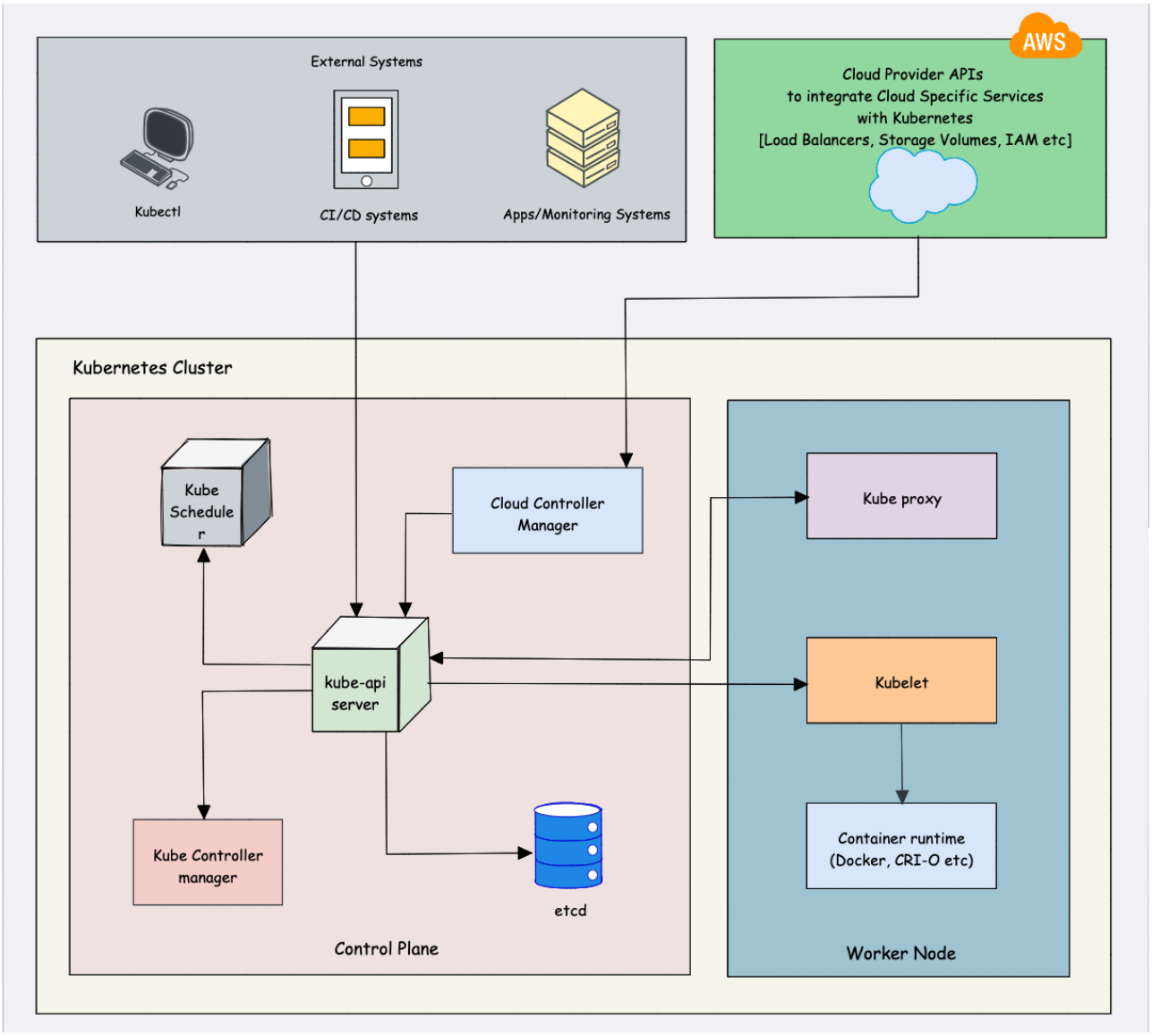
# 02 Kubernetes Architecture and Core Concepts

## Kubernetes Architecture and Core Concepts




## 1. Overview: Control Plane Components

The Control Plane is the "brain" of the Kubernetes cluster. It is responsible for maintaining the desired state of the cluster, meaning it ensures the actual state matches the configuration you provide (e.g., "I want 3 replicas of this application running").

### 1.1. kube-apiserver

- **Role:** The front-end for the Control Plane. It exposes the Kubernetes API.
- **Functionality:**
  - It is the only component that communicates directly with `etcd`.
  - Validates and processes REST requests (e.g., `kubectl apply`, `kubectl get`).
  - Provides the *single source of truth* for the entire cluster state.
  - Handles authentication, authorization, and API registration.

