Date: 25.06.2025

Kubernetes

Contents

1. Introduction

- Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services that facilitates both declarative configuration and automation.
- Kubernetes is a platform that automates the deployment, scaling, and management of containerized applications.
- Key things Kubernetes does for you -
  - Deployment: Launches and manages containers for your apps.
  - Scaling: Adds or removes copies of your app as needed.
  - Self-healing: Restarts or replaces containers if they fail.
  - Rolling Updates/Rollbacks: Updates your app with zero downtime and can revert if something goes wrong

2. Declarative Model of Kubernetes

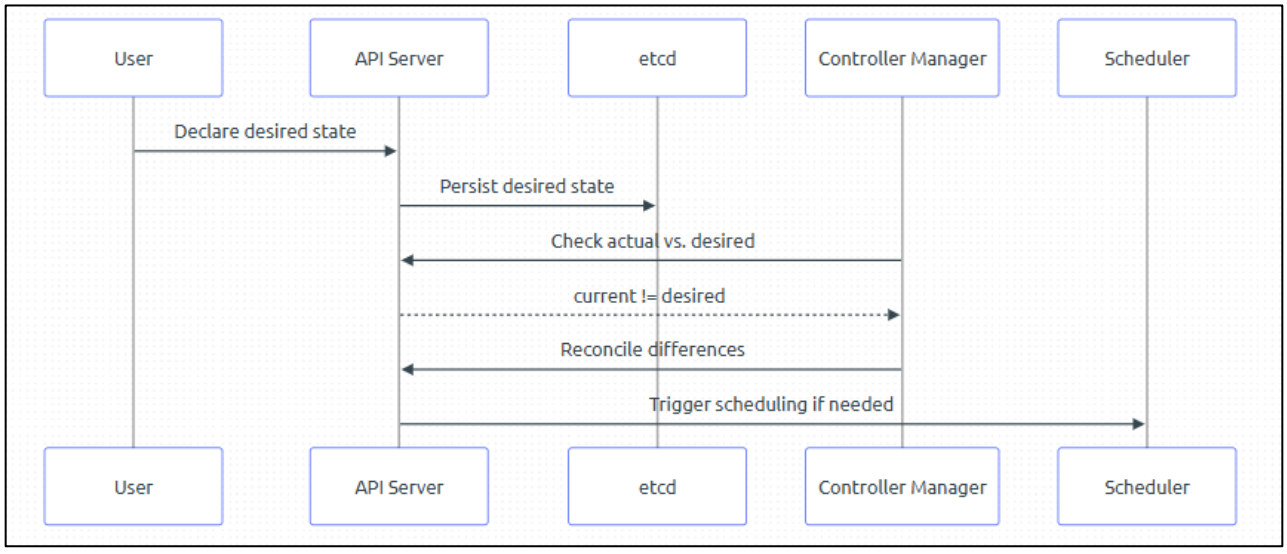
Kubernetes uses a "declarative" approach: you describe how you want your system to look, and Kubernetes works to make it so—automatically.

Three key ideas:

1. Observed State: What's actually running right now.
2. Desired State: What you want running (defined in YAML or JSON).
3. Reconciliation: Kubernetes constantly checks and adjusts to make observed = desired.

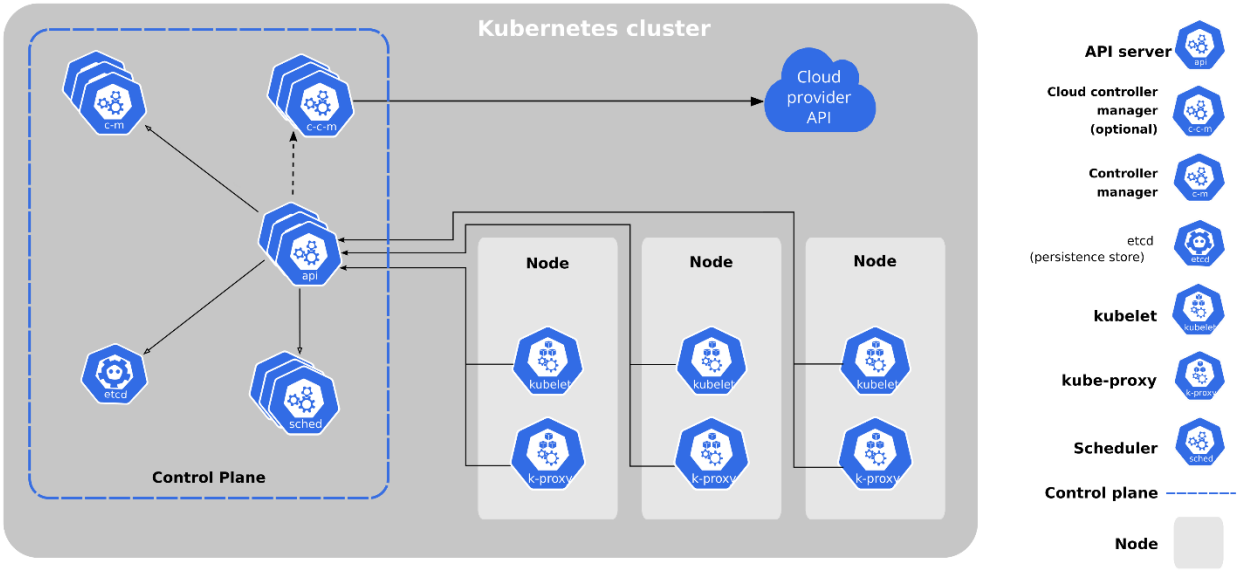
How it works:

- You tell Kubernetes (with kubectl or a YAML file) what you want.
- Kubernetes saves this in its database (etcd).
- Controllers keep checking: does reality match what you asked for?
- If not, Kubernetes takes action to fix it.



3. Core Components

A Kubernetes cluster consists of a control plane plus a set of worker machines, called nodes that run containerized applications. Every cluster needs at least one worker node in order to run Pods.



a. Control Plane Components

Manage the overall state of the cluster.

Component	Description
`kube-apiserver`	The core component server that exposes the Kubernetes HTTP API.
`etcd`	Consistent and highly-available key value store for all API server data.
`kube-scheduler`	Looks for Pods not yet bound to a node, and assigns each Pod to a suitable node.
`kube-controller-manager`	Runs controllers to implement Kubernetes API behavior.
`cloud-controller-manager`	Integrates with underlying cloud provider(s). *(optional)*

b. Node Components

Run on every node, maintaining running pods and providing the Kubernetes runtime environment.

Component	Description
`kubelet`	Ensures that Pods are running, including their containers.
`kube-proxy` *(optional)*	Maintains network rules on nodes to implement Services.
`Container runtime`	Software responsible for running containers.

c. Addons

Addons extend the functionality of Kubernetes.

Component	Description
`DNS`	For cluster-wide DNS resolution.
`Web UI (Dashboard)`	For cluster management via a web interface.
`Container Resource Monitoring`	For collecting and storing container metrics.
`Cluster-level Logging`	For saving container logs to a central log store.

4. Workloads

A workload is an application running on Kubernetes.

a. Pods

A Pod is the smallest unit you can deploy in Kubernetes. It wraps one/more containers that:

- Share the same network (IP and ports)
- Can share storage
- Are managed as a single unit

Most Pods have just one container, but sometimes you need tightly-coupled containers together (like a main app and a helper/sidecar).

Key traits:

- Pods are short-lived and disposable.
- If a Pod crashes, it won't restart by itself (unless managed by something higher-level).
- Pods stick to the Node they're scheduled on until they die.

Use Pods directly for quick experiments or debugging, One-off tasks (but consider Job for this), Learning and testing

b. Deployments

A Deployment is a manager for your Pods. It ensures the right number are running, replaces them if they fail, and allows you to update your app with zero downtime.

With Deployments, you get:

- Automated management of Pod replicas
- Rolling updates and safe rollbacks
- Automatic replacement of failed Pods
- Easy version control for your app

You describe your desired state in a YAML file, and Kubernetes makes it happen.

So while a Pod is the worker, the Deployment manages the workforce and ensures consistency, scalability, and resiliency over time.

A Deployment always manages Pods – you typically never run a Deployment without Pods.

When you apply a Deployment spec:

1. Kubernetes creates a ReplicaSet.
2. The ReplicaSet creates the desired number of Pods.
3. If any Pod dies, the ReplicaSet spawns a replacement.

Use Deployments for production workloads, when you want automatic healing, scaling, and updates, anything that needs to run reliably over time

c. DaemonSet

DaemonSets are designed to manage the deployment of Pods across all nodes in a cluster. They ensure that a specific Pod is running on each node, making them ideal for system-level applications.

