KodeKloud

Kubernetes Made Easy

— Makes Complex Stuff Feel... Simple

Disclaimer

Kubernetes®, Docker®, Amazon Web Services (AWS)®, Microsoft Azure®, Google Cloud Platform (GCP)™, and all other trademarks, service marks, and brand names mentioned in this publication are the property of their respective owners. The use of these names, logos, and marks in this eBook is strictly for educational and informational purposes to assist readers in learning about Kubernetes and related technologies.

KodeKloud is neither affiliated with nor endorsed by the Kubernetes project, the Cloud Native Computing Foundation, or any other trademark holder referenced in this eBook. The inclusion of any third-party trademarks does not imply any partnership, sponsorship, or endorsement by those entities.

All product names, logos, and brands are used for identification purposes only. All rights, titles, and interests in these trademarks remain the property of their respective owners.

This eBook is intended for informational and educational use only. The authors and publishers make no representations or warranties with respect to the accuracy or completeness of the contents of this eBook and disclaim any implied warranties of merchantability or fitness for a particular purpose. The information provided is subject to change without notice.

Table of Contents

SECTION 1: WHAT IS KUBERNETES?

Let’s Not Pretend This Is Easy — But It’s Learnable

> **Kubernetes** is an open-source platform that manages containerized applications.

It automates:

- Deploying applications
- Scaling applications
- Keeping applications running
- Updating applications
- Recovering from failure

What You Should Know Before Moving On

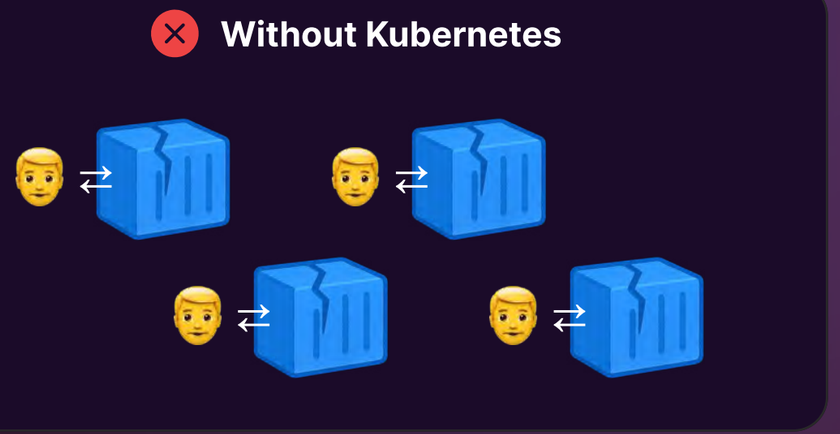
Let’s check your context. This book assumes:

- ✓ You understand what a **container** is (e.g., a Docker container)
- ✓ You've tried running or building containers on your machine
- ✓ You’re looking to manage containers in a more efficient, automated way.

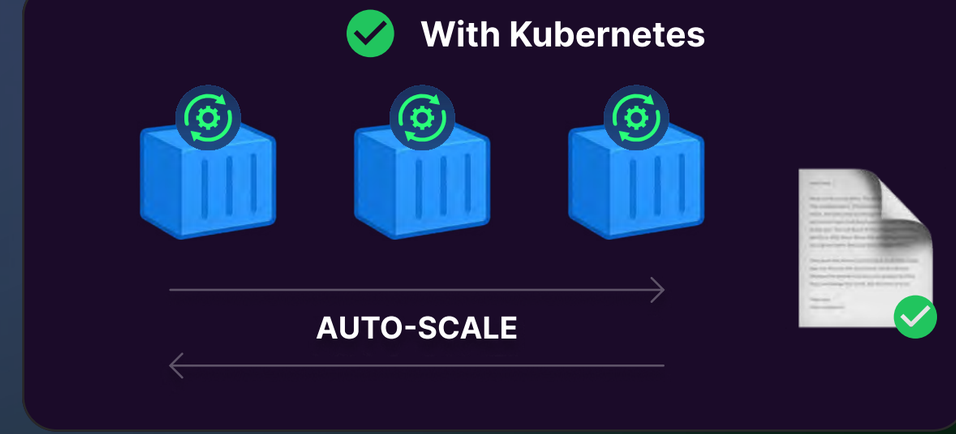
If that’s true — you’re ready for Kubernetes.

> **Not quite there yet?** No worries—you’ll find some quick KodeKloud resources waiting for you at the end of this lesson to get you confidently up to speed.

What Kubernetes Solves



- You run containers manually
- You write custom scripts to scale apps
- You monitor and restart things yourself
- You handle network connections container by container



- Containers are deployed with a command or a YAML file
- Apps scale up or down automatically
- Kubernetes monitors and restarts failed apps
- Networking is handled at the cluster level

What Kubernetes Actually Does

- Starting apps ✅
- Scaling based on traffic ✅
- Monitoring and restarting ✅
- Updating apps safely ✅
- Connecting services ✅

> **Kubernetes** is also often called **K8s**.
> Why? The “K” is the first letter, the “s” is the last letter, and there are 8 letters in between.
> 👉 Even the name sounds cool from the start!

SUMMARY

> Kubernetes = container management done declaratively and automatically

> You give Kubernetes the desired state of your app (via YAML or `kubectl`), and it works continuously to make sure reality matches that state.

Quick Check: Is Kubernetes For You?

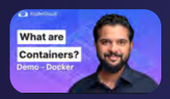
Check everything that applies:

- [ ] I run multiple services or microservices
- [ ] I want to reduce downtime during deployments
- [ ] I want automatic scaling or self-healing
- [ ] I want to update my app without editing dozens of scripts

✅ If you checked two or more, you’re building something Kubernetes can manage well.

You Might Also Like

Why Kubernetes?
Understand why Kubernetes exists and when to actually use it.



What Are Containers, Really?
A short, beginner-friendly video to demystify containers.

Learn Docker in Just 2 Hours
A full hands-on tutorial — no fluff, just what you need.

Practice Makes Perfect: Docker Free Labs
Get real hands-on experience — the kind that sticks.

SECTION 2: CORE CONCEPTS YOU MUST KNOW FIRST

Before You Even Type `kubectl` … Know These.

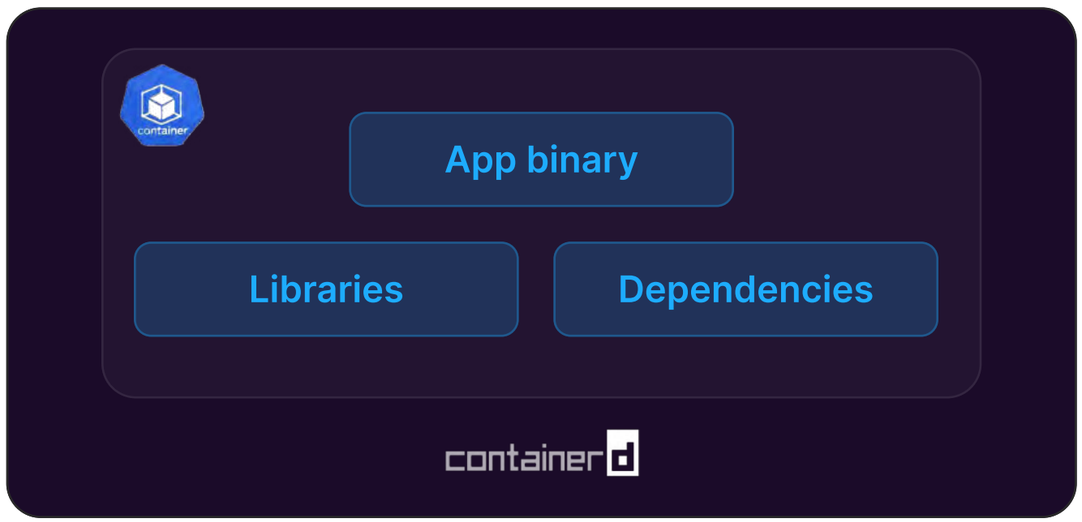
- **You don’t need to learn everything about Kubernetes to get started.**
- **Just understand these five concepts first — they’re the building blocks of everything you’ll learn next.**

Container

You already know this. Let’s solidify it.

- A **container** is an isolated environment that runs your application.
- It includes the application binary, config files, dependencies, and runtime.
- Containers are started by a **container runtime**. The most common runtime today is `containerd` .

📌 Containers are not managed directly in Kubernetes. Kubernetes manages **Pods** — which run containers.



Pod

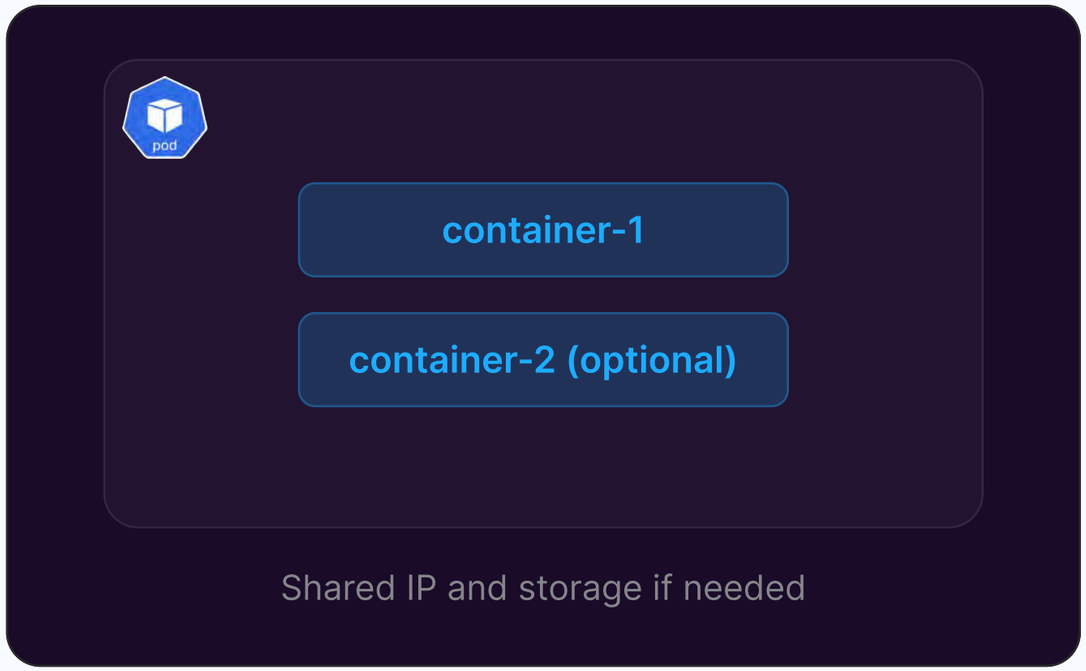
A **Pod** is the smallest deployable object in Kubernetes.

Each Pod:

- Contains **at least one container**
- Shares an IP address and port space across its containers
- Can be restarted or replaced automatically by Kubernetes

Pods are short-lived. When a Pod crashes or is deleted, Kubernetes may replace it with a new Pod.

📌 You will never deploy a raw container. You'll always deploy a Pod (that includes a container).



Node

A **Node** is a machine — either physical or virtual — that runs **Pods**.

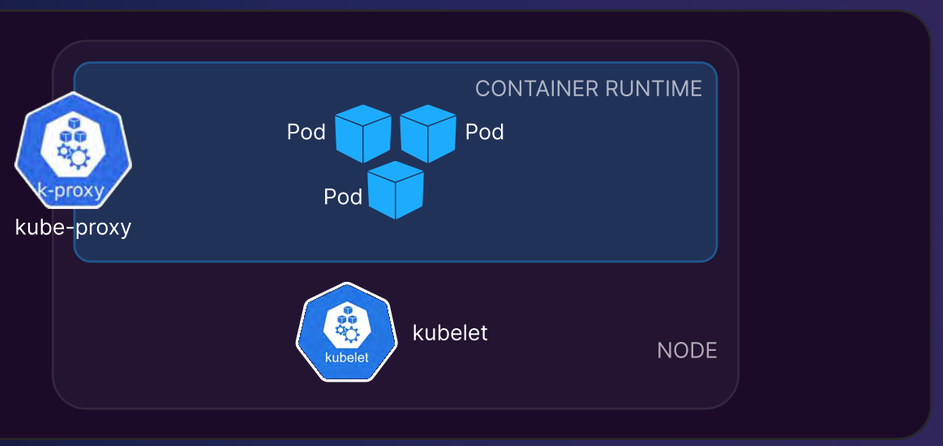
Each **Node** includes:

- A **container runtime** (e.g., containerd)
- A **kubelet**: agent that communicates with the Kubernetes control plane
- A **kube-proxy**: handles network routing inside the cluster

Nodes are registered to the cluster.