### 1.2. etcd

- **Role:** The highly available key-value store where all cluster data is permanently stored.
- **Functionality:**
  - Stores the configuration data, state, and metadata of all Kubernetes objects (Pods, Services, Deployments, etc.).
  - Crucial for consistency and fault tolerance. If `etcd` is lost, the cluster is effectively lost.

### 1.3. kube-scheduler

- **Role:** Watches for newly created Pods that have no assigned Node and selects a Node for them to run on.
- **Functionality:**
  - Determines the best Node based on various factors:
    - Resource requirements (CPU, memory).
    - Hardware/Software/Policy constraints (e.g., Node affinity, anti-affinity, taints, tolerations).
    - Inter-Pod affinity and anti-affinity.
    - Data locality.
  - Once a decision is made, it informs the `kube-apiserver` by binding the Pod to the selected Node.

### 1.4. kube-controller-manager

- **Role:** Runs *controller loops* that continuously monitor the state of the cluster via the API server and make changes to move the current state closer to the desired state.
- **Functionality:** It manages several specific controllers bundled into a single process:
  - **Node Controller:** Responsible for noticing and responding when nodes go down.
  - **Replication Controller (and ReplicaSet Controller):** Ensures the desired number of Pod replicas is running.
  - **Endpoints Controller:** Populates the `Endpoints` object, which connects `Services` to `Pods`.
  - **ServiceAccount & Token Controllers:** Manages API access tokens and Service Accounts.

### 1.5. cloud-controller-manager (Optional)

- **Role:** Embeds cloud-specific control logic into Kubernetes, allowing cluster management to interact with the underlying cloud provider.
- **Functionality:**
  - **Node Controller:** Checks the cloud provider to see if a node terminated after it stops responding.
  - **Route Controller:** Sets up network routes in the cloud for container communication.
  - **Service Controller:** Creates cloud provider load balancers (e.g., AWS ELB, Azure LB, GCP Load Balancer) when a Kubernetes `Service` of type `LoadBalancer` is defined.

## 2. Worker Node Components

Worker Nodes (or simply "Nodes") are the machines that run your actual application containers. They are managed by the Control Plane.

### 2.1. kubelet

- **Role:** The primary "agent" running on every Node. It ensures that containers described in PodSpecs are running and healthy.
- **Functionality:**
  - Communicates with the `kube-apiserver` to receive PodSpecs (the desired state) for the Node.
  - Communicates with the **Container Runtime** (e.g., Containerd, CRI-O) to start, stop, and manage containers.
  - Continuously monitors the health and state of running Pods and reports status back to the `kube-apiserver`.

### 2.2. kube-proxy

- **Role:** A network proxy that runs on every Node, maintaining network rules on the host.
- **Functionality:**
  - Implements the **Service** abstraction.
  - Manages IP address translation (NAT) and simple TCP/UDP/SCTP forwarding for Services.
  - Directs traffic destined for a Service's ClusterIP and Port to the correct backend Pods, ensuring load balancing across all healthy replicas.
  - It typically uses `iptables` or `IPVS` (IP Virtual Server) to implement these rules.

### 2.3. Container Runtime

- **Role:** The software responsible for running containers.
- **Functionality:**
  - Pulls container images from registries (like Docker Hub or ECR).
  - Mounts volumes and sets up networking for the containers.
  - Executes the containers based on the Open Container Initiative (OCI) specification.
  - Common runtimes include **Containerd** and **CRI-O**. (Docker used to be the default, but Kubernetes now uses runtimes compatible with the Container Runtime Interface (CRI)).

## 3. Kubernetes API and kubectl Basics

### 3.1. Kubernetes API

- The Kubernetes API is a **RESTful API** that serves as the foundation for the entire control plane.
- All operations—from creating a Pod to scaling a Deployment—are essentially CRUD (Create, Read, Update, Delete) operations performed on API objects via HTTP requests to the `kube-apiserver`.
- **Declarative vs. Imperative:** Kubernetes strongly favors a **declarative model**, where you *declare* the desired state (often in YAML files), and the system works to achieve it.

### 3.2. kubectl Basics

- `kubectl` is the command-line interface (CLI) used to communicate with the Kubernetes API server.
- **Essential Commands:**
  - `kubectl apply -f <file.yaml>`: **Declarative** way to create or update resources based on a configuration file. This is the primary method.
  - `kubectl create -f <file.yaml>`: **Imperative** way to create a resource (often used for initial setup).

