01 k8s basics and architecture

Container Orchestration and Kubernetes

Why Container Orchestration?

Containerization (e.g., using Docker) packages an application and all its dependencies into a single, portable unit. While managing a single container is simple, modern applications are often **distributed systems** composed of **microservices**, requiring dozens, hundreds, or even thousands of containers.

Managing this scale manually is practically impossible and introduces significant challenges. **Container orchestration** is the process of automating the deployment, scaling, management, networking, and availability of containerized applications across a cluster of hosts.

Challenges Solved by Orchestration:

- **Scaling and Load Balancing:** How do you run multiple copies (replicas) of a container to handle high traffic, distribute incoming network requests across them, and scale them up/down automatically based on demand?
- **High Availability and Self-Healing:** What happens if a server (node) fails, or a container crashes? An orchestration tool must detect the failure and automatically restart the container or reschedule it on a healthy node to maintain application uptime.
- **Networking and Service Discovery:** Containers are ephemeral and their IP addresses change. How do different microservices find and communicate with each other without hardcoding IP addresses?
- **Deployment Management:** How do you update an application to a new version without downtime (**rolling updates**) and safely roll back if the new version has a critical bug?
- **Resource Optimization:** How do you efficiently pack containers onto the available hosts (nodes) to maximize hardware utilization and minimize costs?
- **Configuration and Secret Management:** How do you inject configuration data (e.g., database URLs, API keys) into containers securely and consistently across different environments?

Container orchestration platforms provide a **declarative approach**—you define the *desired state* of your application (e.g., "I want 3 replicas of my web app running"), and the orchestrator is responsible for moving the *actual state* to match the desired state, continually.

Introduction to Kubernetes and its Architecture

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications. Originally designed by Google, it is now maintained by the Cloud Native Computing Foundation (CNCF).

The Core Concept: The Cluster

A Kubernetes cluster is a set of machines (physical or virtual) that run containerized applications. The cluster is divided into two main parts:

1. **The Control Plane (Master Node):** The "brain" of the cluster that manages the overall state and makes global decisions (like scheduling).
2. **Worker Nodes:** The machines that run the actual application workloads (containers) in the form of **Pods**.

Control Plane Components

These components run on the master node(s) and manage the cluster's state:

- **API Server (`kube-apiserver`):**
  - The **front-end** for the Control Plane. It exposes the Kubernetes API.
  - It is the only component that communicates with the cluster's persistent store. All internal and external requests (e.g., from `kubectl`) go through the API Server.
- **etcd:**
  - A **consistent and highly available key-value store** that holds the entire cluster state, configuration data, and metadata.
  - It is the single source of truth for the cluster.
- **Scheduler (`kube-scheduler`):**
  - Watches for newly created **Pods** that have no node assigned.
  - Selects an optimal node for the Pod to run on based on various factors like resource requirements (CPU/Memory), resource availability, policy constraints, and affinity/anti-affinity specifications.
- **Controller Manager (`kube-controller-manager`):**
  - Runs controller loops that regulate the state of the cluster.
  - Each controller attempts to move the current state closer to the desired state. Examples include:
    - **Replication Controller:** Ensures the specified number of Pod replicas is running.
    - **Node Controller:** Notices when a node goes down and handles its status.
    - **Endpoint Controller:** Populates the Endpoints object (used for Service discovery).

