Kubernetes Notes

Index

Sr. No	Topic
1.	KIND Cluster installation on Windows
2.	Manual Scheduling 1. nodeName 2. nodeSelector 3. nodeAffinity 4. Taints and Tolerations
3.	Resource Requests and Resource Limits
4.	Static Pods
5.	Labels and Selectors
6.	Rolling Update and Rollback
7.	ConfigMap
8.	Secret
9.	Init Containers
10.	HPA (Horizontal Pod Autoscaler) and VPA (Vertical Pod Autoscaler)
11.	OS Upgrades: kubectl drain, cordon and uncordon
12.	K8s Cluster Upgrade Process
13.	Backup and Restore Methods in K8s
14.	TLS in K8s
15.	K8s Certificate API
16.	Kubeconfig
17.	Authorization in K8s

Sr. No	Topic
18.	Cluster Role and Cluster Role Binding 1. Role Based Access Control (RBAC) 2. Attribute Based Access Control (ABAC) 3. Webhook Authorization
19.	Service Account, Role and Role Binding
20.	Security Context
21.	Volume and Mounting
22.	Persistent Volume (PV) and Persistent Volume Claim (PVC)
23.	Storage Classes
24.	Statefulsets
25.	Cluster Networking
26.	Ingress and Ingress Controller
27.	Gateway API
28.	Network Policies
29.	Custom Resource Definition
30.	CoreDNS
31.	Links 1. Setup K8s cluster using kubeadm, EKS 2. Helm setup 3. ALB and Ngnix Ingress Controller setup
32.	Deployment Strategies in K8s 1. Rolling Update 2. Recreate 3. Blue-Green Deployment 4. Canary Deployment 5. A/B Testing
33.	K8s Practice Task List

Kind Cluster Installation on Windows

Prerequisite:

- Docker should be installed.

1. **Install Kubectl on windows**

Run the following commands in powershell as administrator:

```
curl.exe -LO
"https://dl.k8s.io/release/v1.33.0/bin/windows/amd64/kubectl.exe"

curl.exe -LO
"https://dl.k8s.io/v1.33.0/bin/windows/amd64/kubectl.exe.sha256"

$(Get-FileHash -Algorithm SHA256 .\kubectl.exe).Hash -eq $(Get-Content
.\kubectl.exe.sha256)

kubectl version --client

kubectl version --client --output=yaml
```

2. **Install Kind (powershell as administrator)**

```
curl.exe -Lo kind-windows-amd64.exe
https://kind.sigs.k8s.io/dl/v0.29.0/kind-windows-amd64

Move-Item .\kind-windows-amd64.exe c:\Kind\kind.exe
```

3. **Add `C:\Kind\` to your System PATH**

Press `Win + S`, search for "Environment Variables" and open it.
Click "Environment Variables…" at the bottom.
Under System variables, find `Path` and click Edit.
Click New and add:

```
C:\Kind\
```

Click OK → OK → OK.

Restart PowerShell or CMD to apply changes.
Test:

```
kind version
```

4. **Create a Kind Cluster with 1 Control Plane + 1 Worker**

Create a config file (`kind-config.yaml`):

```
notepad kind-config.yaml
```

Save below content to kind-config.yaml

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
  - role: worker
```

Create Cluster

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

Verify nodes

```
kubectl get nodes
```

```
NAME                 STATUS   ROLES           AGE   VERSION
kind-control-plane   Ready    control-plane   14m   v1.33.1
kind-worker          Ready    <none>          14m   v1.33.1
```

Manual Scheduling

https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

1. Using `nodeName` (Direct Node Assignment)

You can manually assign a pod to a specific node by specifying the `nodeName` field in the pod definition.

Example: Schedule a Pod on a Specific Node

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: manual-schedule-pod
spec:
  nodeName: kind-worker   # Specify the exact node name
  containers:
  - name: nginx-container
    image: nginx
```

```
NAME                  READY   STATUS    RESTARTS   AGE   IP           NODE          NOMINATED NODE   READINESS GATES
manual-schedule-pod   1/1     Running   0          22s   10.244.1.3   kind-worker   <none>           <none>
```

The pod gets scheduled on the kind-worker node.

Pros:

- Simple and direct assignment.
- No scheduler involvement.

Cons:

- If the node is unavailable, the pod will remain in a Pending state.
- No flexibility for high availability.

2. Using Node Selector (`nodeSelector`)

Instead of specifying an exact node, you can **match labels** on nodes using `nodeSelector`.

Example: Assign Pod to a Node with a Specific Label

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-nodeselector
spec:
  nodeSelector:
    env: production   # Matches nodes with label "env=production"
  containers:
```

```yaml
  - name: nginx
    image: nginx
```

Pros:

- More flexible than `nodeName`.
- Can be used with multiple nodes that share the same label.

Cons:

- If no nodes match the label, the pod stays in **Pending**.
- No advanced scheduling logic (e.g., weight, priority).

3. Using Node Affinity (Preferred Scheduling)

`nodeAffinity` provides more powerful scheduling than `nodeSelector`, allowing **soft (preferred) and hard (required) constraints**. It helps in assigning pods **to specific nodes based on labels**.

Example: Assign Pod to a Node with Affinity

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: env
            operator: In
            values:
            - production
  containers:
  - name: nginx
    image: nginx
```

Pros:

- More flexible than `nodeSelector`.
- Supports **multiple matching conditions**.

Cons:

- More complex than `nodeSelector`.

4. Using Taints and Tolerations

You can mark a node as **"tainted"**, so only pods with a matching **toleration** can run on it.

Example: Taint a Node

```
kubectl taint nodes worker-node-1 key=value:NoSchedule

kubectl taint nodes kind-worker server=database:NoSchedule
```

```
node/kind-worker tainted
```

```
kubectl describe node kind-worder
```

```
Taints:             server=database:NoSchedule
Unschedulable:      false
```

Now, only pods with this toleration can be scheduled on `kind-worker`.

Pod Definition with Toleration

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

You can check if the pod has applied the tolerance by describing it.
`kubectl describe pod pod-with-toleration`

```
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
                             server=database:NoSchedule
```

Pros:

- Ensures only specific workloads run on certain nodes.
- Good for **dedicated nodes** (e.g., GPU workloads).

Cons:

- Requires **manual taint management**.

Types of Taints:When to Use Node Affinity vs. Taints & Tolerations?

Scenario	Use Node Affinity	Use Taints & Tolerations
Assigning workloads to specific nodes	Yes	No
Avoiding scheduling on certain nodes	Yes	Yes
Dedicated infrastructure (GPU, DB nodes)	No	Yes
Soft preference for scheduling	Yes	No
Strict workload isolation	No	Yes

Can You Use Both Together?

Yes! You can use **Node Affinity for preferred placement** and **Taints & Tolerations for strict node restrictions**.

Example:

1. **Use Node Affinity** to **prefer** database workloads on nodes labeled `db-node`.
2. **Use Taints** to **ensure** only database pods can run on those nodes.

Resource Requests & Limits in Kubernetes

https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

In Kubernetes, **resource requests and limits** define how much CPU and memory a pod/container can use.

Requests → The minimum amount of CPU/memory guaranteed for a pod.
Limits → The maximum amount of CPU/memory a pod can use.

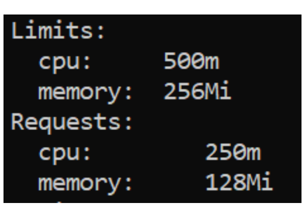
1. Defining Resource Requests & Limits

You can specify resource requests and limits in the pod definition under `resources`.

Example: Setting CPU & Memory Requests/Limits

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-limited-pod
spec:
  containers:
  - name: nginx-container
    image: nginx
    resources:
      requests:
        memory: "128Mi"  # Minimum memory reserved
        cpu: "250m"      # Minimum CPU reserved (0.25 CPU core)
      limits:
        memory: "256Mi"  # Maximum memory allowed
        cpu: "500m"      # Maximum CPU allowed (0.5 CPU core)
```

```
kubectl describe pods resource-limited-pod
```



```
Limits:
  cpu:     500m
  memory:  256Mi
Requests:
  cpu:        250m
  memory:     128Mi
```

2. What Happens When Resources Are Exceeded?

Scenario	Effect on Pod
CPU exceeds limit	The pod is **throttled** (slows down but does not crash).
Memory exceeds limit	The pod is **killed (OOMKilled)** and restarted.
CPU request is too low	The pod may not get enough CPU and run slowly.
Memory request is too low	The pod may run out of memory and crash.

Static Pods

https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/

Static Pods are **managed directly by the Kubelet** rather than the Kubernetes API Server. They are mainly used to run critical components (like `kube-apiserver`, `etcd`) on a node without relying on the control plane.

1. Key Features of Static Pods

Managed by the Kubelet (not the API server).
No replication → Each node runs its own instance.
Manifest files are stored locally on the node (`/etc/kubernetes/manifests/`).
Pods do not appear in `kubectl get deployments` since they are not managed by a controller.
Kubelet automatically restarts static pods if they fail.

2. How to Create a Static Pod

Step 1: Define the Static Pod Manifest

Create a YAML file (e.g., `/etc/kubernetes/manifests/static-pod.yaml`):

```
apiVersion: v1
kind: Pod
metadata:
  name: static-nginx
  labels:
    app: nginx
spec:
```

```
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Step 2: Place the File in the Static Pod Directory

Ensure the Kubelet is configured to check `/etc/kubernetes/manifests/` for static pods:

```
mkdir -p /etc/kubernetes/manifests
mv static-pod.yaml /etc/kubernetes/manifests/
```

Step 3: Verify Static Pod is Running

```
kubectl get pods --all-namespaces
```

Static Pods have a unique name format:

```
static-nginx-<node-name>
```

Where Are Static Pods Used?

Control Plane Components in Self-Managed Kubernetes Clusters

- **Kubernetes Control Plane (Kubeadm-based clusters)**
  - `kube-apiserver`
  - `kube-scheduler`
  - `kube-controller-manager`
  - `etcd`
- These are typically **deployed as Static Pods** on control plane nodes before Kubernetes is fully operational.

Labels and Selectors

https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

In Kubernetes, **labels** are key-value pairs assigned to objects (pods, services, deployments, etc.), and **selectors** are used to filter and manage resources based on those labels.

1. Labels in Kubernetes

Labels help categorize and organize Kubernetes resources.They are **immutable**, meaning once assigned, they **cannot be changed** (only removed and replaced).