📌

- A Node is part of a larger system and works alongside other Nodes.
- Pods (your running applications) run on these Nodes.
- You'll later learn how commands are sent to Nodes to tell them what to run.



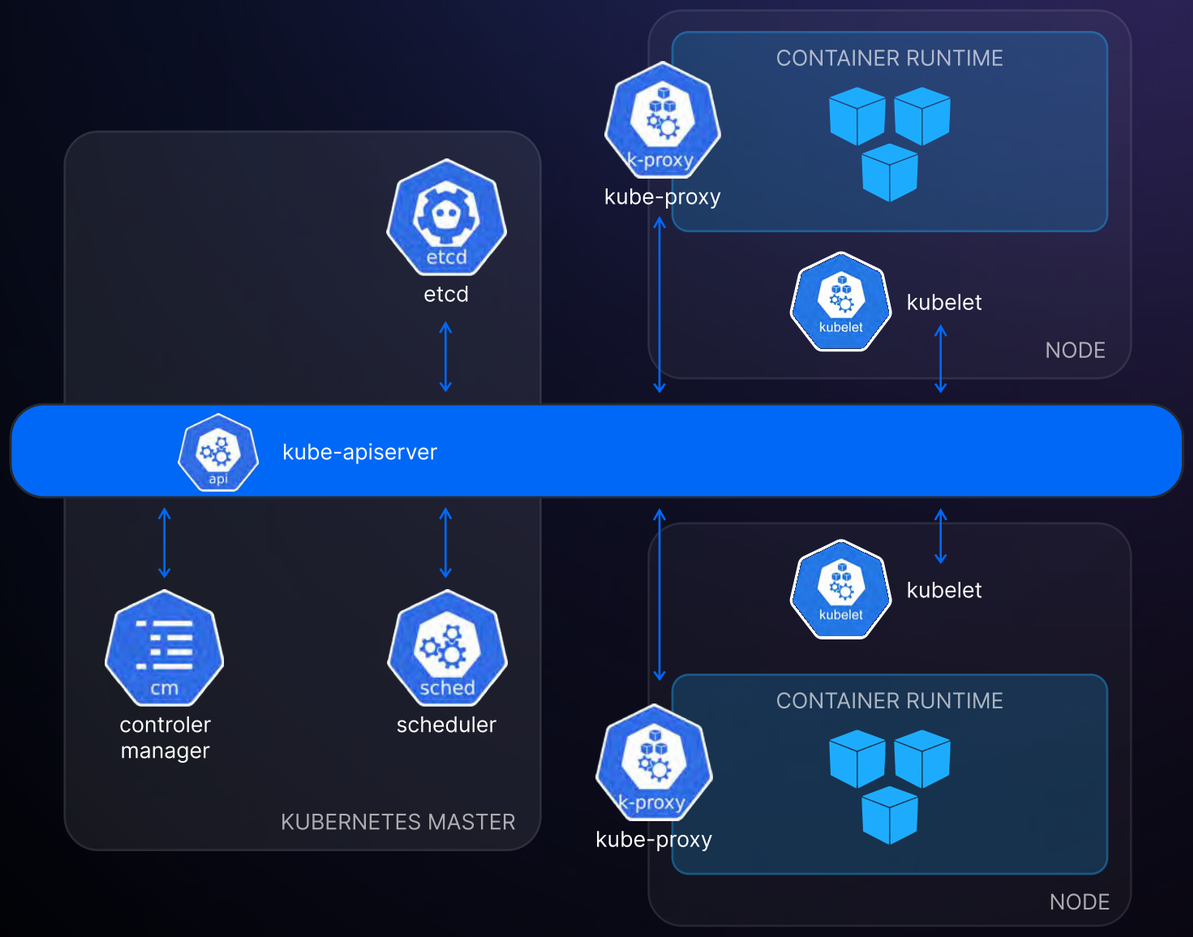
Cluster

A **Kubernetes Cluster** is the full system made of:

- **One or more Control Plane Nodes** (manage the overall system/cluster)
- **One or more Worker Nodes** (run the Pods — the actual application components)

All operations — from creating a Pod to scaling an app — happen **in the cluster**.

📌 When you use Kubernetes, you're interacting with the **cluster**.



kubectl

`kubectl` is the command-line tool used to interact with your Kubernetes cluster.

It let's you:

- View the state of the cluster
- Create/update/delete Pods, Services, etc.
- Apply YAML configuration files
- Debug what's happening

📌 You will use `kubectl` for nearly every task.

```
~$ kubectl get pods
~$ kubectl get all
~$ kubectl apply -f app.yaml
~$ kubectl describe pod <name>
```

🕹️ `kubectl` is your **remote control** for Kubernetes.

```
controlplane ~ → kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
newpods-qbjkl   1/1     Running   0          67s
newpods-7zxfq   1/1     Running   0          67s
newpods-4trjc   1/1     Running   0          67s
nginx           1/1     Running   0          42s
```

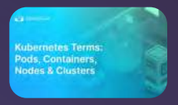
That command? We ran it live — right inside KodeKloud's Free Kubernetes Labs.

Summary Snapshot

CONCEPT	WHAT IT IS	WHY IT MATTERS
Container	Runs your app inside a small, packaged box	Lets you isolate and move apps easily
Pod	Holds one or more containers	The basic unit Kubernetes runs
Node	The machine that runs Pods	Does the actual work
Cluster	A group of Nodes + the control system	The full Kubernetes setup
kubectl	A tool to send commands	How you manage and control Kubernetes

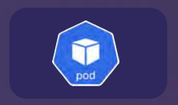
More Reads You'll Love

What Are Pods in Kubernetes? And Why Doesn't It Just Run Containers?



Kubernetes Terminology: Pods, Containers, Nodes & Clusters

What's a Pod, Really?



Kubernetes Pod

SECTION 3: INSIDE THE CLUSTER — WHAT RUNS KUBERNETES?

You Know the Cluster. Now Let’s See What’s Running It.

You now understand:

- **What a Cluster is**
- **What Nodes are**
- **What Pods are**
- **And how kubectl interacts with everything**

Now it’s time to look deeper — into what powers the **cluster itself**.

These are the **Control Plane components**. They don’t run your app — they run Kubernetes.

What Is the Control Plane?

The Control Plane manages the entire cluster.

- Decides what runs where
- Monitors the system
- Responds when things change
- Stores the cluster’s state

📌 When you type `kubectl apply`, you are talking to the Control Plane.

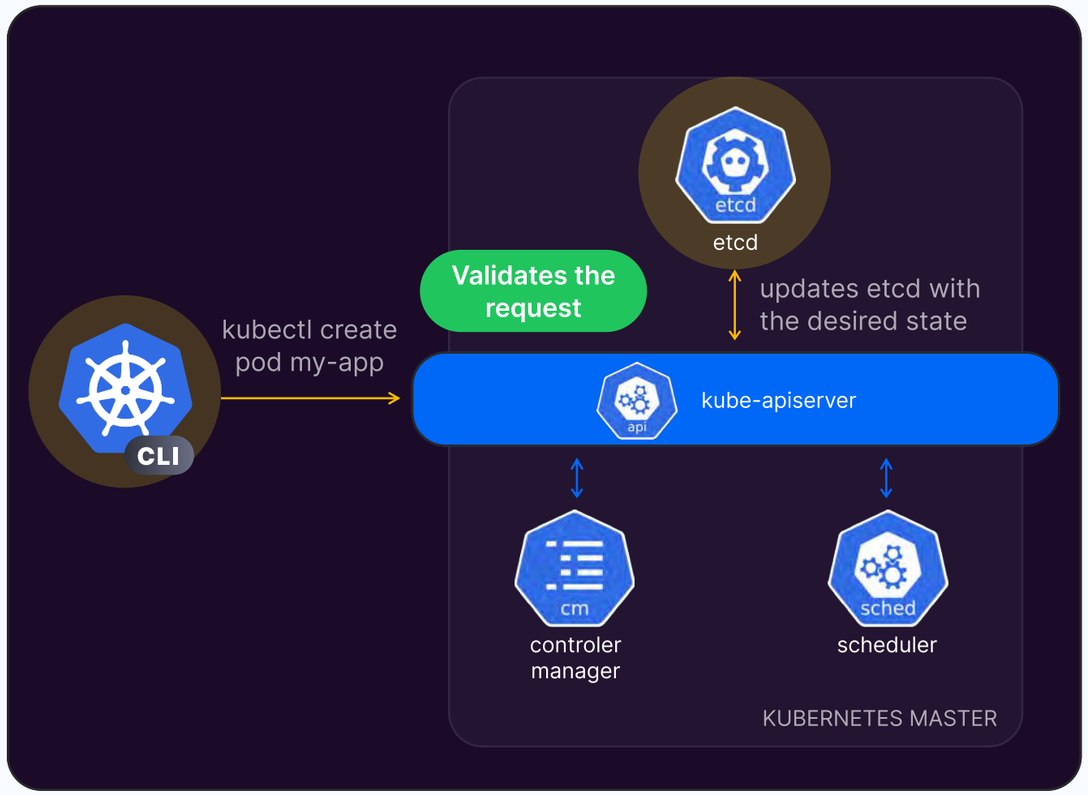
Control Plane Componenets Overview

COMPONENT	PURPOSE
API Server	Front door to the cluster
etcd	Stores all configuration and state
Controller Manager	Watches the cluster and reacts to changes
Scheduler	Decides which Node runs which Pod
Cloud Controller (optional)	Connects K8s to cloud-specific features

Let’s Break Them Down (In Order)

API Server

This is the main **entry point** for all control commands.



- Every `kubectl` command hits the API Server
- Every internal system component communicates through it
- It validates and processes requests
- It writes any state changes to etcd

📌 You never talk directly to etcd. You always go through the API Server.

etcd

This is the **key-value database** that stores:

- Cluster configuration
- Current state of all objects (Pods, Services, Deployments)
- Secrets and config maps

📌 etcd is the **single source of truth** for the cluster’s state. Everything from node health to pod specs lives in etcd.

