)

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-pvc
spec:
  accessModes:
    - ReadWriteMany   # Multiple pods across nodes can use
  storageClassName: efs-sc
  resources:
    requests:
      storage: 5Gi
```

Step 6: Use the PVC in a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: efs-app
spec:
  containers:
    - name: app
      image: nginx
      volumeMounts:
        - name: efs-storage
          mountPath: /usr/share/nginx/html
  volumes:
    - name: efs-storage
      persistentVolumeClaim:
        claimName: efs-pvc
```

◆ Key Differences (NFS vs EFS)

Feature	NFS (Self-managed)	EFS (AWS-managed)
Setup	Requires your own NFS server	Fully managed by AWS
Scaling	Limited by server capacity	Scales automatically
HA/DR	You must configure manually	Multi-AZ by default
Cost	Only server + EBS cost	Pay per use (storage consumed)
Access	RWX supported	RWX supported

✅ **Interview-ready summary:**

"For shared storage in Kubernetes, I can use either self-managed NFS or AWS-managed EFS. With dynamic provisioning via CSI drivers, pods can automatically request storage using PVCs. EFS is preferred in AWS production as it provides multi-AZ high availability and removes operational overhead."

--Internal Storage Vs External Storage :

◆ Internal Storage (Local/Node-level Storage)

- Storage that exists **inside the node itself**.
- Examples:
  - Node's **root disk** (e.g., /var/lib/docker, /var/lib/kubelet).
  - **ephemeral storage** → temporary storage for container logs, emptyDir volumes.
  - Local SSDs/HDDs attached directly to the node.

-**Characteristics**:

- **Fast & low latency** (directly on the node).
- **Ephemeral** → data is lost if pod reschedules or node terminates.
- **Used for**: caching, temporary files, logs.

◆ External Storage (Cluster/Cloud-level Storage)

- Storage **outside the node**, usually network-attached or managed by the cloud provider.
- Examples:
  - **AWS EBS**, **Azure Disk**, **GCP Persistent Disk** → block storage.
  - **AWS EFS**, **NFS**, **Ceph**, **GlusterFS** → shared file systems.
  - **Object storage** like S3 (used indirectly via CSI drivers).

-**Characteristics**:

- **Persistent** → data survives pod rescheduling and node termination.
- **Shared** → some support **RWX (multiple pods)** like NFS/EFS.
- **Used for**: databases, stateful applications, logs, backups.

◆ Quick Interview Example

☞ *"Internal storage in Kubernetes means node-local ephemeral disks or local paths, mainly for temporary data. External storage is persistent and provisioned via PV/PVC with cloud or network storage like EBS, EFS, or NFS. For production, we always prefer external storage for stateful workloads since internal storage vanishes when pods or nodes die."*

-----StatefulSet in Kubernetes?

A StatefulSet is a Kubernetes controller used to manage stateful applications. It ensures stable and unique network identities, persistent storage, and ordered deployment and scaling of pods.

Unlike a Deployment (which is used for stateless applications), a StatefulSet is designed for applications that need to maintain their state, even if the pods are rescheduled.

---Key Features of StatefulSet:

Feature	**Description**
1) Stable identity	Each pod gets a unique and stable name like web-0, web-1, etc.
2) Persistent storage	Each pod gets its own PersistentVolume (via PVC), which persists data even if the pod is deleted or rescheduled.
3) Ordered deployment	Pods are created one at a time, in order (0 → N-1) and deleted in reverse order.
4) Stable network identity	Each pod gets a DNS hostname based on its name (e.g., web-0.my-service).
5) Consistent storage mapping	Storage is not shared between pods; each pod maps to its own volume.

---Storage with StatefulSet

The Storage with StatefulSet YAML file extends the previous StatefulSet example by adding a PersistentVolumeClaim (PVC) for storage using volumeClaimTemplates field. It mounts a volume to the nginx containers in the StatefulSet.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "headless-svc"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: registry.k8s.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: web-pvc
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: web-pvc
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
      storageClassName: ebs-sc
```

-----DaemonSet in Kubernetes

In Kubernetes, workloads are typically deployed using Deployments, StatefulSets, or DaemonSets, depending on the requirement. Among these, a DaemonSet plays a very special role.

--What is a DaemonSet?

A DaemonSet ensures that a copy of a specific Pod runs on all (or selected) nodes in a Kubernetes cluster.

1. If new nodes are added, the DaemonSet automatically deploys the pod to them.
2. If nodes are removed, the pods are also removed automatically.
3. If you delete the DaemonSet, all its managed pods are deleted.

--Use Cases of DaemonSet

DaemonSets are commonly used for infrastructure-level workloads, such as:

1. **Log Collection:** Running Fluentd, Filebeat, or Logstash on every node to collect logs.
2. **Monitoring:** Running Prometheus Node Exporter, Datadog Agent, or New Relic Daemon to collect system metrics.
3. **Networking:** Running CNI plugins, kube-proxy, or network agents.
4. **Storage:** Running storage daemons (like Ceph, GlusterFS) on every node.

--Key Characteristics of DaemonSet

-One pod per node (by default).
-Can target all nodes or a subset of nodes using nodeSelector, affinity, or taints/tolerations.
-Unlike Deployments, scaling is not manual—scaling happens automatically as nodes join or leave.

--DaemonSet YAML Example

Below is a simple DaemonSet YAML manifest that deploys an NGINX pod on every node:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
  namespace: default
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

-----Multi-container Pod

A multi-container pod is a single Kubernetes Pod that runs multiple containers. These containers share the same network namespace, which means:
-They can communicate with each other via localhost.
-They share the same IP address and ports.

Each container can expose its own port, but all ports must be unique within the pod.

--Why Use Multiple Containers in One Pod?

-Sidecar pattern: Add helper containers (e.g., log shipper, proxy).
-Ambassador pattern: Connect to external systems.
-Adapter pattern: Translate or modify data.

--Example: Multi-container Pod with Ports

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80

  - name: busybox-container
    image: busybox
    command: ["sh", "-c", "while true; do echo hello; sleep 10; done"]
    ports:
    - containerPort: 8080
```

--Explanation:

-nginx-container:
-Runs an Nginx web server.
-Exposes port 80.

-busybox-container:
-A simple sidecar that prints "hello" every 10 seconds.
-Exposes port 8080.

-Even though both containers are in the same pod:
They must not conflict on port numbers.
You can access one from the other using localhost:<port>.

--Accessing Ports Between Containers
-Inside the pod:
-Nginx can access BusyBox at localhost:8080.
-BusyBox can access Nginx at localhost:80.

--Summary

Feature	**Description**
-Shared network	-Containers share IP and ports
-Separate processes	-Containers run independently
-Port conflict	-Avoid using the same port in multiple containers
-Use case	-Logging, monitoring, sidecar proxy, adapters

--OR Multi-container Pod YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-example
spec:
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: main-app
    image: my-application
    ports:
    - containerPort: 8080
    volumeMounts:
    - name: shared-data
      mountPath: /data
  - name: log-processor
    image: log-collector
    volumeMounts:
    - name: shared-data
      mountPath: /logs
```

-----Init Containers in Kubernetes

An Init Container is a special type of container that runs before the main application containers in a Pod start. It is used to perform initialization tasks — like setting up config, downloading dependencies, waiting for external services, etc.

--Key Characteristics of Init Containers:

Feature	Description
1. Runs before main containers	-Each Init container runs one at a time, in order.
2. Retries on failure	-If an Init container fails, Kubernetes will restart it until it succeeds.
3. No concurrent execution	-The main containers start only after all Init containers finish successfully.
4. Can access shared volumes	-Init containers can share volumes with main containers.

--Example: Pod with Init Container

```
apiVersion: v1
kind: Pod
metadata:
  name: init-container-demo
spec:
  containers:
  - name: main-app
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html

  initContainers:
  - name: init-script
    image: busybox
    command: ['sh', '-c', 'echo "Welcome from Init Container!" > /data/index.html']
    volumeMounts:
    - name: shared-data
      mountPath: /data
  volumes:
  - name: shared-data
    emptyDir: {}
```

--Explanation:

-Init Container (init-script): Writes an HTML file into a shared volume.

-Main Container (nginx): Serves the HTML content from that shared volume.

-emptyDir volume: Used to share data between the Init and main containers.

--Common Use Cases

1. Configuration/Secret Preparation

```
initContainers:
- name: config-downloader
  image: busybox
  command: ['sh', '-c', 'wget -O /config/appsettings.json http://config-server/v1/settings']
  volumeMounts:
  - name: config-volume
    mountPath: /config
```

2. Database Migration

```
initContainers:
- name: db-migrate
  image: myapp-db-migrations
  env:
  - name: DB_HOST
    value: "postgres"
  command: ['sh', '-c', 'alembic upgrade head']
```

3. Dependency Waiting

```
initContainers:
- name: wait-for-db
  image: busybox
  command: ['sh', '-c', 'until nc -z postgres 5432; do echo waiting for db; sleep 2; done;']
```

4. Security Setup

```
initContainers:
- name: cert-downloader
  image: vault
  command: ['vault', 'login', '-method=aws']
  env:
  - name: VAULT_ADDR
    value: "https://vault.example.com"
```

# init-container.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: init-test

spec:
  initContainers:
  - name: init-container
    image: busybox:latest
    command: ["sh","-c", "echo 'Initalization started ...'; sleep 10; echo 'Initization completed.'"]

  containers:
  - name: main-container
    image: busybox:latest
    command: ["sh","-c", "echo 'Main container started'"]
```

# sidecar-container.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: sidecar-test

spec:
  volumes:
  - name: shared-logs
    emptyDir: {}

  containers:

  - name: main-container                    # produce logs
    image: busybox
    command: ["sh","-c", "while true; do echo 'Hello Dosto' >> /var/log/app.log; sleep 5; done"]
    volumeMounts:
    - name: shared-logs
      mountPath:  /var/log/

    - name: sidecar-container               # display logs
    image: busybox
    command: ["sh","-c", "tail -f /var/log/app.log"]
    volumeMounts:
    - name: shared-logs
      mountPath:  /var/log/
```

----Pod in Kubernetes

A Pod is the smallest and simplest deployable unit in Kubernetes. It represents a single instance of a running process (container) in your cluster.

-It can run:

1. A single container (most common case)
2. Multiple containers that need to work together (sharing storage and network)

--Key Features of a Pod

-Feature	-Description
1. Grouped Containers	-One or more containers running together, sharing network/storage
2. Shared Network	-All containers share the same IP, hostname, and ports
3. Shared Storage	-Can mount shared Volumes for data exchange
4. Ephemeral	-Pods are not self-healing – if they fail, they won't restart unless managed by a controller (e.g., Deployment)
5. Managed by Controllers	-Pods are usually managed by Deployments, ReplicaSets, Jobs, etc.

1) Basic Pod YAML Example

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

-This will run a single nginx container in a pod.

2) Multi-container Pod Example

```
spec:
  containers:
  - name: app
    image: my-app
  - name: sidecar
    image: my-logger
```

-Both containers:

1. Share the same IP address
2. Can communicate over localhost
3. Share mounted Volumes if configured

--Common Use Cases

1. Running single-container apps (e.g., NGINX, API)
2. Running tightly coupled containers (e.g., app + sidecar logger)
3. Serving as units in Deployments, Jobs, or DaemonSets

job.yml

```
kind: Job
apiVersion: batch/v1
metadata:
  name: demo-job
  namespace: nginx
spec:
  completions: 1
  parallelism: 1
  template:
    metadata:
      name: demo-job-pod
      labels:
        app: batch-task
    spec:
      containers:
      - name: batch-container
        image: busybox:latest
        command: ["sh", "-c" ,"echo Hello Dosto! && sleep 10"]
      restartPolicy: Never
```

-----Job in Kubernetes

A Job in Kubernetes is a resource used to run a task to completion, rather than running it continuously like a Deployment or Pod. It ensures that a certain number of pods successfully complete their work, and once done, the Job is finished.

-It's perfect for batch processing, data migration, report generation, or one-time scripts.

--Basic Example of a Kubernetes Job

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

-This runs a single pod that prints a message and exits.

--Key Fields

Field	Description
1. spec.template	Pod template to define what to run
2. restartPolicy: Never	Required for jobs (not Always)
3. backoffLimit	Number of retries before considering the Job failed
4. completions	Total number of successful runs needed
5. parallelism	Number of pods to run in parallel

--Parallel Jobs

Run multiple pods in parallel until all complete:

```
spec:
  completions: 5
  parallelism: 2
```

-This runs 2 pods at a time until 5 finish successfully.

-Restart Policies

Policy	Used In Job?	Meaning
1. Never	✓ Yes	Pod won't be restarted
2. OnFailure	✓ Yes	Pod restarts if it fails
3. Always	✗ No	Not allowed in Jobs

-Job Lifecycle

1. Job is created.
2. Kubernetes creates one or more pods to do the task.
3. Once the required pods successfully complete, the Job is marked complete.
4. Failed pods (up to retry limit) are cleaned up.

----CronJob in Kubernetes?

A CronJob in Kubernetes allows you to run Jobs on a time-based schedule, similar to Linux cron. It is used for recurring tasks like backups, report generation, clean-up scripts, etc.

--Key Concept

A CronJob creates a Job at a scheduled time. Each Job runs to completion (like one-time scripts or tasks).

--Basic CronJob YAML Example

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

--Important Fields

Field	Description
1. schedule	Cron format for when the job runs
2. jobTemplate	Defines the actual job to run
3. restartPolicy	Usually OnFailure or Never
4. startingDeadlineSeconds	Deadline to start the job if it was missed
5. concurrencyPolicy	What to do if the job is still running
6. successfulJobsHistoryLimit	How many successful job histories to keep
7. failedJobsHistoryLimit	How many failed job histories to keep

--Example CronJob Configurations

1. Basic CronJob (Daily Backup)

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: daily-backup
```

```
spec:
  schedule: "0 2 * * *"                    # Runs at 2 AM daily
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: busybox
            command: ["sh", "-c", "tar czf /backup/data-$(date +%s).tar.gz /data"]
          restartPolicy: OnFailure
```

2. CronJob with Concurrency Policy

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: report-generator
spec:
  schedule: "0 5 * * 1"          # Every Monday at 5 AM
  concurrencyPolicy: Forbid   # Skips new run if previous Job is still running
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: report
            image: python:3.9
            command: ["python", "/scripts/generate_report.py"]
          restartPolicy: Never
```

3. CronJob with Successful Job History

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: log-cleaner
spec:
  schedule: "0 0 * * *"               # Midnight daily
  successfulJobsHistoryLimit: 3   # Keeps logs of last 3 successful Jobs
  failedJobsHistoryLimit: 1         # Keeps logs of last 1 failed Job
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: cleaner
            image: alpine
            command: ["sh", "-c", "rm -rf /logs/*.old"]
          restartPolicy: Never
```

cron-job.yml

```
kind: CronJob
apiVersion: batch/v1
metadata:
  name: minute-backup
  namespace: nginx

spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          name: minute-backup
          labels:
            app: minute-backup

        spec:
          containers:
          - name: backup-container
            image: busybox
            command:
            - sh
            - -c
            - >
              echo "Backup Started" ;
              mkdir -p /backups &&
              mkdir -p /demo-data &&
              cp -r /demo-data /backups &&
              echo "Backup Completed" ;
            volumeMounts:
              - name: data-volume
                mountPath: /demo-data
              - name: backup-volume
                mountPath: /backups
          restartPolicy: OnFailure
          volumes:
            - name: data-volume
              hostPath:
                path: /demo-data
                type: DirectoryOrCreate
            - name: backup-volume
              hostPath:
                path: /backups
                type: DirectoryOrCreate
```

------Role, RoleBinding, ClusterRole, and ClusterRoleBinding in Kubernetes

In Kubernetes, RBAC (Role-Based Access Control) is used to manage who can do what within your cluster. It controls access to Kubernetes resources based on the roles assigned to users, groups, or service accounts.

1. Role

A Role defines a set of permissions (rules) to perform actions (verbs like get, list, create, delete) on Kubernetes resources like Pods, Services, ConfigMaps within a specific namespace.

--Example Role (namespace-specific):

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

This Role allows reading pods (get, watch, list) in the dev namespace.

2. RoleBinding

A RoleBinding assigns a Role to a user, group, or service account within the same namespace.

--Example RoleBinding:

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

This RoleBinding gives user alice access to read pods in the dev namespace.

3. ClusterRole

-A ClusterRole is similar to a Role, but it is cluster-wide:
-Can define permissions across all namespaces.
-Can be used for non-namespaced resources (like nodes, persistent volumes, etc.)

--Example ClusterRole:

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

This ClusterRole allows reading pods across all namespaces.

4. ClusterRoleBinding

A ClusterRoleBinding assigns a ClusterRole to a user, group, or service account across the entire cluster.

--Example ClusterRoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-all-pods-binding
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-pod-reader
  apiGroup: rbac.authorization.k8s.io
```

-This gives user alice permission to read pods in all namespaces.

--Comparison Table

-Component	**-Scope**	**-Bound To**	**-Typical Use Case**
1.Role	Namespace	RoleBinding	Grant permissions within a namespace
2.RoleBinding	Namespace	Role or ClusterRole	Apply permissions to namespace subjects
3.ClusterRole	Cluster-wide	ClusterRoleBinding or RoleBinding	Define cluster-wide permissions
4.ClusterRoleBinding	Cluster-wide	ClusterRole	Grant permissions across all namespaces

--Important Notes

1. RoleBinding with ClusterRole:

You can use a RoleBinding to bind a ClusterRole, which will grant the ClusterRole's permissions but only within the RoleBinding's namespace.

2. Predefined ClusterRoles:
Kubernetes includes several default ClusterRoles like:

-view: Read-only access
-edit: Read/write access (except permissions)
-admin: Full control within a namespace
-cluster-admin: Full control cluster-wide

3. Aggregation:
ClusterRoles can aggregate rules from other ClusterRoles using aggregationRule.

4. Verbs Cheat Sheet:

-Read operations: get, list, watch
-Write operations: create, update, patch, delete
-Special operations: use, bind, escalate, impersonate

------Service Account in Kubernetes

A Service Account in Kubernetes is used to provide an identity to pods or internal processes (like controllers or jobs) so they can interact with the Kubernetes API server securely.

--Key Points about Service Accounts

-Feature	**-Description**
1.Used by pods	-A pod uses a service account to authenticate to the Kubernetes API.
2.Auto-mounted	-Kubernetes automatically mounts a token inside the pod at /var/run/secrets/kubernetes.io/serviceaccount/.
3.Default account	-Every namespace has a default service account created automatically.
4.RBAC	-You can assign Roles or ClusterRoles to service accounts using RoleBinding or ClusterRoleBinding.

--Example Use Case

A monitoring agent (like Prometheus) running in a pod needs to read metrics from the Kubernetes API. It uses a service account with read access to pods and nodes.

--Example: Creating a Service Account

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
  namespace: default
```

-This creates a service account named my-service-account in the default namespace.

--Using the Service Account in a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  serviceAccountName: my-service-account
  containers:
  - name: busybox
    image: busybox
    command: ["sh", "-c", "sleep 3600"]
```

-This pod will use the my-service-account, and its token will be mounted automatically.

--Binding Roles to a Service Account (RBAC)

-RoleBinding Example

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: sa-read-pods
  namespace: default
subjects:
- kind: ServiceAccount
  name: my-service-account
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

-This grants the my-service-account permission to read pods in the default namespace.

--Bound Service Account Tokens (Kubernetes 1.22+)

-More secure, time-bound tokens
-Reduced risk from token leakage

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - image: nginx
    name: nginx
  serviceAccountName: my-sa
  volumes:
  - name: kube-api-access
    projected:
      sources:
      - serviceAccountToken:
          path: token
          expirationSeconds: 3600
          audience: api
```

--User Account vs Service Account

-Feature	**-User Account**	**-Service Account**
1. Used by humans	Yes	No
2. Used by pods	No	Yes
3. Managed by	Admin (external)	Kubernetes
4. Default in pods	No	Yes (default)

--Summary

1. Service Accounts are used to authenticate pods to the Kubernetes API.
2. Each pod uses one service account (either default or custom).
3. You can control what a service account can do by attaching RBAC Roles.
4. Ideal for internal communication between components (e.g., controllers, dashboards, metrics agents).

------Security Context in Kubernetes

In Kubernetes, a Security Context defines security-related settings (such as user, group, permissions, capabilities, etc.) for a pod or a container. It helps control what privileges a container or pod has when running inside the cluster.

--Key Features of Security Context

Feature	Description
1. runAsUser	Run the container as a specific Linux user ID (UID)
2. runAsGroup	Set the primary group ID (GID)
3. runAsNonRoot	Enforce that the container does not run as root
4. fsGroup	Set the group ID for mounted volumes
5. readOnlyRootFilesystem	Make the root file system read-only
6. capabilities	Add/drop Linux kernel capabilities

1) Example: Security Context at Pod Level

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  containers:
  - name: app
    image: nginx
    volumeMounts:
    - name: data
      mountPath: /usr/share/nginx/html
  volumes:
  - name: data
    emptyDir: {}
```

-In this example:

-All containers run as user 1000, group 3000.
-Mounted volumes are owned by group 2000.

2) Example: Security Context at Container Level

```
apiVersion: v1
kind: Pod
metadata:
  name: container-secure
spec:
  containers:
  - name: nginx
    image: nginx
    securityContext:
      runAsUser: 1001
      readOnlyRootFilesystem: true
      capabilities:
        drop: ["ALL"]
        add: ["NET_BIND_SERVICE"]
```

-This sets:

-User ID = 1001 for the container

-Root filesystem = read-only

-Linux capabilities = Only allow binding to low-numbered ports (like port 80)

3) Example: Secure Pod Configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: secured-app
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    runAsGroup: 2000
    fsGroup: 3000
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: main
    image: my-app:v1
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: ["ALL"]
      readOnlyRootFilesystem: true
    volumeMounts:
    - name: tmp
      mountPath: /tmp
  volumes:
  - name: tmp
    emptyDir: {}
```

--Where Can Security Context Be Defined?

-Level	-Description
1. Pod	Applies to all containers in the pod (default settings)
2. Container	Overrides pod-level settings for that specific container

--Key Security Context Fields

1. User and Group Controls

-runAsUser: User ID to run the container processes (e.g., avoid root with runAsUser: 1000)
-runAsGroup: Primary group ID for container processes
-runAsNonRoot: Boolean to enforce non-root execution (true prevents root user)

2. Filesystem Controls

-fsGroup: Special supplemental group for volume permissions
-fsGroupChangePolicy: Defines how volume ownership is changed (Always or OnRootMismatch)
-readOnlyRootFilesystem: Mounts root filesystem as read-only (recommended for security)

3. Privilege Controls

-allowPrivilegeEscalation: Prevents child processes from gaining more privileges than parent
-privileged: Gives extended privileges (almost always should be false)
-capabilities: Add/drop Linux capabilities (better than running as root)

4. SELinux/AppArmor/Seccomp

-seLinuxOptions: SELinux labels for the container
-appArmorProfile: AppArmor profile to apply
-seccompProfile: Seccomp profile for syscall filtering

--Summary

-Setting	-Purpose
1. runAsUser	Run container as specific user
2. runAsGroup	Set container's primary group
3. fsGroup	Group ownership of mounted volumes
4. readOnlyRootFilesystem	Prevent changes to root FS
5. capabilities	Add or remove Linux capabilities
6. runAsNonRoot	Force container to avoid root user

--------Scheduling in Kubernetes

-Scheduling is the process where the Kubernetes Scheduler decides which node a pod should run on.
-Kubernetes automatically selects the best node based on:

1. Resource availability (CPU, memory)
2. Constraints like taints/tolerations, node affinity, nodeSelector
3. Custom rules or policies

---What is nodeSelector?

nodeSelector is the simplest form of node selection.
It tells the scheduler: ➡ "Only run this pod on a node that has a specific label."

---How it Works

1. Nodes are labeled with key-value pairs.
2. Pods specify nodeSelector to match those labels.
3. The pod will only run on nodes that have matching labels.

------Step-by-Step Example

◆ 1. Label a Node

```
# kubectl label nodes node1 disktype=ssd
```

This adds a label to the node: disktype=ssd

◆ 2. Use nodeSelector in Pod YAML

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

-Now this pod will only run on nodes labeled with disktype=ssd.

---------Use Cases

Scenario	nodeSelector Usage
1. Use SSD nodes only	disktype: ssd
2. Use GPU nodes	hardware: gpu
3. Run monitoring only on master node	role: master
4. Split workload by region/zone	region: us-east1, zone: us-east1-a

------Limitations of nodeSelector

1. Exact match only — no logic or ranges
2. Not flexible (can't say "run on any of two values")
3. Better for simple use cases

-For advanced scenarios, use node affinity, taints/tolerations, or custom schedulers.

---View Node Labels

```
# kubectl get nodes --show-labels
# kubectl describe node <node-name>
```

-----Summary

Feature	nodeSelector
1. Purpose	Run pod only on matching labeled nodes
2. Based on	Key-value labels
3. Use Case	Simple scheduling rules
4. Alternative	Node affinity (advanced)
5. Command to label	kubectl label nodes node1 key=value

------Node Affinity in Kubernetes?

-Node Affinity is an advanced and flexible way to control pod scheduling on nodes, based on labels — like nodeSelector, but more powerful.
-It allows you to define rules that specify which nodes a pod should (or must) run on, using expressions, operators, and weighting.

--◆ Types of Node Affinity
Kubernetes supports two types of node affinity:

Type	Description
1. requiredDuringSchedulingIgnoredDuringExecutio	Hard rule – Pod must run only on matching nodes
2. preferredDuringSchedulingIgnoredDuringExecution	Soft rule – Scheduler prefers but doesn't require matching nodes

----Example 1: Hard Node Affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: hard-affinity-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: disktype
                operator: In
                values:
                  - ssd
  containers:
    - name: app
      image: nginx
```
-This pod will only run on nodes where disktype=ssd.

----Example 2: Soft Node Affinity (Preferred)

```
apiVersion: v1
kind: Pod
metadata:
  name: soft-affinity-pod
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: region
                operator: In
                values:
                  - us-east1
  containers:
    - name: app
      image: nginx
```

-This pod will prefer nodes in region=us-east1 but can still run elsewhere.

----Supported Operators in matchExpressions

Operator	Description
-In	Key must be in given values
-NotIn	Key must not be in given values
-Exists	Node must have the key
-DoesNotExist	Node must not have the key
-Gt, Lt	For numeric values only (CPU, etc.)

---View Node Labels

```
 # kubectl get nodes --show-labels
 # kubectl describe node <node-name>
```

---Use Cases

Use Case	Node Affinity Rule
-Run on SSD storage nodes	disktype In [ssd]
-Prefer specific region	region In [us-west1] (preferred)
-Exclude certain nodes	hostname NotIn [node1, node2]
-GPU workloads	accelerator Exists

---Summary

Feature	Description
-Node Affinity	Advanced scheduler constraint using labels
-Required Affinity	Pod must run on matching node
-Preferred Affinity	Pod should run on matching node if possible
-Flexibility	More powerful than nodeSelector

------Kubernetes Selectors: Types and Usage

Selectors in Kubernetes are fundamental for identifying and grouping resources. They allow you to define how objects find and interact with each other in a cluster. Here are the main selector types:

1. Label Selectors

Most common type, used across many Kubernetes objects to select resources based on key-value pairs.

-Syntax:

```
selector:
  matchLabels:
    app: frontend
    tier: web
```

-Advanced Matching:

```
selector:
  matchExpressions:
    - {key: environment, operator: In, values: [prod, staging]}
    - {key: version, operator: NotIn, values: [1.0, 1.1]}
```

-Supported Operators: In, NotIn, Exists, DoesNotExist

--Common Uses:

1. Connecting Services to Pods
2. ReplicaSet/Deployment Pod selection
3. Node selection for Pod scheduling

2. Field Selectors

Filter resources based on their metadata fields (not labels).

Syntax:

```
# kubectl get pods --field-selector=status.phase=Running
```

Supported Fields:

-metadata.name
-metadata.namespace
-status.phase (for Pods)
-spec.nodeName (for Pods)

3. Annotation Selectors (Not Native)

While Kubernetes doesn't directly support annotation-based selection, you can filter using:

```
# kubectl get pods -o json | jq '.items[] |
select(.metadata.annotations["custom/annotation"] == "value")'
```

4. Node Selectors

Simple node selection constraint for Pod scheduling.

Example:

```
spec:
  nodeSelector:
    disktype: ssd
    gpu: "true"
```

5. Node Affinity/Anti-Affinity

Advanced replacement for nodeSelector with more expressive syntax.

Types:

-requiredDuringSchedulingIgnoredDuringExecution (hard requirement)

-preferredDuringSchedulingIgnoredDuringExecution (soft preference)

Example:

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: topology.kubernetes.io/zone
          operator: In
          values: [us-west-2a, us-west-2b]
```

6. Pod Affinity/Anti-Affinity

Controls Pod co-location (or separation) based on other Pods' labels.

-Example (Anti-Affinity):

```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchLabels:
          app: cache
      topologyKey: kubernetes.io/hostname
```

7. Final Selector Types

a. Set-Based Selectors: Used in Network Policies and other advanced scenarios:

```
podSelector:
  matchExpressions:
    - {key: role, operator: In, values: [backend, database]}
namespaceSelector:
  matchLabels:
    project: myapp
```

b. Endpoint Selectors: Used by Services to select Pod endpoints:

```
spec:
  selector:
    app: nginx
    tier: frontend
```

--Key Differences Table

-Selector Type	-Scope	-Example Use Case
1. Label	Pods/Nodes/Objects	Service → Pod selection
2. Field	Object metadata	Filter running Pods
3. Node	Nodes	Schedule Pods on SSD nodes
4. Affinity	Advanced placement	Spread Pods across zones
5. Set-Based	Complex rules	Network policy targets

------Label nodes:

```
# kubectl label nodes node2 disktype=ssd
```

--Label namespaces:

```
# kubectl label namespace production environment=production
```

--Label pods:

```
# kubectl label pods my-pod app=web-app tier=frontend
```

--View labels:

```
# kubectl get namespaces/nodes/pods --show-labels
```

Selector Type	**Purpose**	**Example**
Namespace	Select resources in specific namespaces	namespaceSelector: {environment: production}
Node	Schedule pods to specific nodes	nodeSelector: {accelerator: gpu}
Pod	Select pods for services, networking	selector: {app: frontend}

-------Node Affinity in Kubernetes

Node Affinity is a Kubernetes feature that allows you to control which nodes your pods are scheduled on, based on node labels. It’s part of pod scheduling, helping Kubernetes decide where to run a pod.

--Why Use Node Affinity

You use Node Affinity when:

1. You want pods to run only on certain types of nodes (e.g., GPU nodes, SSD nodes).
2. You want to improve performance, cost-efficiency, or compliance by placing pods correctly.

--Node Affinity Types

There are two main types of node affinity:

Type	**Description**
1. RequiredDuringSchedulingIgnoredDuringExecution	-Hard rule – Pod must be scheduled only on nodes that match.
2. PreferredDuringSchedulingIgnoredDuringExecution	-Soft rule – Pod prefers matching nodes, but not mandatory.

-Both types only affect scheduling time — they do not move pods after scheduling.

1) Example 1: Required (Hard) Node Affinity

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: node-type
          operator: In
          values:
          - high-memory
```

-Pod will be scheduled only on nodes with label: node-type=high-memory

2) Example 2: Preferred (Soft) Node Affinity

```
affinity:
  nodeAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      preference:
        matchExpressions:
        - key: zone
          operator: In
          values:
          - us-central1-a
```

-Pod prefers nodes in zone us-central1-a, but can run elsewhere if needed.

--Operators for Node Selection

Node affinity supports several operators:

1) In: Label value is in the provided set
2) NotIn: Label value is not in the provided set
3) Exists: Label with the key exists (don't check value)
4) DoesNotExist: Label with the key doesn't exist
5) Gt: Label value is greater than the specified value (for integers)
6) Lt: Label value is less than the specified value (for integers)

--Node Affinity vs. Node Selector

Feature	**nodeSelector**	**nodeAffinity**
1. Simple equality matching	-Yes	-Yes
2. Multiple requirements	-No	-Yes
3. Soft preferences	-No	-Yes
4. Advanced operators	-No	-Yes
5. Weighted preferences	-No	-Yes

--Best Practices

1. Label nodes meaningfully: Use labels like env=prod, zone=east, instance type=highmem
2. Combine with taints/tolerations: For more complex scheduling scenarios
3. Use weights carefully: Higher weights mean stronger preference (range 1-100)
4. Test affinity rules: Before deploying to production
5. Monitor unschedulable pods: Especially with required affinity rules

--Label Nodes First

To use node affinity, your nodes must have appropriate labels:

```
# kubectl label nodes node1 node-type=high-memory
# kubectl label nodes node2 zone=us-central1-a
```

--Operators in matchExpressions

Operator	**Meaning**
-In	-Key must have one of the listed values
-NotIn	-Key must not have listed values
-Exists	-Key must exist
-DoesNotExist	-Key must not exist

--Summary

Feature	**Description**
1.Node Affinity	-Schedule pods based on node labels
2.Hard Rule	-requiredDuringSchedulingIgnoredDuringExecution
3.Soft Rule	-preferredDuringSchedulingIgnoredDuringExecution
4.Uses	-Performance tuning, hardware requirements, availability zones
5.Labels	-Must be applied to nodes manually or via automation

------Taints & Tolerations in Kubernetes

Taints and tolerations work together in Kubernetes to control which pods can be scheduled on which nodes.

--What is a Taint?

A taint is applied to a node and tells Kubernetes:
"Don't schedule any pods here unless they tolerate this taint."

-It's a way to repel pods from certain nodes unless they meet specific conditions.

-Taint Format: <key>=<value>:<effect>

-Key: A label key (e.g. env)
-Value: A label value (e.g. production)
-Effect: What happens if a pod doesn't tolerate the taint:

-Effect	**-Description**
1. NoSchedule	-Pod won't be scheduled on this node unless it tolerates the taint
2. PreferNoSchedule	- Try to avoid scheduling pods here, but not guaranteed
3. NoExecute	-Existing pods are evicted unless they tolerate it

-Add a taint to a node:

```
# kubectl taint nodes node1 key=value:NoSchedule
```
This means: Only pods with a matching toleration can run on node1.

--What is a Toleration?

A toleration is applied to a pod and says:
"I'm okay with this taint, you can schedule me there."

-Toleration Example in Pod YAML:

```
tolerations:
- key: "env"
  operator: "Equal"
  value: "production"
  effect: "NoSchedule"
```

-This pod can be scheduled on any node with the taint env=production:NoSchedule.

--Summary

-Concept	**-Used on**	**-Purpose**
Taint	Node	Prevent pods from scheduling unless tolerated
Toleration	Pod	Allows pod to run on tainted nodes

--Taints vs. Node Affinity

-Feature	**Taints & Tolerations**	**Node Affinity**
Purpose	Restrict Pods from nodes	Attract Pods to nodes
Behavior	"Avoid unless tolerated"	"Prefer/Require matching nodes"
Use Case	Dedicated nodes, isolation	Optimizing workload placement

--Full Example:

➤ **Taint a node:**

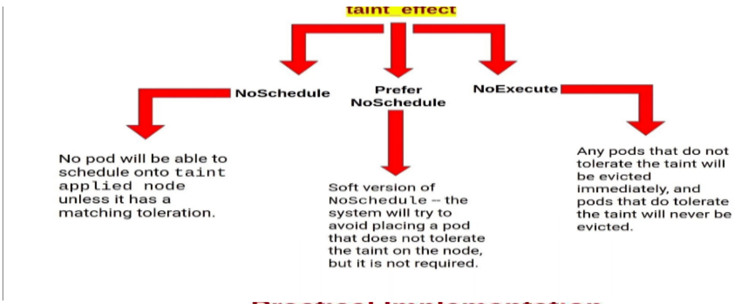
```
#kubectl taint nodes node1 env=production:NoSchedule
```

➤ **Pod toleration YAML:**

```
apiVersion: v1
kind: Pod
metadata:
  name: toleration-pod
spec:
  containers:
  - name: nginx
    image: nginx
  tolerations:
  - key: "env"
    operator: "Equal"
    value: "production"
    effect: "NoSchedule"
```

--Summary

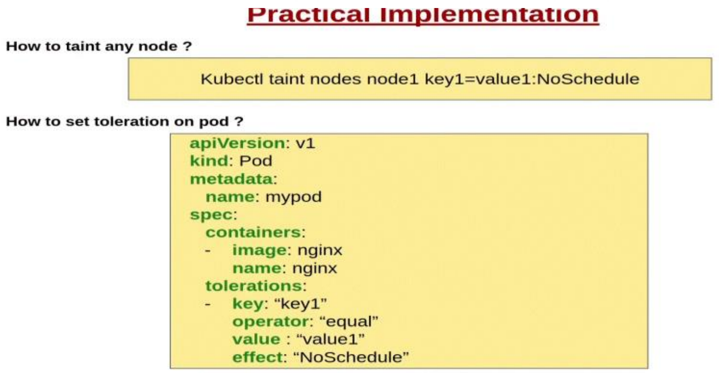
✅ Taints repel Pods from nodes unless they have a matching toleration.
✅ Tolerations allow Pods to run on tainted nodes.
✅ Useful for dedicated nodes, master isolation, and maintenance.



No pod will be able to schedule onto taint applied node unless it has a matching toleration.

Soft version of NoSchedule -- the system will try to avoid placing a pod that does not tolerate the taint on the node, but it is not required.

Any pods that do not tolerate the taint will be evicted immediately, and pods that do tolerate the taint will never be evicted.



Practical Implementation

How to taint any node ?

```
Kubectl taint nodes node1 key1=value1:NoSchedule
```

How to set toleration on pod ?

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  -   image: nginx
      name: nginx
  tolerations:
  -   key: “key1”
      operator: “equal”
      value : “value1”
      effect: “NoSchedule”
```

------Ingress Controller in Kubernetes?

An Ingress Controller is a specialized Kubernetes component that manages access to your services from external clients, typically HTTP or HTTPS traffic. It works in conjunction with an Ingress resource, which defines routing rules for external traffic to internal services.

An Ingress Controller in Kubernetes is a specialized load balancer that manages external access to services in a cluster, typically HTTP/HTTPS traffic. It implements the rules defined in Ingress resources to route traffic to different services based on the hostname or path.

--Key Concepts:

Component	**Description**
1. Ingress	-A Kubernetes object that defines routing rules for HTTP/S traffic.
2. Ingress Controller	-A pod or deployment that implements those rules (e.g., NGINX, Traefik).
3. Service	-Exposes your app internally to the cluster (ClusterIP)

1. Ingress Resource: Defines routing rules (hosts, paths, TLS, etc.)
2. Ingress Controller: The actual implementation that processes these rules (Nginx, Traefik, ALB, etc.)

3. Benefits:
   - -Single entry point for multiple services
   - -Path-based routing
   - -Host-based routing
   - -TLS termination
   - -Load balancing

--Popular Ingress Controllers:

1. NGINX Ingress Controller ✅ (most widely used)
2. Traefik
3. HAProxy
4. Istio Gateway

◆ Common Ingress Controllers

In AWS-based Kubernetes clusters (like **EKS**), the two most common ingress controllers are:

1. **NGINX Ingress Controller**
   - Uses an **AWS Classic Load Balancer (CLB)** or **Network Load Balancer (NLB)** under the hood.
   - Deploys a pod (nginx-ingress-controller) that listens for ingress events.
   - **Typical Flow :** Client → AWS NLB/CLB → NGINX Ingress Pod → Kubernetes Service → Pods
2. **AWS Load Balancer Controller (ALB Ingress Controller)**
   - Creates and manages an **Application Load Balancer (ALB)** directly in AWS.
   - This is the preferred and AWS-native way for EKS clusters.
   - **Typical Flow :** Client → AWS ALB → Target Group (Pods as targets) → Pods directly

---

◆ What AWS Resources Are Created When You Deploy an Ingress Controller

When you deploy and configure an ingress controller (like AWS Load Balancer Controller), the following AWS resources are automatically created:

Resource Type	Description
Elastic Load Balancer (ALB or NLB)	-Handles external traffic. Yes — a **Load Balancer is created** automatically when you create an Ingress resource.
Target Groups	-Created for each Kubernetes Service (the backend). Pods get registered as targets.
Listeners & Listener Rules	-ALB listeners (port 80/443) and rules that match ingress path/host to target groups.
Security Groups	-For the ALB — allows inbound traffic from internet and outbound to cluster nodes.
IAM Roles/Policies	-The controller pod (via IAM role for service account) needs permission to create and manage these AWS resources.

◆ Example Flow (AWS Load Balancer Controller)

1. You deploy the **AWS Load Balancer Controller** in your EKS cluster.
2. You create an **Ingress** resource like this:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-ingress
  annotations:
    kubernetes.io/ingress.class: alb
spec:
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: myapp-service
            port:
              number: 80
```

3. The controller sees this and automatically:
   - Provisions an **Application Load Balancer** in your AWS account.
   - Creates **Target Groups** for each backend service.
   - Adds **Listener Rules** to route / → myapp-service.

✅ **So yes — a Load Balancer (ALB or NLB) *is created automatically*** when you create an Ingress (depending on which ingress controller you use).

◆ Summary

Concept	Description
Ingress Controller	-Kubernetes component that manages external traffic routing.
Ingress Resource	-YAML object that defines routing rules.
AWS Resources Created	-ALB/NLB, Target Groups, Listeners, Security Groups, IAM roles.
Load Balancer Created?	-Yes — automatically by the ingress controller (ALB or NLB).

-----Ingress Controller setup using kind (Kubernetes IN Docker).

Since kind runs Kubernetes clusters in Docker containers, we need extra config to expose the Ingress Controller properly.

--Step-by-Step: Ingress with 2 Services in kind

1. Create a kind cluster with Ingress support

```
kind-config.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: ingress-demo
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
  - containerPort: 443
    hostPort: 443
```

Create the cluster:

```
# kind create cluster --config kind-config.yaml
```

2. Install NGINX Ingress Controller

-Use the official manifest:

```
# kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.10.0/deploy/static/provider/kind/deploy.yaml
```

-Wait until all pods are ready:

```
# kubectl get pods -n ingress-nginx
```

3. Deploy Sample Apps (NGINX + HTTPD)

a) nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

b) httpd-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpd-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      containers:
      - name: httpd
        image: httpd:latest
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: httpd-service
spec:
  selector:
    app: httpd
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

-Apply both:

```
# kubectl apply -f nginx-deployment.yaml
# kubectl apply -f httpd-deployment.yaml
```

4. Create Ingress Resource
ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: myapp.local
    http:
      paths:
      - path: /nginx
        pathType: Prefix
        backend:
          service:
            name: nginx-service
            port:
              number: 80
      - path: /httpd
        pathType: Prefix
        backend:
          service:
            name: httpd-service
            port:
              number: 80
```

Apply: # kubectl apply -f ingress.yaml

5.Access Services via Browser or Curl

Edit your /etc/hosts: 127.0.0.1 myapp.local

Then open:

```
http://myapp.local/nginx
http://myapp.local/httpd
```

Or use curl:

```
curl http://myapp.local/nginx
curl http://myapp.local/httpd
```

-Summary: Task	**Command**
1. Create kind cluster	kind create cluster --config kind-config.yaml
2. Install Ingress Controller	kubectl apply -f <nginx-manifest>
3. Deploy NGINX & HTTPD	kubectl apply -f *.yaml
4. Add myapp.local to /etc/hosts	127.0.0.1 myapp.local
5. Test	curl http://myapp.local/nginx

-----Comparison of Ingress Controller, Load Balancer, and NodePort in Kubernetes, along with their use cases:

1. NodePort

What it is:
-Exposes a service on a static port (default range: 30000-32767) on each worker Node's IP.
-Traffic comes to any Node on this port and gets routed to the service.

How it works:
-Kubernetes assigns a high-numbered port (e.g., 31000).
-External users access the service via http://<NodeIP>:<NodePort>.

Pros:
✅ Simple to set up (good for development/testing).
✅ Works without a cloud provider.

Cons:
❌ Requires manual firewall rules.
❌ Exposes raw ports (not user-friendly).
❌ Not scalable for production.

Example Use Case:
-Temporary access to a service in a local/minikube cluster.

2. LoadBalancer (Service Type)

What it is:
-Automatically provisions an external cloud load balancer (AWS ALB, GCP LB, Azure LB).
-Routes traffic to Kubernetes Services (usually backed by Pods).

How it works:
-Cloud provider assigns a public IP/DNS.
-Traffic goes:
User → Cloud LB → NodePort → Service → Pods

Pros:
✅ Fully managed by the cloud provider.
✅ Handles SSL termination, health checks.
✅ Good for exposing a single service directly.

Cons:
❌ Costly (each LoadBalancer service creates a new LB).
❌ Cloud-specific (won't work in bare-metal clusters).

Example Use Case:
-Exposing a critical service (e.g., a public API) that needs a dedicated IP.

3. Ingress Controller

What it is:
-A reverse proxy (like Nginx, Traefik, AWS ALB Ingress) that manages external access to services.
-Uses rules to route traffic based on hostnames/paths (e.g., /api, /app).

How it works:
-You deploy an Ingress Controller (e.g., Nginx, AWS ALB).
-Define Ingress rules (YAML) to route traffic.

External traffic goes:
User → Ingress Controller → Service → Pods

Pros:
✅ Path/host-based routing (e.g., example.com/api → api-service).
✅ Single entrypoint for multiple services.
✅ Supports TLS termination, rate limiting, rewrites.
✅ Works with cloud or self-managed LBs.

Cons:
❌ Requires setup (not enabled by default).
❌ Slightly complex configuration.

Example Use Case:
-Hosting multiple microservices under one domain (e.g., app.example.com, api.example.com).

--Comparison Table

Feature	NodePort	LoadBalancer	Ingress Controller
Exposes	Static port	Dedicated LB	Routes via rules
Cloud Required?	❌ No	✅ Yes	Optional
Cost	Free	$$$ (per LB)	$ (shared LB)
Use Case	Dev/Testing	Single service	Multiple services
SSL Support	Manual	Yes	Built-in
Scalability	Poor	Good	Excellent

--Which One to Use?
-Development? → NodePort (quick testing).
-Single production service? → LoadBalancer.
-Multiple services under one domain? → Ingress Controller (with a LoadBalancer or NodePort backing it).

