Linux Fest 2025

Docker & Kubernetes Fundamentals

Mohammad Fatemi - Saman Hosseini

Agenda

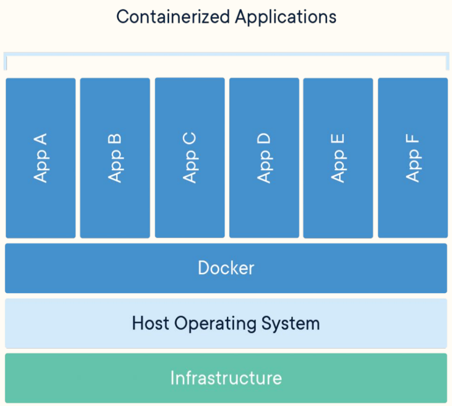
1. Container Fundamentals
2. Docker Basics
3. Docker Advanced Topics
4. Kubernetes Fundamentals
5. k8s Configuration & Networking
6. k8s Advanced Concepts

Container Fundamentals

What Container?

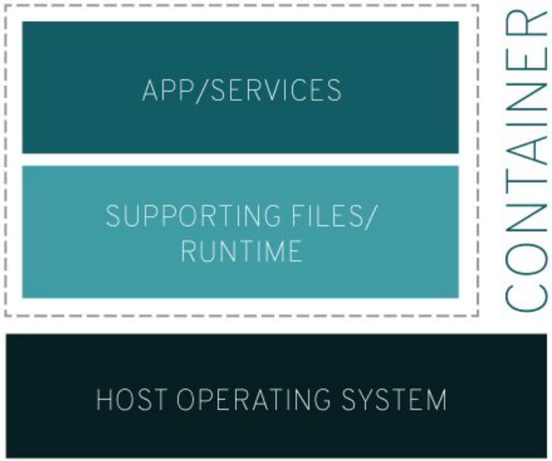
docker.com

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.



redhat.com

A Linux® container is a set of 1 or more processes that are isolated from the rest of the system. All the files necessary to run them are provided from a distinct image, meaning Linux containers are portable and consistent as they move from development, to testing, and finally to production.



cloud.google.com

Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services.

Why Container?

- **Consistency**:
  - Same environment from development to production
  - Consistent deployment patterns
- **Portability**:
  - Run anywhere - laptop, cloud, on-premises
  - Focus on code, not environment setup
- **Isolation**:
  - Applications run independently without conflicts
  - Enable independent service deployment
- **Resource Efficiency**:
  - Lower overhead than VMs
- **Speed**: Start in seconds vs minutes for VMs
- **Scalability**:
  - Easy to scale horizontally
  - Better resource utilization
- **Versioning**:
  - Container images are immutable and versioned
  - Easier rollbacks and recovery

When Container?

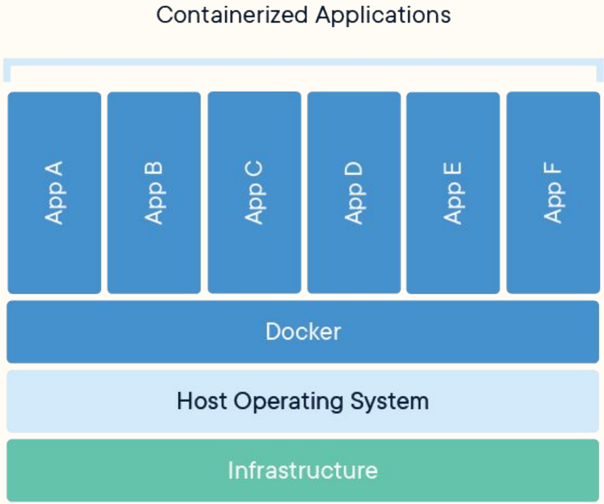
- **1979**: Unix V7 introduces chroot, the first isolation mechanism
- **2000**: FreeBSD Jails extends isolation for file system, users, and networking
- **2001**: Linux VServer project begins, providing resource isolation

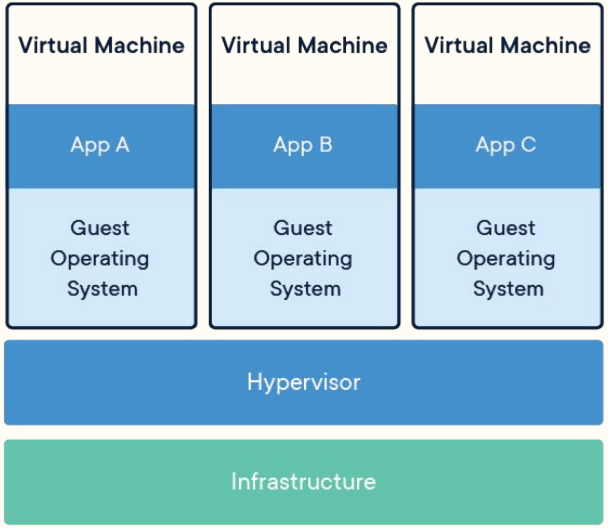
- **2006**: Process Containers introduced (later renamed cgroups)
- **2008**: LXC (Linux Containers) combines cgroups and namespaces
- **2011**: Warden container system developed by CloudFoundry

- **March 2013**: Docker released as open-source project
- **2014**: Google open-sources Kubernetes, based on internal "Borg" system
- **2015**: Open Container Initiative (OCI) established to standardize containers

- **2016**: Docker implements OCI standards
- **2017**: Kubernetes becomes mainstream with cloud provider support
- **2018**: Docker donates containerd to CNCF
- **2019**: Kubernetes graduates from CNCF
- **2022+**: Standardization around OCI and CRI (Container Runtime Interface)

Container vs VM





The hypervisor creates an abstraction layer allowing the VM to access **CPU**, **memory**, and **storage**.

Each VM includes a **full copy of an operating system**, the application, necessary binaries and libraries – taking up tens of GBs. VMs can also be slow to boot.

Source: https://www.docker.com/resources/what-container/

Container vs VM contd.

Feature	Container	Virtual machine
Operating system	Shares the host operating system's kernel	Has its own kernel
Portability	More portable	Less portable
Speed	Faster to start up and shut down	Slower to start up and shut down
Resource usage	Uses fewer resources	Uses more resources
Use cases	Good for portable and scalable applications	Good for isolated applications

https://www.reddit.com/r/explainlikeimfive/comments/spmpzh/eli5_what_are_containers_and_how_do_they_work/

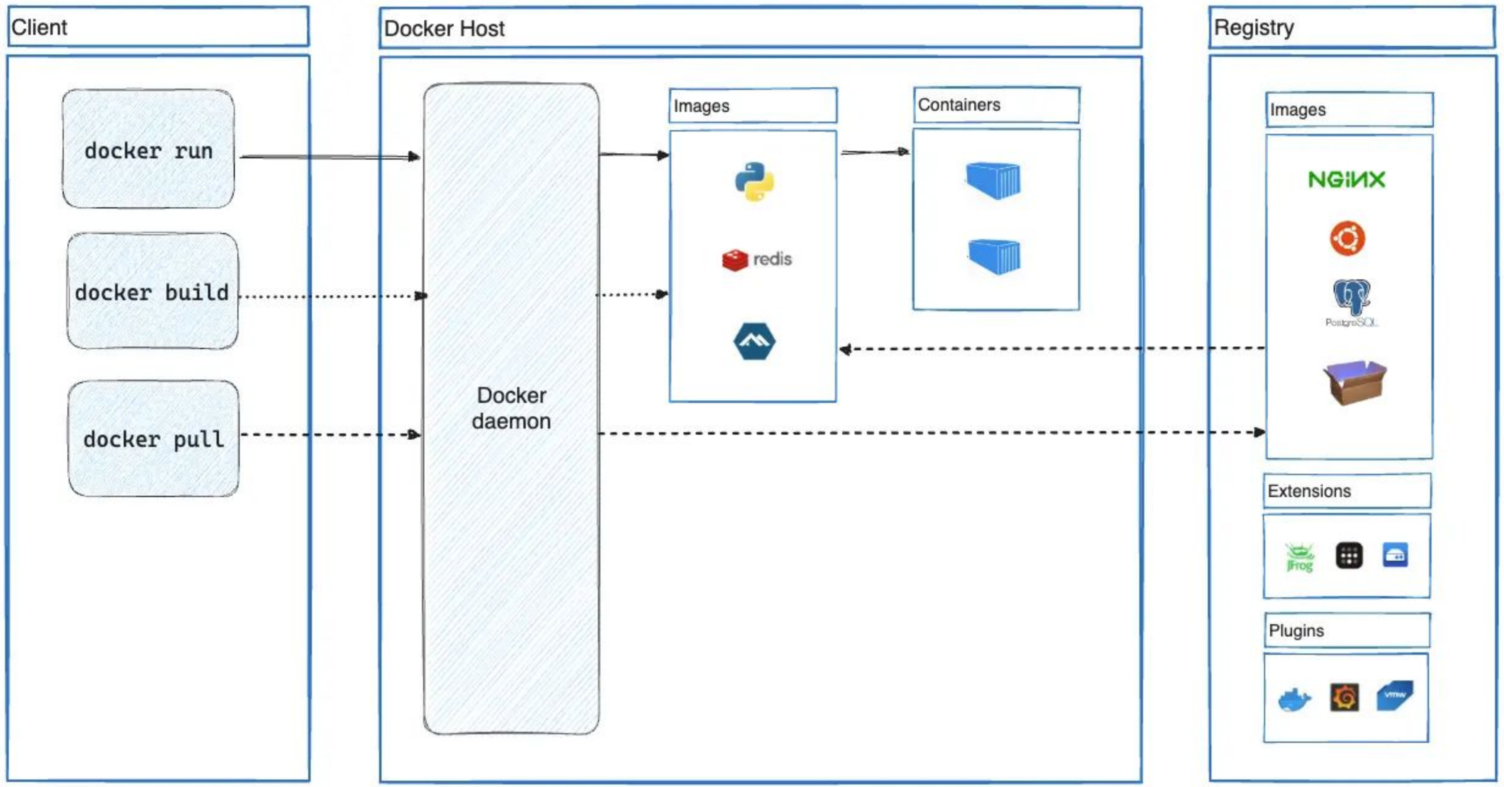
How Container?

It's all about isolation

- Namespaces: Isolation of system resources
  - **MNT**: Filesystem mounts
    - Isolated view of the namespace
  - **PID**: Process isolation
    - Isolated view of running processes
  - **NET**: Network interfaces
    - Provide isolated network environment
  - **UTS**: set namespace Hostname
  - **IPC**: Inter-process communication
  - **USER**: User and group IDs
    - Allows root permission in container but not host
  - **Time**: Custom time setting
    - Fairly new and not widely supported
  - **Control Groups (cgroups)**: Resource limits
    - **CPU, memory, I/O constraints**
    - **Prevents container from overwhelming host**

https://docs.docker.com/get-started/docker-overview/#docker-architecture

What is Docker?



Docker Image

A read-only template with instructions for creating a Docker container

- You can create your own images with help of a Dockerfile
- Consists of **layers** and is **immutable**
  - You can add layers on top of existing layers
  - Each layer is a change to the container filesystem
- Are stored in registries (i.e. Dockerhub)
- Layers adhere to Dockerfile syntax

Take Postgres image as an example.

Docker Registry

Storage and **distribution** solution for Docker images

We have **private** and **public** registries

- Public like Dockerhub or GCR
- Private self-hosted registries
- For sensitive and proprietary applications we should use **private** registries
- For access control use **private** registries

Registry Mirrors

- Caching proxy for registries
  - Faster image pulls
  - Reduced dependency on upstream registries
  - Bypass network/political restrictions :D

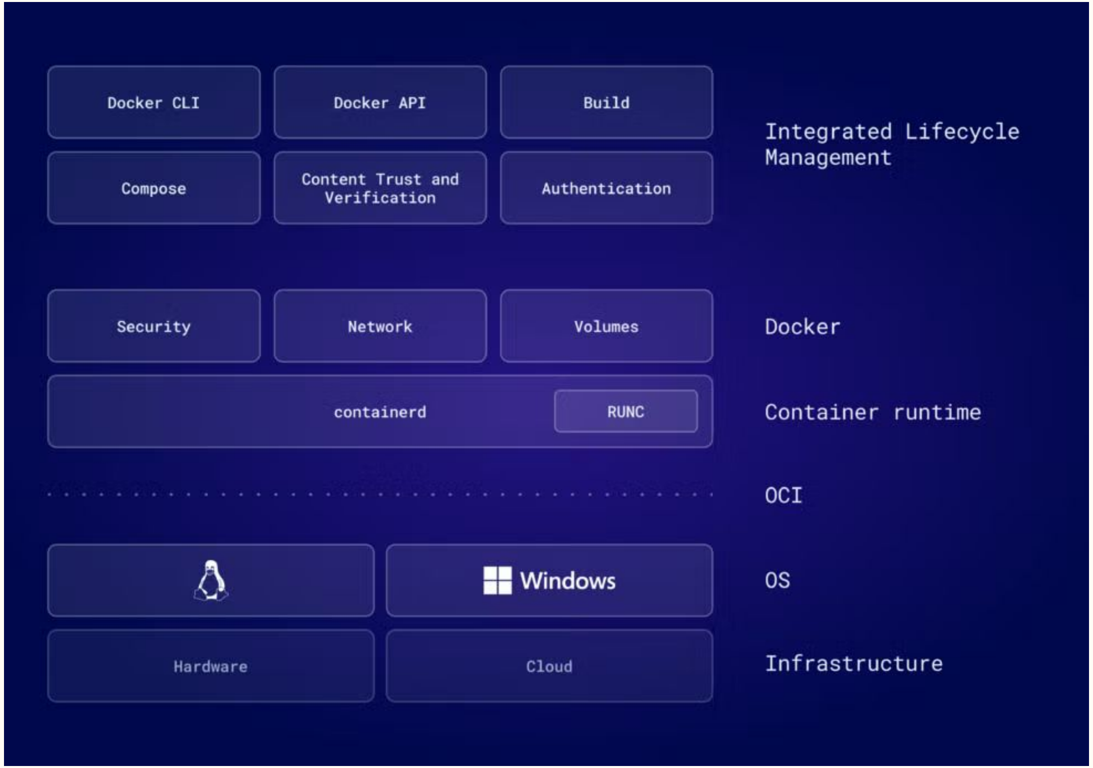
Docker Daemon

Is the core background service of Docker

- Core background service (daemon) of Docker platform
- Manages all container-related operations and resources
  - Images, Containers, Networks, Volumes
  - Manages container runtime operations (containerd)
- Listens for Docker API requests
- Handles container lifecycle management
- Runs as dockerd process on the host system

Docker Daemon cont.