Worker Node Components

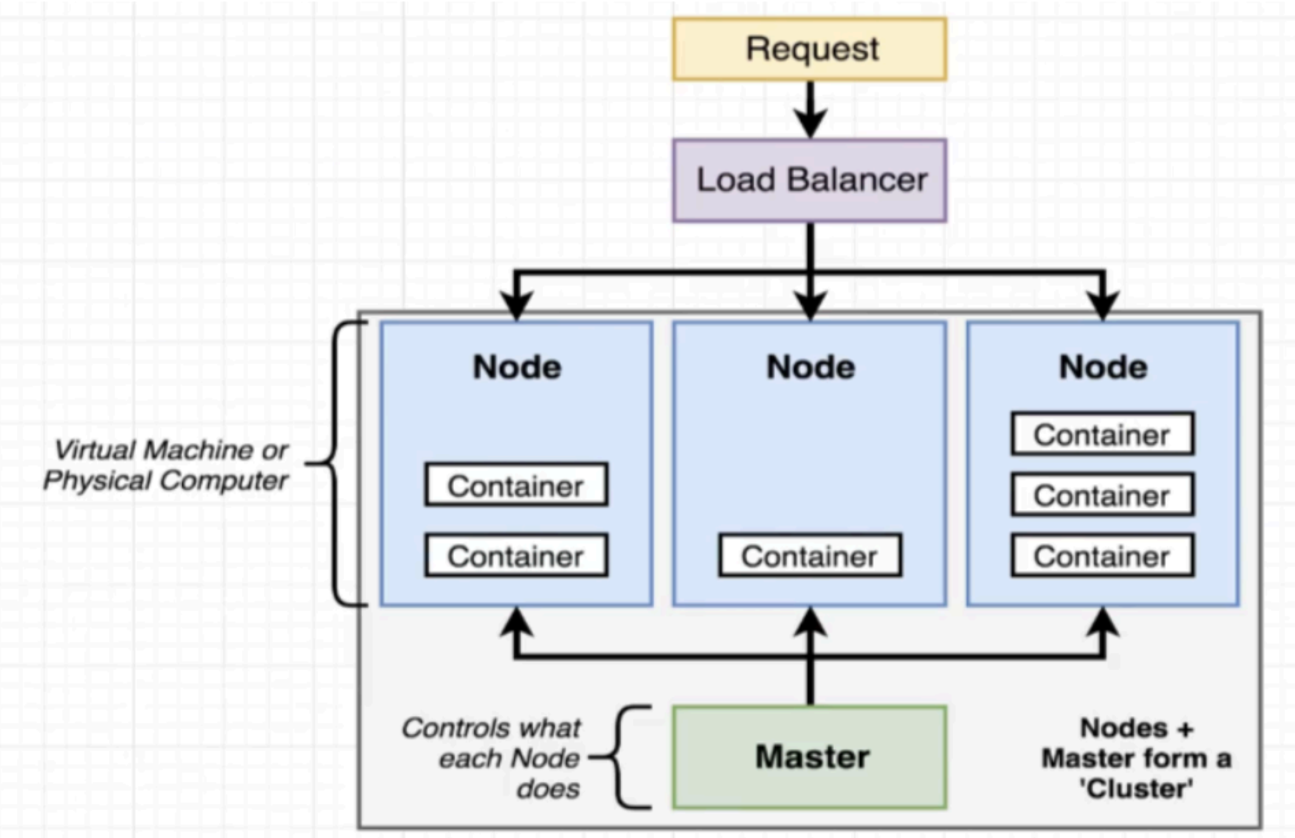
These components run on every worker node and are responsible for running and managing containers:

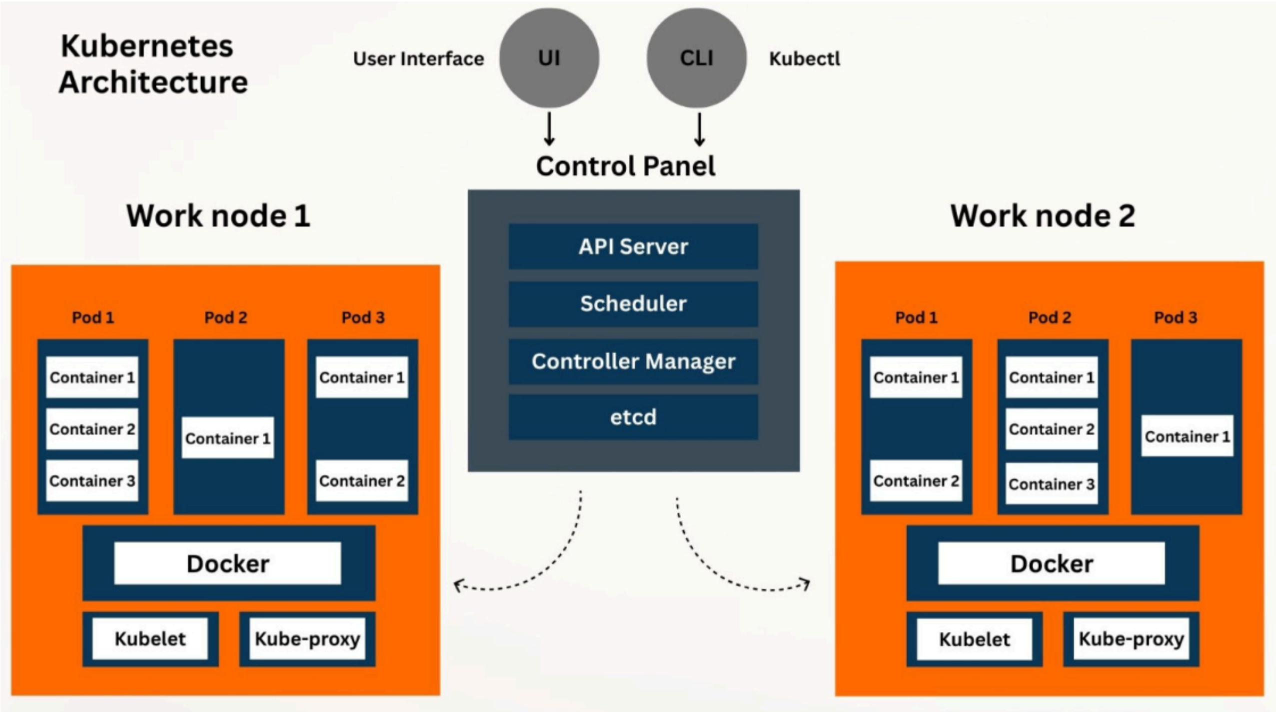
- **Kubelet:**
  - The **primary agent** that runs on each node.
  - It communicates with the Control Plane and ensures that containers described in **Pods** are running and healthy on its node.
  - It reports the node's health and resource utilization back to the master.
- **Container Runtime:**
  - The software responsible for running the containers (e.g., Docker, containerd, CRI-O).