-----Annotations for Ingress in Kubernetes

Ingress Annotations in Kubernetes are used to customize the behavior of the Ingress Controller (like NGINX, Traefik, HAProxy, etc.). They are key-value pairs added to the Ingress object's metadata to enable features like:

-URL rewriting
-TLS redirection
-Rate limiting
-CORS configuration
-Load balancing

--Basic Ingress Example with Annotations

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  rules:
    - host: myapp.local
      http:
        paths:
          - path: /nginx
            pathType: Prefix
            backend:
              service:
                name: nginx-service
                port:
                  number: 80
```

--Common NGINX Ingress Annotations

-Annotation	**-Purpose**
~nginx.ingress.kubernetes.io/rewrite-target	-Rewrites the request URL (e.g. /nginx → /)
~nginx.ingress.kubernetes.io/ssl-redirect	-Forces HTTPS (true or false)
~nginx.ingress.kubernetes.io/force-ssl-redirect	-Enforce HTTPS even without TLS config
~nginx.ingress.kubernetes.io/whitelist-source-range	-Restrict access to specific IP ranges
~nginx.ingress.kubernetes.io/configuration-snippet	-Add custom NGINX directives
~nginx.ingress.kubernetes.io/proxy-body-size	-Set max allowed size for client requests
~nginx.ingress.kubernetes.io/enable-cors	-Enable CORS
~nginx.ingress.kubernetes.io/cors-allow-origin	-Allowed origins for CORS

--Example: Path Rewriting

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

-Requests to /api get rewritten to /
-Example: /api/login → /login

--Example: Force HTTPS

annotations:
```
  nginx.ingress.kubernetes.io/ssl-redirect: "true"
```

-Forces clients to use https:// instead of http://

--Example: Restrict IP Access

annotations:
```
  nginx.ingress.kubernetes.io/whitelist-source-range: "192.168.0.0/24"
```

-Only clients from 192.168.0.x can access the app

--Important Notes

-Annotations are controller-specific — they vary between NGINX, Traefik, HAProxy, etc.
-Always check the docs for your Ingress Controller.
-NGINX: https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/annotations/

--Summary

-Feature	**-Description**
-Annotations	Extend/customize Ingress behavior
-Use Cases	Path rewrite, HTTPS redirect, CORS, rate limiting
-Scope	Only valid for the Ingress Controller you're using
-Location	Placed under metadata.annotations in Ingress YAML

◆ Summary Table – SLA, SLO, SLI

Term	Full Form	Audience	Binding	Example	Relationship
SLA	Service Level Agreement	Customer-facing	✅ Legal/Contractual	99.9% uptime guaranteed	Agreement
SLO	Service Level Objective	Internal team	❌ Not legal	99.95% uptime target	Goal
SLI	Service Level Indicator	Internal metric	❌ Not legal	99.96% successful requests	Measurement

◆ Visual Relationship

SLI → measures → SLO → supports → SLA

Example in words:

-You **measure** uptime using an **SLI**,
-you **target** 99.95% uptime as your **SLO**,
-and you **promise** 99.9% uptime in your **SLA**.

-----Probes in kubernetes

In Kubernetes, probes are used by the kubelet to check the health and status of containers running in a Pod. There are three types of probes:

1. Liveness Probe
-Determines if a container is still running properly.
-If the probe fails, Kubernetes restarts the container.
-Useful for detecting deadlocks or frozen applications.

-Example: A web server becomes unresponsive but the process is still running.

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 5       # Wait 5 sec before first check
  periodSeconds: 10            # Check every 10 sec
```

2. Readiness Probe
-Determines if a container is ready to accept traffic.
-If the probe fails, Kubernetes removes the Pod from Service endpoints (no traffic is sent).
-Useful during application startup or when temporarily unable to serve requests.

-Example: A container needs time to load data before serving requests.

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 10          #Wait 10 sec before first check
  periodSeconds: 5                 # Check every 5 sec
```

3. Startup Probe
-Used for slow-starting containers (e.g., legacy apps).
-Kubernetes waits until this succeeds before applying liveness/readiness checks.
-Once successful, it disables liveness and readiness checks until the container starts.

-Example: A Java app that takes 30 seconds to start.

```
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  failureThreshold: 30       # Allow up to 30 checks
  periodSeconds: 5           # Check every 5 sec (total: 150 sec max)
```

--Example with All Probes

```
containers:
- name: my-app
  image: my-app:v1
  ports:
    - containerPort: 8080
  livenessProbe:
    httpGet:
      path: /health
      port: 8080
  readinessProbe:
    httpGet:
      path: /ready
      port: 8080
  startupProbe:
    httpGet:
      path: /init
      port: 8080
    failureThreshold: 30
    periodSeconds: 5
```

--Probe Methods (Ways to Check Health)

Kubernetes supports 3 methods to perform probes:

-Method	-Description	-Example
httpGet	-Sends an HTTP GET request to a path/port	-/healthz on port 8080
tcpSocket	-Tries to open a TCP connection on a port	-Checks port 3306 for MySQL
exec	-Executes a command inside the container	-["cat", "/tmp/healthy"]

--Probe Configuration Options

-Parameter	-Description
1. initialDelaySeconds	-Time to wait before first probe
2. periodSeconds	-How often to perform the probe
3. timeoutSeconds	-Time before probe times out
4. failureThreshold	-When to consider the container failed
5. successThreshold	-Consecutive successes needed to pass

--Summary

-robe Type	-Use Case	-Action on Failure
1. Liveness	App is stuck or dead	Restart container
2. Readiness	App not ready to serve	Remove from service
3. Startup	App takes time to start	Wait before other probes

-----Autoscaling in Kubernetes

Autoscaling in Kubernetes is a mechanism that automatically adjusts resources (like number of pods or CPU/memory requests) based on the current workload. It improves performance, cost-efficiency, and resource utilization.

-There are three main types of autoscaling in Kubernetes:

-Type	**-Scales**	**-Purpose**
1. Horizontal Pod Autoscaler (HPA)	Number of pods	Based on CPU, memory, or custom metrics
2. Vertical Pod Autoscaler (VPA)	CPU/Memory of pod	Adjusts resource requests/limits of pods
3. Cluster Autoscaler	Number of nodes	Adds/removes nodes in the cluster

A) Horizontal Pod Autoscaler (HPA)

HPA automatically increases or decreases the number of pod replicas in a deployment, statefulset, or replicaset based on resource usage.

-Use Cases:
1. Web apps with variable traffic
2. APIs with unpredictable load

-How it works:
1. Monitors metrics like CPU, memory (using Metrics Server), or custom metrics
2. Scales pod count up or down to match demand

-Sample 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
```

This means: If average CPU usage > 60%, Kubernetes will scale pods between 2 and 10 replicas.

-Enable Metrics Server (required for HPA):
```
# kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

B) Vertical Pod Autoscaler (VPA)

VPA automatically adjusts the CPU and memory requests/limits of containers in a pod based on actual usage.

-Use Cases:
1. Background jobs
2. Apps with steady but changing resource needs

-How it works:
1. Recommends or sets new resource requests/limits
2. Might restart pods to apply changes (depending on mode)

--VPA Modes:

-Mode	**-Behavior**
Off	Only gives recommendations
Initial	Applies recommended values at pod creation
Auto	Continuously updates and can evict pods

-Sample VPA YAML:

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

This means: VPA will monitor resource usage and automatically update pod reuests/limits and restart pods if needed.

C) Cluster Autoscaling (CA)

1. Automatically adds/removes nodes when:
-Pods fail to schedule due to insufficient resources.
-Nodes are underutilized.

2. Works with cloud providers (AWS, GKE, Azure).
-Example: GKE Cluster Autoscaler

```
gcloud container clusters create my-cluster \
  --enable-autoscaling \
  --min-nodes=1 \
  --max-nodes=5
```

-If Pods need more resources, GKE adds nodes.
-If nodes are underused, it removes them.

Summary Table

-Feature	**-HPA**	**-VPA**
1. Scales	Number of pod replicas	Resource (CPU/memory) per pod
2. Use case	Apps with varying load	Apps with changing resource needs
3. Needs Metrics Server	✓ Yes	✓ Yes (plus VPA admission controller)
4. Can restart pods?	✕ No	✓ Yes (to apply new resources)
5. Can be used together?	⚠ Yes, but with caution	Best with custom configs

--Best Practices

1. Use HPA for stateless workloads (scaling replicas is safer).
2. Use VPA for batch jobs (but avoid with StatefulSets).
3. Combine HPA + Cluster Autoscaler for full elasticity.
4. Monitor scaling behavior to avoid thrashing (rapid scaling up/down).

-----Event-Driven Architecture (EDA) in Microservices

is a design pattern where services communicate and react to **events** instead of making direct synchronous calls (like REST API calls).

An **event** is a record of something that happened in the system (e.g., *"Order Placed"*, *"Payment Successful"*, *"Inventory Updated"*).

🔑 **Key Concepts:**

1. **Event Producer**
   - A service that *publishes* an event when something happens.
   - Example: Order Service publishes an event "OrderCreated" after a new order is placed.
2. **Event Broker / Message Queue**
   - Middleware that transports events from producers to consumers.
   - Examples: Kafka, RabbitMQ, AWS SNS/SQS.
3. **Event Consumer**
   - A service that *subscribes* to specific events and performs actions.
   - Example: Inventory Service listens to "OrderCreated" events and reduces stock.

⚙ **How it Works (Flow)**

1. User places an order → **Order Service** saves it and publishes "OrderCreated" event.
2. **Event Broker** (like Kafka) receives the event.
3. **Payment Service** subscribes → processes payment.
4. **Inventory Service** subscribes → updates stock.
5. **Notification Service** subscribes → sends confirmation email.

All services act **independently** but remain consistent via events.

✓ **Benefits**

- **Loose Coupling** → Producers don't need to know consumers.
- **Scalability** → Consumers can scale independently.
- **Asynchronous** → Improves responsiveness, avoids blocking calls.
- **Flexibility** → New services can subscribe without changing producers.

☐ **Event-Driven Architecture in Microservices** → Services communicate by publishing/subscribing to **events** via an event broker instead of direct calls.
☐ **Application** → Real-time systems like **e-commerce orders, payment processing, IoT, fraud detection, stock trading**.
☐ **Advantages** → Loose coupling, scalability, flexibility, real-time responsiveness.
☐ **Disadvantages** → Complex debugging, eventual consistency, event ordering issues, harder monitoring.
☐ **Where it's used now** → Amazon, Netflix, Uber, PayPal, LinkedIn, and modern **cloud-native apps (Kafka, AWS SNS/SQS, Azure Event Hub, GCP Pub/Sub)**.

☞ In short: **Event-driven architecture makes microservices communicate through events instead of direct calls, leading to loosely coupled** (Instead of Tightly Coupled), **scalable, and resilient systems.**

-------KEDA in Kubernetes?

KEDA (Kubernetes Event-Driven Autoscaling) is an open-source component that allows Kubernetes to scale pods based on external events or custom metrics, not just CPU/memory like the default HPA.

-Why Use KEDA?

KEDA is useful when:

1. Your application needs to scale based on external systems (like Kafka, RabbitMQ, Prometheus, Azure Queue, AWS SQS, etc.).
2. You need event-based autoscaling, not just resource-based scaling.
3. You want scale-to-zero capability — saving cost by scaling down to 0 pods.

-How KEDA Works

1. Scaler: Defines a trigger (e.g., queue length in Kafka or number of HTTP requests).
2. ScaledObject: Defines what to scale and when.
3. KEDA monitors the trigger and creates/deletes HPA behind the scenes.
4. When trigger value crosses the threshold, pods are scaled up/down.

-Components of KEDA

-Component	**-Description**
1. ScaledObject	Custom resource that links your deployment to a scaler
2. TriggerAuthentication	(Optional) Auth info for accessing external systems
3. ScaledJob	For event-driven batch jobs (not deployments)
4. Scalers	Plugins for external systems (Kafka, RabbitMQ, etc.)

--Example: KEDA ScaledObject for Kafka

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: kafka-scaler
spec:
  scaleTargetRef:
    name: kafka-consumer-deployment
  minReplicaCount: 0
  maxReplicaCount: 10
  triggers:
  - type: kafka
    metadata:
      bootstrapServers: my-kafka:9092
      topic: my-topic
      consumerGroup: my-group
      lagThreshold: "50"
```

-This will scale the deployment kafka-consumer-deployment between 0 and 10 replicas based on Kafka lag.

--Key Features of KEDA

-Feature	**-Description**
1. Event-driven	- Scale based on events like queue length, database records, etc.
2. Scale to zero	-Saves resources when idle
3. 50+ scalers	-Supports Kafka, RabbitMQ, Redis, Prometheus, AWS SQS, Azure, GCP, etc.
4. Works with HPA	-Creates HPA objects under the hood
5. Fine-grained control	-Cooldown period, polling intervals, activation thresholds

--Example Use Cases

-Use Case	**-Trigger**	**-Benefit**
1. Video processing	Messages in RabbitMQ	Auto-scale worke
2. collection	Prometheus alert	Scale up monitoring agents
3. E-commerce	Redis queue length	Auto-scale order processing

--Summary

-Feature	**-HPA (default)**	**-KEDA**
1. Based on	CPU, memory	Events, queues, custom metrics
2. Scale to zero	❌ Not supported	✅ Yes
3. External integrations	❌ Limited	✅ 50+ systems supported
4. Setup complexity	Simple	Moderate

--To use KEDA, you need to install the KEDA operator:

```
#kubectl apply -f
https://github.com/kedacore/keda/releases/latest/download/keda.yaml
```

--Best Practices

1. Use KEDA for event-driven workloads (e.g., queues, streams).
2. Combine with HPA for hybrid scaling (CPU + events).
3. Monitor scaling behavior to avoid thrashing.

------HPA_VPA

----Apache-Deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
  labels:
    app: apache
spec:
  replicas: 1
  selector:         #Ensures the Deployment manages pods with label app: apache
    matchLabels:
      app: apache
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
      - name: apache
        image: httpd:2.4
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 200m
            memory: 256Mi
---
apiVersion: v1
kind: Service
metadata:
  name: apache-service
  labels:
    app: apache
spec:
  selector:
    app: apache
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

----This YAML:

-Creates a Deployment named apache-deployment.
-Manages 1 Pod running the Apache HTTP Server (httpd:2.4).
-Applies labels for selection and organization.
-Configures resource limits and requests for performance and cluster health.
-Ensures that port 80 is exposed from the container.
-Automatically replaces the Pod if it crashes.

1) apache-hpa.yml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: apache-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: apache-deployment
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 5
```

----This YAML:
-Creates an HPA named apache-hpa.
-Monitors CPU utilization of the apache-deployment.
-Scales Pods between 1 and 5 replicas.
-Tries to maintain average CPU utilization at 5% of the requested CPU (100m in the deployment).

2) apache-vpa.yml

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: apache-vpa
  namespace: default
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: apache-deployment
  updatePolicy:
    updateMode: "Auto" # Options: "Off", "Initial", "Auto"
```

----This YAML:
-Creates a Vertical Pod Autoscaler named apache-vpa.
-Targets the apache-deployment in the default namespace.
-Automatically adjusts CPU and memory requests/limits for Pods based on usage.
-Is set to "Auto" mode, so it can update resource values dynamically during runtime by evicting and restarting Pods.

```yaml
# Apache-Deployment Enhanced Version with Best Practice

apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
  labels:
    app: apache
    tier: web
    environment: production
spec:
  replicas: 3                         # Increased for high availability
  revisionHistoryLimit: 3            # Limits stored revisions for rollback
  strategy:
    type: RollingUpdate              # Default, but explicit is better
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels:
      app: apache
  template:
    metadata:
      labels:
        app: apache
        tier: web
        environment: production
      annotations:
        prometheus.io/scrape: "true"      # For monitoring
        prometheus.io/port: "80"
    spec:
      securityContext:                    # Added security
        runAsNonRoot: true
        runAsUser: 1000
        fsGroup: 2000
      containers:
      - name: apache
        image: httpd:2.4.57-alpine        # More lightweight image
        imagePullPolicy: IfNotPresent
        ports:
        - name: http
          containerPort: 80
          protocol: TCP
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 200m
            memory: 256Mi
        livenessProbe:                    # Health checking
          httpGet:
```

```
          path: /
          port: 80
        initialDelaySeconds: 30
        periodSeconds: 10
      readinessProbe:
        httpGet:
          path: /
          port: 80
        initialDelaySeconds: 5
        periodSeconds: 5
      volumeMounts:
      - name: apache-config
        mountPath: /usr/local/apache2/conf/
        readOnly: true
    volumes:                                    # Example volume
    - name: apache-config
      configMap:
        name: apache-config
```

--Key Improvements Made:
1. High Availability: Increased replicas to 3
2. Rollout Strategy: Added explicit rolling update configuration
3. Security: Added security context to run as non-root
4. Monitoring: Added annotations for Prometheus scraping
5. Health Checks: Added liveness and readiness probes
6. Resource Management: Kept resource limits but with better practices
7. Versioning: Used specific image tag (2.4.57-alpine)
8. Configuration: Added example ConfigMap volume mount
9. Labels/Annotations: Added more metadata for better organization
10. Port Definition: Named the port and added protocol

# Apache-Service Enhanced Version with Best Practice

```
apiVersion: v1
kind: Service
metadata:
  name: apache-service
  namespace: default                # Explicit namespace
  labels:
    app: apache
    tier: web
    environment: production         # Environment tag
  annotations:
    prometheus.io/scrape: "true"    # Enable metrics scraping
    prometheus.io/port: "80"
spec:
  selector:
    matchLabels:
      app: apache
      tier: web                   # More specific selector
  ports:
    - name: http              # Named port for clarity
      protocol: TCP
      port: 80
      targetPort: 80          # Matches containerPort # nodePort:30080
                              # Uncomment for NodePort type
  type: ClusterIP             # Options: ClusterIP, NodePort, LoadBalancer
  sessionAffinity: ClientIP                 # Optional session persistence
  ipFamilyPolicy: PreferDualStack           # IPv4/IPv6 support
  internalTrafficPolicy: Cluster            # Options: Cluster, Local
```

------HPA-Enhanced Version with Best Practices:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: apache-hpa
  namespace: default                          # Explicit namespace
  labels:
    app: apache
    component: autoscaler
    environment: production
  annotations:
    description: "HPA for Apache deployment with CPU scaling"
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: apache-deployment
  minReplicas: 2                             # Minimum 2 for high availability
  maxReplicas: 10                            # Maximum allowed pods
  behavior:                                  # Advanced scaling control
    scaleDown:
      stabilizationWindowSeconds: 300  # 5 min cooldown
      policies:
      - type: Percent
        value: 10                           # Max 10% reduction at once
        periodSeconds: 60
    scaleUp:
      stabilizationWindowSeconds: 60  # 1 min cooldown
      policies:
      - type: Percent
        value: 100                          # Double pods if needed
        periodSeconds: 15
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50            # Target 50% CPU usage
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 70           # Target 70% memory usage
  - type: External                       # Optional custom metrics
    external:
      metric:
        name: requests_per_second
        selector:
          matchLabels:
            app: apache
      target:
        type: AverageValue
        averageValue: 1000               # Target 1000 RPS
```

------VPA-Enhanced Version with Best Practices:

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: apache-vpa
  namespace: default
  labels:
    app: apache
    component: autoscaler
    environment: production
  annotations:
    description: "VPA for Apache deployment with automatic resource
tuning"
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: apache-deployment
  updatePolicy:
    updateMode: "Auto"       # Options: "Off", "Initial", "Recreate", "Auto"
    minReplicas: 2           # Minimum pods to maintain during updates
  resourcePolicy:
    containerPolicies:
    - containerName: "*"      # Applies to all containers
      minAllowed:
        cpu: "100m"
        memory: "128Mi"
      maxAllowed:
        cpu: "2"
        memory: "4Gi"
      controlledResources: ["cpu", "memory"]
      controlledValues: "RequestsAndLimits"
                              # Options: "RequestsOnly", "RequestsAndLimits"
  recommendation:
    updatePeriodSeconds: 3600          # Refresh recommendations hourly
    recommendationHistoryLimit: 24       # Keep 24 hours of history
```

-Summary Table

updateMode	Automatically /Updates Pods?	Applies on Pod Creation?	Use Case
Off	✕ No	✕ No	Manual tuning/monitoring only
Initial	✕ No	✓ Yes	Safe, one-time tuning per Pod
Auto	✓ Yes (evicts Pods)	✓ Yes	-Continuous optimization (careful in prod)

--------Headless Service in Kubernetes

A Headless Service in Kubernetes is a special type of Service that does not have its own cluster IP. Instead of acting as a single stable entry point (like a normal Service), it directly returns the IP addresses of the individual Pods that match the Service's selector.

--How it Works

-You create a Service with:

```
spec:     clusterIP: None
```

-Kubernetes will skip creating a ClusterIP.
-When you query the Service's DNS name:
Instead of getting one IP (the Service IP), you get multiple Pod IPs.
The client can then connect directly to any Pod.

--Why Use a Headless Service?

1) Direct Pod-to-Pod communication
Useful when you want the client to control load balancing instead of Kubernetes.

2) Stateful applications (like databases)
Often used with StatefulSets, where each Pod has a stable DNS name.
Example: mysql-0.mydb.default.svc.cluster.local

3) Custom service discovery
Some applications need to discover all backend Pod IPs to maintain a cluster.

--Example YAML

```
apiVersion: v1
kind: Service
metadata:
  name: my-headless-service
spec:
  clusterIP: None                    # Makes it headless
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 8080
```

--DNS Behavior

1) Normal Service:
my-service.default.svc.cluster.local → 10.96.12.34 (ClusterIP)

2) Headless Service:
my-headless-service.default.svc.cluster.local → PodIP1, PodIP2, PodIP3

--When Not to Use It

1. If you need Kubernetes load balancing, don't make it headless — use a normal Service.
2. If clients don't know how to handle multiple IPs, they may fail to connect.

--When to Use It

1. Stateful Applications
2. Peer Discovery
3. Custom Load Balancing
4. Service Mesh Integration

------Deployment Strategies: Explanation and Code Examples

Deployment strategies define how you roll out new versions of your application to production environments with minimal downtime and risk. Here are the most common deployment strategies with code examples.

1. Recreate Deployment (Big Bang)

-Description: Version A is terminated completely, then version B is deployed. This causes downtime but is simple to implement.

-When to use: For non-critical applications that can tolerate downtime, during maintenance windows.

```
# Kubernetes Recreate Deployment Example
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myapp:v2
        ports:
        - containerPort: 8080
```

2. Rolling Update (Incremental)

-Description: Version B is gradually rolled out while version A is scaled down. No downtime occurs.

-When to use: For most stateless applications where brief version coexistence is acceptable.

```
# Kubernetes Rolling Update Example
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-rolling
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1              # Extra pods over desired count
      maxUnavailable: 1        # Max pods unavailable during update
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: myapp:v2
```

3. Blue-Green Deployment

-Description: Version A (blue = old) runs alongside version B (green = new). Once testing is complete, traffic is switched to green.
-When to use: When you need to minimize risk and have the infrastructure to run two identical environments.

```
# Kubernetes Blue-Green Example (simplified)

# Blue Deployment (current)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-blue
spec:
  replicas: 3
  selector:
    matchLabels:
      version: blue
  template:
    metadata:
      labels:
        version: blue
    spec:
      containers:
      - name: webapp
        image: myapp:v1

# Green Deployment (new)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-green
spec:
  replicas: 3
  selector:
    matchLabels:
      version: green
  template:
    metadata:
      labels:
        version: green
    spec:
      containers:
      - name: webapp
        image: myapp:v2

# Service switching between them
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  selector:
    version: green                # Switch this to blue/green accordingly
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

4.Canary Deployment- (By Changing Replica Count in Deployment)

-Description: Version B is deployed to a small subset of users, then gradually rolled out to everyone.

-Steps: 1.Deploy a small replica with the new version. (10%)
2.Route partial traffic. (50%)
3.Gradually increase if stable. (100%)

-When to use: For testing new versions with real users before full rollout.

# Kubernetes Canary Example using two Deployments

# Stable deployment (90% traffic)
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-stable
spec:
  replicas: 9
  selector:
    matchLabels:
      app: myapp
      track: stable
  template:
    metadata:
      labels:
        app: myapp
        track: stable
    spec:
      containers:
      - name: myapp
        image: myapp:v1
        ports:
        - containerPort: 8080
```

# Canary deployment (10% traffic)
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-canary
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myapp
      track: canary
  template:
    metadata:
      labels:
        app: myapp
        track: canary
    spec:
      containers:
      - name: myapp
        image: myapp:v2
        ports:
            - containerPort: 8080
```

# Service with both deployments

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

5. A/B Testing Deployment

-Description: Different versions are served to different users based on specific criteria (headers, cookies, etc.).
-When to use: For testing different UI versions or features with different user segments.

# Kubernetes A/B Testing using Istio VirtualService

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: myapp
spec:
  hosts:
  - myapp.example.com
  http:
  - match:
    - headers:
        cookie:
          regex: ".*group=alpha.*"
    route:
    - destination:
        host: myapp
        subset: v2
  - route:
    - destination:
        host: myapp
        subset: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: myapp
spec:
  host: myapp
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

6.Shadow Deployment (Mirroring)

-Description: Version B receives mirrored traffic from version A without affecting responses.
-When to use: For testing new versions under real traffic without impacting users.

# Kubernetes Shadow Deployment using Istio

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: myapp
spec:
  hosts:
  - myapp.example.com
  http:
  - route:
    - destination:
        host: myapp
        subset: v1
      weight: 100
    mirror:
      host: myapp
      subset: v2
    mirror_percent: 100
```

--Choosing the Right Strategy :

1. **Recreate:** Simple, downtime acceptable
2. **Rolling Update**: Minimal downtime, stateless apps
3. **Blue-Green:** Zero downtime, quick rollback
4. **Canary:** Gradual rollout, risk mitigation
5. **A/B Testing**: Feature testing with users
6. **Shadow:** Testing with production traffic

--Summary Table :

Strategy	**Downtime**	**Traffic Control**	**Tools Required**
-Recreate	Yes	No	None
-Rolling Update	No	Partial (auto)	None
-Blue/Green	No	Full switch	Manual Service
-Canary	No	Partial traffic	Ingress/Istio
-A/B Testing	No	Conditional	Ingress/Istio
-Shadow	No	Mirror traffic	Service Mesh

-Change App Version (Image): # kubectl set image deployment/online-shop online-shop=amitabhdevops/online_shop_without_footer -n recreate-ns

```
# kubectl apply -f recreate-deployment.yml
```

-----How to upgrade a cluster in kubernetes

Upgrading a Kubernetes cluster depends on whether it's managed (EKS, GKE, AKS) or self-managed (kubeadm, kops, etc.), but the high-level goal is the same:
-Upgrade control plane components first (API server, scheduler, controller manager, etc.)
-Upgrade worker nodes next (kubelet, kube-proxy)
-Maintain backward compatibility during the process so workloads stay online

1. Managed Kubernetes (EKS / GKE / AKS)
Example: AWS EKS

i. Upgrade Control Plane from AWS Console or CLI:
```
# aws eks update-cluster-version --name my-cluster --kubernetes-version 1.28
```

ii. Upgrade Node Groups:
-Create a new node group with the desired version
-Drain old nodes:
```
# kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data
```

iii. Delete the old node group after migration.
-Upgrade Add-ons (CoreDNS, kube-proxy, VPC CNI):
```
# aws eks update-addon --cluster-name my-cluster --addon-name coredns --addon-version <version>
```

2. Self-Managed Cluster (kubeadm)

Upgrade Control Plane Node(s):
1. Check versions:
```
# kubectl get nodes
# kubeadm version
```

2. Upgrade kubeadm:
```
# apt-get update && apt-get install -y kubeadm=<version>
```

3. Plan upgrade:
```
# kubeadm upgrade plan
```

4. Apply upgrade:
```
# kubeadm upgrade apply v1.X.X
```

5. Upgrade kubelet & kubectl:
```
# apt-get install -y kubelet=<version> kubectl=<version>
# systemctl restart kubelet
```

----Step-by-Step: How to Upgrade a Node in Kubernetes

Upgrading a node means upgrading the **kubelet** and **kube-proxy** running on that node to match the desired Kubernetes version. This is typically done after upgrading the **control plane (master)**.

☐ Step 1: Check the Current Cluster Version

```
# kubectl get nodes
# kubectl version --short
```

This shows the current Kubernetes version for both control plane and worker nodes.

☐ Step 2: Cordon the Node (Prevent New Pods)

```
# kubectl cordon <node-name>
```

Explanation:

- Marks the node as **unschedulable**.
- No **new pods** will be scheduled on this node.
- **Existing pods keep running** (not affected).

✓ Example:

```
# kubectl cordon worker-node1
```

Output:

```
node/worker-node1 cordoned
```

☐ Step 3: Drain the Node (Evict Running Pods)

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

Explanation:

- Safely evicts all **pods** from the node.
- Skips **DaemonSet** pods (since they automatically run on all nodes).
- Deletes temporary pod data from **emptyDir volumes**.

✓ Example:

```
kubectl drain worker-node1 --ignore-daemonsets --delete-emptydir-data
```

Result:

Pods will be gracefully terminated and rescheduled on other healthy nodes in the cluster.

☐ Step 4: SSH into the Node and Upgrade Components

If you're using a Linux-based node (e.g., Ubuntu), run:

```
# sudo apt-get update
# sudo apt-get install -y kubelet kubeadm kubectl
# sudo kubeadm upgrade node
# sudo systemctl daemon-reload
# sudo systemctl restart kubelet
```

This updates the node's **kubelet** and **kubeadm** components to the latest compatible version.

☐ **Step 5: Uncordon the Node (Re-enable Scheduling)**

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

Explanation:

- Marks the node as **schedulable** again.
- The scheduler can now place **new pods** on it.

✅ Example:

```
# kubectl uncordon worker-node1
```

Output:

```
node/worker-node1 uncordoned
```

☐ **Step 6: Verify the Upgrade**

```
# kubectl get nodes
```

You should see the node running the **new Kubernetes version**.

🧠 **Quick Summary: Cordon vs Drain**

Command	**Purpose**	**Effect**
kubectl cordon <node>	Prevent new pods from being scheduled on the node	Existing pods **stay running**
kubectl drain <node>	Evict running pods safely to prepare for maintenance or upgrade	Existing pods are **migrated** to other nodes
kubectl uncordon <node>	Make node schedulable again	Scheduler can place **new pods** on it

---

✅ **Example Interview Answer**

"When upgrading a Kubernetes node, I first cordon it to stop new pods from being scheduled, then drain it to safely evict running pods. After upgrading kubelet and kubeadm on that node, I restart kubelet and finally uncordon it to bring it back online. This ensures zero downtime and smooth rolling upgrades."

---**Automate the Kubernetes node upgrade process using Jenkins**, including **drain**, **upgrade**, and **uncordon** steps.

---**Terraform + Jenkins integrated flow** to **upgrade EKS worker nodes (node group) in AWS**.

-----Amazon EKS and Its Complete Installation Steps

Amazon Elastic Kubernetes Service (Amazon EKS) has become the backbone for organizations that want to deploy, scale, and manage containerized applications using Kubernetes in the cloud. It simplifies running Kubernetes without the overhead of managing the control plane, while providing high availability, scalability, and tight integration with AWS services.

In this comprehensive guide, we will explore what EKS is, its benefits, and the detailed step-by-step installation process that enables us to create a production-ready Kubernetes cluster on AWS.

--What is Amazon EKS?

Amazon EKS is a fully managed Kubernetes service offered by AWS. Instead of manually setting up Kubernetes control planes, networking, and security, AWS handles these complex tasks while we focus on running workloads.

--Key features of EKS include:

-Managed Control Plane: AWS automatically provisions and scales the Kubernetes control plane.
-High Availability: Multi-AZ deployments ensure fault tolerance.
-Integration with AWS Services: Works seamlessly with IAM, VPC, ALB, Route53, CloudWatch, and more.
-Automatic Scaling: Integrates with Cluster Autoscaler and HPA (Horizontal Pod Autoscaler).
-Security and Compliance: Built-in compliance with AWS security standards.

--Benefits of Using Amazon EKS

1. No Control Plane Management – AWS takes care of patching, upgrading, and availability.
2. Scalable and Flexible – Easily scale applications using EC2 or AWS Fargate.
3. Enhanced Security – Fine-grained IAM roles and VPC-based isolation for nodes.
4. Multi-Region Availability – Deploy workloads closer to end users.
5. Cost Optimization – Pay only for the worker nodes and resources consumed.

--Pre-Requisites for EKS Installation

Before starting the EKS setup, ensure the following:
-An AWS account with appropriate IAM permissions.
-Installed AWS CLI (latest version).
-Installed kubectl (compatible with Kubernetes version of EKS).
-Installed eksctl (a simple CLI tool to create and manage EKS clusters).
-Configured IAM role with AdministratorAccess for cluster creation.

--Step-by-Step Installation of Amazon EKS

Step 1: Install and Configure AWS CLI

-Install AWS CLI on your system and configure it:

```
# aws configure
```

-Enter the following:
- -AWS Access Key ID
- -AWS Secret Access Key
- -Default region (e.g., us-east-1)
- -Output format (json, table, or text)

Step 2: Install kubectl

-Download and install the Kubernetes command-line tool:

```
# curl -o kubectl https://amazon-eks.s3.us-west-2.amazonaws.com/latest/bin/linux/amd64/kubectl
# chmod +x ./kubectl
# mv ./kubectl /usr/local/bin
```

-Verify installation:

```
# kubectl version --client
```

Step 3: Install eksctl

-eksctl simplifies the creation of EKS clusters:

```
# curl --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz
# mv ./eksctl /usr/local/bin
```

-Check installation:

```
# eksctl version
```

Step 4: Create an IAM Role for EKS Cluster

-Create a role with the following AWS-managed policies:
- AmazonEKSClusterPolicy
- AmazonEKSServicePolicy

-This IAM role will be used by EKS to manage cluster resources.

Step 5: Create the EKS Cluster

-Run the eksctl command to create the cluster:

```
# eksctl create cluster \
--name my-cluster \                → Cluster name
```

```
        --region us-east-1 \                → AWS region
        --nodegroup-name my-nodes \
        --node-type t3.medium \             → EC2 instance type
        --nodes 3 \                 → Number of worker nodes
        --nodes-min 2 \
        --nodes-max 5 \
        --managed                   → Creates a managed node group
```

-This process takes 15–20 minutes.

Step 6: Verify Cluster Creation

Once complete, verify the cluster:

```
# kubectl get svc
# kubectl get nodes
```

-You should see the EKS nodes running.

Step 7: Configure kubeconfig

-Update kubeconfig so kubectl connects to the EKS cluster:

```
# aws eks --region us-east-1 update-kubeconfig --name my-cluster
```

-Verify:

```
# kubectl config get-contexts
```

Step 8: Deploy an Application on EKS

-Create a deployment YAML file (nginx-deployment.yaml):

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

-Apply it:

```
# kubectl apply -f nginx-deployment.yaml
# kubectl get pods
```

Step 9: Expose the Application

-Expose deployment with a LoadBalancer service:

```
# kubectl expose deployment nginx-deployment \
--port=80 \
--type=LoadBalancer
```

-Retrieve the external URL:

```
# kubectl get svc
```

This will provide a public endpoint for accessing the application.

Step 10: Enable Monitoring and Logging

-Integrate CloudWatch Container Insights for logging and monitoring. Use the following command to enable:

```
# eksctl utils associate-iam-oidc-provider --region us-east-1 -
-cluster my-cluster --approve
```

Install the CloudWatch agent and FluentD DaemonSet for log forwarding.

--Best Practices for EKS Installation

-Use Managed Node Groups for automatic upgrades and scaling.
-Enable IAM Roles for Service Accounts (IRSA) for fine-grained permissions.
-Adopt Infrastructure as Code (IaC) with tools like Terraform or AWS CDK.
-Set up Autoscaling (Cluster Autoscaler + HPA).
-Implement Security Best Practices such as network policies and private cluster endpoints.
-Enable Observability with Prometheus, Grafana, and CloudWatch.

--Conclusion

Amazon EKS offers a powerful, scalable, and secure way to run Kubernetes workloads on AWS. By following the step-by-step installation process, we can easily set up a production-ready Kubernetes cluster integrated with AWS services. With best practices such as IAM role management, autoscaling, and observability, EKS ensures robust application deployment in the cloud.

------Amazon Elastic Container Registry (ECR) Explained

AWS Elastic Container Registry (ECR) is Amazon's fully managed Docker container registry service that makes it easy to store, manage, and deploy Docker container images.

--Key Features:

--1. Fully Managed Service

No infrastructure to provision or manage
Automatically scales with your usage
Highly available and durable storage

--2. Security

Integrated with AWS Identity and Access Management (IAM) for access control
Image scanning for vulnerabilities (with Amazon Inspector integration)
Encryption at rest using AWS Key Management Service (KMS)
VPC endpoints available for private connectivity

--3. Integration with AWS Services

Works seamlessly with Amazon ECS, EKS, Lambda, and Batch
Compatible with Docker CLI and Kubernetes
EventBridge integration for workflow automation

--How ECR Works

Repository Creation: You create repositories to store your Docker images
Authentication: Docker client authenticates with ECR using AWS credentials
Image Management: You push/pull images using standard Docker commands
Deployment: Services like ECS/EKS pull images directly from ECR

--Common Operations
Basic Commands

# Authenticate Docker to ECR
# aws ecr get-login-password | docker login --username AWS --password-stdin <account-id>.dkr.ecr.<region>.amazonaws.com

# Create a repository #aws ecr create-repository --repository-name my-app

# Tag an image
# docker tag my-app:latest <account id>.dkr.ecr.<region>.amazonaws.com/my-app:latest

# Push an image
#docker push <account-id>.dkr.ecr.<region>.amazonaws.com/my-app:latest

# Pull an image
# docker pull <account-id>.dkr.ecr.<region>.amazonaws.com/my-app:latest

--Lifecycle Policies (Automated Image Cleanup)

# Example lifecycle policy JSON

```
{  "rules": [
    {
      "rulePriority": 1,
      "description": "Keep last 30 images",
      "selection": {
        "tagStatus": "any",
        "countType": "imageCountMoreThan",
        "countNumber": 30
      },
      "action": {
        "type": "expire"
      }  } ] }
```

# Apply lifecycle policy
  # aws ecr put-lifecycle-policy --repository-name my-app --lifecycle-policy-text file://policy.json

--Pricing Model

Storage: Pay per GB/month for data stored
Data Transfer: Free within same region, standard AWS data transfer rates apply across regions
API Requests: Minimal cost per API call (typically fractions of a cent)

--Best Practices

Use IAM Policies: Restrict access with granular permissions
Enable Image Scanning: For vulnerability detection
Implement Lifecycle Policies: Automate cleanup of old images
Use Cross-Region Replication: For disaster recovery
Monitor with CloudWatch: Track API usage and storage metrics

--Comparison with Other Registries

Feature	**AWS ECR**	**Docker Hub**	**Google Container Registry**
-Managed Service	Yes	Yes	Yes
-AWS Integration	Native	Limited	None
-Private Repos	All	Paid only	All
-Vulnerability Scanning	Yes	Paid only	Yes

----- Managing Pods Across Different Contexts:

In Kubernetes, you can view pods in different contexts (namespaces, clusters, or user configurations) using various kubectl commands and flags. Here are the different ways to set the context when running kubectl get pods:

1. Namespace-Based Views

```
Default namespace:     # kubectl get pods
Specific namespace:    # kubectl get pods -n <namespace>  # e.g., -n kube-system
All namespaces:        # kubectl get pods -A  # or --all-namespaces
Set default namespace permanently:
                       # kubectl config set-context --current --namespace=<namespace>
```

2. Cluster/Context Switching

```
List contexts:              # kubectl config get-contexts
Switch context (cluster + user):# kubectl config use-context <context-name>
Run command in a specific context (without switching):
                            # kubectl get pods --context=<context-name>
```

3. Filtering & Advanced Views

```
Filter by labels:                # kubectl get pods -l app=nginx
Real-time monitoring:            # kubectl get pods -w  # Watch mode
Extended details (node, IP, etc.):   # kubectl get pods -o wide
Custom output formats:           # kubectl get pods -o json/yaml/custom-columns=...
```

--Kubernetes-- Syllabus (with One-Line Explanations)

1. Introduction to Kubernetes

- **What is Kubernetes?** → Container orchestration platform for managing containerized apps.
- **Kubernetes Architecture** → Master (control plane) and worker nodes.
- **Key Components** → API server, etcd, scheduler, controller, kubelet, kube-proxy.
- **Kubernetes vs Docker Swarm** → Advanced orchestration vs simple clustering.

2. Setup & Installation

- **Minikube & Kind** → Local Kubernetes clusters.
- **Kubeadm** → Bootstrap production-grade clusters.
- **Managed Kubernetes** → EKS, GKE, AKS cloud services.
- **kubectl** → CLI for managing Kubernetes.

3. Core Concepts

- **Pods** → Smallest deployable unit in Kubernetes.
- **ReplicaSets** → Ensure desired number of pod replicas.
- **Deployments** → Manage rolling updates and rollbacks.
- **Namespaces** → Logical separation of resources.
- **Labels & Selectors** → Tag and group resources.
- **Annotations** → Store metadata for objects.

4. Networking

- **Cluster Networking Model** → Every pod gets its own IP.
- **Services** → Stable networking for pods.
- **NodePort** → Expose service on a node’s port.
- **ClusterIP** → Internal-only service.
- **LoadBalancer** → External access via cloud load balancer.
- **Ingress & Ingress Controller** → HTTP/HTTPS routing.
- **Network Policies** → Control pod communication.
- **CNI Plugins** → Calico, Flannel, Weave for networking.

5. Storage

- **Volumes** → Ephemeral pod storage.
- **Persistent Volumes (PV)** → Cluster-wide storage resource.
- **Persistent Volume Claims (PVC)** → Request storage for pods.
- **Storage Classes** → Dynamic provisioning of storage.
- **ConfigMaps** → Store configuration data.
- **Secrets** → Store sensitive data securely.

6. Workloads & Scheduling

- **DaemonSets** → Run one pod per node (e.g., monitoring agents).
- **StatefulSets** → Manage stateful applications (databases).
- **Jobs** → Run tasks to completion.
- **CronJobs** → Schedule recurring tasks.
- **Taints & Tolerations** → Control pod placement.
- **Node Selectors &ipconfig Affinity** → Advanced pod scheduling rules.

7. Scaling & Auto Healing

- **Horizontal Pod Autoscaler (HPA)** → Scale pods based on CPU/memory.
- **Vertical Pod Autoscaler (VPA)** → Adjust resource requests automatically.
- **Cluster Autoscaler** → Add/remove nodes dynamically.
- **Self-Healing** → Restart failed pods automatically.

---

8. Security

- **RBAC (Role-Based Access Control)** → Manage user and service permissions.
- **Service Accounts** → Identity for pods.
- **Pod Security Policies (PSP)** → Define security rules for pods.
- **Network Policies** → Restrict pod communication.
- **Secrets Management** → Store passwords, keys securely.
- **Admission Controllers** → Enforce security and compliance.

---

9. Monitoring & Logging

- **kubectl logs & exec** → Debug pod issues.
- **Metrics Server** → Collect resource usage.
- **Prometheus & Grafana** → Cluster monitoring and visualization.
- **ELK / EFK Stack** → Centralized logging.
- **Kube-State-Metrics** → Expose cluster metrics for monitoring.

---

10. Helm & Package Management

- **Helm Charts** → Package manager for Kubernetes.
- **Chart Repositories** → Store and share Helm charts.
- **Templating & Values.yaml** → Customize deployments.
- **Helm 3 vs Helm 2** → Client-only architecture vs Tiller-based.

---

11. CI/CD with Kubernetes

- **GitOps with ArgoCD/Flux** → Manage deployments via Git.
- **Jenkins X** → CI/CD for Kubernetes-native apps.
- **Kubernetes Operators** → Automate complex applications.
- **Canary & Blue-Green Deployments** → Safe rollout strategies.

---

12. Advanced Kubernetes

- **Custom Resource Definitions (CRDs)** → Extend Kubernetes API.
- **Operators** → Manage applications using custom controllers.
- **API Aggregation & Extension** → Extend Kubernetes API capabilities.
- **Multi-Cluster Management** → Manage multiple clusters at scale.
- **Service Mesh (Istio/Linkerd)** → Advanced traffic control & observability.

---

13. Best Practices

- **Resource Requests & Limits** → Prevent resource starvation.
- **Namespace Quotas** → Control resource usage per team/project.
- **Pod Disruption Budgets (PDBs)** → Ensure availability during maintenance.
- **Security Best Practices** → Run non-root containers, use minimal images.
- **Backup & Disaster Recovery** → Use Velero or similar tools.

Command	**Explanation**
kubectl version --client	Check kubectl version
kubectl cluster-info	Check Kubernetes cluster info
kubectl get nodes	List all nodes
kubectl describe nodes	List nodes with detailed info
kubectl get pods	List all pods in default namespace
kubectl get pods -A	List pods in all namespaces
kubectl get deployments	List deployments
kubectl get svc	List services
kubectl describe pod <pod-name>	Describe a pod in detail
kubectl get pod <pod> -o yaml	Get YAML output of any object
kubectl apply -f file.yaml	Apply (create/update) a YAML file
kubectl delete -f file.yaml	Delete a resource using YAML
kubectl delete pod <pod>	Delete a pod
kubectl create -f file.yaml	Create resource from YAML
kubectl edit deployment <deployment>	Edit any running object
kubectl create ns <name>	Create namespace
kubectl get ns	List all namespaces
kubectl config set-context --current --namespace=<ns>	Switch namespace for kubectl
kubectl run test --image=nginx	Run a pod quickly
kubectl expose deployment nginx --port=80 --type=NodePort	Expose deployment as service
kubectl logs <pod>	Show logs of a pod
kubectl logs -f <pod>	Stream logs (live)
kubectl exec -it <pod> -- bash	Execute command inside a pod
kubectl port-forward <pod> 8080:80	Port-forward pod to local machine
kubectl scale deployment nginx --replicas=5	Scale deployment replicas
kubectl autoscale deployment nginx --min=2 --max=10	Autoscale deployment
kubectl rollout restart deployment <name>	Restart deployment
kubectl rollout status deployment <name>	Check rollout status
kubectl rollout undo deployment <name>	Undo deployment rollout
kubectl rollout history deployment <name>	View deployment history
kubectl get events	List all events
kubectl top pod	View pod metrics (if metrics server installed)

kubectl top node	View node metrics
kubectl drain <node> --ignore-daemonsets	Drain a node for maintenance
kubectl cordon <node>	Mark a node as unschedulable
kubectl uncordon <node>	Mark node as schedulable again
kubectl delete pods --all -n <ns>	Delete all pods in a namespace
kubectl delete ns <name>	Delete a namespace
kubectl create secret generic mysecret --from-literal=key=value	Create a secret
kubectl get secrets	View secrets (encoded)
`kubectl get secret <name> -o jsonpath="{.data.key}"	Decode a secret
kubectl get configmaps	List configmaps
kubectl create configmap cm --from-file=config.txt	Create configmap from file
kubectl apply -f folder/	Apply multiple YAML files
kubectl delete -f folder/	Delete all resources in YAML folder
kubectl get pv	List persistent volumes
kubectl get pvc	List persistent volume claims
kubectl describe pvc <name>	Describe PVC
kubectl get endpoints	Get service endpoints
kubectl get ingress	List ingress rules
kubectl describe ingress <name>	Describe ingress
kubectl create sa <name>	Create a service account
kubectl top pods -A	View resource usages
kubectl config view	Export current kubeconfig
kubectl config use-context <context>	Change cluster context
kubectl config get-contexts	List contexts
kubectl delete node <node>	Delete a node from cluster
kubectl debug -it <pod> --image=busybox	Debug a pod by creating ephemeral container
kubectl create job backup --image=alpine	Create job
kubectl get cronjobs	List cronjobs
kubectl delete pod <pod> --force --grace-period=0	Force delete pod

----K8S Components OR Flow of # kubectl apply

1. kubectl (Client)

- You run:
```
# kubectl apply -f deployment.yaml
```
- kubectl reads the YAML manifest and sends a **REST request** (JSON payload) to the **Kube API Server**.
- Authentication & Authorization happen here (via RBAC, certs, or IAM if on EKS).

---

2. Kube API Server

- The **API Server** is the **front door** of Kubernetes.
- It:
  - Validates the request (schema, permissions).
  - Writes the desired state (Deployment spec) into **etcd** (the cluster's database).
- At this stage: **Desired state is stored**.

---

3. etcd (Key-Value Store)

- etcd stores the **entire cluster state**:
  - Pods, Services, ConfigMaps, Deployments, etc.
- It now contains the new Deployment object definition.

---

4. Controller Manager

- The **Deployment Controller** (inside the Controller Manager) notices that a new Deployment object is created.
- It checks:
  - Desired replicas (e.g., 3 pods).
  - Current state (0 pods running).
- It creates a **ReplicaSet object** in etcd via API Server.

---

5. Scheduler

- The **Scheduler** picks up the pending Pods (from the ReplicaSet).
- It decides **which Node** each Pod should run on, based on:
  - Resource availability (CPU, RAM).
  - Node labels, taints, tolerations.
- Scheduler updates Pod's Node assignment in etcd.

