CoreDNS

What it does: Provides **DNS service inside the Kubernetes cluster** so Pods can find each other by name. CoreDNS is the **internal DNS of Kubernetes**. It allows **Pods and Services to find and talk to each other using names instead of IPs**, which constantly change.

Example: `frontend` Pod connects to `backend.default.svc.cluster.local` without knowing IPs.

✅ Real-World Use Case: Microservices Communication

Scenario

You have an **E-commerce application** running on Kubernetes:

- `frontend` (React)
- `backend` (Node.js / Java API)
- `database` (MongoDB)

Each component runs in different Pods, and Pods can restart at any time → **IP addresses change**.

❌ Without CoreDNS (problem)

- Backend Pod IP changes after restart
- Frontend still points to old IP
- Application breaks ❌

✅ With CoreDNS (solution)

You create a Kubernetes **Service** for backend:

```
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: backend
```

```yaml
  ports:
  - port: 3000
```

CoreDNS automatically creates a DNS entry:

```
backend.default.svc.cluster.local
```

🔄 How traffic flows

1. Frontend sends request to:

   ```
   http://backend:3000
   ```

2. CoreDNS resolves `backend` → Service IP
3. Service load-balances traffic to backend Pods
4. Application works even if Pods restart ✅

📌 Another Real-World Example: Database Access

Backend connects to database using DNS:

```
DB_HOST=mongodb.default.svc.cluster.local
```

Even if MongoDB Pods restart or scale, **no config change needed**.

🚀 Enterprise Use Case

- Used in **microservices, EKS, GKE, AKS**
- Mandatory for **service discovery**
- Critical for **zero-downtime deployments**
- Required for **HPA, Ingress, Gateway API**

🎯 Interview One-Liner

CoreDNS enables service discovery in Kubernetes by resolving service names to IPs, allowing microservices to communicate reliably even when Pods change.

storage-provisioner

What it does: Automatically **creates Persistent Volumes (PV)** when a PVC is requested (Dynamic Provisioning). A **storage-provisioner** enables **dynamic storage provisioning** in Kubernetes. It **automatically creates Persistent Volumes (PV)** when an application requests storage using a **PersistentVolumeClaim (PVC)**.

No manual disk creation. No manual PV YAML. Fully automated. ✅

Example: PVC → automatically creates an EBS volume in AWS or local disk in Minikube.

✅ Real-World Use Case: Database in Kubernetes

Scenario

You are deploying **MySQL / MongoDB / PostgreSQL** on Kubernetes.

Databases need **persistent storage**, even if Pods restart or move.

❌ Without storage-provisioner (manual & risky)

1. Admin manually creates disk (EBS, NFS, local disk)
2. Admin creates PV YAML
3. App creates PVC
4. Errors if sizes or access modes mismatch

❌ Slow, error-prone, not scalable

✅ With storage-provisioner (production way)

Step 1: StorageClass (defined once)

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-storage
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp3
```

Step 2: Application requests storage (PVC)

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  storageClassName: fast-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

Step 3: What happens automatically

1. PVC created
2. **storage-provisioner creates a disk (EBS/NFS/local)**
3. PV created and bound
4. Pod mounts storage
5. Data persists even if Pod restarts ✅

📌 Minikube / KIND Example

In Minikube, `storage-provisioner` creates **local disk storage** automatically.