Controller Manager

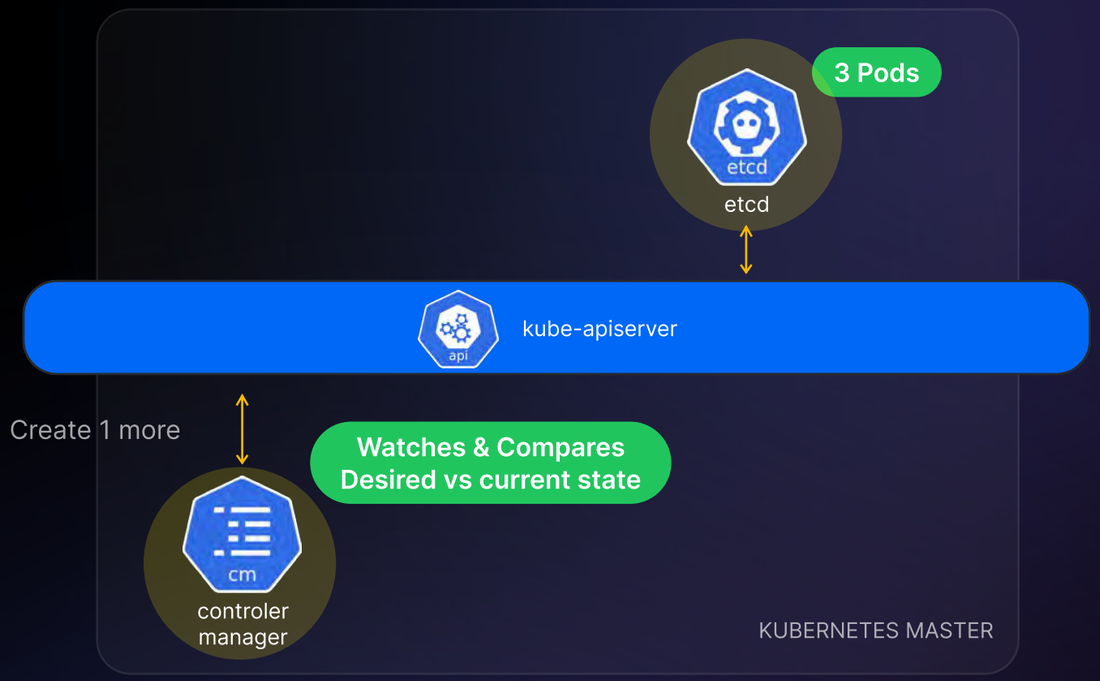
This component **watches** the cluster state via the API Server and **takes action** to keep things in sync.

Some examples of built-in controllers:

- **ReplicaSet Controller** – makes sure the correct number of Pods are running
- **Node Controller** – tracks node availability
- **Service Account Controller** – manages service accounts and tokens

📌 The Controller Manager compares the desired state (in etcd) with the current state, and fixes any mismatch.

KodeKloud



If you said you want 3 replicas of a Pod but only 2 are running:

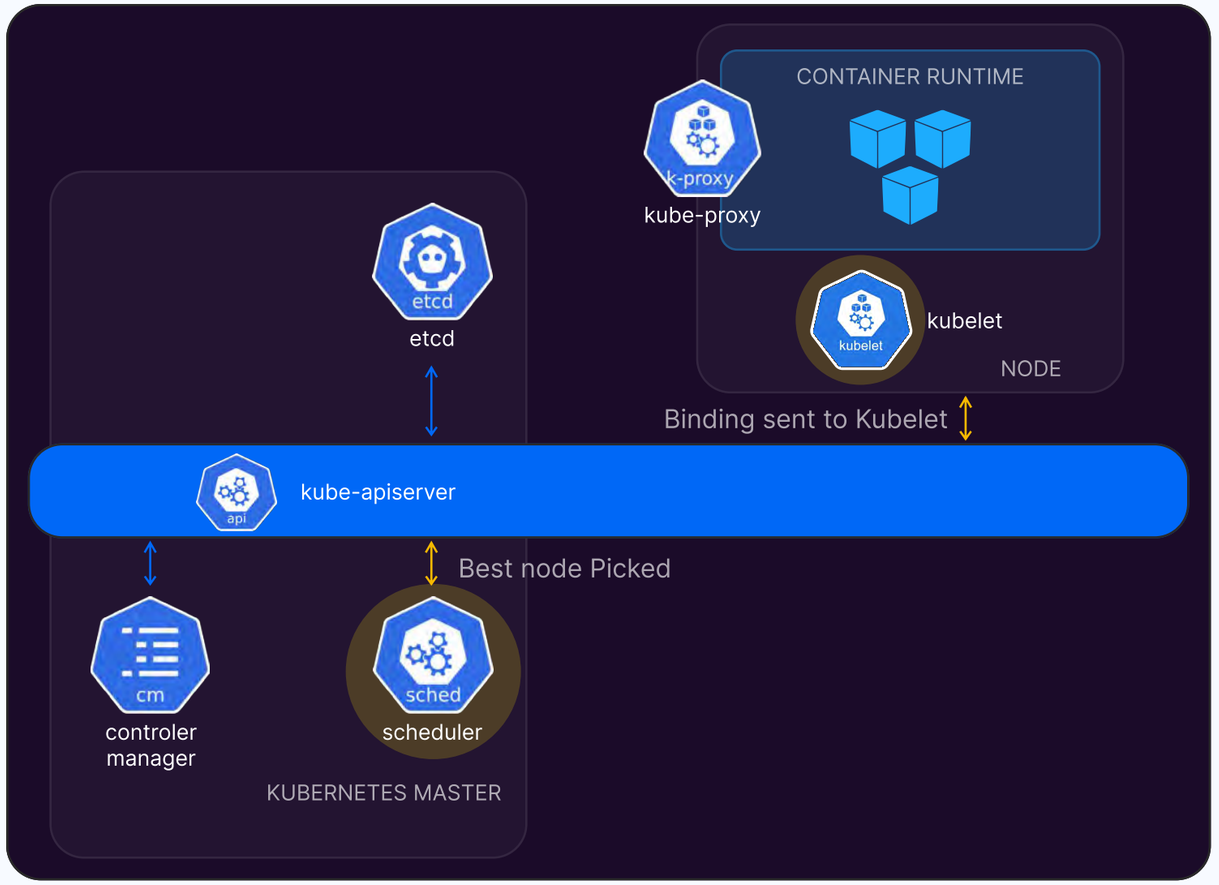
- The Controller notices this
- It tells the API Server to create one more Pod

Scheduler

The Scheduler assigns Pods to Nodes.

- It looks at:
  - Resource availability (CPU, memory)
  - Node labels
  - Affinity rules
- It picks the best Node for a Pod

📌 It doesn’t start the Pod. It binds the Pod to a Node. The kubelet on that Node will actually run the Pod.



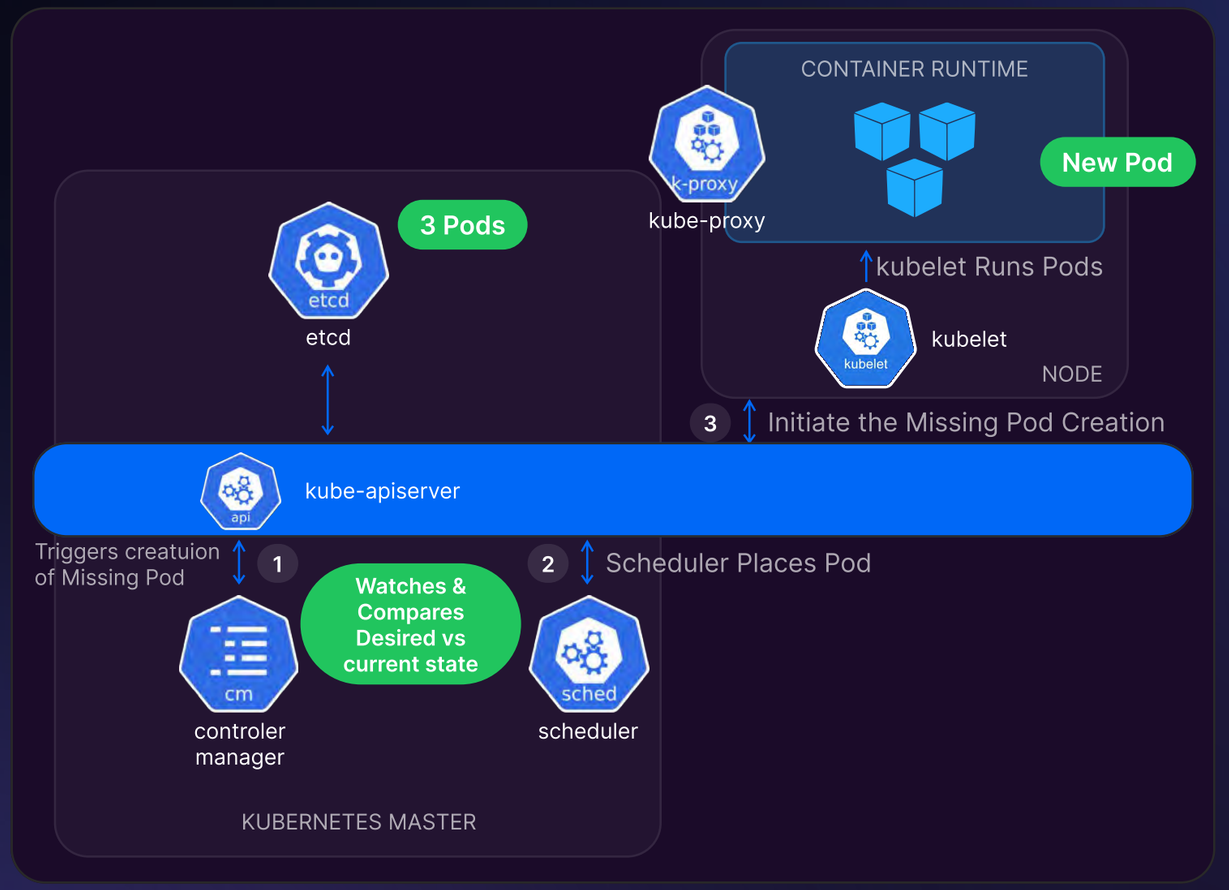
Cloud Controller Manager (optional)

Only used in cloud environments. It integrates Kubernetes with cloud provider APIs (like AWS, GCP, Azure).

It can:

- Automatically create Load Balancers
- Attach cloud storage volumes
- Update node information from cloud metadata

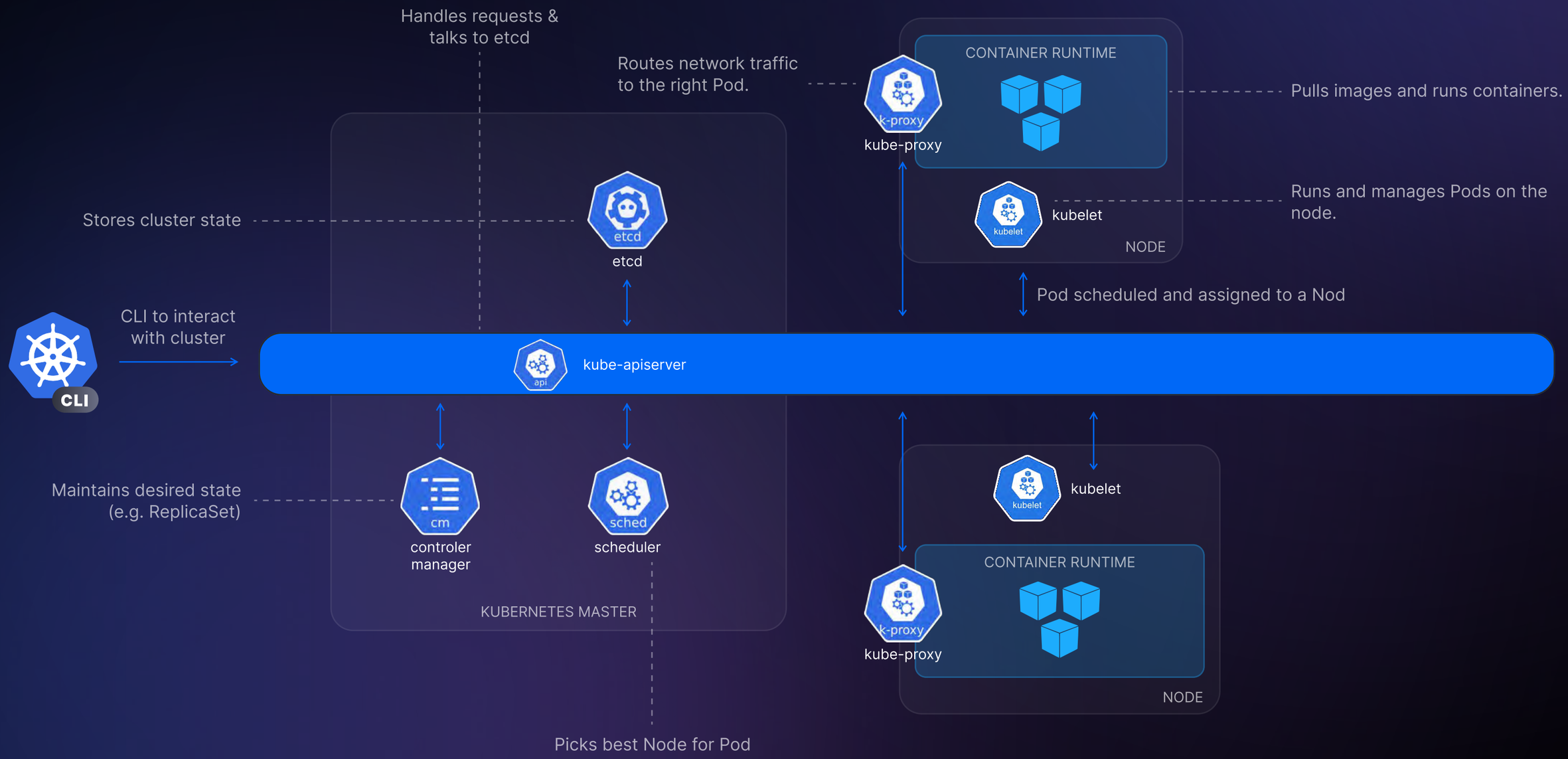
📌 If you're running Kubernetes on bare metal or in a local setup, you won't need this.