Example: Adding Labels to a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: web
    env: production
spec:
  containers:
  - name: nginx
    image: nginx
```

Labels assigned:

- `app: web`
- `env: production`

2. Selectors in Kubernetes

Selectors allow Kubernetes to find and manage resources based on labels.
There are two types of selectors:

a) Equality-Based Selectors (`=`, `==`, `!=`)

Used for **exact matches**.

Example: Selecting Pods with `app=web`

```
kubectl get pods --selector app=web
```

Example: Selecting Pods NOT in production (`env != production`)

```
kubectl get pods --selector 'env!=production'
```

b) Set-Based Selectors (`in`, `notin`, `exists`)

Used for **matching multiple values**.

Example: Selecting Pods where `env` is `staging` or `production`

```
kubectl get pods --selector 'env in (staging, production)'
```

Example: Selecting Pods that have the `env` label (any value)

```
kubectl get pods --selector 'env'
```

3. Example: Using Labels & Selectors in Deployments

Step 1: Define a Deployment with Labels

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

Step 2: Create a Service Using Label Selector

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: web  # This will target all Pods with the label 'app=web'
```

```yaml
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

The **service** will route traffic only to pods that have `app: web`.

Rolling Update and Rollback

https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

Kubernetes **Rolling Update** ensures **zero downtime** when updating an application, while **Rollback** allows reverting to a previous version if something goes wrong.

1. Rolling Update in Kubernetes

A **Rolling Update** gradually replaces **old pods** with **new ones** while keeping the application **available**.

How Rolling Updates Work?

- Kubernetes updates pods **one at a time** instead of replacing them all at once.
- Uses the **ReplicaSet** and **Deployment** controller.
- Ensures that some instances of the application remain running.

Example: Deployment with Rolling Updates

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1    # Extra pod allowed during update
      maxUnavailable: 1  # Maximum unavailable pods during update
  selector:
    matchLabels:
      app: my-app
```

```yaml
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: nginx:1.20  # Initial version
```

Updating the Deployment

If we update the **image** (e.g., from `nginx:1.20` to `nginx:1.21`):

```
kubectl set image deployment/my-app my-container=nginx:1.21
```

- Kubernetes creates a **new pod** with `nginx:1.21`.
- Once it's running successfully, it **deletes one old pod** (`nginx:1.20`).
- This process continues **until all old pods are replaced**.

Checking Rollout Status

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

2. Rolling Back to a Previous Version

If the new version has issues, we can **rollback** to the previous version.

Check Revision History

```
kubectl rollout history deployment my-app
```

It shows something like:

```
REVISION  CHANGE-CAUSE
1         Initial deployment
2         Updated nginx:1.20 to nginx:1.21
```

Rollback to Previous Version

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

This reverts to the previous working version.

Rollback to a Specific Version

```
kubectl rollout undo deployment my-app --to-revision=1
```

Config Map

https://kubernetes.io/docs/concepts/configuration/configmap/

A **ConfigMap** in Kubernetes is an API object that allows you to store **configuration data** in key-value pairs. It helps **decouple configuration from application code**, making it easier to manage and modify settings without rebuilding container images.

Why Use ConfigMap?

Centralized management of configuration
Environment-specific configuration without modifying code
Avoid hardcoding values inside application containers
Can be used in multiple ways (environment variables, command arguments, or mounted files)

Creating a ConfigMap

There are multiple ways to create a ConfigMap:

1. Create from a YAML file

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_ENV: "production"
  DB_HOST: "mysql-service"
  DB_PORT: "3306"
```

Apply it:

```
kubectl apply -f configmap.yaml
```

2. Create from the command line

```
kubectl create configmap app-config --from-literal=APP_ENV=production
--from-literal=DB_HOST=mysql-service
```

Verify:

```
kubectl get configmap app-config -o yaml
```

Using a ConfigMap

You can use ConfigMap in **three** ways:

1. As Environment Variables

Modify your **Deployment**:

```yaml
containers:
  - name: my-app
    image: my-app-image
    env:
      - name: APP_ENV
        valueFrom:
          configMapKeyRef:
            name: app-config
            key: APP_ENV
```

2. As Command-Line Arguments

```yaml
containers:
  - name: my-app
    image: my-app-image
    command: ["my-app"]
    args: ["--db-host=$(DB_HOST)"]
    envFrom:
      - configMapRef:
          name: app-config
```

3. As a Mounted Volume

```yaml
volumes:
  - name: config-volume
    configMap:
      name: app-config
containers:
  - name: my-app
    image: my-app-image
    volumeMounts:
```

```
      - name: config-volume
        mountPath: "/etc/config"
```

This will create files inside `/etc/config/` with keys as filenames and values as content.

Secrets

https://kubernetes.io/docs/concepts/configuration/secret/

A **Secret** in Kubernetes is an object that stores **sensitive data**, such as **passwords, API keys, TLS certificates, and database credentials**. Unlike ConfigMaps, Secrets are **Base64-encoded** and stored more securely in `etcd`.

Why Use Secrets?

Security: Avoid storing sensitive data in plain text inside YAML files
Decoupling: Separate secrets from application code
Controlled Access: Restrict access using RBAC (Role-Based Access Control)
Multiple Usage Options: Can be used as **environment variables, mounted files, or command-line args**

Creating a Secret

There are multiple ways to create a Secret in Kubernetes.

1. Using YAML

Create a Secret in a YAML file:

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
data:
  DB_USER: YXBwLXVzZXI=    # Base64-encoded "app-user"
  DB_PASSWORD: YXBwLXBhc3M=  # Base64-encoded "app-pass"
```

Apply the Secret:

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

2. Using kubectl Command

```
kubectl create secret generic app-secret \
  --from-literal=DB_USER=app-user \
  --from-literal=DB_PASSWORD=app-pass
```

View the Secret:

```
kubectl get secrets app-secret -o yaml
```

(The output will show **Base64-encoded values**)

Using a Secret

You can use Secrets in **three** ways:

1. As Environment Variables

Modify your **Deployment**:

```
containers:
  - name: my-app
    image: my-app-image
    env:
      - name: DB_USER
        valueFrom:
          secretKeyRef:
            name: app-secret
            key: DB_USER
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: app-secret
            key: DB_PASSWORD
```

2. As a Mounted Volume

```
volumes:
  - name: secret-volume
    secret:
      secretName: app-secret
```

```yaml
containers:
  - name: my-app
    image: my-app-image
    volumeMounts:
      - name: secret-volume
        mountPath: "/etc/secrets"
```

This will create files inside `/etc/secrets/` with **DB_USER and DB_PASSWORD**.

3. As Command-Line Arguments

```yaml
containers:
  - name: my-app
    image: my-app-image
    command: ["my-app"]
    args: ["--db-user=$(DB_USER)"]
    envFrom:
      - secretRef:
          name: app-secret
```

Init Containers

https://kubernetes.io/docs/concepts/workloads/pods/init-containers/

An **Init Container** is a special type of container in Kubernetes that **runs before the main application container** starts. These containers perform **setup tasks** such as **waiting for dependencies, setting up configurations, or pulling external data**.

Why Use Init Containers?

- Run setup tasks before the main app starts
- Ensure dependencies are available (e.g., database is ready)
- Separate responsibilities (init tasks vs. main app logic)
- Improve security (run privileged operations separately)
- Retry mechanisms for handling failures

Example: Init Container in a Deployment

Here's an example of a **Pod with an Init Container** that waits for a MySQL database to be ready **before starting the main application**.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  initContainers:
  - name: init-check-db
    image: busybox
    command: ['sh', '-c', 'until nc -z mysql-service 3306; do echo
waiting for db; sleep 5; done;']
  containers:
  - name: main-app
    image: my-app-image
    ports:
    - containerPort: 8080
```

How Does It Work?

1. **Init Container (`init-check-db`)**
- Uses `busybox` to check if `mysql-service` is available on port `3306`.
- If the database is not available, it waits and retries every `5` seconds.
2. **Main Application Container (`main-app`)**
- Starts **only after the Init Container completes** successfully.

Multiple Init Containers

You can define **multiple Init Containers** in the order they should execute. Each Init Container **must complete successfully** before the next one starts.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-init-example
spec:
  initContainers:
  - name: setup-permissions
    image: busybox
    command: ['sh', '-c', 'chmod 777 /app']
  - name: download-config
```

```
    image: busybox
    command: ['sh', '-c', 'wget -O /app/config.json
https://example.com/config.json']
  containers:
  - name: main-app
    image: my-app-image
    ports:
    - containerPort: 8080
```

Execution Order

1. **First Init Container (`setup-permissions`)** → Sets up permissions
2. **Second Init Container (`download-config`)** → Downloads configuration
3. **Main Application (`main-app`)** starts only when **both Init Containers succeed**

HPA (Horizontal Pod Autoscaler) and VPA (Vertical Pod Autoscaler)

https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Kubernetes provides **Horizontal Pod Autoscaler (HPA)** and **Vertical Pod Autoscaler (VPA)** to dynamically adjust resources based on workload demand.

Horizontal Pod Autoscaler (HPA)

Scales the number of pods in a Deployment, ReplicaSet, or StatefulSet based on **CPU, memory, or custom metrics**.

How HPA Works?

- Monitors pod resource usage (e.g., **CPU, Memory**)
- If usage **exceeds a threshold**, HPA increases pod replicas.
- If usage **drops**, HPA reduces pod replicas.

Example: HPA Based on CPU Usage

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
spec:
  scaleTargetRef:
```

```yaml
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Explanation

- **Monitors CPU utilization** of `my-app` deployment.
- If CPU usage **exceeds 50%**, it increases replicas (up to `10`).
- If CPU usage **drops below 50%**, it reduces replicas (minimum `2`).

Enable Metrics Server for HPA

HPA requires a metrics server:

```
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/down
load/components.yaml
```

Check HPA status:

```
kubectl get hpa
```

Vertical Pod Autoscaler (VPA)

Automatically adjusts CPU & memory requests and limits for pods instead of changing replica count.

How VPA Works?

- Monitors **CPU & Memory usage** of running pods.
- Recommends or directly updates **resource requests** for pods.

- When resource changes are needed, **pods restart** with updated limits.

Example: VPA with Automatic Resource Adjustment

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

Explanation

- `Auto` **mode** updates pod resources automatically.
- `Off` **mode** only recommends but doesn't update resources.
- `Initial` **mode** sets resources **only at pod creation**.