```
kubectl get pvc
kubectl get pv
```

You'll see PV created **without writing PV YAML**.

🚀 Enterprise Use Cases

- Databases (MySQL, PostgreSQL, MongoDB)
- StatefulSets
- CI/CD tools (Jenkins, Nexus, SonarQube)
- Logging systems (ELK, Prometheus)

ngf-nginx-gateway-fabric

What it does: An **NGINX-based Gateway API implementation** for managing traffic using Kubernetes Gateway API. **ngf-nginx-gateway-fabric** is an **NGINX-based implementation of the Kubernetes Gateway API**. It provides **modern, scalable traffic management** (HTTP/HTTPS routing, TLS, multi-service routing) and is the **successor to traditional Ingress**.

Example: Routes HTTP traffic to multiple services using `Gateway` and `HTTPRoute` instead of Ingress.

ngf-nginx-gateway-fabric – Real-World Use Case with Example

Think of it as:

Ingress → Next generation traffic control using Gateway API + NGINX

✅ Real-World Use Case: Microservices Traffic Routing

Scenario

You run multiple services in Kubernetes:

- `frontend`
- `catalog`
- `orders`
- `payments`

All services must be accessed via **one domain** with **path-based routing** and **TLS**.

❌ Old Way (Ingress problem)

- Complex YAML
- Limited extensibility
- Hard to manage multi-team ownership
- Poor separation of infra vs app routing

✅ New Way with NGF (Gateway API)

Step 1: Gateway (Infrastructure team)

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: web-gateway
spec:
  gatewayClassName: nginx
  listeners:
  - name: http
    protocol: HTTP
    port: 80
```

Step 2: HTTPRoute (Application team)

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: app-routes
spec:
  parentRefs:
  - name: web-gateway
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /orders
    backendRefs:
```

```yaml
  - name: orders
    port: 8080
```

🔄 Traffic Flow

1. User hits `https://shop.example.com/orders`
2. **NGF (NGINX Gateway Fabric)** receives request
3. Routes traffic using Gateway API rules
4. Forwards request to `orders` service Pod
5. Supports retries, timeouts, TLS, and load balancing ✅

🚀 Enterprise Use Cases

- Multi-tenant Kubernetes clusters
- Platform teams managing gateways
- Microservices traffic routing
- Zero-downtime deployments
- Blue-Green / Canary releases

🧠 Why Companies Prefer NGF

- Built on **NGINX (battle-tested)**
- Uses **Gateway API (future-proof)**
- Clear separation:
  - Infra team → Gateway
  - App team → Routes
- Better than classic Ingress

🎯 Interview One-Liner

ngf-nginx-gateway-fabric is an NGINX-based Gateway API implementation that provides advanced, scalable traffic management in Kubernetes, replacing traditional Ingress.

🚦 Popular Kubernetes Gateway API Implementations

1. Istio Gateway

Use case: Advanced traffic management & service mesh

- Supports **HTTP, HTTPS, TCP**
- Deep integration with **Istio service mesh**
- Ideal for **zero-trust, mTLS, canary, traffic splitting**

Example: Route 90% traffic to v1, 10% to v2 (canary release)

2. Kong Gateway

Use case: API management & external traffic

- Built-in **rate limiting, auth, plugins**
- Works well for **API-first platforms**

Example: Apply OAuth2 + rate limiting on `/api/*` routes

3. Traefik Gateway

Use case: Simple, cloud-native ingress & gateway

- Auto-discovers services
- Lightweight and easy to operate

Example: Expose services automatically when HTTPRoute is created

4. Envoy Gateway

Use case: High-performance, cloud-native gateway

- Backed by **Envoy Proxy**
- Used heavily in **cloud providers**

Example: Layer 7 routing with retries, timeouts, and observability

5. HAProxy Gateway

Use case: High-throughput, low-latency traffic

- Enterprise-grade reliability
- Popular in finance and telecom

Example: TCP and HTTP routing for legacy + modern apps

6. GKE Gateway (Google Cloud)

Use case: Managed Gateway API on GKE

- Native integration with **Google Load Balancers**
- Automatic TLS and scaling

Example: Expose services with Google-managed HTTPS LB

7. AWS Gateway API (ALB Controller)

Use case: AWS-native traffic routing

- Uses **Application Load Balancer (ALB)**
- Integrated with IAM, ACM, WAF

Example: Public HTTPS access via ALB using Gateway API

📌 How to choose (interview-ready)

Need	Best Choice
Simple & NGINX based	NGINX Gateway Fabric
API management	Kong
Service mesh	Istio
High performance	Envoy
Cloud-native (AWS/GCP)	AWS / GKE Gateway

🎯 Interview One-Liner

> **Gateway API is implemented by multiple providers like NGINX, Istio, Kong, Traefik, Envoy, and cloud-native gateways, each optimized for traffic management, security, and scalability.**

kubelet

What it does: Runs on every worker node and **ensures Pods and containers are running** as defined.

kubelet is the **node-level agent** that runs on **every worker node** in a Kubernetes cluster. It makes sure that **containers described in Pod specs are actually running** on that node.

Example: Starts containers, reports Pod status to API Server, restarts failed containers.

Think of kubelet as:

> **"The executor of Kubernetes instructions on a node."**

🔧 What kubelet is responsible for

- Talks to **kube-apiserver**
- Pulls container images

- Starts & stops containers via container runtime (containerd/CRI-O)
- Performs **health checks (liveness/readiness)**
- Reports Pod & node status
- Mounts volumes and secrets

✅ Real-World Use Case: Application Pod Lifecycle

Scenario

You deploy a backend application:

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

🔄 What happens behind the scenes

1. **Scheduler** assigns the Pod to `worker-node-1`
2. **kubelet on worker-node-1**:
    - Pulls the Docker image
    - Creates containers
    - Mounts PVCs & secrets
    - Starts the Pod
3. kubelet continuously checks container health
4. If container crashes → kubelet restarts it automatically ✅

📌 Example: Health Check