DaemonSets are commonly used for -

- Log Collection: Deploying log collection agents on each node.
- Monitoring: Running monitoring agents to collect metrics from nodes.
- Networking: Managing network services like DNS or proxy servers.

Key Features -

- Automatic Updates: Automatically adds Pods to new nodes when they are added to the cluster.
- Selective Deployment: Can be configured to deploy Pods only to specific nodes using node selectors.
- Rolling Updates: Supports rolling updates to update Pods without downtime.

DaemonSets can be managed using various Kubernetes features -

- Node Selectors: Control which nodes a DaemonSet's Pods are scheduled on.
- Tolerations: Allow DaemonSet Pods to run on nodes with specific taints.
- Update Strategy: Configure rolling updates to minimize disruption.

d. Kubernetes Operators

Operators are software extensions that use custom resources to manage applications and their components. They automate tasks beyond the capabilities of standard Kubernetes resources, following Kubernetes principles like the control loop.

Purpose of Operators

- Installation: Deploying and configuring applications.
- Management: Managing runtime configurations.
- Scaling: Adjusting resources based on workloads.
- Healing: Detecting and recovering from failures.
- Upgrades: Updating applications to new versions.

Benefits of Using Operators

- Consistency: Provides a consistent way to manage applications.
- Automation: Reduces manual intervention.
- Scalability: Manages resources efficiently.

Advanced Operator Features

- Event Handling: Operators can respond to Kubernetes events to maintain desired state.
- Custom Metrics: Use custom metrics to make informed scaling decisions.
- Backup and Restore: Implement application-specific backup and restore logic.

e. Custom Resource Definitions

CRDs allow you to define custom resources within the Kubernetes API, enabling the management of application-specific data and configurations.

Benefits of Using CRDs

- Custom Resources: Tailor resources to your application's needs.
- Declarative Management: Use Kubernetes' API for management.
- Integration: Seamlessly integrate with Kubernetes tools.

Advanced CRD Features

- Schema Validation: Define validation rules for custom resources to ensure data integrity.
- Versioning: Manage different versions of CRDs to support application evolution.
- Subresources: Use subresources like status and scale for additional functionality.

f. Helm

Helm is the package manager for Kubernetes.

Helm lets you define, install, and upgrade Kubernetes applications using packages called charts (a package of Kubernetes resources).

Benefits of Using Helm

- Simplifies Deployment: Bundle all your resources in one chart for easy deployment.
- Versioning: Upgrade and roll back apps with a single command.
- Reuse: Share charts across teams and environments.
- Customization: Use templates and values to adapt to any setup.
- Dependency Management: Charts can depend on other charts.

Helm Architecture

- Helm Client: Command-line tool for managing charts.
- Helm Server (Tiller): Only in Helm v2. In Helm v3+, the client talks directly to the Kubernetes API server (no Tiller).

How Helm Works

- Charts: Collections of files describing Kubernetes resources.
- Values Files: Override default settings for different environments.
- Templates: Dynamically generate manifests.
- Releases: Each deployment of a chart is a release.
- Repositories: Collections of charts you can share and reuse.

Advanced Helm Features

- Hooks: Allow you to run scripts at specific points in a release lifecycle.
- Lifecycle Management: Manage the lifecycle of applications with upgrade and rollback capabilities.
- Managing Dependencies: Use the requirements.yaml file to manage chart dependencies.

g. Jobs

A Job runs a Pod (or Pods) to completion.

Perfect for:

- One-time tasks
- Batch processing
- Migrations or post-deployment hooks

Kubernetes guarantees the Job runs to completion – even if a Pod crashes or a node fails, a new Pod will be scheduled.

h. CronJobs

A CronJob runs Jobs on a repeating schedule, just like Linux cron.

Use CronJobs for:

- Periodic cleanup tasks
- Scheduled reports
- Time-based batch jobs

CronJobs use the same jobTemplate spec as regular Jobs.

Field	Meaning
Minute	0–59
Hour	0–23
Day of Month	1–31
Month	1–12
Day of Week	0–6 (Sun=0)

i. Init Containers

Init containers run before regular containers in a Pod start. They're designed to perform setup tasks that need to complete before your main application launches — like setting config values, creating folders, or waiting for external services.

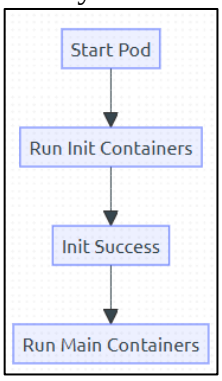
Why Use Init Containers?