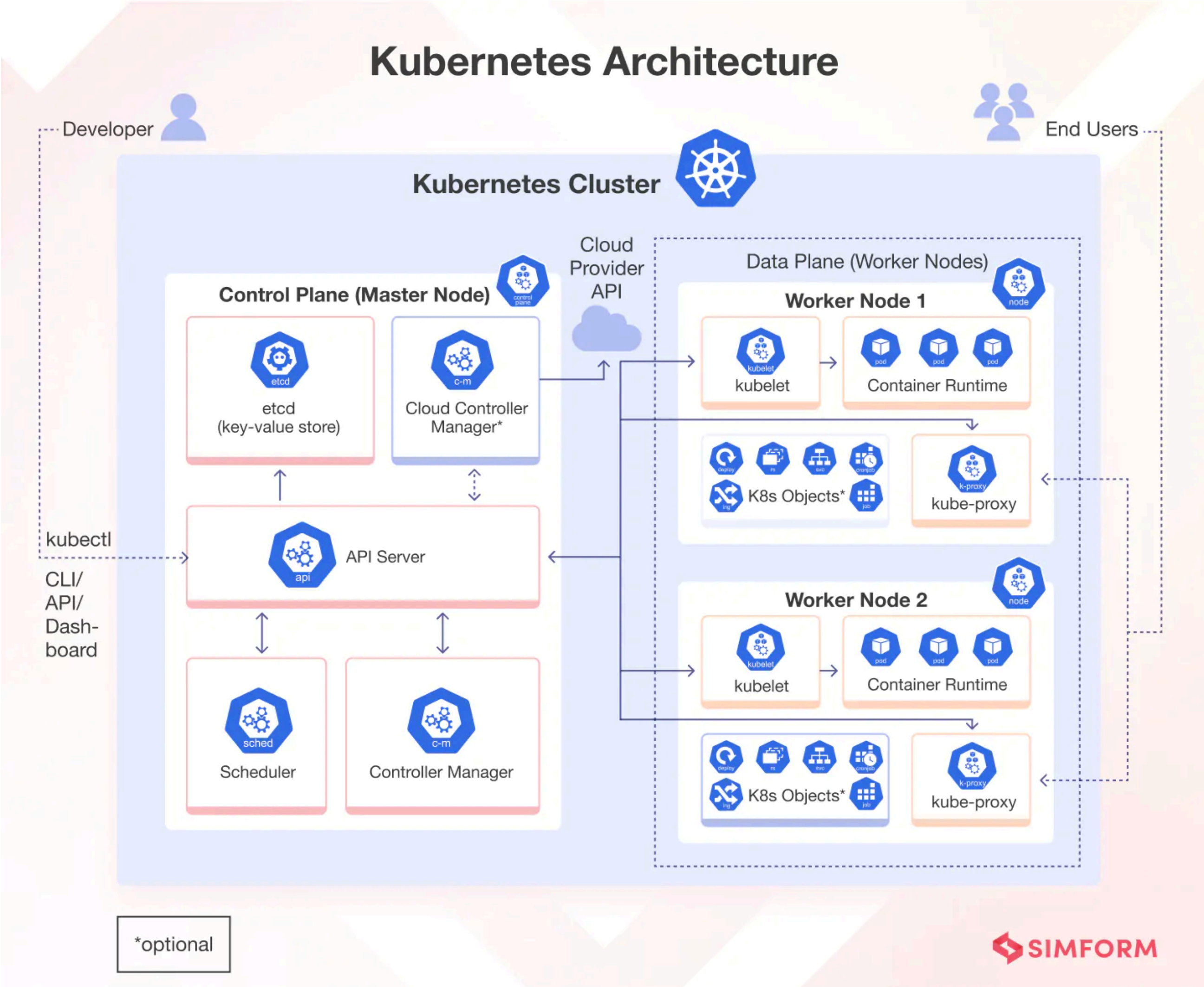
    - The Kubelet interacts with the Container Runtime to start, stop, and manage container lifecycles.
- **Kube-Proxy (`kube-proxy`):**
    - A network proxy that runs on each node.
    - It maintains network rules on the nodes, allowing network communication to your Pods from outside and inside the cluster (e.g., implementing **Services**).

Key Kubernetes Objects (Abstractions)

- **Pod:** The **smallest deployable unit** in Kubernetes. It is a group of one or more containers that are deployed together on the same host and share network and storage resources. A Pod represents a single instance of an application.
- **Deployment:** A declarative way to manage the creation and scaling of **Pods** and **ReplicaSets**. It handles rolling updates and rollbacks, allowing for zero-downtime application updates.
- **Service:** An abstraction that defines a logical set of **Pods** and a policy by which to access them. Services provide a **stable IP address and DNS name** for a set of Pods, enabling internal service-to-service communication and external exposure of the application.