Check VPA recommendations:

```
kubectl get vpa
```

HPA vs. VPA: Key Differences

Feature	HPA (Horizontal Scaling)	VPA (Vertical Scaling)
Scaling Type	Adjusts **replica count**	Adjusts **CPU & memory requests**
How Does It Works?	Adds/removes pods	Increases/decreases pod resources
Use Case	Handle variable traffic	Optimize resource utilization
Effect on Pods	No pod restart needed	Restarts pods to apply changes
Best for	Stateless apps, APIs, microservices	Stateful apps, batch processing

Can HPA and VPA Work Together?

Not directly! HPA scales pods **horizontally**, while VPA changes **resource requests**.
Solution: Use HPA **for scaling pods** and VPA in **recommendation mode** (not auto-updating).

When to Use What?

HPA → When **traffic fluctuates**, and you need to scale pods dynamically.
VPA → When an app **consumes high CPU/memory**, and you want efficient resource allocation.
Both → When using HPA for **replica scaling** and VPA in **recommendation mode**.

OS Upgrades: kubectl drain, cordon, and uncordon

https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/

These commands are used for **managing node availability** during maintenance, upgrades, or troubleshooting.

`kubectl cordon`

Marks a node as unschedulable (new pods won't be scheduled, but existing pods continue to run).
Does not evict existing pods.

Example

```
kubectl cordon node-name
```

Explanation

- The node is **marked as unschedulable**.
- New pods **will not be scheduled** on this node.
- Running pods **stay unaffected**.

Check the node status:

```
kubectl get nodes
```

Output:

```
NAME         STATUS                              ROLES    AGE    VERSION
```

```
node-name    Ready,SchedulingDisabled   worker   10d    v1.26.0
```

`kubectl drain`

Evicts all pods from a node and marks it as unschedulable.
Used before maintenance or node shutdown.

Example

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

Explanation

- **Evicts running pods** (except DaemonSet pods).
- **Removes EmptyDir volumes** (if `--delete-emptydir-data` is used).
- **Marks the node as unschedulable**.

`kubectl uncordon`

Makes a node schedulable again after it was cordoned or drained.

Example

```
kubectl uncordon node-name
```

Explanation

- **Allows new pods** to be scheduled on the node.
- **Restores normal operation** of the node.

Summary Table

Command	Effect
`kubectl cordon`	Prevents scheduling new pods but keeps existing pods running.

`kubectl drain`	Evicts all pods (except DaemonSets) and prevents new pod scheduling.
`kubectl uncordon`	Makes the node schedulable again.

Use Case:
Maintenance: Drain before shutting down a node.
Upgrades: Cordon to prevent scheduling, then drain to move pods.
Recovery: Uncordon to bring the node back into service.

Kubernetes Cluster Upgrade Process

https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/

Upgrading a Kubernetes cluster is crucial to ensure security, performance, and compatibility with new features. Below is a step-by-step guide to upgrading a Kubernetes cluster.

Step 1: Check the Current Kubernetes Version

Run the following command to check the cluster version:

```
kubectl version --short
```

Check the node versions:

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

Step 2: Check Available Versions

If using **kubeadm**, list available versions:

```
apt update && apt-cache madison kubeadm
```

For **yum (RHEL-based)**:

```
yum list --showduplicates kubeadm
```

Step 3: Upgrade `kubeadm`

Upgrade **kubeadm** to the desired version:

```
sudo apt-get install -y kubeadm=<desired-version>
```

Verify the upgrade:

```
kubeadm version
```

Step 4: Plan the Upgrade

```
sudo kubeadm upgrade plan
```

This will show the recommended versions for **control plane** and **worker nodes**.

Step 5: Upgrade the Control Plane

Run the upgrade:

```
sudo kubeadm upgrade apply <desired-version>
```

Verify that the **control plane** components are upgraded:

```
kubectl get pods -n kube-system
```

Step 6: Upgrade Kubelet & Kubectl

After upgrading `kubeadm`, upgrade `kubelet` and `kubectl`:

```
sudo apt-get install -y kubelet=<desired-version>
kubectl=<desired-version>
```

Restart `kubelet`:

```
sudo systemctl restart kubelet
```

Step 7: Upgrade Worker Nodes

Perform the following steps **for each worker node**:

1. **Drain the Node** (move workloads to other nodes):

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

2. **SSH into the Node** and upgrade `kubeadm`:

```
sudo apt-get install -y kubeadm=<desired-version>
```

3. **Perform the Upgrade**:

```
sudo kubeadm upgrade node
```

4. **Upgrade `kubelet` and `kubectl`**:

```
sudo apt-get install -y kubelet=<desired-version>
kubectl=<desired-version>

sudo systemctl restart kubelet
```

5. **Uncordon the Node** (bring it back online):

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

Step 8: Verify the Upgrade

Check all nodes:

```
kubectl get nodes
```

Ensure all nodes are **Ready** and running the new version.

Best Practices

- Upgrade one **minor version at a time** (e.g., `1.25 -> 1.26`).
- Always **drain worker nodes** before upgrading.
- Use **staging environments** before upgrading production.
- **Backup etcd** if using a self-managed control plane.
- Verify that workloads are running smoothly after the upgrade.

Backup and Restore Methods in Kubernetes

https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/

Backing up and restoring a Kubernetes cluster is critical for disaster recovery, migration, and maintaining business continuity. Here are the key methods:

1. **Backing Up and Restoring ETCD (for Cluster Configuration & State)**

ETCD is the **key-value store** that holds the entire Kubernetes cluster state, including deployments, services, and configurations.

Backup ETCD

If you have direct access to the control plane, you can back up ETCD with:

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

This creates a snapshot of ETCD.

Restore ETCD

To restore the cluster from a snapshot:

```
ETCDCTL_API=3 etcdctl snapshot restore /backup/etcd-snapshot.db \

  --data-dir=/var/lib/etcd
```

Then restart the ETCD service.

Best for: Full cluster recovery

2. **Using Velero (for Application-Level Backup & Restore)**

Velero is a popular open-source tool for backing up and restoring Kubernetes workloads.

Install Velero

```
velero install --provider aws --bucket <your-bucket> --secret-file
./credentials-velero --use-restic
```

Backup Resources

```
velero backup create my-backup --include-namespaces=my-namespace
```

Restore Resources

```
velero restore create --from-backup my-backup
```

Best for: Backing up deployments, services, secrets, and persistent volumes.

3. Manually Export YAML Files

If you just need to back up and restore Kubernetes resources (excluding persistent data), you can export YAML files.

Backup Resources

```
kubectl get all --all-namespaces -o yaml > cluster-backup.yaml
```

Restore Resources

```
kubectl apply -f cluster-backup.yaml
```

Best for: Configuration backup, quick recovery of resources.

4. Backup and Restore Persistent Volumes (PV)

Kubernetes Persistent Volumes store application data. To back up PVs:

Backup PV Data

If using **AWS EBS**, **GCP PD**, or **Azure Disks**, create snapshots:

```
aws ec2 create-snapshot --volume-id vol-xxxxxxxxxxxx
```

If using **NFS** or local storage:

```
rsync -av /mnt/pv-data/ /backup/pv-data/
```

Restore PV Data

Restore from snapshots or copy back the files:

```
rsync -av /backup/pv-data/ /mnt/pv-data/
```

Best for: Persistent data recovery in stateful application

Which Backup Method Do You Need?

- **Full cluster recovery?** → Use ETCD backup.
- **Application-level recovery?** → Use Velero.

- **Configuration backup?** → Export YAML files.
- **Persistent data backup?** → Use snapshots or rsync.

Security

TLS in Kubernetes (K8s)

https://kubernetes.io/docs/tasks/tls/certificate-issue-client-csr/
https://kubernetes.io/docs/tasks/tls/managing-tls-in-a-cluster/

What is TLS?

Transport Layer Security (TLS) is a cryptographic protocol that **secures communication** between different components in Kubernetes, such as:

- **Kubernetes API Server ↔ etcd**
- **Kubernetes API Server ↔ Kubelet**
- **Pods ↔ Services** (via Ingress with TLS termination)
- **Users ↔ Kubernetes API Server**

Why is TLS Important in Kubernetes?

- **Ensures secure communication** by encrypting data.
- **Verifies identity** of clients and servers.
- **Prevents Man-in-the-Middle (MITM) attacks**.
- **Mandatory for Kubernetes control plane components** (API server, Kubelet, etcd).

Key TLS Components in Kubernetes

Component	Description
CA (Certificate Authority)	Signs and verifies TLS certificates.
Certificate (.crt file)	Used for authentication of Kubernetes components.

Private Key (.key file)	Used to prove ownership of a certificate.
kube-apiserver	Uses TLS to secure communication with etcd, Kubelet, and users.
etcd	Uses TLS to ensure encrypted access from API server.
Kubelet	Uses TLS to authenticate and encrypt API server communication.
Ingress Controller	Uses TLS for HTTPS traffic termination.

Kubernetes TLS Certificates and Their Roles

File	Purpose	Used By
`etcd/ca.crt`	CA certificate for etcd authentication	API server
`apiserver-etcd-client.crt`	API server's client certificate to authenticate with etcd	API server
`apiserver-etcd-client.key`	Private key for API server authentication with etcd	API server
`apiserver.crt`	API server's certificate for serving requests	API server
`apiserver.key`	Private key for the API server	API server

`ca.crt`	Root CA certificate for signing TLS certs	Used cluster-wide
`front-proxy-client.crt`	Client cert for API aggregation layer	API server
`kubelet.crt`	TLS certificate for Kubelet authentication	Kubelet
`kubelet.key`	Private key for Kubelet authentication	Kubelet

How TLS Works in Kubernetes

Control Plane TLS Flow (API Server ↔ etcd)

1. **kube-apiserver** connects to **etcd** using `etcd-ca.crt` to validate the etcd server.
2. **etcd** presents its certificate, signed by `etcd-ca.crt`.
3. **kube-apiserver** presents its **client certificate** (`apiserver-etcd-client.crt`) for authentication.
4. **etcd verifies** kube-apiserver's identity using the CA.
5. **Secure communication is established**.

Worker Node TLS Flow (API Server ↔ Kubelet)

1. **kube-apiserver** connects to **Kubelet** for scheduling and logs.
2. **Kubelet presents its certificate (`kubelet.crt`)**, signed by CA.
3. **API server verifies the certificate** and communicates securely.

Ingress TLS Flow (User ↔ Service)

1. User accesses the service via HTTPS.

2. **Ingress Controller presents a TLS certificate**.
3. The user's browser **validates the certificate using CA**.
4. A **secure HTTPS connection** is established.

How to Generate TLS Certificates in Kubernetes

Kubernetes uses **`kubeadm`** or **manual OpenSSL commands** to generate certificates.