- `kubectl get <resource_type>`: Lists resources (e.g., `kubectl get pods`, `kubectl get svc`). Add `-o wide` for more details.
- `kubectl describe <resource_type> <name>`: Shows detailed information about a specific resource, including its status, events, and related components.
- `kubectl logs <pod_name>`: Prints the logs for a container in a Pod. Add `-c <container_name>` for multi-container Pods.
- `kubectl exec -it <pod_name> -- /bin/bash`: Executes a command inside a running container (useful for debugging).
- `kubectl delete <resource_type> <name>`: Deletes a resource.

# 4. Basic Concepts: Pods, Deployments, Services

## 4.1. Pods (The smallest deployable unit)

- A **Pod** is the fundamental unit of execution in Kubernetes. It represents a single instance of a running process in a cluster.
- **Encapsulation:** A Pod can contain one or more tightly coupled containers that share the same network namespace and storage volumes.
- **Shared Context:**
  - **Networking:** All containers in a Pod share the same IP address and port space. They can communicate with each other using `localhost`.
  - **Storage:** They can share mounted volumes for persistent or temporary data.
- **Ephemeral Nature:** Pods are designed to be *ephemeral* (short-lived) and disposable. They should be managed by higher-level controllers like Deployments or ReplicaSets.

## 4.2. Deployments

- A **Deployment** is a controller that provides declarative updates for Pods and ReplicaSets.
- **Role:** Manages the desired state of a replicated application.
- **Functionality:**
  - Defines *how* an application should be run (e.g., image, resource limits).
  - Defines *how many* replicas should exist (via its underlying ReplicaSet).
  - Handles **Rolling Updates**: A controlled way to update an application's image or configuration with zero downtime.
  - Handles **Rollbacks**: Allows easy reversion to a previous stable version of the application.

## 4.3. Services (Stable networking abstraction)

- A **Service** is an abstract way to expose an application running on a set of Pods as a network service.
- **Role:** Provides a stable IP address and DNS name for a dynamically changing group of Pods.
- **Functionality:**
  - Uses **Label Selectors** to find the set of target Pods.
  - The Service IP remains constant, even if the Pods are destroyed and recreated with new IPs.
  - Provides **load balancing** across the backend Pods.
- **Key Service Types:**
  - `ClusterIP` **(Default):** Exposes the Service on an internal cluster IP. Only reachable from within the cluster. Used for internal communication.

- `NodePort`: Exposes the Service on a static port (the NodePort) on every Node's IP. External traffic can reach the service via `<NodeIP>:<NodePort>`. Used for simple external access or testing.
- `LoadBalancer`: Exposes the Service externally using a cloud provider's load balancer. A cloud IP is provisioned, and the Service automatically routes external traffic to the Pods.
- `ExternalName`: Maps the Service to a DNS name outside the cluster.

# 5. Pod Lifecycle and Networking

## 5.1. Pod Phases

A Pod progresses through a defined lifecycle, summarized by its **Phase**:

1. `Pending`: The Pod has been accepted by the cluster but one or more of the container images have not been created. This includes time spent waiting for scheduling and image download.
2. `Running`: The Pod has been bound to a Node, and at least one container is running, or is in the process of starting or restarting.
3. `Succeeded`: All containers in the Pod have terminated successfully, and will not be restarted. This is typical for **Job** Pods.
4. `Failed`: All containers have terminated, and at least one container terminated in failure (non-zero exit code).
5. `Unknown`: The state of the Pod could not be obtained, usually due to communication error with the host Node.

## 5.2. Restart Policy

Defined in the PodSpec, this policy dictates what happens when a container inside a Pod terminates. It is only relevant for Pods managed by a `kubelet` (i.e., not when the Pod is already being managed by a controller like a Deployment).

- `Always` **(Default for Deployments):** The container is always restarted if it terminates.
- `OnFailure`: The container is restarted only if it terminates with a non-zero exit code (failure).
- `Never` **(Default for Jobs):** The container is never restarted.

## 5.3. Pod Networking Model

Kubernetes enforces a **flat network space** where:

- **Every Pod gets its own unique IP address** within the cluster.
- Pods on the same Node can communicate directly.
- Pods on different Nodes can communicate directly **without NAT** (Network Address Translation).
- Containers within a Pod communicate via `localhost` and share the Pod's IP.

This network model is implemented by a **Container Network Interface (CNI)** plugin (e.g., Calico, Flannel, Cilium).

# 6. ReplicaSets and Deployments

## 6.1. ReplicaSets (RS)

- **Role:** A controller that ensures a specified number of Pod replicas is running at all times.
- **Functionality:**
  - Watches for Pods matching its **label selector**.
  - Creates new Pods if the current count is less than the desired replica count.
  - Deletes excess Pods if the current count is greater than the desired count.
  - **Key Difference from ReplicationController (Legacy):** ReplicaSets support more advanced set-based selectors (e.g., matching a label value that is "in" a list of values), whereas ReplicationControllers only support equality-based selectors.