Summary Snapshot

COMPONENT	ROLE
API Server	Front door to the cluster, processes all requests
etcd	Stores cluster state and configuration
Controller Manager	Keeps cluster in sync with desired state
Scheduler	Assigns new Pods to Nodes
Cloud Controller	Manages cloud-specific integration (optional)

Inside a Kubernetes Cluster



Q1: Quick Recall Test

Match the job to the component:

CONCEPT	COMPONENT
Stores every object in the cluster	_____
Assigns a Pod to a Node	_____
You interact with it via `kubectl`	_____
Notices if a Node goes down	_____

Recommended Resource

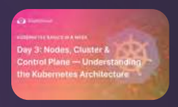
If you'd like to deepen your understanding of the Kubernetes Architecture you just saw, this short video will be a great next step.

Kubernetes Architecture in 10 Minutes

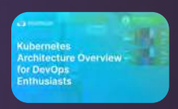
by Mumshad Mannambeth on the KodeKloud YouTube channel

Further Reading

If you prefer reading or want to explore concepts at your own pace, these blogs will help reinforce your understanding:



Understanding Nodes, Clusters & the Kubernetes Control Plane



Kubernetes Architecture Overview

Kubernetes Concepts Explained

SECTION 4: HOW A POD IS CREATED (STEP-BY-STEP BEHIND THE SCENES)

You typed kubectl apply. What really happens next?

Your Input

Let's say you apply a simple Pod definition like this:

```
~$ kubectl apply -f my-pod.yaml
```

1 Step 1: Your YAML Hits the API Server

- kubectl sends the YAML manifest to the API Server
- The API Server validates the manifest:
  - Are required fields present?
  - Is the syntax correct?
  - Is the Pod spec acceptable?

✅ **If valid** → request moves forward

❌ **If invalid** → error is returned immediately to your terminal

WHAT YOU ALREADY KNOW:

- The API Server is the entry point to the Kubernetes cluster.
- All requests go through it.

2 Step 2: API Server Stores the Request in etcd

- Once validated, the API Server stores the desired state of the Pod in etcd
- etcd now holds a record: "This Pod should exist in the cluster."

WHAT YOU ALREADY KNOW:

etcd stores all configuration and state information.

3 Step 3: Controller Notices a Pod Is Missing

- The **Controller Manager** (specifically the Pod controller) detects that a new Pod object has been added in etcd
- It notices: “This Pod doesn’t exist yet on any Node”
- It signals the Scheduler to make a placement decision

WHAT YOU ALREADY KNOW:

The **Controller Manager** watches for differences between desired and current state.

4 Step 4: Scheduler Selects the Best Node

- The **Scheduler** checks:
  - CPU and memory availability
  - Taints/tolerations
  - Node affinity
  - Node selectors
- It selects a suitable Worker Node
- It writes the decision back to the API Server, binding the Pod to that Node

WHAT YOU ALREADY KNOW:

The **Scheduler** doesn’t start the Pod. It just tells Kubernetes where to run it.

5 Step 5: Kubelet on the Chosen Node Takes Over

- The **kubelet** on the selected Node receives the binding information
- It pulls the Pod spec via the API Server
- It:
  - Downloads the container image (via `containerd`)
  - Creates and starts the container(s)
  - Monitors the Pod's health

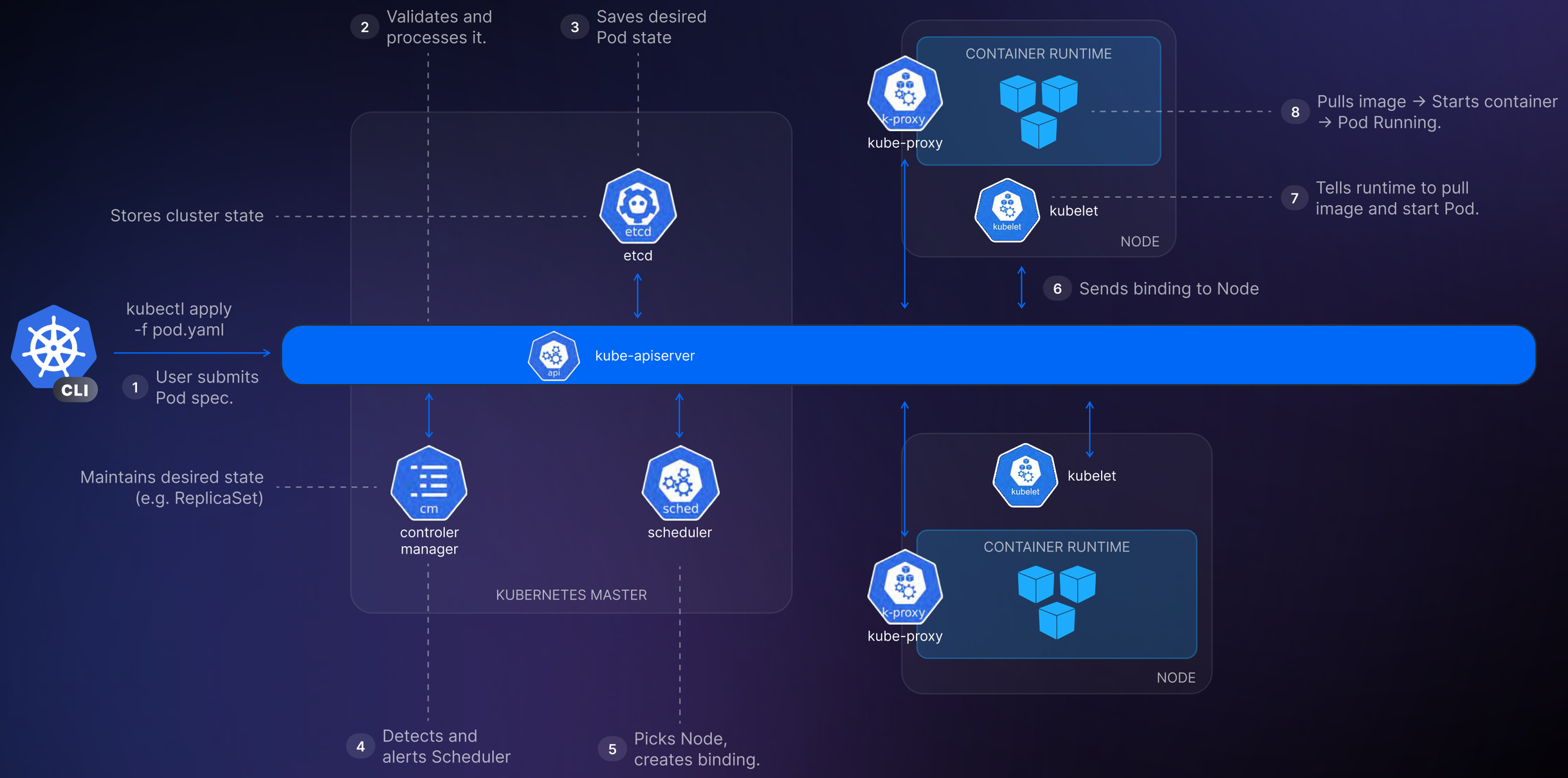
WHAT YOU ALREADY KNOW:

The **kubelet** is the local agent on each Node that talks to the API Server and runs Pods.

6 Step 6: Pod Is Now Running

- Once the containers are started:
  - The Pod is marked as `Running`
  - You’ll see it with: `~$ kubectl get pods`
- If something goes wrong (e.g., image not found), the kubelet retries and reports status

Kubernetes Pod Creation Flow



Summary Snapshot

STEP	ACTION	COMPONENT INVOLVED
1	YAML sent to API Server	kubectl → API Server
2	Desired state stored	API Server → etcd
3	Pod creation detected	Controller Manager
4	Node selected for the Pod	Scheduler
5	Pod created and monitored	kubelet (Worker Node)
6	Pod enters Running state	kubelet/containerd

Quick Debug Commands

Here's how to observe this in a real cluster:

```
~$ kubectl apply -f my-pod.yaml     # Submit Pod to cluster
~$ kubectl get pods                 # Check Pod status
~$ kubectl describe pod my-pod      # Detailed Pod info & events
~$ kubectl get events --sort-by=.metadata.creationTimestamp     # List events in order
```

Q2: Think Like Kubernetes

You applied a Pod. The container image failed to pull.

Which component reports this error?

Want to Go Even Deeper on Pods?

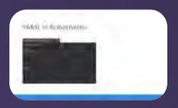
You’re picking up Kubernetes fast — now let’s lock it in with some real-world practice!

Read:

What Are Pods in Kubernetes? (KodeKloud Blog)

Get a super beginner-friendly breakdown of what Pods actually are and why they matter.

Watch:



Learn & Implement Kubernetes: POD Definition with YAML (YouTube)

Follow Mumshad as he shows you how to write your own Pod YAML — step-by-step.

Practice:

KodeKloud Free Pods Lab

Don’t just learn — try it live! Spin up Pods right now without installing anything.

SECTION 5: SERVICES IN KUBERNETES — HOW PODS ARE EXPOSED

Your Pod Is Running. But… Who Can Reach It?

WHAT YOU ALREADY KNOW:

- How a Pod is created
- How it runs inside a Node
- How Kubernetes brings everything together

But here's the next critical question:

> How do **users**, **apps**, or **other Pods** actually connect to your Pod?

✅ The answer: **Services**

What's a Service?

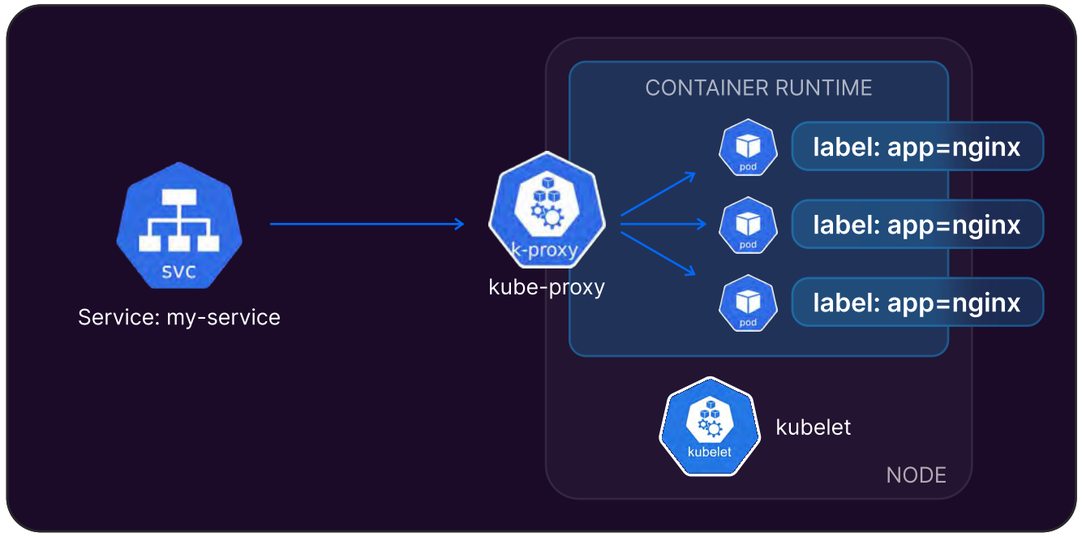
A Service is a stable, permanent way to access a group of Pods.