```yaml
livenessProbe:
  httpGet:
    path: /health
    port: 8080
```

If `/health` fails:

- kubelet kills the container
- kubelet restarts it

🚀 Enterprise Use Cases

- Ensures **self-healing applications**
- Enforces resource limits (CPU/memory)
- Enables **autoscaling and resilience**
- Required for StatefulSets, Jobs, and DaemonSets

🧠 Where kubelet runs

- Runs as a **system service** on each node
- NOT a Pod
- Communicates securely with API Server

🎯 Interview One-Liner

kubelet is the node agent that ensures Pods and containers are running as defined and continuously reports their health and status to the Kubernetes control plane.

kube-proxy

What it does: Handles **network routing and load balancing** for Kubernetes Services.

kube-proxy runs on **every worker node** and is responsible for **network routing and load balancing for Kubernetes Services**.

Example: A request to `ClusterIP` is forwarded to one of the backend Pods.

Think of kube-proxy as:

> “The traffic controller that sends Service requests to the right Pod.”

🔧 What kube-proxy is responsible for

- Implements **Service networking**
- Handles **ClusterIP, NodePort, and LoadBalancer** traffic
- Load-balances traffic across Pods
- Programs **iptables / IPVS** rules on nodes
- Enables Pod-to-Service communication

✅ Real-World Use Case: Service Load Balancing

Scenario

You deploy a backend app with 3 replicas:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: backend
  ports:
  - port: 80
```

Pods:

- backend-pod-1
- backend-pod-2
- backend-pod-3

🔄 How traffic flows

1. Frontend sends request to:

   ```
   http://backend
   ```

2. **CoreDNS** resolves `backend` → ClusterIP

3. **kube-proxy** routes traffic to one of the backend Pods
4. Traffic is load-balanced automatically ✅

📌 NodePort Example

If Service type is `NodePort` :

```
curl http://<node-ip>:30080
```

kube-proxy forwards the request to a backend Pod.

🚀 Enterprise Use Cases

- Microservices communication
- Internal service discovery
- Load balancing without external LB
- Works with Ingress & Gateway APIs

🧠 How kube-proxy works internally

Modes:

- **iptables** (default, simple)
- **IPVS** (high performance, scalable)

🧠 Where kube-proxy runs

- Runs as a **DaemonSet**
- One Pod per node
- Requires kernel networking support

🎯 Interview One-Liner

> kube-proxy implements Kubernetes Service networking by routing and load-balancing traffic to Pods using iptables or IPVS rules.

