Kubernetes

An Overview of Modern Container Orchestration

Abstract

Modern software systems are expected to be highly available, scalable, and resilient while supporting rapid deployment cycles. Although containerization technologies such as Docker simplify application packaging and portability, they do not address the operational challenges of managing applications at scale. Kubernetes has emerged as the industry-standard container orchestration platform by providing automated deployment, scaling, service discovery, load balancing, and self-healing capabilities. This article presents an overview of Kubernetes, its architecture, core components, deployment workflow, and managed Kubernetes services.

1. Introduction

The rapid adoption of microservices and cloud-native architectures has fundamentally transformed the way software applications are designed and deployed. Modern applications often consist of multiple independent services that communicate over a network and must remain available despite hardware failures, software updates, and changing user demand.

Traditional deployment approaches, where applications are installed directly on physical or virtual machines, introduce several operational challenges:

- Environment inconsistencies between development and production

- Limited scalability
- Manual application deployment and monitoring
- Increased downtime during updates
- Difficulty recovering from infrastructure failures

Containerization solved many of these challenges by packaging applications together with their runtime environment. However, managing hundreds or thousands of containers across multiple machines requires an orchestration platform capable of automating operational tasks. Kubernetes was developed to address these challenges.

2. From Containers to Container Orchestration

A container packages an application together with its libraries, dependencies, and runtime environment into a lightweight, portable execution unit. Unlike virtual machines, containers share the host operating system kernel, making them significantly more resource-efficient.

Docker became the most widely adopted container platform because it allows developers to build an application once and execute it consistently across different environments.

Although containers simplify deployment, organizations still face operational questions:

- How should containers be distributed across multiple servers?
- How can failed containers be restarted automatically?
- How can applications scale during periods of increased demand?
- How can traffic be balanced across multiple application instances?
- How can updates be performed without interrupting users?

Container orchestration platforms automate these operational responsibilities. Kubernetes has become the de facto standard for container orchestration due to its scalability, portability, and extensibility.

3. Kubernetes Architecture

A Kubernetes cluster consists of two major components: the Control Plane and Worker Nodes. The diagram below illustrates how these components interact to manage and execute workloads.

Figure 1: Kubernetes Cluster Architecture — Control Plane and Worker Nodes

Control Plane

The Control Plane is responsible for managing the overall state of the cluster. Rather than executing application workloads, it continuously monitors the cluster and makes scheduling and management decisions. Its primary responsibilities include maintaining the desired cluster state, scheduling workloads onto worker nodes, monitoring cluster health, and handling communication with users and external tools.

Component	Responsibility
API Server	Acts as the primary entry point into the Kubernetes cluster. Every operation performed by users, automation tools, or applications is processed through the API Server.
etcd	A distributed key-value database that stores the cluster's persistent configuration and state, including Deployments, Services, Pods, ConfigMaps, and Secrets.
Scheduler	Determines the most suitable Worker Node for newly created Pods based on resource availability, scheduling policies, and node constraints.

Component	Responsibility
Controller Manager	Continuously compares the desired state with the actual cluster state, and automatically takes corrective actions whenever discrepancies occur.

Worker Nodes

Worker Nodes execute the actual application workloads. Each Worker Node contains the following components:

Component	Responsibility
Kubelet	Communicates with the Control Plane and ensures that Pods assigned to the node are running correctly.
Container Runtime	Responsible for pulling container images, creating containers, and managing their execution. Common runtimes include containerd and CRI-O.
Kube Proxy	Manages network communication within the cluster by routing traffic to the appropriate Pods and providing internal load balancing.

4. Core Kubernetes Objects

Kubernetes defines a set of core API objects that describe how applications are deployed, accessed, and managed.

Pod

A Pod is the smallest deployable unit in Kubernetes. It represents one or more containers that share networking and storage resources while operating as a single execution unit. Although Pods may contain multiple tightly coupled containers, most production workloads deploy one application

container per Pod.

Deployment

A Deployment manages the lifecycle of Pods and maintains the desired application state. Instead of manually creating Pods, developers define a Deployment using a YAML configuration file. Kubernetes continuously ensures that the required number of Pod replicas are available, and provides rolling updates, automatic rollback, horizontal scaling, and self-healing.

Service

Pods are ephemeral resources whose IP addresses change whenever they are recreated. A Service provides a stable network endpoint that allows applications to communicate reliably without depending on individual Pod addresses. Services also provide internal load balancing, service discovery, and stable DNS names.

Ingress

Ingress manages external access to applications running inside the cluster. It routes incoming HTTP and HTTPS traffic to the appropriate Services based on host names or URL paths, while providing centralized traffic management.

5. Declarative Configuration

One of Kubernetes' defining characteristics is its declarative approach to infrastructure management. Instead of writing instructions describing how resources should be created, developers describe what the desired infrastructure should look like using YAML manifests. The Control Plane continuously compares this desired configuration with the actual cluster state and automatically performs corrective actions whenever inconsistencies occur.

Example: Deployment Manifest

The following YAML defines a Deployment that runs three replicas of an nginx web server container, with resource limits and a label selector:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
          resources:
            requests:
              memory: "64Mi"
              cpu: "250m"
            limits:
              memory: "128Mi"
              cpu: "500m"
```

Example: Service Manifest

The following YAML defines a Service that exposes the Deployment above on port 80, routing traffic to all Pods with the label app: nginx:

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

Once applied with kubectl apply -f manifest.yaml, Kubernetes takes ownership of ensuring the cluster matches this specification — automatically creating, replacing, or scaling Pods as needed.

6. Deployment Workflow

The deployment lifecycle in Kubernetes follows a well-defined sequence:

1. The developer builds a container image.
2. The image is stored in a container registry.
3. A Deployment manifest is submitted using kubectl apply.
4. The API Server validates and accepts the request.
5. etcd stores the desired cluster configuration.
6. The Controller Manager detects that new Pods must be created.
7. The Scheduler selects suitable Worker Nodes.
8. Kubelet instructs the Container Runtime to pull the required image.
9. The container starts inside a Pod.

10. Services expose the application for internal communication; Ingress manages external access.

This declarative workflow enables Kubernetes to automate deployment, scaling, recovery, and networking without manual intervention.

7. Managed Kubernetes Services

Although Kubernetes can be deployed and managed manually, maintaining the Control Plane requires significant operational expertise. Cloud providers therefore offer managed Kubernetes services that abstract much of the infrastructure complexity.

Cloud Provider	Managed Service
Microsoft Azure	Azure Kubernetes Service (AKS)
Amazon Web Services	Elastic Kubernetes Service (EKS)
Google Cloud	Google Kubernetes Engine (GKE)

In managed environments, the cloud provider operates the Control Plane, performs upgrades, monitors cluster health, and manages infrastructure availability. Developers primarily focus on building applications and deploying workloads.

8. Conclusion

Kubernetes has become the foundation of modern cloud-native application deployment by providing an automated platform for container orchestration. Its architecture enables applications to remain highly available, fault tolerant, and scalable while reducing operational overhead through automation.

By combining containerization with declarative infrastructure management, Kubernetes simplifies the deployment of distributed applications across on-premises and cloud environments. As organizations continue adopting microservices and DevOps practices, Kubernetes remains one of the most important technologies for building resilient and production-ready software systems.

Revision #2
Created 2026-07-02 11:13:05 UTC by Anagha N
Updated 2026-07-02 13:36:11 UTC by Anagha N