1. **docker** run busybox (client)
2. **dockerd**
    a. Parse and validate
    b. Pull Image if not present locally
3. **containerd** creates container from image
    a. Setup env and namespaces and isolations
4. Delegate running container to **runc**
5. Monitor and control container in **containerd**



Docker Client

The `docker` cli used to interact with docker components

1. Installed with docker engine
   a. Autocomplete install guide
   b. On macos orbstack is better :D
2. Get started by running `docker --help`
3. Read the cheatsheets:
   a. https://dockerlabs.collabnix.com/docker/cheatsheet/
   b. https://docs.docker.com/get-started/docker_cheatsheet.pdf

Installing Docker

Linux

Install using package managers (requires proxy for package manager). Link

Install binaries using instructions

Macos

Recommended:

`brew install orbstack`

Windows

Install docker-desktop

https://www.docker.com/products/docker-desktop/

Questions?

Exercise

Install docker and run hello-word image

Docker Basics

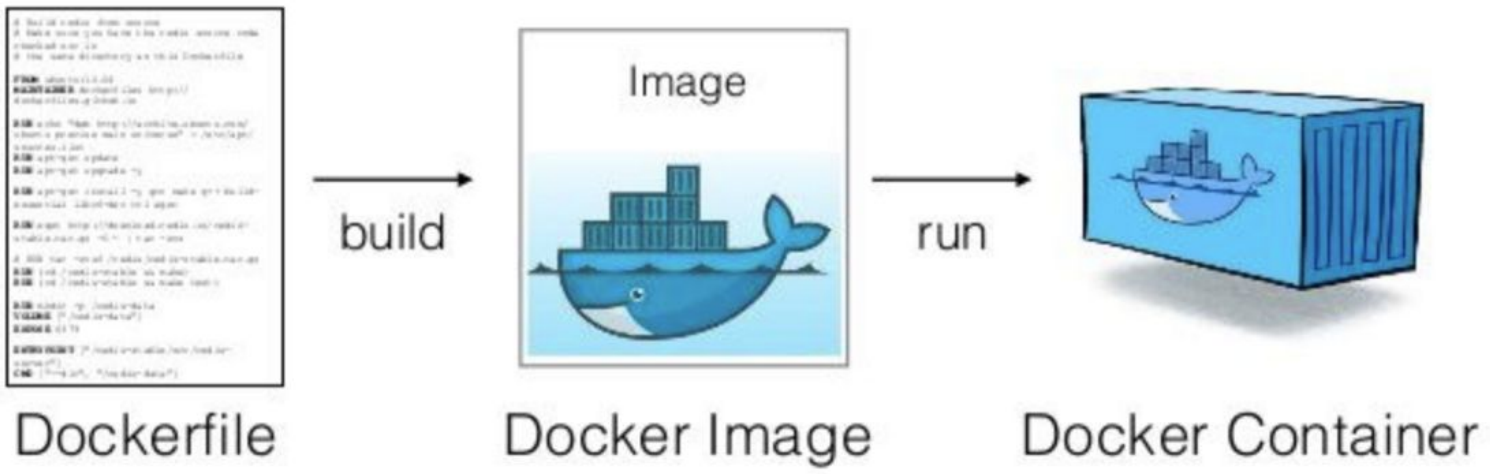
Docker Images

- **Image naming**
  - `[REGISTRY_HOSTNAME/][USERNAME/]IMAGE_NAME[:TAG]`
  - `Common tags: latest, version numbers`
- **Download image**
  - `docker pull <image>`
- **List, view images**
  - `docker images`
  - `docker inspect <image>`
- **Cleanup images**
  - `docker image prune`
  - `docker system prune`

Running Containers from Images

- **Run a Container**
  - `docker run <image>`
- **Running and Executing commands**
  - `docker run -it <image> sh`
- **Auto remove after exit**
  - `--rm flag`
  - One-off commands: `docker run --rm <image> command`
- **Commit & Save container**
- **Detached vs Interactive**

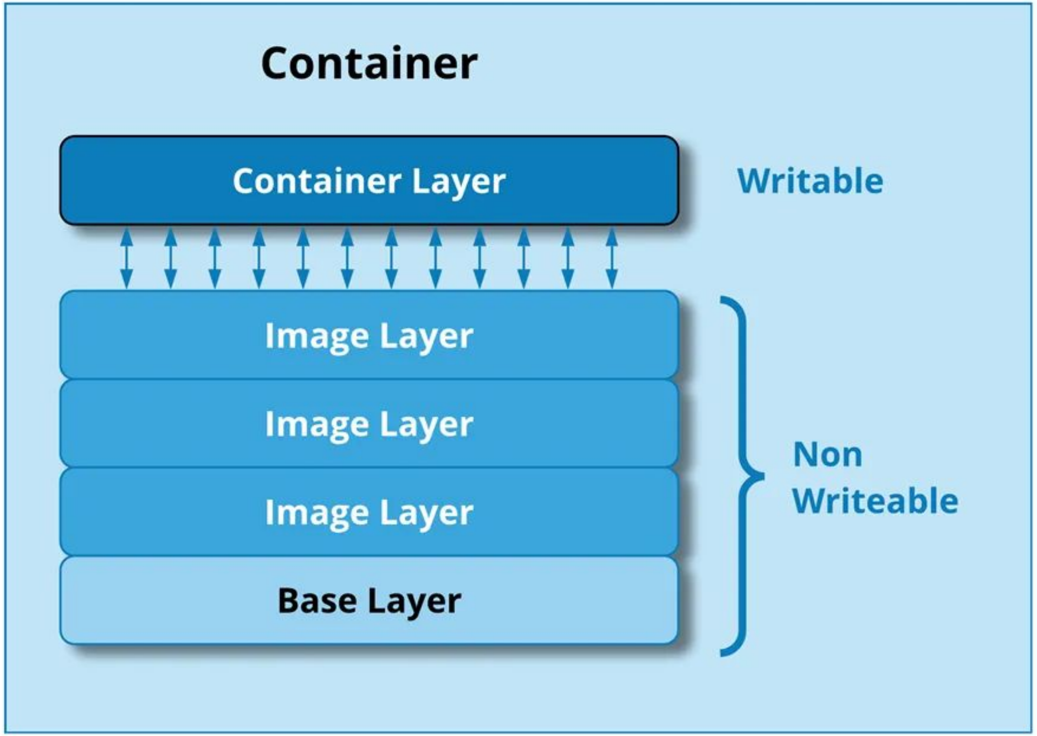
Dockerfiles



Writing Effective Dockerfiles

- **Base Image**
  - Alpine
  - Debian
  - Distroless
- **Commands**
  - RUN
  - COPY
  - ADD
  - WORKDIR
  - CMD & ENTRYPOINT
- **Context and .dockerignore**
  - Build context: All files in the current directory are sent to the Docker daemon
  - Context management: Avoid sending unnecessary files
  - .dockerignore

Image Layers



Do we need all the compile time dependencies (source codes, documentations, helper tools, etc.) to run the application?

Efficient Dockerfiles (Multi-Stage Builds)

How large is a compiled hello world Go application?

- What about the container image for this application?
- Can we reduce the image size?

Separating Runtime base image and Compile/Build base image will help **a lot**.

To do so, copy final artifacts from build stage to runtime stage.

Efficient Dockerfiles cont. (Caching)

Image Layers

- Each instruction results in a new layer
- The checksum of the final state of the container is used for caching
  - What if a layer changes?

So? **The order of instructions matter**!

Best practices

- Use .dockerignore when possible (why?)
- Group related commands in single `RUN instruction`
- Place frequently changing instructions last
- Separate dependency download and compile commands
- Consider multi-stage builds for compiled applications

Problem/Challenge

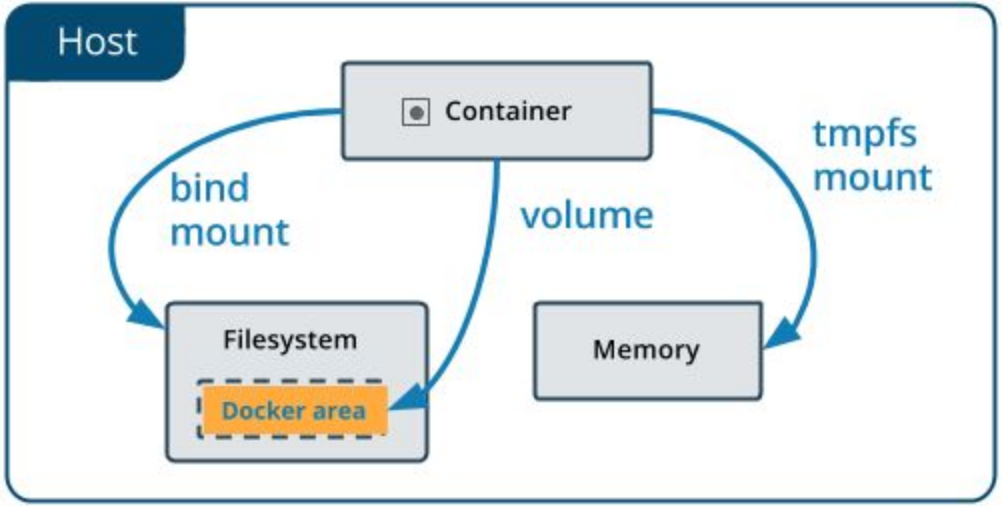
Docker Networking Basics

- **Network Drivers**
  - `bridge (default network driver)`
  - `host (remove network isolation between the container and the Docker host)`
  - `overlay`
  - `macvlan`
  - `ipvlan`
- **Creating Networks**
  - `docker network create`
- **Internal DNS & Container Name Resolution**
- **Port Mapping**

Volume Management & Persistent Storage

volume

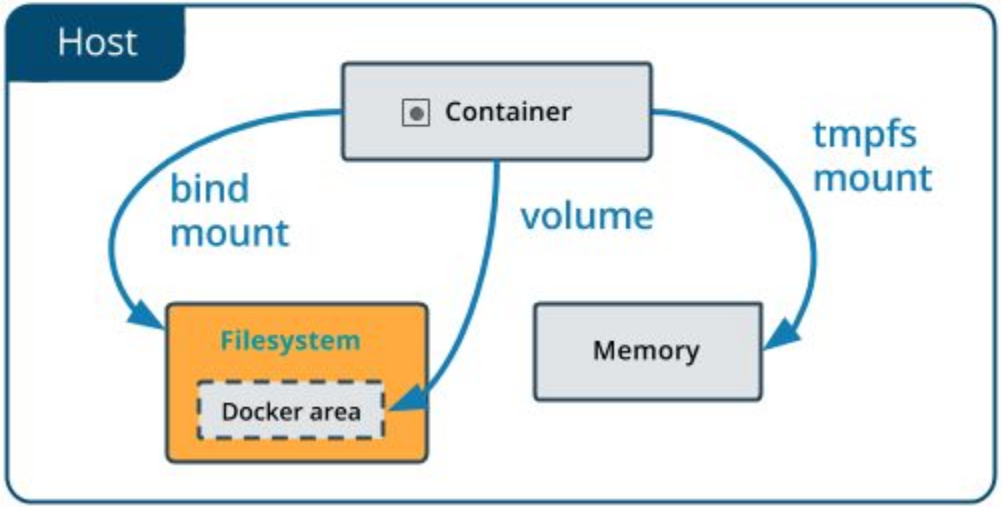
- stored in a part of the host filesystem
- managed by Docker (/var/lib/docker/volumes/ on Linux)
- the best way to persist data in Docker



Volume Management & Persistent Storage (Cont.)

bind mount

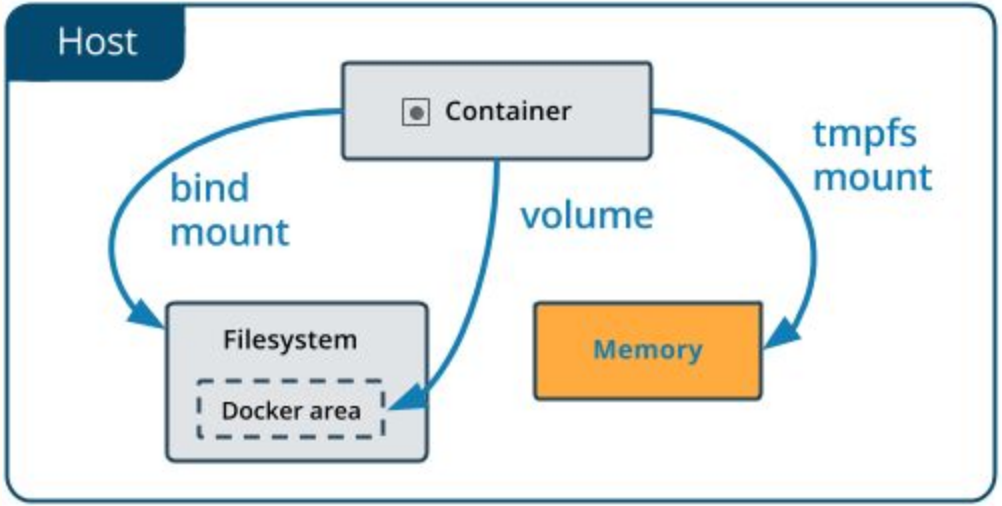
- limited functionality compared to volumes
- a file or directory on the host machine is mounted into a container
- Non-Docker processes on the Docker host can modify them at any time



Volume Management & Persistent Storage

tmpfs mount

- stored in the host system's memory only
- files written there won't be persisted
- useful to temporarily store sensitive files that you don't want to persist



Exercise

Create a Dockerfile for one of your applications in any compiled language.

Make use of the best practices for caching and multi-stage builds.

You can use 1995parham/Zang as the project to write Dockerfile for.

Advanced Containers

Docker Compose

What to do when we need multiple containers?