Using kubeadm to Generate Certificates

```
kubeadm certs generate-csr --config
/etc/kubernetes/kubeadm-config.yaml
```

This will generate:

- `apiserver.crt`
- `apiserver.key`
- `etcd/ca.crt`
- `apiserver-etcd-client.crt`
- `apiserver-etcd-client.key`

TLS Secrets for Ingress

To enable TLS in Kubernetes services, create a **TLS Secret**:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-tls-secret
type: kubernetes.io/tls
data:
  tls.crt: <base64-encoded cert>
  tls.key: <base64-encoded key>
```

Apply the secret:

```
kubectl apply -f my-tls-secret.yaml
```

Use it in an **Ingress resource**:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  tls:
  - hosts:
    - myapp.example.com
    secretName: my-tls-secret
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              number: 443
```

Debugging TLS Issues

Check Certificates

```
openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout
```

Check if TLS Secret Exists

```
kubectl get secret my-tls-secret -o yaml
```

Check if API Server is Using TLS

```
kubectl describe pod kube-apiserver -n kube-system | grep tls
```

Kubernetes Certificates API

https://kubernetes.io/docs/reference/access-authn-authz/certificate-signing-requests/

What is the Certificates API in Kubernetes?

The **Certificates API** is a built-in Kubernetes API used for **requesting, signing, and managing TLS certificates**. It helps **automate certificate management** within the cluster and ensures **secure communication** between Kubernetes components.

Why Use the Kubernetes Certificates API?

- **Automates certificate signing** for Kubernetes components.
- **Ensures secure TLS communication** between API server, Kubelet, and other cluster resources.
- **Integrates with Certificate Authorities (CA)** to approve or reject requests.
- **Used by kubelet for TLS bootstrapping** when a node joins a cluster.

How the Certificates API Works

Workflow of Certificate Signing

1. A Kubernetes component (e.g., Kubelet) generates a **Certificate Signing Request (CSR)**.
2. The CSR is submitted to the Kubernetes **Certificates API**.
3. An **admin manually approves** or an automated controller signs the request.
4. The Certificates API issues a **signed certificate**.
5. The requester **retrieves and uses the certificate** for secure communication.

Certificate Signing Request (CSR)

A CSR is a request for a **new TLS certificate** in Kubernetes.

Example CSR YAML File

```yaml
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: my-kubelet-csr
spec:
  request: <BASE64_ENCODED_CSR>
  signerName: kubernetes.io/kube-apiserver-client
  usages:
```

```
- digital signature
- key encipherment
- client auth
```

Key fields in CSR:

- **`request`**: The actual certificate request (Base64-encoded).
- **`signerName`**: Specifies the signer (e.g., API server).
- **`usages`**: Defines how the certificate will be used (client authentication, server authentication, etc.).

Managing CSRs in Kubernetes

List Pending CSRs

```
kubectl get csr
```

Approve a CSR Manually

```
kubectl certificate approve my-kubelet-csr
```

Deny a CSR

```
kubectl certificate deny my-kubelet-csr
```

View CSR Details

```
kubectl get csr my-kubelet-csr -o yaml
```

Fetch the Signed Certificate

Once approved, download the signed certificate:

```
kubectl get csr my-kubelet-csr -o jsonpath='{.status.certificate}' |
base64 -d > kubelet.crt
```

Who Uses the Certificates API?

Component	Purpose

Kubelet	Uses CSRs for TLS bootstrapping when joining a cluster.
Ingress Controllers	Request TLS certificates for HTTPS traffic.
API Server	Uses client certificates for authentication.
Mutual TLS (mTLS) Apps	Request certificates for secure communication between services.

Automatic vs. Manual Certificate Approval

- **Manual Approval:** Admins must approve or deny CSRs.
- **Automatic Approval:** Kubernetes controllers or external tools (e.g., cert-manager) automatically sign CSRs.

Enable automatic signing using cert-manager:

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: my-ca-issuer
spec:
  ca:
    secretName: my-ca-secret
```

Debugging Certificates API Issues

Check Pending CSRs

```
kubectl get csr
```

Inspect Logs for Issues

```
kubectl logs -n kube-system kube-controller-manager
```

Verify Signed Certificate

```
openssl x509 -in kubelet.crt -text -noout
```

Kubeconfig in Kubernetes

https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/
https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/

What is Kubeconfig?

- **Kubeconfig** is a configuration file that stores cluster access information for `kubectl` and other Kubernetes clients.
- It defines **clusters, users, contexts, and authentication mechanisms** for connecting to a Kubernetes cluster.
- Default location: `~/.kube/config` (Linux/macOS) or `%USERPROFILE%\.kube\config` (Windows).

Key Components of a Kubeconfig File

A **kubeconfig** file consists of:

- **Clusters**: Defines Kubernetes API server endpoints.
- **Users**: Specifies authentication credentials.
- **Contexts**: Links a cluster with a user.
- **Current-context**: Specifies the default cluster and user.

Sample Kubeconfig File

```yaml
apiVersion: v1
kind: Config
clusters:
- name: my-cluster
  cluster:
    server: https://192.168.1.100:6443
    certificate-authority: /path/to/ca.crt
users:
- name: admin
  user:
```

```yaml
    client-certificate: /path/to/client.crt
    client-key: /path/to/client.key
contexts:
- name: my-context
  context:
    cluster: my-cluster
    user: admin
current-context: my-context
```

Kubeconfig Structure Explained

Field	Description
`clusters`	Defines the Kubernetes API server address and CA certificate.
`users`	Contains authentication credentials (certificates, tokens, or passwords).
`contexts`	Binds a cluster with a user for easy switching.
`current-context`	Specifies the active context used by `kubectl`.

Managing Kubeconfig

View Current Context

```
kubectl config current-context
```

List All Contexts

```
kubectl config get-contexts
```

Switch Contexts

```
kubectl config use-context my-context
```

Set a New Cluster

```
kubectl config set-cluster my-cluster \

  --server=https://192.168.1.100:6443 \

  --certificate-authority=/path/to/ca.crt
```

Set a New User

```
kubectl config set-credentials admin \

  --client-certificate=/path/to/client.crt \

  --client-key=/path/to/client.key
```

Set a New Context

```
kubectl config set-context my-context \

  --cluster=my-cluster --user=admin
```

Authentication Methods in Kubeconfig

- **Certificate-based Authentication**: Uses `client-certificate` and `client-key`.
- **Token-based Authentication**: Uses a Bearer Token.
- **Basic Authentication**: Uses a username and password (not recommended).
- **OIDC Authentication**: Uses OpenID Connect for external authentication providers.

Example Token Authentication

```
users:
- name: admin
  user:
    token: "your-token-here"
```

Merging Multiple Kubeconfig Files

If you have multiple clusters, you can merge configurations using:

```
export KUBECONFIG=/path/to/config1:/path/to/config2
```

```
kubectl config view --merge --flatten
```

Troubleshooting Kubeconfig Issues

Check the Current Configuration

```
kubectl config view
```

Verify Authentication Issues

```
kubectl get nodes
```

If unauthorized, check token or certificate permissions.

Debug API Server Connection

```
kubectl cluster-info
```

Authorization

https://kubernetes.io/docs/reference/access-authn-authz/authorization/
https://kubernetes.io/docs/reference/access-authn-authz/rbac/
https://kubernetes.io/docs/reference/access-authn-authz/abac/
https://kubernetes.io/docs/reference/access-authn-authz/node/
https://kubernetes.io/docs/reference/access-authn-authz/webhook/

What is Authorization in Kubernetes?

- **Authorization** determines **what actions a user or service account can perform** in the cluster after authentication.
- Once a user is authenticated, Kubernetes checks **authorization policies** to decide if the request should be allowed or denied.

Authorization Modes in Kubernetes

Kubernetes supports multiple **authorization modes**:

Mode	Description

RBAC (Role-Based Access Control)	Uses roles and role bindings to control access.
ABAC (Attribute-Based Access Control)	Uses policies defined in a file to grant permissions.
Webhook Authorization	Calls an external service for authorization decisions.
Node Authorization	Grants permissions specifically to kubelet nodes.
AlwaysAllow (deprecated)	Allows all requests. Not recommended.
AlwaysDeny (deprecated)	Denies all requests.

1. Role-Based Access Control (RBAC) – Most Common Mode

RBAC is the default and most widely used authorization mode in Kubernetes. It allows you to define permissions for users, groups, and service accounts at both the **namespace level** and the **cluster level**.

How RBAC Works

RBAC consists of four key components:

1. **Role** – Defines permissions within a namespace.
2. **ClusterRole** – Defines permissions for the entire cluster.
3. **RoleBinding** – Grants a Role to a user, group, or service account within a namespace.
4. **ClusterRoleBinding** – Grants a ClusterRole to a user, group, or service account across the cluster.

Example: Read-Only Access to Pods in the "default" Namespace

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
```

```yaml
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

Binding the Role to a User

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

Now, `dev-user` can only read pods in the `default` namespace.

2. Attribute-Based Access Control (ABAC) – Deprecated

ABAC allows **fine-grained, policy-based authorization** using JSON or YAML policy files.

How ABAC Works

- You define an **authorization policy file**.
- Kubernetes checks the request **against the policy**.
- If the policy allows it, the request is granted.

Example ABAC Policy File

```json
{
  "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
```

```
  "kind": "Policy",
  "spec": {
    "user": "dev-user",
    "namespace": "default",
    "resource": "pods",
    "readonly": true
  }
}
```

This allows `dev-user` to read pods in the `default` namespace.

How to Enable ABAC?

- Start the API server with `--authorization-mode=ABAC`.
- Provide the policy file using `--authorization-policy-file=/path/to/policy.json`.

ABAC is considered outdated and is no longer recommended. Use RBAC instead.

3. Webhook Authorization – External Policy Engine

Webhook authorization allows **external services** to make authorization decisions.

How Webhook Authorization Works

1. Kubernetes **sends an HTTP request** to an external **authorization server**.
2. The external server **evaluates the request** and returns an `allow` or `deny` response.
3. The API server allows or denies the request based on the response.

Example Webhook Configuration

```
apiVersion: apiserver.config.k8s.io/v1
kind: Webhook
metadata:
  name: auth-webhook
webhook:
  url: "https://auth.example.com/validate"
```

Kubernetes will send authorization requests to `https://auth.example.com/validate`.

Example Webhook Response