- Wait for a database or service to become available
- Download configuration or secrets from an external source
- Run database migrations or setup scripts
- Perform one-time initialization logic before the main app starts

Init containers always run to completion before normal containers are started. If an init container fails, the Pod restarts and tries again.

Use them for waiting on services, running migrations, or any pre-launch logic.
They run sequentially and must succeed before your main containers start.

Lifecycle Flow



- If any init container fails, Kubernetes restarts the entire Pod.
- Init containers run **sequentially**, not in parallel.
- Once completed, they're never run again.

5. State and Config

a. ConfigMaps (Non–sensitive Data)

A ConfigMap is a key-value store for plain-text configuration.
Use it for:

- Environment settings (like LOG_LEVEL, API_BASE_URL)
- Hostnames, ports, feature flags
- Complete config files or CLI arguments

b. Secrets (Sensitive Data)

Secrets are also key-value stores – but for private data:

- Passwords, tokens, API keys
- SSH keys or TLS certs
- Docker registry credentials

Kubernetes encodes all Secret values in base64 (for transport, not real security).

c. Storage

Type	Description
emptyDir	Temporary, pod-level storage. Deleted when Pod is gone.
hostPath	Mounts a path on the host node. Avoid in production.
configMap / secret	Used for injecting configs/secrets into containers.
persistentVolume	Abstracts physical storage (EBS, NFS, GCE PD, etc.).
volumeClaimTemplate	Used by StatefulSets to dynamically provision volumes.

i. Volumes

Basic volumes are attached to a Pod spec and live as long as the Pod.
Use emptyDir for scratch space or caching — not persistent data.

ii. PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs)

To persist data beyond the life of a Pod, use the PV + PVC model:

- A PV is a piece of actual storage (disk, NFS, etc.)
- A PVC is a user's request for storage
- Kubernetes binds them together dynamically

Modes

Volume Modes	Access Modes
• Filesystem (default): mounts as a directory • Block: exposes the raw device	• ReadWriteOnce: one node read/write • ReadOnlyMany: multiple nodes read-only • ReadWriteMany: shared read/write (NFS, etc.)

iii. StorageClasses

A StorageClass defines how storage should be provisioned dynamically.
The cluster admin defines StorageClasses; PVCs can request one by name.

Dynamic vs Static Provisioning

- Dynamic: PVC automatically provisions a volume using a StorageClass.
- Static: Admin manually creates PVs, and users bind to them with matching PVCs

d. StatefulSets

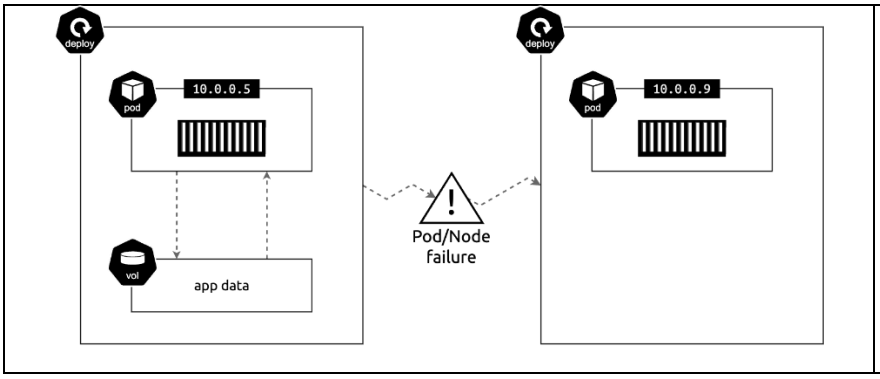
A StatefulSet is a Kubernetes controller for running stateful apps—apps that need each Pod to keep its identity and storage, even if rescheduled. Think databases, message queues, or anything that can't just be replaced with a blank copy.