Imagine we have a web service consisting:

1. HTTP server on port 8080
2. PostgreSQL server on port 5432
3. Redis Server on port 6379

What’s the best way to start such application?

Docker Compose cont.

Docker Compose file

Define each container as a `service` in a file named
docker-compose.yaml

For each service, define:

- image
- ports
- volumes
- networks
- command
- etc.

You can visit the specification here. Also, consult this cheatshet for working with `docker compose` CLI.

Exercise

Do you have a multi dependency application? Try to write a docker compose file for it.

If you don't you can use this repo (don't just use the existing docker compose file :D)

Container Orchestration

What to do when we have multiple containers?

- How do we control number of containers?
- How do we handle failure recovery (restart)?
- What about networking between multiple containers?
- How do we manage deployment of individual containers?
- And so on?

"Container orchestration automatically provisions, deploys, scales, and manages containerized applications without worrying about the underlying infrastructure."

https://cloud.google.com/discover/what-is-container-orchestration

Container Orchestration cont.

What does a container provisioning platform provide?

- Provisioning and deployment
- Scaling containers up or down
    - Provide some levels of fault tolerance
- Allocating resources between containers
    - CPU
    - Memory
    - Storage
- Performance and health monitoring of the application
- Networking
    - Service Discovery
    - Manage **inter** and **intra** cluster communications
    - Load balancing

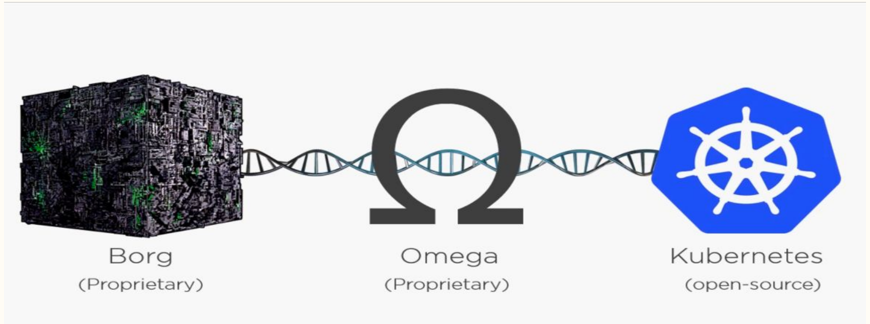
Container Orchestration cont.

History of container orchestration tools

Early days (2000s)

2014-2015

2015 - present



Container orchestration cont.

Docker compose vs Container orchestration tools

	Docker Compose	Container Orchestration Tools (e.g., Kubernetes)
Purpose	Simplifies running multi-container apps on a single host	Manages containers at scale across multiple hosts.
Use Case	Local development and small-scale setups.	Production environments with high availability.
Scaling	No built-in scaling.	Automated scaling based on demand.
Self-Healing	Not supported.	Automatically restarts failed containers.
Networking	Basic networking between containers.	Advanced networking and service discovery.
Storage	Limited persistent storage options.	Supports persistent storage across clusters.
Complexity	Simple and easy to use.	Higher learning curve and setup complexity.
Best For	Local setups and testing.	Large-scale, distributed, and production systems.

Container Security

Security best practices

- Use official and trusted docker images (why?)
  - Don’t forget that trusted images with known vulnerability is as bad as shady images.
- Avoid unnecessary tools (even shell) when possible (why?)
- Manage container privileges
  - Limit security capabilities
  - Use non-root user in Dockerfile
  - Mount read-only files as read-only :D
- SELinux and AppArmor

Container Inspection

Container Inspection

Command	Purpose	Detail Level	Performance Impact
docker inspect	Comprehensive container details	High	Low
docker ps	Container list	Medium	Very Low
docker logs	Container logs	Medium	Low
docker top	Running processes	Low	Low
docker stats	Resource usage	Real-time	Medium

Exercise

Can you find a file called “*secret.txt*” within

“*smf8/pingpong-server:latest*” docker image?

P.S. Is it safe to run it locally without any considerations in your system?

Kubernetes Fundamentals

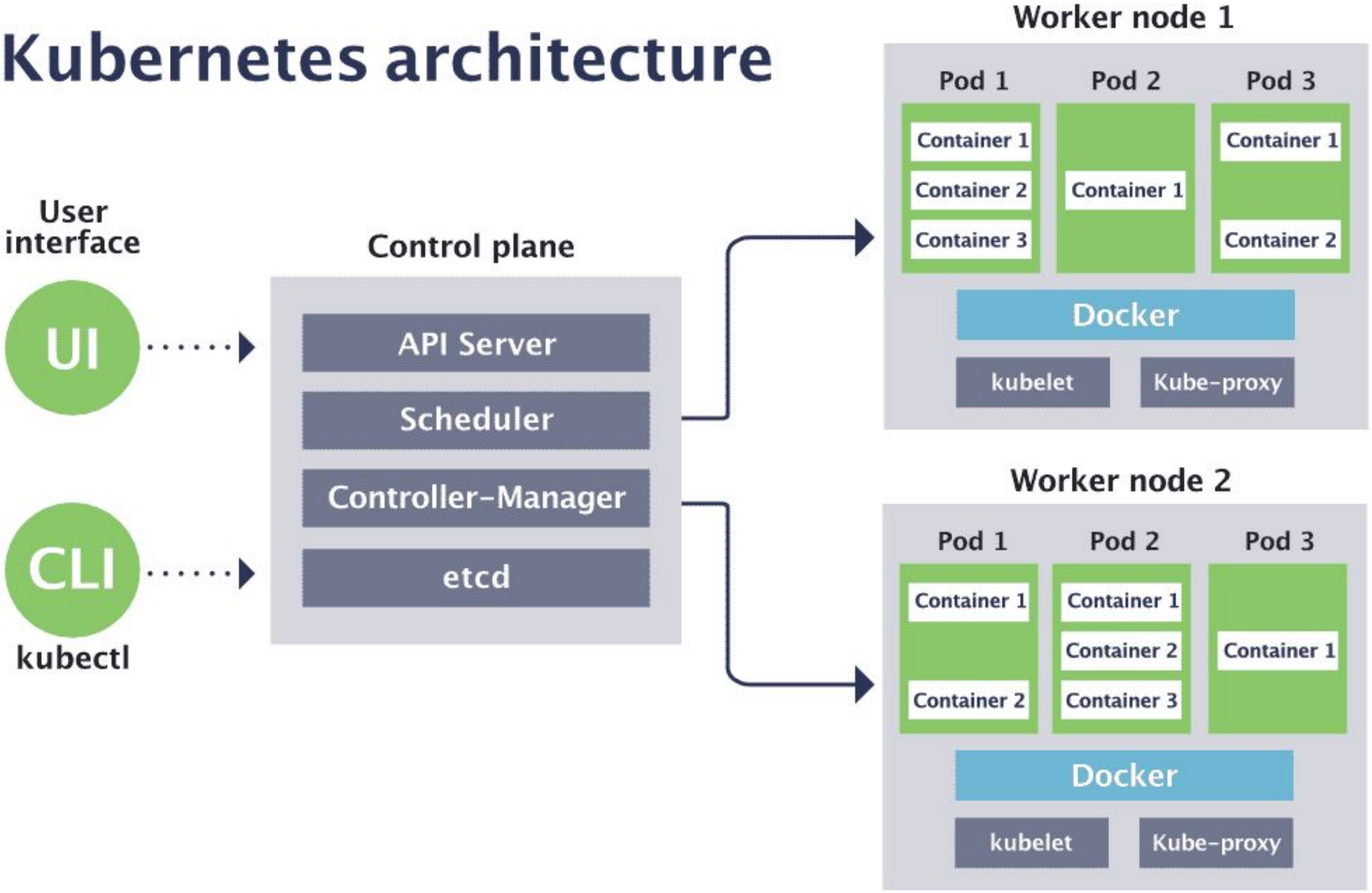
Why k8s?

- 🚢 Container Orchestration and Lifecycle Management
- ⚖️ Automated Scaling
- 🔁 Rolling Updates and Rollbacks
- 🧠 Service Discovery and Load Balancing
- 📡 Declarative Configuration and Desired State Management

Why k8s?

- 🔄 Automated Rollout of Configurations and Secrets
- 🔒 Improved Security and Isolation
- 📦 Persistent Storage Management
- 📊 Observability (Monitoring, Logging, Tracing)
- 💥 Fault Tolerance and High Availability
- 🏗️ Extensibility and Ecosystem

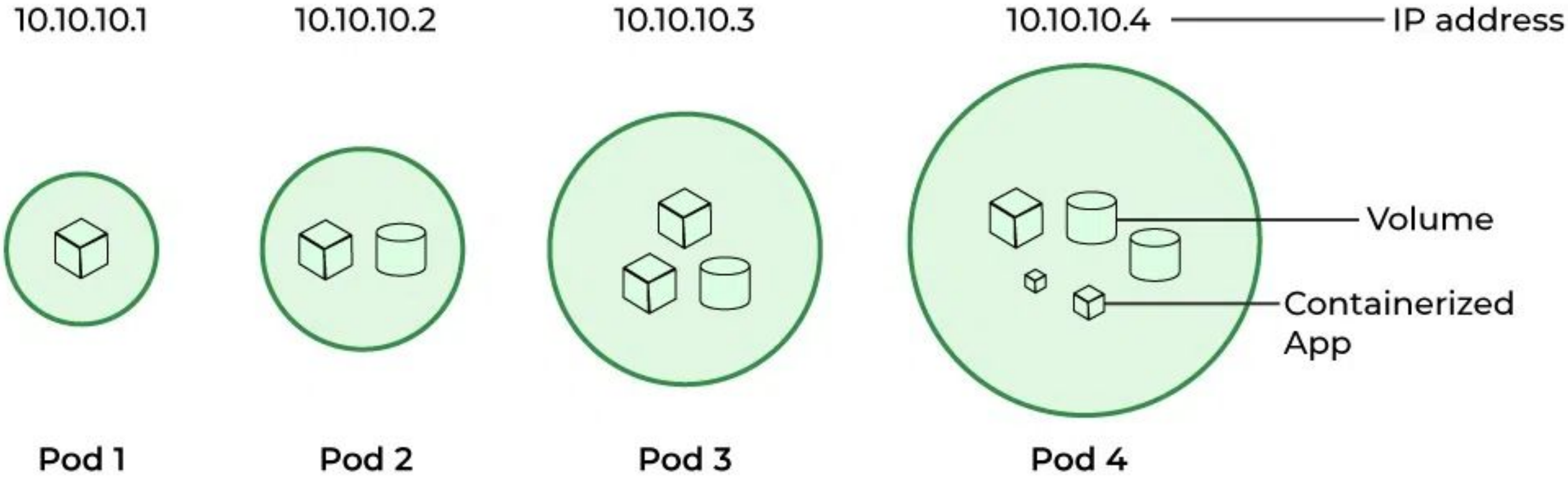
Kubernetes architecture



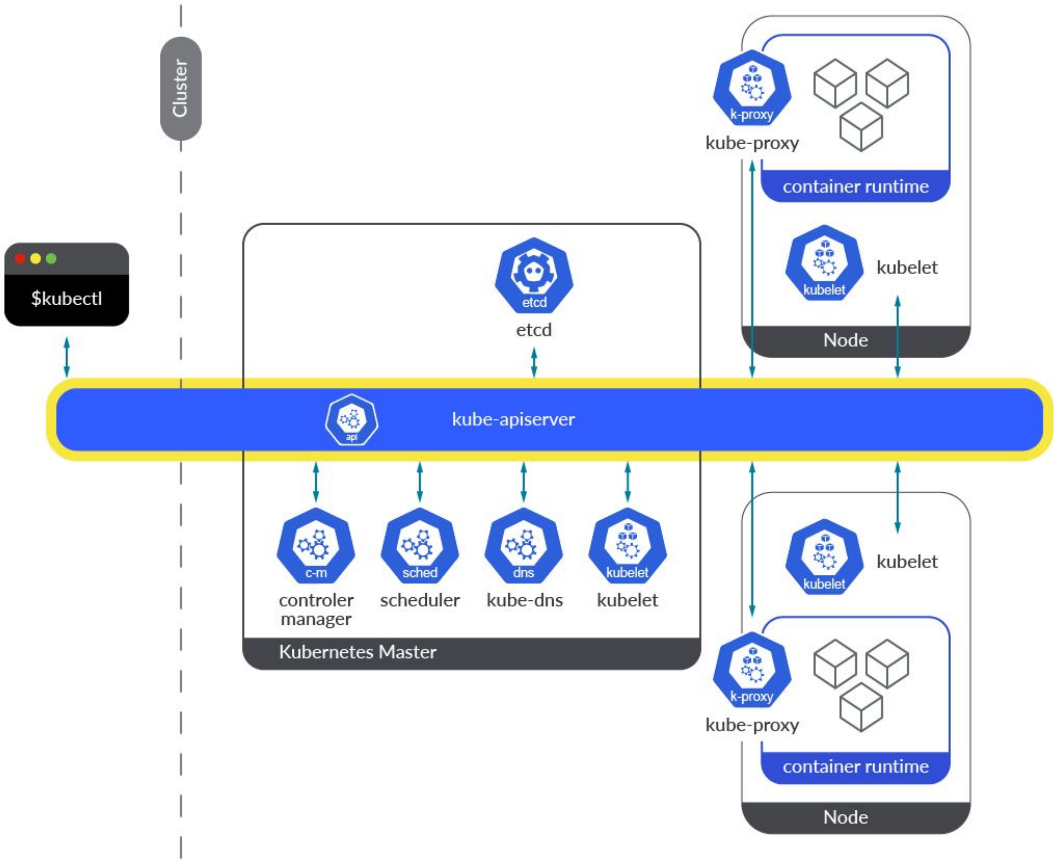
pod

- A Pod is the basic execution unit in Kubernetes
- It represents one or more tightly coupled containers that:
  - Share the same network namespace
  - Share volumes (storage)
  - Run on the same node
- Pods are ephemeral — they are designed to be created, destroyed, and recreated

Pod(s)


10.10.10.1
10.10.10.2
10.10.10.3
10.10.10.4
IP address
Volume
Containerized
App
Pod 1
Pod 2
Pod 3
Pod 4