---

6. Kubelet (on Worker Node)

- The **Kubelet** on the chosen Node gets the Pod spec from API Server.
- It talks to **Container Runtime** (Docker, containerd, CRI-O) to pull the image and start the container.

---

7. Container Runtime

- Pulls the image from registry (e.g., ECR, DockerHub).
- Starts the container.

8. Kubelet Updates API Server

- Reports back: "Pod is running" (status updates).
- API Server writes the updated state into etcd.

---

9. User Sees Status

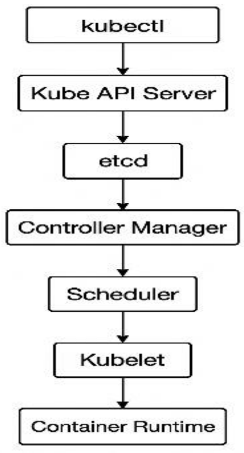
- When you run:

```
# kubectl get pods
```

- kubectl queries API Server → API Server fetches state from etcd → returns running status.

--Workflow :

1. **kubectl** → Sends request (YAML manifest) to API Server.
2. **Kube API Server** → Validates request and updates etcd.
3. **etcd** → Stores desired cluster state.
4. **Controller Manager** → Reconciles state, creates ReplicaSets/Pods.
5. **Scheduler** → Assigns Pods to appropriate Nodes.
6. **Kubelet** → Receives Pod specs and manages containers on the Node.
7. **Container Runtime** → Pulls images and runs containers.



◆ Summary (Interview-Ready Answer)

"When I run kubectl apply, the manifest goes to the API Server, which authenticates and stores it in etcd. The Controller Manager notices the new Deployment and creates ReplicaSets. The Scheduler assigns Pods to Nodes, and the Kubelet on those Nodes pulls images via the Container Runtime and starts containers. Finally, the Kubelet reports status back to API Server, which updates etcd. So, the API Server is the entry point, etcd is the source of truth, controllers enforce desired state, the scheduler assigns workloads, and kubelets actually run them."

-----How have you achieved 40% cut down in deployedment time as mentioned in resume?

🎯 **Situation**
Earlier, deployments were **manual / semi-automated**, involving:

- Manual approvals,
- Multiple Jenkins jobs triggered separately,
- Large Docker image builds from scratch,
- Manual Kubernetes rollouts with downtime.

This resulted in **long deployment cycles** (say 30–40 minutes).

🚀 **Actions I Took**

1. **CI/CD Pipeline Optimization**
   - Implemented **Jenkins pipelines** with automated build → test → deploy stages.
   - Added **parallel stages** for running unit tests, linting, and security scans to save time.
   - Introduced **Slack/Email notifications** to reduce manual monitoring.
2. **Docker Build Improvements**
   - Optimized Dockerfiles using **multi-stage builds** and **layer caching**.
   - Reduced image size by ~50%, cutting build & push time significantly.
3. **Artifact & Dependency Caching**
   - Configured Jenkins to **cache dependencies** (Python pip, Node npm, Maven, etc.) so builds didn't fetch packages from scratch each time.
4. **Infrastructure Automation**
   - Used **Helm charts** for Kubernetes deployments instead of raw YAML → standardized and reusable templates.
   - Enabled **rolling updates** with proper readiness probes → zero downtime and faster rollout.
5. **Environment Provisioning**
   - Automated environment creation with **Terraform** (instead of manual setup).
   - Reduced setup overhead during deployments.

📊 **Result**

- Average deployment time dropped from **~35 minutes to ~20 minutes** (≈40% faster).
- Reduced manual intervention → fewer errors and rollbacks.
- Improved developer productivity since code changes reached production faster.

✅ **Sample Answer in Interview Style**:
"In my previous role, deployments were taking around 35–40 minutes because of manual steps, heavy image builds, and sequential testing. I optimized our Jenkins pipelines by introducing parallel test execution, Docker layer caching, and multi-stage builds. I also implemented Helm charts for Kubernetes deployments and automated infrastructure provisioning with Terraform. These improvements cut down deployment time to about 20 minutes — roughly a 40% reduction — while also making releases more reliable and less error-prone."

----Host your application on EC2 but route traffic like this:

- chaitanya.com/home → backend service A (maybe running on EC2 or a path inside app)
- chaitanya.com/profile → backend service B
- (and maybe more paths in future)

This is a classic **URL path-based routing** setup. Here's how I would design it using AWS services:

◆ AWS Services to Use

1. **Amazon Route 53**
    - To manage the domain (chaitanya.com).
    - Create a DNS record (A or CNAME) pointing to a Load Balancer.
2. **Elastic Load Balancer (ALB – Application Load Balancer)**
    - ALB supports **path-based routing**.
    - Example rules:
        - /home → Target Group A (EC2 instance or container).
        - /profile → Target Group B.
3. **EC2 (Application Servers)**
    - Your application (or multiple apps/services) run here.
    - Each Target Group in ALB points to one or more EC2 instances.

(Optional) **Auto Scaling Group** if you want HA & scaling.

◆ Step-by-Step Setup

1. Domain Setup in Route 53

- Register domain chaitanya.com in Route 53 (or transfer if external).
- Create a **Hosted Zone**.
- Add an **A Record** pointing chaitanya.com → ALB DNS name.

---

2. Create Application Load Balancer (ALB)

- Go to EC2 → Load Balancers → Create **Application Load Balancer**.
- Select **internet-facing**.
- Configure **listeners** (usually port 80, optionally 443 with SSL).

---

3. Define Target Groups

- Target Group A → points to EC2 instances running /home service.
- Target Group B → points to EC2 instances running /profile service.

---

4. Configure Path-Based Routing Rules

In ALB Listener (port 80 or 443):

- Rule 1: If path is /home* → forward to **Target Group A**.
- Rule 2: If path is /profile* → forward to **Target Group B**.
- Default Rule → forward to a default app or return 404.

---

5. Test

- Visit:
    - http://chaitanya.com/home → should go to Service A.
    - http://chaitanya.com/profile → should go to Service B.

◆ **Architecture Diagram (in words)**

User -> Route53 (DNS) -> ALB (path-based routing) -> Target Group(s) -> EC2 instances

---

✅ **Why ALB?**

- Supports **path-based** and **host-based** routing.
- Works with multiple services behind one domain.
- Scales automatically.

----Let's extend the same idea but in Kubernetes (EKS)

In AWS EKS, instead of directly managing EC2 + ALB target groups, we let **Ingress + ALB Ingress Controller** (a.k.a. AWS Load Balancer Controller) handle routing.

◆ **Services to Use in AWS EKS Setup**

1. **Amazon Route 53**
   - Same as before → chaitanya.com domain managed in Route 53.
   - Create an A record → point to ALB created by the Ingress Controller.

2. **AWS Load Balancer Controller (Ingress Controller)**
   - Installed in your EKS cluster.
   - Automatically provisions an **Application Load Balancer (ALB)** for Ingress objects.

3. **Kubernetes Ingress**
   - Define **path-based routing rules** for /home, /profile.

4. **Kubernetes Services & Pods**
   - /home → Service A → Pods A.
   - /profile → Service B → Pods B.

◆ **Step-by-Step Setup**

1. Install AWS Load Balancer Controller

- Deploy the controller via Helm:
- helm repo add eks https://aws.github.io/eks-charts
- helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
-   -n kube-system \
-   --set clusterName=<your-cluster-name>
- This makes EKS able to create/manage ALBs from Ingress.

2. Create Kubernetes Services

For /home app:
```
apiVersion: v1
kind: Service
metadata:
  name: home-service
spec:
  selector:
    app: home-app
  ports:
    - port: 80
      targetPort: 8080
```
For /profile app:
```
apiVersion: v1
kind: Service
metadata:
  name: profile-service
spec:
  selector:
    app: profile-app
  ports:
    - port: 80
      targetPort: 8080
```

---

3. Create Ingress with Path-Based Routing

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: chaitanya-ingress
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
spec:
  rules:
  - host: chaitanya.com
    http:
      paths:
      - path: /home
        pathType: Prefix
        backend:
          service:
            name: home-service
            port:
              number: 80
      - path: /profile
        pathType: Prefix
        backend:
          service:
            name: profile-service
            port:
              number: 80
```

---

4. Update Route 53

- The Ingress Controller will provision an ALB.
- Get ALB DNS:
- kubectl get ingress chaitanya-ingress.
- Add Route 53 A record → point chaitanya.com to ALB DNS.

5. Test

- http://chaitanya.com/home → Pods in home-service.
- http://chaitanya.com/profile → Pods in profile-service.

◆ **Architecture (in words)**

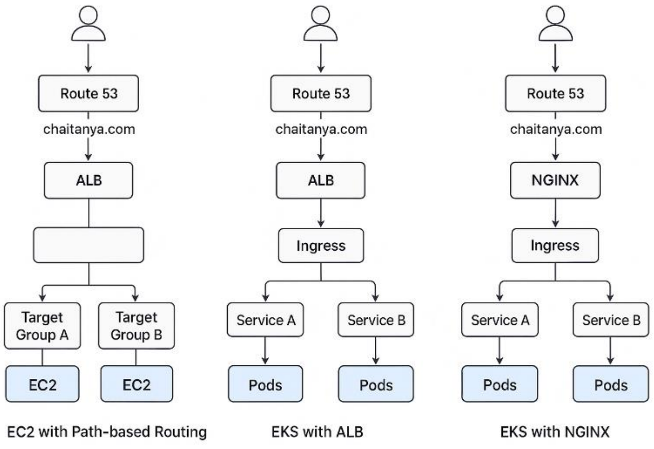
```
User -> Route 53 -> ALB (created by Ingress Controller)
    -> Ingress rules (/home, /profile)
        -> Service A -> Pods A
        -> Service B -> Pods B
```

✅ **In short:**

- **EC2 setup** → You manually create ALB + target groups.
- **EKS setup** → You write **Ingress YAML**, and AWS Load Balancer Controller auto-creates ALB with path-based routing.

----ALB Ingress Controller vs Nginx Ingress Controller

☐ **External traffic** → Use **ALB Ingress Controller** (Route53 → ALB → app).
☐ **Internal traffic / microservice-to-microservice routing** → Use **Nginx Ingress Controlle**



-----Karpenter

Karpenter is an **open-source Kubernetes cluster autoscaler** developed by AWS.
Unlike the older **Cluster Autoscaler**, Karpenter makes scaling decisions **faster and more flexible**, directly interacting with the AWS APIs.

🎯 How I Utilized Karpenter

1. **Faster Node Provisioning**
   - Configured Karpenter in my EKS cluster to launch EC2 instances within **seconds** when pods were unschedulable.
   - This helped handle **traffic spikes** in our e-commerce app during sales hours.
2. **Cost Optimization**
   - Set up **consolidation policies** to downscale underutilized nodes.
   - Used **Spot + On-Demand mixed instances** — Karpenter automatically picked the cheapest instance type that satisfied pod requirements.
   - Achieved **20–30% cost savings** compared to static node groups.
3. **Flexible Instance Selection**
   - Instead of predefining instance types in ASGs, I configured Karpenter with **InstanceProfile + EC2NodeClass** so it could pick from a wide range of EC2 families/sizes.
   - Example: if my workload needed 8 vCPUs + 32 GB RAM, Karpenter could choose between m5.2xlarge, r5.2xlarge, or c5.2xlarge.
4. **Workload-aware Scheduling**
   - Applied nodeSelector, affinity, and tolerations so Karpenter launched different node types for:
     - **General workloads** → On-Demand instances.
     - **Batch/CI workloads** → Spot instances.
     - **GPU workloads** → G4dn nodes.
5. **Cluster Right-Sizing**
   - Enabled **consolidation** to automatically terminate underutilized nodes and reschedule pods, keeping the cluster lean.

-----How to connect a Kubernetes application (Pod/Deployment) to an AWS RDS database, using secrets, config, and YAML manifests.

☐ **Scenario** We want to:

- Use an **Amazon RDS MySQL** database.
- Deploy an **application in Kubernetes (EKS)** that connects to it.
- Securely store DB credentials in Kubernetes Secrets.
- Pass the connection details to the app via environment variables.

⚙ Step 1: Create Amazon RDS Database

In AWS Console:

1. Go to **RDS** → **Databases** → **Create Database**
2. Choose engine: **MySQL / PostgreSQL / MariaDB / Aurora**
3. Choose deployment type: **Standard Create**
4. Set:
   - DB instance identifier: mydb
   - Master username: admin
   - Master password: MySecret123
   - Public access: **Yes** (for testing) or use **VPC peering** if private.
5. Note down:
   DB Endpoint: mydb.xxxxxx.ap-south-1.rds.amazonaws.com
   DB Port: 3306

⚙ Step 2: Create Kubernetes Secret for DB Credentials

Create a file named **db-secret.yaml**:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: YWRtaW4=              # base64 encoded value of 'admin'
  password: TXlTZWNyZXQxMjM=     # base64 encoded value of 'MySecret123'
```

◆ **Encode values using base64:**

```
# echo -n 'admin' | base64
# echo -n 'MySecret123' | base64
```

Apply it: `# kubectl apply -f db-secret.yaml`

⚙ Step 3: Create ConfigMap for Database Connection Info

File: **db-configmap.yaml**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  db-host: mydb.xxxxxx.ap-south-1.rds.amazonaws.com
  db-port: "3306"
  db-name: mydatabase
```

Apply it:

```
# kubectl apply -f db-configmap.yaml
```

⚙ Step 4: Create Deployment for Application

File: **app-deployment.yaml**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp-container
        image: myrepo/myapp:latest      # Replace with your app image
        ports:
        - containerPort: 8080
        env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: db-config
              key: db-host
        - name: DB_PORT
          valueFrom:
            configMapKeyRef:
              name: db-config
              key: db-port
        - name: DB_NAME
          valueFrom:
            configMapKeyRef:
              name: db-config
              key: db-name
        - name: DB_USER
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: username
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: password
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
  - port: 80
    targetPort: 8080
  type: LoadBalancer
```

Apply it: `# kubectl apply -f app-deployment.yaml`

⚙ Step 5: Verify Connection

1. Check Pods: `# kubectl get pods`
2. View logs: `# kubectl logs <pod-name>`
3. The application should connect to the RDS endpoint using env vars:

```
DB_HOST=mydb.xxxxxx.ap-south-1.rds.amazonaws.com
DB_USER=admin
DB_PASSWORD=MySecret123
DB_PORT=3306
```

⚙ Optional: Test Connection from Pod

You can test DB connectivity from inside a pod:

```
# kubectl exec -it <pod-name> -- bash
# apt-get update && apt-get install -y mysql-client
# mysql -h mydb.xxxxxx.ap-south-1.rds.amazonaws.com -u admin -p
```

✓ Summary

Step	Task	File
1	Create RDS in AWS	—
2	Create Secret (DB credentials)	db-secret.yaml
3	Create ConfigMap (host/port)	db-configmap.yaml
4	Create Deployment + Service	app-deployment.yaml
5	Test connection	kubectl exec

--Project tree

```
k8s-rds-demo/
    ├─ app/
    │    ├─ app.py
    │    ├─ requirements.txt
    │    └─ Dockerfile
    ├─ k8s/
    │    ├─ db-secret.yaml
    │    ├─ db-configmap.yaml
    │    └─ app-deployment.yaml
    ├─ Jenkinsfile
    └─ README.md
```

--Restricting communication between pods in different namespaces

--Kubernetes Network Policies. ---**"Service-name.namespace.svc.cluster.local"**

Goal	Method
Block all traffic to/from a namespace	Apply a default-deny-ingress Network Policy.
Allow traffic only within the same namespace	Combine default-deny with an allow-same-namespace policy.
Allow traffic from a specific pod in another namespace	Use a policy with namespaceSelector and a nested podSelector.
Completely isolate two namespaces	Apply default deny and intra-namespace allow policies to both.

---Benefits of using secerets over the configmap

Feature	Secret	ConfigMap	Benefit
Encryption at Rest	✓ Yes	✕ No	Preves someone with access to your etcd database (the Kubernetes backend store) from reading the sensitive data. The data is stored encrypted.
Reduced Risk of Accidental Exposure	✓ Optional	✕ No	Kubernetes can protect Secrets from being accidentally exposed in log output or kubectl describe output.
In-Memory Storage	✓ tmpfs	✕ Regular FS	When mounted as a volume, Secrets are stored on a tmpfs (RAM-backed filesystem) on the node, not written to disk.

---CrashLoopBackOff & ImagePullBackOff

- **CrashLoopBackOff** → Pod starts but fails repeatedly → check **logs/config/resources**.
- **ImagePullBackOff** → Pod can't even start → check **image name, registry access, credentials**.

---Check Namespace Resource Usage

☐ Use kubectl get resourcequota -n <ns> and kubectl describe resourcequota -n <ns> → to see **limits (hard) and usage**.

☐ Use kubectl top pod -n <ns> → to see **real-time usage** even if no quotas are applied.

Q- How will you ensure zero downtime for your application?

Answer- "To ensure zero downtime, I follow a few strategies depending on the application. For stateless services running on Kubernetes, I use **rolling updates** with proper readiness and liveness probes so traffic is only routed to healthy pods. For critical releases, I prefer **blue-green deployments** or **canary rollouts**, where I can gradually shift traffic and monitor metrics. At the infrastructure level, I always deploy behind a load balancer with health checks. On the data side, I make sure schema changes are **backward-compatible** so both old and new versions can coexist during the rollout. Finally, I rely on automation and monitoring — if the new version fails health checks or metrics spike, the pipeline automatically rolls back. This way, users don't experience downtime during deployments."

Q- Which is best Rolling Deployment Blue-Green Deployment Canary Deployment

Answer- "There's no universal best — it depends on context. For **frequent, low-risk releases**, I prefer **rolling updates** since they're cost-efficient. For **critical applications** where rollback speed matters, I go with **blue-green deployments**. And for **progressive delivery or large-scale systems**, **canary deployments** are best since they let me test with a small % of traffic before a full rollout."

---Why EKS Cluster Uses Private Subnets for Nodes

Security best practices:

- Worker nodes should not be directly accessible from the internet.
- Only the **EKS control plane** and **load balancers** handle external communication.
- Pods inside the private subnets still need **outbound internet** (for pulling images, updates, etc.), so they use the **NAT Gateway**.

✓ Summary Table

Resource	**Created Automatically?**	**Purpose**	**Internet Access**
Public Subnet	✓	Load Balancers, Bastion Hosts	✓ Direct via Internet Gateway
Private Subnet	✓	EKS Worker Nodes, Pods	☑ Indirect via NAT Gateway
NAT Gateway	✓	Allow private nodes to access internet	One-way (outbound only)
Internet Gateway	✓	Provides internet access to public subnets	Two-way

---

🔍 Example Real-Time Flow

1. Pod (in private subnet) → Pulls image from Docker Hub
2. Traffic goes:
   Private Subnet → NAT Gateway → Internet Gateway → Docker Hub
3. Response comes back the same way.
4. The node never exposes its private IP publicly.

Q-) When we create a 3 node cluster on amazon eks then how many vm (EC2) created on aws by this eks cluster and why ?

✅ **Answer Summary** [1Node= Max 110 Pods, 1Pod=Unlim Cont.10 Recomd]

Component	**Managed By**	**EC2 Instances Created**	**Visible in EC2 Console**	**Description**
Control Plane	AWS	Multiple (3) (hidden)	✕ No	AWS-managed master nodes (high availability)
Worker Nodes (3-node cluster)	You	**3 EC2 instances**	✅ Yes	Run your workloads (pods, containers)

Perfect 👍 — Let's go step-by-step so you understand **what exactly AWS creates** when you set up a **3-node EKS cluster** — whether via **eksctl**, **AWS Console**, or **Terraform**.

☐ **Step 1: You Create the EKS Cluster**

When you run a command like:

```
# eksctl create cluster --name demo-cluster --region us-east-1 --nodes 3
```

AWS automatically provisions several components behind the scenes.

◆ **1. Amazon VPC and Networking Resources**

If you don't specify an existing VPC, EKS automatically creates a new one with all required network components:

Resource	**Count**	**Purpose**
VPC	1	Isolated network environment for your EKS cluster
Subnets	6 (3 public + 3 private)	2 per AZ across 3 Availability Zones for HA
Internet Gateway (IGW)	1	Enables internet access for public subnets
NAT Gateway	3	Allows private subnets to access the internet for updates/images, 1 Per AZ
Route Tables	6	Controls routing between public/private subnets
Security Groups	2+	One for control plane, one for worker nodes
Elastic IPs (EIPs)	3	One per NAT Gateway

✅ **Purpose:** Ensures your EKS nodes and pods can communicate securely inside the VPC and access the internet when needed.

◆ **2. EKS Control Plane (AWS Managed)**

AWS automatically provisions and manages:

- **3 master nodes (hidden)** — distributed across **3 Availability Zones** for high availability.
- These run:
  - Kubernetes **API server**
  - **etcd** (cluster store)
  - **Controller manager**
  - **Scheduler**

✅ **You don't see or manage these VMs.**
✅ **You just pay a fixed control-plane fee (~$0.10 per hour).**

◆ **3. Node Group (Your Worker Nodes)**
EKS (or eksctl) creates:

Resource	**Count**	**Purpose**
Auto Scaling Group	1	Manages scaling of your worker nodes
EC2 Instances	**3**	Worker nodes that run your pods
IAM Role	1	Permissions for EC2 to connect to EKS (e.g., pull images, write logs)
Node Security Group	1	Allows communication with control plane & between nodes

✅ Each EC2 node typically uses an **Amazon Linux 2 AMI** with **kubelet**, **containerd**, and **aws-iam-authenticator** pre-installed.

◆ **4. IAM Roles and Policies**
EKS automatically creates or attaches:

IAM Role	**Purpose**
EKS Cluster Role	Lets EKS manage networking and API calls
EKS Node Role	Lets nodes pull from ECR, log to CloudWatch, etc.
EKS Service Role (optional)	Used by the AWS Load Balancer Controller, etc.

◆ **5. CloudFormation Stack (If using eksctl)**

When you use eksctl, it creates:

- 1 **CloudFormation stack for the EKS cluster**
- 1 **stack for the node group**

You can see these under **CloudFormation** → **Stacks** in the AWS console.

◆ **6. Optional Supporting Components**

If you choose to enable them:

Resource	**Purpose**
CloudWatch Logs Group	Stores cluster and application logs
Elastic Load Balancer (ALB/NLB)	Created automatically when you deploy a Service of type LoadBalancer
ECR Repository	For storing Docker images
Amazon S3 bucket	For Terraform/eksctl state files or backups

✅ **Final Summary Table**

Component	**Count**	**Managed By**	**Visible**	**Purpose**
Control Plane (master nodes)	3	AWS	✕ No	Run Kubernetes core services
Worker Nodes	3	You	✅ Yes	Run your pods
VPC	1	You	✅ Yes	Cluster networking
Subnets	6	You	✅ Yes	Split across 3 AZs
Security Groups	2+	You	✅ Yes	Traffic control
NAT Gateways	3	You	✅ Yes	Internet access for private nodes
IAM Roles	2–3	You	✅ Yes	Permissions for EKS + nodes

-----Kubernetes Deployment stage with credentials as pipeline stage in jenkins

In modern DevOps pipelines, **Jenkins and Kubernetes** form a powerful combination for continuous integration and deployment. Integrating Kubernetes as a **deployment stage** within a Jenkins pipeline enables automated delivery of containerized applications while maintaining **security through credential management**. In this guide, we'll walk through how to configure, automate, and secure a **Kubernetes deployment stage with credentials** in Jenkins pipelines — a key skill for any DevOps engineer.

--Understanding Jenkins Pipeline for Kubernetes Deployment

A **Jenkins pipeline** is an automated process that defines how code moves from development to production. When integrated with Kubernetes, Jenkins can build Docker images, push them to container registries, and deploy them to a Kubernetes cluster using declarative or scripted pipelines.

A typical CI/CD workflow includes:

1. **Build Stage** – Building the Docker image.
2. **Push Stage** – Pushing the image to a container registry (like ECR or Docker Hub).
3. **Deploy Stage** – Deploying the image to Kubernetes using deployment manifests.
4. **Post Actions** – Notifications or clean-up actions.

The **Kubernetes Deployment Stage** ensures that the application is deployed securely using Jenkins credentials without exposing sensitive information.

--Why Use Credentials in Jenkins Pipelines

Managing credentials securely is a cornerstone of DevOps automation. Jenkins provides a **Credentials Binding Plugin**, allowing sensitive data such as **Kubeconfig files**, **Docker registry tokens**, or **AWS IAM roles** to be accessed securely inside pipeline stages.

Types of Credentials Commonly Used:

- **Kubernetes Service Account Token**
- **Kubeconfig Secret Files**
- **Docker Hub / ECR Credentials**
- **SSH Keys for Git Access**
- **API Tokens for Cluster Access**

By using credentials, we avoid hardcoding secrets in the pipeline and enable **role-based access control (RBAC)** for deployments.

--Step-by-Step: Configuring Jenkins for Kubernetes Deployment

Step 1: Add Kubernetes Credentials in Jenkins

1. Navigate to: **Manage Jenkins** → **Credentials** → **Global** → **Add Credentials**
2. Choose the type of credential:
   - **Kubeconfig file** for cluster access.
   - **Secret text** for service account tokens.

3. Provide:
   - **ID:** kubeconfig
   - **Description:** Kubernetes cluster credentials
4. Save it.

This credential will later be referenced in your pipeline.

Step 2: Configure Jenkins Agent with kubectl

Ensure your Jenkins agent (or node) has:

- **kubectl CLI installed**
- **Access to Kubernetes cluster**
- **Docker and AWS CLI (if pushing to ECR)**

You can verify by running:

```
# kubectl version --client
```

Step 3: Create a Jenkins Declarative Pipeline

Below is an example of a **Jenkinsfile** that includes a Kubernetes deployment stage using credentials securely:

```
pipeline {
    agent any

    environment {
        REGISTRY = '1234567890.dkr.ecr.us-east-1.amazonaws.com'
        IMAGE_NAME = 'nodejs-app'
    }

    stages {

        stage('Checkout Code') {
            steps {
               git branch: 'main', url: 'https://github.com/example/nodejs-app.git'
            }
        }

        stage('Build Docker Image') {
            steps {
                script {
                    sh 'docker build -t $REGISTRY/$IMAGE_NAME:latest .'
                }
            }
        }

        stage('Push to ECR') {
            steps {
                withCredentials([usernamePassword(credentialsId: 'aws-ecr-creds',
usernameVariable: 'AWS_ACCESS_KEY_ID', passwordVariable:
'AWS_SECRET_ACCESS_KEY')]) {
                    sh '''
                        aws ecr get-login-password --region us-east-1 | docker login
--username AWS --password-stdin $REGISTRY
                        docker push $REGISTRY/$IMAGE_NAME:latest
                    '''
                }
         }   }
        stage('Deploy to Kubernetes') {
            steps {
                withCredentials([file(credentialsId: 'kubeconfig', variable:
'KUBECONFIG')]) {
                    sh '''
                        kubectl apply -f k8s/deployment.yaml
                        kubectl apply -f k8s/service.yaml
                        kubectl rollout status deployment/nodejs-app
                    '''
                }
            }
        }
    }

    post {
        success {
            mail to: 'team@company.com',
                 subject: 'Deployment Successful',
                 body: 'Kubernetes Deployment completed successfully.'
        }
        failure {
            mail to: 'team@company.com',
                 subject: 'Deployment Failed',
                 body: 'Please check Jenkins console output for errors.'
        }
    }
}
```

Explanation of Key Parts:

- **withCredentials()** securely binds stored secrets to environment variables.
- **$KUBECONFIG** points to the file containing Kubernetes cluster access credentials.
- **kubectl apply** ensures the latest deployment manifest is applied.
- **post{} block** handles notifications for build success or failure.

Step 4: Define Kubernetes Deployment and Service YAMLs

A typical deployment manifest used by Jenkins might look like this:

```
# deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodejs-app
  labels:
    app: nodejs-app

spec:
  replicas: 2
  selector:
    matchLabels:
      app: nodejs-app
  template:
    metadata:
      labels:
        app: nodejs-app
    spec:
      containers:
      - name: nodejs-app
        image: 1234567890.dkr.ecr.us-east-1.amazonaws.com/nodejs-
app:latest
        ports:
        - containerPort: 3000
```

# service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nodejs-service
spec:
  selector:
    app: nodejs-app
  ports:
  - port: 80
    targetPort: 3000
  type: LoadBalancer
```

These manifests can be stored inside the repository under a k8s/ directory and applied automatically during the Jenkins pipeline run.

Step 5: Secure Credential Handling Best Practices

1. **Never expose credentials** in Jenkins logs.
2. Use **"file" or "secret text"** credential types based on the requirement.
3. Apply **Role-Based Access Control (RBAC)** in Kubernetes to restrict access.
4. Integrate **Vault or AWS Secrets Manager** with Jenkins for secret rotation.
5. Regularly **rotate credentials** and review permission scopes.

Step 6: Adding Slack or Email Notifications

To improve visibility, integrate Slack or Email notifications in the post section of the pipeline.

Example:

```
post {
   success {
       slackSend(channel: '#devops-alerts', message: "✅ Deployment
successful to Kubernetes!")
   }
   failure {
       slackSend(channel: '#devops-alerts', message: "❌ Deployment failed.
Check Jenkins logs.")
   }  }
```

-This ensures real-time updates for every deployment action.

--Advanced Tips for Kubernetes Deployment with Jenkins

- **Use Helm Charts:** Replace raw YAML with Helm for more flexible templating.
- **Integrate SonarQube and Trivy:** Add security and code-quality checks before deployment.
- **Enable Blue-Green or Canary Deployments:** Manage zero-downtime updates.
- **Use Jenkins Shared Libraries:** Simplify repetitive deployment logic.
- **Add Approval Gates:** Use Jenkins input steps for manual approvals in production stages.

--Troubleshooting Common Issues

Issue	**Cause**	**Solution**
kubectl not found	Jenkins agent missing kubectl	Install kubectl in Jenkins node
Forbidden error	Invalid Kubernetes credentials	Recheck Kubeconfig permissions
Docker push failed	Registry authentication issue	Verify registry credentials
Timeout during rollout	Incorrect deployment YAML	Validate YAML syntax and service exposure

--Conclusion

Integrating a **Kubernetes deployment stage with credentials** in Jenkins pipelines ensures secure, automated, and repeatable application deployments. By leveraging Jenkins credentials, RBAC policies, and structured pipelines, teams can achieve efficient **CI/CD workflows** across multiple environments. This method minimizes human error, enhances security, and improves overall deployment speed.

With the growing adoption of **container orchestration**, understanding how to integrate Jenkins and Kubernetes effectively is an essential DevOps skill that drives productivity and scalability.

--AWS-- Syllabus (with One-Line Explanations)

1. Introduction to Cloud & AWS

- **What is Cloud Computing?** → On-demand IT resources over the internet.
- **AWS Overview** → Amazon's cloud platform with 200+ services.
- **AWS Global Infrastructure** → Regions, Availability Zones, Edge Locations.
- **Free Tier & Pricing** → Cost management and free trial services.

---

2. Compute Services

- **EC2 (Elastic Compute Cloud)** → Virtual servers in the cloud.
- **EBS (Elastic Block Store)** → Persistent block storage for EC2.
- **EFS (Elastic File System)** → Scalable file storage.
- **Lambda** → Serverless compute service.
- **Auto Scaling** → Adjusts EC2 capacity automatically.
- **Elastic Beanstalk** → PaaS for easy app deployment.
- **Lightsail** → Simple VPS for small applications.

---

3. Storage Services

- **S3 (Simple Storage Service)** → Object storage for any type of data.
- **Glacier** → Low-cost archival storage.
- **Storage Gateway** → Hybrid cloud storage integration.
- **Snowball** → Physical device for data migration.

---

4. Networking & Content Delivery

- **VPC (Virtual Private Cloud)** → Isolated virtual network.
- **Subnets, Route Tables, Gateways** → Network segmentation and routing.
- **ELB (Elastic Load Balancer)** → Distributes traffic across instances.
- **CloudFront** → Content delivery network (CDN).
- **Direct Connect** → Dedicated private connection to AWS.
- **Route 53** → Scalable DNS and domain registration.

---

5. Databases

- **RDS (Relational Database Service)** → Managed SQL databases.
- **Aurora** → High-performance relational database.
- **DynamoDB** → Managed NoSQL database.
- **ElastiCache** → In-memory caching (Redis/Memcached).
- **Neptune** → Graph database service.
- **Redshift** → Data warehousing and analytics.

---

6. Security, Identity & Compliance

- **IAM (Identity & Access Management)** → Manage users, roles, policies.
- **KMS (Key Management Service)** → Encryption key management.
- **CloudHSM** → Hardware security module for key storage.

- **AWS Organizations** → Manage multiple AWS accounts.
- **Shield & WAF** → DDoS protection & web firewall.
- **Secrets Manager & Parameter Store** → Securely store secrets.

---

7. Monitoring & Management

- **CloudWatch** → Monitoring and logging.
- **CloudTrail** → Record API activity and governance.
- **Config** → Track resource configuration changes.
- **Trusted Advisor** → Cost and performance recommendations.
- **Systems Manager** → Manage and automate tasks across EC2 & AWS.

---

8. Application Integration

- **SQS (Simple Queue Service)** → Message queuing service.
- **SNS (Simple Notification Service)** → Publish/subscribe notifications.
- **EventBridge (CloudWatch Events)** → Event-driven workflows.
- **Step Functions** → Serverless orchestration of workflows.

---

9. Analytics & Big Data

- **Athena** → Query S3 data using SQL.
- **EMR (Elastic MapReduce)** → Big data processing using Hadoop/Spark.
- **Kinesis** → Real-time data streaming.
- **QuickSight** → Business intelligence dashboards.
- **Glue** → ETL (Extract, Transform, Load) service.

---

10. Machine Learning & AI

- **SageMaker** → Build, train, and deploy ML models.
- **Rekognition** → Image and video analysis.
- **Comprehend** → NLP service.
- **Lex** → Conversational chatbot service.
- **Translate, Polly, Transcribe** → Language translation, speech synthesis, and speech-to-text.

---

11. Developer & DevOps Tools

- **CodeCommit** → Git-based source control.
- **CodeBuild** → Build automation.
- **CodeDeploy** → Deployment automation.
- **CodePipeline** → CI/CD pipeline orchestration.
- **Cloud9** → Cloud-based IDE.
- **X-Ray** → Application tracing.

---

12. Migration & Transfer

- **Database Migration Service (DMS)** → Migrate databases to AWS.
- **Server Migration Service (SMS)** → Migrate servers to AWS.
- **Snowball & Snowmobile** → Large-scale data transfer.
- **DataSync** → Automated on-prem to AWS data transfer.

13. Hybrid & Edge Services

- **Outposts** → Run AWS on-premises.
- **Local Zones** → Low-latency infrastructure in cities.
- **Wavelength** → AWS at telecom 5G edge.

---

14. Enterprise & Governance

- **Billing & Cost Explorer** → Track and analyze costs.
- **Budgets** → Set spending limits.
- **Service Catalog** → Manage approved AWS resources.
- **Control Tower** → Set up and govern multi-account AWS environments.

---

15. Serverless & Modern Architectures

- **API Gateway** → Create/manage APIs.
- **AppSync** → GraphQL API service.
- **Fargate** → Serverless container service.
- **ECS/EKS** → Container orchestration for Docker/Kubernetes.

--AWS--

----Amazon EC2 Auto Recovery?

--Auto Recovery automatically recovers your EC2 instance if it becomes impaired due to underlying AWS hardware or software issues (like host failure, hardware crash, or network connectivity loss).
--Unlike Auto Scaling (which replaces the instance), Auto Recovery brings back the same instance, so:

- -Same instance ID
- -Same private/public IP
- -Same EBS volumes
- -Same Elastic IPs, IAM role, security groups, placement group, etc.

☞ This is important for stateful workloads where you don't want a new instance but the same one recovered.

--When Does Auto Recovery Trigger?
-System Status Check fails (hardware/host issue) for 2+ minutes.

-Examples:

- -Loss of system power
- -Loss of network connectivity
- -Host hardware failure
- -AWS software issues on the host

⚠ Note: It does not recover if the issue is inside the instance itself (like kernel panic, misconfigured app, or full disk). That's Instance Status Check failure.

--How to Configure EC2 Auto Recovery

1. AWS Console
-Go to EC2 → Instances → Select instance → Actions → Monitor and troubleshoot → Manage CloudWatch alarms.
Create an alarm on:
StatusCheckFailed_System >= 1 for 2 consecutive periods of 1 minute

-Set the alarm action: Recover this instance.

2. AWS CLI

```
aws cloudwatch put-metric-alarm \
 --alarm-name "AutoRecoveryAlarm" \
 --metric-name StatusCheckFailed_System \
 --namespace AWS/EC2 \
 --statistic Minimum \
 --period 60 \
 --evaluation-periods 2 \
 --threshold 1 \
 --comparison-operator GreaterThanOrEqualToThreshold \
 --dimensions Name=InstanceId,Value=i-0abcd1234efgh5678 \
 --alarm-actions arn:aws:automate:us-east-1:ec2:recover
```

3. Terraform

```
 resource "aws_cloudwatch_metric_alarm" "ec2_auto_recovery" {
  alarm_name          = "ec2-auto-recovery"
  comparison_operator = "GreaterThanOrEqualToThreshold"
  evaluation_periods  = "2"
  metric_name         = "StatusCheckFailed_System"
  namespace           = "AWS/EC2"
  period              = "60"
  statistic           = "Minimum"
  threshold           = "1"

  dimensions = {
    InstanceId = "i-0abcd1234efgh5678"
  }

  alarm_actions = [
    "arn:aws:automate:us-east-1:ec2:recover"
  ]
}
```

--Best Practices for Auto Recovery

-Use it for critical single-instance workloads where you can’t afford to lose IP or config.
-Combine with EBS snapshots + AMI backups for extra resilience.
-If you need scale-out + self-healing, use Auto Scaling Groups instead of Auto Recovery.
-Enable CloudWatch alarms & notifications (SNS/Slack) so ops team knows when a recovery happened.
-Place in Multi-AZ architecture for better HA.

---Regional Services & Global Services in AWS

“Regional services are tied to a specific AWS region (like EC2, RDS, VPC, EKS/ECS, S3 Bucket), meaning resources live only in that region. Global services (like IAM, Route 53, CloudFront) work across all regions and don’t require choosing a region.

Typically, compute/storage are regional, while identity, DNS, and content delivery are global.”

-----list of important Linux directories in an EC2 instance

◆ 1. General Linux Directories for EC2 Troubleshooting

--These are common across all tools:

/var/log/
Stores system and application logs.

Important files:
/var/log/syslog or /var/log/messages → OS/system-level issues.
/var/log/auth.log → SSH authentication failures.
/var/log/dmesg → Kernel & hardware issues.

/etc/
Stores configuration files for system & services.
Example: /etc/ssh/sshd_config for SSH configs, /etc/fstab for mounts.

/home/
User directories (e.g., /home/ec2-user or /home/ubuntu).
SSH keys, custom scripts, Ansible playbooks often stored here.

/tmp/
Temporary files, often used by Terraform, Ansible, and CI/CD jobs.
Useful when debugging failed installations.

/var/run/
Stores runtime info like PID files (important when troubleshooting stuck services).

◆ 2. Terraform

--Terraform itself doesn't install a service, but when running on EC2:

/home/<user>/.terraform.d/
Stores Terraform plugins and credentials.

/var/log/cloud-init.log & /var/log/cloud-init-output.log
Logs from EC2 instance initialization when Terraform provisions instances.
Useful to debug bootstrap scripts.

◆ **3. Ansible**

Ansible is agentless, so it runs from control nodes (like EC2).

/etc/ansible/
Default location for ansible.cfg, inventory files.

/var/log/ansible.log (if logging enabled in ansible.cfg)
Helps debug playbook failures.

~/.ansible/tmp/
Stores temporary files/modules copied to managed nodes during play execution.

◆ **4. Jenkins**

When Jenkins runs on EC2:

/var/lib/jenkins/
Main Jenkins home directory (jobs, builds, plugins, configs).

/var/log/jenkins/jenkins.log
Primary Jenkins log file for troubleshooting job/build issues.

/etc/default/jenkins or /etc/sysconfig/jenkins
Jenkins service startup configuration.

/var/cache/jenkins/
Caches for Jenkins builds.

◆ **5. Docker**

If Docker is installed on EC2:

/var/lib/docker/
Docker images, containers, volumes stored here.
Useful if containers not starting properly.

/var/run/docker.sock
Socket used by Docker CLI/Daemons (permission issues often occur here).

/etc/docker/daemon.json
Docker daemon configuration.

/var/log/docker.log or journalctl logs → journalctl -u docker
Troubleshoot Docker daemon issues.

◆ 6. Kubernetes (when installed on EC2)

--Kubernetes components create several directories:

/etc/kubernetes/
Stores cluster configs like kubelet.conf, admin.conf.

/var/lib/kubelet/
Nde-specific pod and container data.

/var/log/pods/
Pod logs stored on the node (before collected by logging tools).

/var/log/containers/
Symlinked logs from container runtimes.

/etc/cni/net.d/
CNI (Calico/Flannel) networking config files.

/var/lib/etcd/
etcd data directory (if master node).

/var/log/kube-apiserver.log, /var/log/kubelet.log (or journalctl -u kubelet)
Critical logs for API server and kubelet troubleshooting.

◆ 7. Combined Troubleshooting Flow

--When something breaks in EC2 for these tools, you typically check in this order:

System logs → /var/log/syslog or /var/log/messages.

Tool-specific logs/configs:

Ansible → /etc/ansible/, ansible logs/tmp
Kubernetes → /etc/kubernetes/, /var/log/pods/, /var/log/containers/
Terraform → /var/log/cloud-init*, ~/.terraform.d/
Jenkins → /var/lib/jenkins/, /var/log/jenkins/
Docker → /var/lib/docker/, journalctl -u docker

Runtime issues → /tmp/, /var/run/

---EC2 Troubleshooting Cheat Sheet (DevOps Tools)

Tool	Directory / File	Purpose / What to Check
Linux (General)	/var/log/syslog / /var/log/messages	OS/system logs, crashes, reboots
	/var/log/auth.log	SSH login/authentication issues
	/var/log/dmesg	Kernel/hardware/network issues
	/etc/	System & service configs (e.g., /etc/ssh/sshd_config)
	/home/<user>/	User files, SSH keys, scripts
	/tmp/	Temporary files, troubleshooting failed installations
	/var/run/	PID and socket files for services
Terraform	~/.terraform.d/	Plugins, credentials
	/var/log/cloud-init.log	Instance initialization logs
	/var/log/cloud-init-output.log	Debugging provisioning/bootstrap
Ansible	/etc/ansible/	ansible.cfg, inventory
	/var/log/ansible.log *(if enabled)*	Playbook execution logs
	~/.ansible/tmp/	Temporary files/modules copied during play
Jenkins	/var/lib/jenkins/	Jenkins home (jobs, builds, plugins, configs)
	/var/log/jenkins/jenkins.log	Main Jenkins troubleshooting log
	/etc/default/jenkins or /etc/sysconfig/jenkins	Service startup configs
	/var/cache/jenkins/	Build cache files
Docker	/var/lib/docker/	Images, containers, volumes
	/var/run/docker.sock	Docker CLI/daemon socket (common permission issue)
	/etc/docker/daemon.json	Docker daemon config
	journalctl -u docker or /var/log/docker.log	Docker daemon logs
Kubernetes	/etc/kubernetes/	Cluster configs (kubelet.conf, admin.conf)
	/var/lib/kubelet/	Node pod/container data
	/var/log/pods/	Pod logs (per namespace/app)
	/var/log/containers/	Container logs (symlinked to pod logs)
	/etc/cni/net.d/	CNI network configs (Calico/Flannel)
	/var/lib/etcd/	etcd data (master node)
	/var/log/kube-apiserver.log, /var/log/kubelet.log *(or journalctl -u kubelet)*	API server & kubelet troubleshooting

-----All types of healthcheks in AWS

Type of Health Check	Service	Purpose	Layer	Who Acts
EC2 Status Check	EC2	Is the instance's host & OS healthy?	Infrastructure/OS	AWS (System) / You (Instance)
ALB/NLB Health Check	ELB	Is the target able to handle requests?	Application/Network	Load Balancer
ECS Health Check	ECS	Is the app inside the container healthy?	Application (Container)	ECS Service
K8s Liveness/Readiness	EKS	Is the pod alive and ready for traffic?	Application (Container)	Kubelet
Auto Scaling Health Check	EC2 ASG	Should this instance in the fleet be replaced?	Fleet Management	Auto Scaling Group
Route 53 Health Check	Route 53	Should DNS traffic be failed over to a backup?	DNS/Global	Route 53 DNS System

-----How will you troubleshoot slow performance Linux server

◆ 1. Check System Load & CPU Usage

- Run:
- uptime
- top
- htop   # if installed
- mpstat -P ALL 1
- Look for:
  - **Load average** (if it's consistently higher than CPU cores → bottleneck).
  - Processes consuming high CPU.
  - High %iowait (indicating disk bottleneck).

◆ 2. Check Memory Usage

- Run:
- free -m
- vmstat 1 5
- top
- Look for:
  - Very little free memory.
  - High swap usage (system may be swapping → slows performance).

◆ 3. Check Disk I/O

- Run:
- iostat -x 1 5
- df -h
- du -sh /var/log/*
- Look for:
  - High disk utilization (%util close to 100).
  - Full or almost full disks (≥90% usage).
  - Large log files filling storage.

◆ **4. Check Network Bottlenecks**

- Run:
- ifconfig
- ip -s link
- ss -tulpn
- netstat -i
- Look for:
    - Dropped packets.
    - Very high bandwidth usage.
    - Saturated network interfaces.

◆ **5. Check Processes & Services**

- Run:
- ps aux --sort=-%cpu | head
- ps aux --sort=-%mem | head
- systemctl list-units --type=service --state=running
- Look for:
    - Memory/CPU hogging processes.
    - Zombie processes.
    - Services consuming unexpected resources.

◆ **6. Check Logs for Errors**

- Run:
- dmesg | tail -20
- journalctl -xe
- tail -f /var/log/messages
- tail -f /var/log/syslog
- Look for:
    - Kernel errors.
    - Hardware failures (disk, memory, NIC).
    - Application crashes.

◆ **7. Check Application-Specific Issues**

- If it's a web/app server:
    - Is the app handling too many connections?
    - Are database queries slow?
    - Is caching enabled (Redis/Memcached)?
    - Any recent code changes?

◆ **8. Check Hardware Health**

- Run:
- sensors   # CPU temp if lm-sensors installed
- smartctl -a /dev/sda
- Look for overheating, failing disks.

◆ **9. Check Security / Misuse**

- Run:
- who
- w
- last
- ps aux | grep bitcoin
- Look for:
    - Unauthorized logins.
    - Malware/crypto mining processes.

◆ **10. Take Corrective Actions**

- Kill or restart resource-heavy processes.
- Add swap if memory is tight.
- Clear disk space if full.
- Restart hung services.
- If load is consistently high → **scale up** (bigger instance) or **scale out** (load balancing).

----Summary: Your Troubleshooting Cheat Sheet

Symptom	**Top Command**	**Deep Dive Tools**
High CPU	top, htop	pidstat, perf top, vmstat 1
High Memory	top, htop	free -h, vmstat 1, slabtop
High I/O Wait	top (look at %wa)	iostat -xz 1, iotop, pidstat -d
High Load	top (load avg)	All of the above - it's a symptom, not a cause.
Network Slowness	nethogs, iftop	ping, netstat -i, traceroute
Process Issues	top	strace -p <PID>, lsof -p <PID>

------HTTP Error Codes & Meanings

1. **401 Unauthorized**
   - Request is missing or has invalid authentication credentials.
   - User/service is **not logged in** or token/key is invalid.
2. **403 Forbidden**
   - Authentication is successful, but user/service **doesn't have permission**.
   - Identity is valid but lacks required role/authorization.
3. **404 Not Found**
   - The requested resource/path doesn't exist.
   - Could be wrong API endpoint, misconfigured ingress, or wrong service mapping.
4. **Timeout (e.g., 504 Gateway Timeout)**
   - Request took too long, service didn't respond.
   - Could be due to pod crash, app slowness, network routing issues, or wrong service target

Q- I have disk space related issue on linux server. Server is getting slow. How will you troubleshoot?

Answer- "If disk space issues are slowing down the Linux server, I first check usage with df -h to see which partition is full. Then I use du -sh to identify large directories and files, and find to locate oversized files. I also check inodes with df -i, logs in /var/log, and open deleted files with lsof | grep deleted. If I/O is the problem, I monitor using iostat and iotop. To fix it, I clear old logs, configure log rotation, truncate large files, and move or delete unnecessary data. For cloud servers like AWS EC2, I also consider resizing the EBS volume. Finally, I ensure preventive measures like log rotation and monitoring alerts are in place so the issue doesn't reoccur."

---- Increasing disk space on a Linux server

1. **Increase the disk** at the hypervisor/cloud level.
2. **Make the OS aware** of the new disk size.
3. **Grow the partition** to use the new space.
4. **Resize the filesystem** to fill the partition.

-----Increase Volume in AWS