## 6.2. Deployments vs. ReplicaSets

- **Deployment (Higher-Level Abstraction):** You rarely interact with a ReplicaSet directly. You define a **Deployment**, which automatically creates and manages the underlying ReplicaSet.
- **Rolling Updates and Rollbacks:** The key value of the Deployment is its ability to manage **changes** to the Pod Template:
  1. When you update the Deployment (e.g., change the container image), the Deployment creates a **new ReplicaSet** with the updated Pod template.
  2. It gradually scales up the new ReplicaSet and simultaneously scales down the old one. This is a **Rolling Update**.
  3. If the new version fails, you can use `kubectl rollout undo` to revert the Deployment, which simply scales down the new ReplicaSet and scales up the previous one (the Rollback).
  4. Deployments manage the history of these ReplicaSets, allowing for easy transitions between versions.

# 7. Service Discovery and Load Balancing

## 7.1. Service Discovery

Service discovery is the process by which a client (e.g., one Pod) finds the network address (IP and Port) of a service (e.g., a group of backend Pods). Kubernetes offers two main forms:

1. **DNS (Recommended):**
   - Kubernetes runs an internal DNS service (usually **CoreDNS**).
   - When you create a Service named `my-backend`, CoreDNS automatically creates a DNS record: `my-backend.<namespace>.svc.cluster.local`.
   - Clients inside the cluster can simply resolve `my-backend` (if in the same namespace) or the fully qualified domain name (FQDN) to get the Service's stable ClusterIP.
2. **Environment Variables (Legacy):**
   - When a Pod starts, the `kubelet` injects environment variables for every active Service into that Pod's containers.
   - *Limitation:* This only works for Services created *before* the Pod is started. DNS is more flexible.

### 7.2. Load Balancing

- **Internal Load Balancing (via `kube-proxy`):**
  - When a client connects to a Service's ClusterIP, the `kube-proxy` (using `iptables` or `IPVS`) intercepts the traffic and randomly distributes the connection to one of the healthy backend Pods listed in the associated `Endpoints` object. This is layer 4 (TCP/UDP) load balancing.
- **External Load Balancing (via `Ingress`):**
  - For sophisticated external access, an **Ingress** object is used.
  - An **Ingress** manages external access to services in a cluster, typically HTTP/HTTPS.
  - It requires an **Ingress Controller** (e.g., Nginx, Traefik, Istio) to be running, which acts as the intelligent reverse proxy and Layer 7 (HTTP) load balancer, enabling features like path-based routing (`/api` goes to service A, `/web` goes to service B) and SSL termination.

## 8. Managed vs Self-Hosted Kubernetes

The choice between Managed and Self-Hosted Kubernetes primarily comes down to **who manages the Control Plane**.

| Feature | Self-Hosted Kubernetes (e.g., via kubeadm) | Managed Kubernetes (e.g., GKE, EKS, AKS) |
|---|---|---|
| **Control Plane Management** | You are fully responsible for provisioning, upgrading, scaling, monitoring, securing, and backing up all Control Plane components (`etcd`, API Server, Scheduler, etc.). | The Cloud Provider (or vendor) handles the entire Control Plane lifecycle, including high availability, patching, and upgrades. |
| **Worker Node Management** | You manage the Worker Nodes (OS, patching, scaling, security). | Varies by provider/service (e.g., GKE Autopilot manages both Control Plane and Worker Nodes; standard GKE/EKS often lets you manage the worker nodes). |
| **Customization/Flexibility** | Maximum control. You can install custom security policies, configure network plugins (CNI), and fine-tune every component. | Limited control. You can only customize within the guardrails set by the provider. |
| **Operational Overhead** | Very high. Requires a dedicated, expert team for 24/7 maintenance and troubleshooting of infrastructure. | Low. Reduces operational burden, allowing teams to focus on application development. |
| **Cost** | You pay only for the underlying compute resources (VMs). Infrastructure cost is lower, but operational (labor) cost is higher. | You pay for the underlying compute resources **plus** a management fee for the Control Plane (though sometimes this is free). |
| **Best For** | Highly specific environments (bare-metal, on-premise), maximum security/compliance requirements, or teams with deep Kubernetes expertise. | Almost all standard production workloads. Teams prioritizing speed, reliability, and low maintenance. |

**Conclusion:** For most organizations, **Managed Kubernetes** is the preferred choice as it significantly lowers the operational barrier and allows engineers to focus on application logic rather

than infrastructure maintenance.