🔌 Popular CNI (Container Network Interface) Plugins

(CNI = Pod networking: IP assignment & Pod-to-Pod communication)

🔹 Common CNIs

- **Calico** – Networking + Network Policies (widely used in production)
- **Flannel** – Simple overlay networking (beginner-friendly)
- **Cilium** – eBPF-based, high performance, can replace kube-proxy
- **Weave Net** – Simple mesh networking
- **Kindnet** – Default CNI for KIND clusters
- **AWS VPC CNI** – Native AWS networking for EKS
- **Azure CNI** – Native networking for AKS
- **GKE Dataplane v2** – Google's CNI (eBPF-based)
- **Antrea** – VMware-backed, based on Open vSwitch

🔹 What CNIs handle

- Pod IP assignment
- Pod-to-Pod communication
- Cross-node networking
- (Some CNIs) Network policies

🔁 kube-proxy (Service Networking)

(kube-proxy = Service IP routing & load balancing)

🔹 kube-proxy Modes

- **iptables mode** (default)
    - Simple, widely used

    - Uses Linux iptables rules

- **IPVS mode**
    - High performance
    - Better for large clusters
- **userspace mode** (deprecated ❌)

🔹 What kube-proxy handles

- ClusterIP, NodePort, LoadBalancer Services
- L4 load balancing
- Routing Service traffic to Pods

🧠 Special Case: kube-proxy Replacement

Some CNIs **replace kube-proxy functionality internally**:

- **Cilium (eBPF)**
- **GKE Dataplane v2**

➡️ kube-proxy Pod may not run, but **Service routing still exists**

🎯 Interview One-Liner

> **CNI plugins provide Pod networking, while kube-proxy (or its replacement) provides Service networking and load balancing in Kubernetes.**

🔑 Quick Memory Trick

```
CNI        → Pod IP & Pod traffic
kube-proxy → Service IP & load balancing
---


### **kube-scheduler**

**What it does:**
```

Decides **which node a Pod should run on** based on resources and constraints.

kube-scheduler is the **control plane component** that decides **which worker node a Pod should run on**.

Example:
Schedules a Pod on the node with enough CPU and memory.

Think of kube-scheduler as:

“The brain that places Pods on the right nodes.”

🧠 What kube-scheduler considers

When scheduling a Pod, it evaluates:

* Available **CPU & memory**
* **Node labels & selectors**
* **Taints and tolerations**
* **Affinity / anti-affinity rules**
* **Topology spread constraints**
* Resource requests & limits

✅ Real-World Use Case: Placing Pods Correctly

Scenario

You have 3 nodes:

* `node-1` → high CPU
* `node-2` → GPU enabled
* `node-3` → general purpose

A Pod requires a GPU.

```yaml
nodeSelector:
  accelerator: nvidia

# 🔄 What happens
```

1. Pod is created
2. kube-scheduler filters nodes
3. Only `node-2` matches
4. Pod is scheduled to `node-2` ✅

📌 Example: High Availability (Anti-Affinity)

```yaml
podAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
  - labelSelector:
      matchLabels:
        app: backend
    topologyKey: kubernetes.io/hostname
```

➡️ Scheduler spreads Pods across different nodes to avoid single-node failure.

🚀 Enterprise Use Cases

- High availability applications
- Cost-optimized workloads
- GPU / special hardware scheduling
- Multi-tenant clusters

🧠 Where kube-scheduler runs

- Runs as a **static Pod** on control-plane nodes
- Communicates via **kube-apiserver**
- Does NOT run on worker nodes

🎯 Interview One-Liner

> kube-scheduler assigns Pods to the most suitable worker nodes based on resource availability, constraints, and policies.

kube-controller-manager

What it does: Runs controllers that **maintain the desired state** of the cluster.

kube-controller-manager runs a set of **controllers** that continuously **ensure the actual cluster state matches the desired state** defined in Kubernetes objects.

Example: If a Pod crashes, the ReplicaSet controller creates a new Pod automatically.

Think of it as:

> "The auto-fix engine of Kubernetes."

🧠 What controllers do

Each controller watches the API Server and **takes action when something is missing or broken**.