📌 Why it's needed:

- Pods come and go. Each one gets a new IP.
- You can't rely on Pod IPs — they're not fixed.
- Services provide a **consistent virtual IP** that routes traffic to the right Pods.

How Services Work (in Plain Steps)

1. **You create a Service** (usually via YAML)
2. **The Service selects matching Pods using labels**
3. **Kubernetes assigns the Service a virtual IP**
4. **Any request to that IP gets routed to one of the matching Pods**

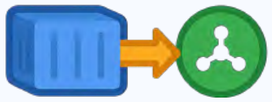


WHAT YOU ALREADY KNOW:

- **Pods are ephemeral. They get replaced often.**
- **Services give you a fixed way to reach them, even as they change.**

Types of Kubernetes Services

Let's break them down one by one.



ClusterIP (Default)

- Accessible only within the cluster
- Other Pods can use it to connect to your app

📌 Good for: Internal communication between apps

```yaml
type: ClusterIP
```

NodePort

- Opens a specific port on every Node's IP address
- Forwards traffic to the Service → matching Pods

📌 Good for: Basic external access to apps

📌 Limitation: Port number is fixed and high (default: 30000–32767)

```yaml
type: NodePort
ports:
  - port: 80
    nodePort: 30080
```

Then access with:

```
http://<NodeIP>:30080
```

LoadBalancer

- Provisions an **external IP** using a cloud provider's load balancer
- Routes traffic to the Service

📌 Good for: Production-ready access on cloud platforms

📌 Requires: Cloud setup like AWS, GCP, Azure

```yaml
type: LoadBalancer
```

ExternalName

- Maps a Service to an external DNS name
- Doesn't route traffic — just redirects to another domain

📌 Good for: Redirecting in-cluster traffic to external services

```yaml
type: ExternalName
externalName: example.com
```

⚠️ All YAML snippets shown here are partial examples focused on demonstrating changes in the type field of the Service. To apply them, wrap each snippet inside a full YAML manifest including apiVersion, kind, and metadata.

Service Example (YAML)

Here's a basic Service that exposes a set of Pods running NGINX:

service.yaml

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

📌 This will:

- Match any Pod with `app=nginx`
- Route requests on port 80 to those Pods

Summary Snapshot

TYPE	ACCESSIBLE FROM	EXPOSES TO	USE CASE
ClusterIP	Inside the cluster	Cluster IP	Internal services
NodePort	Outside the cluster	Node IP + port	Basic external access
LoadBalancer	Outside the cluster	External IP (cloud)	Production-grade cloud access
ExternalName	Inside the cluster	DNS redirect	Point to external domain services

TYPE	WHEN TO USE
ClusterIP →	http://nginx-service.default.svc.cluster.local Accessing internal services within the cluster
NodePort →	http://<NodeIP>:30080 Exposing services externally for quick access
LoadBalancer →	http://123.45.67.89 Public access to production-grade services
ExternalName →	http://api.external-service.com Redirecting to external APIs or third-party services

RECAP: WHY USE A SERVICE?

- **Pod IPs change — Services stay stable**
- **Services allow load-balancing across Pods**
- **Kubernetes tracks which Pods are ready and sends traffic accordingly**
- **Services are essential for both in-cluster communication and external access**

Pods Vs Services

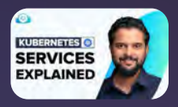
CONCEPT	PURPOSE	LIFECYCLE
Pod	Runs your app container(s)	Short-lived
Service	Provides stable access to Pods	Long-lived & stable

Q3: Quick Test (Fill-in-the-blank style):

You want only internal access between microservices.	Use: _____
You want to expose your app to the internet on AWS.	Use: _____
You want to connect to another domain (like abc.test.com).	Use: _____

Want to Learn More About Kubernetes Services?

You're just a few clicks away from mastering them:

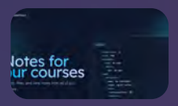


Watch this quick & clear YouTube lesson by Mumshad:

Kubernetes - Services Explained in 15 Minutes!

Prefer reading? This blog breaks it down beautifully:

Kubernetes Services Explained



KodeKloud Notes App — a learner favorite:

See the Services section here

Perfect for quick reviews & revision!

Ready to get hands-on? Practice in our free labs:

Explore the Kubernetes Services Labs

SECTION 6: LABELS, SELECTORS & NAMESPACES — HOW KUBERNETES ORGANIZES EVERYTHING

When You Have 100s of Pods… How Does Kubernetes Keep Track?

WHAT YOU ALREADY KNOW:

- Pods are created and managed by Kubernetes
- Services connect to Pods using label matching

Now it's time to understand how Kubernetes keeps things **organized** using three important mechanisms:

- **Labels**
- **Selectors**
- **Namespaces**

🏷️ Labels — Add Meaning to Your Objects

A **label** is a key-value pair that you attach to any Kubernetes object:

✨ Pods ✨ Services ✨ Deployments ✨ Nodes

…and more

Labels are **not unique** — multiple objects can share the same labels.

📌 Labels are not for display. They are for **filtering**, **grouping**, and **querying**.

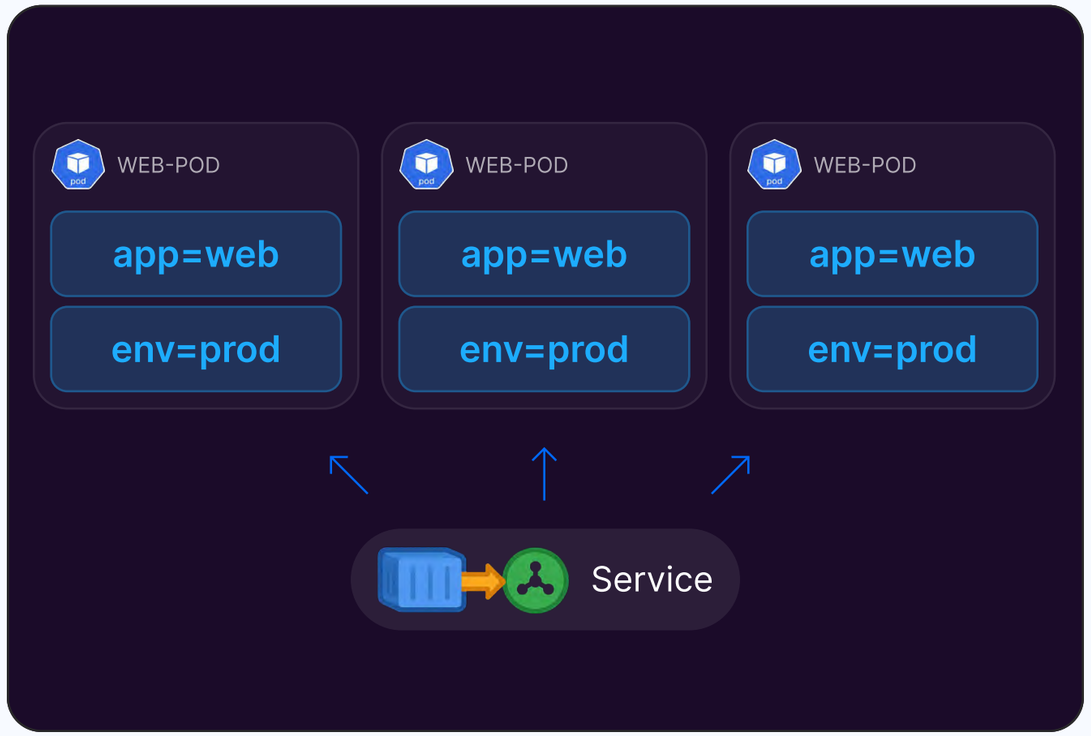
Example:

Here's a Pod with two labels:

```yaml
metadata:
  name: web-pod
  labels:
    app: web
    env: prod
```

📌 This is not the full Pod YAML — just a snippet to show how labels are defined under metadata.

This Pod is now tagged as: `app: web` `env: prod`



Multiple Pods grouped with the same labels → all accessible by a single Service

Selectors — Find Objects by Labels

A **selector** tells Kubernetes:

> **Find objects that match these labels.**

This is how:

- **Services** find the right Pods
- **Deployments** manage groups of Pods
- You query specific resources using `kubectl`

Example:

Here's how a Service uses a selector to match Pods:

yaml

```yaml
selector:
  app: web
```

Any Pod with `app=web` will receive traffic from this Service.

📌 This is the selector section in a Service YAML that matches Pods with app: web.

CLI Example:

Here's how a Service uses a selector to match Pods:

```
~$ kubectl get pods -l app=web

NAME                   READY   STATUS    RESTARTS   AGE
web-5f89f5c7d9-pxj9d   1/1     Running   0          2m
web-5f89f5c7d9-zl92v   1/1     Running   0          2m
web-5f89f5c7d9-wx712   1/1     Running   0          2m
```

This will return only Pods labeled `app=web` .

WHAT YOU ALREADY KNOW:

- Services route traffic to Pods using label selectors
- This is that exact mechanism in action.

Namespaces — Isolate and Organize Resources

A namespace is a virtual cluster within your actual Kubernetes cluster.

Namespaces help you:

- Group related resources
- Isolate environments (e.g., dev, test, prod)
- Avoid name collisions between teams or apps

📌 All objects (Pods, Services, Deployments, etc.) exist in a namespace.

Common Namespace Operations

View all namespaces:

```
~$ kubectl get namespaces
```

View all Pods in a specific namespace:

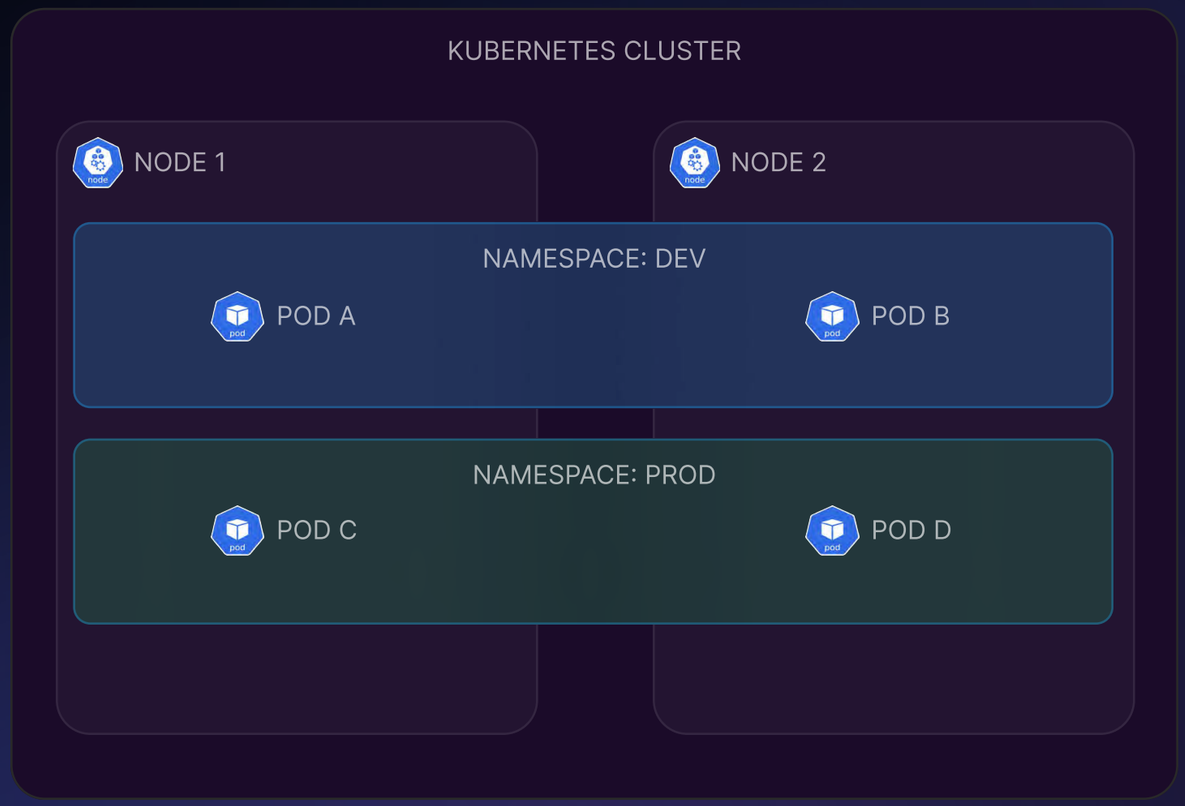
```
~$ kubectl get pods -n dev
```

Create a new namespace:

```
~$ kubectl create namespace staging
```

Default Namespaces in Every Cluster:

NAMESPACE	PURPOSE
default	Used when no other namespace is specified.
kube-system	Holds system-level components like kube-dns, controller-manager, etc.
kube-public	Publicly readable data (e.g., cluster-info ConfigMap) — minimal usage.
kube-node-lease	Used for node heartbeat leases to improve scalability of node monitoring.