Use a StatefulSet when your app needs:

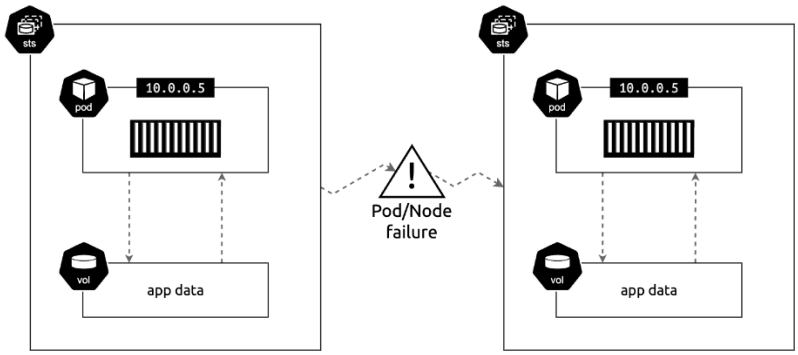
- Stable network identity (like pod-0, pod-1)
- Persistent storage per Pod that sticks around if the Pod is rescheduled
- Ordered startup, scaling, and deletion

Examples: Databases (PostgreSQL, Cassandra), Zookeeper, Kafka, etc.

StatefulSets guarantee identity and storage for each Pod, while Deployments just care about keeping the right number of Pods running (not which is which).



Pod reschedule = new IP
Broken volume mount

	Pod is recreated with the same IP Same volume

Key Features

Feature	Deployment	StatefulSet
Pod name	Random (e.g., pod-abc123)	Stable (e.g., web-0, web-1)
Pod start/delete order	Any	Ordered
Persistent Volume Claim	Shared/ephemeral	One per Pod
DNS hostname	Random	Stable via headless service

Networking & DNS
Pods in a StatefulSet get predictable hostnames which is enabled by the headless Service

Volume Behaviour
Each Pod gets its own PVC and these volumes are retained even if the Pod is deleted.

e. Resource requests and Limits

Kubernetes lets you control how much CPU and memory each container is guaranteed and allowed to use. This is done through resource requests and limits in the container spec.

Term	Purpose	Scheduler Uses?	Enforced at Runtime?
requests	Minimum resources guaranteed to a container	Yes	No
limits	Maximum resources a container can use	No	Yes

- Requests are used during scheduling. Kubernetes places Pods on nodes that can satisfy their requested resources.
- Limits prevent a container from exceeding a set threshold.

Why It Matters

- Too low requests → Your Pod may get scheduled on a crowded node and experience performance issues.
- No limits → A container can consume all resources and cause noisy neighbor problems.
- Too low limits → Can result in OOMKills or throttled CPU.

CPU Behavior

- If a container exceeds its CPU limit, it is throttled—not killed.
- If you don't set a limit, the container may consume all available CPU.

Memory Behavior

- If memory usage exceeds the limit, the container is killed with an OOMKill (Out of Memory).
- Kubernetes does not restart it unless it's part of a higher-level controller (like a Deployment).

f. Environment Variables

Kubernetes allows you to configure runtime behavior of containers using environment variables, and to monitor their health using liveness and readiness probes. These features are essential for building reliable, configurable, and observable applications in the cluster.

You can pass key-value pairs into containers using environment variables. These can be hardcoded, referenced from ConfigMaps, Secrets, or even dynamically derived from field references.

Static Environment Variables	ConfigMap	

`env:`
`  - name: LOG_LEVEL`
`  value: "debug"`
	`envFrom:`
	`  - configMapRef:`
	`  name: app-config`	

Individual Keys	Secret	Pod Metadata

`env:`
`  - name: APP_PORT`
`  valueFrom:`
`  configMapKeyRef:`
`  name: app-config`
`  key: port`
	`env:`
	`  - name: DB_PASSWORD`
	`  valueFrom:`
	`  secretKeyRef:`
	`  name: db-secret`
	`  key: password`
		`env:`
		`  - name: POD_NAME`
		`  valueFrom:`
		`  fieldRef:`
		`  fieldPath: metadata.name`

g. Probes

Kubernetes uses probes to check if a container is:

- Alive (liveness probe): Whether the app should be restarted
- Ready (readiness probe): Whether the app is ready to receive traffic
- Started (startup probe): Whether the app has finished starting up

Each probe runs a check (HTTP request, TCP socket, or command) and takes action based on success or failure.

Probe	Description
Liveness Probe	Restarts the container if the probe fails repeatedly.
Readiness Probe	Used to signal when the container is ready to receive traffic. If the probe fails, the Pod is removed from Service endpoints.
Startup Probe	Useful for applications that take a long time to initialize. Prevents premature liveness failures during startup.

6. Cluster

a. Namespace

Namespaces in Kubernetes allow you to divide cluster resources between multiple users or teams. They provide logical isolation and help with multi-tenancy, access control, and resource management.