```
{
  "apiVersion": "authorization.k8s.io/v1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": true
  }
}
```

The request is allowed if `allowed: true`.

When to Use Webhook Authorization?

- When you need to integrate with **custom authentication systems**.
- When using **external policy engines** like **Open Policy Agent (OPA)**.

4. Node Authorization – For Kubelets

Node Authorization is a special authorization mode **for kubelet nodes**. It ensures that a node can only access resources related to the workloads scheduled on it.

How Node Authorization Works

1. Nodes authenticate using **certificates issued by the Kubernetes CA**.
2. Kubernetes **checks if the node is authorized** to perform the requested action.
3. Nodes are allowed only to:
    - Read **pods assigned to them**.
    - Read **configmaps and secrets referenced by their pods**.
    - Read/write **endpoints for services their pods use**.

How to Enable Node Authorization?

Start the API server with:

```
--authorization-mode=Node
```

Example of Allowed Request for a Node

A node (`node-1`) can **get** the details of a pod running on it:

```
{
  "apiVersion": "authorization.k8s.io/v1",
  "kind": "SubjectAccessReview",
  "spec": {
    "user": "system:node:node-1",
    "resource": "pods",
    "verb": "get",
    "namespace": "default"
  }
}
```

Node Authorization is enabled by default and is critical for security!

5. AlwaysAllow (Deprecated) & AlwaysDeny

These are simple authorization modes used for **testing purposes**.

AlwaysAllow

- **Allows all requests.**
- Start API server with `--authorization-mode=AlwaysAllow`.
- **Not recommended for production!**

AlwaysDeny

- **Denies all requests.**
- Start API server with `--authorization-mode=AlwaysDeny`.
- Useful for debugging.

Enabling Multiple Authorization Modes

You can enable multiple authorization modes **at the same time**:

```
--authorization-mode=RBAC,Node,Webhook
```

Kubernetes will check each mode in order and allow the request if any mode grants permission.

Cluster Role and Cluster Role Binding

https://kubernetes.io/docs/reference/access-authn-authz/rbac/

What is a ClusterRole?

A **ClusterRole** is a **Kubernetes Role** that defines permissions **at the cluster level** rather than within a specific namespace. It grants access to **cluster-wide resources** like:

- Nodes
- Namespaces
- Persistent Volumes
- Cluster-wide API groups

What is a ClusterRoleBinding?

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

Difference Between Role & ClusterRole

Feature	Role	ClusterRole
Scope	Namespace-specific	Cluster-wide
Can manage Namespaces?	No	Yes
Can manage Nodes?	No	Yes
Can manage Persistent Volumes?	No	Yes
Can manage ClusterRoles?	No	Yes

Example: Read-Only Access to All Pods in the Cluster

Step 1: Define a ClusterRole

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

This ClusterRole allows reading all Pods in the entire cluster.

Step 2: Bind the ClusterRole to a User

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-pod-reader-binding
subjects:
- kind: User
  name: dev-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Now, `dev-user` can read all pods across all namespaces.

When to Use ClusterRole vs Role?

Use Case	Use Role	Use ClusterRole
Grant access to a namespace	Yes	No

Grant access to all namespaces	No	Yes
Manage Nodes, Namespaces, PersistentVolumes	No	Yes
Grant admin access across cluster	No	Yes

Common ClusterRole Examples

- Cluster-wide Read-Only Access
- Cluster-wide Admin Access
- Allow ServiceAccounts to Access Resources

Service Account

https://kubernetes.io/docs/concepts/security/service-accounts/

What is a Service Account?

A **Service Account (SA)** is a special type of Kubernetes account used by **Pods** to authenticate with the Kubernetes API. Unlike user accounts, service accounts are managed within Kubernetes and do not require passwords.

Key Features:
Used by Pods to interact with the Kubernetes API.
Automatically created in each namespace (`default` SA).
Can be associated with specific RBAC roles for access control.

Why Do We Need Service Accounts?

By default, Pods run with the **default service account** in their namespace. However, in scenarios where different Pods need different levels of access to cluster resources, we create custom Service Accounts and bind them with specific permissions.

Example Use Cases:
A Pod needs to list all other Pods.
A monitoring system (e.g., Prometheus) needs access to API metrics.
A Pod needs access to ConfigMaps or Secrets.

Creating a Custom Service Account

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

This creates a new Service Account named `my-service-account` in the `default` namespace.

Binding a Service Account to a Role (RBAC Example)

To grant permissions, we need to bind the Service Account to a **Role** or **ClusterRole** using a **RoleBinding** or **ClusterRoleBinding**.

Creating a Role for Pod Read Access

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

Creating a RoleBinding for the Service Account

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

```
  apiGroup: rbac.authorization.k8s.io
```

Now, `my-service-account` can list and read all Pods in the `default` namespace.

Assigning a Service Account to a Pod

To use the Service Account in a Pod, define it in the Pod spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  namespace: default
spec:
  serviceAccountName: my-service-account
  containers:
  - name: my-container
    image: busybox
```

Now, `my-pod` will use `my-service-account` when accessing the Kubernetes API.

Viewing Service Accounts & Tokens

List all Service Accounts in a namespace:

```
kubectl get serviceaccounts -n default
```

Describe a Service Account:

```
kubectl describe serviceaccount my-service-account -n default
```

Get the associated token for authentication:

```
kubectl get secret $(kubectl get sa my-service-account -o
jsonpath='{.secrets[0].name}') -o jsonpath='{.data.token}' | base64
--decode
```

This token is used for authenticating API requests from the Pod.

Default vs Custom Service Accounts

Feature	Default Service Account	Custom Service Account
Automatically created?	Yes	No
Permissions	Minimal (varies by cluster)	Defined by RoleBinding
Assigned to Pods by default?	Yes	No (must be explicitly set)
Can be deleted?	No	Yes

When to Use Service Accounts?

When a Pod needs to interact with the Kubernetes API.
When restricting API access for specific workloads.
When integrating Kubernetes with external services (e.g., monitoring, logging).

Security Context in Kubernetes

https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

A **Security Context** in Kubernetes defines security settings for a **Pod** or a **Container**. It helps enforce security policies like running as a non-root user, restricting privilege escalation, and setting file system permissions.

Setting Security Context in Kubernetes

A **SecurityContext** can be applied at two levels:
Pod Level → Affects all containers in the Pod.
Container Level → Affects only a specific container.

Example: Security Context at Pod Level

The following Pod runs as a **non-root user** (`1000`) and prevents privilege escalation.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  securityContext:
    runAsUser: 1000   # Run as user ID 1000
    runAsGroup: 3000  # Run as group ID 3000
    fsGroup: 2000     # Group ownership of mounted volumes
  containers:
  - name: secure-container
    image: nginx
    securityContext:
      allowPrivilegeEscalation: false
```

Key Points:

- `runAsUser`: Ensures the container does not run as root.
- `runAsGroup`: Assigns a group ID to the container process.
- `fsGroup`: Changes the group ownership of mounted volumes.
- `allowPrivilegeEscalation: false`: Prevents privilege escalation (e.g., using `sudo`).

Example: Security Context at Container Level

You can specify security settings for **individual containers** inside a Pod.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secure-container-pod
spec:
  containers:
  - name: secure-container
    image: busybox
    command: ["sleep", "3600"]
```

```yaml
  securityContext:
    runAsUser: 2000
    capabilities:
      add: ["NET_ADMIN"]   # Add capability for network management
      drop: ["ALL"]        # Drop all other capabilities
```

Key Points:

- `capabilities.add`: Adds specific Linux capabilities (e.g., `NET_ADMIN` for network configuration).
- `capabilities.drop`: Drops all other unnecessary capabilities to minimize security risks.

Important Security Context Fields

Field	Description
`runAsUser`	Runs the container as a specific user ID (instead of root).
`runAsGroup`	Assigns a group ID to the container process.
`fsGroup`	Sets the group ID for mounted volumes.
`allowPrivilegeEscalation`	Prevents privilege escalation (e.g., `sudo`).
`privileged`	Allows running containers in **privileged mode** (dangerous).
`capabilities`	Grants or removes specific Linux capabilities.

`readOnlyRootFilesystem`	Makes the container file system read-only for security.
`seccompProfile`	Enforces system call restrictions (e.g., `RuntimeDefault`).

Example: Restricting Privileged Containers

Running a **privileged** container can be dangerous. Use **`privileged: false`** to restrict it.

```
apiVersion: v1
kind: Pod
metadata:
  name: non-privileged-pod
spec:
  containers:
  - name: app-container
    image: alpine
    command: ["sleep", "3600"]
    securityContext:
      privileged: false  # Prevents full system access
```

Why?

- Privileged containers **bypass** security restrictions and gain full access to the host system.
- Should only be used for system-level tools (e.g., network plugins).

Example: Enforcing Read-Only Filesystem

Prevents modifications to the container filesystem.

```
apiVersion: v1
kind: Pod
metadata:
  name: readonly-rootfs-pod
spec:
```

```yaml
  containers:
  - name: secure-app
    image: nginx
    securityContext:
      readOnlyRootFilesystem: true
```

Why?

- Blocks malware from modifying system files.
- Prevents unauthorized changes inside the container.

Volumes in Kubernetes

https://kubernetes.io/docs/concepts/storage/volumes/
https://kubernetes.io/docs/concepts/storage/persistent-volumes/

In Kubernetes, a **Volume** is a directory accessible to containers in a Pod, used to **persist data** across container restarts or share data between containers. Containers by default have an **ephemeral** filesystem—data is lost when the container stops. Volumes solve that.

Why Use Volumes?

- Data **persistence** beyond container lifetime
- **Sharing** data between containers in the same Pod
- **Storing config**, secrets, or logs
- **Mounting external storage** (e.g., NFS, EBS, PVCs)

Types of Volumes in Kubernetes

Volume Type	Use Case
`emptyDir`	Temporary scratch space shared between containers in a Pod

`hostPath`	Mounts a file/directory from the host node
`configMap`	Mounts a ConfigMap as files
`secret`	Mounts a Secret as files
`persistentVolumeClaim` (PVC)	Mounts a persistent volume for long-term storage
`nfs`	Mounts a shared NFS volume
`awsElasticBlockStore`	Mounts an AWS EBS volume
`gcePersistentDisk`	Mounts a GCP persistent disk
`projected`	Combines multiple volume sources (e.g., ConfigMap + Secret)
`ephemeral`	Inline volume lifecycle tied to the Pod

Example 1: `emptyDir` Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
```

```yaml
  - name: app
    image: busybox
    command: ["sleep", "3600"]
    volumeMounts:
    - mountPath: /tmp/data
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

Data inside `/tmp/data` will persist **as long as the Pod lives**.