- Namespaces group Pods logically, not physically.
- Nodes can run Pods from multiple namespaces.

Summary Table

CONCEPT	WHAT IT IS	WHY IT MATTERS
Label	Key-value pair on objects	Used to group and filter resources
Selector	Way to find objects using labels	Lets controllers or CLI target specific objects
Namespace	Virtual cluster within the cluster	Organizes and isolates resources (e.g., dev vs prod)

Q4: Check Your Understanding:

What would this command return?

```
~$ kubectl get pods -l env=prod -n staging
```

Q5: Bonus Challenge (Think Like a Pro)

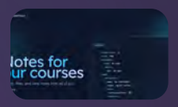
What does this command do?

```
~$ kubectl get pods -l 'tier in
(frontend,backend)' -n dev --field-
selector=status.phase=Running
```

Recommended Resources to Boost Your Understanding

Network Namespaces Basics Explained in 15 Minutes

Struggling to understand how Kubernetes isolates traffic between apps? This short video breaks down network namespaces — a foundational concept for mastering Kubernetes networking. Don’t skip this if you're serious about leveling up!



KodeKloud Notes: Labels & Selectors (KCNA)

Still unsure about the difference between labels, selectors, and field selectors? This guide clears it all up with simple, exam-friendly explanations. Perfect for quick revision or deep dives!

SECTION 7: DEPLOYMENTS & REPLICASETS — RUNNING APPS AT SCALE

You Don’t Just Want a Pod. You Want a System That Keeps It Running.

YOU NOW KNOW:

- **Pods run your containerized app**
- **Services expose those Pods**
- **Labels help group Pods**
- **Selectors connect objects**

BUT...

- **What happens if a Pod crashes?**
- **What if you want 5 Pods instead of 1?**
- **What if you want to update your app without downtime?**

> That’s where **Deployments** and **ReplicaSets** come in.

What’s a ReplicaSet?

A **ReplicaSet** ensures that a specific number of **identical Pods** are running at all times.

📌 If a Pod crashes, the ReplicaSet creates a new one.

📌 If you scale up to more replicas, it creates more Pods with the same spec.

Example:

A ReplicaSet with `replicas: 3` means:

- Kubernetes will make sure **3 matching Pods** are always running.
- If one dies, a new one is started automatically.

```yaml
replicas: 3
selector:
  matchLabels:
    app: web
```

📌 This is a partial YAML showing how replicas and matchLabels are defined inside a Deployment or ReplicaSet spec.

💡 **WHAT YOU ALREADY KNOW:**

Labels are used to match Pods to Services — and also to ReplicaSets.

So What's a Deployment?

A **Deployment** is a higher-level object that:

- Creates a ReplicaSet
- Manages rolling updates
- Handles rollbacks
- Defines how your app should run

📌 Most of the time, you won't create a ReplicaSet directly — you'll use a Deployment.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: nginx
          image: nginx:1.21
          ports:
            - containerPort: 80
```

This will:

- Create a ReplicaSet with 3 Pods
- Use the NGINX image version `1.21`
- Ensure the Pods are labeled `app=web`

New Concept: Pod Template

Inside a Deployment, you see this block:

```yaml
template:
  spec:
    containers:
```

This is the **Pod Template** — a blueprint Kubernetes uses to create identical Pods.

📌 Any time a Pod needs to be recreated, it will be based on this template.

What Happens When You Update the Image?

Let's say you change the image from `nginx:1.21` to `nginx:1.22` and apply it:

```
~$ kubectl apply -f web-deployment.yaml
```

Kubernetes will:

1. Start new Pods with the new image
2. Wait for them to become healthy
3. Delete the old Pods
4. Complete the update — **without downtime**

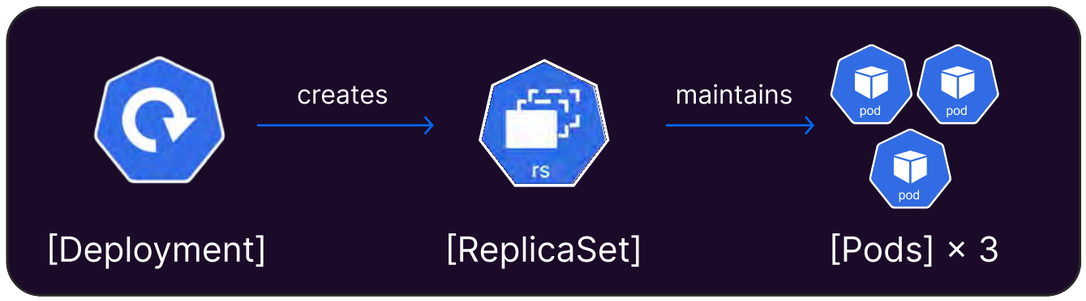
Can You Roll Back?

Yes. You can roll back to a previous version with:

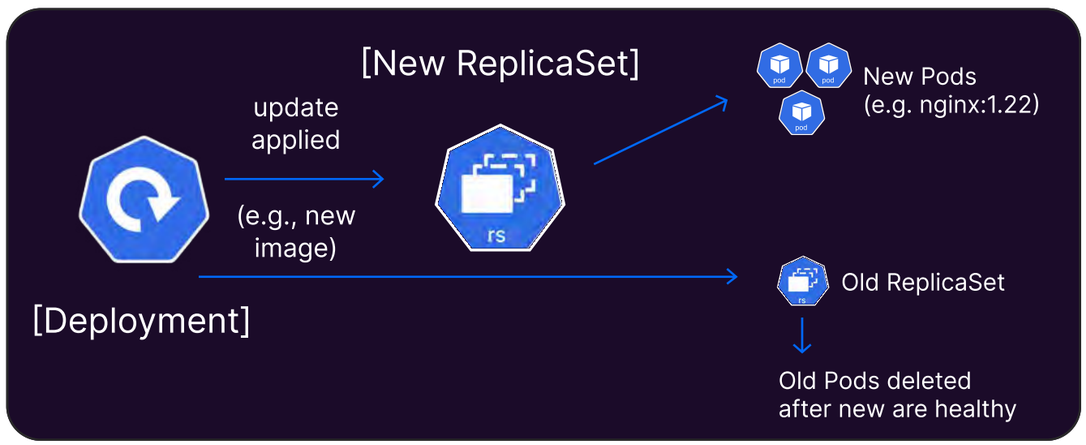
```
~$ kubectl rollout undo deployment web-deployment
```

Kubernetes keeps track of the **deployment history**.

Deployment Flow



Rolling Update Flow



kubectl Deployment Commands

```
# List all deployments
kubectl get deployments

# Show detailed info about a specific deployment
kubectl describe deployment web-deployment

# Scale the deployment to 5 replicas
kubectl scale deployment web-deployment --replicas=5

# Check rollout status (is the update complete?)
kubectl rollout status deployment web-deployment

# Roll back to the previous version
kubectl rollout undo deployment web-deployment
```

Summary Table

CONCEPT	WHAT IT DOES	WHY IT MATTERS
ReplicaSet	Ensures a fixed number of Pods are running	Auto-healing, consistent scaling
Deployment	Manages ReplicaSets + handles updates/rollbacks	Safer changes, simplified control
Pod Template	Blueprint for new Pods	Used when creating/replacing Pods

Q6: Quick Review

What object is responsible for keeping Pods running? → _____

What object helps you update your app version? → _____

Where do Pods get recreated from? → _____

What's Next? Try It Out Yourself

You've seen how the commands work—now it's time to get your hands dirty and try them out in real environments.

Practice It, Don't Just Read It

Use these free labs to apply what you've learned:

Kubernetes ReplicaSet Lab

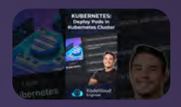
Learn how Kubernetes keeps your pods running.

Kubernetes Deployment Lab

Practice rolling out and managing app updates.

Prefer a Walkthrough First?

Check out this short demo of how a real Kubernetes Deployment works on KodeKloud Engineer(KKE):



Watch the Demo

Want Tasks Like This Every Day?

If you're wondering how to practice these kinds of real-world scenarios consistently, KodeKloud Engineer is built just for that. It's a platform that feels like working at an actual IT firm.

You get:

- Daily DevOps tasks
- Real-world challenges
- Step-by-step learning through doing

Explore KodeKloud Engineer

SECTION 8: CONFIGMAPS, SECRETS, AND VOLUMES — GIVING PODS CONFIGURATION, CREDENTIALS, AND STORAGE

Your App Works. Now Let’s Make It Useful.

You’ve deployed your app using Deployments.

It runs, scales, updates, and rolls back. But here’s what it **can’t** do yet:

- Load configuration from a file or environment variable
- Access credentials securely
- Save data between restarts

That’s where **ConfigMaps**, **Secrets**, and **Volumes** come in.

ConfigMap — For Plain Configuration Data

A **ConfigMap** lets you store:

- Key-value pairs
- App settings
- Environment variables
- Command-line args

📌 It’s for non-sensitive data.

Example ConfigMap:

yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_MODE: "production"
  APP_PORT: "8080"
```

Use It in a Pod (as env variables):

```yaml
containers:
  - name: my-app
    image: myapp:latest
    envFrom:
      - configMapRef:
          name: app-config
```

📌 This is a partial snippet showing how to load environment variables from a ConfigMap. Place it under spec.template.spec.containers in a Pod or Deployment.

🎯 Your container now receives `APP_MODE` and `APP_PORT` from the ConfigMap as environment variables.

🔐 Secret — For Sensitive Data

Secrets are like ConfigMaps — but **encrypted** and **base64-encoded**.