Common controllers include:

- Node Controller
- ReplicaSet Controller
- Deployment Controller
- Job Controller
- Endpoint Controller
- Namespace Controller

✅ Real-World Use Case: Self-Healing Pods

Scenario

You deploy an app with 3 replicas:

```
spec:
  replicas: 3
```

🔄 What happens

1. One Pod crashes
2. ReplicaSet count becomes 2
3. **ReplicaSet controller** detects mismatch
4. New Pod is created automatically
5. Desired state restored to 3 Pods ✅

📌 Another Example: Node Failure

- Worker node goes down
- **Node controller** marks it `NotReady`
- Pods are rescheduled on healthy nodes

🚀 Enterprise Use Cases

- Auto-healing applications
- Auto-scaling support
- Continuous reconciliation
- Enforcing cluster policies

🧠 Where kube-controller-manager runs

- Runs as a **static Pod** on control-plane nodes
- Communicates via **kube-apiserver**
- Does NOT run on worker nodes

🎯 Interview One-Liner

> **kube-controller-manager continuously monitors the cluster and runs controllers that reconcile the desired state with the actual state.**

🔑 Mental Model

```
Desired State (YAML)
        ↓
Controller Manager
        ↓
Actual State Fixed Automatically
```

kube-apiserver

What it does: The **main entry point** of the Kubernetes cluster; all requests go through it. **kube-apiserver** is the **front door and brain interface** of Kubernetes. Every action in a Kubernetes cluster **must go through the API Server**.

Example: `kubectl get pods` → API Server → etcd → response returned.

kube-apiserver – Real-World Explanation with Example

Think of it as:

> "The control center that validates, processes, and stores all cluster requests."

🧠 What kube-apiserver is responsible for

- Exposes the **Kubernetes API**
- Authenticates and authorizes requests (**AuthN/AuthZ**)
- Validates requests and schemas
- Communicates with **etcd** to store/retrieve cluster state
- Acts as the **only component** that talks directly to etcd

Scenario

You run:

```
kubectl get pods
```

🔄 What happens behind the scenes

1. `kubectl` sends request to **kube-apiserver**
2. API Server:
    - Authenticates user (cert/IAM/token)
    - Authorizes via RBAC
3. Reads Pod data from **etcd**
4. Returns response to kubectl

➡️ **No API Server = No cluster access**

📌 Real-World Example: Creating a Pod

```
kubectl apply -f app.yaml
```

Flow:

```
kubectl → kube-apiserver → validation → etcd → controllers → kubelet
```

🚀 Enterprise Use Cases

- Central access point for:
    - kubectl
    - CI/CD pipelines
    - Controllers & schedulers
- Enforces **security and governance**
- Enables multi-tenant clusters
- Required for autoscaling & GitOps

🧠 Where kube-apiserver runs

- Runs as a **static Pod** on control-plane nodes
- Highly available (multiple replicas)
- Exposed securely (TLS)

🎯 Interview One-Liner

> **kube-apiserver is the central management component of Kubernetes that handles all API requests, enforces security, and stores cluster state in etcd.**

🔑 Mental Model

```
Users / CI / Controllers
          ↓
     kube-apiserver
          ↓
         etcd
```

etcd

What it does: A **distributed key-value store** that stores all cluster state and configuration.

etcd is a **distributed, consistent key-value database** that stores **all Kubernetes cluster state and configuration**.

Example: Stores Pod specs, secrets, ConfigMaps, and node information.

Think of it as:

> **"The single source of truth for the entire Kubernetes cluster."**

If **etcd is lost** → **the cluster is lost**.