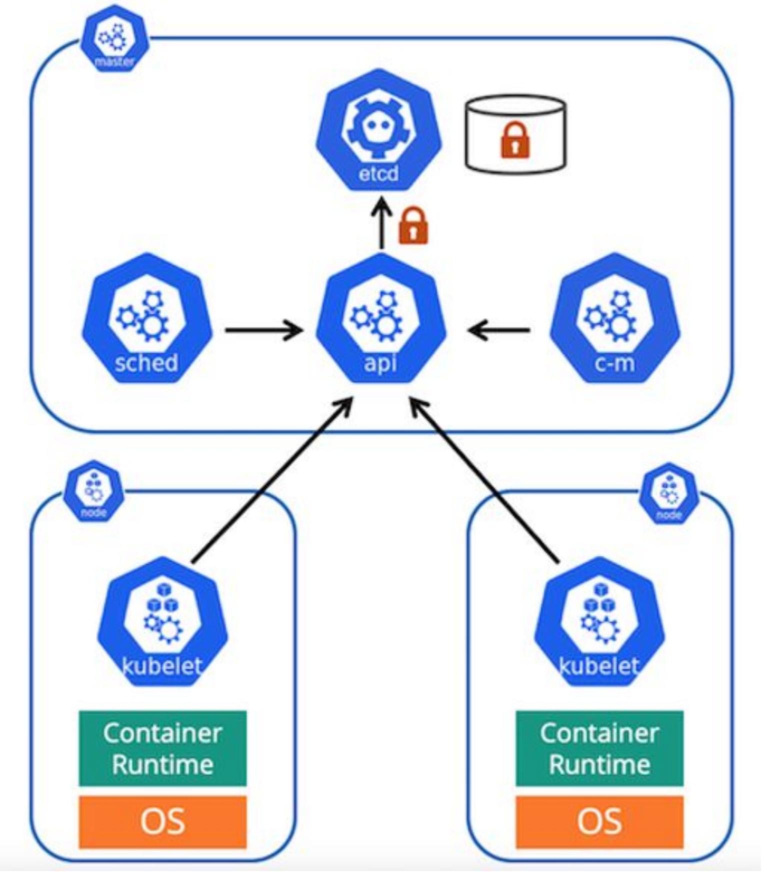
Kubernetes Control Plane (API Server)



kube-apiserver

- Frontend of the Kubernetes control plane (RESTful API)
- Accepts and validates requests (kubectl, clients, other components)
- Authenticates, authorizes, and processes API calls
- Stores all cluster state in etcd
- Communicates with all other control plane and node components

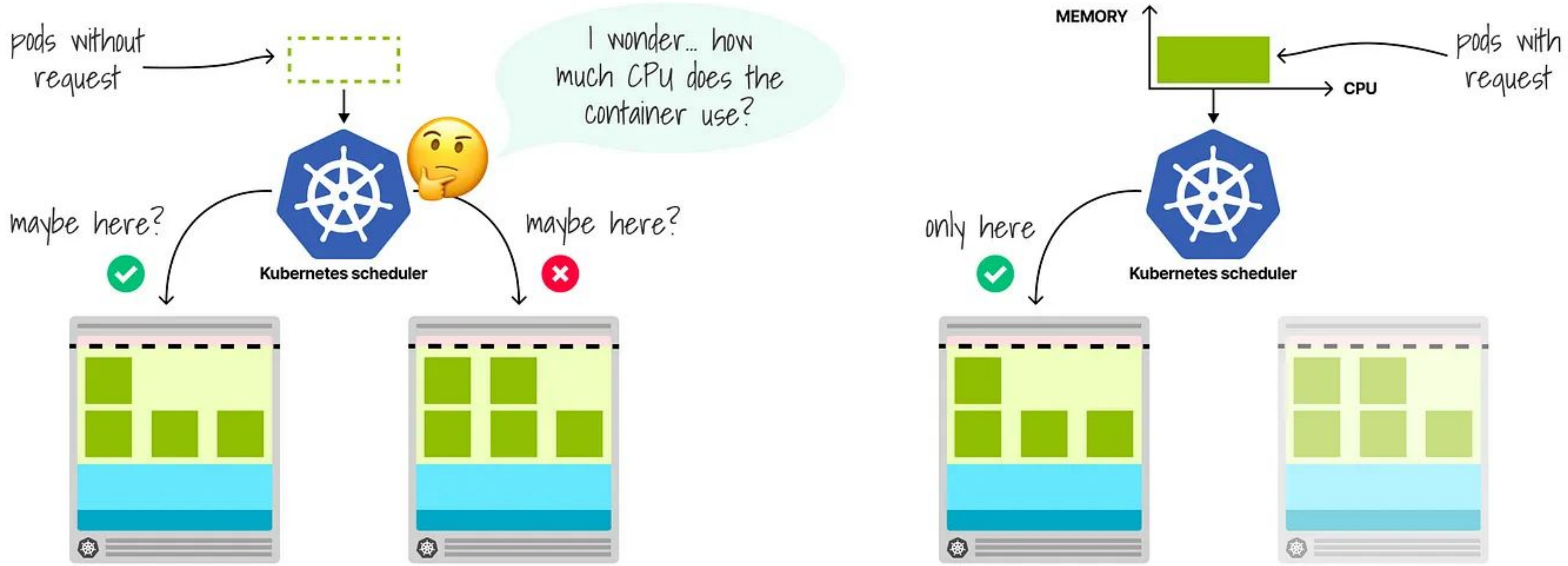
Kubernetes Control Plane (etcd)



etcd

- Distributed, consistent key-value store
- Holds the entire cluster state (config, secrets, node info, etc.)
- Highly available and fault-tolerant
- Requires regular backup for disaster recovery

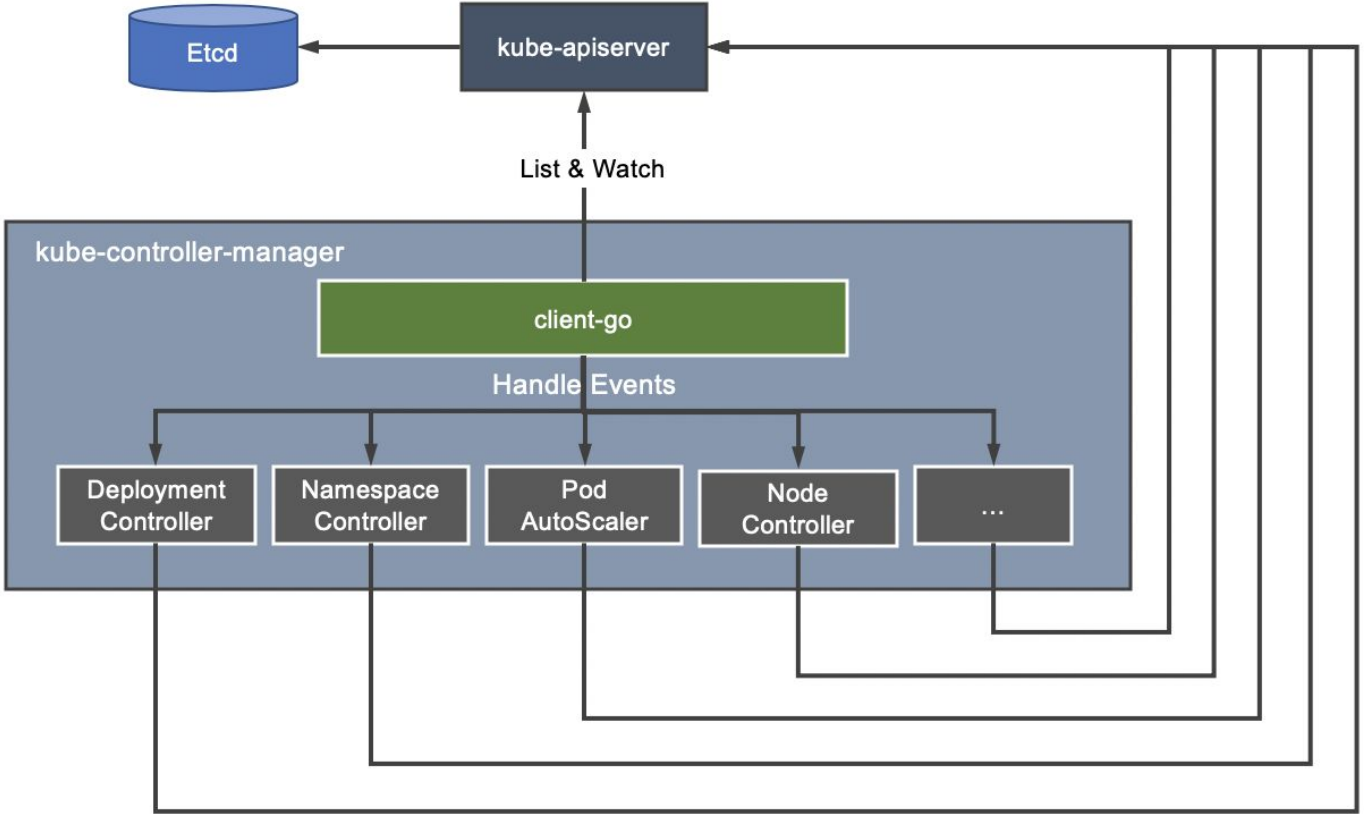
Kubernetes Control Plane (Scheduler)



kube-scheduler

- Assigns newly created pods to suitable nodes
- Scheduling decisions are based on:
  - Resource availability (CPU, memory)
  - Taints/tolerations, affinities/anti-affinities
  - Node selectors or labels
  - Pod priority, topology, etc.

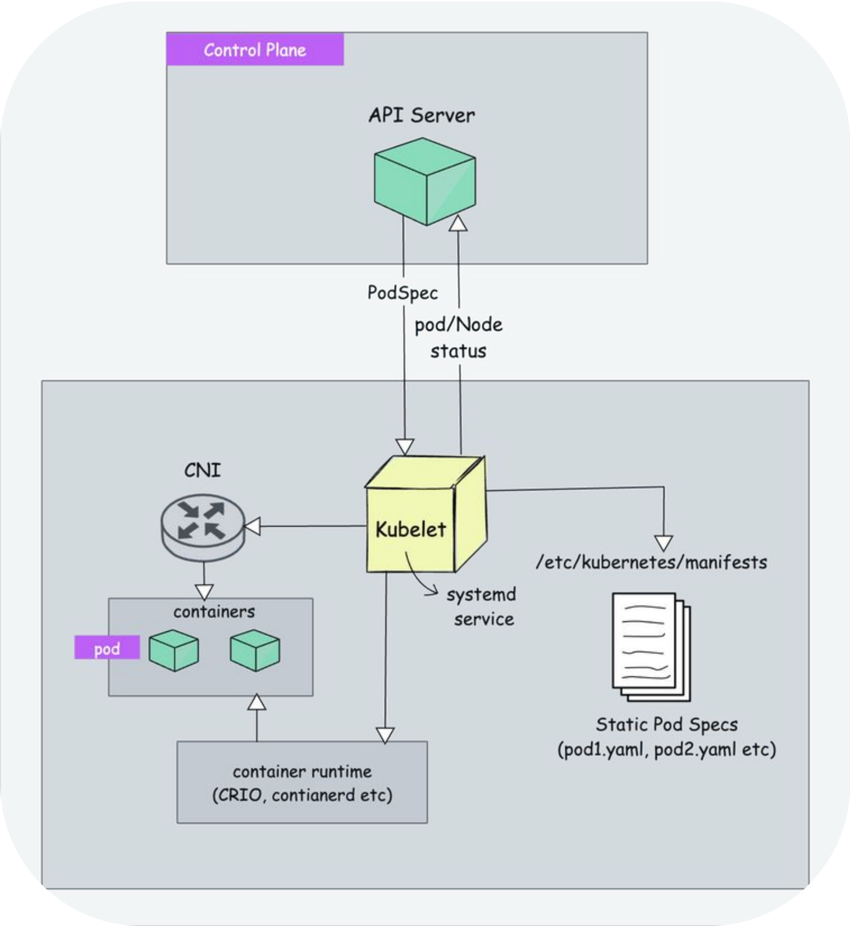
Kubernetes Control Plane (Controller Manager)



kube-controller-manager

- Node Controller – Monitors node health (e.g., heartbeat)
- Replication Controller – Ensures desired number of pod replicas
- Endpoint Controller – Manages Endpoints for Services
- Namespace Controller – Handles lifecycle of namespaces
- Service Account & Token Controller – Creates default accounts and credentials