Kubernetes Architecture
User Interface
UI
CLI
Kubectl
Control Panel
API Server
Scheduler
Controller Manager
etcd
Work node 1
Pod 1
Pod 2
Pod 3
Container 1
Container 2
Container 3
Container 1
Container 1
Container 2
Docker
Kubelet
Kube-proxy
Work node 2
Pod 1
Pod 2
Pod 3
Container 1
Container 2
Container 1
Container 2
Container 3
Container 1
Docker
Kubelet
Kube-proxy



Kubernetes vs Docker Swarm vs Other Platforms

Feature	Kubernetes (K8s)	Docker Swarm	Apache Mesos (with Marathon)
Complexity	**High.** Steeper learning curve, more components to manage.	**Low.** Simpler to set up, highly integrated with the Docker ecosystem.	**Medium/High.** Designed to manage a data center's resources, not just containers.
Ecosystem & Community	**Massive.** The industry standard, vast tooling, and active community.	**Moderate.** Good integration with Docker tools, but smaller ecosystem for advanced features.	**Moderate/Niche.** Powerful, but often used for diverse workloads (containers, Big Data, etc.).
Scalability (Max Nodes)	**High.** Can scale to thousands of nodes, designed for massive scale.	**Medium.** Suitable for small to medium clusters.	**Extremely High.** Designed to scale to tens of thousands of nodes (Data Center OS).
Networking	**Advanced.** Supports a wide range of networking models (e.g., CNI), and sophisticated Service objects.	**Simple.** Built-in overlay networking, simpler load balancing.	**Customizable.** Highly flexible resource management.

Feature	Kubernetes (K8s)	Docker Swarm	Apache Mesos (with Marathon)
Auto-Scaling	**Native.** Built-in Horizontal Pod Autoscaler (HPA) and Cluster Autoscaler.	**Limited/External.** Requires manual scaling or third-party tools/scripts.	**Framework-dependent.** Managed via frameworks like Marathon.
Use Case	Complex, large-scale microservices, fluctuating workloads, high customization needs.	Quick deployment, small to medium-sized clusters, simplicity is key, existing Docker user base.	Environments with diverse workloads (Big Data, HPC, containers) and extreme fault-tolerance needs.

When to Use Kubernetes

While powerful, Kubernetes adds an operational overhead. It is a good fit when the added complexity is justified by the application's requirements.

Use Cases Where Kubernetes Excels

- **Microservices Architecture:** When your application is composed of many independent services that need advanced service discovery, load balancing, and independent scaling.
- **High Availability (HA) and Resilience:** For mission-critical applications where downtime must be minimized. K8s' self-healing capabilities are crucial for ensuring application continuity against node or container failure.
- **Complex CI/CD Pipelines:** When you need a reliable, repeatable, and automated deployment platform that supports rolling updates, canary deployments, and blue/green deployments across different environments (Dev, Test, Prod).
- **Fluctuating Workloads:** For applications with unpredictable or periodic traffic spikes (e.g., e-commerce, seasonal events). The **Horizontal Pod Autoscaler (HPA)** can automatically scale up and down to match demand and optimize cloud costs.
- **Cloud and Vendor Portability:** When you need to ensure your application can run consistently across different public cloud providers (AWS, Azure, GCP) or on-premises data centers without significant changes.
- **Multi-Tenancy:** When running applications for multiple customers (tenants) on the same cluster, leveraging **Namespaces** to provide logical isolation and resource quotas.

When to Avoid or Delay Kubernetes

- **Simple, Small Applications:** For a small application (e.g., a single-container website) that doesn't need high availability or complex scaling, the operational overhead of managing a K8s cluster is generally not worth the benefit. Simple solutions like **Docker Compose** or managed services (e.g., AWS Fargate, Google Cloud Run) are better choices.
- **Small Teams/Limited Expertise:** If your team lacks the necessary skills, adopting Kubernetes can significantly slow down development due to the steep learning curve and the complexity of debugging cluster-level issues.

- **Monolithic Applications with No Scaling Needs:** If you have a traditional application that is not broken into microservices and can handle its load on one or two VMs, a traditional deployment model will be simpler and more cost-effective.

https://aws.plainenglish.io/kubernetes-architecture-c93cb9c798d8