```
1. AWS Console → EC2 → Volumes → Modify Volume
                                          # Increase the EBS volume size.
       # lsblk
       # df -h
2. sudo growpart /dev/xvda 1                   # Grow the Partition
3. sudo resize2fs /dev/xvda1                   # Resize the Filesystem
   OR  sudo xfs_growfs -d /
       # df –h
```

----Find Which Directory is Consuming Space

--Summary:

- du -sh /*
- sudo du -sh /var/* | sort -rh | head -10
- sudo du -ah /var | sort -rh | head -20

- Start with df -h → find full partition.
- Use du -sh + sort → locate heavy dirs.
- Use find/du -ah → spot large files.
- Check /var/log, /tmp, Docker/K8s directories.
- Use ncdu for quick exploration.

----One-liner cmd run to delete log files older than 30 days on a Linux server:

```
# find /var/log -type f -name "*.log" -mtime +30 -exec rm -f {} \;
```

----What is AWS Lambda?

AWS Lambda is a **serverless compute service** that runs code in response to **events** without you needing to manage servers.

◆ AWS Services that Trigger / Work with Lambda

-- Developer Tools

- **AWS CodeCommit** → Repo changes trigger Lambda.
- **AWS CodePipeline** → Can invoke Lambda as part of pipeline.
- **AWS CloudFormation** → Custom resources backed by Lambda.

🎯 Interview-Ready Answer

"AWS Lambda is the serverless compute service, but its real power comes from integrations. It works with services like **S3, DynamoDB, API Gateway, EventBridge, SQS, SNS, Kinesis, CloudFront (Lambda@Edge), Step Functions, and CloudWatch**. This allows Lambda to react to events such as file uploads, database changes, API requests, or scheduled tasks. In short, Lambda can be triggered by most AWS services that produce events, making it the backbone of event-driven architectures in AWS."

A) Input Services (Events that can trigger Lambda)-

These are the **event sources** that invoke Lambda:

1️⃣ Storage & Databases

- **Amazon S3** → Object created/deleted events
- **Amazon DynamoDB Streams** → Table updates
- **Amazon RDS / Aurora** → Database activity events

2️⃣ Messaging & Event Routing

- **Amazon SQS** → New messages in a queue
- **Amazon SNS** → Notifications published
- **Amazon EventBridge (CloudWatch Events)** → Service events or scheduled rules
- **Amazon Kinesis Data Streams / Firehose** → Real-time data streams

3️⃣ Networking & APIs

- **Amazon API Gateway** → REST/HTTP requests
- **Application Load Balancer (ALB)** → Direct routing to Lambda
- **Amazon CloudFront (Lambda@Edge)** → Edge events

4️⃣ Security & Monitoring

- **Amazon CloudWatch Logs** → Log events
- **Amazon CloudWatch Alarms / Metrics** → Alarm state changes
- **AWS Config** → Configuration compliance events
- **Amazon GuardDuty / Security Hub / Macie** → Security findings

5️⃣ IoT & Developer Tools

- **AWS IoT Core** → Device messages
- **AWS CodeCommit** → Repository changes
- **AWS CodePipeline** → Pipeline stages
- **AWS CloudFormation (Custom Resources)** → Stack lifecycle events

---

B) Output Services (Actions Lambda can trigger)-
Once executed, Lambda can **send data or trigger other services**:

1️⃣ Storage & Databases

- **Amazon S3** → Store files or processed output
- **Amazon DynamoDB** → Write application data
- **Amazon RDS / Aurora** → Insert/update database records

2️⃣ Messaging & Event Routing

- **Amazon SQS** → Send messages to a queue
- **Amazon SNS** → Publish notifications
- **Amazon EventBridge** → Send custom events to the event bus
- **Amazon Kinesis** → Push processed records

3️⃣ Networking & APIs

- **Amazon API Gateway** → Return HTTP responses
- **Application Load Balancer (ALB)** → Send back responses
- **External APIs** → Call third-party services (via HTTP/SDKs)

4️⃣ Orchestration & Monitoring

- **AWS Step Functions** → Start workflows
- **Amazon CloudWatch Logs** → Write execution logs
- **Amazon CloudWatch Metrics** → Publish custom metrics

5️⃣ Notifications & Integrations

- **Amazon SES (Simple Email Service)** → Send emails
- **Slack / Microsoft Teams (via webhooks)** → Send notifications
- **Other external systems** → Any endpoint via API destination

---

🎯 **Interview-Ready Summary**
"Lambda can be triggered by many AWS services — like S3, DynamoDB, SQS, SNS, Kinesis, EventBridge, API Gateway, and CloudWatch. Once executed, Lambda can trigger downstream actions such as writing to S3/DynamoDB, sending messages to SNS/SQS/EventBridge, invoking Step Functions, or even calling external APIs. This makes Lambda the core engine for event-driven architectures in AWS."

----What is Amazon CloudWatch?

Amazon CloudWatch is a **monitoring and observability service** in AWS.
It collects, analyzes, and visualizes:

- **Metrics** (like CPU usage, memory, latency, errors)
- **Logs** (from applications or services)
- **Events/Alarms** (automated responses to system changes)

It helps you **monitor applications, infrastructure, and AWS resources** in real-time.

◆ **Key Features of CloudWatch**

Feature	**Description**
Metrics	-Numerical data about resources (e.g., EC2 CPUUtilization, Lambda duration).
Logs	-Stores and analyzes logs from applications and AWS services.
Alarms	-Triggers notifications or actions when metrics cross thresholds.
Dashboards	-Custom visual panels for system health.
Events (Rules)	-Automates responses to specific AWS events (e.g., trigger Lambda).

◆ **Integration of CloudWatch with Lambda**
AWS **Lambda** automatically integrates with **CloudWatch** for:

1. **Logging**
2. **Metrics**
3. **Alarms / Event triggers**

Let's explain each one clearly 👇

☐ **1. CloudWatch Logs with Lambda**

- Every Lambda function automatically **creates a log group** in CloudWatch:
- /aws/lambda/<function-name>
- All your print() or console.log() statements from the Lambda code are stored there.

✅ **Example:**
Suppose you have a Lambda function named process-orders.
When it runs:

```
def lambda_handler(event, context):
    print("Order received:", event)
    return {"status": "Order processed"}
```

AWS will automatically:

- Create a CloudWatch Log Group → /aws/lambda/process-orders
- Store logs for every invocation:
- START RequestId: xxxx Version: $LATEST
- Order received: {'order_id': 123, 'item': 'Laptop'}
- END RequestId: xxxx
- REPORT Duration: 150 ms Memory Used: 90 MB

You can view these logs in **CloudWatch** → **Logs** → **Log groups**.

⚙ 2. CloudWatch Metrics with Lambda

CloudWatch collects **Lambda performance metrics** automatically, such as:

Metric	**Description**
Invocations	-Number of times function executed
Duration	-Time taken by function
Errors	-Count of failed executions
Throttles	-Times Lambda was limited by concurrency
IteratorAge	-For stream-based functions (Kinesis/DynamoDB)
ConcurrentExecutions	-Number of functions running at same time

✅ **Example:**

You can create a **CloudWatch Alarm**:

- Metric: Errors
- Condition: If Errors > 5 in 5 minutes
- Action: Notify via **SNS** or **invoke another Lambda**

🔔 3. CloudWatch Events (EventBridge) with Lambda

You can use **CloudWatch Events** (now part of *Amazon EventBridge*) to **trigger Lambda functions** automatically on specific schedules or AWS events.

✅ **Example: Trigger a Lambda every 5 minutes**

You can create a rule:

```
{
  "source": ["aws.events"],
  "detail-type": ["Scheduled Event"]
}
```

with a **cron expression**:

```
cron(0/5 * * * ? *)
```

This triggers your Lambda function every 5 minutes.

Use Case Examples:

- Automatically clean up old S3 files
- Send hourly system health reports
- Trigger alerts if EC2 CPU > 80%

🧠 Complete Example: Monitoring a Lambda with CloudWatch
Scenario:
You have a Lambda function that processes orders.
You want to:

- View logs of each execution
- Get alerts if it fails
- Automatically retry using CloudWatch alarm

Steps:

1. **Create Lambda Function** – "process-orders"
2. **Lambda automatically sends logs** → CloudWatch Logs
3. **Go to CloudWatch** → **Metrics** → **Lambda** → **process-orders**
4. Create a **CloudWatch Alarm**:
    - Metric: Errors
    - Condition: > 3 errors in 5 mins
    - Action: Send notification to **SNS** → which triggers another Lambda to alert the DevOps team.

✓ In Summary

Feature	**Role in Lambda**
CloudWatch Logs	Captures Lambda function logs
CloudWatch Metrics	Monitors performance and errors
CloudWatch Alarms	Alerts or triggers actions
CloudWatch Events	Triggers Lambda on schedule or event

----Steps to Schedule an AWS Lambda function using CloudWatch Events:

Steps :

1. **Go to AWS Management Console** → Open **Lambda** service.
2. **Create a new Lambda function**
    - Choose *Author from scratch*
    - Name your function
    - Select runtime (e.g., Python, Node.js)
    - Write and deploy your code
3. **Go to CloudWatch (or EventBridge)** → Open **Rules** section.
4. **Create a new rule**
    - Choose **Schedule** as event source
    - Define time using:
        - rate(5 minutes) for periodic execution
        - or cron(0 12 * * ? *) for cron expression
5. **Add target**
    - Select **Lambda function** as target
    - Choose your created Lambda function
6. **Add permission automatically** (CloudWatch can invoke Lambda).
7. **Name and create rule** (e.g., daily-lambda-trigger).
8. **Verify in Lambda** → Under *Configuration* → *Triggers*, you should see CloudWatch rule attached.
9. **Check CloudWatch Logs** for Lambda execution results.

-----AWS Cloudwatch EventBridge?

- **Amazon EventBridge** is a **serverless event bus service**.
- It helps you **connect applications using events**.
- Think of it as a **central hub** where different AWS services, your applications, or SaaS tools can send and receive events.

◆ How It Works

1. **Event Sources**
   - AWS services (like S3, EC2, Lambda, CodePipeline).
   - Custom applications (you can publish your own events).
   - SaaS apps (e.g., Zendesk, Shopify, Datadog).
2. **Event Bus**
   - A channel inside EventBridge that receives and routes events.
   - AWS provides a **default event bus**, but you can create custom ones too.
3. **Rules**
   - Rules filter events based on patterns (e.g., "when an EC2 instance changes state").
   - Each rule defines which events should be sent to which targets.
4. **Targets**
   - Once rules match an event, EventBridge sends it to a **target**.
   - Targets can be: **Lambda, SQS, SNS, Step Functions, Kinesis, ECS, API destinations, etc.**

◆ Example Use Cases

1. **Automating EC2 Events**
   - When an EC2 instance stops → EventBridge triggers a Lambda → Sends Slack alert.
2. **S3 File Processing**
   - File uploaded to S3 → EventBridge rule → Sends event to Step Functions → Orchestrates processing pipeline.
3. **Security & Compliance**
   - AWS Config detects non-compliance → EventBridge → Invokes Lambda to remediate.
4. **SaaS Integration**
   - GitHub PR opened → EventBridge rule → Triggers CodePipeline build.
5. **Scheduled Events**
   - Replace **cron jobs** with EventBridge rules that trigger Lambda at scheduled intervals.

◆ Benefits

- **Serverless** → Fully managed, no servers to maintain.
- **Scalable** → Handles millions of events per second.
- **Flexible** → Works with AWS services, custom apps, and SaaS apps.
- **Decoupled Architecture** → Producers and consumers don't know each other, only the event bus.

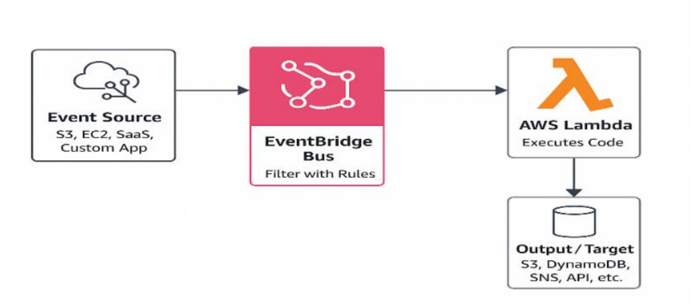
🎯 **Interview-Ready Answer**

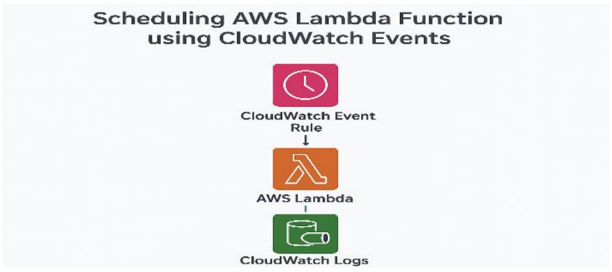
"Amazon EventBridge is a serverless event bus service that allows applications, AWS services, and SaaS apps to communicate using events. It works by receiving events from sources, applying rules to filter them, and then routing them to targets like Lambda, SQS, or Step Functions. Common use cases include automating resource changes, triggering workflows on file uploads, integrating SaaS tools, or running scheduled jobs. EventBridge helps build scalable, decoupled, event-driven architectures without managing infrastructure."

-----AWS Cloudwatch EventBridge with AWS Lambda

◆ **Step-by-Step Flow**

1. **Event Source** → Something happens in AWS (e.g., EC2 stopped, S3 file uploaded) or in a custom/SaaS app.
2. **EventBridge Bus** → The event is sent to EventBridge.
   - Rules filter events (e.g., only EC2 stop events).
   - Routes event to one or many targets.
3. **AWS Lambda** → If Lambda is a target, it gets the event as JSON.
   - Lambda runs your custom code (business logic).
4. **Output / Target** → Lambda can then:
   - Store results in **S3/DynamoDB**
   - Send messages via **SNS/SQS**
   - Start workflows in **Step Functions**
   - Call external APIs





-----Elastic Network Interface (ENI)

- An **ENI** is a **virtual network card** that you can attach to an **EC2 instance** in a VPC.
- It represents a logical **networking component** with the following properties:
  - Primary **private IPv4** address (mandatory).
  - One or more **secondary private IPv4** addresses.
  - One **Elastic IP (EIP)** address per private IP.
  - One or more **security groups**.
  - A **MAC address**.
  - Source/Destination check flag.

◆ **Types of ENIs**

1. **Primary ENI** → Created automatically when you launch an EC2 instance.
2. **Secondary ENI** → Can be attached/detached from instances (even moved to another instance).

◆ Use Cases of ENI

- **High Availability (HA):**
  Move an ENI (with its IPs and security groups) from a failed instance to a standby instance.
- **Network/Traffic Separation:**
  One ENI for management traffic, another for application traffic.
- **Security Isolation:**
  Attach multiple security groups via multiple ENIs.
- **Multiple IPs:**
  Use secondary private IPs for hosting multiple websites/applications on the same EC2.

◆ **Example**

☞ If you have a web server and a monitoring agent, you can attach:

- **ENI-1** → connected to public subnet with security group for HTTP/HTTPS.
- **ENI-2** → connected to private subnet with security group for monitoring traffic.

◆ **Key Points**

- An instance in a VPC always has **at least one ENI** (primary).
- Secondary ENIs can be created and attached manually.
- ENIs can be **moved across instances** (except the primary one).
- Supports **IPv4 & IPv6**.

✅ **Interview-ready 1-liner:**
"An Elastic Network Interface (ENI) is a virtual network card in AWS EC2 that holds IPs, MAC, and security groups. It allows flexible networking, such as moving IPs between instances for high availability or separating traffic."

----◆ Common EC2 Instance Classes in Real-Time Production

1. General Purpose (Balanced)

- **Families**: t3, t4g, m5, m6i, m7g
- **Use case**: Web servers, microservices, small/medium databases.
- ✅ Example:
  - t3.medium, t3.large → cost-effective for low-to-medium workloads.
  - m5.large, m5.xlarge → stable production workloads.

---

2. Compute Optimized

- **Families**: c5, c6i, c7g
- **Use case**: High-performance computing, APIs with heavy CPU load, data processing.
- ✅ Example: c5.xlarge, c6i.2xlarge → used for **batch jobs, ML inference, gaming servers**.

---

3. Memory Optimized

- **Families**: r5, r6i, x1, z1d
- **Use case**: Large in-memory databases, caching (Redis, Memcached), analytics.
- ✅ Example: r5.large, r6i.xlarge → used for **databases like PostgreSQL, MySQL, MongoDB**.

---

4. Storage Optimized

- **Families**: i3, i4i, d3
- **Use case**: Workloads needing **high IOPS storage** (NoSQL DBs, ElasticSearch, Hadoop).
- ✅ Example: i3.large, i3en.xlarge.

---

5. Accelerated Computing (GPU/ML)

- **Families**: p3, p4, g4, g5, inf1
- **Use case**: Machine learning training, GPU rendering, AI workloads.
- ✅ Example: p3.2xlarge, g5.xlarge.

◆ Real-Time Production Choices (Typical)

- **Web Applications / Microservices** → m5.large or t3.medium (autoscaling).
- **Databases (RDS/EC2-hosted DBs)** → r5.large or r6i.xlarge.
- **Batch Processing / APIs** → c5.xlarge.
- **Big Data (ElasticSearch, Kafka)** → i3.xlarge.
- **Machine Learning** → p3.2xlarge or g5.xlarge.

🎯 **Interview-Ready Answer** *"In real-time production, the instance type depends on workload. For web apps we often use m5.large or t3.medium, for databases r5.large, for compute-heavy APIs c5.xlarge, and for ML workloads GPU instances like p3.2xlarge. We usually combine them with Auto Scaling Groups to handle variable traffic and optimize cost."*

-----What is t3a.xlarge in AWS EC2

t3a.xlarge is an **Amazon EC2 instance type** that belongs to the **T3a family** — which is part of the **"burstable general purpose" instances** in AWS.

It's designed to provide a **balance of compute, memory, and network performance** for workloads that don't need full CPU all the time but occasionally "burst" to higher performance.

⚙ Basic Configuration of t3a.xlarge

Parameter	**Specification**
vCPUs	4 virtual CPUs
Memory (RAM)	16 GiB
Network Performance	Up to 5 Gbps
Storage	EBS-Only (Elastic Block Store)
Architecture	64-bit (x86_64 / AMD)
Processor Type	AMD EPYC (2.5 GHz base clock)
EBS Bandwidth	Up to 2,880 Mbps
Baseline CPU Utilization	40% (burstable to 100%)
Burstable Performance	Yes – uses CPU Credits
Supported Virtualization	HVM (Hardware Virtual Machine)

⚡ How Burstable Performance Works

T3a instances use a **CPU Credit System**:

- When the instance runs **below baseline CPU**, it **earns CPU credits**.
- Each **credit = 1 vCPU running at 100% for 1 minute**.
- When the instance **needs more CPU**, it uses stored credits to "burst".
- Ideal for workloads with **occasional spikes**.

You can run T3a instances in two modes:

1. **Standard Mode** – earns and spends credits (default).
2. **Unlimited Mode** – can burst beyond credits (you pay extra if you exceed baseline).

☐ T3a Instance Type Variants

Instance Type	**vCPUs**	**Memory (GiB)**	**Use Case Example**
t3a.nano	2	0.5	Tiny apps, testing
t3a.micro	2	1	Small websites, dev servers
t3a.small	2	2	Low traffic apps
t3a.medium	2	4	Web servers, staging
t3a.large	2	8	Small databases, app servers
t3a.xlarge	4	16	Medium databases, app clusters
t3a.2xlarge	8	32	Multi-user apps, analytics, batch workloads

☐ **Common Use Cases for t3a.xlarge**
✅ Web servers and APIs
✅ Application servers (medium workloads)
✅ Development and test environments
✅ Build servers (Jenkins, CI/CD)
✅ Small to medium databases
✅ Microservices with variable load

💰 **Why Choose t3a.xlarge**

- **Cost-efficient** (10% cheaper than Intel T3)
- **Good performance burst** for variable workloads
- **Flexible scaling** using CPU credits
- **Ideal for steady + Occasional peak usage**

⚖️ **Comparison: t3a.xlarge vs m5.xlarge**

Feature	t3a.xlarge	m5.xlarge
Instance Family	T3a (Burstable General Purpose)	M5 (General Purpose – Fixed Performance)
vCPUs	4	4
Memory (RAM)	16 GiB	16 GiB
Processor Type	AMD EPYC	Intel Xeon Platinum 8175M / 8259CL
Base CPU Performance	Burstable (baseline ~40%)	Constant full CPU
CPU Credits System	✅ Yes (Earn/Use credits)	❌ No (Always full CPU)
Network Performance	Up to 5 Gbps	Up to 10 Gbps
EBS Bandwidth	Up to 2.8 Gbps	Up to 4.75 Gbps
Cost	💰 ~10–20% cheaper	💰💰 More expensive
Use Case	Variable workloads with occasional bursts	Constant or CPU-intensive workloads
Performance Consistency	Variable (depends on credits)	Consistent (always full CPU)
Ideal For	Web servers, APIs, dev/test, small DB	Databases, analytics, backend services
Burstable Mode	✅ Yes (Standard/Unlimited)	❌ No
Region Availability	Available in most AWS regions	Available in all major regions
Pricing Example (us-east-1)	~$0.1344/hour	~$0.192/hour

🧠 **Simple Understanding**

Scenario	Recommended Instance
You need **low-cost** instance for **fluctuating workloads** (apps, dev, testing)	☐ **t3a.xlarge**
You need **consistent CPU power** (databases, heavy computation)	🧱 **m5.xlarge**

✅ **Summary**

If you want:	**Choose:**
Cost-saving with occasional performance spikes	**t3a.xlarge**
Steady, predictable high performance	**m5.xlarge**

☐ **Example Use Cases**

t3a.xlarge	**m5.xlarge**
CI/CD pipelines (Jenkins, GitLab runners)	Database servers (RDS, PostgreSQL, MySQL)
Microservices / APIs	Application servers with constant load
Web servers (Nginx, Apache)	Data processing or analytics
Development / QA environment	Production-grade backend systems

---

-----Burstable General Purpose Instance in AWS

A **Burstable General Purpose Instance** (like the **T3**, **T3a**, or **T4g** families) in AWS EC2 is a type of **virtual machine** that provides a **baseline level of CPU performance**, and the ability to **"burst" above that baseline** when your application needs more power.

In simple words — It normally runs at a modest CPU level to save cost, but can temporarily "boost" to high performance during spikes in workload.

How It Works (Concept)

AWS uses a **CPU Credit System** to manage this burstable performance:

1. **Baseline Performance:**
    - Each instance earns **CPU credits per hour** based on its size (e.g., t3a.small earns fewer credits than t3a.xlarge).
    - These credits allow a certain percentage of CPU usage (like 20–40% of a full vCPU).
2. **Credit Earning:**
    - When your instance uses **less CPU than baseline**, it **earns credits**.
    - Credits are saved in a "credit bucket".
3. **Burst Mode:**
    - When the workload spikes (e.g., high traffic or batch job), it **spends saved CPU credits** to use **100% CPU power** for a short time.
4. **Back to Normal:**
    - After the burst, the instance goes back to baseline and continues earning credits again.

Example - Imagine you have a **T3a.medium** web server:

- During normal hours, your app uses 10% CPU → you **earn credits**.
- Suddenly, traffic spikes → the instance **bursts to 100% CPU** using those credits.
- Once the spike is over → it returns to normal usage.

You only pay for a **low-cost instance**, but still get **temporary high performance** when needed.

------Should we deploy our app on a VM (traditional deployment) or on Kubernetes (container orchestration)?

---

1. VM-based Deployment (Traditional / Lift & Shift)

✅ **When to Choose**

- **Simple apps** with low complexity (e.g., a single web app + DB).
- **Monolithic architecture** (all in one package, not microservices).
- **Low scalability needs** — small traffic, occasional scaling by adding another VM.
- **Legacy code** that is not container-ready.
- **Tight deadlines** (fastest to set up).

⚡ **Characteristics**

- One VM = One app (or a few apps).
- Scaling → add bigger VM or spin up more VMs (manual or autoscaling).
- Easier for **small teams** (less infra overhead).
- Infra managed by **AWS EC2, GCP Compute Engine, Azure VM**.

---

2. Kubernetes-based Deployment (Modern / Cloud-Native)

✅ **When to Choose**

- **Microservices architecture** (app split into multiple services: auth, catalog, payments).
- **High scalability** required (apps serving millions of requests).
- **Frequent releases** → CI/CD heavy workflows.
- **Containerized workloads** (Docker images ready).
- **Multiple environments** (Dev, QA, Staging, Prod namespaces).
- Need for **service discovery, auto-healing, rolling updates**.

⚡ **Characteristics**

- Runs **containers** (Docker).
- Scaling → horizontal pod autoscaler (HPA).
- Built-in **service discovery & load balancing**.
- Better **resource utilization** (multiple services per node).
- Infra managed by **EKS (AWS), GKE (Google), AKS (Azure), OpenShift, self-managed K8s**.

---

3. Differentiating Environments (Dev, QA, Staging, Prod)

You differentiate environments based on:

1. **Code Branch/Tag Strategy**
   - feature/* → Dev
   - release/* → QA/Staging
   - main + vX.Y.Z tag → Production
2. **Service Configuration**
   - Dev: Debug mode ON, lightweight DB, fewer replicas.
   - QA: Test datasets, service mocks, medium infra.
   - Prod: Full scaling, real DB, monitoring, alerts enabled.
3. **Infrastructure Setup**
   - VM route → Separate VMs per environment (e.g., dev-app.vm, qa-app.vm, prod-app.vm).
   - Kubernetes route → Separate namespaces (dev, qa, staging, prod) in the same or different clusters.

-----AWS Secrets Manager

AWS Secrets Manager is a **fully managed service** that helps you **store, manage, and retrieve secrets** securely.

- A "secret" can be a **database password, API key, OAuth token, or any sensitive credential**.
- It integrates with AWS services (like RDS, EC2, EKS, Lambda) to fetch secrets securely at runtime.
- Supports **automatic rotation**, **encryption with AWS KMS**, and **fine-grained access control** via IAM.

◆ Advantages of AWS Secrets Manager

1. **No Hardcoding of Secrets** – Credentials are never stored in code/config files.
2. **Automatic Secret Rotation** – Built-in support for rotating RDS/MySQL/Postgres credentials.
3. **IAM-based Access Control** – Only authorized apps/users can retrieve secrets.
4. **Encryption at Rest** – All secrets encrypted with **AWS KMS** keys.
5. **Audit & Compliance** – Every secret access is logged in **AWS CloudTrail**.
6. **Centralized Management** – One place to manage secrets across multiple applications and environments.

◆ Steps to Create and Access Secrets

Step 1: Create Secret in AWS Console

1. Go to **AWS Secrets Manager** service.
2. Click **Store a new secret**.
3. Choose **Credentials for RDS database** (or Other type for API keys, etc.).
4. Enter key-value pairs, e.g.:
```
{
  "username": "dbadmin",
  "password": "MySecurePass123"
}
```
5. Choose **Encryption Key** (default AWS KMS key is fine).
6. Name your secret (e.g., myAppDBSecret).
7. Save it.

Step 2: Access Secret via AWS CLI

```
# aws secretsmanager get-secret-value --secret-id myAppDBSecret --query SecretString --output text
```

✅ **Summary:**

- **AWS Secrets Manager** is a secure way to manage credentials.
- **Advantages**: automatic rotation, IAM security, encryption, audit logging.
- **Steps**: Create → Store → Access via CLI/SDK at runtime.

Here’s the **full Terraform code** with **Create & Use AWS Secrets Manager Secret**:

◆ Terraform Code – Create & Use AWS Secrets Manager Secret

```
provider "aws" {
  region = "us-east-1"
}

            # 1. Create a secret
resource "aws_secretsmanager_secret" "db_secret" {
  name        = "myAppDBSecret"
  description = "Database credentials for 3-tier application"
}

            # 2. Store secret value (username & password as JSON)
resource "aws_secretsmanager_secret_version" "db_secret_value" {
  secret_id     = aws_secretsmanager_secret.db_secret.id
  secret_string = jsonencode({
    username = "dbadmin"
    password = "MySecurePass123!"
  })
}

            # 3. IAM role for EC2/EKS app tier to access the secret
resource "aws_iam_role" "app_role" {
  name = "app-tier-role"
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"
      Principal = {
        Service = "ec2.amazonaws.com" # or "eks.amazonaws.com"
      }
      Action = "sts:AssumeRole"
    }]
  })
}

            # 4. IAM policy to allow access to the secret
resource "aws_iam_role_policy" "secret_access_policy" {
  name = "app-secret-access"
  role = aws_iam_role.app_role.id

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"
      Action = [
        "secretsmanager:GetSecretValue"
      ]
      Resource = aws_secretsmanager_secret.db_secret.arn
    }]
  })
}
```

```
          # 5. Fetch secret values (latest version)
data "aws_secretsmanager_secret" "db_secret" {
  name = aws_secretsmanager_secret.db_secret.name
}

data "aws_secretsmanager_secret_version" "db_secret_value" {
  secret_id = data.aws_secretsmanager_secret.db_secret.id
}

          # 6. Decode secret JSON into local variables
locals {
  db_credentials =
jsondecode(data.aws_secretsmanager_secret_version.db_secret_value.secret
_string)
}

          # 7. Example usage: Create an RDS instance using the secret
resource "aws_db_instance" "mydb" {
  allocated_storage    = 20
  engine               = "mysql"
  engine_version       = "8.0"
  instance_class       = "db.t3.micro"
  db_name              = "myappdb"

          # 8. Use secret values from Secrets Manager
  username             = local.db_credentials["username"]
  password             = local.db_credentials["password"]

  skip_final_snapshot  = true
}
```

◆ **How This Works**

1. **aws_secretsmanager_secret** → Creates a secret container.
2. **aws_secretsmanager_secret_version** → Stores the actual secret (username & password in JSON).
3. **aws_iam_role** → Role for EC2/EKS app tier.
4. **aws_iam_role_policy** → Grants permission to fetch the secret.
5. Added **data sources** aws_secretsmanager_secret & aws_secretsmanager_secret_version to fetch the stored secret.
6. Used **jsondecode** to parse the JSON (username, password).
7. Passed the values into **aws_db_instance** to provision an RDS DB with credentials stored in Secrets Manager.

-----Auto Scaling

Auto Scaling means **automatically adjusting computing resources** (like servers, containers, or pods) based on the **current load or demand**.

☐ Why Auto Scaling Is Important

- ⚙️ Handles variable workloads automatically
- 💰 Reduces cost by scaling down during low traffic
- 🧠 Improves performance and reliability
- 🔄 Avoids manual intervention for resource management

◆ Types of Auto Scaling (Based on Where You Use It)

Let's break this into **AWS** and **Kubernetes** contexts.

☐ In AWS

AWS provides **EC2 Auto Scaling** and **EKS/ECS Auto Scaling**.

1. Dynamic Scaling

- Automatically increases or decreases capacity **based on metrics**.
- Metrics include CPU utilization, memory, request count, etc.
- Example:
  - If CPU > 80% → Add EC2 instance
  - If CPU < 30% → Remove EC2 instance

☐ Subtypes of Dynamic Scaling:

Type	Description
Target Tracking Scaling	Maintains a target metric (e.g., 50% CPU). Most commonly used.
Step Scaling	Adds/removes instances in steps (e.g., add 2 if CPU > 70%, add 4 if CPU > 90%).
Simple Scaling	One rule triggers one scaling action. Older method.

---

2. Scheduled Scaling

- Scaling occurs at **specific times** you define.
- Example:
  - Add more EC2s every weekday at 9 AM (business hours)
  - Scale down at 8 PM

⏰ Used when workload patterns are **predictable**.

---

3. Predictive Scaling

- Uses **machine learning** to forecast future traffic and adjusts capacity **before** load changes.
- Example:
  - Predicts traffic spike at 10 AM → scales up in advance.

🧠 Best for steady, recurring traffic patterns (e.g., e-commerce websites).

☐ In Kubernetes (EKS, GKE, AKS)

Kubernetes offers 3 layers of auto scaling:

1. Horizontal Pod Autoscaler (HPA)

- Scales **Pods** based on CPU, memory, or custom metrics.
- Example:
  If CPU > 80%, increase pod replicas from 3 → 6.

```
# kubectl autoscale deployment myapp --min=3 --max=10 --cpu-percent=80
```

2. Vertical Pod Autoscaler (VPA)

- Adjusts **resources (CPU/RAM)** of pods automatically.
- Example:
  If a pod constantly uses more memory → increase its limits.

---

3. Cluster Autoscaler

- Scales **nodes (EC2 instances)** in the cluster based on pending pods.
- Example:
  If there aren’t enough nodes to schedule new pods → adds a new EC2 node.

✅ **Summary Table**

Layer	**Environment**	**Resource Scaled**	**Example**
Dynamic Scaling	AWS	EC2 instances	Add EC2 when CPU > 80%
Scheduled Scaling	AWS	EC2 instances	Scale up at 9 AM
Predictive Scaling	AWS	EC2 instances	Forecasts traffic patterns
HPA	Kubernetes	Pods	Add pods if CPU > 70%
VPA	Kubernetes	Pod resources	Increase memory limit
Cluster Autoscaler	Kubernetes	Nodes	Add node if pods pending

-----What is Disaster Recovery (DR) in AWS

Disaster Recovery (DR) in AWS is a **strategy and process** to recover your IT systems, data, and applications after a **disaster** (such as system failure, data corruption, cyberattack, or natural disaster).

The main goal of DR is to **minimize downtime (RTO)** and **data loss (RPO)** so your business can **quickly resume operations**.

☐ What We Do in DR

In a Disaster Recovery plan, we focus on:

1. **Data Backup**
   - Regularly backing up data (e.g., databases, files, configurations).
   - Store backups in multiple AWS Regions or Availability Zones.
2. **Replication**
   - Continuously replicate data and applications from the primary (production) site to a secondary (DR) site.
3. **Automation**
   - Use Infrastructure as Code (IaC) tools like **CloudFormation** or **Terraform** to quickly recreate infrastructure in another region.
4. **Failover & Failback**
   - Automatically switch to the DR environment (failover) when production fails.
   - Return to the main site (failback) after recovery.
5. **Testing**
   - Regularly test the DR process to ensure reliability.
6. **Monitoring & Alerts**
   - Use AWS monitoring tools to detect failures early and trigger DR actions.

⚙ AWS Tools and Services Used in DR

Service	**Purpose in DR**
AWS Backup	Centralized backup for EC2, RDS, EFS, DynamoDB, etc.
Amazon S3 + Cross-Region Replication (CRR)	Store and replicate data across regions.
Amazon RDS Read Replicas / Multi-AZ	High availability and replication for databases.
AWS Elastic Disaster Recovery (DRS)	Fully managed service for server replication and recovery.
AWS CloudFormation / Terraform	Automate infrastructure recovery.
Route 53	DNS failover to redirect traffic to the DR site.
CloudWatch + SNS	Monitoring and alerts during failure events.
AWS Lambda	Automate failover scripts or DR orchestration.
AWS Organizations & IAM	Secure access and governance during DR.

☐ Different DR Strategies (Methods)

AWS defines **four main Disaster Recovery strategies**, depending on cost and recovery time:

Strategy	Description	RTO/RPO	Cost	Example
1. Backup & Restore	Regularly back up data to S3 or Glacier; restore when disaster occurs.	High (hours to days)	💰 Low	S3 backups, EBS snapshots
2. Pilot Light	Keep minimal infrastructure (like databases) always running; rest of infra launched during DR.	Medium	💰💰 Medium	RDS live, EC2 created later
3. Warm Standby	Run a scaled-down version of the app in another region; scale up during DR.	Low	💰💰💰 Medium–High	Smaller EC2 instances in DR region
4. Multi-Site (Active-Active)	Both regions are active and handle traffic simultaneously.	Very Low (seconds)	💰💰💰💰 High	Global Load Balancing using Route 53

🛡 How to Prevent or Minimize DR

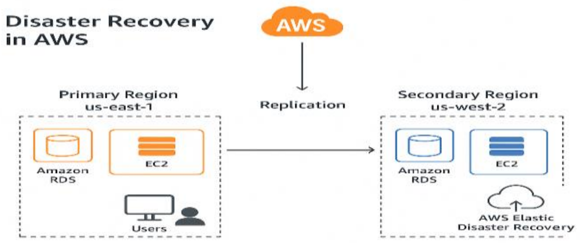
You can't always prevent disasters, but you can **reduce the impact** with:

- **Multi-AZ & Multi-Region deployment**
- **Auto Scaling** to handle failures automatically
- **Regular Backups**
- **Automated monitoring and alerts**
- **Security best practices** (IAM policies, GuardDuty, WAF)
- **Testing and simulation** of DR events

✅ Example Scenario

Imagine you have an e-commerce app hosted in **us-east-1**:

- Data replicated to **us-west-2** using **S3 CRR** and **RDS Read Replica**
- **Route 53** configured for DNS failover
- **AWS Elastic Disaster Recovery** replicates EC2 instances to the secondary region
- When the primary region fails, traffic automatically shifts to **us-west-2** with minimal downtime.



-----What is AWS Elastic Disaster Recovery (AWS DRS)

AWS Elastic Disaster Recovery (DRS) is a **fully managed service** that helps you **minimize downtime and data loss** during disasters by **replicating your on-premises or cloud-based servers** (EC2 or physical/virtual machines) to AWS in real time.

When a disaster occurs, AWS DRS lets you **quickly launch recovery instances** (copies of your servers) in AWS with just a few clicks.

⚙️ How AWS Elastic Disaster Recovery Works

Here's the **step-by-step process** 👇

1. Install the Replication Agent

- You install the **AWS Replication Agent** on your source servers (these could be on-premises or EC2 instances).
- This agent continuously replicates data (disk changes) to AWS.

2. Continuous Data Replication

- The agent replicates data **at the block level** (not file level) to a **Staging Area Subnet** in AWS.
- This staging area uses low-cost storage (EBS snapshots + small EC2 instances), so it's **cost-efficient**.

3. Maintain Continuous Sync

- The replication is **asynchronous** but near real-time.
- If your source system changes (new data, updates), those changes are automatically synced to AWS.

4. Launch Recovery Instances (Failover)

- When a **disaster** happens (like system crash, region failure, ransomware, etc.):
  - You trigger a **Failover** from the AWS console.
  - AWS DRS quickly **launches fully functional EC2 instances** from your replicated data in the target region.
- These instances mirror your original servers — same configuration, OS, applications, and data.

5. Failback (Return to Normal)

- Once your primary site is back online, you can use **Failback** to sync the latest data **back from AWS to your original environment**, ensuring no data loss.

Key Components in AWS DRS

Component	**Description**
Source Server	The original system you are protecting.
Replication Agent	Software installed on the source server to replicate data to AWS.
Staging Area Subnet	Low-cost area in AWS to store replicated data.
Recovery Instance	EC2 instance launched during failover (your live copy).
Failback Agent	Used to send data back to the source after recovery.

🚀 **Benefits of AWS Elastic Disaster Recovery**
✅ **Low-cost** – Uses cheap storage until recovery needed.
✅ **Fast recovery** – Launches servers in minutes during DR.
✅ **Cross-region support** – Protect workloads across AWS regions.
✅ **Simple testing** – You can run DR drills without affecting production.
✅ **Automation** – Integrated with CloudFormation, Lambda, Route53 for failover automation.

🌍 **Example: How It Works During a Disaster**
You have a web app running on **on-premises servers** or in **us-east-1** region.
You've configured **AWS Elastic Disaster Recovery** to replicate to **us-west-2**.

Normal Time:

- Replication Agent sends continuous data to AWS staging area.
- Everything stays synced but idle (low cost).

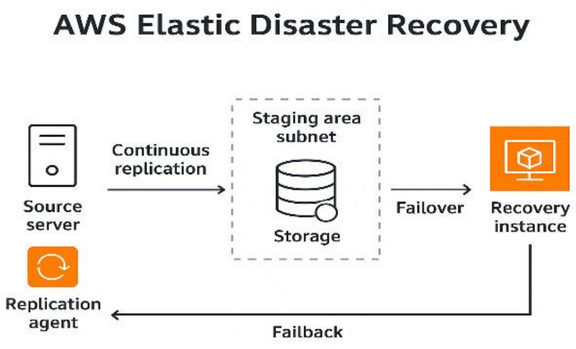
When Disaster Happens:

1. Your primary servers go down.
2. You open the AWS console → **Elastic Disaster Recovery** → **Launch Recovery Instances.**
3. AWS spins up EC2 servers in **us-west-2** (your DR site).
4. DNS (Route 53) redirects users to the DR region automatically.
5. Your app resumes operation with minimal downtime (a few minutes).

After Disaster:

- Once your primary servers are fixed,
- Perform **Failback** → data syncs from AWS to your original site.
- Turn off recovery instances to save cost.

🧠 **Simple Analogy**
Think of AWS DRS as **a live clone of your system** sitting quietly in AWS.
When your real system fails — the clone **wakes up instantly** and takes over.



Step-by-Step Explanation of how to create backups for AWS VMs (EC2 instances):

☐ 1. Methods to Create Backups

You can back up EC2 instances in three main ways:

1. **Create Amazon Machine Image (AMI)** — Full backup of the instance (OS + data + configuration)
2. **Create EBS Snapshots** — Backup of volumes attached to the instance
3. **Automate Backups** — Using AWS Backup or Lifecycle Manager (DLMM)

☐ 2. Create Backup Using Amazon Machine Image (AMI)

An AMI is a snapshot of the entire instance, including:

- The root volume (OS, applications)
- Any attached EBS volumes (if selected)
- Instance configuration (network, security groups, etc.)

◆ **Steps:**

1. Go to the **AWS Management Console**
2. Open **EC2 Dashboard**
3. Select **Instances**
4. Select the instance you want to back up
5. Click **Actions** → **Image and templates** → **Create image**
6. Enter:
   - **Image name** (e.g., webserver-backup-2025-11-08)
   - **Description**
   - Select whether to include **additional volumes**
7. Click **Create image**
8. AWS will create an **AMI** and corresponding **EBS snapshots**

✓ **Benefits:**

- Can launch new instances using this AMI anytime.
- Great for full instance recovery or cloning environments.

🖫 3. Create Backup Using EBS Snapshot

EBS snapshots are **incremental backups** of your EBS volumes (root or data volumes).

◆ **Steps:**

1. Open **EC2 Dashboard**
2. Select **Volumes** under **Elastic Block Store**
3. Choose the volume you want to back up
4. Click **Actions** → **Create snapshot**
5. Enter a **description** (e.g., "Database Volume Backup")
6. Click **Create snapshot**

✓ **Benefits:**

- Snapshots are stored in **Amazon S3 (managed by AWS)**
- Can restore volumes or create new ones
- Fast and cost-effective (incremental storage)

⚙ 4. Automate Backups Using AWS Backup or Data Lifecycle Manager

☐ Option 1: AWS Backup Service

A centralized backup service that supports EC2, RDS, EFS, DynamoDB, etc.

Steps:

1. Go to **AWS Backup Console**
2. Click **Create backup plan**
3. Choose:
   - Backup frequency (daily, weekly, etc.)
   - Retention period
4. Assign **resources** (select EC2 instances)
5. Backups will be created and managed automatically

✅ Supports cross-region and cross-account backup copies

☐ Option 2: Data Lifecycle Manager (DLMM)

Automates **EBS snapshots** creation and deletion.

Steps:

1. Go to **EC2** → **Lifecycle Manager**
2. Click **Create Lifecycle Policy**
3. Choose **EBS Snapshot Policy**
4. Set:
   - **Target volumes**
   - **Backup schedule**
   - **Retention period**
5. Save the policy — AWS will handle periodic backups automatically

---

🧠 5. Restore Process

- **From AMI:**
  Go to **EC2** → **AMIs** → **Launch instance from image** → Configure → Launch → Restore your VM
- **From Snapshot:**
  Go to **EC2** → **Snapshots** → **Create volume from snapshot** → Attach to a new or existing instance

---

🔐 6. Best Practices

✅ Use **AWS Backup** for centralized management
✅ Enable **encryption** for all backups
✅ Store backups in **multiple regions** (cross-region copies)
✅ Tag your backups (e.g., Environment=Prod, Backup=true)
✅ Periodically **test restore procedures**

🏁 Summary

Method	Type	Use Case	Automation
AMI	Full VM backup	Entire instance recovery	Manual
EBS Snapshot	Volume-level backup	Data or OS volume restore	Manual or DLMM
AWS Backup	Centralized backup	Enterprise-wide backup management	Automated

---

Examples to automate EC2 backups (both **AWS CLI** and **Terraform**) in simple and real-time formats.

☐ 1. Automate EC2 Backup Using AWS CLI

✅ Create an AMI Backup of an EC2 Instance

```
aws ec2 create-image \
  --instance-id i-0abcd1234efgh5678 \
  --name "ec2-backup-$(date +%Y-%m-%d)" \
  --description "Daily backup of production EC2 instance" \
  --no-reboot
```

Explanation:

- --instance-id → ID of your EC2 instance
- --name → Backup name with current date
- --no-reboot → Prevent instance from rebooting during backup
- The command creates an **AMI** and **EBS snapshots**

✅ Create EBS Snapshot Backup

```
aws ec2 create-snapshot \
  --volume-id vol-0abcd1234efgh5678 \
 --description "Daily snapshot of database volume"
```

Optional (tag your snapshot):

```
aws ec2 create-tags \
  --resources snap-0abcd1234efgh5678 \
  --tags Key=Environment,Value=Production
Key=Backup,Value=True
```

Automate it daily:
You can use **AWS Lambda + CloudWatch Event Rule (Scheduler)** to trigger this CLI command every day at a specific time.

⚙ 2. Automate EC2 Backup Using Terraform

You can automate AMI or EBS snapshots creation and retention with Terraform.

✅ Example 1: Create EC2 Instance + Daily AMI Backup

```
provider "aws" {
  region = "ap-south-1"
}

resource "aws_instance" "web" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t3.micro"
  tags = {
    Name = "webserver"
  }
}