Namespaces are useful when:

- You need to isolate environments (e.g., dev, staging, prod)
- You want to enforce resource quotas and limits
- You want to implement RBAC per team or application

Viewing Namespaces

- kubectl get namespaces
- kubectl get ns

Creating a Namespace

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev-team
```

Apply it:

```
kubectl apply -f namespace.yaml
```

Using Namespaces with kubectl

```
kubectl get pods -n dev-team
kubectl create deployment nginx
--image=nginx -n dev-team
```

To temporarily switch namespace context:

```
kubectl config set-context --current
--namespace=dev-team
```

Default Namespaces

Namespace	Purpose
default	Used when no other namespace is specified
kube-system	Kubernetes control plane components (DNS, scheduler)
kube-public	Readable by all users, often used for public bootstrap
kube-node-lease	Heartbeats for node status

Namespaced vs Cluster-Scoped Resources

Namespaced	Cluster-Scoped
Pods, Deployments, PVCs	Nodes, PersistentVolumes
ConfigMaps, Secrets	Namespaces, CRDs
Services	StorageClasses, RBAC Roles

Resource Quotas and Limits

You can enforce limits on namespaces using:

ResourceQuota	Caps total resources in the namespace
LimitRange	Sets default limits per Pod/container

Cleanup

To delete a namespace and everything inside it:

```
kubectl delete namespace dev-team
```

b. Kubernetes API

The Kubernetes API is the primary interface to interact with your cluster.
It's declarative, versioned, extensible, and serves as the backbone of the control plane.

At its core, the API is a RESTful interface that lets you manage API objects such as:

- Pods
- Deployments
- Services
- ConfigMaps
- Namespaces
- Custom Resources (via CRDs)

Everything in Kubernetes — from kubectl to the scheduler — interacts with the API server.

Anatomy of a Kubernetes Object
Every resource in Kubernetes follows a common structure:

```
apiVersion: apps/v1        # API group + version
kind: Deployment           # Type of object
metadata:
  name: my-app
  namespace: default
spec:                      # Desired state (defined by user)
  replicas: 2
status:                    # Actual state (set by the system)
  replicas: 2
```

Field	Description
`apiVersion`	Group/version the object belongs to
`kind`	The object type (e.g., Pod, Service)
`metadata`	Info like name, namespace, labels, annotations
`spec`	The desired configuration (user-defined)
`status`	The actual observed state (set by controllers)

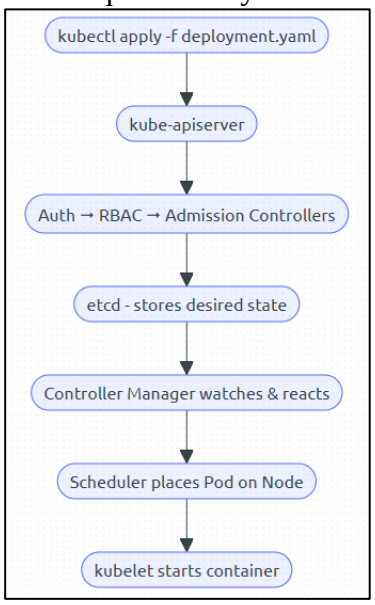
API Server Role

The kube-apiserver is the front door to your cluster. It handles:

- All incoming requests from users, kubectl, controllers, and web UIs
- Validation of requests and schemas
- Authentication & Authorization
- Admission control
- Persisting cluster state in etcd

It's stateless and horizontally scalable — multiple API server instances can run behind a load balancer.

API Request Lifecycle



API Groups & Versions

Kubernetes organizes APIs into groups and versions.

Path Prefix	Group Name	Used For
`/api/v1`	Core (no group name)	Pods, ConfigMaps, Services
`/apis/apps/v1`	apps group	Deployments, StatefulSets
`/apis/batch/v1`	batch group	Jobs, CronJobs
`/apis/networking.k8s.io/v1`	Networking group	Ingress, NetworkPolicies
`/apis/custom.group/v1`	Custom Resource group	CRDs, Operators

You can discover all groups and versions with:

- kubectl api-versions
- kubectl api-resources

CRUD via kubectl

kubectl is a CLI wrapper over raw HTTP requests to the API. Examples:

Action	HTTP Equivalent
`kubectl get pods`	GET /api/v1/namespaces/default/pods
`kubectl apply -f file.yaml`	PATCH or POST to relevant endpoint
`kubectl delete pod nginx`	DELETE /api/v1/namespaces/default/pods/nginx

Declarative vs Imperative

- Imperative: You tell Kubernetes how to do something (kubectl run, kubectl expose)
- Declarative: You define the desired state, and the system reconciles (kubectl apply -f)

Kubernetes is fundamentally declarative — the controller manager continually works to match the actual state to the desired state.

Working with the API Directly

- Enable the API proxy: `` `kubectl proxy` ``
- Browse live cluster data in your browser: `` `http://localhost:8001/api` ``

- Also try: `curl http://localhost:8001/api/v1/namespaces/default/pods`

Custom Resource Definitions (CRDs)
CRDs allow you to extend the Kubernetes API with your own types.

API Security Flow
Every API request goes through:

- Authentication – Who is making the request?
- Authorization (RBAC) – Are they allowed to perform this action?
- Admission Controllers – Mutate or validate the request
- Validation – Is the object schema correct?
- Persistence – If approved, store in etcd

c. Quotas and Limits

In a multi-tenant Kubernetes environment, you need to make sure no single team or workload can hog all the resources.
Kubernetes provides two key tools for this: ResourceQuotas and LimitRanges.
These help admins enforce fair resource allocation, cost controls, and capacity planning.

i. ResourceQuota

A ResourceQuota sets a hard cap on the total resource usage (CPU, memory, object counts, etc.) within a namespace.
If the sum of all Pods in the namespace exceeds the quota, new requests are denied.

ii. LimitRange

A LimitRange sets default values and upper/lower bounds for container-level resource usage within a namespace.
It ensures developers don't accidentally omit or misuse resource definitions.
A guardrail to prevent containers from consuming too much by default.

When to Use Quotas vs LimitRanges

Feature	ResourceQuota	LimitRange
Scope	Namespace-wide	Per container
Controls total usage	Yes	No
Sets defaults	No	Yes
Enforces boundaries	Yes (hard enforcement)	Yes (via defaults and min/max)
Common Use	Multi-team environments	Developer guardrails

Monitor usage with `kubectl describe quota` or metrics dashboards.

7. Networking

a. Overview

Networking in Kubernetes is simple on the surface, but powerful under the hood.

Core Principles of Kubernetes Networking

- Each Pod gets a unique IP
- No NAT between Pods
- All containers within a Pod share the same network namespace
- All Pods can reach each other
- Flat network model (no IP masquerading between Pods)
- Services provide stable access (endpoints) to Pods
- Pods are ephemeral — Services give them a consistent IP + DNS name

Network Abstraction Layers

Layer	Purpose
Pod Network	Every Pod gets an IP, routable in-cluster
Service	Provides a stable endpoint for Pod groups
Ingress	Exposes HTTP/S services externally
NetworkPolicy	Controls traffic between Pods (optional)

DNS in Kubernetes

Kubernetes includes built-in DNS resolution for:

- Services: `my-service.my-namespace.svc.cluster.local`
- Pods (not recommended for direct use)

DNS is powered by CoreDNS by default, running in the kube-system namespace.