They're used to store things like:

- Passwords
- API tokens
- TLS certificates

📌 Avoid putting secrets directly in your Pod spec or images.

service.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque                # default type for generic secrets
data:
  username: YWRtaW4=        # base64 for 'admin'
  password: cGFzc3dvcmQ=    # base64 for 'password'
```

Use It in a Pod (as env variables):

```yaml
env:
  - name: DB_USER
    valueFrom:
      secretKeyRef:
        name: db-secret
        key: username
```

📌 The container will get `DB_USER=admin` without seeing the raw value in the YAML.

✅ Load All Secret Keys as Env Vars (using `envFrom`)

```yaml
containers:
  - name: my-app
    image: myapp:latest
    envFrom:
      - secretRef:
          name: db-secret
```

WHAT YOU ALREADY KNOW:

- You've used labels to organize, and Deployments to control Pods.
- Now you're attaching values from outside — without rebuilding your image.

Volumes — For Persisting or Sharing Data

A **Volume** provides storage to containers running inside a Pod.

You can use Volumes to:

- Persist data between container restarts
- Share files between containers in the same Pod
- Mount ConfigMaps or Secrets as files

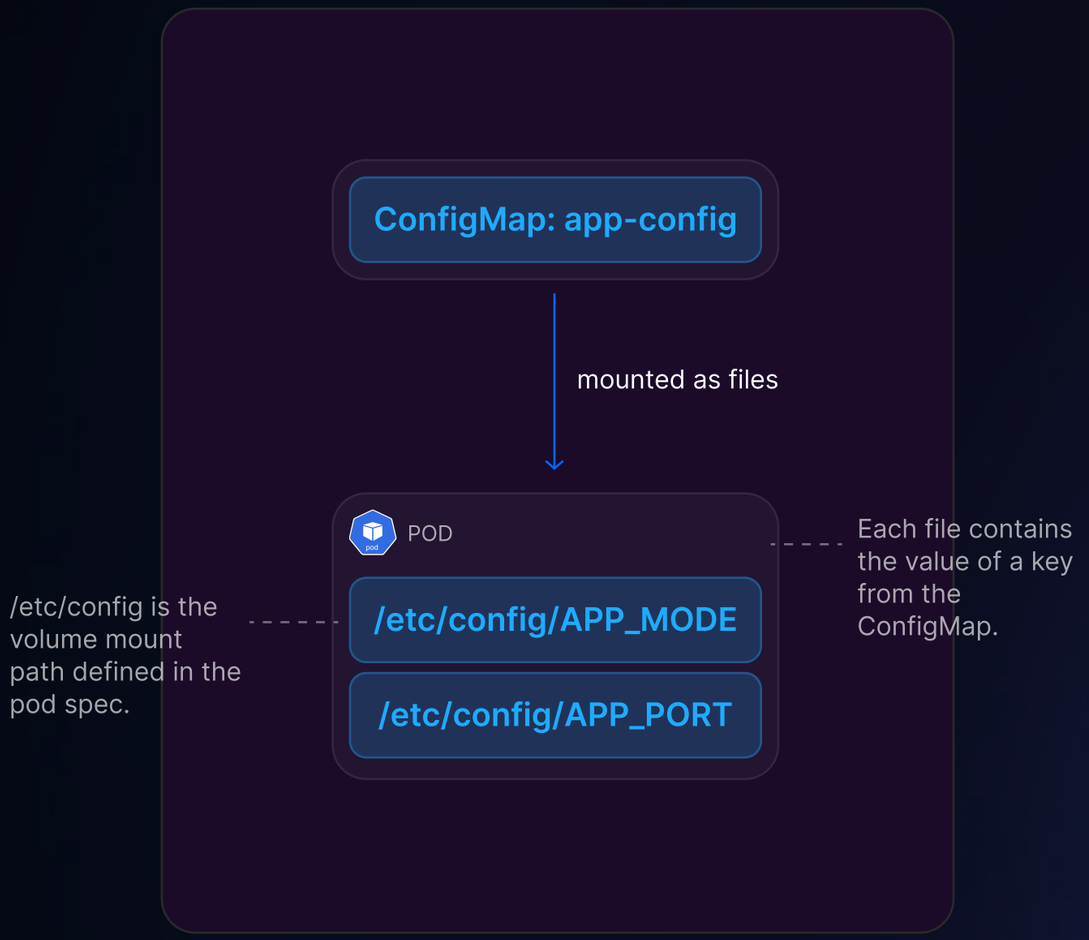
Example: Mounting a ConfigMap as a Volume

```yaml
volumes:
  - name: config-volume
    configMap:
      name: app-config

containers:
  - name: app
    volumeMounts:
      - name: config-volume
        mountPath: /etc/config
```

📌 All keys in the ConfigMap will become files inside `/etc/config`

⚠️ All YAMLs shown are partial and focus only on the key concept. Wrap them with full spec structure (like spec, template, etc.) when using in real manifests.



Types of Volumes (Quick Look)

VOLUME TYPE	USE CASE
emptyDir	A blank folder for your app — deleted when the Pod is gone.
hostPath	Uses a folder from the Node's computer — ⚠ be careful, it's shared.
configMap	Mounts config key-value pairs as files inside your app container.
secret	Mounts sensitive data (like passwords or tokens) as files inside your app.
persistentVolumeClaim	Connects to permanent storage backed by local, NFS, or cloud storage to keep data safe.

Summary Table

FEATURE	WHAT IT DOES	WHEN TO USE IT
ConfigMap	Provides plain (non-sensitive) config data	For app settings like environment variables (env vars)
Secret	Provides **sensitive** config data	For confidential info like passwords, tokens, API keys
Volume	Stores and shares data between pods	For saving logs, config files, or database data

Q7: Quick Review

Which resource would you use to inject API keys? → _____

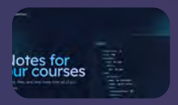
How do you mount a config value as a file inside a Pod? → Use a _____

Do ConfigMaps store encrypted data? → _____

Want to Master Kubernetes Secrets?

Dive Deeper into Secrets (CKAD Notes)

Unlock the full potential of Secrets with hands-on guidance from our CKAD course notes:



Read the CKAD Notes on Kubernetes Secrets ›

Explore Real-World Use Cases

Understand how Secrets work under the hood and avoid common mistakes in real Kubernetes environments:

Read the KodeKloud Blog on Kubernetes Secrets ›

SECTION 9: HEALTH CHECKS, PROBES & READINESS GATES — KEEPING YOUR APP STABLE

Your App Is Running... But Is It Working?

By now, you've deployed an app, exposed it via a Service, injected configs and secrets, and maybe even added persistent storage.

But Kubernetes doesn't know if your app is:

- Crashed internally?
- Still starting up?
- Failing to handle traffic?

To help Kubernetes **know when to act**, we use **Probes**.

Kubernetes Probes: The Three Types

Kubernetes supports **three types of probes**, each with a specific purpose:

PROBE TYPE	PURPOSE	WHAT HAPPENS IF IT FAILS
Liveness	Is the app still running?	Pod is restarted
Readiness	Is the app ready to receive traffic?	Pod is removed from Service
Startup	Is the app done initializing?	Prevents liveness checks until the app is ready

WHAT YOU ALREADY KNOW:

Pods can be restarted or recreated. These probes help Kubernetes decide when to do that.

Liveness Probe — Detect App Crashes

- Used to check if your app is **alive and responding**.
- If the probe fails, Kubernetes **restarts the container**.

Liveness Probe Using HTTP

```yaml
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
```

📌 This is a partial YAML. Add it inside the containers: section of your Pod or Deployment spec.

💡 This sends a request to http://<pod>:8080/healthz every 5 seconds.

If it fails, the Pod is considered unhealthy and will be restarted.

Readiness Probe — Detect Traffic-Ready Apps

Used to check if the app is **ready to serve users**.

If it fails, the Pod stays running, but is **removed from the Service endpoint list**.

Readiness Probe Using TCP

```yaml
readinessProbe:
  tcpSocket:
    port: 5432
  initialDelaySeconds: 5
  periodSeconds: 10
```

📌 Partial YAML — add this inside the containers: section.

💡 This checks if your app is accepting TCP connections on port 5432.

CLI Impact:

Even if `kubectl get pods` shows the Pod as **Running**, it may still be marked as **Not Ready** — meaning it won't receive traffic.

```
~$ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
my-app-68d95fdf9c-abcde   0/1     Running   0          15s
```

Startup Probe — Give It Time to Boot

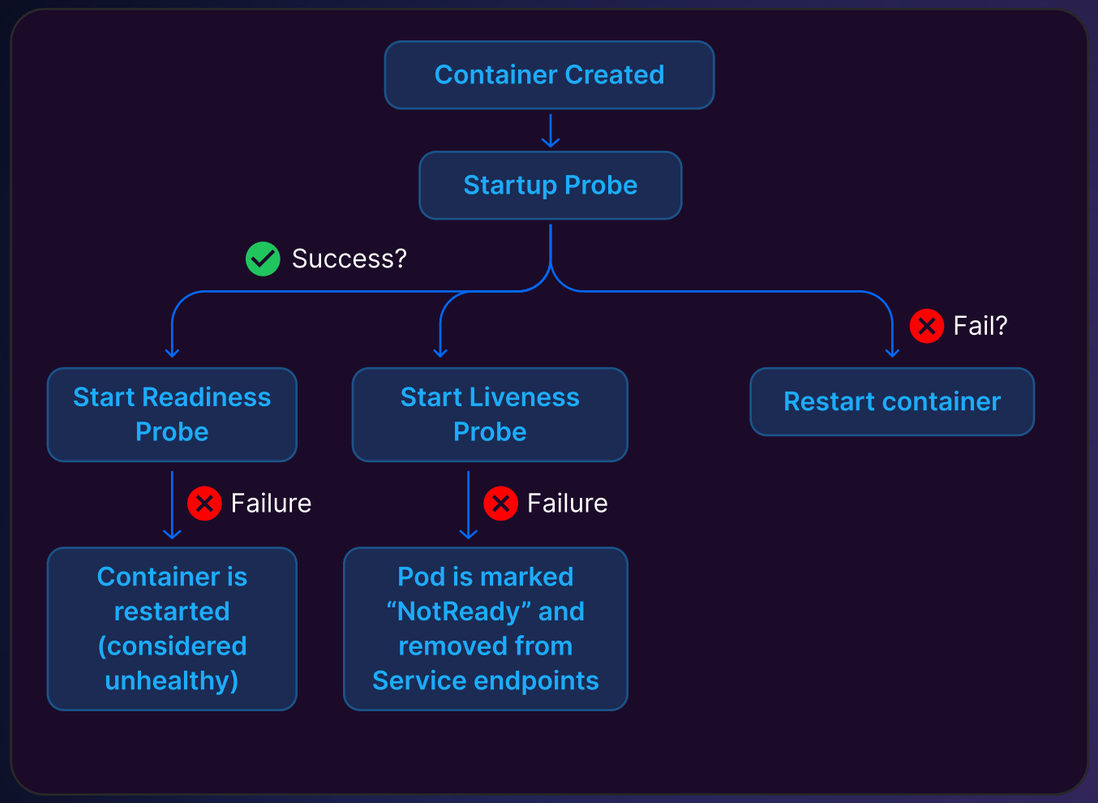
- Useful for **slow-starting apps** (e.g., large Java apps, DBs).
- A **Startup Probe** runs **once**, before liveness or readiness probes begin.
- If it fails, the container is restarted.

Startup Probe Using HTTP

yaml

```yaml
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  failureThreshold: 30
  periodSeconds: 5
```

Try checking /startup on port 8080 every 5 seconds. Wait until it passes or fail 30 times before giving up and restarting the container. This gives the container 30 × 5 = 150 seconds to start before considering it failed.



Common Config Options (All Probes)

FIELD	WHAT IT MEANS (SIMPLE EXPLANATION)	WHY IT MATTERS
initialDelaySeconds	Wait time before starting the first check	Gives your app time to start up before probing begins
periodSeconds	Time between each check	Controls how often Kubernetes checks if the app is OK
timeoutSeconds	Max time to wait for a response	Avoids hanging—K8s considers the check failed if no reply in time
failureThreshold	How many failures before Kubernetes takes action	Prevents restarting too quickly for temporary glitches
successThreshold	How many successes in a row to consider the app healthy	Useful mostly for readiness probes to avoid flapping

Summary Table

PROBE TYPE	WHAT IT CHECKS	WHAT HAPPENS IF IT FAILS
Liveness Probe	Is the app still running or stuck?	Pod is restarted by Kubernetes
Readiness Probe	Is the app ready to serve requests?	Pod is marked “Not Ready” and removed from Service
Startup Probe	Has the app finished starting up properly?	Pod is restarted if startup takes too long or fails

Q8: Quick Review

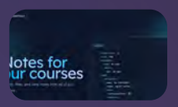
Which probe protects a Pod from getting traffic before it’s ready? → _____

Which probe lets you wait before even starting health checks? → _____

Which probe causes a container restart if it keeps failing? → _____

Theory is great — but real skills come from doing.

Ready to put your Readiness and Liveness knowledge to the test?



Jump into this hands-on solution guide and see probes in action inside Kubernetes!

SECTION 10: TYING IT ALL TOGETHER

Let’s Actually Build Something Real Now.

What You’re About to Deploy

You’ll create a small web app setup that includes:

- A Deployment (2 Pods running NGINX)
- ConfigMap (for app settings)
- Secret (for secure credentials)
- Probes (to monitor app health)
- NodePort Service (to access the app)

Yes — it’s everything you’ve learned so far. In one simple, working example.

1 ConfigMap (Store config)

configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_MODE: "production"
```