# Lifecycle policy to automate AMI backups
resource "aws_dlm_lifecycle_policy" "ec2_ami_backup" {
  description        = "Daily AMI backup policy for EC2 instance"
  execution_role_arn =
"arn:aws:iam::123456789012:role/AWSDataLifecycleManagerDefaultRole"
  state              = "ENABLED"

  policy_details {
    resource_types = ["INSTANCE"]

    schedule {
      name = "DailyBackup"
      create_rule {
        interval      = 24
        interval_unit = "HOURS"
        times         = ["02:00"]
      }
      retain_rule {
        count = 7
      }

      tags_to_add = {
        Backup = "Daily"
      }
    }

    target_tags = {
      Backup = "True"
    }
  }

  tags = {
    Name = "Daily-AMI-Backup"
  }
}
```

Explanation:

- The policy automatically creates **daily AMI backups**
- Keeps **last 7 days** of backups
- Targets only EC2 instances **tagged with Backup=True**
- Works automatically without manual commands

✅ **Example 2: Create EBS Snapshot Backup Policy**

```
resource "aws_dlm_lifecycle_policy" "ebs_snapshot_backup" {
  description        = "Daily EBS snapshot backup"
  execution_role_arn =
"arn:aws:iam::123456789012:role/AWSDataLifecycleManagerDefaultRole"
  state              = "ENABLED"

  policy_details {
    resource_types = ["VOLUME"]

    schedule {
      name = "DailySnapshot"
      create_rule {
        interval      = 24
        interval_unit = "HOURS"
        times         = ["01:00"]
      }

      retain_rule {
        count = 5
      }
    }

    target_tags = {
      Backup = "True"
    }
  }

  tags = {
    Name = "Daily-EBS-Backup"
  }
}
```

🧠 Best Practice Automation Setup

Component	Tool	Purpose
EC2 Snapshot Creation	AWS CLI / Terraform	Automate volume backups
Schedule Backups	AWS Lambda + CloudWatch	Time-based automation
Centralized Backup Management	AWS Backup	Multi-service backup policy
Infrastructure as Code	Terraform	Consistent and repeatable setup

---

-----E**nhanced version** of your **daily backup shell script** that not only backs up a folder at **7 PM every day** but also **automatically uploads the backup file to an AWS S3 bucket** for safe cloud storage.

☐ 1. Script Objective

Backup local folder (e.g., /home/ec2-user/project)
→ Store backup locally (e.g., /backup/daily/)
→ Upload backup file to **Amazon S3 bucket** (e.g., s3://my-project-backup-bucket/)
→ Run automatically at **7:00 PM every day**

🌀 2. Shell Script: daily_backup_s3.sh

```
#!/bin/bash
# ---------------------------------------------------
# Script Name: daily_backup_s3.sh
# Description: Daily folder backup and upload to S3 at 7 PM
# Author: Sudarshan
# ---------------------------------------------------

# === Configuration ===

SOURCE_DIR="/home/ec2-user/project"       # Folder to back up
DEST_DIR="/backup/daily"                  # Local backup storage folder
S3_BUCKET="s3://my-project-backup-bucket"
                                          # S3 bucket name (replace this)
RETENTION_DAYS=7                          # Keep local backups for 7 days
LOG_FILE="/var/log/backup.log"            # Log file path

# === Preparation ===

mkdir -p "$DEST_DIR"

# === Generate backup filename ===

DATE=$(date +"%Y-%m-%d_%H-%M")
BACKUP_NAME="project_backup_$DATE.tar.gz"
BACKUP_PATH="$DEST_DIR/$BACKUP_NAME"

# === Create the backup ===

echo "[$(date)] 🚀 Starting backup..." >> "$LOG_FILE"
tar -czf "$BACKUP_PATH" "$SOURCE_DIR"

if [ $? -eq 0 ]; then
    echo "[$(date)] ✅ Local backup created: $BACKUP_PATH" >> "$LOG_FILE"
else
```

```
    echo "[$(date)] ❌ Backup creation failed for $SOURCE_DIR" >> "$LOG_FILE"
    exit 1
fi

# === Upload to AWS S3 ===
aws s3 cp "$BACKUP_PATH" "$S3_BUCKET/" --storage-class STANDARD_IA

if [ $? -eq 0 ]; then
    echo "[$(date)] ☁ Backup uploaded to S3: $S3_BUCKET/$BACKUP_NAME" >> "$LOG_FILE"
else
    echo "[$(date)] ❌ Failed to upload backup to S3" >> "$LOG_FILE"
    exit 1
fi

# === Remove old backups (older than RETENTION_DAYS) ===
find "$DEST_DIR" -type f -mtime +$RETENTION_DAYS -name "*.tar.gz" -exec rm -f {} \;
echo "[$(date)] ☐ Old backups older than $RETENTION_DAYS days deleted." >> "$LOG_FILE"

echo "[$(date)] 🎉 Backup job completed successfully!" >> "$LOG_FILE"

# === Send Success Email ===
echo -e "Backup completed successfully.\n\nBackup File: $BACKUP_PATH\nS3 Location: $S3_BUCKET/$BACKUP_NAME\nDate: $(date)\n" > "$EMAIL_BODY"
aws ses send-email \
  --from "$FROM_EMAIL" \
  --destination "ToAddresses=$TO_EMAIL" \
  --message "Subject={Data='$SUBJECT_SUCCESS',Charset=utf8},Body={Text={Data='$(cat $EMAIL_BODY)',Charset=utf8}}" \
  2>> "$LOG_FILE"

echo "[$(date)] 🎉 Backup job completed successfully and email sent." >> "$LOG_FILE"
```

⚙ 3. Setup Steps

Step 1 — Save and make executable

```
# sudo nano /home/ec2-user/daily_backup_s3.sh
# chmod +x /home/ec2-user/daily_backup_s3.sh
```

Step 2 — Configure AWS CLI

Make sure your system has AWS CLI configured:

```
aws configure
```

Enter:

- AWS Access Key ID
- AWS Secret Access Key
- Default region (e.g., ap-south-1)

Also ensure the IAM user/role has **S3 full access or put-object permissions**.

⏰ 4. Create a Cron Job (Run Daily at 7 PM)

Open cron:

```
crontab -e
```

Add this line:

```
# 0 19 * * * /home/ec2-user/daily_backup_s3.sh
```

This runs the script **every day at 7:00 PM**.

☐ 5. Example Log Output (/var/log/backup.log)

```
[Sat Nov  8 19:00:00 2025] 🚀 Starting backup...
[Sat Nov  8 19:00:02 2025] ✅ Local backup created:
/backup/daily/project_backup_2025-11-08_19-00.tar.gz
[Sat Nov  8 19:00:10 2025] ☁ Backup uploaded to S3: s3://my-project-
backup-bucket/project_backup_2025-11-08_19-00.tar.gz
[Sat Nov  8 19:00:10 2025] ☐ Old backups older than 7 days deleted.
[Sat Nov  8 19:00:10 2025] 🎉 Backup job completed successfully!
```

✅ 6. Optional Enhancements

- Use **aws s3 sync** to mirror all backups:
- aws s3 sync /backup/daily s3://my-project-backup-bucket/
- Send email/SNS notification after upload success/failure
- Encrypt backups before uploading (e.g., gpg)
- Store in **S3 Glacier** for long-term archival:

```
aws s3 cp "$BACKUP_PATH" "$S3_BUCKET/" --storage-class GLACIER
```

----Different types of data base in aws

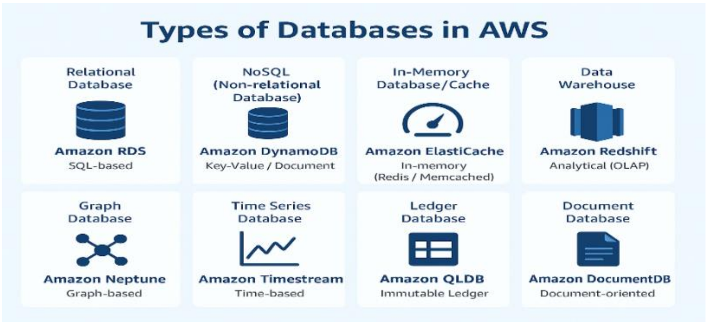
AWS offers **different types of databases** to handle various use cases like relational, non-relational (NoSQL), in-memory, time-series, graph, and more.

💡 Simplified Real-world Mapping

Use Case	Best AWS Database
Traditional web app (e.g., e-commerce site)	Amazon RDS / Aurora
High-traffic apps needing low latency	DynamoDB
Caching frequent queries	ElastiCache
Analytics / BI workloads	Redshift
Social networking / recommendation	Neptune
IoT and time-stamped metrics	Timestream
Financial transactions / auditing	QLDB
JSON / semi-structured data	DocumentDB

✅ Example Interview Tip

"AWS offers different databases for different data models. RDS and Aurora are relational, DynamoDB is NoSQL for scalability, Redshift is analytical, ElastiCache is in-memory, Neptune is graph-based, and Timestream handles time-series data. Each service is fully managed and optimized for its purpose."



-----Relational and Non-relational databases

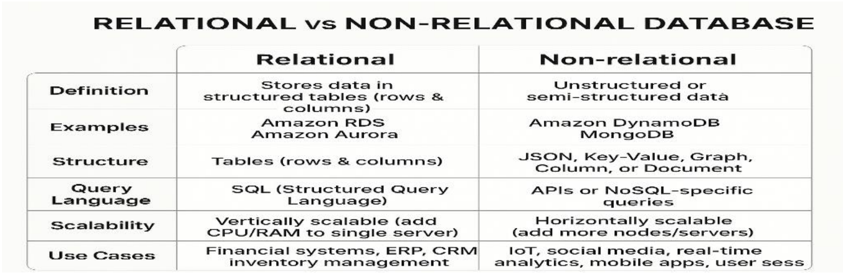
Type	Description
Relational Database (RDBMS)	Stores data in **structured tables** (rows & columns). Each table has relationships (foreign keys) with others. Amazon RDS, Amazon Aurora, MySQL, PostgreSQL, Oracle, SQL Server
Non-relational Database (NoSQL)	Stores **unstructured or semi-structured data** like JSON, key-value pairs, graphs, or documents — no strict schema or relations. Amazon DynamoDB, MongoDB, Cassandra, CouchDB, Redis

✓ Quick Summary

Aspect	Relational	Non-relational
Data Format	Structured	Unstructured / Semi-structured
Schema	Fixed	Dynamic
Scalability	Vertical	Horizontal
Query	SQL	API / JSON-based
Relationships	Supported (Foreign keys)	Not required
Example (AWS)	RDS, Aurora	DynamoDB, DocumentDB
Ideal For	Structured data, ACID transactions	Big data, high scalability, flexible schema

Simple Example :

- **Relational (SQL):** A banking system — accounts and transactions with strict schema and relations.
- **Non-relational (NoSQL):** An e-commerce site storing user carts and product catalogs in JSON — flexible and fast.



RELATIONAL vs NON-RELATIONAL DATABASE

	Relational	Non-relational
Definition	Stores data in structured tables (rows & columns)	Unstructured or semi-structured data
Examples	Amazon RDS Amazon Aurora	Amazon DynamoDB MongoDB
Structure	Tables (rows & columns)	JSON, Key-Value, Graph, Column, or Document
Query Language	SQL (Structured Query Language)	APIs or NoSQL-specific queries
Scalability	Vertically scalable (add CPU/RAM to single server)	Horizontally scalable (add more nodes/servers)
Use Cases	Financial systems, ERP, CRM inventory management	IoT, social media, real-time analytics, mobile apps, user sess

--Git &GitHub----Syllabus(with One-Line Explanations)

1. Introduction

- **What is Git?** → A distributed version control system.
- **What is GitHub?** → Cloud-based platform for hosting Git repositories and collaboration.
- **Git vs GitHub** → Git is the tool, GitHub is the service.

---

2. Git Basics

- **Installing Git** → Setup via package managers.
- **Git Configuration** → Set username, email (git config).
- **Repositories** → Local and remote repo concepts.
- **Basic Workflow** → git add, git commit, git push, git pull.

---

3. File Lifecycle & Staging

- **Working Directory** → Local changes.
- **Staging Area (Index)** → Prepares files for commit.
- **Repository (History)** → Permanent commit storage.
- **git status** → Shows current file states.

---

4. Branching & Merging

- **Branches** → Isolated lines of development.
- **git branch** → Create/list branches.
- **git checkout / switch** → Move between branches.
- **Merging** → Combine changes into a branch.
- **Fast-forward vs Recursive Merge** → Merge strategies.

---

5. Collaboration with GitHub

- **Cloning Repos** → git clone from GitHub.
- **Remote Management** → git remote add, git fetch.
- **Push & Pull** → Sync local and remote repos.
- **Pull Requests (PRs)** → Propose and review changes.
- **Forks** → Copy of repo for independent work.

---

6. Git Logs & History

- **git log** → View commit history.
- **git diff** → Show file differences.
- **git blame** → Show who changed each line.
- **Tags** → Mark versions/releases.

---

7. Undoing Changes

- **git restore** → Discard local changes.
- **git reset** → Unstage/rollback commits.
- **git revert** → Safely undo commits.
- **git clean** → Remove untracked files.

---

8. Advanced Branching Strategies

- **Feature Branch Workflow** → Isolate features.
- **Gitflow Workflow** → Structured release branches.
- **Trunk-Based Development** → Frequent small commits to main.
- **Rebasing** → Reapply commits on top of another branch.
- **Cherry-Pick** → Apply specific commits.

---

9. Stashing & Patching

- **git stash** → Save temporary work.
- **git stash pop/apply** → Reapply stashed work.
- **git format-patch** → Generate patch files.

---

10. Collaboration & Open Source

- **Issues** → Track bugs and tasks.
- **Projects** → Kanban-style boards for work tracking.
- **GitHub Discussions** → Team communication.
- **GitHub Actions** → CI/CD automation.

---

11. Security & Access

- **SSH Keys** → Secure repo access.
- **Personal Access Tokens** → Authenticate GitHub CLI/API.
- **Branch Protection Rules** → Prevent direct pushes.
- **Code Owners** → Auto-assign reviewers.

---

12. Git Tools & Integrations

- **Submodules** → Manage nested repos.
- **Hooks** → Automate Git actions.
- **Git LFS** → Manage large files.
- **Integrations** → GitHub with Slack, Jira, CI/CD tools.

---

13. Best Practices

- **Commit Messages** → Write meaningful messages.
- **Small Commits** → Keep changes manageable.
- **Code Reviews** → Improve code quality.
- **Branch Naming Conventions** → Feature/bug/hotfix branches.

-----Git Commands

1️⃣ git init
Scenario: You created a new project manually and want to start tracking it in Git.
```
# git init
```

2️⃣ git clone
Scenario: Your team lead asks you to download the project from GitHub.
```
# git clone https://github.com/company/project.git
```

3️⃣ git status
Scenario: You made some changes and want to know which files are modified but not committed.
```
# git status
```

4️⃣ git add
Scenario: You updated 3 files but only want to stage one file.
```
# git add app.py
```

5️⃣ git commit
Scenario: After staging changes, you want to save them with a message.
```
# git commit -m "Fixed login bug"
```

6️⃣ git log
Scenario: You want to see the history of commits with author and timestamp.
```
# git log
```

7️⃣ git diff
Scenario: You want to see what changes were made before committing.
```
# git diff
```

8️⃣ git push
Scenario: After committing, you want to upload code to GitHub main branch.
```
# git push origin main
```

9️⃣ git pull
Scenario: Your teammate pushed new code and you want to sync your repo.
```
# git pull origin main          = Fetch+Merge
```

🔟 git fetch
Scenario: You want to see remote changes **without merging** into your local branch.
```
# git fetch origin
```

1️⃣2️⃣ git checkout
Scenario: You want to switch to the development branch.
```
# git checkout dev
```

1️⃣1️⃣ git branch
Scenario: You want to create a new branch for a new feature.
```
# git branch feature-login
```

13 git merge
Scenario: You finished a feature and want to merge it into main.
```
# git checkout main
# git merge feature-login
```

---

14 git reset
Scenario: You committed a wrong file and want to undo last commit (but keep changes).
```
# git reset --soft HEAD~1
```

---

15 git revert
Scenario: You want to undo a commit **safely without deleting history**.
```
# git revert <commit-id>
```

---

16 git stash
Scenario: You are working on a file but need to switch branch urgently without committing.
```
# git stash
# git checkout dev
# git stash pop
```

---

17 git remote
Scenario: You want to check which remote repository is linked to your project.
```
# git remote -v
```

---

18 git tag
Scenario: You want to mark a stable release version like v2.1
```
# git tag v2.1
# git push origin v2.1
```

---

19 git cherry-pick
Scenario: You want to copy one specific commit from dev to main without merging whole branch.
```
# git cherry-pick <commit-id>
```

---

20 git rebase
Scenario: You want to keep clean commit history instead of merge commits.
```
# git rebase main
```

🔥 Most asked Interview Questions Based on These Commands

Question	**Short Hint Answer**
Difference between git merge and git rebase?	Merge → keeps history, Rebase → rewrites history (cleaner).
git reset vs git revert?	Reset deletes history, Revert creates new commit.
git pull vs git fetch?	Pull = fetch + merge, Fetch = only download.
What is staging area in Git?	Area before commit (after git add).
When do we use git stash?	To save work temporarily without commit.

◆ **Difference between git merge and git rebase**

Feature	**git merge**	**git rebase**
Purpose	Combines two branches	Moves/rewrites commits on top of another branch
Commit history	Creates a **new merge commit** – history is preserved	**Rewrites history** – linear and cleaner
Safe to use on shared branches?	✅ Yes, safe	❌ No, not safe for shared branches (rewrites history)
History style	Shows branching structure	Looks like all work happened in a straight line
Used when	You want to keep full history	You want clean linear history (no merge commits)

--Example

You are on feature branch and want to update it with latest main code.

1. Using merge:

```
# git checkout feature
# git merge main
```

Creates a merge commit:

```
A---B---C (main)
\
D---E (feature)
\
M (merge commit)
```

2. Using rebase:

```
# git checkout feature
# git rebase main
```

Rewrites history into linear form:

```
A---B---C---D'---E' (feature rebased)
```

✅ Rebase = cleaner history

! But **never rebase a branch already pushed & shared with team**

◆ **Difference between git reset and git revert**

Feature	**git reset**	**git revert**
Purpose	Moves branch pointer to previous commit	Creates new commit that undoes changes
Affects history	❌ Yes (changes commit history)	✅ No (preserves history)
Safe for shared branch?	❌ No (rewrites history)	✅ Yes (recommended for public branches)
Types	--soft, --mixed, --hard	Only one type
Usage example	Undo last commit completely	Undo commit but keep record of change

Example

# git reset --hard HEAD~1

Removes last commit and its changes completely (dangerous).
Before: A --- B --- C (HEAD)
After : A --- B (HEAD) ← commit C deleted

# git revert <commit-id>

Creates new commit that reverses the changes from commit C.
A --- B --- C --- C' (revert)
✅ revert = safe, good for production
! reset = good only for local cleanup (before push)

🔥 **Interview Quick Answers**

✅ **When to use merge?**
When working in team and want to keep full history.

✅ **When to use rebase?**
When you want clean linear history for a feature branch before merging.

✅ **When to use reset?**
When you want to undo local commits before pushing.

✅ **When to use revert?**
When you want to undo changes in remote/shared branch safely.

✅ Real-time Git Workflow Used in Companies (Feature Branch Workflow) (in DevOps, Agile, and CI/CD environments.)

🔄 **Workflow Steps**

1. **main or master branch**
   - Always contains **production-ready/stable code**
   - Protected branch (cannot push directly)
2. **develop branch (optional in some companies)**
   - Integrated code for next release
   - Used for testing (QA/UAT)
3. **Feature Branch**
   - Developer creates a new branch for a task/feature/bug fix
4. git checkout -b feature/login-api
   - Work is isolated from others
5. **Commit & Push**
6. git add .
7. git commit -m "Added login API"
8. git push origin feature/login-api
9. **Create Pull Request (PR) / Merge Request (MR)**
   - Developer requests review from team
   - Code review, quality check, sonar, unit tests, pipelines run
10. **Merge to develop or main**
    - Usually squash merge or rebase + merge
11. **Release Branch (optional)**
    - Created before production release
      Example: release/v1.5
12. **Hotfix Branch**
    - For production bugs   # git checkout -b hotfix/payment-issue

---

◈ **"You committed wrong code in production, what will you do?"**

```
# git log                        # find bad commit ID
# git revert <commit-id>
# git push origin main
```

-----Explain - Hotfix Branch in production , and how to fix this

A **Hotfix branch** is a special Git branch created **directly from the production branch (usually main or master)** to fix urgent bugs found in production.

-Hotfix branches are used when:

- A critical bug is detected in production
- The development team cannot wait for the next release
- The fix must be done **quickly and safely** without disturbing ongoing development work on dev or feature branches

-Hotfix prevents the team from:

- Touching the main codebase directly
- Interrupting other developers working on a release
- Delivering incomplete features accidentally

☐ Why Hotfix Branch is Needed

-Imagine this scenario:

- Production is running from main
- Developers are working on new features in dev or feature branches
- Suddenly a **payment issue** appears in production
- You cannot push to main directly
- You cannot merge dev because it contains new features not ready for release

⍈ So you create a **hotfix branch** ONLY to fix the bug.

Step-1 : **How to Create a Hotfix Branch (Currently Working on Dev)**

-Example:

```
# git checkout main
# git pull origin main
# git checkout -b hotfix/payment-issue
```

-Now you fix the code and commit:

```
# git add .      # git commit -m "Fix: payment gateway crash issue"
```

Step-2 : **How to Apply Hotfix to Production**

-Step 1: Push the hotfix branch

```
# git push origin hotfix/payment-issue
```

-Step 2: Create a Pull Request (PR/MR)

Source → hotfix/payment-issue
Target → main

Once approved → merge to main

-Step 3: Deploy main to production

Your CI/CD pipeline automatically deploys the fix.

Step-3 : **Important Step: Merge Hotfix Back into Dev**

-Because hotfix code must be available in future releases.

-After merging into main, merge it into development:

```
# git checkout dev            # git pull origin dev
# git merge main      // OR merge hotfix directly
# git push origin dev
```

-This prevents:

✕ Same bug happening again ✕ Feature branches missing the fix

----Git Branching Strategy

A **Git branching strategy** is a **set of rules or patterns** that teams follow to manage branches in Git.

It defines **how developers create, name, merge, and manage branches** for features, bug fixes, releases, and production support.

The goal is to:

- Organize development work.
- Avoid conflicts.
- Support different environments (dev, QA, prod).
- Enable smooth releases.

◆ Why Do We Need a Branching Strategy?

- Multiple developers work on the same project → prevents code conflicts.
- Different environments (dev, staging, prod) need different versions of code.
- Faster releases → small changes can be merged without waiting.
- Better collaboration → clear process for code reviews.

◆ Common Git Branching Strategies

1. Feature Branching

- Each new feature/bug is developed in a separate branch.
- Merged back to main (or develop) after completion.
- Example:
  - Branch: feature/login-api
  - Merge: git merge feature/login-api

---

2. Git Flow

- A structured model with multiple branches.
- Branch types:
  - main → production-ready code.
  - develop → integration branch.
  - feature/* → new features.
  - release/* → pre-release testing.
  - hotfix/* → urgent fixes.
- Good for **large teams with multiple releases**.

---

3. GitHub Flow

- Very simple.
- Branch from main → create feature branch → open Pull Request → merge to main → deploy.
- Best for **continuous deployment**.

---

4. GitLab Flow

- Similar to GitHub Flow but adds **environment branches** like dev, staging, prod.
- Good for teams with **multiple deployment environments**.

---

5. Trunk-Based Development

- Developers work on short-lived branches (1–2 days max).
- Frequently merge into main (the trunk).
- Often used with **CI/CD + feature flags**.
- Best for **microservices and fast-moving teams**.

◆ Example: Choosing Strategy

- **Small startup, frequent releases** → GitHub Flow.
- **Enterprise with long QA cycles** → Git Flow.
- **CI/CD with microservices** → Trunk-Based Development.
- **Multiple environments** → GitLab Flow.

-----Git tagging strategy

🚀 Jenkins CI/CD Workflow (Example)

1. **Developer pushes code** → merged into main branch.
2. **Tag is created** → e.g., git tag -a v1.4.2-qa -m "QA release 1.4.2"
3. **Push tag to remote** → git push origin v1.4.2-qa
4. **Jenkins detects tag push** → pipeline runs.
   - If tag ends with -dev → deploy to **Dev** EKS namespace.
   - If tag ends with -qa → deploy to **QA** namespace.
   - If tag ends with -staging → deploy to **Staging** namespace.
   - If tag matches plain vX.Y.Z → deploy to **Production** namespace.
5. **Notification** → Jenkins sends Slack/Email + updates Jira/GitHub Release.

✅ Example Tagging Flow for v2.0.0 Release

1. Developer tests locally → v2.0.0-dev
2. QA testing → v2.0.0-qa
3. Pre-production validation → v2.0.0-staging
4. Final release → v2.0.0

-----Let's build a **Jenkinsfile snippet** that uses the **Git tag naming convention** to decide which **EKS namespace** to deploy into.

📝 Jenkinsfile (Tag-based EKS Deployment)

```
pipeline {
    agent any

    environment {
        AWS_REGION = "ap-south-1"           // change as per your setup
        CLUSTER_NAME = "ecommerce-eks"   // change as per your setup
    }

    stages {
        stage('Checkout') {
            steps {
                checkout scm
                script {
                                                // Get tag name
                    TAG_NAME = sh(script: "git describe --tags --abbrev=0",
returnStdout: true).trim()
                    echo "Building for Git Tag: ${TAG_NAME}"

                                                // Decide environment
                    if (TAG_NAME.endsWith("-dev")) {
                        DEPLOY_ENV = "dev"
                    } else if (TAG_NAME.endsWith("-qa")) {
                        DEPLOY_ENV = "qa"
                    } else if (TAG_NAME.endsWith("-staging")) {
                        DEPLOY_ENV = "staging"
                    } else {
                                    // Default = production (no suffix in tag)
                        DEPLOY_ENV = "prod"   }
                    echo "Deploying to EKS namespace: ${DEPLOY_ENV}"
            }
        }
    }

    stage('Build Docker Image') {
        steps {
            script {
                sh """
                    docker build -t myrepo/ecommerce-app:${TAG_NAME} .
                    docker push myrepo/ecommerce-app:${TAG_NAME}
                """
            }
        }
    }

    stage('Deploy to EKS') {
        steps {
            script {
                sh """
                    aws eks --region ${AWS_REGION} update-kubeconfig --
name ${CLUSTER_NAME}
                    kubectl set image deployment/ecommerce-app ecommerce-
container=myrepo/ecommerce-app:${TAG_NAME} -n ${DEPLOY_ENV}
                    kubectl rollout status deployment/ecommerce-app -n
${DEPLOY_ENV}
                """
            }
        }
    }
}

post {
    success {
        echo "✅ Deployment successful to ${DEPLOY_ENV} with tag
${TAG_NAME}"
    }
    failure {
        echo "❌ Deployment failed for ${TAG_NAME}"
    } } }
```

🔎 **How it Works**

1. **Extract Tag** → Jenkins fetches the Git tag (v1.4.2-qa).
2. **Decide Namespace** → suffix (-dev, -qa, -staging, or nothing = prod`).
3. **Build & Push Docker Image** → uses ${TAG_NAME} for traceability.
4. **Deploy to EKS** → updates the deployment in the right namespace.
5. **Post Actions** → send Slack/Email (can be added).

✅ **Example Runs**

- Tag: v1.4.2-dev → deploys to **Dev namespace**
- Tag: v1.4.2-qa → deploys to **QA namespace**
- Tag: v1.4.2-staging → deploys to **Staging namespace**
- Tag: v1.4.2 → deploys to **Production namespace**

📌 Git Tagging Strategies with Examples & Use Cases :

Strategy	Description	Example Tags	Use Case
Semantic Versioning	Follows MAJOR.MINOR.PATCH format for releases.	v1.0.0, v2.3.1	Standard software releases, open-source projects, libraries, APIs.
Pre-release Tags	Adds suffix for alpha, beta, or release candidate builds.	v1.2.0-alpha, v1.2.0-beta1, v1.2.0-rc1	Testing builds, QA validation, preview releases for early adopters.
Environment-based	Tags indicate deployment environment.	v1.4.2-dev, v1.4.2-qa, v1.4.2-staging, v1.4.2 (prod)	CI/CD pipelines to trigger environment-specific deployments.
Date-based Versioning	Uses release date as tag. Useful for frequent or nightly builds.	2025.09.18, 2025.09.18-nightly	SaaS apps, nightly builds, daily deployments where date matters more than version.
Build/CI Number	Includes CI/CD build number or commit hash for traceability.	v1.0.0+build123, release-4567, v1.2.0-githash	Enterprise CI/CD pipelines, artifact tracking, debugging deployments.
Hybrid Strategy	Combines SemVer with environment or build metadata for clarity.	v2.0.1-qa+build78, v3.1.0-staging-rc2	Large projects with multiple teams, enterprise CI/CD with strict traceability.
Lightweight Tags	Simple tag pointing to a commit (no metadata).	git tag v1.0.0	Internal milestones, quick references, developers marking checkpoints.
Annotated Tags	Full tag with message, author, timestamp (recommended for official releases).	git tag -a v1.0.0 -m "Release v1.0.0"	Production releases, open-source project releases, long-term maintained builds.

-----Environments (Dev → QA → Staging → Prod)

In **DevOps**, software moves through multiple **environments** from development to production.
Each environment serves a specific purpose in testing, validation, and release of applications.

🌱 1. Development Environment (Dev)

Purpose:
Where developers write, build, and test code locally or in shared environments.
Work done:

- Writing and testing new code or features.
- Unit testing (testing individual components).
- Code commits and pushes to version control (like GitHub, Bitbucket).
- CI/CD pipeline builds triggered automatically (using Jenkins, GitHub Actions, etc.).
- Container creation (Docker build).

Tools used:
VS Code, Git, Jenkins, Docker, SonarQube, Nexus.

☐ 2. Testing / QA Environment

Purpose:
Used by QA engineers to verify application behavior and quality before release.
Work done:

- Integration testing (how services interact).
- Regression testing (ensure new code doesn't break existing features).
- Security scanning and code analysis (Trivy, OWASP, SonarQube).
- Test automation (Selenium, JUnit, PyTest).
- Generating test reports and sharing with developers.

Tools used:
Jenkins, Selenium, SonarQube, Trivy, JMeter.

⚙️ 3. Staging / Pre-Production Environment

Purpose:
Simulates the **production environment** as closely as possible. Used for final validation.
Work done:

- Final end-to-end testing.
- Load and performance testing.
- UAT (User Acceptance Testing).
- Infrastructure testing (Terraform, Helm, Kubernetes manifests).
- CI/CD deployment verification.

Tools used:
Kubernetes (EKS), Helm, Prometheus, Grafana, Terraform, Jenkins.

🚀 4. Production Environment (Prod)

Purpose:
The **live environment** accessed by real users.
Work done:

- Deployment of stable, tested code.
- Monitoring and alerting (CloudWatch, Grafana, Prometheus).
- Backup and recovery setup.
- Auto Scaling and Load Balancing (ALB/NLB).
- Incident response and rollback if needed.

Tools used:
AWS (EKS, EC2, CloudWatch), Grafana, Prometheus, Datadog.

☁ 5. Disaster Recovery (DR) / Backup Environment (Optional)

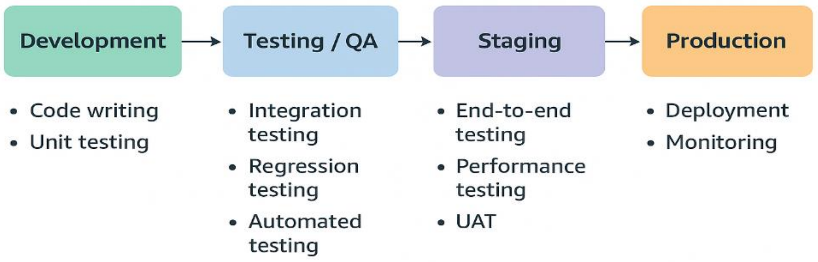
Purpose:
Used for **high availability** and **business continuity** in case the production system fails.
Work done:

- Data replication and backup configuration.
- Failover setup and testing.
- Regular recovery drills.

Tools used:
AWS S3, Route53 failover, RDS Multi-AZ, Backup services.

☐ Summary Table

Environment	**Purpose**	**Key Activities**	**Common Tools**
Development	Build & test new code	Code writing, unit testing	Git, Jenkins, Docker
Testing / QA	Validate features	Integration, regression, automation tests	SonarQube, Selenium
Staging	Final pre-prod check	UAT, performance testing	Kubernetes, Helm
Production	Live environment	Deployment, monitoring	EKS, CloudWatch, Grafana
DR / Backup	Failover protection	Backup & recovery	S3, Route53, RDS



----Cloud Computing Service Models

Cloud services are generally divided into **three main categories**:

Layer	Name	Managed By	Example Use Case
☐ **IaaS**	Infrastructure as a Service	User manages OS, apps	You control servers, networking
⚙ **PaaS**	Platform as a Service	AWS manages runtime	You focus on code deployment
💻 **SaaS**	Software as a Service	Fully managed by AWS/vendor	You use the app directly

---

◆ 1. IaaS (Infrastructure as a Service)

☞ **You manage most things** — OS, applications, runtime, etc.
AWS only provides **raw infrastructure**: compute, storage, network.

☐ **What AWS Manages**

- Physical servers
- Networking
- Virtualization layer

👨‍💻 **You Manage**

- OS, patches, security
- Applications and scaling

📦 **AWS IaaS Services**

Service	Description
Amazon EC2	Virtual machines in AWS. You manage OS, packages, scaling, etc.
Amazon EBS	Block-level storage for EC2.
Amazon S3	Object storage — you manage data, AWS manages infrastructure.
Amazon VPC	Virtual network configuration.
Elastic Load Balancer (ELB)	Distributes traffic between EC2 instances.
AWS IAM	Identity and access control (used with all layers).

🧠 **Example Use Case**
You want full control over your servers (e.g., install Jenkins manually on EC2, manage patching and scaling).

◆ 2. PaaS (Platform as a Service)

☞ AWS manages **servers, OS, runtime, scaling**.
You only manage **your code and data**.

☐ **What AWS Manages**

- Infrastructure
- Operating system
- Runtime environment
- Auto scaling, patching

👨‍💻 **You Manage**

- Application code
- Configuration

🎁 AWS PaaS Services

Service	Description
AWS Elastic Beanstalk	Deploy web apps — AWS handles EC2, ELB, scaling automatically.
Amazon RDS	Managed database (MySQL, PostgreSQL, etc.) — AWS handles backups, patching.
Amazon EKS / ECS (Managed)	You define containers; AWS manages control plane/scaling.
AWS Lambda	Serverless compute — just run code, no server management.
AWS Fargate	Run containers without managing EC2 instances.
AWS API Gateway	Create/manage APIs easily.

🧠 Example Use Case

You push your application code to Elastic Beanstalk or Lambda — AWS handles provisioning and scaling automatically.

◆ 3. SaaS (Software as a Service)

☞ **Fully managed applications**.

You just **use** the software — no control over infrastructure or platform.

☐ What AWS Manages

- Everything: infrastructure, OS, runtime, and the app itself.

👩‍💻 You Manage

- Only user settings and data (if applicable)

🎁 AWS SaaS Services / Examples

Service	Description
Amazon WorkMail	Managed business email service.
Amazon Chime	Video conferencing (like Zoom).
AWS Managed Microsoft AD	Managed Active Directory.
Amazon QuickSight	Business analytics and dashboards.
AWS Backup / AWS Config	Fully managed backup and compliance services.

🧠 Example Use Case

You use **QuickSight** to analyze business data or **WorkMail** for company email — you don't manage any infrastructure.

◆ 📊 Summary Table

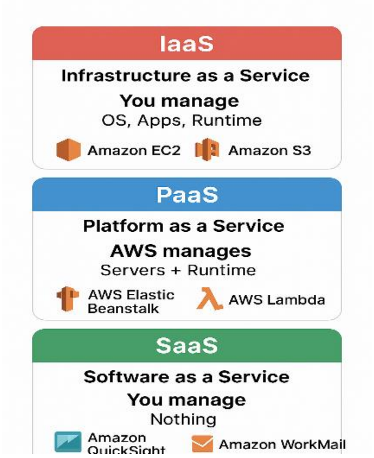
Feature	IaaS	PaaS	SaaS
User manages	OS, Apps, Runtime	App Code only	Nothing (just use app)
AWS manages	Servers, Storage	Servers + Runtime	Everything
Control level	High	Medium	Low
Examples (AWS)	EC2, EBS, VPC, S3	RDS, Lambda, EKS, Beanstalk	QuickSight, WorkMail, Chime

🎯 Quick Memory Tip

Model	Focus On	Keyword
IaaS	Servers	"You manage"
PaaS	Code	"Deploy fast"
SaaS	Software	"Use only"

☐ Real Example (DevOps Scenario)

Stage	Service	Category	Role
Build	Jenkins on EC2	IaaS	You maintain instance and Jenkins setup
Deploy	Elastic Beanstalk	PaaS	AWS deploys your web app
Monitor	Amazon QuickSight	SaaS	You view reports only



--Ansible-- ----Syllabus (with One-Line Explanations)

1. Introduction to Ansible

- **What is Ansible?** → Open-source automation tool for configuration, orchestration, and provisioning.
- **Ansible Architecture** → Control node, managed nodes, inventory, modules, and playbooks.
- **Agentless Model** → Uses SSH/WinRM (no agents needed).

---

2. Setup & Installation

- **Installing Ansible** → Setup via package managers (yum, apt, pip).
- **Configuration Files** → Main config (ansible.cfg).
- **Inventory** → Define hosts and groups (/etc/ansible/hosts).

---

3. Ansible Core Concepts

- **Modules** → Reusable units to manage systems (e.g., file, yum, copy).
- **Tasks** → Individual actions executed by Ansible.
- **Plays & Playbooks** → YAML files describing automation steps.
- **Ad-hoc Commands** → Quick one-time tasks without playbooks.

---

4. Variables & Facts

- **Variables** → Store reusable values.
- **Facts** → System information gathered from hosts.
- **Variable Precedence** → Order of variable resolution.
- **Vault** → Encrypt sensitive variables.

---

5. Conditionals, Loops & Handlers

- **Conditionals** → Run tasks based on conditions.
- **Loops** → Repeat tasks for multiple items.
- **Handlers** → Triggered tasks upon change (e.g., restart service).

---

6. Templates & Files

- **Jinja2 Templates** → Dynamic file creation.
- **copy & template Modules** → Push files to hosts.
- **lineinfile & blockinfile** → Modify file content.

---

7. Roles & Reusability

- **Roles** → Structured way to organize playbooks.
- **Role Directory Structure** → tasks, handlers, vars, templates, files.
- **Ansible Galaxy** → Community roles and collections repository.

8. Ansible Collections & Plugins

- **Collections** → Distribute roles, modules, and plugins.
- **Lookup Plugins** → Fetch external data.
- **Callback Plugins** → Customize output.
- **Filter Plugins** → Modify data in playbooks.

---

9. Ansible Networking & Cloud

- **Network Automation** → Manage routers, switches (Cisco, Juniper, etc.).
- **Cloud Modules** → AWS, Azure, GCP resource provisioning.
- **Dynamic Inventory** → Fetch hosts dynamically from cloud providers.

---

10. Error Handling & Debugging

- **Error Handling** → Use ignore_errors, failed_when.
- **Debugging** → Use debug module for troubleshooting.
- **Check Mode** → Dry-run execution without applying changes.

---

11. Advanced Ansible

- **Ansible Tower / AWX** → Web UI & RBAC for Ansible automation.
- **Ansible Vault** → Encrypt playbooks and secrets.
- **Ansible Tags** → Run selected parts of playbooks.
- **Ansible Parallelism** → Control execution speed with forks.

---

12. CI/CD & Integrations

- **Jenkins + Ansible** → Automate deployments in pipelines.
- **GitOps with Ansible** → Infrastructure as Code (IaC).
- **Ansible with Kubernetes** → Automating container environments.

---

13. Best Practices

- **Idempotency** → Ensure repeatable, predictable runs.
- **Organizing Playbooks** → Modular and reusable structure.
- **Testing with Molecule** → Test Ansible roles and playbooks.

Command	**Explanation**
ansible --version	Check installed Ansible version
ansible all -m ping	Test connectivity to all hosts
ansible all -a "uptime"	Run a shell command on all hosts
ansible -i inventory.ini all -m ping	Ping hosts using a specific inventory file
ansible all -b -a "systemctl restart nginx"	Run command with sudo privileges
ansible all -a "df -h"	Check disk usage on all hosts
ansible all -a "free -m"	Check memory usage
ansible all -m copy -a "src=/tmp/file dest=/tmp/"	Copy file to remote hosts
ansible all -m fetch -a "src=/var/log/syslog dest=./logs/"	Fetch files from remote hosts
ansible all -b -m apt -a "name=nginx state=present"	Install package using APT
ansible all -b -m yum -a "name=httpd state=present"	Install package using YUM
ansible all -b -m service -a "name=nginx state=started"	Start a service on remote hosts
ansible all -b -m user -a "name=testuser state=present"	Create a new user
ansible-inventory -i inventory.ini --list	List all hosts and groups from inventory
ansible-inventory --host server1	Show variables for a specific host
ansible-inventory --graph	Display inventory graph
ansible-playbook site.yml	Run a playbook
ansible-playbook -i inventory.ini site.yml	Run playbook with specific inventory
ansible-playbook site.yml --tags install	Run only tasks with specific tag
ansible-playbook site.yml --skip-tags debug	Skip tasks with a tag
ansible-playbook site.yml --start-at-task "Install Nginx"	Start playbook from a specific task
ansible-playbook site.yml --limit server1	Run playbook for one host
ansible-playbook site.yml --syntax-check	Check playbook syntax
ansible-playbook site.yml --check	Dry run (no changes applied)
ansible-playbook site.yml -vvv	Run playbook with verbose logging
ansible localhost -m debug -a "var=hostvars"	View all gathered variables
ansible-vault encrypt secrets.yml	Encrypt a file

ansible-vault decrypt secrets.yml	Decrypt a file
ansible-vault edit secrets.yml	Edit encrypted file
ansible-playbook site.yml --ask-vault-pass	Run playbook using vault password
ansible all -m setup	Gather system facts
ansible all -m setup -a "filter=ansible_distribution"	Get specific fact
ansible-galaxy install geerlingguy.nginx	Install role from Ansible Galaxy
ansible-galaxy init myrole	Create role structure
ansible all -u ec2-user -a "whoami"	Execute command as specific user
ansible all -m shell -a "bash script.sh"	Run shell script on remote hosts
ansible webservers -a "uptime"	Run command only on host group
ansible all -m ping -u ubuntu --private-key ~/.ssh/id_rsa	Ping hosts using SSH key
ansible-playbook site.yml --extra-vars "version=1.0"	Pass variable via CLI
ansible-playbook site.yml --extra-vars "@vars.json"	Pass variables from JSON/YAML file
ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook site.yml	Disable SSH host key checking
ansible-playbook site.yml -f 20	Run tasks in parallel using forks
ansible-config view	View active Ansible configuration
ansible-config list	List all configuration options
ansible-config dump --only-changed	Show only modified config values
ansible all -b -a "reboot"	Reboot all servers
ansible all -a "ss -tulpn"	Check open ports
ansible all -a "cat /etc/os-release"	Check OS version
ansible all -a "find / -name nginx.conf"	Search for a file on remote hosts

--Ansible—

-----Ansible Role Vs Ansible Playbook.

📜 Ansible Playbook

- **What it is**: A YAML file that defines **a set of tasks** to run on target machines.
- **Structure**: Contains **hosts**, **tasks**, **variables**, **handlers**, etc.
- **Purpose**: Orchestrates what needs to be done (like "install Nginx, start service, open firewall").
- **Granularity**: High-level instructions for automation.

☞ Example Playbook:

```
- name: Install and start Nginx
  hosts: webservers
  become: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present

    - name: Start Nginx
      service:
        name: nginx
        state: started
```

📦 Ansible Role

- **What it is**: A **structured, reusable collection** of tasks, variables, templates, files, and handlers.
- **Purpose**: Organizes playbooks into reusable components.
- **Directory structure**:

```
roles/
  nginx/
    tasks/
      main.yml
    handlers/
      main.yml
    templates/
    files/
    vars/
    defaults/
    meta/
```

- **Use case**:
  - Encourages **reusability and modularity**.
  - Can be shared (via **Ansible Galaxy**).
  - Makes playbooks cleaner.

☞ Example Playbook using a Role:

```
- name: Deploy webserver
  hosts: webservers
  roles:
    - nginx
```

Here, all the logic (install, configure, start nginx) lives in the **role** nginx.

✓ **Key Difference**

Feature	**Playbook**	**Role**
Definition	A YAML file that describes automation steps	A structured, reusable collection of tasks and related files
Granularity	High-level orchestration	Modular building block
Reusability	Limited	Highly reusable across playbooks/projects
Structure	Flat (just tasks, handlers, etc.)	Standard directory layout (tasks, vars, templates, handlers, etc.)
Use Case	Quick automation scripts	Large, reusable, complex automation projects

☞ Think of it like this:

- **Playbook =** Recipe (how to cook a dish)
- **Role =** Ingredients + kitchen tools (organized, reusable pieces you use in many recipes)

Example: Install & Configure Nginx

1. As a Playbook (simple, direct approach)

```
# file: nginx-playbook.yml
- name: Install and configure Nginx
  hosts: webservers
  become: yes

  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
      when: ansible_os_family == "Debian"

    - name: Copy Nginx config
      template:
        src: nginx.conf.j2
        dest: /etc/nginx/nginx.conf
      notify: Restart nginx

    - name: Ensure Nginx is running
      service:
        name: nginx
        state: started
        enabled: yes

  handlers:
    - name: Restart nginx
      service:
        name: nginx
        state: restarted
```

2. Refactored into a Role (reusable)
Role structure:

```
roles/
  nginx/
    tasks/
      main.yml
    handlers/
      main.yml
    templates/
      nginx.conf.j2
    defaults/
      main.yml
```

---

roles/nginx/tasks/main.yml

```
- name: Install Nginx
  apt:
    name: nginx
    state: present
  when: ansible_os_family == "Debian"

- name: Copy Nginx config
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
  notify: Restart nginx

- name: Ensure Nginx is running
  service:
    name: nginx
    state: started
    enabled: yes
```

---

roles/nginx/handlers/main.yml

```
- name: Restart nginx
  service:
    name: nginx
    state: restarted
```

---

roles/nginx/defaults/main.yml

```
nginx_port: 80
```

Playbook using the role:

```
# file: site.yml
- name: Deploy webserver with Nginx
  hosts: webservers
  become: yes
  roles:
    - nginx
```

✅ Now the **role** is reusable. If you want to use Nginx in another project, you just include - nginx instead of rewriting tasks.

⚖️ Comparison

- **Playbook only** → Good for small, one-off automations.
- **Role** → Clean, modular, reusable, great for large environments.

----Ansible Galaxy

- **Ansible Galaxy** is a **public repository of community-contributed roles and collections**.
- It works like a package manager (pip for Python, npm for Node.js).
- You can **download, share, and reuse roles** instead of reinventing the wheel.

✅ Advantages of Galaxy Roles

- Save time (don't reinvent basic services like Nginx, MySQL, Docker).
- Roles are **well-tested and widely used**.
- Can be **customized** with variables.
- You can publish your own roles too.

📊 Ansible Galaxy Roles vs Custom Roles

Feature	**Galaxy Role (Community)**	**Custom Role (Self-written)**
Source	Downloaded from Ansible Galaxy or GitHub	Written and maintained by your team
Speed	Fast → ready-made, saves time	Slower → you write everything yourself
Use Case	Common services (Nginx, MySQL, Docker, Kubernetes, etc.)	Organization-specific automation (custom apps, internal tools)
Customization	Configurable with role variables	Fully customizable (you control all tasks, handlers, templates)
Quality	Varies (popular roles like geerlingguy.* are high quality, but some may lack updates)	Guaranteed to fit your environment and standards
Maintenance	Maintained by community (may or may not keep up-to-date)	Maintained by your team (you control updates)
Reusability	Very high → widely reusable across projects	Also reusable, but only within your org unless you publish it
Learning Curve	Easier → you only need to know how to configure it with variables	Requires more Ansible knowledge (role structure, tasks, handlers, etc.)

✅ **Rule of Thumb**:

- Use **Galaxy roles** when: installing **common, standard software** (Nginx, Apache, Docker, Kubernetes, databases).
- Use **custom roles** when: automating **internal apps, workflows, or unique infrastructure setups** that Galaxy doesn't cover.

✅ How to add new worker nodes into Ansible inventory (hosts file)
✅ Step-by-step process to add worker nodes and run configuration

---

📌 1. How We Update Configuration of Nodes Using Ansible

In Ansible, configuration of nodes is controlled through:

1. **Inventory file** (hosts) – contains list of nodes
2. **Playbooks** – define tasks to apply configurations
3. **SSH connection** – must be passwordless (SSH key-based login)
4. **Ansible roles / modules** – configure packages, services, users, etc.

Example Flow :

1. Update Ansible inventory (/etc/ansible/hosts)
2. Write playbook (update-config.yml)
3. Run playbook (ansible-playbook update-config.yml)

📌 2. How to Add Worker Nodes to Ansible Host

✅ Step 1: Add Worker Node Entry in Inventory File **/etc/ansible/hosts**