```
nslookup my-service.default.svc.cluster.local
```

Pod-to-Pod Communication

- All Pods are routable via their internal IP addresses
- No need for manual port forwarding
- Backed by a Container Network Interface (CNI) plugin (e.g., Calico, Flannel)

Service Types

Type	Description
ClusterIP	default; internal-only
NodePort	exposes on every node
LoadBalancer	cloud provider external IP
ExternalName	DNS alias

b. Services

Pods are short-lived—they can appear and disappear at any time. A Service gives you a stable way to talk to a group of Pods, no matter how often those Pods restart or move.

What Is a Service?

A Kubernetes Service is like a switchboard operator for your Pods:

- Selects a group of Pods (using labels)
- Gives them a stable IP and DNS name
- Forwards traffic to the right Pods, even as they change

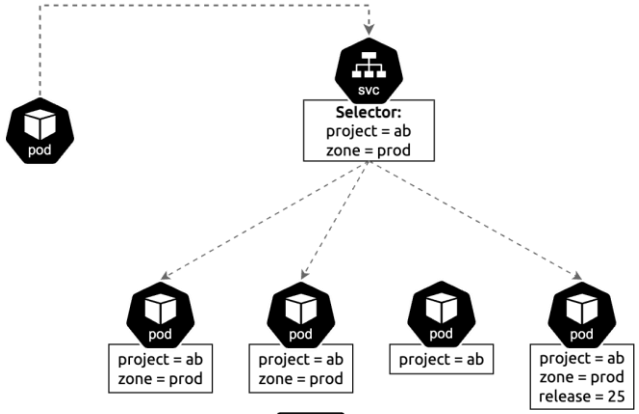
i. ClusterIP (default)

A ClusterIP Service is for internal communication only.

- Internal IP address (e.g., 10.x.x.x)
- DNS: web.default.svc.cluster.local
- Default Service type

Use When:

- Apps need to talk to each other inside the cluster (e.g., frontend ↔ backend)
- No external access needed



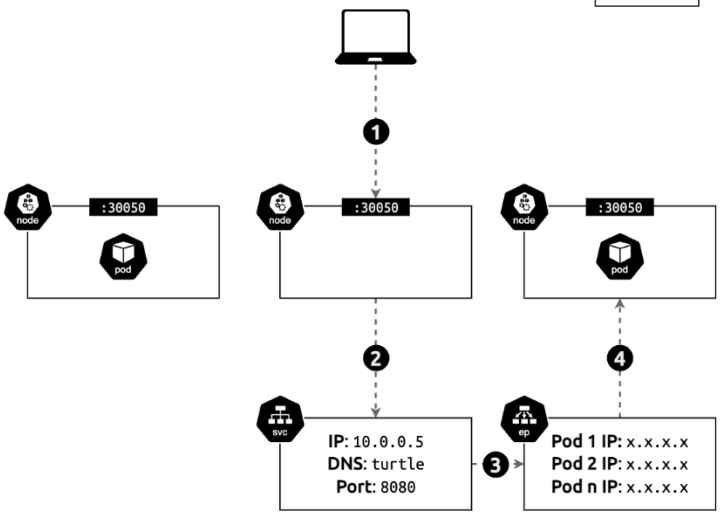
ii. NodePort

A NodePort Service lets people outside your cluster reach your app using a static port on every node.

- Uses each node's IP + port (range: 30000–32767)
- Forwards traffic from node to the right Pods

Use When:

- Testing external access without a LoadBalancer
- You don't have a cloud provider (e.g., on-prem clusters)



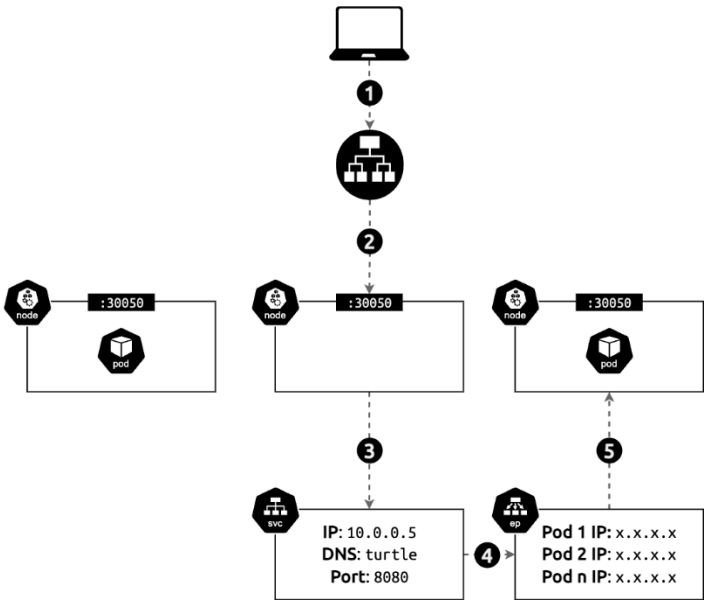
iii. LoadBalancer

A LoadBalancer Service provisions an external cloud load balancer (if supported by your environment).

- Only works with cloud providers (GCP, AWS, Azure)
- Assigns a public IP and balances across backing Pods
- Combines NodePort + external LB behind the scenes

Use When:

- You want public access to your app in a cloud environment
- You need external DNS + SSL termination (with Ingress)



iv. ExternalName (Special Case)

Maps a Kubernetes Service to an external DNS name.

- No selectors or backing Pods
- Useful for referencing external databases, APIs, etc.

Summary Table

Type	Visibility	Use Case	Requires Cloud
ClusterIP	Internal only	Pod-to-Pod communication	No
NodePort	Exposes on node IP	Direct external access via port	No
LoadBalancer	External IP	Cloud load balancer with public IP	Yes
ExternalName	DNS redirect	External services via DNS	No