📌 Creates a key `APP_MODE` with value `production`

2 Secret (Store sensitive data)

secret.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
data:
  username: YWRtaW4=       # admin
  password: c2VjdXJlcGFzcw==  # securepass
```

📌 These values are base64-encoded. Don’t worry — Kubernetes will decode them inside the Pod.

3 Deployment (Run the app, inject values, monitor health)

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: nginx
          image: nginx:1.21
          ports:
            - containerPort: 80
          envFrom:
            - configMapRef:
                name: app-config
          env:
            - name: ADMIN_USER
              valueFrom:
                secretKeyRef:
                  name: app-secret
                  key: username
            - name: ADMIN_PASS
              valueFrom:
                secretKeyRef:
                  name: app-secret
                  key: password
          livenessProbe:
            httpGet:
              path: /
              port: 80
            initialDelaySeconds: 10
            failureThreshold: 3
          readinessProbe:
            httpGet:
              path: /
              port: 80
            initialDelaySeconds: 5
            failureThreshold: 2
```

What This Does:

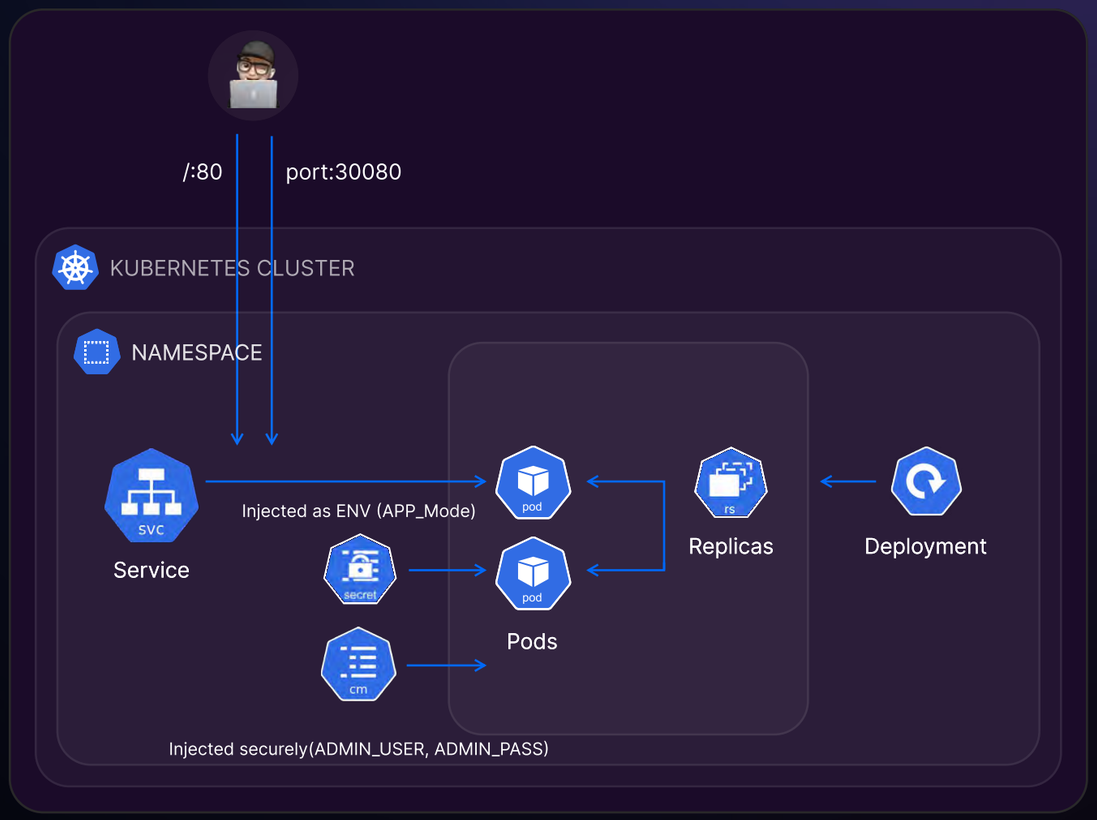
- Spins up 2 Pods using nginx
- Loads config via APP_MODE=production
- Injects username/password from the Secret
- Runs /healthz checks to know:
  - Is the app alive? (Liveness)
  - Is it ready for traffic? (Readiness)

4 Service (Expose the app)

service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  type: NodePort
  selector:
    app: web            # Matches the Deployment label
  ports:
    - port: 80          # Port exposed by the service
      targetPort: 80    # Port on the container to forward to
      nodePort: 30080   # Port on the node (VM) to access externally
```

📌 You'll access your app using
http://<Node-IP>:30080



Step-by-Step: Essential kubectl Commands for Deploying and Managing Your App

```
# Apply the ConfigMap definition
kubectl apply -f configmap.yaml

# Apply the Secret definition
kubectl apply -f secret.yaml

# Deploy your application (creates Pods via Deployment)
kubectl apply -f deployment.yaml

# Optional: Watch Pod status to confirm they start successfully
kubectl get pods -w

# Optional: If issues, debug a Pod
kubectl describe pod <pod-name>
kubectl logs <pod-name>

#  Apply Service to expose your app
kubectl apply -f service.yaml

# View service details (e.g., NodePort)
kubectl get service web-service

# Optional: Access the app via port-forwarding (if NodePort not reachable)
kubectl port-forward service/web-service 8080:80

# Optional: Check env vars injected into the Pod (verify config/secret)
kubectl exec -it <pod-name> -- env

# List all running Pods in the current namespace
kubectl get pods

# Show full details of a specific Pod (useful for troubleshooting)
kubectl describe pod PODNAME
```

🚀 That's It — You Deployed It All

You've now:

- ✅ Created a running app
- ✅ Injected external config and secrets
- ✅ Exposed it to users
- ✅ Monitored its health in real time

> **That's a production-grade foundation in Kubernetes.**

KUBERNETES NEXT STEPS ROADMAP

What Now?

You've finished **Kubernetes Made Easy** — and you've already:

- ✅ Learned core components (Pods, Services, Deployments, etc.)
- ✅ Understood how Kubernetes works behind the scenes
- ✅ Built and deployed a full working app
- ✅ Injected configs, secrets, and added health checks

🎉 That's a solid foundation.

Now, here's what to do next:

1

PHASE 1 **Get Comfortable With Real-World YAML**

Next Moves:

- ✅ Practice writing YAML without copy-pasting
- ✅ Try creating:
  - Deployments with different replicas
  - Services with different types (ClusterIP, NodePort, LoadBalancer)
  - Pods with mounted ConfigMaps and Secrets as files

📌 Use kubectl explain to explore syntax:

```
~$ kubectl explain deployment.spec.template.spec.containers
```

```
KIND:       Deployment
VERSION:    apps/v1

FIELD:      containers <[]Object>

DESCRIPTION:
  List of containers belonging to the pod. Containers cannot currently
  be added or removed. There must be at least one container in a Pod.
  Cannot be updated.

  Each container in a pod must have a unique name (DNS_LABEL).
  Cannot be updated.
```

2 PHASE 2 Explore Kubernetes Tools (In-Cluster + Local)

What to Learn:

- Helm (Kubernetes package manager)
- kubectl port-forward and exec
- K9s (terminal UI for cluster management)
- Lens (GUI dashboard for clusters)

Try

```
~$ kubectl exec -it <pod-name> -- /bin/sh
```

📌 Works if the container has /bin/sh (common in Alpine, Debian). May fail in images like NGINX or BusyBox without a shell.

3 PHASE 3 Learn Cluster-Level Resources

Suggested Topics:

- Namespaces (multi-team/project setups)
- Resource limits & requests
- Taints & tolerations
- Node affinity / anti-affinity
- Network policies

These help when you're working in larger teams or production clusters.

4 PHASE 4 Focus on Kubernetes Security

Start With:

- Role-Based Access Control (RBAC)
- Secrets best practices
- Pod Security Standards
- Admission controllers
- Audit logging

📌 Take the **Kubernetes Security Basics** course (like KCNA/KCSA if you plan to certify)

KCSA Course: >

KCNA Course: >

5 PHASE 5 Practice with Real Clusters

Ways to Try:

- Use Minikube or kind locally
- Spin up clusters on:
  - GKE (Google Cloud)
  - EKS (AWS)
  - AKS (Azure)
- Try Kubernetes playgrounds (like KodeKloud)

KodeKloud Playgrounds >

Final Tips

- Build mini-projects:
  - A simple web app + DB
  - A backend + frontend combo
- Read the Kubernetes Docs — they're excellent
- Use `kubectl get all -A` and `describe` often
- Watch logs: `kubectl logs <pod-name>`

Optional: Go Beyond

IF YOU LIKED THIS...	YOU MIGHT WANT TO...
YAML + basics	Try Helm or Kustomize
Pods + configs	Try StatefulSets and Persistent Volumes
Services + traffic	Explore Ingress controllers and DNS
Manual deployments	Build CI/CD with ArgoCD or Flux
Core concepts solid	Go for CKA or KCNA certification

You've Got the Foundation — Now Build on It

Explore These 5 Courses to Become a Certified Kubeastronaut

KCNA

KCSA

CKS

CKAD

kubestronaut

Become a Kubestronaut in 2025 with Mumshad

CKA

Quiz answers

Q1: Quick Recall Test

Answer: etcd, Scheduler, API Server, Controller Manager

Q2: Think Like Kubernetes

Answer: The kubelet — it runs on the Node and handles container startup.

Q3: Quick Test (Fill-in-the-blank style):

Answer: ClusterIP, LoadBalancer, ExternalName

Q4: Check Your Understanding:

Answer: All Pods in the `staging` namespace that are labeled `env=prod`

Q5: Bonus Challenge (Think Like a Pro)

Answer: Lists all **running Pods** in the dev namespace with the label tier=frontend or tier=backend.

Q6: Quick Review

Answer: ReplicaSet, Deployment, Pod Template

Q7: Quick Review

Answer: Secret, Volume, No

Q8: Quick Review

Answer: Readiness, Startup, Liveness

My name is Mumshad Mannambeth and I'm the founder of KodeKloud. In 2018, I published my first online course. My goal was to make complex DevOps technologies simple and easy to understand. In July 2019, I launched KodeKloud to provide an immersive learning experience to students all over the world.

The key to achieving this goal was the seamless integration between video lectures and hands-on labs. I didn't want the students to just learn the theory, but to truly understand the technology through practice.

The students recognized these efforts. By July 2021, KodeKloud reached 200,000 students. Over 750,000 students around the world have taken my courses.

During this period of fast growth, I built a team of dedicated experts who helped KodeKloud become what it is today. A smooth, easily accessible and affordable educational platform that helps students become DevOps experts.

Mumshad Mannambeth

Founder & CEO of Kodekloud

Why KodeKloud?

90+ Courses | **700+** Hands-on Labs | **1,000,000+** Students | **4.8** Rating

Our Mission

Our mission at KodeKloud is to foster DevOps excellence and mastery. We're dedicated to equipping learners with hands-on, practical skills for digital transformation, emphasizing efficient learning, timely skill application, and effective teaching methods to shape the future of technology professionals.

KodeKloud is dedicated to empowering students with the everything they need to learn, develop, and change the world.

(4.8) based on 900 reviews

Trustpilot

4.8 out of 5 stars