```
[master]
master-node ansible_host=192.168.1.10 ansible_user=ec2-user

[workers]
worker-node1 ansible_host=192.168.1.11 ansible_user=ec2-user
worker-node2 ansible_host=192.168.1.12 ansible_user=ec2-user
```

You can create groups like [workers], [db], [web] etc.

✅ Step 2: Enable Passwordless SSH from Ansible Control Node

Run below command from Ansible host:

```
# ssh-copy-id ec2-user@192.168.1.11
# ssh-copy-id ec2-user@192.168.1.12
```

✓ This allows Ansible to connect without password
✓ Works only if public key already exists (~/.ssh/id_rsa.pub)

OR # Generate SSH key pair (if not already done)

```
ssh-keygen -t rsa -b 4096 -C "ansible@controller"

# Copy public key to all worker nodes
ssh-copy-id -i ~/.ssh/id_rsa.pub ec2-user@192.168.1.11
ssh-copy-id -i ~/.ssh/id_rsa.pub ec2-user@192.168.1.12
ssh-copy-id -i ~/.ssh/id_rsa.pub ec2-user@192.168.1.13
```

✅ Step 3: Test SSH Connectivity

```
ansible workers -m ping
Expected output:
worker-node1 | SUCCESS => ...
worker-node2 | SUCCESS => ...
```

📌 3. Example Playbook to Configure Worker Nodes update-workers.yml

```
---
- name: Update and configure worker nodes
  hosts: workers
  become: yes

  tasks:
    - name: Update package repository
      apt:
        update_cache: yes

    - name: Install Docker
      apt:
        name: docker.io
        state: present

    - name: Enable and start Docker service
      systemd:
        name: docker
        enabled: yes
        state: started
```

✅ Step 4: Run Playbook

```
ansible-playbook update-workers.yml
```

📌 4. Adding a New Worker Node (Step-by-Step)

Step	Action
1	Launch new worker VM / EC2 instance
2	Add hostname + IP in /etc/ansible/hosts under [workers] group
3	Copy SSH key: ssh-copy-id user@node-ip
4	Test connection: ansible workers -m ping
5	Apply playbook: ansible-playbook update-workers.yml

📌 5. Example: Adding New Worker Node Entry

Append to hosts file:

```
[workers]
worker-node3 ansible_host=192.168.1.13 ansible_user=ec2-user
```

📌 6. If Using Dynamic Inventory (AWS/EKS)

Instead of static /etc/ansible/hosts, use:

```
ansible-inventory -i aws_ec2.yml --list
```

Or use **Terraform → Ansible dynamic inventory**.

✅ Summary

Task	Command / File
Add worker node	Edit /etc/ansible/hosts
Enable SSH	ssh-copy-id user@IP
Test connection	ansible workers -m ping
Update config	ansible-playbook update-workers.yml

-----JAR File, WAR File

◆ **JAR File (Java ARchive)**

- **What it is**: A package file format used to bundle **Java classes, libraries, and metadata** into a single file (.jar).
- **Purpose**: To **distribute and run Java applications or libraries** easily.

✅ **Created By**:

- **Build tools**:
  - **Maven** (mvn package)
  - **Gradle** (gradle build)
  - **Ant**

✅ **Used By**:

- **Java Runtime Environment (JRE)** → java -jar myapp.jar
- **Application servers / microservices**
- **Spring Boot apps** → packaged as executable JARs

☞ Example: A **microservice** in Spring Boot is usually built as a .jar file.

◆ **WAR File (Web Application ARchive)**

- **What it is**: A package file format used to bundle **Java web applications** (Servlets, JSPs, HTML, JavaScript, etc.) into a single file (.war).
- **Purpose**: To **deploy web applications on application servers**.

✅ **Created By**:

- **Build tools**:
  - **Maven** (mvn package with packaging = war)
  - **Gradle** (gradle war)

✅ **Used By**:

- **Java application servers / servlet containers**:
  - **Apache Tomcat**
  - **JBoss / WildFly**
  - **WebLogic**
  - **GlassFish**

☞ Example: A **traditional Java web app** (JSP/Servlet) is packaged as .war and deployed on **Tomcat**.

◆ **WAR vs JAR in DevOps Context :-**

Aspect	**JAR**	**WAR**
-Stands for	-Java ARchive	-Web Application ARchive
-Content	-Java classes, resources, libraries	-Web components (Servlets, JSP, HTML, etc.)
-Execution	-java -jar myapp.jar (standalone app)	-Needs servlet container (e.g., Tomcat)
-Deployment	-Run directly as microservice	-Deploy to app server
-Modern Use	-Spring Boot / Microservices	-Legacy Java EE web apps

◆ **Example Flow with Tools (DevOps) :-**

1. **Developer writes code** in Java/Spring.
2. **Build tool (Maven/Gradle)** compiles → creates .jar or .war.
3. **CI/CD (Jenkins, GitHub Actions, GitLab CI)** builds artifact (jar/war).
4. **Artifact Repository** (Nexus, Artifactory, S3) stores the .jar/.war.
5. **Deployment**:
    - .jar → run directly in container (Docker + Kubernetes).
    - .war → deploy to Tomcat/WildFly in VM or container.

🎯 **Interview-Ready Answer:**

"JAR and WAR are Java packaging formats. JAR is used for standalone Java apps or microservices, created by tools like Maven or Gradle and executed directly with Java runtime. WAR is used for web apps, created by the same tools but deployed into servlet containers like Tomcat or JBoss. In DevOps pipelines, Jenkins builds these artifacts, stores them in Nexus/Artifactory, and deploys them either in containers (for JAR) or in app servers (for WAR)."

--Jenkins-- --Syllabus (with One-Line Explanations)

1. Introduction to Jenkins

- **What is Jenkins?** → Open-source automation server for CI/CD.
- **Jenkins Features** → Plugins, extensibility, distributed builds.
- **Jenkins Architecture** → Master-agent architecture for scalability.

2. Installation & Setup

- **Installing Jenkins** → Setup via WAR, Docker, or package managers.
- **Jenkins Dashboard** → Web UI for managing jobs and pipelines.
- **User Management** → Add users, roles, and permissions.
- **Jenkins Home Directory** → Stores jobs, plugins, and configurations.

3. Jobs & Builds

- **Freestyle Jobs** → Simple build configurations.
- **Build Triggers** → Start jobs via SCM, schedule, or webhook.
- **Build Steps** → Define actions (compile, test, deploy).
- **Post-Build Actions** → Notifications, archiving artifacts, publishing reports.

4. Plugins

- **Plugin Manager** → Install and update plugins.
- **Common Plugins** → Git, Docker, Pipeline, Slack, SonarQube.
- **Custom Plugins** → Extend Jenkins with custom logic.

5. Pipelines

- **Pipeline as Code** → Define CI/CD in Jenkinsfile.
- **Declarative Pipeline** → Structured pipeline syntax.
- **Scripted Pipeline** → Groovy-based flexible syntax.
- **Stages & Steps** → Define workflow stages and tasks.
- **Post Section** → Run tasks after pipeline (success/failure).

6. SCM (Source Code Management)

- **Git Integration** → Clone and build code from Git repos.
- **SVN/Mercurial** → Alternative SCM support.
- **Webhook Integration** → Trigger builds automatically on commits.

7. Build Tools & Testing

- **Maven/Gradle Integration** → Build Java projects.
- **npm & Python Integration** → Build and test applications.
- **JUnit & TestNG Reports** → Publish test results.
- **Code Coverage Tools** → Integrate Jacoco, Cobertura, etc.

8. Distributed Builds

- **Master-Agent Setup** → Offload builds to worker nodes.
- **Agent Types** → SSH, JNLP, Docker agents.
- **Node Labels** → Assign jobs to specific agents.

9. CI/CD with Jenkins

- **Continuous Integration (CI)** → Automate builds and tests.
- **Continuous Delivery (CD)** → Automate packaging and staging.
- **Continuous Deployment** → Auto-deploy to production.
- **Blue Ocean** → Modern pipeline visualization.

---

10. Integrations

- **Docker Integration** → Build and deploy Docker images.
- **Kubernetes Integration** → Deploy workloads to K8s clusters.
- **Slack/Email Notifications** → Real-time build alerts.
- **SonarQube Integration** → Code quality analysis.
- **GitHub & Bitbucket** → SCM + pull request builds.

---

11. Security & Administration

- **Authentication** → LDAP, GitHub OAuth, SAML.
- **Authorization** → Role-based access control.
- **Securing Jenkins** → Hardening, secrets management.
- **Backup & Restore** → Protect Jenkins data.

---

12. Monitoring & Logging

- **System Logs** → Monitor Jenkins activity.
- **Build History** → Track job execution.
- **Metrics & Health Checks** → Performance monitoring with plugins.

---

13. Advanced Jenkins

- **Pipeline Libraries** → Share reusable pipeline code.
- **Multibranch Pipelines** → Build multiple branches automatically.
- **Matrix Builds** → Test across multiple environments.
- **Parallel Stages** → Run tasks in parallel for speed.

---

14. Best Practices

- **Pipeline as Code (IaC)** → Store pipelines in Git.
- **Small, Modular Jobs** → Improve reusability and debugging.
- **Use of Agents & Containers** → Isolate builds.
- **Automated Testing & Quality Gates** → Ensure reliability.

--Jenkins—

------CI, CD & CDx

◆ **Continuous Integration (CI)**

- Developers frequently merge their code changes into a **shared repository** (like GitHub, GitLab, Bitbucket).
- Each merge triggers an **automated build and test process**.
- Goal → **detect bugs early** and ensure the new code works with existing code.

☞ Example: You push code → Jenkins/GitHub Actions runs unit tests → you know immediately if something breaks.

◆ **Continuous Delivery (CD)**

- Extends CI by **automatically preparing code for release**.
- After build and tests, the code is **packaged and deployed to a staging or test environment**.
- Deployment to **production is still a manual step** (for approval/release).
- Goal → software is **always in a deployable state**.

☞ Example: After CI, app is deployed to **staging** → QA tests it → release manager approves → then deploy to production.

◆ **Continuous Deployment (CDx)**

- Goes one step further than Continuous Delivery.
- Every change that passes automated tests is **automatically deployed to production**.
- No manual approvals.
- Goal → deliver new features/bug fixes to users **as fast as possible**.

☞ Example: You push code → tests pass → app is automatically deployed to **production**.

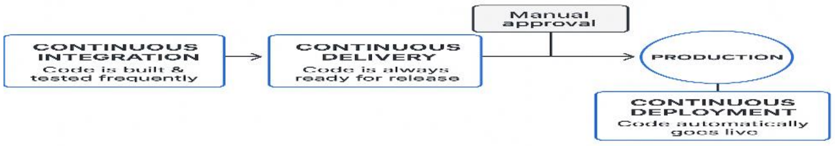
◆ **Key Differences**

Practice	**What It Ensures**	**Production Deployment**
Continuous Integration	Code is built & tested frequently	✗ Not part of CI
Continuous Delivery	Code is always ready for release	✓ Manual approval
Continuous Deployment	Code automatically goes live	✓ Automatic

---

✓ **In short**:

- **CI** → Make sure code integrates correctly.
- **CD (Delivery)** → Make sure code is always ready to release.
- **CD (Deployment)** → Make sure code is automatically released to users.


CONTINUOUS INTEGRATION
Code is built & tested frequently
CONTINUOUS DELIVERY
Code is always ready for release
Manual approval
PRODUCTION
CONTINUOUS DEPLOYMENT
Code automatically goes live

------CI,CD,CDx with the Devops tools

◆ **Continuous Integration (CI) → *Build & Test Phase***
Goal: Integrate code frequently, run builds & tests automatically.
☞ **Tools**:

- **Version Control**: GitHub, GitLab, Bitbucket
- **CI Servers**: Jenkins, GitHub Actions, GitLab CI, CircleCI, TravisCI
- **Build Tools**: Maven, Gradle, npm, pip
- **Testing Tools**: JUnit, Selenium, PyTest

◆ **Continuous Delivery (CD) → *Deploy to Staging / Ready for Production***
Goal: Ensure the application is always in a **deployable state**. Deployment to production needs **manual approval**.
☞ **Tools**:

- **CI/CD Orchestrators**: Jenkins, GitLab CI, ArgoCD
- **Configuration Management**: Ansible, Chef, Puppet
- **Containerization**: Docker
- **Artifact Repositories**: Nexus, JFrog Artifactory, AWS CodeArtifact
- **Infrastructure as Code**: Terraform, AWS CloudFormation
- **Cloud Services**: AWS Elastic Beanstalk, Azure DevOps, Google Cloud Build

---

◆ **Continuous Deployment (CDx) → *Automatic Production Deployment***
Goal: Every code change that passes automated tests is deployed **straight to production**.
☞ **Tools**:

- **Kubernetes** (with Helm, Kustomize for deployments)
- **ArgoCD / FluxCD** (GitOps-based automated deployments)
- **Spinnaker** (progressive delivery, automated pipelines)
- **AWS CodePipeline + CodeDeploy**
- **Canary / Blue-Green Tools**: Istio, Linkerd, Flagger

◆ **Full CI/CD/CDx Pipeline Example (Tool Chain)**

1. **Developer commits code** → GitHub/GitLab/Bitbucket.
2. **CI kicks in** → Jenkins/GitHub Actions builds & tests the app.
3. **CD (Delivery)** → App packaged into Docker image, pushed to registry (ECR/DockerHub), deployed to **staging** using Terraform + Ansible.
4. **CD (Deployment)** → ArgoCD automatically syncs Git state with **production Kubernetes cluster**.

✅ **In short**:

- **CI Tools** → Jenkins, GitHub Actions, GitLab CI, CircleCI.
- **CD (Delivery) Tools** → Jenkins, Ansible, Terraform, Docker, CloudFormation.
- **CD (Deployment) Tools** → ArgoCD, Spinnaker, FluxCD, Kubernetes, AWS CodeDeploy.

-----**Jenkins project types**, their **features**, and **real use**

☐ Types of Jenkins Projects (Jobs) & Their Uses

When you click **"New Item"** in Jenkins, you see multiple project/job types.

-Each one serves a different purpose.

---

1 Freestyle Project

✔ **What it is:** -A simple, flexible job that supports:

- Shell commands
- Build tools (Maven, Gradle, Ant)
- Source code checkout
- Email notifications
- Post-build actions

- It's easy to configure using GUI but **does not use Jenkinsfile**.

✔ **Features:**

- Simple GUI-based configuration
- Supports shell scripts
- Can use parameters
- Supports build triggers (cron, Git webhooks)
- Can archive artifacts
- Can use Jenkins credentials

✔ **Use Cases:**

Use Case	**Example**
Running simple scripts	Shell/Python/Ansible jobs
Building software	Maven/Gradle build
Automating small tasks	Cleanup scripts, backup scripts
One-time quick jobs	Test automation, cron jobs
Legacy pipeline tasks	Old projects not using Jenkinsfile

- Best for beginners or simple automations.

---

2 Pipeline Project (Most Used Today)

✔ **What it is:** -A pipeline job that runs a **Jenkinsfile**, either:

- directly written in Jenkins
- or stored in Git (Pipeline script from SCM)
  - This is the modern standard.

✔ **Features:**

- Declarative or Scripted pipeline
- Multiple stages (build, test, deploy, notify)
- Supports parallel execution
- Error handling
- Reusable functions
- Full CI/CD automation

- Integration with Docker, Kubernetes, AWS, Azure, GCP
- withCredentials()
- Environment variables
- Checkout from Git

✔ **Use Cases:**

Use Case	**Example**
Full CI/CD pipeline	Build → Test → Docker Build → Deploy
Terraform automation	terraform apply/destroy
Kubernetes deployment	Deploy to EKS, AKS, GKE
AWS automation	EC2, S3, EKS operations
DevOps pipelines	Microservices CI/CD

Best for modern DevOps workflows.

---

3 Multibranch Pipeline

✔ **What it is:** **-** Automatically creates **one pipeline per Git branch**.

-If your repo has: - Main - dev - feature/login

-Jenkins creates **three separate jobs**, each using the Jenkinsfile inside that branch.

✔ **Features:**

- Auto-detects new branches
- Auto-deletes removed branches
- Each branch runs its own Jenkinsfile
- Best for GitHub/GitLab/Bitbucket
- Perfect for Pull Requests
- Supports shared libraries

✔ **Use Cases:**

Use Case	**Example**
Large DevOps projects	Microservices with many branches
PR validation	Auto-build PR branches
GitOps organizations	Branch-based deployment

Best for teams using Git heavily.

---

4 Folder

✔ **What it is:** **-**A way to organize multiple Jenkins jobs into folders.

✔ **Features:**

- Group related jobs
- Apply permissions to folder
- Apply configurations to entire folder
- Useful for large Jenkins installations

✔ **Use Cases:**

- Organizing jobs for each team (Dev, QA, Prod)

- Organizing jobs per project
- Organizing microservices (service1, service2, etc.)

5️⃣ **External Job** (Rarely Used)

✔ **What it is:** -Tracks execution of a job that runs **outside Jenkins**.

✔ **Features:**
- Jenkins monitors external script
- Useful when Jenkins cannot execute job directly

✔ **Use Cases:**
- Monitoring cron jobs that run on servers
- Tracking batch jobs running outside Jenkins

6️⃣ **Maven Project** (Legacy)

✔ **What it is:** **-**A special project type designed for **Maven builds**.

✔ **Features:**
- Jenkins analyzes POM.xml
- Auto-detect modules
- Auto-build dependencies

✔ **Use Cases:**
- Large Java projects using Maven
- Legacy Jenkins jobs

Today replaced by **Pipeline project + Maven plugin**.

📌 **Quick Comparison Table**

Project Type	**Supports Jenkinsfile?**	**Difficulty**	**Use Case**
Freestyle	✗ No	Easy	Simple scripts / builds
Pipeline	✔ Yes	Medium	Full CI/CD automation
Multibranch Pipeline	✔ Yes	Medium–Advanced	Multi-branch Git workflows
Maven Project	✗ No	Easy	Legacy Java builds
External Job	✗ No	Hard	Monitoring external scripts
Folder	✗ No	Easy	Organizing projects

🎯 **Which one should YOU use?**

-Since you are working with:

✔ Terraform ✔ AWS CLI
✔ Docker ✔ EKS
✔ GitHub

-You should use: 👉 **Pipeline Project (Pipeline script from SCM)**

-This is the best and modern approach.

-----What is a Multi-Branch Pipeline in Jenkins?

A **Multi-Branch Pipeline** in Jenkins is a special type of pipeline job that can **automatically create and manage separate pipelines for each branch** in a source code repository (like GitHub, GitLab, or Bitbucket).

Each branch in your repository has its own pipeline based on a Jenkinsfile stored in that branch.

🧠 Why We Use It

In modern CI/CD workflows:

- Developers work on **multiple branches** (e.g., dev, test, feature/login, main)
- Each branch might need its own **build, test, and deploy pipeline**
- Instead of manually creating separate Jenkins jobs for each branch, a **Multi-Branch Pipeline** handles all branches automatically

---

⚙️ How It Works

1. Jenkins scans your Git repository (via webhooks or polling).
2. For each branch containing a Jenkinsfile, Jenkins automatically:
   - Creates a **pipeline job** for that branch
   - Executes the stages defined in that branch’s Jenkinsfile
3. When a branch is deleted, Jenkins automatically removes the corresponding job.

☐ Example Folder Structure

```
my-app-repo/
├── Jenkinsfile         # For main branch
├── src/
│   └── app.py
├── feature/login/
│   └── Jenkinsfile     # Branch-specific Jenkinsfile
└── dev/
    └── Jenkinsfile
```

Each branch can have its own **own Jenkinsfile** defining its **own build logic**.

☐ Jenkinsfile Example

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo "Building branch: ${env.BRANCH_NAME}"
                sh 'mvn clean package'
            }
        }

        stage('Test') {
            steps {
                echo "Running tests for ${env.BRANCH_NAME}"
                sh 'mvn test'
            }
        }

        stage('Deploy') {
            when {
                branch 'main'
            }
            steps {
                echo "Deploying from main branch"
            }
        }
    }
}
```

🔧 Steps to Create a Multi-Branch Pipeline in Jenkins

1. Open **Jenkins Dashboard**
2. Click **"New Item"**
3. Enter a name (e.g., my-multibranch-pipeline)
4. Select **"Multibranch Pipeline"**
5. Click **OK**
6. Under **"Branch Sources"**:
   - Add your **GitHub or Git repository URL**
   - Add credentials if needed
7. Configure **Scan Interval** (e.g., every 5 minutes)
8. Save and let Jenkins scan the repo

✓ Jenkins will:

- Detect all branches with Jenkinsfile
- Create individual pipeline jobs automatically

📊 Benefits of Multi-Branch Pipeline

Advantage	Explanation
Automation	Jenkins auto-discovers and manages branches
Branch Isolation	Each branch has its own build/test pipeline
No manual job creation	Simplifies CI/CD setup for GitFlow
Better testing	Developers can test features independently
Supports Pull Requests	Automatically runs pipeline for PRs

☐ How It Works in Multi-Branch Setup

- **Jenkins scans** your Git repository.
- Each branch (dev, feature/*, main) runs this **same Jenkinsfile**.
- The behavior changes based on branch name:

Branch	Pipeline Actions
feature/*	Build + Unit Test
dev	Build + Test + SonarQube + Push Image
main	Full CI/CD: Build + Test + Push + Deploy

🧠 Real Example in DevOps

When using **GitHub + Jenkins + AWS**:

- Developer pushes code to a new branch (feature/login)
- Jenkins detects the branch automatically
- Runs Jenkinsfile from that branch:
  - Builds Docker image
  - Runs unit tests
  - Pushes image to ECR (for main only)

✅ Summary

Topic	Explanation
What	-A Jenkins job type that automatically handles multiple Git branches
How	-Uses branch scanning and branch-specific Jenkinsfiles
Why	-Simplifies CI/CD for teams working with multiple branches
Example	-Each branch (dev, test, main) runs its own pipeline automatically

---

-----How to add a new EC2 instance as a Jenkins slave node (agent).

There are two main ways:

1. **Static agent (permanent node)** – manually register EC2 in Jenkins.
2. **Dynamic agent (on-demand)** – via Jenkins plugins like EC2 plugin, Kubernetes plugin, etc.

I'll explain **static/manual way first**, since it's the most common for EC2.

🚀 Steps: Adding EC2 as a Jenkins Slave Node (A-Static)

1. Launch EC2 instance

- Spin up a new EC2 instance (Amazon Linux/Ubuntu is common).
- Install Java (Jenkins agent requires Java runtime).

```
# sudo apt update -y
# sudo apt install openjdk-11-jdk -y
    Check:
            #java -version
```

---

2. Prepare Jenkins Master

- Go to Jenkins UI → **Manage Jenkins** → **Manage Nodes & Clouds** → **New Node**.
- Enter a **node name** (e.g., ec2-slave-1).
- Select **Permanent Agent**.
- Configure:
  - **Remote root directory**: /home/jenkins (or wherever you want workdir).
  - **Labels**: e.g., build, test (used in pipeline to target this node).
  - **Usage**: "Use this node as much as possible" (default).
  - **Launch method**: *Launch agents via SSH*.

---

3. Configure SSH Access

- Ensure your Jenkins master can connect to EC2 over **SSH (port 22)**.
- Add EC2's SSH keypair to Jenkins credentials:
  - **Manage Jenkins** → **Credentials** → Add → Kind = SSH Username with private key.
  - Username = ubuntu (or ec2-user, depending on AMI).
  - Private key = EC2 keypair PEM file.

---

4. Complete Node Configuration

- Back in **node configuration**:
  - Host = EC2 public/private IP.
  - Credentials = Select the SSH key you just added.
  - Save.

Jenkins will try to connect → install the agent JAR → and register the node.

5. Test Connection

- Node should show as **online**.
- Run a freestyle/pipeline job with label = node's label (e.g., build).

Example Pipeline:

```
pipeline {
    agent { label 'build' }        // runs only on EC2 slave node
    stages {
        stage('Check') {
            steps {
                sh 'hostname'
                sh 'echo Running on EC2 Jenkins Agent'
            }
        } } }
```

B- **Alternate: Dynamic Scaling**

If you want EC2 agents to **auto-scale**:

- Install **EC2 plugin** (Jenkins → Manage Plugins → Amazon EC2 plugin).
- Configure AWS credentials + EC2 AMI + instance size.
- Jenkins will launch new EC2 agents on-demand and terminate them when idle.

✅ **Summary**:

- Install Java on EC2.
- Add node in Jenkins UI (Permanent Agent).
- Use SSH + credentials to connect Jenkins master → EC2.
- Use labels to target jobs to that EC2 node.

----Jenkins pipeline (Jenkinsfile) that decides whether to deploy on **EC2 slave** or **Kubernetes pod** dynamically, based on the environment variable.

◆ **Scenario**

- **EC2 slave node** is used for **VM-based deployments**.
- **Kubernetes agent (podTemplate)** is used for **K8s-based deployments**.
- The choice is made based on an environment variable (DEPLOY_TARGET).

🔎 **How It Works**

1. **agent none** → prevents global agent, so each stage picks its own.
2. **DEPLOY_TARGET parameter** → decides where to run (EC2 or K8s).
   - Example: Pass DEPLOY_TARGET=ec2 or DEPLOY_TARGET=k8s in Jenkins job params.
3. **EC2 Stage** → runs on a Jenkins slave node (label: ec2-slave).
4. **K8s Stage** → spins up a Kubernetes pod with kubectl inside.

✅ **Example Usage**

- If DEPLOY_TARGET=ec2:
  - Pipeline runs on EC2 slave node → calls deploy-to-vm.sh.
- If DEPLOY_TARGET=k8s:
  - Pipeline spins up a Kubernetes pod → calls deploy-to-k8s.sh.

-----How do you store your credentials in jenkins? Like api tokens, github login passwords.

In Jenkins, you **never hardcode credentials** directly in a Jenkinsfile or pipeline. Instead, you store them securely in **Jenkins Credentials Store** and reference them when needed.
Here's how it works step by step:

◆ 1. Jenkins Credentials Store

Jenkins has a built-in **Credentials plugin** that stores:

- **Username & password**
- **API tokens**
- **SSH private keys**
- **Secret text** (like GitHub tokens, AWS keys)
- **Secret files**

They are encrypted on disk and masked in logs when used.

◆ 2. Adding Credentials

1. Go to **Jenkins Dashboard** → **Manage Jenkins** → **Credentials**.
2. Choose scope:
   - **Global** (available everywhere).
   - **Folder/Job specific** (restricted).
3. Add the credential type:
   - **Username with password** (for GitHub login).
   - **Secret text** (for API tokens, PATs).
   - **SSH key** (for Git access).

Each credential gets an **ID** that you'll use in pipelines.

◆ 3. Using Credentials in Pipelines

(a) Secret Text (e.g., GitHub API Token)

```
pipeline {
    agent any
    stages {
        stage('GitHub API') {
            steps {
                withCredentials([string(credentialsId: 'github-token', variable:
'GHTOKEN')]) {
                    sh 'curl -H "Authorization: token $GHTOKEN"
https://api.github.com/user'
                }  }  }  }  }
```

(b) Username & Password

```
withCredentials([usernamePassword(credentialsId: 'git-cred',
usernameVariable: 'USER', passwordVariable: 'PASS')]) {
    sh 'git clone https://$USER:$PASS@github.com/myrepo.git'  }
                withCredentials([usernamePassword(
                    credentialsId: 'aws-creds',
                    usernameVariable: 'AWS_ACCESS_KEY_ID',
                    passwordVariable: 'AWS_SECRET_ACCESS_KEY'  )]) {   }
```

(c) Secret Files
If you store a JSON key (like for Google Cloud):

```
withCredentials([file(credentialsId: 'gcp-key', variable: 'KEYFILE')]) {
    sh 'gcloud auth activate-service-account --key-file=$KEYFILE'
}
```

(d) SSH Private Key

```
sshagent(['my-ssh-key-id']) {
    sh 'git clone git@github.com:myorg/myrepo.git'
}
```

◆ 4. Best Practices

- Use **credentialsId**, never hardcode secrets.
- Keep credentials **scoped to folder/job** if possible (least privilege).
- Rotate API tokens regularly.
- Use **Jenkins Secret Masking** to avoid leaking tokens in logs.
- For cloud access (AWS, GCP, Azure), prefer using **IAM roles / workload identity** over static credentials.

✅ **In summary:**
- Credentials are stored in **Jenkins Credentials Store** (encrypted).
- They are injected into pipelines at runtime using **withCredentials** or **sshagent**.

------ **Jenkins pipeline** that provisions infrastructure (Terraform/CloudFormation) for **multiple AWS accounts** whenever a **new customer signs up**.

🏗 High-Level Flow

1. **Trigger** → New customer event (manual Jenkins input, API call, or SCM webhook).
2. **Parameters** → Customer name, AWS account ID, region, environment (dev/prod).
3. **Assume Role** → Jenkins assumes an IAM role in the customer's AWS account (via STS).
4. **Infrastructure as Code (IaC)** → Terraform/CloudFormation provisions infra.
5. **Deploy** → Create VPC, subnets, security groups, EKS/EC2/RDS/etc.
6. **Output** → Store state (Terraform backend in S3 + DynamoDB), notify via Slack/Email.

⚙️ **Jenkins Pipeline (Declarative)**

Here’s an example with **Terraform + multiple AWS accounts**:

```
pipeline {
   agent any

   parameters {
      string(name: 'CUSTOMER_NAME', defaultValue: '', description: 'New
customer name')
      string(name: 'AWS_ACCOUNT_ID', defaultValue: '', description: 'Target
AWS Account ID')
      choice(name: 'AWS_REGION', choices: ['us-east-1', 'us-west-2', 'ap-
south-1'], description: 'AWS Region')
      choice(name: 'ENV', choices: ['dev', 'staging', 'prod'], description:
'Environment')
   }

   environment {
      TF_VERSION = '1.8.5'
      AWS_ROLE_NAME = 'JenkinsProvisionRole'    // IAM Role in each account
      AWS_CREDS = credentials('aws-master-account-creds')        // Jenkins
credentials for master account
   }

   stages {
      stage('Prepare') {
         steps {
            echo "Provisioning infra for customer:
${params.CUSTOMER_NAME}"
            sh 'terraform --version'
         }
      }

      stage('Assume Role in Target AWS Account') {
         steps {
            script {
                                    // Assume role into target AWS account
               def role_arn =
"arn:aws:iam::${params.AWS_ACCOUNT_ID}:role/${env.AWS_ROLE_NAME}
"
               sh """
               CREDS=$(aws sts assume-role \
                  --role-arn ${role_arn} \
                  --role-session-name jenkins-${params.CUSTOMER_NAME} \
                  --region ${params.AWS_REGION} \
                  --output json \
                  --profile ${AWS_CREDS_USR})

               export AWS_ACCESS_KEY_ID=$(echo $CREDS | jq -r
.Credentials.AccessKeyId)
               export AWS_SECRET_ACCESS_KEY=$(echo $CREDS | jq -r
.Credentials.SecretAccessKey)
               export AWS_SESSION_TOKEN=$(echo $CREDS | jq -r
.Credentials.SessionToken)
               """
            }
         } }
```

```
        stage('Terraform Init & Plan') {
            steps {
                sh """
                cd infra/${params.ENV}
                terraform init -backend-config="bucket=my-tf-state-bucket" -
backend-config="key=${params.CUSTOMER_NAME}/terraform.tfstate" -
backend-config="region=${params.AWS_REGION}"
                terraform plan -var customer=${params.CUSTOMER_NAME} -var
region=${params.AWS_REGION} -out plan.tfplan
                """
            }
        }

        stage('Terraform Apply') {
            steps {
                input message: "Approve infrastructure creation for
${params.CUSTOMER_NAME} in ${params.ENV}?"
                sh """
                cd infra/${params.ENV}
                terraform apply -auto-approve plan.tfplan
                """
            }
        }
    }

    post {
        success {
            echo "Infrastructure provisioned successfully for customer:
${params.CUSTOMER_NAME}"
            // Optionally send Slack/Email notification
        }
        failure {
            echo "Failed to provision infra for ${params.CUSTOMER_NAME}"
        }
    } }
```

📌 Key Points

- **Multiple AWS Accounts**:
  - Each AWS account must have an **IAM Role (JenkinsProvisionRole)** that Jenkins can assume.
  - Jenkins uses **STS AssumeRole** to get temporary credentials.
- **Terraform Backend**:
  - Use **S3 + DynamoDB** for state locking (per customer).
- **Customer Separation**:
  - Each customer has its own **state file (customer_name/terraform.tfstate)**.
- **Approval**:
  - Uses input step before applying changes.

✅ Example Use Case

- Customer A signs up → Pipeline triggered with params: CUSTOMER_NAME=A, AWS_ACCOUNT_ID=111111111111, ENV=prod
- Jenkins assumes role in **account A** → runs Terraform → provisions infra.
- Customer B signs up → Same pipeline, but runs in **account B**.

-----Write sample terraform code which is in git and we apply through the jenkins so write code and pipeline and all steps

Nice — below is a complete, ready-to-use example showing:

- a small Terraform project (AWS) stored in Git
- a Jenkins Declarative pipeline (Jenkinsfile) that checks out the repo, runs terraform fmt/validate/plan, and applies on approval
- steps to prepare Jenkins (credentials, plugins, agents) and a small bootstrap step to create the remote state S3 + DynamoDB lock (via AWS CLI)

Everything is parameterized and uses Jenkins Credentials (no secrets in repo). Replace placeholder values (bucket names, AWS account IDs, region, credential IDs) with your own.

---

1) Repo structure (example)

```
repo-root/
├─ terraform/
│  ├─ main.tf
│  ├─ variables.tf
│  ├─ outputs.tf
│  ├─ provider.tf
│  ├─ backend.tf
│  └─ versions.tf
└─ Jenkinsfile
```

---

2) Terraform files

terraform/provider.tf
terraform/backend.tf
terraform/versions.tf
terraform/variables.tf
terraform/main.tf
terraform/outputs.tf

---

3) Jenkinsfile (Declarative pipeline) — put at repo root

```
pipeline {
  agent { label 'linux && terraform' } // set agent label that has terraform & awscli installed
  environment {
    AWS_CREDENTIALS_ID = 'aws-creds'           // Jenkins credential ID for AWS (access/secret) or use role
    TF_VAR_tfstate_bucket = 'my-terraform-state-bucket' // or set via Jenkins credentials or params
    TF_VAR_tfstate_lock_table = 'my-terraform-locks'
    TF_VAR_aws_region = 'us-east-1'
    TF_ENV = 'dev'
  }
  options {
    skipDefaultCheckout(false)
    timestamps()
  }
```

```
 stages {
   stage('Checkout') {
     steps {
       git branch: 'main', url: 'https://github.com/your-org/your-repo.git'
     }
   }

   stage('Terraform Fmt & Validate') {
     steps {
       dir('terraform') {
         sh 'terraform -version'
         sh 'terraform fmt -check || (terraform fmt && echo "fmt fixed")'
         sh 'terraform init -backend-
config="bucket=${TF_VAR_tfstate_bucket}" -backend-
config="key=envs/${TF_ENV}/terraform.tfstate" -backend-
config="region=${TF_VAR_aws_region}" -backend-
config="dynamodb_table=${TF_VAR_tfstate_lock_table}"'
         sh 'terraform validate'
       }
     }
   }

   stage('Terraform Plan') {
     steps {
       withCredentials([[$class: 'AmazonWebServicesCredentialsBinding',
accessKeyVariable: 'AWS_ACCESS_KEY_ID', secretKeyVariable:
'AWS_SECRET_ACCESS_KEY', credentialsId:
"${env.AWS_CREDENTIALS_ID}"]]) {
         dir('terraform') {
           sh '''
             terraform plan -var="aws_region=${TF_VAR_aws_region}" \
                            -var="environment=${TF_ENV}" \
                            -var="tfstate_bucket=${TF_VAR_tfstate_bucket}" \
                            -var="tfstate_lock_table=${TF_VAR_tfstate_lock_table}"
\
                            -out=tfplan
           '''
           sh 'terraform show -no-color tfplan > plan.txt || true'
           archiveArtifacts artifacts: 'terraform/plan.txt', allowEmptyArchive:
true
           // optionally publish plan as artifact
         }
       }
     }
   }

   stage('Approval') {
     when {
       expression { return env.TF_ENV == 'prod' } // require approval for prod
     }
     steps {
       input message: "Apply terraform to ${env.TF_ENV}?", ok: "Apply"
     }
   }

   stage('Terraform Apply') {
     steps {
       withCredentials([[$class: 'AmazonWebServicesCredentialsBinding',
accessKeyVariable: 'AWS_ACCESS_KEY_ID', secretKeyVariable:
```

```
'AWS_SECRET_ACCESS_KEY', credentialsId:
"${env.AWS_CREDENTIALS_ID}"]]) {
          dir('terraform') {
            // Using the saved plan file if exists (safer)
            sh '''
              if [ -f tfplan ]; then
                terraform apply -input=false tfplan
              else
                terraform apply -auto-approve -
var="aws_region=${TF_VAR_aws_region}" -var="environment=${TF_ENV}"
-var="tfstate_bucket=${TF_VAR_tfstate_bucket}" -
var="tfstate_lock_table=${TF_VAR_tfstate_lock_table}"
              fi
            '''
            sh 'terraform output -json > outputs.json || true'
            archiveArtifacts artifacts: 'terraform/outputs.json',
allowEmptyArchive: true
          }
        }
      }
    }
  }

  post {
    success {
      echo 'Terraform apply succeeded'
    }
    failure {
      echo 'Terraform failed'
    }
    always {
      cleanWs()
    }
  }
}
```

Notes on the Jenkinsfile

- It expects an agent with Terraform and AWS CLI installed, or you can use containerized agents.
- Uses withCredentials + AmazonWebServicesCredentialsBinding — you must add Jenkins AWS credentials as type “AWS Credentials” (access key/secret) and name it aws-creds (or change the ID).
- The pipeline initializes backend with backend-config flags to avoid putting secrets into files.
- For prod, it waits for manual approval (input step). Modify logic as needed.

---

4) Jenkins preparation (one-time)

1. **Install required plugins**
   - Pipeline
   - Credentials Binding
   - Amazon Web Services Credentials (or CloudBees AWS Credentials)
   - Git
   - Blue Ocean (optional)
   - GitHub Branch Source (if using multibranch)

2. **Create credentials**
   - Go to Jenkins > Credentials > System > Add credentials:
     - Type: **AWS Credentials** (Access Key/Secret) — ID: aws-creds
     - OR store as username/password or secret text and adapt pipeline.
3. **Create an agent that has terraform & awscli**
   - Either:
     - Use a Jenkins agent VM with Terraform & AWS CLI installed
     - OR use Docker-based agent; e.g. agent { docker { image 'hashicorp/terraform:latest' } } but ensure awscli is present (you can build a custom image)
   - Ensure the agent's machine has network access to AWS.
4. **Create Jenkins Job**
   - Create a Multibranch Pipeline (recommended) pointed to your Git repo, or a regular pipeline and point to Jenkinsfile in repo.

---

5) Bootstrap S3 backend + DynamoDB lock (one-time steps)

Terraform S3 backend bucket and DynamoDB table must exist before terraform init. You can create them using AWS CLI or an initial ephemeral Terraform run with a local backend.

Option A — AWS CLI (fast)

```
# replace values
AWS_REGION=us-east-1
BUCKET=my-terraform-state-bucket
TABLE=my-terraform-locks

aws s3api create-bucket --bucket $BUCKET --region $AWS_REGION --create-bucket-configuration LocationConstraint=$AWS_REGION
aws s3api put-bucket-versioning --bucket $BUCKET --versioning-configuration Status=Enabled
aws dynamodb create-table \
  --table-name $TABLE \
  --attribute-definitions AttributeName=LockID,AttributeType=S \
  --key-schema AttributeName=LockID,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --region $AWS_REGION
```

Option B — bootstrap Terraform run using local backend

- Create a temporary bootstrap/ Terraform folder that creates S3 bucket + DynamoDB with terraform apply using local state.
- After it's created, remove local state and switch the real project to the S3 backend.

---

6) Best practice & security notes

- **Do not store AWS keys in Git.** Use Jenkins credentials or IAM role attached to agent.
- **Use remote state locking** (DynamoDB) to avoid concurrent apply.
- **Use workspaces or multiple backend keys** (key = envs/${var.environment}/terraform.tfstate) to separate envs.

- **Store sensitive variables in Jenkins credentials** and inject into pipeline via withCredentials.
- **Review plans** — don’t auto-approve production; require manual approval.
- **Audit**: Keep CloudTrail and S3 bucket logging for state bucket.
- **Terraform versioning**: pin provider and terraform versions to avoid drift.

---

7) Example run (what Jenkins does, step-by-step)

1. Developer pushes changes to main branch (terraform code or variables).
2. Jenkins Multibranch detects commit and starts pipeline (or webhook triggers).
3. Pipeline Checkout stage pulls repo.
4. Pipeline runs terraform fmt and terraform init (S3 backend + locking).
5. Pipeline runs terraform plan (outputs plan artifact).
6. For non-prod: pipeline proceeds to terraform apply automatically; for prod it pauses for manual approval.
7. Pipeline applies the plan (using Jenkins AWS credentials).
8. Pipeline archives outputs.json and posts success/failure.
9. Clean workspace.

---

8) Quick tips for adding to this example

- Replace the sample EC2 with EKS / RDS modules when moving to production SaaS infra.
- Break infra into modules (modules/vpc, modules/compute) and call them from env-specific root modules.
- Add terraform fmt and tflint stages for linting.
- Use terraform-docs to auto-generate docs in CI.

If you want, I can:

- convert the Jenkinsfile to a *declarative agent with Docker* (so Terraform + AWS CLI run inside a container), or
- provide a small bootstrap/ Terraform for S3 + DynamoDB so you can create backend purely with Terraform, or
- produce a ready-to-run Docker image Dockerfile that has both terraform and awscli for the Jenkins agent.

🧠 What is "Cache Dependencies" in Jenkins?

When Jenkins runs a pipeline or job, it often needs to **download dependencies** (like Maven packages, npm modules, Python libraries, Docker layers, etc.).
If Jenkins downloads them **fresh every time**, builds become **slow** and consume **extra bandwidth**.
➞ **Caching dependencies** means **reusing previously downloaded build dependencies** across builds or stages instead of downloading them again.

⚙️ Common Dependency Types to Cache

Tool / Language	What to Cache	Typical Location
Maven / Gradle	.m2/repository or .gradle	~/.m2/repository
Node.js (npm/yarn)	node_modules, cache dir	~/.npm, ~/.cache/yarn
Python (pip)	pip cache	~/.cache/pip
Docker	Docker image layers	/var/lib/docker
Go / Rust	Package cache	~/go/pkg/mod, ~/.cargo

☐ Summary

Concept	Explanation
Cache dependencies	Reuse previously downloaded build dependencies.
Why important	Improves build speed, efficiency, and stability.
Where configured	Jenkins job config, pipeline script, or plugins.
How configured	Persistent workspace, pipeline caching plugin, or Docker volumes.

--Terraform—--Syllabus (with One-Line Explanations)

1. Introduction & Basics

- **What is Terraform?** → Infrastructure as Code (IaC) tool for automating cloud provisioning.
- **Terraform vs Other IaC Tools** → Difference from Ansible, CloudFormation, Pulumi.
- **Installation & Setup** → Install Terraform CLI on local machine.
- **Terraform Workflow** → Init → Plan → Apply → Destroy cycle.

---

2. Core Concepts

- **Providers** → Plugins that let Terraform manage resources (AWS, Azure, GCP, etc.).
- **Resources** → Basic building blocks that define infrastructure components.
- **Data Sources** → Read existing cloud data and use it in configs.
- **Input Variables** → Parameters to make configs reusable.
- **Output Values** → Export information after deployment (e.g., IP address).
- **State Files (terraform.tfstate)** → Keeps track of infrastructure.
- **Terraform CLI Commands** → init, plan, apply, destroy, etc.

---

3. Configuration Language

- **HCL (HashiCorp Configuration Language)** → Language used for writing .tf files.
- **Expressions & Functions** → Use built-in functions for dynamic values.
- **Conditionals & Loops** → count, for_each, and if for reusable infra.

---

4. Terraform State Management

- **Local State** → State file stored locally.
- **Remote State** → Store state in backend like S3, GCS, or Terraform Cloud.
- **State Locking** → Prevents multiple users from modifying infra at the same time.
- **State Commands** → Inspect or manipulate state (terraform state list, show).

---

5. Modules (Reusability)

- **What are Modules?** → Group of resources used together.
- **Root Module vs Child Modules** → Main config vs reusable components.
- **Using Public Modules (Terraform Registry)** → Import ready-made modules.
- **Creating Custom Modules** → Write your own reusable code.

---

6. Provisioners & Lifecycle

- **Provisioners** → Run scripts/commands inside or after resource creation.
- **Local-Exec & Remote-Exec** → Run commands locally or on remote server.
- **Resource Lifecycle Rules** → Control create-before-destroy, ignore changes, etc.

7. Terraform Backends

- **Default Local Backend** → State stored locally.
- **Remote Backends (S3, GCS, etc.)** → Centralized state storage.
- **Terraform Cloud Backend** → Managed backend with collaboration features.

---

8. Terraform Workspaces

- **What are Workspaces?** → Isolated state environments (e.g., dev, test, prod).
- **Workspace Commands** → terraform workspace new, select, list.

---

9. Terraform Functions & Expressions

- **String Functions** → join, split, format.
- **Numeric & Collection Functions** → min, max, length.
- **Conditional Functions** → lookup, ternary operators.

---

10. Terraform with Cloud Providers

- **AWS with Terraform** → EC2, S3, VPC provisioning.
- **Azure with Terraform** → Resource groups, VM deployment.
- **GCP with Terraform** → Compute, storage, networking resources.

---

11. Terraform Advanced Features

- **Dynamic Blocks** → Create nested configurations dynamically.
- **For-Each vs Count** → Different looping strategies.
- **Depends-On** → Control resource creation order.
- **Custom Providers** → Build your own plugins.

---

12. Terraform Security

- **Sensitive Variables** → Hide secrets from logs.
- **Secrets Management** → Use Vault, AWS Secrets Manager, or SSM.
- **IAM Policies with Terraform** → Enforce least privilege.

---

13. Terraform Testing & Validation

- **Linting with terraform validate** → Check for syntax errors.
- **Formatting with terraform fmt** → Keep code clean.
- **Unit Testing** → Test infrastructure using Terratest or Kitchen-Terraform.