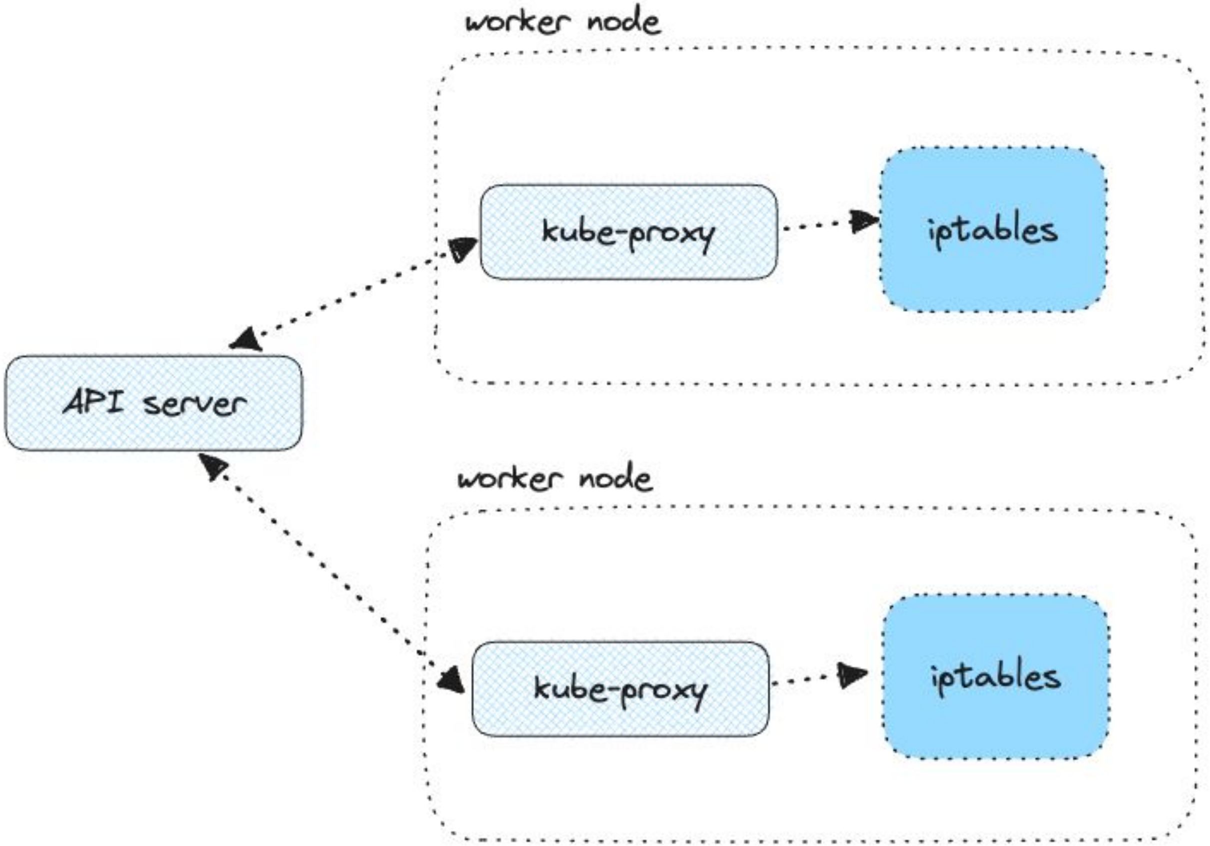
Kubernetes Worker Node (Kubelet)



kubelet

- Agent that runs on each worker node
- Registers the node with the cluster
- Monitors and manages the pod lifecycle
- Ensures containers are running in desired state
- Talks to the container runtime (e.g., containerd, CRI-O)

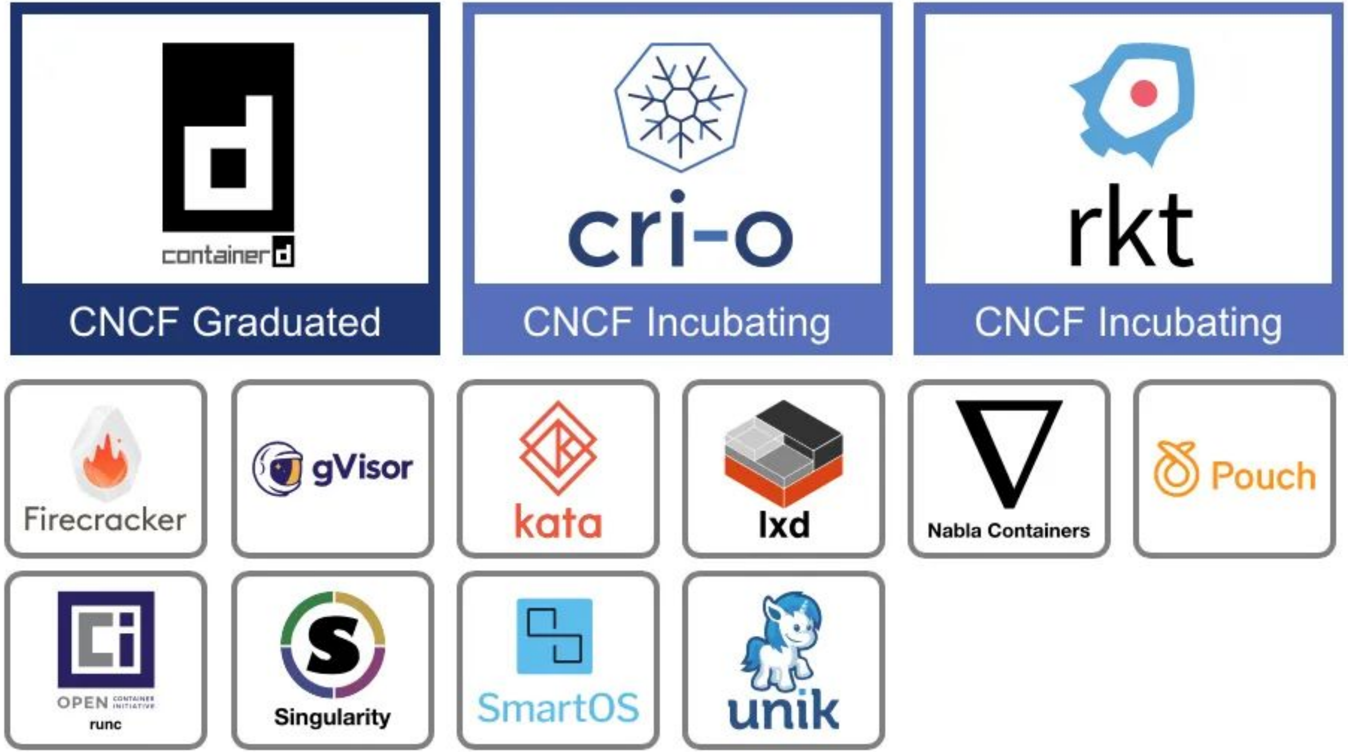
Kubernetes Worker Node (Kube Proxy)



kube-proxy

- Handles network routing for Services
- Implements NAT and forwarding rules (iptables, IPVS)
- Ensures communication between pods across nodes
- Load balances traffic to service endpoints

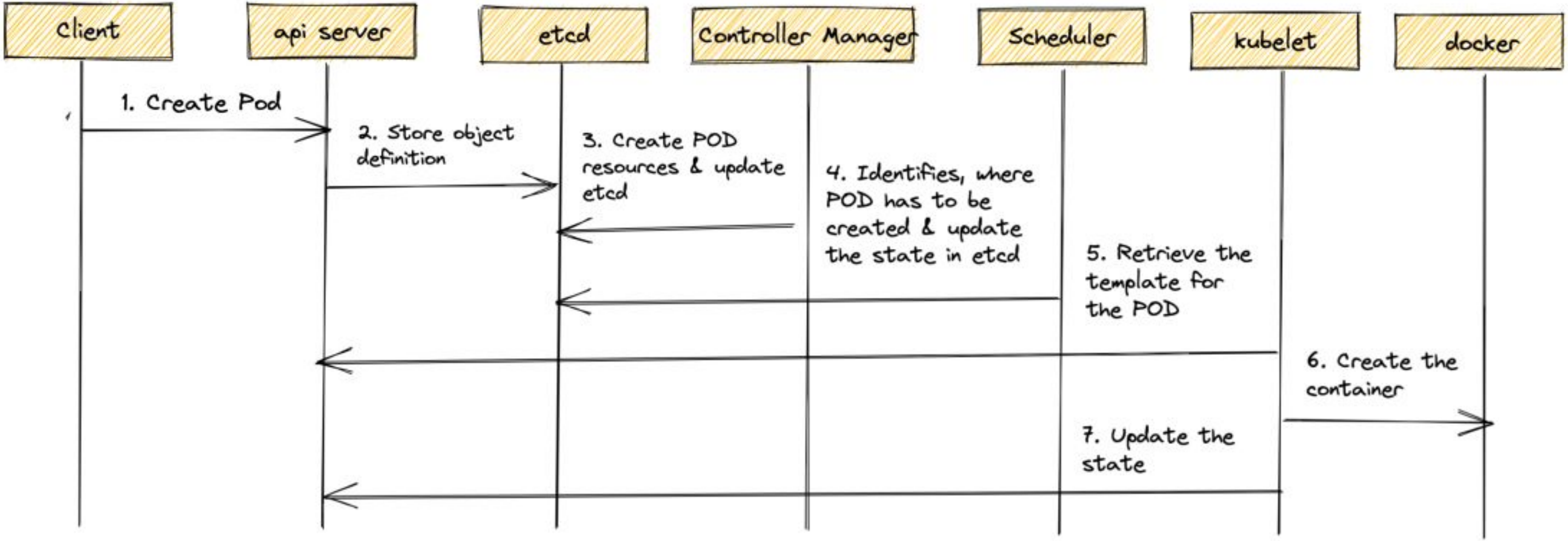
Kubernetes Worker Node (Container Runtime)



container runtime

- Responsible for running containers
- Complies with Kubernetes CRI (Container Runtime Interface)
- Pulls container images, starts/stops containers, manages logs

Kubernetes Flow

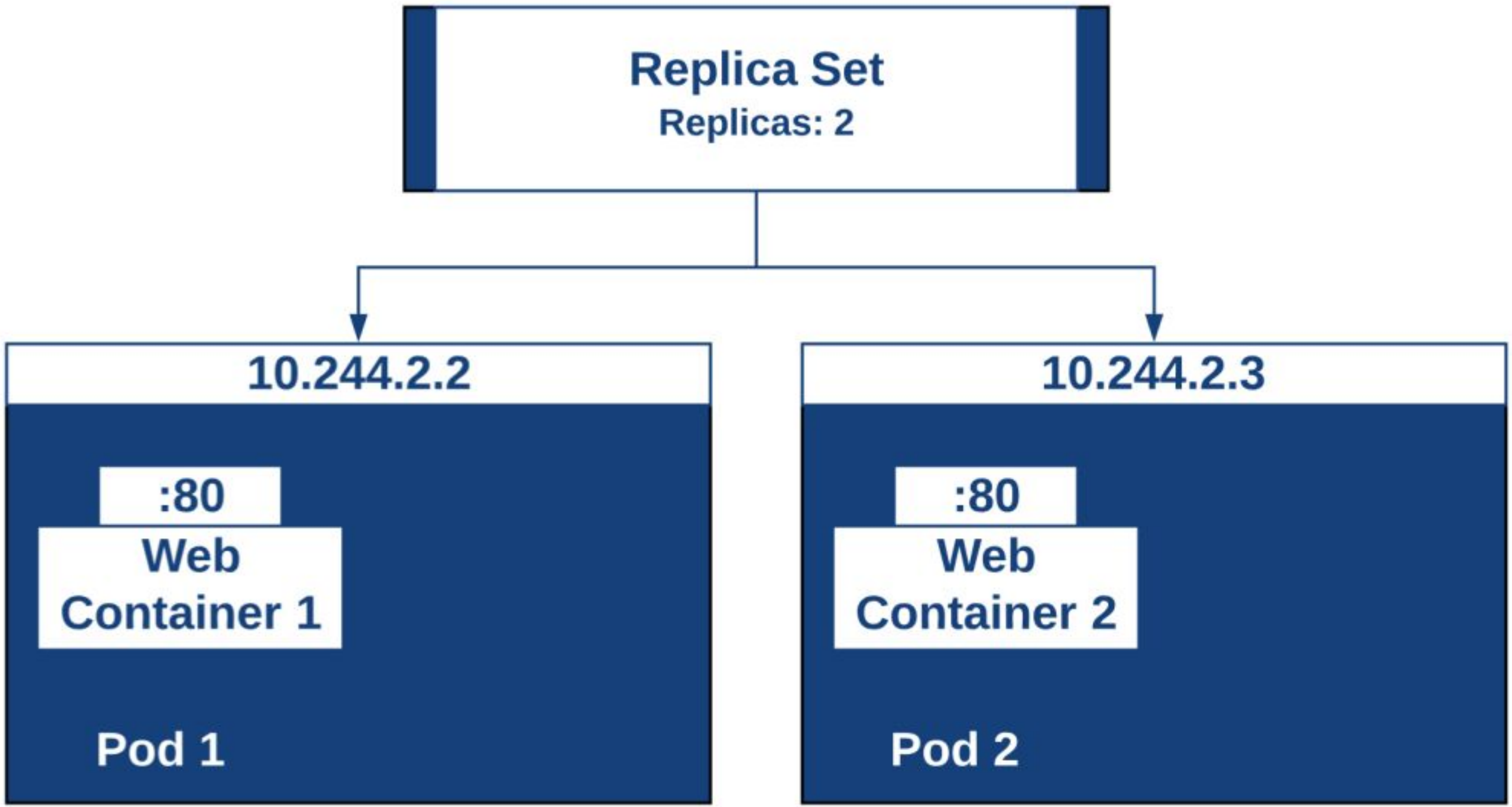


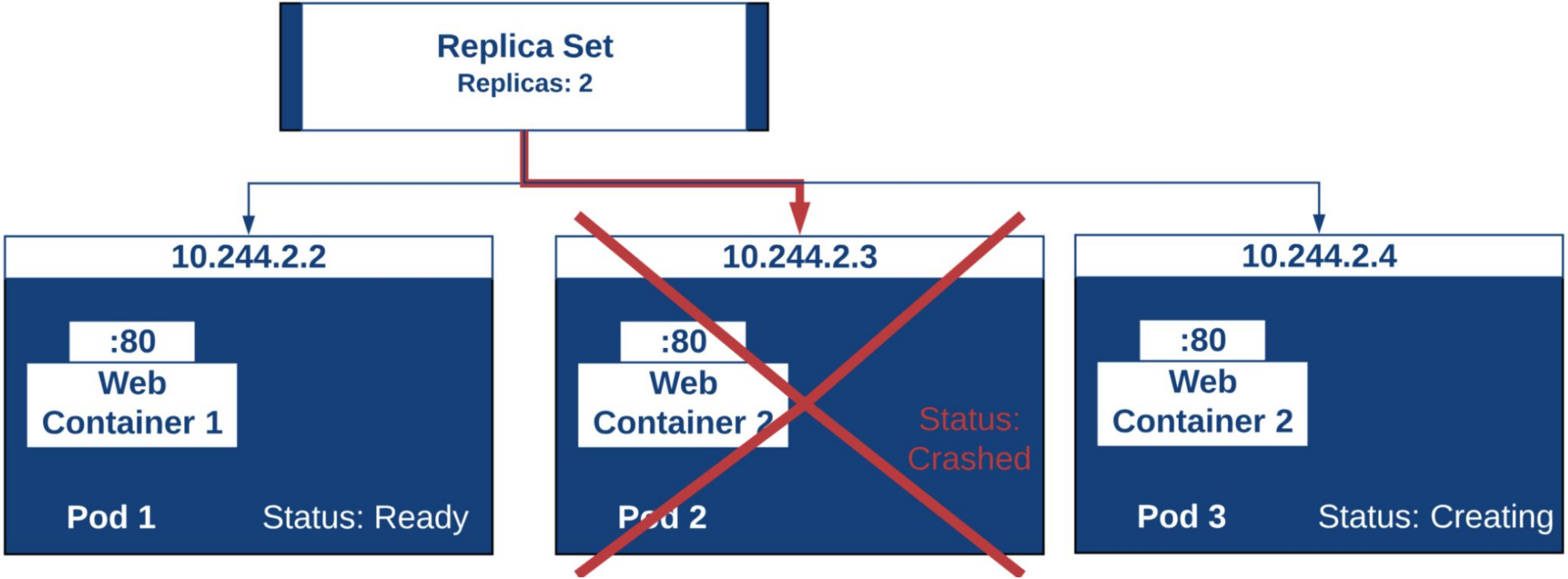
pod

- A Pod is the basic execution unit in Kubernetes
- It represents one or more tightly coupled containers that:
  - Share the same network namespace
  - Share volumes (storage)
  - Run on the same node
- Pods are ephemeral — they are designed to be created, destroyed, and recreated