🧠 What is stored in etcd

- Pod definitions & status
- Node information
- Deployments, Services, ReplicaSets
- ConfigMaps & Secrets
- RBAC rules
- Cluster configuration & metadata

➡️ **Nothing runs without etcd data**

✅ Real-World Use Case: Pod Creation

Scenario

You apply a Deployment:

```
kubectl apply -f app.yaml
```

🔄 What happens

1. Request goes to **kube-apiserver**
2. API Server validates the request
3. **Object is written to etcd**
4. Controllers read from etcd
5. Scheduler schedules Pods
6. kubelet runs Pods

➡️ etcd makes the desired state durable and recoverable ✅

📌 Real-World Use Case: Cluster Recovery

- Control plane crashes
- New control plane is started

- Reads state from **etcd**
- Cluster is restored exactly as before

🚀 Enterprise Best Practices

- Run etcd in **odd numbers (3 or 5 nodes)**
- Enable **TLS encryption**
- Take **regular etcd backups**
- Never expose etcd publicly
- Monitor disk I/O & latency

🧠 Where etcd runs

- Runs as a **static Pod** on control-plane nodes
- Communicates only with **kube-apiserver**
- Uses **RAFT consensus** for consistency

🎯 Interview One-Liner

> etcd is a highly available, distributed key-value store that holds the complete state of a Kubernetes cluster.

🔑 Mental Model

```
Cluster State (Truth)
        ↓
       etcd
        ↑
  kube-apiserver only
```

kindnet

What it does: A **CNI (Container Network Interface) plugin** used by KIND clusters for Pod networking.

Example: Assigns IPs to Pods and enables Pod-to-Pod communication.

metrics-server

What it does: Collects **CPU and memory usage metrics** from nodes and Pods.

metrics-server is a **cluster add-on** that collects **CPU and memory usage metrics** from Kubernetes nodes and Pods.

Example: Used by `kubectl top pods` and Horizontal Pod Autoscaler (HPA).

Think of it as:

> **"The resource usage meter for Kubernetes."**

Without metrics-server, Kubernetes **cannot autoscale Pods**.

🧠 What metrics-server provides

- CPU usage per Pod
- Memory usage per Pod
- CPU & memory usage per node
- Metrics for **HPA (Horizontal Pod Autoscaler)**

> ❗ It does **NOT** store long-term metrics (that's Prometheus' job).

✅ Real-World Use Case: Auto Scaling Applications

Scenario

You run a backend app with HPA:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
spec:
  scaleTargetRef:
```

```yaml
    kind: Deployment
    name: backend
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

🔄 What happens

1. metrics-server collects CPU usage from kubelet
2. HPA reads metrics
3. CPU > 70% → Pods scale up
4. CPU < 70% → Pods scale down

➡️ **Automatic scaling works only because of metrics-server** ✅

📌 CLI Example

```
kubectl top pods
kubectl top nodes
```

If metrics-server is missing:

```
error: Metrics API not available
```

🚀 Enterprise Use Cases

- Horizontal Pod Autoscaling
- Cluster monitoring (basic)
- Resource optimization

- Capacity planning (short-term)

🧠 Where metrics-server runs

- Runs as a **Deployment** in `kube-system`
- Scrapes metrics from **kubelet**
- Uses Metrics API (`metrics.k8s.io`)

🎯 Interview One-Liner

> **metrics-server collects real-time CPU and memory metrics from Pods and nodes and enables Kubernetes autoscaling.**

🔑 Mental Model

```
kubelet → metrics-server → HPA / kubectl top
```

🧠 Control Plane Components

(Manage and control the Kubernetes cluster)

- **kube-apiserver** – Central API entry point for all cluster operations.
- **etcd** – Distributed key-value store holding cluster state and configuration.
- **kube-scheduler** – Assigns Pods to suitable worker nodes.
- **kube-controller-manager** – Runs controllers to maintain desired cluster state.

🖥️ Node (Worker) Components

(Run applications and handle networking on worker nodes)

- **kubelet** – Ensures Pods and containers are running as defined.

- **kube-proxy –** Handles Service networking and load balancing.
- **kindnet (CNI)** – Provides Pod-to-Pod networking and IP assignment.

🧩 Add-ons / Cluster Services

(Extend Kubernetes functionality)

- **CoreDNS** – Provides DNS-based service discovery inside the cluster.
- **metrics-server** – Collects CPU and memory metrics for Pods and nodes.
- **storage-provisioner** – Enables dynamic Persistent Volume provisioning.
- **ngf-nginx-gateway-fabric** – Implements Gateway API using NGINX for traffic routing.