---

14. Terraform CI/CD

- **Terraform with Jenkins/GitHub Actions** → Automate deployments.
- **Plan & Apply in Pipeline** → Safe infra deployment with approvals.
- **Drift Detection** → Detect infra changes outside Terraform.

---

15. Terraform Enterprise / Cloud

- **Terraform Cloud** → SaaS for remote state & collaboration.
- **Terraform Enterprise** → Self-hosted version for enterprises.
- **Sentinel Policies** → Enforce policy-as-code (compliance & security).

Command	**Explanation**
terraform version	Check installed Terraform version
terraform init	Initialize Terraform & download providers
terraform fmt	Format Terraform files to standard style
terraform validate	Validate Terraform configuration syntax
terraform plan	Show what changes Terraform will apply
terraform apply	Apply Terraform changes to create/update infra
terraform apply -auto-approve	Apply without confirmation prompt
terraform destroy	Destroy all Terraform-managed resources
terraform destroy -auto-approve	Destroy infrastructure without prompt
terraform state list	List all resources tracked in state
terraform state show <resource>	Show details of a resource in state
terraform state rm <resource>	Remove resource from state without deleting it
terraform state mv <old> <new>	Rename/move resource in state
terraform output	Show all output variables
terraform output <name>	Show specific output variable
terraform output -json	Show outputs in JSON format
terraform providers	Display providers used in configuration
terraform graph	Show dependency graph of resources
mkdir -p modules/<name>	Create module folder structure
terraform import <res> <id>	Import existing resource into Terraform
terraform workspace list	List all workspaces
terraform workspace new <name>	Create a new workspace
terraform workspace select <name>	Switch workspace
terraform apply -var="key=value"	Pass variable from CLI
terraform apply -var-file="file.tfvars"	Use variables file
terraform apply -var-file="a.tfvars" -var-file="b.tfvars"	Use multiple variables files
export TF_LOG=DEBUG	Enable debugging logs
export TF_LOG_PATH=./file.log	Save debug logs to file
unset TF_LOG	Disable Terraform logs
terraform login	Login to Terraform Cloud
terraform logout	Logout from Terraform Cloud
terraform init -upgrade	Re-download/upgrade provider plugins
terraform init -reconfigure	Reconfigure backend settings
rm -f .terraform.tfstate.lock.info	Remove Terraform lock file
rm -rf .terraform	Delete .terraform directory
terraform fmt -recursive	Format all Terraform files in subfolders
terraform plan -destroy	Show what will be destroyed
terraform show	Display full Terraform state

--Terraform--

----Data Source in Terraform

A **Data Source** in Terraform is a way to **fetch and use existing information** from outside Terraform, instead of creating new infrastructure.

```
data "aws_vpc" "default" {
  default = true                          # We can use Filter here
}
resource "aws_subnet" "public" {
  vpc_id = data.aws_vpc.default.id      # Use VPC ID from existing default VPC
  cidr_block = "10.0.1.0/24"
}
```

---Passing Outputs & Values Between Modules in TF

- **Source Module:** output "value_name" { value = resource.attribute }

```
output "vpc_id" {
  value = aws_vpc.main.id                  # vpc_id = aws_vpc.myVPC.id
}
```

- **Root Module:** value = module.source_module.value_name

```
module "compute" {
   source = "./modules/compute"
   vpc_id = module.network.vpc_id # module.<MODULE NAME>.<OUTPUT NAME>
   }
```

- **Target Module:** variable "value_name" {} and then var.value_name

```
vpc_id = var.vpc_id            # Used the value passed in variable “vpc_id”
```

---Terraform import vs Terraform refresh

Feature	**Terraform import**	**Terraform refresh**
-Purpose	-Add an existing resource into state	-Sync state with real infra
-Changes .tf file?	-✗ No (you must write it)	-✗ No
-Changes state file?	-✓ Yes (adds new resource)	-✓ Yes (updates existing resource info)
-Use Case	-Adopt unmanaged infra	-Update state drift

--In short: import **is for onboarding,** refresh **is for syncing.**

---Terraform vs CloudFormation

Feature	Terraform	CloudFormation
-Cloud Support	-Multi-cloud (AWS, Azure, GCP, etc.)	-AWS-only (vendor lock-in)
-Language	-HCL (simple, readable)	-JSON/YAML (verbose)
-State Management	-External (S3, DynamoDB, etc.)	-Internal (managed by AWS)
-Rollback	-Manual (fix & re-apply)	-Automatic rollback
-Community	-Huge, active, many modules	-Smaller, AWS-focused
-Speed	-Fast, parallel apply	-Slower stack updates
-Modularity	-Strong (modules, registry)	-Nested stacks (less flexible)
-Best For	-Multi-cloud, hybrid, flexibility	-AWS-only shops, built-in safety

----Managing secrets (passwords, API keys, tokens, etc.) in Terraform

◆ Ways to Use Secrets in Terraform

1. Terraform Variables (Sensitive Variables)

- Mark variables as **sensitive** so they don’t show in CLI output or logs.

```
variable "db_password" {
  type      = string
  sensitive = true
}

resource "aws_db_instance" "example" {
  identifier = "mydb"
  engine     = "mysql"
  username   = "admin"
  password   = var.db_password
}
```

✅ Safer than plain variables.
⚠ But don’t hardcode in terraform.tfvars or .tf files. Use .tfvars + gitignore.

---

1. Environment Variables

- Pass secrets via environment variables.

```
# export TF_VAR_db_password="SuperSecret123"
# terraform apply
```

✅ Keeps secrets out of code.
⚠ Anyone with shell access can still read them.

---

2. **Secret Manager Services**

- Store secrets in **AWS Secrets Manager**, **AWS SSM Parameter Store**, or **Vault**, and fetch them dynamically.

Example with AWS Secrets Manager:

```
data "aws_secretsmanager_secret_version" "db_pass" {
  secret_id = "prod/db/password"
}

resource "aws_db_instance" "example" {
  identifier = "mydb"
  engine     = "mysql"
  username   = "admin"
  password   =
data.aws_secretsmanager_secret_version.db_pass.secret_string
}
```

✅ Best practice → secrets never live in Terraform files.

3. **HashiCorp Vault Integration**

- Use Vault provider to retrieve secrets dynamically.

```
provider "vault" {
  address = "https://vault.example.com"  }
data "vault_generic_secret" "db" {
  path = "secret/data/db"
}

resource "aws_db_instance" "example" {
  username = "admin"
  password = data.vault_generic_secret.db.data["password"]
}
```

✅ Industry standard for secret management.

---

4. **.tfvars / Backend Encryption**

- If using .tfvars, store them securely (e.g., in S3 with encryption or Terraform Cloud with Vault).
- Add .tfvars to .gitignore to avoid leaking secrets into Git.

◆ **Best Practices**

- **Never hardcode secrets** in .tf files.
- **Use sensitive variables** to hide secrets from logs.
- **Integrate with secret managers** (AWS Secrets Manager, Vault, SSM).
- **Encrypt state files** (Terraform state may store secrets).
- Use **remote backends** (e.g., S3 + DynamoDB + KMS) with encryption enabled.

-----Lifecycle Arguments in Terraform.

🌐 What is lifecycle in Terraform?

The **lifecycle block** in Terraform lets you control **how resources are created, updated, and destroyed**.

It goes **inside a resource block** and defines special behaviors for that resource.

⚡ Common Lifecycle Arguments

1. create_before_destroy

- Ensures **new resource is created before destroying the old one**.
- Prevents downtime during replacement.
- **Use Case**: Deployments, rolling updates.

```
resource "aws_instance" "web" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  lifecycle {
    create_before_destroy = true
  }
}
```

2. prevent_destroy

- Prevents Terraform from **destroying a resource**.
- If a terraform destroy or replacement would remove it, Terraform throws an error.
- **Use Case**: Critical data resources (RDS, S3 buckets).

```
resource "aws_s3_bucket" "prod_data" {
  bucket = "prod-critical-data"

  lifecycle {
    prevent_destroy = true
  }
}
```

3. ignore_changes

- Tells Terraform to **ignore specific attributes** if they change outside of Terraform (manual updates).
- Prevents Terraform from constantly trying to revert manual or external changes.
- **Use Case**: Fields managed outside Terraform (e.g., tags, auto-scaling, IPs).

```
resource "aws_instance" "web" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
  lifecycle {
    ignore_changes = [
      tags["LastUpdated"],              # Ignore LastUpdated tag
      user_data                         # Ignore manual user_data changes
    ]
  }
}
```

---

4. replace_triggered_by (Terraform v1.2+)

- Defines **dependencies that force replacement** when they change.
- If the listed resource or attribute changes, the current resource will be **re-created**.

```
resource "aws_instance" "web" {
  ami           = "ami-123456"
  instance_type = "t2.micro"

  lifecycle {
    replace_triggered_by = [
      aws_security_group.web_sg.id,   # Replace instance if SG changes
      aws_ebs_volume.data             # Replace instance if volume changes
    ]
  }
}
```

✅ Summary Table

Lifecycle Argument	Purpose
create_before_destroy	Avoid downtime by creating new resource before destroying old one
prevent_destroy	Protects critical resources from accidental deletion
ignore_changes	Ignores certain attribute changes made outside Terraform
replace_triggered_by	Forces replacement when another resource or attribute changes

---

⚡ Best Practices

- Use prevent_destroy for **production databases, critical buckets**.
- Use create_before_destroy for **servers, load balancers, node groups**.
- Use ignore_changes for **attributes managed by external tools**.
- Use replace_triggered_by for **tight dependencies between resources**.

--Docker— ---Syllabus (with One-Line Explanations)

1. Introduction to Docker

- **What is Docker?** → A containerization platform for building, shipping, and running applications.
- **Docker vs Virtual Machines** → Containers are lightweight, faster than VMs.
- **Docker Architecture** → Client, daemon, images, containers, registry.

---

2. Installation & Setup

- **Installing Docker** → Setup on Linux, Windows, Mac.
- **Docker Engine** → Core runtime to run containers.
- **Docker CLI & Commands** → Command-line interface for managing Docker.

---

3. Docker Images

- **What are Images?** → Read-only templates for containers.
- **Docker Hub & Registries** → Store and share images.
- **docker build** → Create custom images.
- **Dockerfile** → Instructions to build images.
- **Image Layers & Caching** → Efficient image builds.

---

4. Docker Containers

- **What are Containers?** → Running instances of images.
- **docker run** → Start a new container.
- **docker ps** → List running containers.
- **docker exec** → Run commands inside a container.
- **Container Lifecycle** → Start, stop, restart, remove.

---

5. Docker Networking

- **Bridge Network** → Default network for containers.
- **Host Network** → Shares host's network stack.
- **Overlay Network** → Multi-host communication (Swarm/K8s).
- **Port Mapping** → Expose container ports.
- **DNS & Service Discovery** → Container name resolution.

---

6. Docker Volumes & Storage

- **Volumes** → Persistent data storage outside containers.
- **Bind Mounts** → Map host directories into containers.
- **tmpfs Mounts** → Store data in memory.
- **Volume Drivers** → External storage integration.

---

7. Docker Compose

- **What is Compose?** → Tool to define multi-container apps.
- **docker-compose.yml** → YAML file for services.
- **Scaling Services** → Run multiple container instances.
- **Environment Variables in Compose** → Parameterize configs.

---

8. Docker Swarm (Orchestration)

- **Swarm Mode** → Native Docker clustering.
- **Services & Tasks** → Define long-running apps.
- **Managers & Workers** → Node roles in Swarm.
- **Load Balancing in Swarm** → Distribute traffic across nodes.

---

9. Docker with Kubernetes

- **Kubernetes vs Docker Swarm** → Different orchestration approaches.
- **Running Docker in Kubernetes** → Container runtime support.
- **kubectl with Docker** → Manage containers in clusters.

---

10. Security in Docker

- **Docker User Namespace** → Isolate container permissions.
- **Secrets Management** → Store credentials securely.
- **Image Scanning** → Detect vulnerabilities.
- **Best Practices** → Minimal base images, least privilege.

---

11. Monitoring & Logging

- **Docker Logs** → View container logs.
- **Docker Stats** → Monitor CPU, memory usage.
- **Log Drivers** → Forward logs to external systems.
- **Integration with Prometheus/ELK** → Advanced monitoring.

---

12. CI/CD with Docker

- **Docker in Jenkins/GitHub Actions** → Build & deploy pipelines.
- **Docker Registry in CI/CD** → Push and pull images.
- **Blue-Green Deployments** → Zero-downtime deployments.

---

13. Advanced Docker

- **Multi-Stage Builds** → Optimize image size.
- **Docker Contexts** → Manage multiple environments.
- **Docker Plugins** → Extend Docker functionality.
- **Rootless Docker** → Run without root privileges.

---

14. Best Practices

- **Small, Modular Images** → Keep images lightweight.
- **Use Official Base Images** → Ensure reliability and security.
- **Tagging Strategy** → Proper versioning of images.
- **Resource Limits** → Control CPU & memory usage.

----Commands

Command	**Explanation**
docker --version	Check Docker version installed
docker info	Check detailed Docker info
docker ps	List all running containers
docker ps -a	List all containers (including stopped)
docker images	List all images
docker pull <image>	Pull an image from Docker Hub
docker build -t <name> .	Build an image from Dockerfile
docker build --no-cache -t <name> .	Build image without cache
docker run <image>	Run a container from image
docker run -d <image>	Run container in detached mode
docker run -d -p 8080:80 <image>	Run container with port mapping
docker run --name <name> <image>	Run container with name
docker run -v /host:/container <image>	Run container with volume mount
docker start <container>	Start a stopped container
docker stop <container>	Stop a running container
docker restart <container>	Restart a container
docker rm <container>	Remove a stopped container
docker container prune	Remove all stopped containers
docker rmi <image>	Remove an image
docker rmi -f <image>	Force remove an image
docker image prune -a	Remove all unused images
docker logs <container>	View container logs
docker logs -f <container>	Follow container logs (live)
docker exec -it <container> bash	Execute command inside container
docker cp <container>:/path /host/path	Copy files container → host
docker cp /host/path <container>:/path	Copy files host → container
docker inspect <container>	Inspect container details
docker stats	Check resource usage of containers
docker top <container>	View running container processes
docker stop $(docker ps -q)	Stop all running containers
docker rm $(docker ps -aq)	Remove all containers
docker rmi $(docker images -q)	Remove all images
docker save -o image.tar <image>	Save image to tar file
docker load -i image.tar	Load image from tar file
docker export <container> > container.tar	Export a container’s filesystem
docker import container.tar <name>	Import container tar as image
docker tag <imageid> <username>/<repo>:tag	Tag an image
docker login	Login to Docker Hub
docker push <username>/<repo>:tag	Push image to Docker Hub
docker search <name>	Search for images on Docker Hub

docker network create <network>	Create docker network
docker network ls	List networks
docker network connect <network> <container>	Connect container to network
docker network disconnect <network> <container>	Disconnect container from network
docker volume create <volume>	Create named volume
docker volume ls	List volumes
docker volume rm <volume>	Remove volume
docker volume inspect <volume>	Inspect volume
docker system df	View Docker disk space usage
docker system prune -a	Remove all unused data
docker compose version OR docker-compose --version	Check Docker Compose version
docker compose up OR docker-compose up	Start services using docker-compose
docker compose up -d	Start in background
docker compose down	Stop all services
docker compose up --build	Rebuild containers
docker compose logs -f	View docker-compose logs
docker compose config	Validate docker-compose file
docker compose up --scale web=3 -d	Scale containers in compose

--Docker—

-----Dockerfile commands

A **Dockerfile** is a script containing instructions (commands) that Docker uses to build an image. Each command creates a new **layer** in the image. Below is a list of the most common **Dockerfile commands** :

Basic Commands

2. **FROM**
   - Sets the base image.
   - Example:
     # FROM ubuntu:20.04
   → Start from Ubuntu 20.04 as the base image.

---

3. **LABEL**
   - Adds metadata to the image.
   - Example:
     # LABEL maintainer="you@example.com" version="1.0"

---

5. **RUN**
   - Executes commands in a new layer during build.
   - Used to install packages or configure the image.
   - Example:
     # RUN apt-get update && apt-get install -y python3

---

4. **COPY**
   - Copies files/folders from host (local machine) to the image.
   - Example:
     # COPY app.py /app/

---

5. **ADD**
   - Like COPY, but with extra features:
     - Can extract **tar archives**.
     - Can fetch files from a **URL**.
   - Example:
     # ADD https://example.com/file.tar.gz /tmp/

---

6. **WORKDIR**
   - Sets the working directory for subsequent commands.
   - Example:
     # WORKDIR /app

---

7. **CMD**
- Defines the **default command** that runs when the container starts.
- Only one CMD is allowed (last one wins).
- Example:
  # CMD ["python3", "app.py"]

8. **ENTRYPOINT**
    - Defines the **main command** to run inside the container.
    - Works with CMD (CMD provides default arguments).
    - Example:
      ```
      # ENTRYPOINT ["python3"]
      # CMD ["app.py"]
      → Runs as: python3 app.py
      ```

---

9. **EXPOSE**
    - Documents the port the container listens on.
    - Does not actually publish the port (that's done with docker run -p).
    - Example:
      ```
      # EXPOSE 8080
      ```

---

10. **ENV**
    - Sets environment variables.
    - Example:
      ```
      # ENV APP_ENV=production
      ```

---

11. **ARG**
    - Defines build-time variables (used only when building).
    - Example:
      ```
      # ARG VERSION=1.0
      # RUN echo "Building version $VERSION"
      ```

---

12. **VOLUME**
    - Defines a mount point for persistent data.
    - Example:
      ```
      # VOLUME /data
      ```

---

13. **USER**
    - Sets the user to run commands inside the container.
    - Example:
      ```
      # USER appuser
      ```

---

14. **HEALTHCHECK**
    - Defines how Docker checks if the container is healthy.
    - Example:
      ```
      # HEALTHCHECK --interval=30s CMD curl -f http://localhost:8080/
      || exit 1
      ```

---

15. **SHELL**
    - Changes the default shell used in RUN commands.
    - Example:
      ```
      # SHELL ["/bin/bash", "-c"]
      ```

⚡ **Summary of Lifecycle Commands**:

- **Build-time**: FROM, RUN, COPY, ADD, ARG
- **Container runtime**: CMD, ENTRYPOINT, ENV, EXPOSE, VOLUME, USER, HEALTHCHECK

------CMD vs ENTRYPOINT in Dockerfile

◆ CMD

- Defines the **default command or arguments** for the container.
- Can be **overridden** when you run docker run <image> <command>.
- Only **one CMD** is allowed (the last one wins).
- Syntax options:
  - Shell form:
    ```
    # CMD echo "Hello"
    ```
  - Exec form (recommended):
    ```
    # CMD ["python3", "app.py"]
    ```

◆ ENTRYPOINT

- Defines the **main executable** of the container.
- Not easily overridden — any command you pass to docker run is treated as **arguments to the ENTRYPOINT**.
- Useful when you want the container to **always run a specific command**.
- Syntax (exec form is preferred):
  ```
  # ENTRYPOINT ["python3"]
  # CMD ["app.py"]
  → Runs as: python3 app.py
  ```

⚖ Key Difference

Feature	CMD	ENTRYPOINT
-Purpose	-Default command/args	-Fixed command (container "entry point")
-Override with docker run	-Fully overridden	-Only arguments can be overridden
-Best for	-Providing defaults	-Forcing a specific behavior

🔧 Examples

1. Using CMD only

```
FROM ubuntu
CMD ["echo", "Hello from CMD"]
```

- Running:
  ```
  # docker run myimage
  → Output: Hello from CMD
  ```
- Override:
  ```
  # docker run myimage echo "Hi"
  → Output: Hi
  ```

2. Using ENTRYPOINT only
FROM ubuntu
ENTRYPOINT ["echo"]

- Running:
  # docker run myimage Hello
  → Output: Hello
- Trying override:
  # docker run myimage Hi
  → Output: Hi (treated as argument, not new command)

---

3. Using ENTRYPOINT + CMD together
FROM ubuntu
ENTRYPOINT ["echo"]
CMD ["Hello from CMD"]

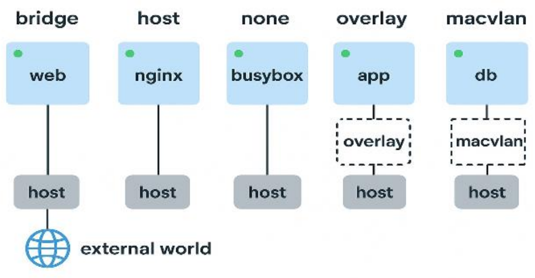
- Running:
  # docker run myimage
  → Output: Hello from CMD
- Override CMD:
  # docker run myimage Hi
  → Output: Hi (CMD overridden, ENTRYPOINT still executes)

✓ **Best Practice**

- Use **ENTRYPOINT** for the main command.
- Use **CMD** to provide default arguments.

-----Docker Networks

Network Type	Scope	Use Case
1.Bridge	Single host	-Default; inter-container communication on same host
2.Host	Single host	-High-performance, direct host networking
3.None	Single host	-No network (isolated workloads)
4.Overlay	Multi-host	-Swarm/K8s: multi-node communication
5.Macvlan	Multi-host	-Containers with direct LAN access, legacy apps



----Cloud Native Buildpacks

Cloud Native Buildpacks are tools that transform application source code into container images, without needing a Dockerfile.

They provide:
-Standardized and automated container image creation
-Security, compliance, and speed
-Language/framework-aware build process

Buildpacks were originally created by Heroku, and Cloud Native Buildpacks are now maintained by the Cloud Native Computing Foundation (CNCF).

--How Cloud Native Buildpacks Work

The CNB workflow has two main phases:

1. Detection Phase
-Determines which buildpacks apply to the source code.
-Example: A Node.js app will trigger a Node.js buildpack.

2. Build Phase
Buildpacks prepare the app:
-Install runtimes (Java, Node, Python)
-Run dependency installs
-Set up environment
-Package the app

--Components of Cloud Native Buildpacks

Component	Description
-Builder	-A collection of buildpacks, lifecycle, and base image
-Buildpack	-Logic to detect and build specific app types (e.g., Python, Go)
-Lifecycle	-The engine that manages detection, building, and image creation
-Stack	-Base image + build image used to create final image
-Platform	-Tool (like pack CLI or Tekton) that runs the lifecycle

--Build Lifecycle

```
[Source Code]
     ↓
[Detection]
     ↓
[Buildpacks Apply]
     ↓
[Build Process]
     ↓
[Container Image Ready]
```

--Why Use Buildpacks Instead of Dockerfiles?

Feature	Buildpacks	Dockerfile
-Ease of use	Auto-detects languages	Manual setup
-Security	Frequent updates	Manual image updates
-Build caching	Yes (layer caching)	Depends on Dockerfile
-Multi-language apps	Easier to build	Manual configuration
-CI/CD integration	Simple and declarative	More control needed

---How to Use Cloud Native Buildpacks (with pack CLI)

Step 1: Install the pack CLI
Official site: https://buildpacks.io/docs/tools/pack/

```
# brew install buildpacks/tap/pack  # macOS
```

Step 2: Create an Image from Source

```
# pack build my-app-image --path . --builder paketobuildpacks/builder:base
```

my-app-image: Name of your Docker image
--path .: Path to source code
--builder: Specifies the builder image to use

Step 3: Run the Image

```
# docker run -p 8080:8080 my-app-image
```

--Popular Buildpacks Providers
1. Paketo Buildpacks (by VMware)
2. Heroku Buildpacks
3. Google Cloud Buildpacks
4. Salesforce Buildpacks

--CNB Integration with Platforms

-Kubernetes via:
1.Tekton Pipelines + Buildpacks Task
2.kpack (Kubernetes-native image builder)
3.Knative

-CI/CD tools: GitHub Actions, GitLab CI, CircleCI, etc.
-Cloud Providers:
1.Google Cloud Build
2.Azure App Service
3.Heroku
4.VMware Tanzu

--Security & Compliance
-No root user by default
-Regular updates to buildpacks
-Minimal and known base images

--Example Folder (Node.js App)

```
/myapp
  ├── package.json
  ├── server.js
```

Build image:

```
# pack build my-node-app --builder paketobuildpacks/builder:base
```

No need for Dockerfile

--Summary

-Feature	**-Description**
No Dockerfile Needed	Buildpacks detect and build automatically
Secure & Updated Images	Community-maintained buildpacks
Framework Detection	Java, Node, Go, Python, etc.
Integration Friendly	Works with CI/CD, k8s, GitOps
Layered Caching	Faster rebuilds
Easy Testing	Fast local builds with pack CLI

--OTHER TOOLS—

-----1) Jira

◆ What is Jira?

- **Jira** is a **project management and issue tracking tool** developed by Atlassian.
- Widely used in **Agile and DevOps teams** for managing tasks, bugs, incidents, sprints, and releases.
- Integrates with CI/CD tools (Jenkins, GitHub, Bitbucket, GitLab, etc.) and monitoring tools.

◆ Uses of Jira in DevOps :

Jira plays a vital role in multiple stages of the **DevOps lifecycle**:

1. **Planning** – Define epics, stories, and tasks for development teams.
2. **Coding** – Link Jira issues to Git commits and pull requests.
3. **Building** – Track build status with integrations to Jenkins or CI/CD pipelines.
4. **Testing** – QA teams log bugs and track fixes within Jira.
5. **Releasing** – Automate deployments and update Jira tickets after successful releases.
6. **Monitoring** – Automatically create incidents when production issues are detected.
7. **Feedback Loop** – Gather insights and feed them back into planning for continuous improvement.

◆ How a DevOps Engineer Works with Jira :

- **Sprint Planning**: Work on stories/tasks assigned in Jira.
- **CI/CD Integration**: Connect Jenkins or GitHub Actions pipelines with Jira → build/deploy status automatically updates in tickets.
- **Incident Response**: When a production issue is raised in Jira, DevOps engineers investigate logs, infra, or pipelines and update the ticket with resolution.
- **Automation**: Use Jira's automation to trigger actions (e.g., auto-assign tickets, send Slack alerts).
- **Reporting**: Provide metrics on deployments, lead time, and incidents through Jira dashboards.

🎯 Interview-Ready Answer :

"Jira is a project management and issue tracking tool used in Agile and DevOps workflows. In DevOps, we use it for sprint planning, bug tracking, incident management, and release tracking. It integrates with CI/CD tools like Jenkins and GitHub so tickets automatically update with code commits, builds, and deployments. As a DevOps engineer, I use Jira to track my tasks, respond to incidents, and link infrastructure changes or pipeline deployments back to Jira issues to maintain full traceability."

--Real-Time Jira + Jenkins integration workflow
--HOW **DevOps engineers use Jira** to manage, automate, and track the complete CI/CD lifecycle.

🚀 Real-Time Jira DevOps Workflow (with Jenkins Integration)

Let's assume you're part of a DevOps team automating deployment pipelines for an e-commerce application.

☐ Tools Involved

Tool	Purpose
Jira	Track and manage development & deployment work
GitHub / Bitbucket	Source code repository
Jenkins	CI/CD automation
Slack	Notifications and alerts
AWS / Kubernetes (EKS)	Deployment platform

🧠 Step-by-Step Workflow

Step 1️⃣ – Create a Jira Ticket

- A **developer or DevOps engineer** creates a Jira issue:
  - **Type:** Story or Task
  - **Title:** "Implement Jenkins CI/CD pipeline for microservice X"
  - **Description:** Define pipeline stages, add test automation, and deploy to dev environment.
  - **Assignee:** Sudarshan (DevOps Engineer)
  - **Status:** *To Do*

👉 **Purpose:** This ticket will track all commits, builds, and deployments for that task.

Step 2️⃣ – Link Jira Issue to Code Repository

- Jira integrates with **GitHub/Bitbucket**.
- The developer includes the Jira issue ID in their commit message:
  ```
  git commit -m "JIRA-102: Added Jenkinsfile for CI/CD pipeline"
  ```
- Jira automatically links that commit and branch to the issue.

👉 **Purpose:** Every code change is traceable to a Jira issue (audit + accountability).

Step 3️⃣ – Jenkins Build Triggered Automatically

- When a commit is pushed:
  - Jenkins is triggered via **webhook** from GitHub.
  - Pipeline stages:
    1. Checkout Code
    2. Build (Maven, Gradle, npm, etc.)
    3. Run Tests
    4. Docker Build & Push
    5. Deploy to Dev Environment

🧠 Jenkinsfile Example:

```
pipeline {
  stages {
    stage('Build') { steps { sh 'mvn clean package' } }
    stage('Test') { steps { sh 'mvn test' } }
    stage('Dockerize') { steps { sh 'docker build -t app:v1 .' } }
    stage('Deploy') { steps { sh 'kubectl apply -f deployment.yaml' } }
  }
  post {
    success {
      jiraIssueUpdate issueKey: 'JIRA-102', comment: 'Build Successful ✅'
    }
    failure {
      jiraIssueUpdate issueKey: 'JIRA-102', comment: 'Build Failed ❌'
    }
  }
}
```

Step 4️⃣ — Jenkins Updates Jira Automatically

- Using the **Jira Plugin for Jenkins**, build results are sent to Jira:
  - If build passes → Jira status moves to **Testing**
  - If build fails → Jira status moves to **In Review** or **Reopen**
  - Comment automatically added in Jira issue:

✅ Jenkins Build #24: Success. Deployed to Dev Environment.

👉 **Purpose:** Automates communication between CI/CD tool and project tracker.

Step 5️⃣ — QA or UAT Testing

- QA team tests the deployed version.
- If any issue found:
  - QA creates a **Bug ticket** in Jira linked to original story.
  - Bug ID added to Jenkins for tracking (e.g., JIRA-115).

👉 **Purpose:** Continuous feedback loop for faster fixes.

Step 6️⃣ — Deploy to Production

- Once QA approves:
  - Jenkins pipeline promotes build from *staging* → *production*.
  - Deployment results automatically posted to Jira:

☐ Deployment to Production: Successful. Version: v1.0.4

- Jira issue transitions automatically:
  “In Progress” → “Deployed” → “Done”

Step 7️⃣ — Notifications via Slack

- Jira and Jenkins both send Slack notifications:
  - “JIRA-102: Build Success ✅ by Jenkins #24”
  - “Production Deployment Completed ☐”
- DevOps team instantly knows pipeline and release status.

☐ Final Workflow Summary Diagram (Text Format)

```
Jira Ticket Created (JIRA-102)
    ↓
Developer commits code (with issue ID)
    ↓
Jenkins triggers CI/CD build
    ↓
Build/Deploy results sent to Jira
    ↓
QA testing & feedback (linked bugs)
    ↓
Promote to Production
    ↓
Auto-update Jira → "Done"
    ↓
Slack Notifications sent
```

Benefits of This Integrated Workflow

Benefit	Description
Traceability	Each commit → build → deployment → Jira issue
Automation	Jira statuses updated via Jenkins
Visibility	Teams can monitor deployments from Jira dashboard
Reduced Manual Work	No need to manually change statuses or notify team
Audit & Compliance	Full change log available in Jira

How to Explain in Interview

Here's a concise way to say it:

"In our DevOps process, we integrated Jira with Jenkins and GitHub. Every code commit references a Jira issue ID, triggering an automated CI/CD pipeline in Jenkins. Build results and deployment updates are pushed back to Jira, automatically updating issue status and notifying the team via Slack. This ensures complete traceability from development to deployment, reduces manual work, and maintains transparency across the pipeline."

---Scrum, Epic, Story, Task, Bug

1. What is Scrum?

Scrum is an **Agile framework** used for managing complex software development.

It organizes work into short, repeatable cycles called **Sprints** (usually 1–2 weeks).

---

2. What is an Epic?

An **Epic** is a **large body of work** that can be divided into smaller parts (called Stories or Tasks).

Think of an Epic as a **big project goal** or **major feature**.

---

3. What is a Story (User Story)?

A **Story** (or **User Story**) is a **specific functionality** or **requirement** from the user's perspective.

---

4. What is a Task?

A **Task** is a smaller technical or operational activity required to complete a Story.

---

5. What is a Bug?

A **Bug** represents a problem or defect found during testing or in production.

---

6. Jira Work Flow — How It All Connects

Here's how everything fits together 👇

```
EPIC
├── STORY 1 → Create Jenkins pipeline for Service A
│       ├── TASK 1 → Install Jenkins
│       ├── TASK 2 → Write Jenkinsfile
│       ├── TASK 3 → Configure GitHub webhook
│
├── STORY 2 → Create Docker build & push automation
│       ├── TASK 1 → Write Dockerfile
│       ├── TASK 2 → Push to ECR
│
└── STORY 3 → Configure Kubernetes deployment
        ├── TASK 1 → Write Deployment YAML
        ├── TASK 2 → Test Deployment in EKS
```

---

7. Typical Jira Workflow (Status Flow)

Each issue (Epic, Story, Task, Bug) follows a **workflow**, e.g.:

To Do → In Progress → In Review → Testing → Done

- **To Do** – Created but not started
- **In Progress** – Work has begun
- **In Review** – Awaiting code review / testing
- **Testing** – QA verifying
- **Done** – Completed and verified

📄 *This flow can be customized per team.*

---

8. How Scrum Flow Works in Jira

1️⃣ Product Owner creates EPICS (big features)
2️⃣ Breaks EPICS into STORIES (specific functionalities)
3️⃣ Each STORY has TASKS (technical work)
4️⃣ During Sprint Planning → Team selects Stories & Tasks for current Sprint
5️⃣ During Sprint → Work moves from "To Do" → "In Progress" → "Done"
6️⃣ End of Sprint → Review & Retrospective → Next Sprint starts

---

🧠 **How to Explain in an Interview**

"In Jira, we follow the Scrum framework. We create Epics for large initiatives like CI/CD automation or Infrastructure setup. Each Epic is broken down into Stories that represent deliverable functionalities. These Stories are further divided into Tasks for implementation. We manage these during sprints using Scrum boards, tracking progress from 'To Do' to 'Done'. Bugs are logged and linked to their relevant stories for visibility. This helps maintain clear hierarchy, accountability, and smooth sprint delivery."

-----2) Fluentd

Fluentd is a popular open-source log collector, processor, and forwarder. In Kubernetes, it is typically deployed as a DaemonSet, so every node sends its logs to a central storage like Elasticsearch, Loki, or Cloud services.

Fluentd is an **open-source data collector** used for **log aggregation**.
It collects logs from different sources → filters & formats them → and sends them to various destinations (like Elasticsearch, S3, CloudWatch, etc.).

🧠 Simple Explanation

Think of **Fluentd** as a **log traffic controller** 🚦

It gathers logs from:

- Applications (e.g., Nginx, Kubernetes pods, Node.js)
- System logs
- Containers

Then it forwards them to:

- **Elasticsearch** (for Kibana dashboards)
- **AWS CloudWatch**
- **S3**, **Datadog**, **Splunk**, etc.

☐ Fluentd in a DevOps Stack

```
[Application / Pods]
       ↓
    Fluentd
       ↓
[Elasticsearch / CloudWatch / S3 / Loki / Datadog]
```

So in Kubernetes:

- Fluentd runs as a **DaemonSet** (one pod per node)
- Collects container logs from /var/log/containers/
- Sends them to your logging backend

⚙️ Advantages of Fluentd

✅ Unified logging layer – collects from multiple sources
✅ Supports 500+ plugins (inputs, filters, outputs)
✅ Lightweight and reliable
✅ Works perfectly in Kubernetes logging architecture
✅ Easily integrates with EFK Stack (Elasticsearch, Fluentd, Kibana)

☐ Fluentd Installation in Kubernetes using Helm

Fluentd → AWS CloudWatch Logs

We'll configure Fluentd to collect Kubernetes container logs and push them to **AWS CloudWatch**.

◆ Step 1: Prerequisites

- IAM Role or IAM User with permissions:

```
{
  "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "logs:CreateLogGroup",
          "logs:CreateLogStream",
          "logs:PutLogEvents",
          "logs:DescribeLogStreams"
        ],
        "Resource": "*"
      }
    ]
  }
```

Attach this role to your EKS worker nodes or service account using **IRSA**.

◆ **Step 2: Create Helm values file values-cloudwatch.yaml**

```
# Fluentd configuration for AWS CloudWatch

cloudwatch:
  enabled: true
  region: ap-south-1                      # Change your AWS region
  log_group_name: /eks/cluster-logs       # CloudWatch log group name
  log_stream_prefix: from-fluentd         # Stream prefix

# Fluentd input plugin
input:
  tail:
    path: /var/log/containers/*.log
    pos_file: /var/log/fluentd-containers.pos
    tag: kube.*
    parser: json

# Output plugin configuration
output:
  type: cloudwatch_logs
  log_group_name: /eks/cluster-logs
  auto_create_stream: true
  region: ap-south-1
```

◆ **Step 3: Install Fluentd**

```
helm install fluentd fluent/fluentd -n logging --create-namespace -f values-cloudwatch.yaml
```

◆ **Step 4: Verify**

Check logs:

```
# kubectl get pods -n logging
# kubectl logs -n logging <fluentd-pod-name>
```

Then go to **AWS Console → CloudWatch → Log groups → /eks/cluster-logs**

✅ You'll see Kubernetes container logs there!

-----3) Kafka in DevOps

- **Apache Kafka** is a **distributed event streaming platform** used for **real-time data pipelines** and **stream processing**.
- Works on **publish-subscribe model** with topics, producers, and consumers.

◆ **Applications of Kafka**

1. **Log aggregation** – collect logs from microservices.
2. **Real-time monitoring** – stream metrics to tools like Prometheus, ELK, or Splunk.
3. **Data pipelines** – connect apps, databases, and analytics systems.
4. **Event-driven architecture** – trigger services on events (e.g., order placed → notify shipping).
5. **Stream processing** – analyze live data (fraud detection, clickstream analysis).

◆ **DevOps Engineer's Work with Kafka**

- **Deploy & manage** Kafka clusters (on-prem or AWS MSK).
- **Configure topics & partitions** for scaling and fault tolerance.
- **Set up monitoring & alerting** (Grafana, Prometheus, CloudWatch).
- **Automate** with CI/CD for Kafka connectors and consumers.
- **Ensure security** (SSL, SASL, IAM policies in MSK).
- **Troubleshoot performance issues** (latency, consumer lag).

◆ Here's a **Kafka in DevOps workflow diagram**:

- **Developers** trigger builds → **Jenkins CI/CD**.
- Jenkins publishes events/logs → **Kafka Cluster**.
- **Producers** (apps, services) also send data to Kafka.
- Kafka distributes data to:
    - **Consumers** (for monitoring/logging).
    - **Prometheus/Grafana** (metrics visualization).
    - **ELK Stack** (centralized logging).

◆ **In short:**

- Kafka = event streaming backbone.
- Applications = log aggregation, monitoring, real-time data pipelines, CI/CD events.
- DevOps engineer's role = **deploy, secure, monitor, scale, and automate Kafka clusters and pipelines**.

✅ **One-liner summary for interview:**

"Kafka is a distributed event streaming platform. In DevOps, we use it for log aggregation, monitoring, real-time pipelines, and event-driven systems. As a DevOps engineer, I deploy, monitor, secure, and scale Kafka clusters, automate topic/connector setups, and ensure smooth data flow between services."

-----4) ArgoCD

ArgoCD (**Argo Continuous Delivery**) is a **GitOps-based continuous delivery tool** for Kubernetes.

- It keeps Kubernetes applications in sync with the desired state stored in **Git repositories** (Helm charts, Kustomize, or YAML manifests).
- Any change in Git → ArgoCD automatically (or manually) applies it to the Kubernetes cluster.
- Provides a **UI, CLI, and API** to manage deployments.

◆ ArgoCD Installation Steps (on Kubernetes)

1. Install ArgoCD

Run in your Kubernetes cluster (example: namespace argocd):

- Create namespace

  # kubectl create namespace argocd

- Install ArgoCD components

#kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

---

2. Expose ArgoCD API Server

Option A: For local testing (port-forward):

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

--Access UI: https://localhost:8080

Option B: For production, expose via Ingress/LoadBalancer.

---

3. Get Initial Admin Password

# Get admin password

# kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d; echo

--Login → Username: admin, Password: <above>

---

4. Install ArgoCD CLI (optional but useful)

# Linux

# curl -sSL -o /usr/local/bin/argocd https://github.com/argoproj/argo-cd/releases/latest/download/argocd-linux-amd64
chmod +x /usr/local/bin/argocd

# Verify

# argocd version

◆ Integration Steps with GitHub & Kubernetes

Step 1: Connect GitHub Repo

- In ArgoCD UI → **Settings** → **Repositories** → **Connect Repo**.
- Provide:
  - Repository URL (https://github.com/your-org/your-repo.git)
  - Authentication method (HTTPS with token or SSH key).

Step 2: Define Application (link GitHub to Kubernetes)
Using CLI (example):

```
# argocd app create myapp \
    --repo https://github.com/your-org/your-repo.git \
    --path k8s-manifests \
    --dest-server https://kubernetes.default.svc \
    --dest-namespace default
```

- --repo → GitHub repo URL
- --path → Folder inside repo containing Kubernetes YAML/Helm/Kustomize
- --dest-server → Kubernetes API server (default = in-cluster)
- --dest-namespace → Namespace where app will be deployed

Step 3: Sync the App

```
# First sync
    # argocd app sync myapp

# Check status
    # argocd app get myapp
```

ArgoCD will pull manifests from GitHub and deploy them to Kubernetes.

Step 4: Enable Auto-Sync (optional)

```
# argocd app set myapp --sync-policy automated
```

Now, whenever code is updated in GitHub, ArgoCD will automatically apply it to Kubernetes.

◆ End-to-End Flow

1. Developer pushes **YAML/Helm updates** → GitHub repo.
2. ArgoCD detects changes → compares Git (desired state) vs. Kubernetes (live state).
3. If different → ArgoCD applies changes → Kubernetes updated automatically.

✅ Summary:

- **Install**: Deploy ArgoCD to Kubernetes.
- **Integrate with GitHub**: Connect repo (HTTPS/SSH).
- **Integrate with Kubernetes**: Define argocd app mapping repo → cluster.
- **Sync**: Manual or auto-sync ensures Kubernetes matches Git.

-----5) SonarQube (With Jenkins)

In DevOps, SonarQube is used as a **continuous code quality and security gate**.

- It integrates with the CI/CD pipeline (like Jenkins) to **scan code automatically** on every build.
- Ensures code meets **quality, maintainability, and security standards** before deployment.
- Helps developers fix issues early in the development lifecycle (Shift Left approach).

Steps to Integrate SonarQube with Jenkins

1. **Install SonarQube Server**
   - Deploy SonarQube locally, on a VM, or via Docker/Kubernetes.
2. **Install SonarQube Scanner in Jenkins**
   - Go to **Manage Jenkins** → **Plugins** → **Install "SonarQube Scanner"**.
3. **Configure SonarQube in Jenkins**
   - Go to **Manage Jenkins** → **System** → **SonarQube Servers** → **Add**
   - Provide:
     - Name: SonarQube
     - Server URL (e.g., http://<sonarqube-server>:9000)
     - Authentication Token (from SonarQube).
4. **Configure SonarQube Scanner**
   - Go to **Manage Jenkins** → **Global Tool Configuration** → **SonarQube Scanner** → **Add**
   - Name: SonarScanner
5. **Add Project to SonarQube**
   - Create a new project in SonarQube.
   - Get the project key and token.

--Jenkins Pipeline Code (Declarative)

Here’s the pipeline stage code for SonarQube analysis:

```
pipeline {
    agent any

    tools {
        maven 'Maven3'           // If using Maven
        jdk 'JDK11'             // If using Java 11
    }

    environment {
        SONARQUBE = credentials('sonar-token') // Jenkins credentials ID for
SonarQube token
    }
    stages {
        stage('Checkout') {
            steps {
                git branch: 'main', url: 'https://github.com/your-org/your-repo.git'
            }
        }

        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }

        stage('SonarQube Analysis') {
            steps {
                withSonarQubeEnv('SonarQube') {
                    sh 'mvn sonar:sonar -Dsonar.projectKey=my-project -
Dsonar.login=$SONARQUBE'
                }
            }
        }

        stage('Quality Gate') {
            steps {
                timeout(time: 2, unit: 'MINUTES') {
                    waitForQualityGate abortPipeline: true
                }
            }
        }
    }
}
```

☞ **This pipeline will:**

1. Checkout code.
2. Build the project.
3. Run **SonarQube analysis**.
4. Wait for **SonarQube quality gate** result – fails the pipeline if quality standards are not met.

-----6) Trivy

Trivy is an **open-source vulnerability scanner** developed by Aqua Security.

- Scans **containers, Kubernetes, filesystems, and Git repos** for:
  - **OS package vulnerabilities** (e.g., Alpine, Ubuntu, CentOS)
  - **Application dependencies** (Java, Python, Node.js, etc.)
  - **Misconfigurations** (Docker, Kubernetes, Terraform, AWS, etc.)
  - **Secrets** (hardcoded passwords, API keys, tokens).
- It's lightweight, fast, and commonly used in **DevSecOps pipelines**.

☞ In **DevOps**, Trivy is integrated with **Jenkins pipelines** to ensure no vulnerable images/configs reach production.

◆ Installation Steps (Linux & Docker)

1. Install on Linux

--Download and install

```
# sudo apt-get install wget apt-transport-https gnupg lsb-release -y
# wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | sudo apt-key add -
# echo deb https://aquasecurity.github.io/trivy-repo/deb $(lsb_release -sc) main | sudo tee /etc/apt/sources.list.d/trivy.list
# sudo apt-get update
# sudo apt-get install trivy –y
```

2. Verify Installation

```
# trivy --version
```

3. Install via Docker (alternative)

```
# docker run aquasec/trivy --version
```

◆ Integration with Jenkins

1. Install Plugins

- Jenkins needs **Docker Pipeline Plugin** (if scanning images).

2. Configure Jenkins Node

- Ensure Jenkins agent has **Docker** and **Trivy installed**.
- Or run Trivy via Docker in the pipeline.

3. Store Credentials (Optional)

- If pushing images to private registry (like AWS ECR/DockerHub), configure credentials in Jenkins.

◆ Jenkins Pipeline for Trivy (Declarative)

Example pipeline that:

- Builds a Docker image.
- Runs **Trivy scan**.
- Publishes a vulnerability report.

```
pipeline {
   agent any
   environment {  IMAGE_NAME = "myapp:latest"  }

   stages {
      stage('Checkout') {   }
      stage('Build Docker Image') {   }

      stage('Trivy Scan') {
         steps {
                                             // Run Trivy scan and save results
            sh '''
               trivy image --exit-code 0 --severity LOW,MEDIUM,HIGH
$IMAGE_NAME > trivy-report.txt
               trivy image --exit-code 1 --severity CRITICAL $IMAGE_NAME
>> trivy-report.txt
            '''
         }
      }

      stage('Publish Trivy Report') {
         steps {
            publishHTML(target: [
               allowMissing: false,
               alwaysLinkToLastBuild: true,
               keepAll: true,
               reportDir: '.',
               reportFiles: 'trivy-report.txt',
               reportName: 'Trivy Security Report'
            ])
         }
      }
   }  }
```

◆ How This Helps in DevSecOps

- Every CI/CD run checks Docker images (or code) for **vulnerabilities & misconfigurations**.
- Pipeline **fails** if critical vulnerabilities are found.
- Reports provide visibility into **security risks** before deployment.

✓ **Summary:**

- **Trivy** = Vulnerability & misconfiguration scanner.
- **Installation** = Install via package manager or Docker.
- **Integration with Jenkins** = Run scans in pipeline stages.
- **Pipeline** = Builds image → Scans with Trivy → Publishes report → Blocks insecure builds.

-----7) OWASP

OWASP (**Open Web Application Security Project**) is a non-profit organization that focuses on improving the **security of software**.