Example 2: `hostPath` Volume

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-demo
spec:
  containers:
  - name: busybox
    image: busybox
    command: ["sleep", "3600"]
    volumeMounts:
    - name: myhostpath
      mountPath: /mnt/data
  volumes:
  - name: myhostpath
    hostPath:
      path: /data/host
      type: DirectoryOrCreate
```

Use with caution—**ties your Pod to a specific node**, reducing portability.

Example 3: Mounting a `secret`

```yaml
volumes:
- name: secret-volume
  secret:
```

```
    secretName: my-app-secret
```

Example 4: Using a `PersistentVolumeClaim`

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

Data persists even if the Pod is deleted or rescheduled.

Volume Lifecycle

- Pod starts → Kubernetes **attaches/mounts volume**
- Containers use volume paths defined in `volumeMounts`
- Pod ends → Volume (except PVC-based) is deleted

Best Practices

- Use `emptyDir` only for **scratch data** or **cache**
- Use PVCs for **stateful apps** (like MySQL, PostgreSQL)
- Use `secret` and `configMap` for injecting sensitive/config data
- Avoid `hostPath` in production unless absolutely needed

Persistent Volume (PV) & Persistent Volume Claim

https://kubernetes.io/docs/concepts/storage/persistent-volumes/

What is a Persistent Volume (PV)?

A **Persistent Volume** is a piece of **storage in the cluster** that has been provisioned by an admin or dynamically created using a StorageClass.

It's a **cluster resource**, like CPU or RAM, and it's **independent of the Pod lifecycle** — meaning it won't be deleted even if the Pod using it gets terminated.

Example: `PersistentVolume`

```
apiVersion: v1
```

```yaml
kind: PersistentVolume
metadata:
  name: app-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data/app
```

What is a Persistent Volume Claim (PVC)?

A **Persistent Volume Claim** is a request for storage by a user or a Pod. It describes:

- How much storage it needs
- What access mode (e.g., ReadWriteOnce)
- Optional: which StorageClass to use

Example: `PersistentVolumeClaim`

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: app-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  selector:
    matchLabels:
      vol: mysql
```

PVC binds to a matching PV that satisfies its request.

Using a PVC in a Pod

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

How It All Works Together

1. Admin creates a **PV** (or it is dynamically provisioned).
2. User creates a **PVC**.
3. Kubernetes matches the PVC to an available PV.
4. Pod uses the PVC to mount storage.

Access Modes

Mode	Description
`ReadWriteOnce`	One node can read/write
`ReadOnlyMany`	Multiple nodes can read only

`ReadWriteMany`	Multiple nodes can read/write

Storage Classes

You can use `storageClassName` to dynamically provision volumes using external provisioners like:

- AWS EBS
- GCE Persistent Disk
- NFS
- GlusterFS

Storage Classes

https://kubernetes.io/docs/concepts/storage/storage-classes/

What is a StorageClass in Kubernetes?

A **StorageClass** is a way to define **different types of storage** (like SSDs, HDDs, encrypted volumes, etc.) in a Kubernetes cluster.

It tells Kubernetes **how to provision a Persistent Volume dynamically** when a PVC (PersistentVolumeClaim) is created **with a specific `storageClassName`**.

Why Use StorageClass?

- Automates volume provisioning — **no need to manually create PVs**.
- Enables multiple storage options for different workloads.
- Provides advanced features like **encryption, replication, or snapshots**.

Key Fields of a StorageClass

Field	Description

`provisioner`	The plugin used to provision the storage (e.g., AWS EBS, GCE PD, etc.)
`parameters`	Custom settings for the provisioner (e.g., type, iops)
`reclaimPolicy`	What happens to the volume when PVC is deleted (`Retain` or `Delete`)
`volumeBindingMode`	When the volume is bound to a node (`Immediate` or `WaitForFirstConsumer`)

Example: StorageClass (AWS EBS)

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  encrypted: "true"
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

PVC Using This StorageClass

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-claim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: fast-ssd
  resources:
```

```yaml
    requests:
      storage: 5Gi
```

When this PVC is created, Kubernetes will **dynamically provision a PV using the `fast-ssd` StorageClass**.

Default StorageClass

If a StorageClass is marked as default:

```yaml
annotations:

  storageclass.kubernetes.io/is-default-class: "true"
```

PVCs **without** `storageClassName` will use this one automatically.

Statefulsets

https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

What is a StatefulSet?

A **StatefulSet** is a Kubernetes controller used to manage **stateful applications**. Unlike Deployments, StatefulSets are designed for applications that:

- Require **stable, persistent storage**.
- Need **unique network identities**.
- Must start and stop in **a specific order**.

Why Use StatefulSets?

Because some apps (like databases) need:

- Stable **volume mounts** across pod restarts.
- Predictable **pod names** (e.g., `mysql-0`, `mysql-1`).
- Ordered startup and shutdown (e.g., master starts before replicas).
- Stateful failover and recovery.

Examples: MySQL, PostgreSQL, Cassandra, Kafka, Zookeeper.

Key Features

Feature	StatefulSet Behavior
Persistent Volumes	Uses **PersistentVolumeClaim (PVC)** templates
Stable Pod Names	Pods are named like `app-0`, `app-1`, etc.
Ordered Deployment	Pods are started **sequentially**
Ordered Termination	Pods are **terminated in reverse order**
Pod Identity	Each pod gets its **own DNS name**

Basic Example: StatefulSet with PVC

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "web"
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
```

```yaml
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Behavior of the Above Example

- Creates 3 pods: `web-0`, `web-1`, `web-2`.
- Each pod gets its **own PersistentVolume**.
- Pods are created in order: `web-0` → `web-1` → `web-2`.
- Deletion also happens in reverse.

Important Notes

Requires a **Headless Service** to manage DNS.

```yaml
apiVersion: v1

kind: Service
metadata:
  name: web
spec:
  clusterIP: None
  selector:
```

```yaml
    app: web
  ports:
  - port: 80
```

You **cannot scale down and reuse** the PV of a deleted pod easily (you must delete manually unless `ReclaimPolicy` is `Delete`).

Cluster Networking

https://kubernetes.io/docs/concepts/cluster-administration/networking/

Cluster networking in Kubernetes is a **fundamental concept** that enables communication between different components in the cluster—**Pods, Services, Nodes**, and **external clients**. Here's a breakdown to help you understand it clearly.

1. What Is Cluster Networking?

Kubernetes networking enables:

- **Pod-to-Pod communication** (across nodes)
- **Pod-to-Service communication**
- **External-to-Service access**
- **Node-to-Pod communication**

2. Key Concepts

a. Pod Network

Each **Pod gets its own IP** address. This allows direct communication between Pods without NAT.

- Kubernetes expects all Pods to **communicate with each other freely** (flat network).
- Example: `pod-A` on node-1 can directly reach `pod-B` on node-2.

b. Node Network

Each **Node** has a unique IP, and **Pods use this IP for routing traffic** to Pods on other nodes.

c. Service Network

Services in Kubernetes get a **stable virtual IP** (ClusterIP) and **DNS name** via `kube-dns` or `CoreDNS`.

- The service routes to one of the backend Pods using **kube-proxy**.

d. Cluster DNS

- All Services are automatically assigned DNS names like:
  `my-service.my-namespace.svc.cluster.local`
- `CoreDNS` resolves these names inside the cluster.

3. Networking Components

Component	Role
CNI plugin	Manages Pod networking and IP assignment
kube-proxy	Handles Service routing via iptables/ipvs
CoreDNS	Resolves internal service and Pod DNS names

4. CNI (Container Network Interface)

Kubernetes uses **CNI plugins** to configure Pod networks.

Popular plugins:

- Calico
- Flannel
- Weave
- Cilium

Each plugin implements the Kubernetes networking model.

5. How Traffic Flows

Pod-to-Pod

- Traffic flows via CNI.
- Routed even if Pods are on different nodes.

Pod-to-Service

- DNS name is resolved via CoreDNS.
- kube-proxy forwards request to one of the Pod endpoints.

External-to-Service (Ingress/NodePort/LoadBalancer)

- External clients access the service using:
  - **NodePort**: `nodeIP:nodePort`
  - **LoadBalancer**: External IP via cloud provider
  - **Ingress**: Host-based or path-based routing

Real-World Example

Let's say we have:

- **frontend-pod** → wants to call → **backend-service**

1. `frontend-pod` uses DNS: `backend-service.default.svc.cluster.local`
2. DNS resolves to ClusterIP: `10.96.0.10`
3. kube-proxy on node routes request to one of backend Pods.

Network Policies (Optional)

To restrict traffic between Pods:

- Define **NetworkPolicy** to allow or deny traffic.

- Supported by most CNI plugins (e.g., Calico, Cilium).

Traffic Type	Supported by Default	How
Pod ↔ Pod	Yes	Via CNI
Pod ↔ Service	Yes	kube-proxy
External ↔ Service	Yes	NodePort / LoadBalancer / Ingress
Node ↔ Pod	Yes	Native routing

Ingress

https://kubernetes.io/docs/concepts/services-networking/ingress/
https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

What is Ingress in Kubernetes?

Ingress is a Kubernetes object that lets you **expose HTTP and HTTPS routes** from **outside the cluster** to services **inside the cluster**.

Instead of exposing each service separately using a `NodePort` or `LoadBalancer`, Ingress provides a **centralized entry point** to manage external access.

Why use Ingress?

- Centralized routing of traffic
- Route based on **hostnames** (e.g. `api.example.com`)
- Route based on **paths** (e.g. `/login`, `/products`)
- Support for **SSL/TLS termination**

- More control with **annotations** and custom rules

Key Components

1. **Ingress Resource**
    - A YAML configuration that defines routing rules
2. **Ingress Controller**
    - A pod running inside the cluster (like NGINX or Traefik)
    - It watches for Ingress resources and applies the routing rules
3. **Service**
    - The Kubernetes service that the Ingress routes to

Ingress Example

Ingress YAML

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /app1
        pathType: Prefix
        backend:
          service:
            name: app1-service
            port:
              number: 80
      - path: /app2
```

```yaml
          pathType: Prefix
          backend:
            service:
              name: app2-service
              port:
                number: 80
```

What it does:

- Requests to `myapp.example.com/app1` go to `app1-service`
- Requests to `myapp.example.com/app2` go to `app2-service`

How It Works (Flow):

Client (browser/curl)
 ↓
Ingress Controller (e.g., NGINX)
 ↓
Ingress Resource (routing logic)
 ↓
Kubernetes Service
 ↓
Pod

TLS with Ingress (Optional)

You can terminate HTTPS traffic using Ingress by specifying certificates:

```yaml
tls:
- hosts:
  - myapp.example.com
  secretName: tls-secret
```

You can generate this secret with your TLS certs or use tools like **cert-manager** with **Let's Encrypt**.