ReplicaSet

- Ensures a specified number of pod replicas are running at all times
- Automatically replaces failed pods to maintain the desired count
- Used indirectly through Deployments (rarely managed directly)
- Selector-based – matches pods with specific labels to manage them

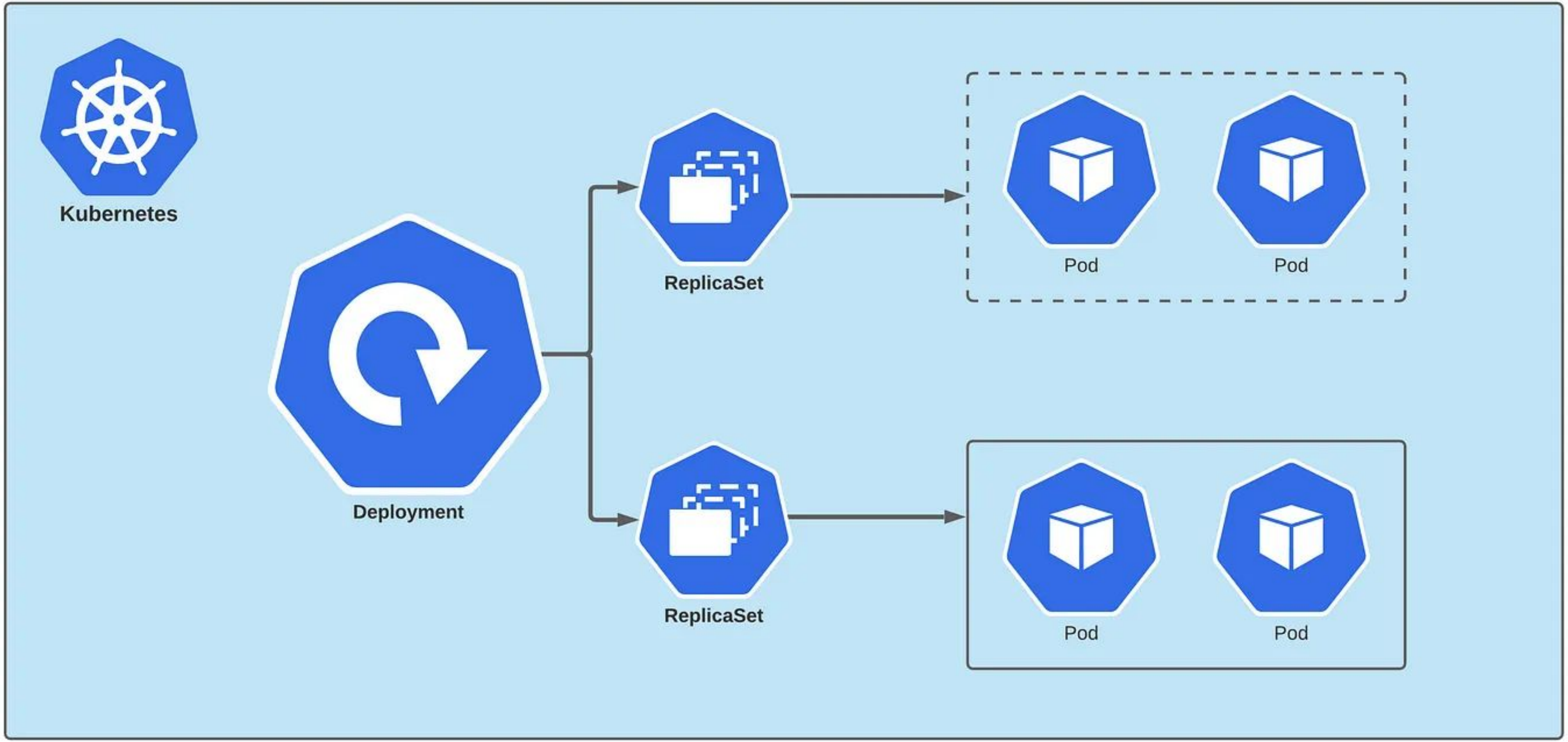

Replica Set
Replicas: 2
10.244.2.2
:80
Web
Container 1
Pod 1
10.244.2.3
:80
Web
Container 2
Pod 2



Replica Set
Replicas: 2
10.244.2.2
:80
Web
Container 1
Pod 1
Status: Ready
10.244.2.3
:80
Web
Container 2
Status:
Crashed
Pod 2
10.244.2.4
:80
Web
Container 2
Pod 3
Status: Creating

Deployment

- Manages ReplicaSets and provides declarative updates to Pods and ReplicaSets
- Supports rolling updates, rollbacks, and version history
- Ideal for stateless applications
- Automatically replaces failed pods and scales up/down as needed

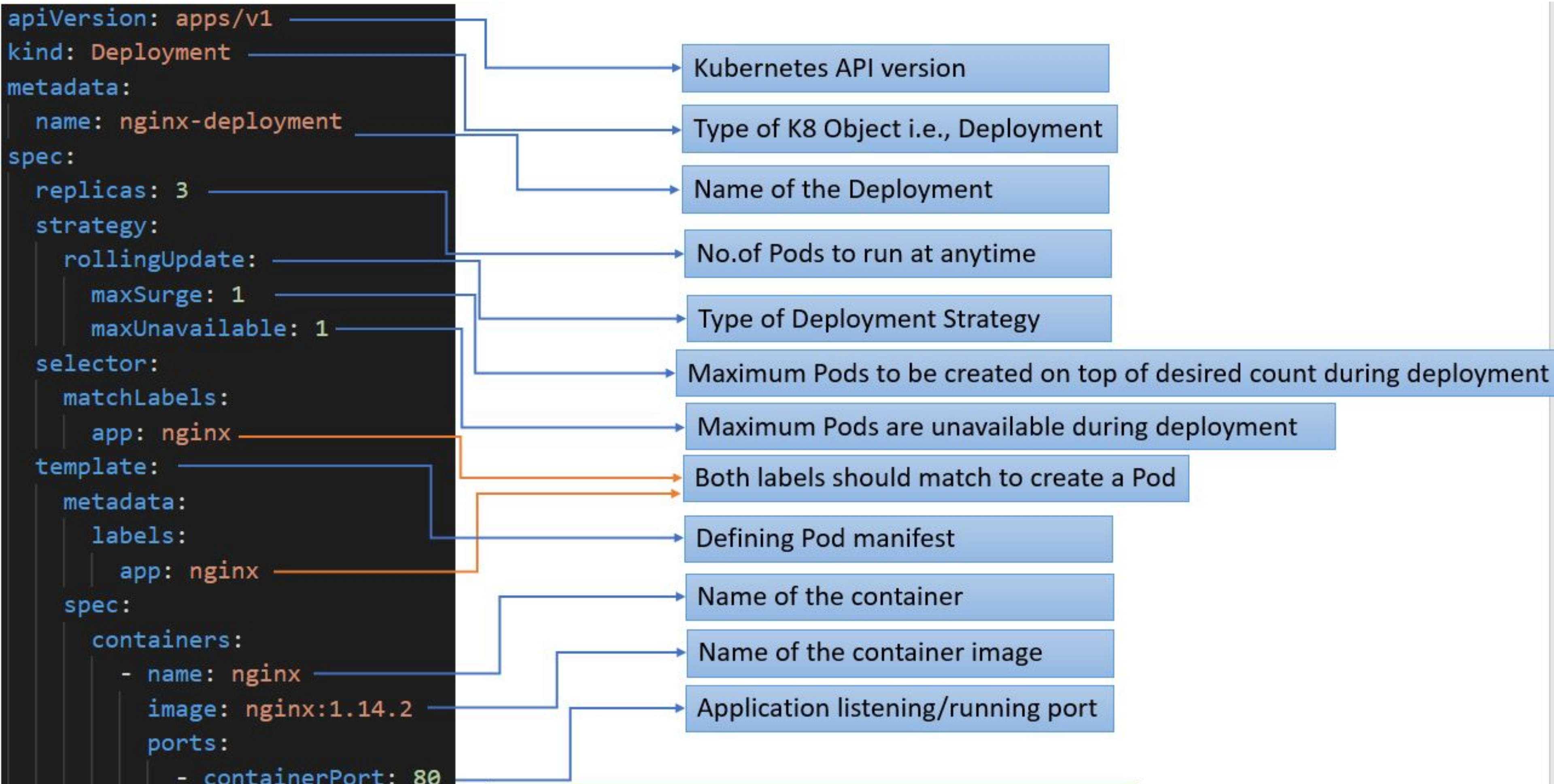

Kubernetes
Deployment
ReplicaSet
ReplicaSet
Pod
Pod
Pod
Pod

```
controlplane ~ ➜  kubectl create deployment mynginx --image=nginx
deployment.apps/mynginx created

controlplane ~ ➜  kubectl create deployment myapache --image=httpd
deployment.apps/myapache created

controlplane ~ ➜  kubectl create deployment mycaddy --image=caddy
deployment.apps/mycaddy created

controlplane ~ ➜  kubectl create deployment mytomcat --image=tomcat
deployment.apps/mytomcat created
```

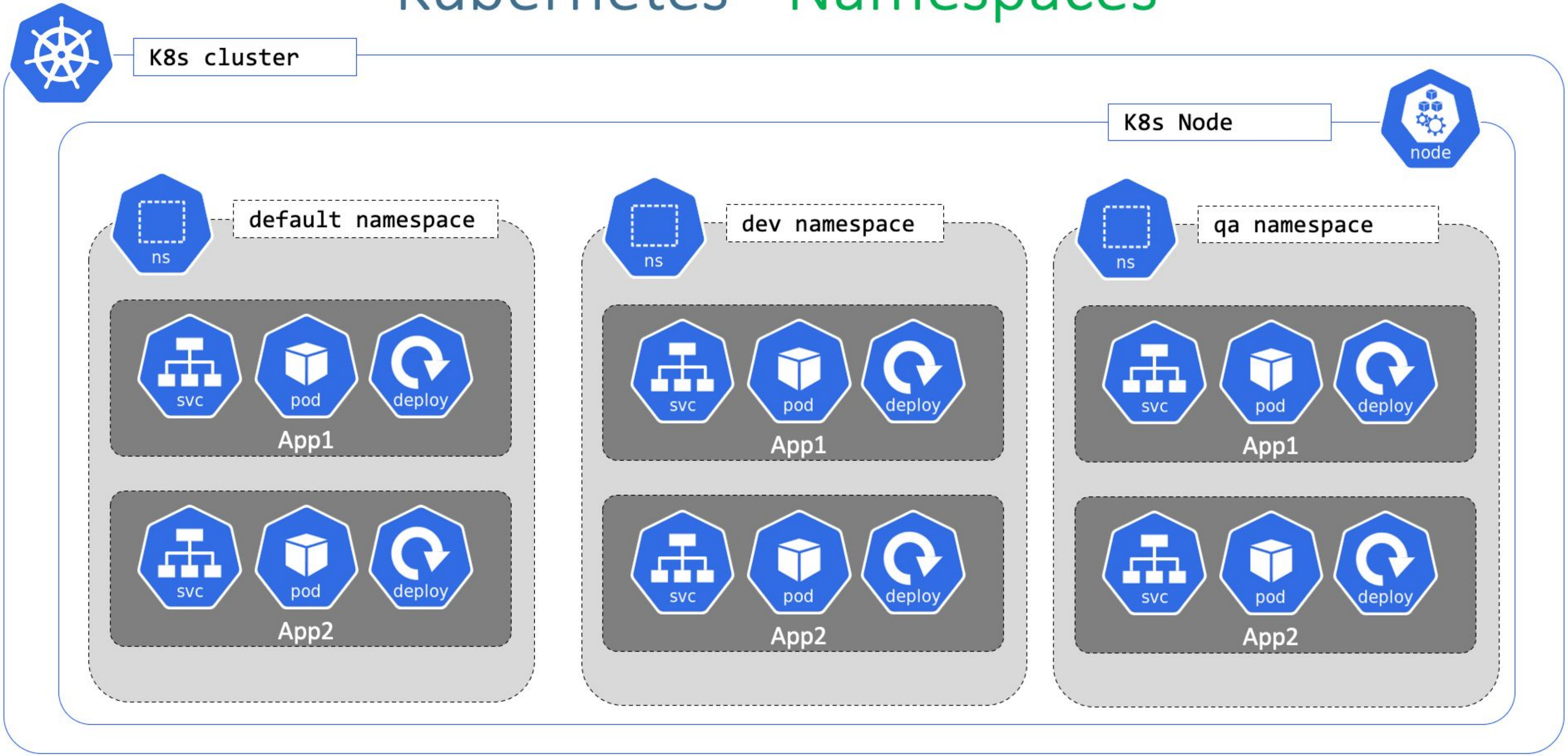


Deployment manifest: nginx-deployment.yaml

Namespace

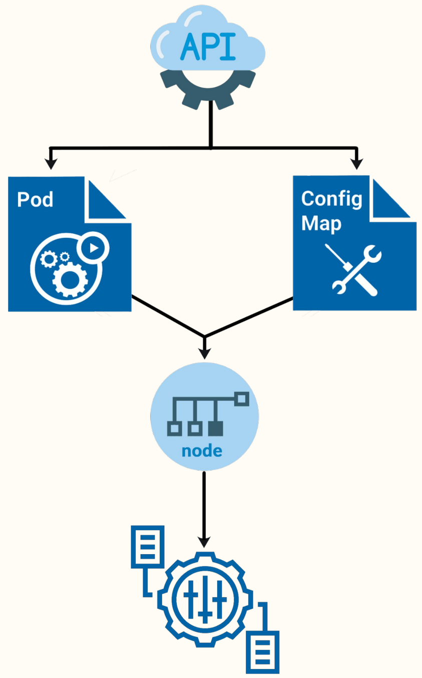
- Virtual clusters within a Kubernetes cluster
- Useful for isolating environments (dev, staging, prod) or teams
- Resources are scoped within a namespace unless explicitly stated as cluster-wide