- It provides **guidelines, tools, and standards** for application security.
- The most famous project is the **OWASP Top 10**, which lists the most critical web application security risks (e.g., SQL Injection, XSS, Broken Authentication, etc.).
- OWASP also maintains tools like:
  - **OWASP ZAP (Zed Attack Proxy)** → used for security scanning of web applications.
  - **Dependency-Check** → identifies vulnerable dependencies in your code.

☞ In DevOps, OWASP tools are integrated into **CI/CD pipelines (like Jenkins)** to automate security checks.

◆ Installation Steps (OWASP ZAP as example)

1. Install OWASP ZAP (Zed Attack Proxy)

On Linux:

```
# sudo apt update
# sudo apt install zaproxy -y
```

On Docker (recommended for CI/CD):

```
# docker pull owasp/zap2docker-stable
```

2. Verify Installation

```
# zap.sh -version
```

◆ Integration with Jenkins

1. Install OWASP ZAP Plugin (optional)

- Go to **Manage Jenkins** → **Plugin Manager** → **Available**.
- Search for **OWASP ZAP** → Install.

(You can also run OWASP ZAP via CLI or Docker without plugin.)

2. Configure Jenkins for OWASP

- Add OWASP ZAP as a build tool (if installed locally).
- Or configure **Docker Agent** to pull and run owasp/zap2docker-stable.

3. Setup Credentials (if testing secured apps)

- Store app credentials or tokens in Jenkins credentials store.

◆ Jenkins Pipeline for OWASP ZAP (Declarative)

Here’s an example Jenkins pipeline that runs a ZAP security scan against a web app:

```
pipeline {
    agent any

    stages {
        stage('Checkout') {          }
        stage('Build & Deploy') {        }

        stage('OWASP ZAP Security Scan') {
            steps {
                script {
                                            // Run ZAP baseline scan against target app
                    sh '''
                        docker run --rm owasp/zap2docker-stable zap-baseline.py \
                                    -t http://your-app-url.com \
                                    -r zap_report.html
                    '''
                }
            }  }

        stage('Publish Report') {
            steps {
                publishHTML(target: [
                    allowMissing: false,
                    alwaysLinkToLastBuild: true,
                    keepAll: true,
                    reportDir: '.',
                    reportFiles: 'zap_report.html',
                    reportName: 'OWASP ZAP Report'
                ])
            }
        }
    }  }
```

◆ How This Helps in DevSecOps

- Every Jenkins build runs **OWASP ZAP** scan.
- Generates a **security report** (zap_report.html).
- You can **fail the pipeline** if high-severity vulnerabilities are found.

✅ Summary:

- **OWASP** → Security best practices + tools (ZAP, Dependency-Check).
- **Installation** → Install ZAP via apt or Docker.
- **Integration with Jenkins** → Run ZAP in pipeline stages.
- **Pipeline** → Automates vulnerability scanning in CI/CD.

---Summary Comparison Table

Feature / Tool	**SonarQube**	**Trivy**	**OWASP ZAP**
Type	SAST (Static)	Vulnerability Scanner	DAST (Dynamic)
Scans	Source Code	Images, IaC, Repos	Running App
Focus	Code Quality & Security	Dependencies & Configs	Web App Security
Stage Used	Build Time (Pre-deploy)	Build or Deploy Time	Runtime (Post-deploy)
Integration	Jenkins, GitHub, GitLab	CI/CD, Kubernetes, Docker	CI/CD, Web apps
Output	Code issues	CVE reports, config issues	Runtime vulnerabilities
Example Use	Analyze Java code for bad practices	Scan Docker image for CVEs	Test web app for XSS

🚀 Typical DevSecOps Flow

1. **SonarQube** → Scan source code in CI/CD before building
2. **Trivy** → Scan Docker image and IaC before deployment
3. **OWASP ZAP** → Run after deployment to test web app security

Together, they provide **end-to-end security coverage**:
Code → Container → Application Runtime

-----8) Bitbucket

In **DevOps**, **Bitbucket** is a **Git-based source code management (SCM) tool** that helps teams **store, manage, and collaborate** on code efficiently throughout the **software development lifecycle**.

☐ Definition

Bitbucket is a **version control repository hosting service** developed by **Atlassian** (the same company behind Jira, Confluence, and Bamboo).
It supports **Git** (and earlier, Mercurial) for managing code, versioning, and collaboration.

🚀 Role of Bitbucket in DevOps

In DevOps, Bitbucket acts as a **central code repository** integrated with the CI/CD pipeline.
It helps developers **collaborate on code**, **review changes**, **track issues**, and **automate deployments** using pipelines.

⚙️ Key Features

Feature	Description
Git Repository Management	Host private/public Git repositories with access control.
Branching & Merging	Enables creation of branches for features, bug fixes, etc., and merging via pull requests.
Pull Requests & Code Review	Developers can review, comment, and approve changes before merging.
Bitbucket Pipelines (CI/CD)	Built-in CI/CD tool for automating build, test, and deploy processes directly from Bitbucket.
Integration with Jira	Seamless connection with Jira for issue tracking and linking commits to Jira tickets.
Access Control & Permissions	Manage user and group permissions to secure repositories.
Webhooks & Integrations	Integrates with tools like Jenkins, Docker, AWS, and Kubernetes for end-to-end automation.

♾️ How Bitbucket Fits in DevOps Lifecycle

DevOps Phase	Bitbucket's Role
Plan	Integrates with Jira for issue tracking and sprint planning.
Code	Central place for code collaboration using Git.
Build	Use Bitbucket Pipelines or integrate with Jenkins for builds.
Test	Automate testing via pipelines or external tools.
Deploy	Deploy directly from pipelines to AWS, Kubernetes, etc.
Monitor	Integration with monitoring tools (e.g., Datadog, Prometheus) for feedback loops.

🧠 Example Workflow

1. Developer creates a new branch → feature/login-page
2. Writes and commits code → pushes to Bitbucket
3. Creates a **Pull Request (PR)** for review
4. After approval → merged into main branch
5. **Bitbucket Pipeline** triggers build, test, and deployment automatically
6. Jira issue automatically updates to "Done"

🎬 Real-World Use

- **Startups** and **enterprises** use Bitbucket for source control and CI/CD.
- Example: A team uses **Bitbucket + Jira + Confluence + Bamboo** — all Atlassian tools — for a complete DevOps ecosystem.

✅ Advantages

- Strong integration with Atlassian tools (Jira, Confluence)
- Built-in CI/CD with **Bitbucket Pipelines**
- Secure access controls
- Easy collaboration through pull requests and code reviews

-----9) Harness

In **DevOps**, **Harness** is a **Continuous Delivery (CD) and Continuous Deployment platform** that automates the process of deploying, verifying, and rolling back applications safely and efficiently.

It's designed to simplify CI/CD pipelines with **automation, AI-driven verification, and advanced deployment strategies** like canary or blue-green — all with minimal manual effort.

☐ Definition

Harness is an **AI-powered software delivery platform** that helps DevOps teams automate:

- Continuous Integration (CI)
- Continuous Delivery (CD)
- Feature Flags
- Cloud Cost Management
- Chaos Engineering
- Security Testing Orchestration

It was founded by **Jyoti Bansal (founder of AppDynamics)** to make complex deployment processes **simple, safe, and fast**.

⚙ Why Harness in DevOps?

Traditional CI/CD tools (like Jenkins) often require:

- Manual scripting
- Custom plugins
- Maintenance of build agents

Harness, on the other hand, offers a **self-service platform** with:

- Visual pipelines
- Built-in rollback logic
- AI/ML-based verification
- Tight integrations with Git, Kubernetes, AWS, etc.

So it drastically reduces the manual effort needed for reliable deployments.

🧠 Harness in the DevOps Lifecycle

DevOps Stage	**Harness Role**
Plan	Integrates with Jira, GitHub, or GitLab for tracking features and issues.
Code & Build (CI)	Uses Harness CI module to build, test, and package applications.
Deploy (CD)	Automates deployments to Kubernetes, AWS, Azure, GCP, etc.
Verify	Uses machine learning to analyze logs and metrics (from Prometheus, Datadog, etc.) to detect anomalies.
Monitor & Rollback	Automatically rolls back if errors or anomalies are detected post-deployment.

🚀 Key Features

Feature	Description
Continuous Delivery (CD)	Automates end-to-end deployment with support for blue-green, canary, or rolling updates.
Continuous Integration (CI)	Allows you to build and test code with containers in a scalable cloud-native environment.
Feature Flags	Enables gradual feature rollouts without redeploying code.
Cloud Cost Management (CCM)	Analyzes cloud spend and provides cost-optimization insights.
Security Testing Orchestration (STO)	Integrates security scans into CI/CD pipelines.
Chaos Engineering	Tests app resilience by simulating real-world failures.
AI-Powered Verification	Automatically verifies deployments by analyzing monitoring tool data.

☐ Example Use Case: Deploying to Kubernetes

A typical Harness CD pipeline:

1. **Connects to GitHub/Bitbucket** → detects new commits
2. **Builds a Docker image** using Harness CI
3. **Pushes the image to ECR or GCR**
4. **Deploys to EKS/Kubernetes** with blue-green or canary strategy
5. **Verifies** deployment health using Prometheus or Datadog metrics
6. **Auto-rolls back** if anomalies are detected

🖥 Pipeline Example (Conceptual)

Harness pipelines are visual, but conceptually they look like this:

Build → Test → Deploy to Staging → Verify → Manual Approval → Deploy to Prod → Verify → Rollback (if needed)

Each step can integrate with external tools like:

- GitHub Actions (for CI)
- AWS EKS, ECS, Lambda
- Helm charts
- Terraform (for infra)
- Prometheus, New Relic (for monitoring)
- Jira (for ticket tracking)

✅ Advantages of Harness

- **No scripting required** — visual pipelines
- **AI-based verification** — auto rollback on failure
- **Multi-cloud support** — AWS, Azure, GCP, Kubernetes, etc.
- **Security and governance** — RBAC, audit logs, policy controls
- **Fast onboarding** — integrates easily with Git and cloud tools
- **Cost optimization** — reduces waste with built-in cost visibility

🏢 Companies Using Harness

- **eBay**
- **American Express**
- **SoulCycle**
- **Delphix**
- **Charles Schwab**

These companies use Harness for **safe, automated deployments** across complex environments.

🔐 Summary

Category	**Harness Role**
Tool Type	DevOps Automation Platform
Primary Focus	Continuous Delivery (CD) & Deployment Verification
Core Benefit	Faster, safer, AI-powered software delivery
Integrations	GitHub, Bitbucket, Jenkins, AWS, Kubernetes, Prometheus, Jira

---

-----Clear and Professional way to explain your work on a Web App (SaaS) in a **DevOps interview** — step by step.

---

1. Start with a One-Line Overview

"I've been working on a Web-based SaaS application where I implemented end-to-end DevOps practices — from CI/CD automation to containerized deployment on the cloud, ensuring scalability, reliability, and continuous delivery."

---

2. Explain the Architecture (High-Level)

"Our application follows a typical 3-tier architecture —
Frontend (React/Angular),
Backend (Node.js/Python/Java),
and **Database** (MySQL/PostgreSQL).
We deployed it as a **multi-tenant SaaS application** on AWS using **Kubernetes (EKS)** for orchestration and **Docker** for containerization."

---

3. Highlight Your DevOps Role and Tools

DevOps Phase	What You Can Say in Interview
1. Source Code Management	-"We used GitHub/GitLab for version control with branching strategies like feature, develop, and main branches."
2. CI/CD Pipeline	-"I created Jenkins pipelines (or GitHub Actions) for build, test, and deploy stages with automated notifications in Slack and Jira integration."
3. Infrastructure as Code (IaC)	-"I provisioned infrastructure using Terraform and managed configuration with Ansible to ensure consistent environments across multiple AWS accounts."
4. Containerization	-"Each microservice was containerized using Docker and stored in Amazon ECR."
5. Orchestration	-"Kubernetes handled scaling and deployment across clusters; Helm charts were used for release management."
6. Monitoring & Logging	-"We integrated Prometheus and Grafana for monitoring, and ELK stack/OpenSearch for log aggregation."
7. Security & Secrets	-"Secrets were managed using AWS Secrets Manager and IAM roles; CI pipelines had secret masking for security."

---

4. Explain Deployment Strategy

"We followed a **Blue-Green or Rolling Deployment** strategy to ensure zero downtime during releases. All deployments were automated and triggered by Git commits or tag creation."

---

5. Show Impact (Quantifiable Results)

"After automating the CI/CD pipeline and container deployments, our deployment time reduced by **40%**, and environment inconsistencies dropped significantly."

---

6. Example Summary (2-Minute Interview Answer)

"In my current project, I work on a multi-tenant SaaS web application deployed on AWS.
I designed and implemented CI/CD pipelines using Jenkins and Terraform for automated infrastructure provisioning.
Our microservices are containerized with Docker and deployed on EKS using Helm.
For monitoring, we use Prometheus and Grafana, while logs are centralized in ELK.
I also automated security checks and secret rotation via AWS Secrets Manager.
These practices reduced manual intervention, improved release frequency, and enhanced system reliability."

---

7. Pro Tip – Customize Based on Role

If you're interviewing for:

- **DevOps Engineer:** focus on automation, CI/CD, monitoring.
- **SRE:** focus on uptime, performance, observability.
- **Cloud Engineer:** focus on AWS services and scalability.

-----AWS-DevOps All PORTS Details

◆ 1. Understanding Port Basics

- **Ports** are communication endpoints for network connections.
- **Range:**
  - **0–1023** → Well-known (system) ports
  - **1024–49151** → Registered ports (used by specific apps)
  - **49152–65535** → Dynamic/private ports (used temporarily by clients)

◆ 2. Common Ports in AWS & Linux Administration

Port	**Protocol**	**Purpose / Service**	**Used In**
20, 21	TCP	FTP (File Transfer Protocol)	Transferring files
22	TCP	SSH (Secure Shell)	Login to EC2 / Linux servers
23	TCP	Telnet	Legacy remote shell (avoid)
25	TCP	SMTP	Mail transfer
53	TCP/UDP	DNS	Route53, CoreDNS
67, 68	UDP	DHCP	Dynamic IP assignment
80	TCP	HTTP	Web servers, ALB, ELB, Ingress
110	TCP	POP3	Mail retrieval
123	UDP	NTP	Time sync (used by servers)
143	TCP	IMAP	Mail
161, 162	UDP	SNMP	Network monitoring
389	TCP	LDAP	Directory services
443	TCP	HTTPS (SSL)	Secure web (ALB, ELB, API Gateway)
465	TCP	SMTPS	Secure email
514	UDP	Syslog	Logging servers
587	TCP	SMTP (Mail submission)	Email clients
873	TCP	Rsync	Data sync
1024+	TCP	Dynamic app communication	Kubernetes, Jenkins agents, etc.

◆ 3. Ports Commonly Used in AWS Services

Service	**Port(s)**	**Description**
EC2 SSH	22	Access Linux EC2 instances
EC2 RDP	3389	Access Windows EC2 instances
S3	443	HTTPS access to buckets
RDS MySQL	3306	Database connections
RDS PostgreSQL	5432	Database connections
RDS MSSQL	1433	SQL Server
ElastiCache Redis	6379	Cache connections
ElastiCache Memcached	11211	Cache connections
EKS API Server	443	Kubernetes control plane
Load Balancer Health Checks	80/443	Application endpoints
VPC Peering / Internal Services	Custom	As per app design

◆ 4. Ports Commonly Used in DevOps Tools

Tool	**Default Port(s)**	**Purpose**
Jenkins	8080 (HTTP), 8443 (HTTPS)	CI/CD web UI
SonarQube	9000	Code quality dashboard
Nexus / Artifactory	8081	Artifact repository
Docker Registry	5000	Private container registry
Kubernetes API Server	6443	Cluster management API
Kubelet	10250	Node-agent communication
ETCD	2379–2380	K8s key-value store
Prometheus	9090	Metrics collection & query
Alertmanager	9093	Alerts from Prometheus
Node Exporter	9100	OS-level metrics exporter
Grafana	3000	Dashboard visualization
Loki	3100	Log aggregation
Tempo / Jaeger	14268 / 16686	Tracing data collection/UI
Elasticsearch	9200	Search and analytics engine
Kibana	5601	Log dashboard
Logstash	5044	Log collection input
Fluentd / Fluent Bit	24224	Log forwarding
RabbitMQ	5672 (AMQP), 15672 (Web UI)	Messaging queue
Kafka	9092	Event streaming
Vault (HashiCorp)	8200	Secret management
Consul	8500	Service discovery
MinIO / S3 compatible	9000	Object storage
Terraform Cloud Agent	443 (HTTPS)	Remote backend
GitLab	80 / 443 / 8080 / 22	SCM and CI/CD
Bitbucket	7990	Git hosting
Argo CD	8080 (default), 443 (secured)	GitOps deployment
Harbor Registry	8080 / 4443	Container registry UI & API
OpenTelemetry Collector	4317 / 55680	Metrics/traces receiver

◆ 5. Ports in Kubernetes (EKS)

Component	Port	Description
API Server	6443	Kubernetes API
Controller Manager	10257	Control plane process
Scheduler	10259	Scheduling
Kubelet	10250	Node management
Kube Proxy	10249	Network proxy metrics
Node Exporter	9100	Node metrics
Prometheus	9090	Scrapes exporters
Grafana	3000	Displays dashboards

◆ 6. Ports for Monitoring Stack Example

Tool	Port	Role
Node Exporter	9100	Exposes host metrics
cAdvisor	8080	Container metrics
Prometheus	9090	Scrapes and stores metrics
Alertmanager	9093	Handles alerts
Grafana	3000	Visualizes Prometheus metrics

◆ 7. Security Group and Firewall Notes

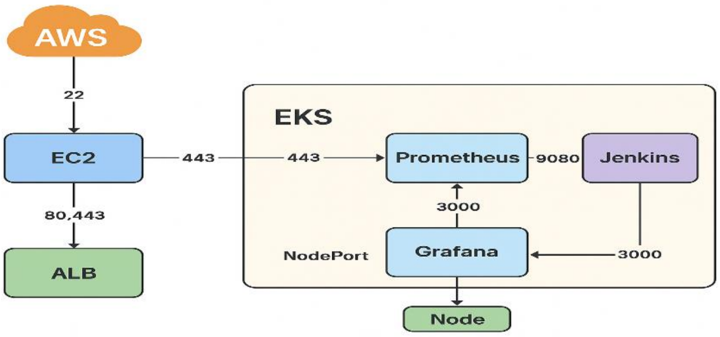
In **AWS Security Groups**, you'll commonly open:

- **22** → SSH (for EC2)
- **80 / 443** → Web access
- **8080, 9000, 9090, 3000** → For DevOps tools (as per requirement)
- Restrict by **CIDR** or **private subnet access** for internal tools.

◆ 8. Quick Reference Summary

Port Range	Use Case
1–1023	OS & network core services
1024–32767	Application servers (Jenkins, SonarQube, Prometheus, etc.)
32768–65535	Ephemeral ports for clients, dynamic assignments

--Visual of AWS–DevOps port connections



-----Nginx vs Apache

1. Basic Difference

Feature	Apache	Nginx
Type	Process-based web server	Event-based web server
Working style	Creates a new process/thread for each request	Handles many requests in a single thread using an event loop
Performance	Slower under heavy traffic	Faster and more efficient under high load
Configuration files	Uses .htaccess and httpd.conf	Uses nginx.conf
Developed by	Apache Software Foundation (1995)	Igor Sysoev (2004)

2. Usage (Where Each is Commonly Used)

Use Case	Best Choice	Why
Small websites or legacy apps	**Apache**	Easy to configure, wide module support
High-traffic sites / APIs / Reverse Proxy	**Nginx**	Handles more connections efficiently
Static content (HTML, images, videos)	**Nginx**	Faster static file serving
Dynamic apps (PHP, Python, Perl)	**Apache (with mod_php)**	Built-in modules for dynamic content
Load balancing or reverse proxy	**Nginx**	Lightweight and efficient for proxying traffic

3. Advantages

1. **Apache Advantages**
   - Easy to set up for beginners
   - Supports .htaccess for per-directory configuration
   - Large community and extensive modules
   - Better for dynamic content (works directly with backend modules)

2. **Nginx Advantages**
   - High performance & scalability
   - Uses less memory and CPU
   - Excellent for serving static files
   - Works great as a **reverse proxy**, **load balancer**, or **API gateway**
   - Modern design — ideal for microservices and containers

Example Combination (Best Practice)

Many modern setups use **both together**:

- **Nginx** as a **reverse proxy** in front of **Apache**.
- Nginx handles static files and load balancing.
- Apache handles dynamic content.

Client → Nginx (reverse proxy) → Apache (backend)

🌐 What is a Reverse Proxy in Nginx?

A **reverse proxy** is like a **middleman** between the **client (user)** and your **backend servers (apps, APIs, or web servers)**.

☞ It **receives requests** from users → then **forwards them** to one or more backend servers → and **sends the response** back to the user.

🧠 Simple Example (How it works)

Without reverse proxy:
User → directly → App Server (e.g., Apache)

With Nginx reverse proxy:
User → Nginx (Reverse Proxy) → Apache / Node.js / Flask App

Nginx hides the backend servers from users and manages all traffic smartly.

✓ Summary

Feature	Nginx (Reverse Proxy)	Apache
Role	Middle layer between user & backend	Web server (serves app content)
Performance	High, handles many connections	Good for dynamic content
Used for	Load balancing, SSL termination, routing	Hosting apps (WordPress, PHP, etc.)
Example	Nginx → Apache	Apache directly serves users

⚙ What Nginx Reverse Proxy Does

Function	Description
Load balancing	Distributes traffic between multiple backend servers
Caching	Stores copies of responses to serve faster next time
SSL termination	Handles HTTPS encryption so backend servers can use plain HTTP
Security layer	Hides internal server details and filters bad requests
Centralized routing	Routes traffic to different apps (e.g., /api → Node.js, /app → Apache)

--where real-world companies use Nginx and Apache

1. Nginx — Real-World Usage

Nginx is designed for **speed, scalability, and handling millions of connections**, so it's used by high-traffic, cloud-native companies.

Company	Use Case	Why Nginx?
Netflix	Serves billions of video streaming requests per day	Nginx acts as a **reverse proxy + load balancer** to distribute requests to backend video servers. It can handle massive concurrent connections efficiently.
Airbnb	API gateway and reverse proxy	Nginx routes API traffic and improves latency between microservices.
GitHub	Serves static assets (images, JS, CSS)	Nginx's event-driven architecture handles static files super fast.

Instagram	Load balancing and caching	Nginx manages heavy photo and API requests efficiently.
Twitch	Reverse proxy and streaming	Nginx manages live stream requests and handles millions of concurrent viewers.
Dropbox	Load balancing & SSL termination	Nginx provides secure, scalable traffic distribution between data centers.
Cloudflare	Core of its CDN & proxy platform	Built on Nginx to handle high concurrency, caching, and security.

✅ **Summary of why companies use Nginx:**

- Handles **massive traffic efficiently**
- Perfect for **microservices**, **reverse proxy**, **API gateway**
- Fast **static content serving**
- Excellent for **SSL termination**, **caching**, **load balancing**

2. Apache — Real-World Usage

Apache is widely used by **traditional web applications** and **content management systems (CMS)** — especially PHP-based stacks.

Company / Platform	**Use Case**	**Why Apache?**
Wikipedia	Runs MediaWiki (PHP-based)	Apache integrates tightly with PHP and supports .htaccess for flexible configs.
LinkedIn (early years)	Classic web serving	Apache was used before LinkedIn moved to more scalable architectures.
Facebook (early years)	Served PHP content	Used Apache + mod_php before switching to HipHop VM and Nginx.
WordPress.com	Hosts millions of PHP WordPress sites	Apache supports PHP natively and easily customizes per site.
IBM, Adobe, Oracle websites	Static + dynamic business web portals	Apache is stable, easy to configure, and enterprise-trusted.
Government / University websites	Hosting multiple legacy applications	Apache supports multiple virtual hosts and legacy apps well.

✅ **Summary of why companies use Apache:**

- **Strong support for dynamic content (PHP, Perl, Python)**
- **Easy configuration** using .htaccess
- Great for **shared hosting** (many small websites on one server)
- Huge ecosystem and **enterprise reliability**

🧠 **Typical Architecture in Modern Companies**

Many companies actually **use both** together:

Example: Netflix-style setup

User → Nginx (Reverse Proxy / CDN Layer) → Apache / Node / Python backend servers

- **Nginx** handles routing, caching, SSL
- **Apache** (or another backend) processes the business logic or serves dynamic content

-----GitOps in Devops

GitOps is a modern **DevOps practice** that uses **Git** as the **single source of truth** for managing infrastructure and application configurations.
In simple words:
☞ Everything (infrastructure, application configuration, Kubernetes manifests, etc.) is stored and version-controlled in a **Git repository**.
☞ When you make any change (like updating a deployment file), that change is pushed to Git.
☞ An automated tool then **detects the change** and **applies it automatically** to the running environment (like Kubernetes cluster).

◆ **Key Concepts of GitOps:**

1. **Single Source of Truth**
   - All configuration files and code are stored in Git repositories.
   - Git history tracks every change.
2. **Automation**
   - Tools automatically sync what's in Git with what's running in the cluster (for example, using **Argo CD** or **Flux**).
3. **Continuous Reconciliation**
   - The system continuously checks that the running environment matches the Git configuration.
   - If something changes manually, the system corrects it back to match Git (self-healing).
4. **Pull-based Deployment**
   - Instead of pushing changes manually, the GitOps operator **pulls updates** from Git and applies them safely.

◆ **Benefits of GitOps:**

- ✅ **Version control:** Every change is tracked in Git.
- ✅ **Auditability:** You can see who changed what and when.
- ✅ **Consistency:** Ensures environments stay in sync with Git.
- ✅ **Easy rollback:** Simply revert the Git commit to roll back changes.
- ✅ **Improved security:** No direct kubectl or manual access to production.

◆ **Tools used in GitOps:**

- **Argo CD** (by CNCF)
- **Flux CD**
- **Jenkins X**
- **GitLab CI/CD with GitOps patterns**

◆ **Example (Kubernetes scenario):**

1. You define your Kubernetes deployment YAML files in a Git repo.
2. You commit and push an update to the repo (e.g., new Docker image version).
3. Argo CD notices the change and automatically updates your Kubernetes cluster to match it.
4. If you need to roll back, just revert the Git commit — Argo CD reverts the cluster.

◆ **Real-world use case:**

- **Companies like Amazon, Intuit, and Spotify** use GitOps to manage their Kubernetes deployments for automation, reliability, and fast recovery.

--Real Example of GitOps Workflow with Argo CD (step-by-step)

🚀 GitOps Workflow with Argo CD

Step 1: Prerequisites

You need:

- A **Kubernetes cluster** (EKS, AKS, GKE, or Minikube)
- **kubectl** configured for your cluster
- A **Git repository** (GitHub or GitLab)
- **Argo CD** installed in your cluster

Step 2: Install Argo CD

You can install Argo CD in the Kubernetes cluster using Helm or kubectl.

Using kubectl:

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

Access Argo CD UI:

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

Then open in your browser: 👉 https://localhost:8080

-Default username: admin
-To get password:

```
# kubectl -n argocd get secret argocd-initial-admin-secret -o
jsonpath="{.data.password}" | base64 -d
```

Step 3: Set Up Your Git Repository

Example Git repository structure:

```
gitops-repo/
│
├── app/
│   ├── deployment.yaml
│   ├── service.yaml
│   └── ingress.yaml
│
└── kustomization.yaml
```

- These YAML files define your Kubernetes resources (deployment, service, etc.).
- You can update them anytime (for example, change image version).

Step 4: Connect Argo CD to Git Repo

In the **Argo CD UI**, click **"NEW APP"** and fill details like:

Field	Example
Application Name	myapp
Project	default
Sync Policy	Automatic
Repository URL	https://github.com/username/gitops-repo.git
Path	app/
Cluster	https://kubernetes.default.svc
Namespace	default

Then click **Create** ✅

Step 5: Argo CD Deploys Automatically

- Argo CD detects the YAML files in your Git repo.
- It creates the Kubernetes resources automatically in your cluster.
- Any change in Git triggers **auto-sync**, updating your cluster.

Step 6: Update Your App

Example — update Docker image tag in deployment.yaml:

```
containers:
 - name: myapp
  image: amitabhdevops/myapp:v2
```

Commit and push to Git:

```
# git add .
# git commit -m "Update app image to v2"
# git push
```

✅ Argo CD will **detect the commit**, **pull the changes**, and **update your Kubernetes cluster automatically**.

Step 7: Rollback (Easy Undo)

If something breaks:

1. Revert your last Git commit:
2. git revert HEAD
3. git push
4. Argo CD will automatically roll your cluster back to the previous working version.

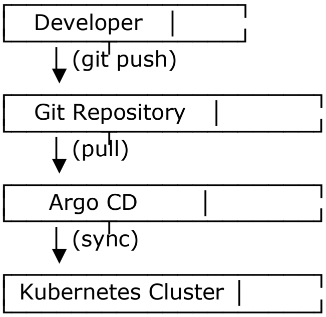
Step 8: Monitor & Manage

- Argo CD UI shows application **status** (Synced / Out of Sync / Healthy).
- You can click the app to view **pods, deployments, and sync history**.

✅ **Benefits of GitOps with Argo CD**

- Full **automation** (no manual kubectl apply)
- **Audit trail** via Git commits
- **Quick rollback** through Git history
- **Consistent deployments** across environments

◆ **GitOps Architecture Overview**



--Helm-- in Kubernetes

Helm is a package manager for Kubernetes, similar to apt for Ubuntu or yum for CentOS. It simplifies the deployment and management of Kubernetes applications by using Helm Charts, which are pre-configured application resources.

--Why Use Helm

Without Helm, deploying complex applications (e.g., with 10+ YAML files) is hard to manage. Helm:

-Reduces manual YAML writing
-Supports versioning, upgrades, and rollback
-Makes it easy to share and reuse application configurations
-Supports templating for dynamic values (like service name, image version,replicas

--Key Concepts

Concept	**Description**
1. Chart	-A Helm package. Contains YAML files and templates for Kubernetes r esources.
2. Release	-A running instance of a Helm chart in a Kubernetes cluster.
3. Repository	-A location where charts are stored and shared (e.g., artifacthub.io).
4. Values.yaml	-A file to define custom values passed to templates (like variables).
5. Templates	-YAML files with Go-based templating syntax used to generate Kubernetes resources dynamically.

--Helm Chart Structure

```
my-chart/
├── Chart.yaml            # Chart metadata
├── values.yaml           # Default configuration values
├── templates/            # Templated Kubernetes YAML files
│   ├── deployment.yaml
│   ├── service.yaml
│   └── _helpers.tpl      # Helper templates
```

--Basic Helm Commands

```
# Add a Helm chart repository = helm repo add bitnami
https://charts.bitnami.com/bitnami

# Update repo = helm repo update

# Search for a chart = helm search repo nginx

# Install a chart = helm install my-nginx bitnami/nginx

# Upgrade release = helm upgrade my-nginx bitnami/nginx --set replicaCount=3
# Uninstall = helm uninstall my-nginx

# List releases = helm list

# Dry run to see rendered YAML = helm install my-nginx bitnami/nginx --dry-run --
debug
```

--Real-World Example:
Deploy a WordPress site with just one command:

```
#helm install my-wordpress bitnami/wordpress
```

This creates pods, services, PVCs, secrets, etc., all configured through templates in the chart.

--Helm vs kubectl

Feature	kubectl	helm
1. Raw YAML needed	✓ Yes	✗ No (uses templates)
2. Easy upgrades	✗ Manual	✓ Yes (helm upgrade)
3. Version control	✗ Hard	✓ Yes
4. Reuse config	✗ Not easy	✓ Yes (values.yaml)

--When to Use Helm
-Deploying multi-component apps
-Releasing apps across multiple environments (dev, staging, prod)
-Automating CI/CD pipelines
-Managing frequent updates and rollbacks

-----Kubernetes Helm Commands: Basic to Advanced

Helm is the package manager for Kubernetes, simplifying application deployment, upgrades, and management. Below is a categorized list of Helm commands from basic to advanced.

--Summary: -Basic: install, upgrade, uninstall, list, repo
-Intermediate: --set, -f values.yaml, get, show, lint
-Advanced: dependency, history, plugin, create, verify

--1. Basic Helm Commands

-Chart Management

Command	Description
1. helm search hub <keyword>	Search Helm Hub for charts (e.g., helm search hub nginx)
2. helm search repo <keyword>	Search charts in added repositories
3. helm repo add <repo-name> <repo-url>	Add a Helm repository (e.g., helm repo add bitnami https://charts.bitnami.com/bitnami)
4. helm repo list	List all configured Helm repositories
5. helm repo update	Update local repository cache
6. helm repo remove <repo-name>	Remove a Helm repository

-Installation & Management

Command	Description
1. helm install <release-name> <chart>	Install a chart (e.g., helm install my-nginx bitnami/nginx)
2. helm list	List deployed releases
3. helm status <release-name>	Check the status of a release
4. helm uninstall <release-name>	Uninstall a release
5. helm upgrade <release-name> <chart>	Upgrade a release (e.g., helm upgrade my-nginx bitnami/nginx)
6. helm rollback <release-name> <revision>	Rollback to a previous version (e.g., helm rollback my-nginx 1)

--2. Intermediate Helm Commands

-Customizing Deployments

Command	Description
1. helm install <release> <chart> --set key=value	Override values at install (e.g., --set replicaCount=2)
2. helm install <release> <chart> -f values.yaml	Use a custom values file
3. helm get values <release-name>	View configured values for a release
4. helm show values <chart>	Display default values for a chart

-Debugging & Inspection

Command	Description
1. helm template <release> <chart>	Render templates locally (dry-run)
2. helm lint <chart-path>	Check a chart for errors
3. helm get manifest <release-name>	View generated Kubernetes manifests
4. helm get hooks <release-name>	List hooks for a release
5. helm get notes <release-name>	Show release notes

--3. Advanced Helm Commands

-Dependency Management

Command	Description
1. helm dependency update <chart>	Update chart dependencies
2. helm dependency build <chart>	Rebuild charts/ directory
3. helm dependency list <chart>	List chart dependencies

-Release & History Management

Command	Description
1. helm history <release-name>	View release history
2. helm rollback <release> <revision>	Revert to a specific revision
3. helm pull <chart> --untar	Download and extract a chart

-Plugins & Custom Charts

Command	Description
1. helm plugin install <plugin-url>	Install a Helm plugin
2. helm plugin list	List installed plugins
3. helm create <chart-name>	Scaffold a new Helm chart
4. helm package <chart-path>	Package a chart into a .tgz file
5. helm verify <chart.tgz>	Verify chart integrity

-Security & Secrets

Command	Description
1. helm install --generate-name	Auto-generate a release name
2. helm install --atomic	Automatically rollback on failure
3. helm install --dry-run --debug	Simulate an install with debug output
4. helm env	Show Helm environment variables

--4. Helm 3 vs Helm 2 Differences

Helm 3	Helm 2
1. No Tiller (server-side component)	Required Tiller (security risk)
2. helm uninstall (replaces helm delete)	helm delete
3. Secrets store release info (instead of ConfigMaps)	Used ConfigMaps
4. Better dependency management	Less efficient dependency handling

-----Enhanced values.yaml file example for a Helm chart:

-Configurability
-Clear structure
-Secure defaults
-Resource limits
-Readiness & liveness probes
-Support for autoscaling
-Use of annotations and labels
-Externalized configuration

--Enhanced values.yaml (Best Practices Example)

```yaml
# values.yaml

# Global labels for all resources
global:
  labels:
    app.kubernetes.io/managed-by: "Helm"
    app.kubernetes.io/version: "1.0.0"

replicaCount: 2

image:
  repository: myapp/image
  tag: "1.0.0"
  pullPolicy: IfNotPresent

imagePullSecrets: []
# Example: [ { name: myregistrykey } ]

nameOverride: ""
fullnameOverride: ""

serviceAccount:
  create: true
  name: ""

podAnnotations: {}
podLabels: {}

podSecurityContext:
  runAsNonRoot: true
  runAsUser: 1000
  fsGroup: 2000
```

```yaml
securityContext:
  readOnlyRootFilesystem: true
  allowPrivilegeEscalation: false
  capabilities:
    drop:
      - ALL

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
  className: "nginx"
  annotations: {}
  hosts:
    - host: myapp.local
      paths:
        - path: /
          pathType: Prefix
  tls: []
  # - secretName: myapp-tls
  #   hosts:
  #     - myapp.local

resources:
  limits:
    cpu: "500m"
    memory: "256Mi"
  requests:
    cpu: "200m"
    memory: "128Mi"

autoscaling:
  enabled: true
  minReplicas: 2
  maxReplicas: 5
  targetCPUUtilizationPercentage: 80
  # targetMemoryUtilizationPercentage: 75

livenessProbe:
  httpGet:
    path: /healthz
    port: http
  initialDelaySeconds: 10
  periodSeconds: 10
  failureThreshold: 3
```

```
readinessProbe:
  httpGet:
    path: /ready
    port: http
  initialDelaySeconds: 5
  periodSeconds: 5
  failureThreshold: 3

nodeSelector: {}
tolerations: []
affinity: {}

config:
  logLevel: info
  enableFeatureX: true
  maxConnections: 100

env:
  - name: ENVIRONMENT
    value: "production"
  - name: LOG_FORMAT
    value: "json"

volumes: []
volumeMounts: []

# Optional secrets
secretEnv: []
# Example:
# - name: DB_PASSWORD
#   secretName: myapp-secrets
#   key: db-password
```

--Best Practices Followed

Best Practice	**Example**
🔒 Security Context	Drop all capabilities, run as non-root
🎁 Configurable Probes	Liveness/readiness defined
🎯 Autoscaling Enabled	Configurable HPA settings
🚀 Image Pull Policy	Set to IfNotPresent
💻 Pod Labels & Annotations	Fully customizable
🌐 Ingress Config	TLS and path configuration included
🧠 Resources Defined	CPU and memory requests & limits
🔐 Secret Management	Example for secure environment injection
☐ Structure	Clear and modular for customization

------Step-by-step guide to install Prometheus and Grafana using Helm in a Kubernetes cluster:

--Pre-requisites
1. A running Kubernetes cluster
2. kubectl configured to access your cluster
3. Helm installed and configured

--Step-by-step Installation
--Step 1: Add Helm Repositories

```
 # helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
 # helm repo add grafana https://grafana.github.io/helm-charts
 # helm repo update
```

--Step 2: Create a Namespace (Optional but Recommended)

```
 # kubectl create namespace monitoring
```

--Step 3: Install Prometheus using Helm

```
 # helm install prometheus prometheus-community/prometheus \ --
namespace monitoring
```

-This installs:
-Prometheus server, Alertmanager, Node exporter, Pushgateway

--Step 4: Install Grafana using Helm

```
helm install grafana grafana/grafana \
  --namespace monitoring \
  --set adminPassword=admin123 \
  --set service.type=NodePort
```

-adminPassword=admin123 sets the Grafana admin password.

--Step 5: Access Grafana Dashboard

1. Get the Grafana NodePort:
```
            # kubectl get svc -n monitoring grafana
```

2. Open the service in your browser: http://<NodeIP>:<NodePort>

-Use: -Username: admin -Password: admin123 (or whatever you set above)

--Step 6: Configure Prometheus as Data Source in Grafana

-In Grafana UI:
 1. Go to ⚙ Configuration → Data Sources → Add data source
 2. Choose Prometheus
 3. URL: http://prometheus-server (or use the full service name: http://prometheus.monitoring.svc.cluster.local)
 4. Click Save & Test

--Step 7: Import Dashboards (Optional)

-You can import ready-made dashboards:
 1. Go to + Create → Import
 2. Use a dashboard ID like 1860 (Node Exporter Full)
 3. Click Load, select Prometheus as source, and import

-Optional: Port Forward (For Local Access)

-If you don't have a LoadBalancer/NodePort:

```
 # kubectl port-forward svc/grafana 3000:80 -n monitoring
 # kubectl port-forward svc/prometheus-server 9090:80 -n monitoring
```

Now open:
 1. Grafana: http://localhost:3000
 2. Prometheus: http://localhost:9090

--Uninstall Instructions
```
 # helm uninstall grafana -n monitoring
 # helm uninstall prometheus -n monitoring
 # kubectl delete namespace monitoring
```

Customized Installation (for production): # prometheus-values.yaml
```
alertmanager:
  enabled: true

server:
  persistentVolume:
    enabled: true
    size: 50Gi
  resources:
    limits:
      cpu: 1000m
      memory: 2Gi
    requests:
      cpu: 500m
      memory: 1Gi
```

Customized Installation (for production : # grafana-values.yaml
```
persistence:
  enabled: true
  size: 10Gi

adminUser: admin
adminPassword: "your-secure-password"

service:
  type: LoadBalancer

datasources:
  datasources.yaml:
    apiVersion: 1
    datasources:
    - name: Prometheus
      type: prometheus
      url: http://prometheus-server.monitoring.svc.cluster.local
      access: proxy
      isDefault: true
```

-----Components in Prometheus Installation

Prometheus itself has a **modular ecosystem**, but it doesn't automatically install everything.

Core components:

1. **Prometheus Server**
   - Scrapes metrics from targets (like Node Exporter, cAdvisor, app exporters).
   - Stores metrics in its own time-series database (TSDB).
2. **Alertmanager (optional, separate binary)**
   - Handles alerts from Prometheus.
   - Routes them to Slack, Email, PagerDuty, etc.
3. **Exporters (separate, optional)**
   - **Node Exporter** → OS & hardware metrics.
   - **cAdvisor / kube-state-metrics** → Kubernetes & pod/container metrics.
   - Database exporters (MySQL, Postgres, Redis, etc.).
   - Custom exporters (written using Prometheus client libraries).

🏗 In Kubernetes (EKS, GKE, etc.)

- Prometheus usually runs as a **Pod** (via Helm chart or operator).
- Other components:
  - **Node Exporter** runs as a **DaemonSet** (one pod per node).
  - **kube-state-metrics** → collects Kubernetes object states (deployments, pods, nodes, etc.).
  - **cAdvisor** → container-level resource usage (sometimes built into kubelet).

✅ So in short:

- Installing Prometheus = **only Prometheus server**.
- You **must install Node Exporter separately** if you need node metrics.
- Prometheus ecosystem has multiple exporters for **nodes, pods, network, databases, and apps** — but they are not bundled automatically.

-----Prometheus Installation Via HELM

🎯 Prometheus via Helm

There are mainly **two Helm charts** people use:

1. **prometheus-community/prometheus**
   → Installs only the **core Prometheus stack**.
2. **prometheus-community/kube-prometheus-stack**
   (formerly maintained by CoreOS as kube-prometheus)
   → Installs **Prometheus + Alertmanager + Exporters + Grafana + CRDs**.

---

1. If you install prometheus-community/prometheus
This chart installs:

- **Prometheus Server** → runs as a Deployment/StatefulSet.
- **Alertmanager** → optional, enabled by default.
- **Prometheus Pushgateway** → optional, for push metrics.
- **ServiceMonitors/ConfigMaps** → for Prometheus config.

☞ It **does NOT** install Node Exporter or kube-state-metrics.
☞ You must install those separately if you want node/pod metrics.

2. If you install prometheus-community/kube-prometheus-stack
This chart installs a **full monitoring stack**:

- **Prometheus** (server, config, storage).
- **Alertmanager**.
- **Node Exporter** (as a **DaemonSet**, one per node).
- **kube-state-metrics** (for Kubernetes objects like pods, deployments, nodes).
- **Prometheus Operator** (simplifies managing Prometheus/Alertmanager via CRDs).
- **Grafana** (with prebuilt dashboards).
- **ServiceMonitors + PodMonitors CRDs** (for automatic scraping).

☞ This is the most common choice for Kubernetes clusters because it's **plug-and-play monitoring for nodes, pods, workloads, and cluster health**.

✅ **In short:**

- **prometheus chart** → Only Prometheus + (optional) Alertmanager + Pushgateway.
- Use **prometheus Helm chart** if you want a **minimal Prometheus setup** and plan to add exporters manually.
- **kube-prometheus-stack chart** → Full monitoring stack (Prometheus, Alertmanager, Node Exporter, kube-state-metrics, Grafana, etc.).
- Use **kube-prometheus-stack Helm chart** if you want a **ready-made monitoring solution** with Prometheus, Grafana, Node Exporter, and Kubernetes metrics out of the box.

📊 Prometheus Helm Charts Comparison

Feature / Component	**prometheus (lightweight)**	**kube-prometheus-stack (full stack)**
Prometheus Server	✅ Installed	✅ Installed
Alertmanager	✅ Optional (enabled by default)	✅ Installed
Pushgateway	✅ Optional	❌ Not included (install separately if needed)
Node Exporter (DaemonSet)	❌ Not included	✅ Installed (one per node)
kube-state-metrics	❌ Not included	✅ Installed (K8s object metrics)
Grafana	❌ Not included	✅ Installed (with dashboards)
Prometheus Operator	❌ Not included	✅ Installed (manages Prometheus/Alertmanager)
CRDs (ServiceMonitor, PodMonitor, PrometheusRule)	❌ Not included	✅ Installed
Preconfigured Dashboards	❌ None	✅ Many (nodes, pods, API server, etc.)
Use Case	Basic Prometheus setup (good if you want full control, lightweight)	Complete monitoring stack for Kubernetes (production-ready)

-----Node Exporter Work in Prometheus & Grafana

1. Node Exporter (Data Collector)

- **What it is**: Node Exporter is an agent (binary) installed on each Linux machine (EC2, VM, bare-metal).
- **What it does**: It collects **host-level metrics** such as:
  - CPU usage
  - Memory usage
  - Disk I/O
  - Filesystem usage
  - Network statistics
- **How it exposes data**:
  -Node Exporter runs an HTTP server (default port 9100).
  -If you visit http://<node_ip>:9100/metrics, you'll see metrics in **Prometheus text-based format** like:
- node_cpu_seconds_total{cpu="0",mode="user"} 12345.6
- node_memory_MemFree_bytes 2048000000

→ It does **not push** data anywhere, it just **exposes** metrics.

---

2. Prometheus (Scraper + TSDB)

- **What it is**: Prometheus is the **time-series database and monitoring system**.
- **How it collects**:
  - Prometheus is **pull-based**: it scrapes metrics endpoints (like Node Exporter).
  - In prometheus.yml config, you define scrape jobs:
    ```
    scrape_configs:
      - job_name: "node"
        static_configs:
          - targets: ["<node1_ip>:9100", "<node2_ip>:9100"]
    ```
  - Prometheus then requests metrics every scrape_interval (default: 15s).
- **What it stores**: Prometheus stores these metrics in its **time-series database** (TSDB).

---

3. Grafana (Visualization)

- **What it is**: Grafana is a **dashboard & visualization tool**.
- **How it gets data**:
  - Grafana **does not collect data itself**.
  - Instead, you configure Prometheus as a **data source** in Grafana.
- **How it works**:
  - When you create a panel, Grafana runs a **PromQL query** against Prometheus (e.g. rate(node_cpu_seconds_total{mode="user"}[5m])).
  - Prometheus returns the **time-series data**.
  - Grafana visualizes it in dashboards (graphs, gauges, tables, etc.).

🔄 **Full Flow**

1. **Node Exporter** collects system metrics → exposes them at :9100/metrics.
2. **Prometheus** scrapes Node Exporter regularly → stores data in its TSDB.
3. **Grafana** queries Prometheus → displays metrics in dashboards.

✅ **Important Note**:

- If you want **application-level metrics** (like requests/sec, errors), Node Exporter alone is not enough. You need **application exporters** (e.g. for MySQL, Redis, Nginx, or custom app using Prometheus client libraries).
- But the process is the same: **Exporter** → **Prometheus scrape** → **Grafana visualization**.