An **Ingress Controller** is a **Kubernetes component** responsible for **fulfilling Ingress resources** — i.e., it **routes external HTTP/HTTPS traffic** to the right services within your cluster based on the rules you define in an Ingress.

- **Ingress Resource**: *A set of traffic routing rules* (like `/api → backend`, `/ → frontend`).
- **Ingress Controller**: *The actual server (usually NGINX, HAProxy, Traefik, etc.) that applies those rules and handles the traffic.*

How it Works (High Level)

1. You deploy an **Ingress Controller** (e.g. `nginx-ingress-controller`) as a pod in your cluster.
2. You create an **Ingress Resource** with routing rules.
3. The Ingress Controller watches for Ingress Resources.
4. Based on those rules, it:
    - Listens for traffic on port **80/443**
    - Forwards requests to the correct **Service** → **Pod**

Common Ingress Controllers

Controller	Description
NGINX	Most widely used, solid performance
Traefik	Dynamic config, great for microservices
HAProxy	High-performance load balancer
AWS ALB Ingress	Works with AWS Application Load Balancer

Istio Ingress	Envoy-based, part of Istio service mesh

Ingress Controller:

https://blog.saeloun.com/2023/03/21/setup-nginx-ingress-aws-eks/

Setting up nginx-ingress controller

https://amod-kadam.medium.com/setting-up-nginx-ingress-controller-with-eks-f27390bcf804

https://amod-kadam.medium.com/setting-up-a-tls-certificate-with-nginx-ingress-controller-with-amazon-eks-15801a2faf39

AWS ALB Ingress Controller

https://aws.amazon.com/blogs/opensource/kubernetes-ingress-aws-alb-ingress-controller/

https://github.com/iam-veeramalla/aws-devops-zero-to-hero/blob/main/day-22/alb-controller-add-on.md

https://github.com/aws/eks-charts/blob/master/stable/aws-load-balancer-controller/README.md

Gateway API

https://kubernetes.io/docs/concepts/services-networking/gateway/

The **Gateway API** in Kubernetes is the **next-generation networking API** designed to replace and improve upon the older **Ingress API**. It provides more **flexibility, extensibility, and role-oriented separation of concerns** for managing traffic into and within your Kubernetes cluster.

Why Gateway API?

The `Ingress` API is simple but **limited**:

- It bundles routing and load balancing into a single resource.
- Hard to manage in large teams.
- Hard to extend across use-cases (multi-tenant, mesh, etc.).

The **Gateway API** was designed to address those issues and work well across:

- **North-South traffic** (outside → cluster)
- **East-West traffic** (in-cluster service-to-service)

Core Concepts

Here are the key building blocks of the Gateway API:

Resource	Purpose
GatewayClass	Defines the kind of load balancer or proxy (like NGINX, Istio, etc.)
Gateway	Defines an instance of a GatewayClass (a load balancer)
HTTPRoute	Defines routing rules (like Ingress rules)
TLSRoute, **TCPRoute**, **UDPRoute**	Handle non-HTTP traffic
BackendRefs	Define where traffic is sent (Services or other backends)

Example Demo

1. GatewayClass (defines type of load balancer)

```
apiVersion: gateway.networking.k8s.io/v1beta1
kind: GatewayClass
metadata:
  name: example-gatewayclass
spec:
  controllerName: nginx.org/gateway-controller
```

2. Gateway (instantiates a listener on port 80)

```
apiVersion: gateway.networking.k8s.io/v1beta1
```

```yaml
kind: Gateway
metadata:
  name: my-gateway
spec:
  gatewayClassName: example-gatewayclass
  listeners:
  - name: http
    protocol: HTTP
    port: 80
    allowedRoutes:
      namespaces:
        from: Same
```

3. HTTPRoute (defines routing rule)

```yaml
apiVersion: gateway.networking.k8s.io/v1beta1
kind: HTTPRoute
metadata:
  name: my-route
spec:
  parentRefs:
  - name: my-gateway
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /app
    backendRefs:
    - name: my-service
      port: 80
```

Real-World Benefits

- **Decouples infrastructure from routing** (Gateway from Routes)
- **Multiple teams** can manage routing independently
- Works for **multi-tenant** environments

- Supports **advanced features** like mTLS, headers-based routing, etc.
- **Works with Service Mesh** (e.g., Istio, Linkerd)

Summary

Component	Think of it as…
GatewayClass	Load Balancer Template
Gateway	Load Balancer Instance
HTTPRoute	URL Routing Rules
BackendRef	Destination Services

To use the Gateway API, a controller is required. In this lab, we will install NGINX Gateway Fabric as the controller. Follow these steps to complete the installation:

Install the Gateway API resources

```
kubectl kustomize
"https://github.com/nginx/nginx-gateway-fabric/config/crd/gateway-api/standard?ref=v1.5.1" | kubectl apply -f -
```

Deploy the NGINX Gateway Fabric CRDs

```
kubectl apply -f
https://raw.githubusercontent.com/nginx/nginx-gateway-fabric/v1.6.1/deploy/crds.yaml
```

Deploy NGINX Gateway Fabric

```
kubectl apply -f
https://raw.githubusercontent.com/nginx/nginx-gateway-fabric/v1.6.1/deploy/nodeport/deploy.yaml
```

Verify the Deployment

```
kubectl get pods -n nginx-gateway
```

View the nginx-gateway service

```
kubectl get svc -n nginx-gateway nginx-gateway -o yaml
```

Update the nginx-gateway service to expose ports 30080 for HTTP and 30081 for HTTPS

```
kubectl patch svc nginx-gateway -n nginx-gateway --type='json' -p='[
  {"op": "replace", "path": "/spec/ports/0/nodePort", "value": 30080},
  {"op": "replace", "path": "/spec/ports/1/nodePort", "value": 30081}
]'
```

Network Policies

https://kubernetes.io/docs/concepts/services-networking/network-policies/

A **NetworkPolicy** in Kubernetes defines **how pods are allowed to communicate** with:

- Other pods
- Namespaces
- IP blocks (like external IP ranges)

By default, **all traffic is allowed** between pods in Kubernetes (assuming no NetworkPolicy is applied).
Once you apply a NetworkPolicy to a pod **selector**, only the traffic **explicitly allowed** in the policy is permitted — **everything else is denied**.

Basic Structure of a NetworkPolicy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-app-traffic
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: myapp
  policyTypes:
  - Ingress
```

```yaml
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
  egress:
  - to:
    - ipBlock:
        cidr: 8.8.8.0/24
```

Key Concepts

Term	Meaning
`podSelector`	Selects pods this policy applies to
`ingress`	Incoming traffic rules
`egress`	Outgoing traffic rules
`from` / `to`	Who can send/receive traffic
`policyTypes`	Direction: `Ingress`, `Egress`, or both

Example Scenarios

1. Allow traffic only from frontend to backend

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-to-backend
```

```yaml
  namespace: app-ns
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
```

This applies to pods labeled `app=backend`
Only pods labeled `app=frontend` can connect to backend pods.

2. Deny all traffic to a pod

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: sensitive
  policyTypes:
  - Ingress
```

No `ingress` rules = **no traffic allowed in**
Used to isolate sensitive pods.

3. Allow egress to specific IP range

```yaml
egress:
- to:
  - ipBlock:
      cidr: 10.0.0.0/24
```

Used to allow connections to specific external IP ranges (e.g., internal databases, APIs).

Important Notes

- Your **CNI plugin (e.g., Calico, Cilium, Weave)** must support NetworkPolicies.
- Policies are **additive**: multiple policies can apply to the same pod.
- If **no policies** apply to a pod: all traffic is allowed.
- If **one policy applies**, it acts as a **default deny + explicit allow** model.

Custom Resource Definitions (CRDs) in Kubernetes

https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

What is a CRD?

A **Custom Resource Definition (CRD)** lets you **extend Kubernetes** by defining your own custom objects. It allows you to create **new resource types**—just like Pods, Deployments, or Services—but tailored for your application or tools.

Example: Why Use a CRD?

Let's say you're building a tool to manage **database instances**. Kubernetes doesn't have a native object like `DatabaseInstance`. You can create a **CRD** like this:

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: databaseinstances.mycompany.com
spec:
  group: mycompany.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                engine:
```

```
                  type: string
                version:
                  type: string
  scope: Namespaced
  names:
    plural: databaseinstances
    singular: databaseinstance
    kind: DatabaseInstance
    shortNames:
      - dbi
```

This creates a new resource kind:

```
kubectl get databaseinstances
```

Example: Using the New Resource

Once CRD is installed, you can create a custom resource:

```
apiVersion: mycompany.com/v1
kind: DatabaseInstance
metadata:
  name: my-db
spec:
  engine: postgres
  version: "14"
```

Kubernetes will now **accept and store** this object, just like it does with Pods.

What About Logic?

By default, a CRD just stores the object. To add **logic**, like "create a PostgreSQL pod if this resource is created", you write a **controller** (usually using the **Operator pattern**).

This controller watches your custom objects and performs actions.

Use Cases for CRDs

Use Case	Example

Databases as a service	`Database`, `RedisCluster`, `Postgres`
CI/CD pipelines	`PipelineRun`, `Build`, `Trigger`
Custom app lifecycle	`AppDeployment`, `CanaryRelease`
Observability	`AlertRule`, `MetricScraper`

Tools to Help

- **Kubebuilder**: Scaffolds CRDs + Controllers (Go).
- **Operator SDK**: Build production-ready Kubernetes Operators.
- **Crossplane**: Manages infrastructure using CRDs.

Summary

Concept	Description
CRD	Defines new Kubernetes resource type
Custom Resource	Instance of that type (like a Pod is to Deployment)
Controller/Operator	Adds behavior/automation to those custom objects

CoreDNS in Kubernetes

https://kubernetes.io/docs/tasks/administer-cluster/coredns/

What is CoreDNS?

CoreDNS is the **DNS server used inside Kubernetes clusters**.
It handles **name resolution** for Services, Pods, and external domains.

- Runs as a **Deployment** in the `kube-system` namespace.
- Pods send DNS queries to it (usually `10.96.0.10`).
- It's responsible for turning names like `backend-svc.default.svc.cluster.local` into IP addresses.

When is CoreDNS Used?