Kubernetes - Namespaces



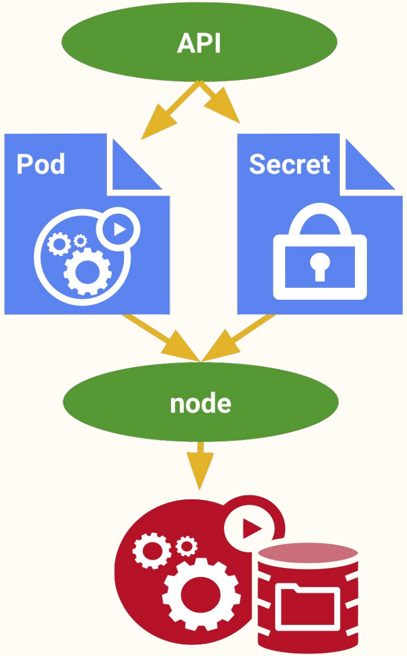
ConfigMap

- Provides configuration data as key-value pairs
- Used to configure application settings without rebuilding images
- Can be mounted as files or injected as environment variables



Secret

- Stores sensitive data like passwords, tokens, and keys
- Base64-encoded and can be encrypted at rest
- Used the same way as ConfigMaps (env vars or mounted as files)



```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  database_host: "192.168.0.1"
  debug_mode: "1"
  log_level: "verbose"
```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4=  # Base64-encoded value
  password: MWYyZDFlMmU2N2Rm  # Base64-encoded value
```

Let’s get hands-on

apply

attach

logs

create

describe

get

exec

run

delete

Kubernetes Networking

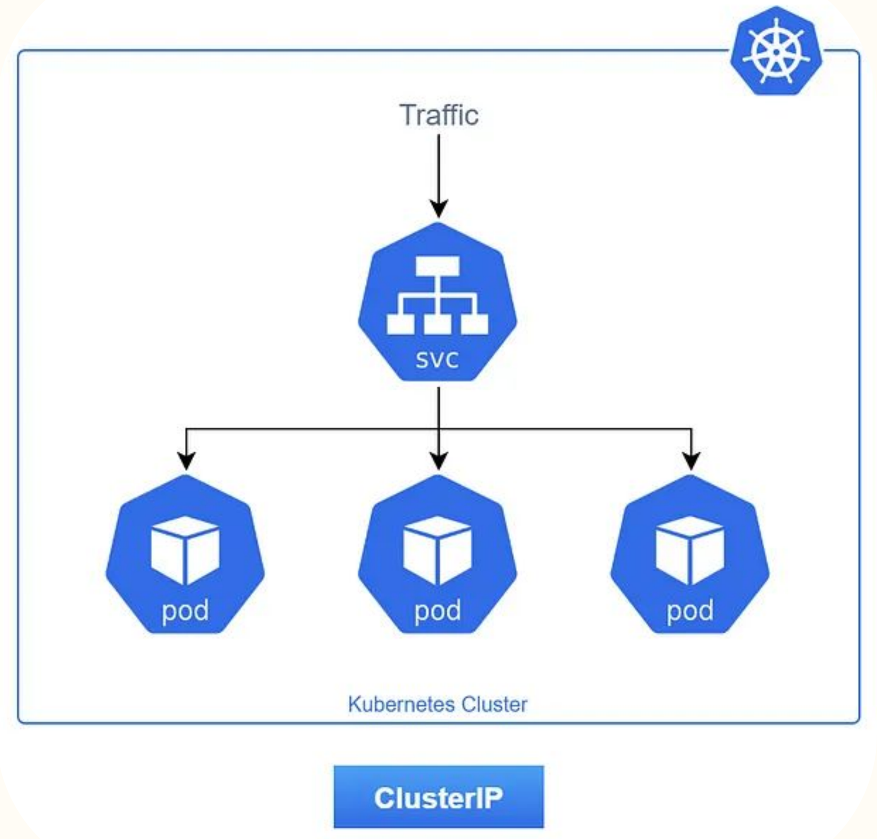
Service

An abstract way to expose an application running on a set of Pods as a network service:

- Pods are non-permanent resources
- Each Pod gets its own IP address
- Single DNS name for a set of Pods
- load-balance across them

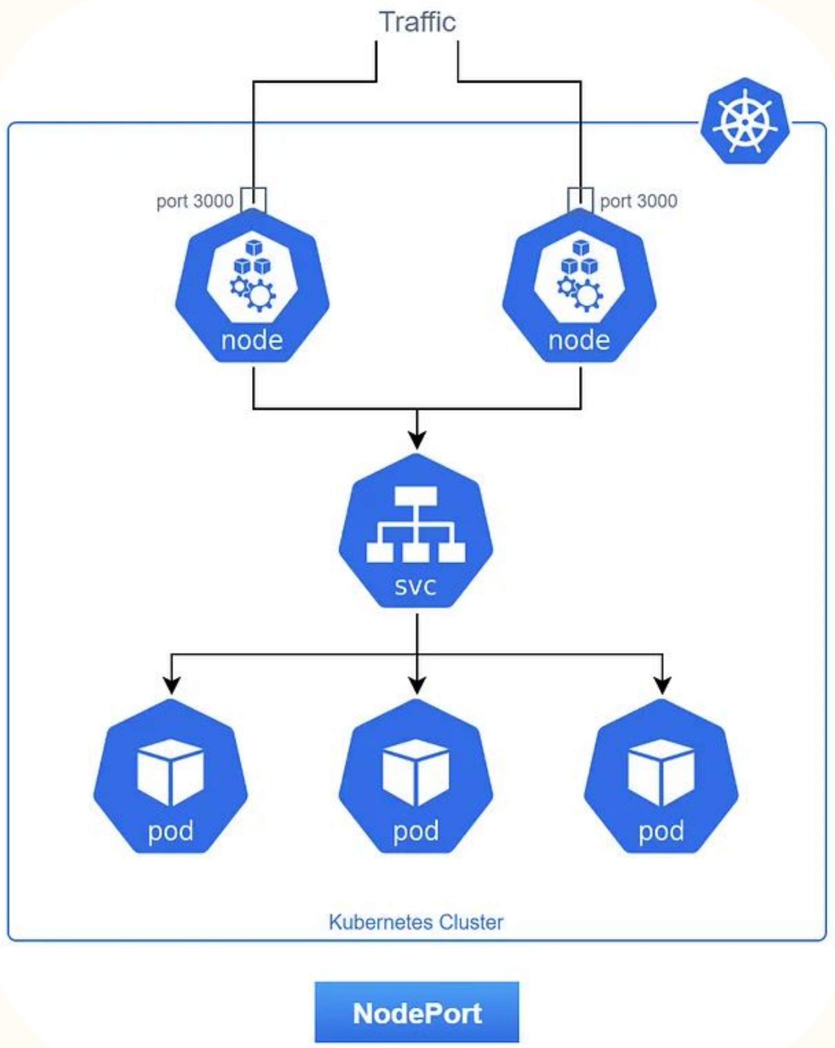
ClusterIP (default)

- Purpose: Exposes the service on an internal IP in the cluster
- Accessibility: Only accessible from within the cluster
- Use Case: Ideal for internal services like backend APIs or databases
- Default Behavior: If no type is specified, ClusterIP is used



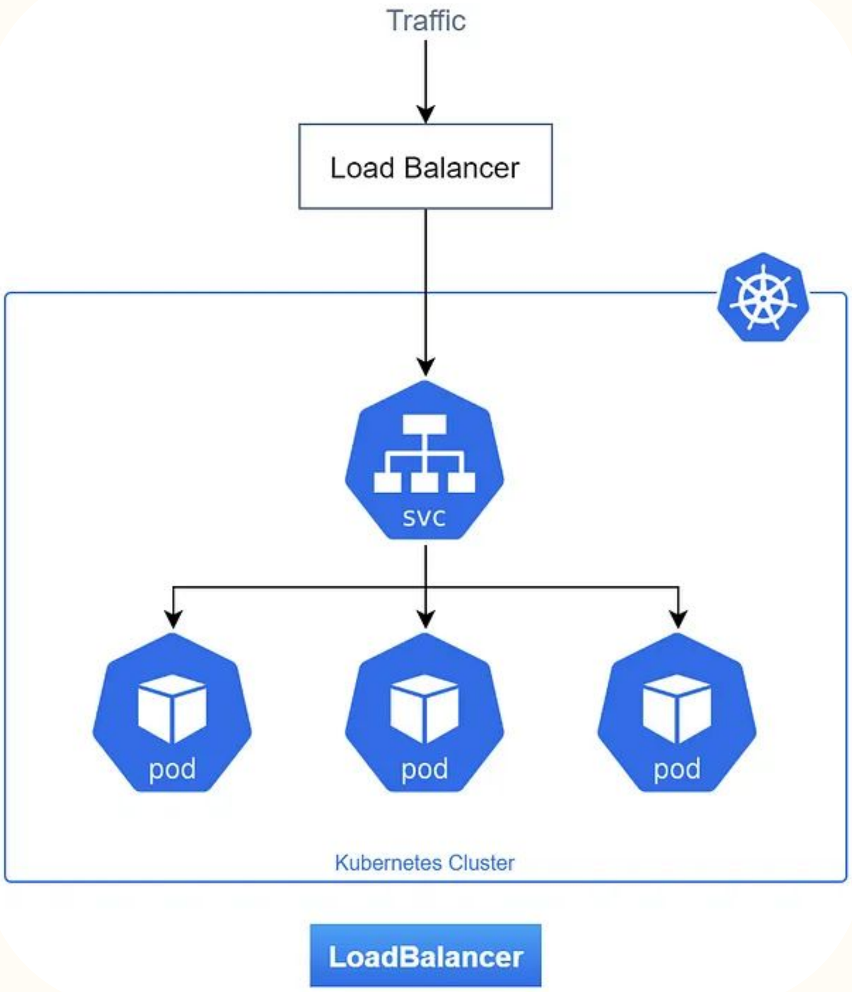
NodePort

- Purpose: Exposes the service on a static port (30000–32767) on each node’s IP
- Accessibility: Accessible externally via <NodeIP>:<NodePort>
- Use Case: Quick and simple way to expose a service externally without a load balancer
- Routing: Traffic is forwarded to the ClusterIP service