Scenario	Uses CoreDNS?	Notes
Pod calling a Service	Yes	e.g., `curl http://backend-svc` — resolved via CoreDNS
Pod calling an external website	Yes	e.g., `curl https://api.github.com` — forwarded to upstream DNS
Pod calling another Pod by hostname	Sometimes	Only if using DNS names, not direct IP
Pod-to-Pod with Service	Yes	Most common — service name is resolved to ClusterIP
Ingress Controller routing to Service	No	Uses Kubernetes API, not DNS
Gateway API routing to Service	No	Also uses Kubernetes API, not DNS

How CoreDNS Works

Example: `frontend` Pod wants to call `backend-svc`

1. **App sends a DNS request**:
   DNS query for `backend-svc.default.svc.cluster.local`.
2. **CoreDNS receives it**:
   CoreDNS matches the `.svc.cluster.local` zone and contacts the **Kubernetes**

API.

3. **Kubernetes API replies**:
   It gives the **ClusterIP** of `backend-svc`.
4. **CoreDNS returns the IP** to the Pod.
5. **App connects** to the resolved IP.

CoreDNS enables **service discovery** in Kubernetes.

DNS Names Inside Kubernetes

Name	Meaning
`backend-svc`	Service in the **same namespace**
`backend-svc.default`	Service in `default` namespace
`backend-svc.default.svc`	Service in `svc` zone in `default` NS
`backend-svc.default.svc.cluster.local`	Full FQDN

CoreDNS Architecture

CoreDNS Pod Setup:

- Usually runs 2+ replicas for HA
- Exposes port 53/UDP for DNS queries
- Configured using a `Corefile`

Sample `Corefile`:

```
.:53 {
    errors
```

```
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    forward . /etc/resolv.conf
    cache 30
}
```

What This Does:

- Handles `.cluster.local` zone via `kubernetes` plugin
- Forwards unknown domains to `/etc/resolv.conf` (e.g., Google DNS)
- Caches results for 30 seconds

When CoreDNS Is *Not* Used

Ingress and Gateway API Controllers

Both use the **Kubernetes API**, not DNS, to route traffic to services:

Controller	Uses CoreDNS?	Why Not?
Ingress	No	Knows services via K8s API
Gateway API	No	Uses Gateway → Route → Service directly

Example: End-to-End Request Flow (Frontend → Backend → DB)

1. **Client** hits Ingress/Gateway → routes to `frontend-svc`
2. **Frontend Pod** calls `backend-svc` → DNS request to CoreDNS → resolved to ClusterIP
3. **Backend Pod** calls `postgres-svc` → DNS request to CoreDNS → resolved to ClusterIP

Diagram

```
[Client]
   ↓
[Ingress or Gateway Controller] — uses API (no DNS)
   ↓
[frontend Pod]
   ↓ DNS lookup
[CoreDNS → resolves backend-svc]
   ↓
[backend Pod]
   ↓ DNS lookup
[CoreDNS → resolves postgres-svc]
   ↓
[PostgreSQL Pod]
```

Some useful links

Kubeadm Master-Worker node setup

https://github.com/LondheShubham153/kubestarter/tree/main/Kubeadm_Installation_Scripts_and_Documentation

Ports: 6443, 10250

HELM Setup

https://github.com/LondheShubham153/kubestarter/tree/main/examples/helm

EKS Cluster Setup

https://theshubhamgour.hashnode.dev/eksonaws

Small Projects

1. Deploy on local minikube
2. Deploy on kubeadm master-node server
3. Deploy using helm chart
4. Deploy on EKS
5. Deploy using ArgoCD, along with K8's dashboard and Prometheus-Grafana monitoring setup.
6. Deploy using Ingress Controller (nginx, ALB)

Setting up nginx-ingress controller

https://amod-kadam.medium.com/setting-up-nginx-ingress-controller-with-eks-f27390bcf804

https://amod-kadam.medium.com/setting-up-a-tls-certificate-with-nginx-ingress-controller-with-amazon-eks-15801a2faf39

AWS ALB Ingress Controller

https://aws.amazon.com/blogs/opensource/kubernetes-ingress-aws-alb-ingress-controller/

https://github.com/iam-veeramalla/aws-devops-zero-to-hero/blob/main/day-22/alb-controller-add-on.md

https://github.com/aws/eks-charts/blob/master/stable/aws-load-balancer-controller/README.md

K8s deployment using ArgoCD with K8s dashboard and Grafana-Prometheus monitoring setup.

https://github.com/LondheShubham153/k8s-kind-voting-app/blob/main/kind-cluster/commands.md

https://youtu.be/Kbvch_swZWA?si=fL-IrGCy68YUX7St

Three-Tier Application Deployment on EKS

https://github.com/LondheShubham153/TWSThreeTierAppChallenge

https://docs.google.com/document/d/1nlDc1yE0ebp5ZD37MIvR_kct4spPmBL8ul5DyQvAaCY/edit?tab=t.0

Deployment Strategies in K8s

In Kubernetes, **deployment strategies** define how updates (like app version changes) are rolled out to Pods in a controlled manner. These strategies are crucial in production environments to ensure **high availability** and **minimal downtime**.

1. Rolling Update (Default)

What it is:
Gradually updates Pods **one at a time** (or a few at a time), replacing old ones with new ones, while keeping the application available.

How it works:

- Kubernetes terminates a few old Pods and creates a few new ones.

- `maxUnavailable`: How many old Pods can be down at once.
- `maxSurge`: How many extra Pods can be added above the desired count temporarily.

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 1
```

Example: You have 3 replicas, `maxSurge: 1`, `maxUnavailable: 1`. K8s can scale up to 4 Pods during the update and tolerate 1 Pod being unavailable.

Use case: Most common for web apps and stateless services needing **zero-downtime** and **gradual rollout**.

2. Recreate

What it is:
Kills all existing Pods and then creates new ones.

How it works:

- No overlap between old and new Pods.
- All old Pods are deleted before new ones come up.

```
strategy:
  type: Recreate
```

Use case:
Useful when:

- The app **can't handle multiple versions running in parallel**.
- There are **breaking changes** (e.g., DB schema changes).
- Downtime is acceptable during the update.

Drawback: There is **service downtime** between the stop and start phases.

3. Blue-Green Deployment

What it is:
Two identical environments (Blue = old, Green = new). Traffic is switched only when the new version is ready.

How it works:

- Deploy new version alongside existing one.
- Verify the green (new) version.
- Switch Ingress/Service to point to green Pods.
- If something breaks, you can switch back to blue.

Manual setup: Typically uses 2 Deployments + manual traffic switch via:

- `kubectl`
- Ingress changes
- DNS update
- Service selector change

Use case:
Great for **zero-downtime**, **safe rollbacks**, and **user acceptance testing (UAT)** in production.

Drawback:
Double resource usage during deployment.

4. Canary Deployment

What it is:
Releases new version to a **small percentage of users**, gradually increasing exposure if no issues are found.

How it works:

- Deploy new version with fewer replicas.
- Route 5–10% of traffic to new version.
- Observe logs, metrics, and errors.
- If healthy, increase traffic and eventually replace old version.

How to implement:

- Multiple Deployments (one with fewer replicas).
- Traffic splitting via:
    - **Service Mesh** (e.g., Istio)
    - **Ingress controller with weights**
    - **Argo Rollouts**

Use case:
Ideal for **progressive delivery**, **fault detection**, and **low-risk releases**.

Drawback:
More complex setup, especially with traffic routing.

5. A/B Testing

What it is:
Route users to different versions based on **specific conditions** (e.g., headers, cookies, geography).

How it works:

- Deploy multiple versions (A and B).
- Use Ingress or service mesh rules to send users selectively.
- Evaluate metrics per version.

Use case:
Used for **feature testing**, **UX experimentation**, or **personalization**.

Implementation tools:

- Ingress + header-based routing
- Istio + VirtualService
- Flagger, Argo Rollouts

Drawback:
Requires **advanced routing** and **metrics integration**.

Practice Tasks

Beginner to Intermediate Level Tasks

1. Deploy a Multi-Tier Web Application => Done

- Frontend: React or Nginx static site
- Backend: Node.js/Flask/Java
- Database: PostgreSQL/MySQL
  Use ConfigMaps, Secrets, PVCs for DB

2. Rolling Updates and Rollbacks

- Deploy an app with multiple versions
- Practice:
  - Rolling updates (`kubectl rollout`)
  - Rollback on failure

3. Ingress Setup => Done

- Use `ingress-nginx` to expose:
  - `/api` to backend service
  - `/` to frontend service
    Add TLS using cert-manager + Let's Encrypt

4. Horizontal Pod Autoscaling (HPA)

- Deploy an app that generates CPU load
- Use `kubectl autoscale` to scale based on CPU Use `stress` tool to test

5. Persistent Volumes (PV & PVC) => Done

- Deploy a WordPress app or MySQL
- Use `hostPath` or `local` storage on-prem Backup/restore data via PVC

Advanced Tasks (Real-World)

6. CI/CD with GitHub Actions or Jenkins

- Automate:
  - Docker build & push
  - Apply manifests using `kubectl` Trigger on code push

7. Secrets Management

- Store secrets using:
  - Kubernetes Secrets
  - HashiCorp Vault (integrate with K8s) Rotate secrets automatically

8. Monitoring & Logging Stack => Done

- Deploy:
  - Prometheus + Grafana
  - Loki or EFK (Elasticsearch, Fluentd, Kibana) Add alerts for memory/CPU thresholds

9. Pod Disruption Budgets (PDB) + Node Draining

- Apply PDBs to critical workloads
- Test draining a node (`kubectl drain`)
  Prevent downtime

10. Network Policies

- Create policies to:
  - Allow frontend → backend
  - Deny backend → frontend Use Calico or Cilium as CNI

11. Helm Chart Packaging => Done

- Create a custom Helm chart for an app
- Include `values.yaml`, `templates/`, hooks Deploy via `helm install`

12. Simulate Node Failure

- Kill kubelet on a node
- Observe:
    - Pod rescheduling
    - Alerts from monitoring system
      Enable node auto-repair logic if using kubeadm

Ops Tasks

13. Kubernetes Backup & Restore => Done

- Backup etcd (if using kubeadm)
- Backup persistent volumes
- Restore on a new cluster

14. Cluster Upgrade (kubeadm) => Done

- Upgrade Kubernetes version safely Test upgrade on test cluster

15. Set Up Kube Dashboard with RBAC

- Enable dashboard
- Create admin user with proper RoleBinding Secure with HTTPS and token logic