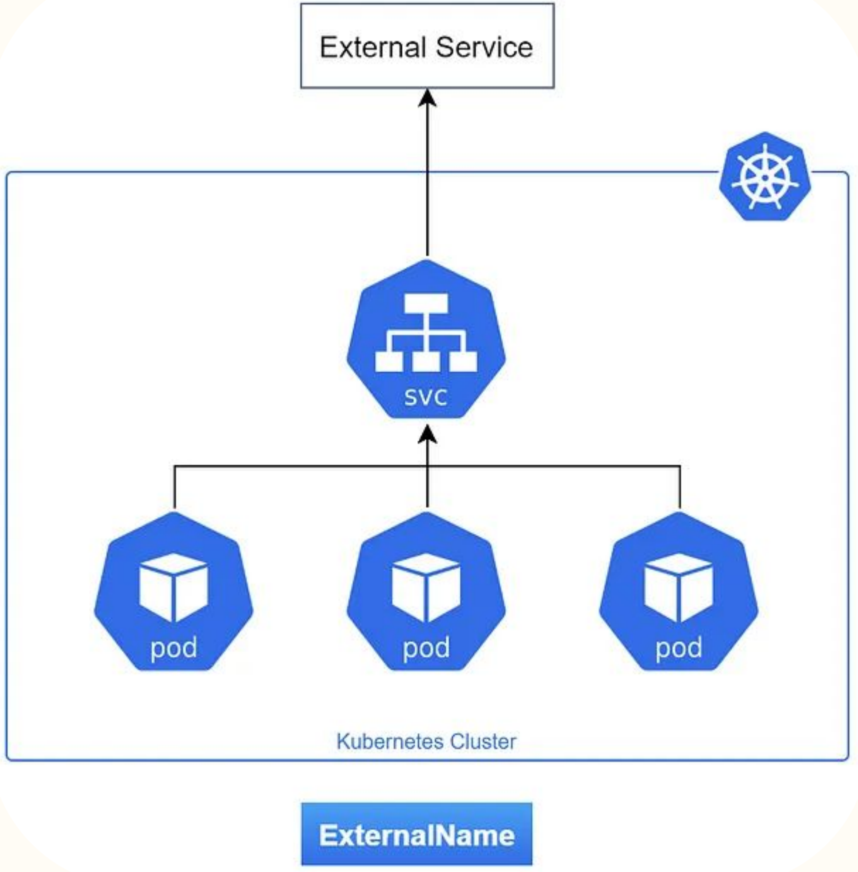
LoadBalancer

- Purpose: Provisions an external load balancer (from cloud provider) to expose the service
- Accessibility: Externally accessible via the load balancer's IP
- Use Case: Production-grade access to services in cloud environments
- Requires: Cloud provider integration (e.g., MetalLB, LoxiLB)



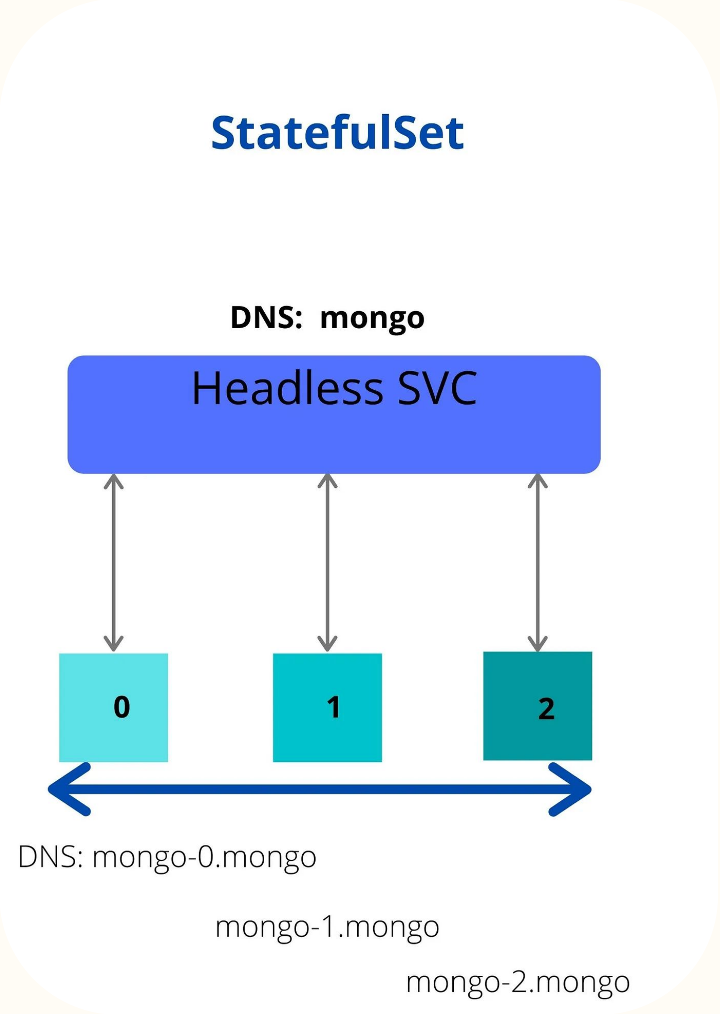
ExternalName

- Purpose: Maps the service to a DNS name (external to the cluster)
- Functionality: Returns a CNAME record redirecting to an external service
- Use Case: Accessing external databases or services via a Kubernetes service name
- No selector or endpoints: Doesn’t route to a pod

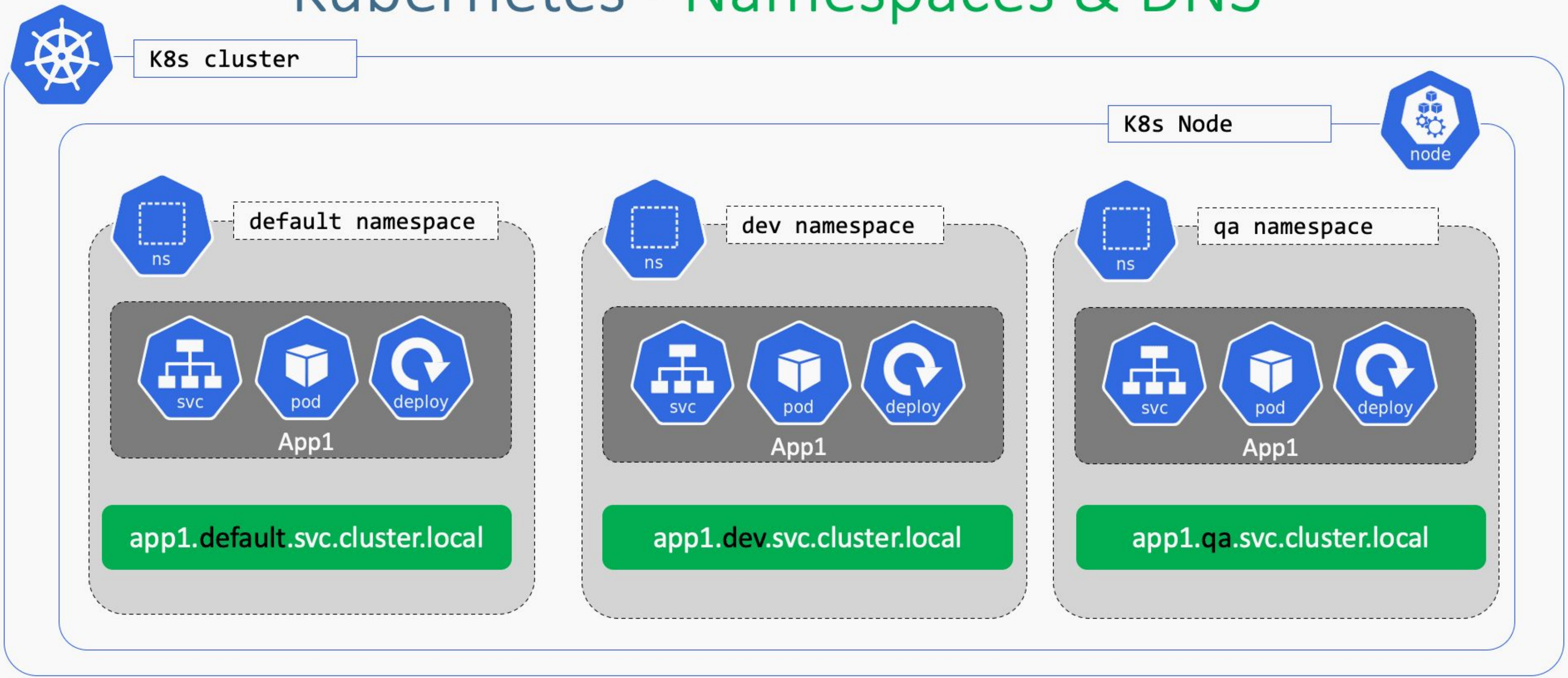


Headless

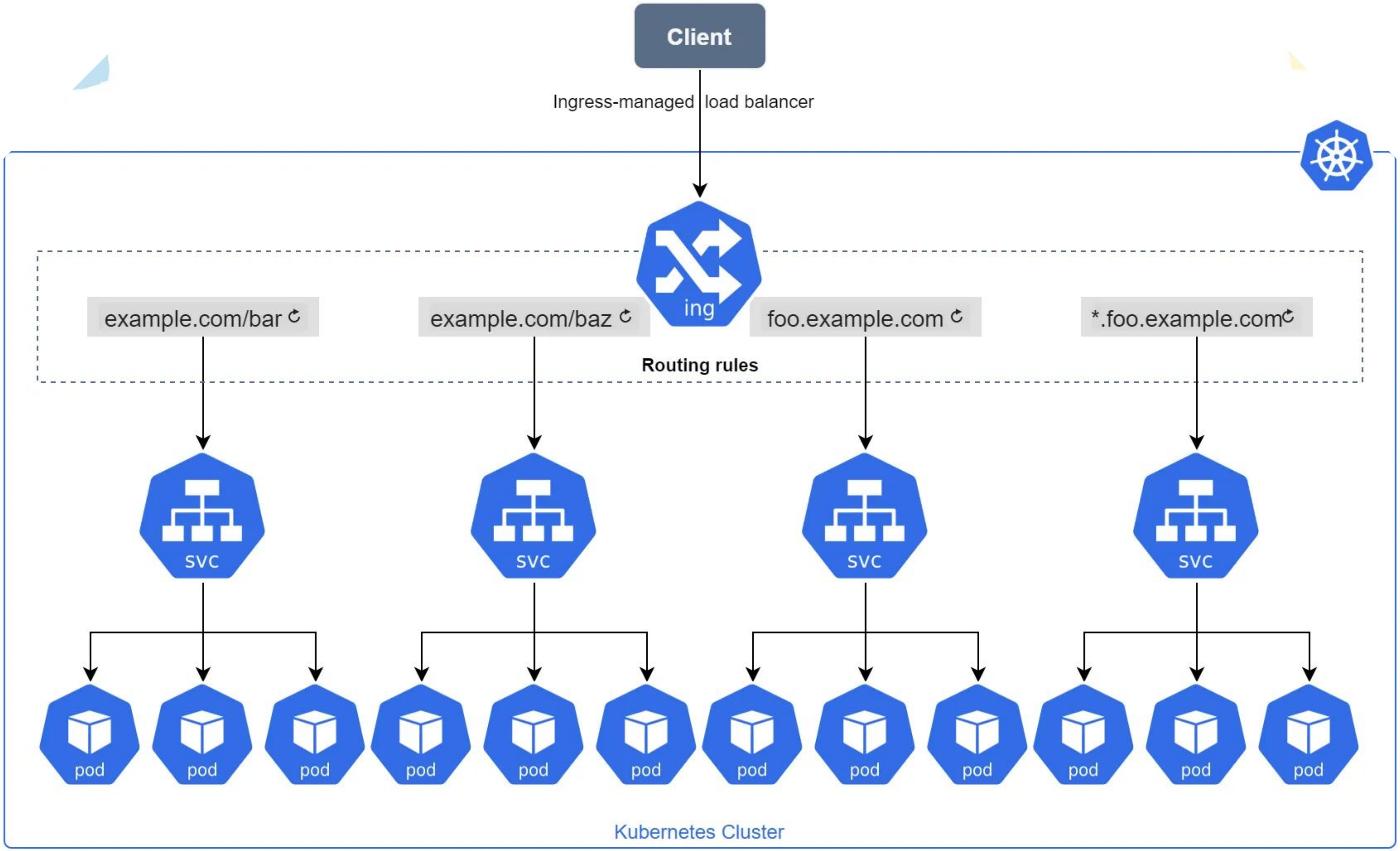
- Set via: clusterIP: None
- Purpose: Does not assign a cluster IP; DNS resolves to pod IPs directly
- Use Case: Stateful applications (e.g., databases) that need direct pod-to-pod communication
- Functionality: Works well with StatefulSets and DNS-based service discovery



Kubernetes - Namespaces & DNS



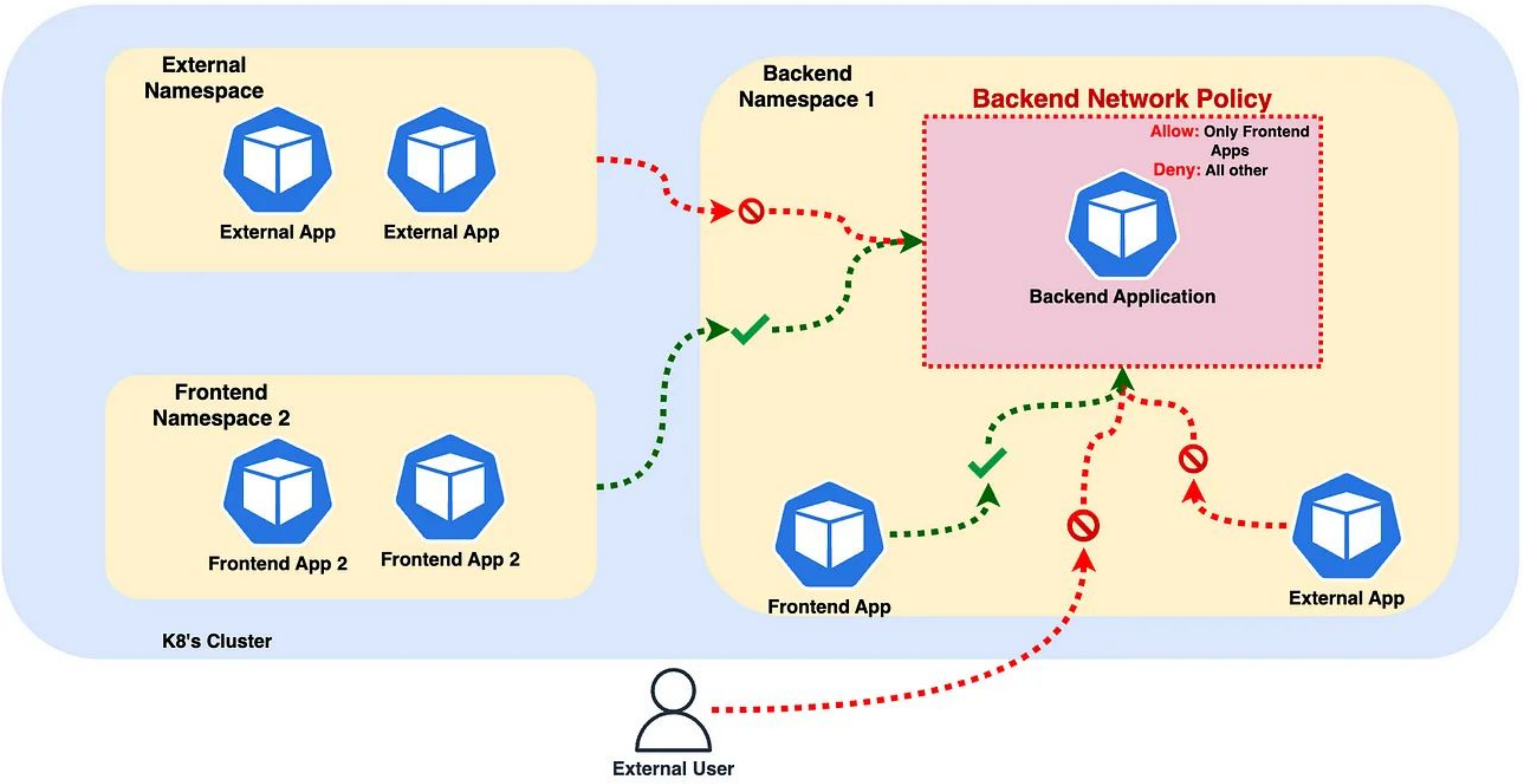
<service-name>.<namespace-name>.svc.cluster.local


Client
Ingress-managed load balancer
ing
example.com/bar
example.com/baz
foo.example.com
*.foo.example.com
Routing rules
svc
svc
svc
svc
pod
pod
pod
pod
pod
pod
pod
pod
pod
pod
pod
pod
Kubernetes Cluster

Ingress

- Manages HTTP(S) access to services from outside the Kubernetes cluster
- Acts like a Layer 7 (HTTP) load balancer
- Host-based routing: Route foo.example.com to one service, bar.example.com to another
- Path-based routing: Route /api to one service, /web to another
- TLS termination: Handle HTTPS using TLS certificates (often from Let’s Encrypt)
- Rewrite/redirect rules (controller specific)

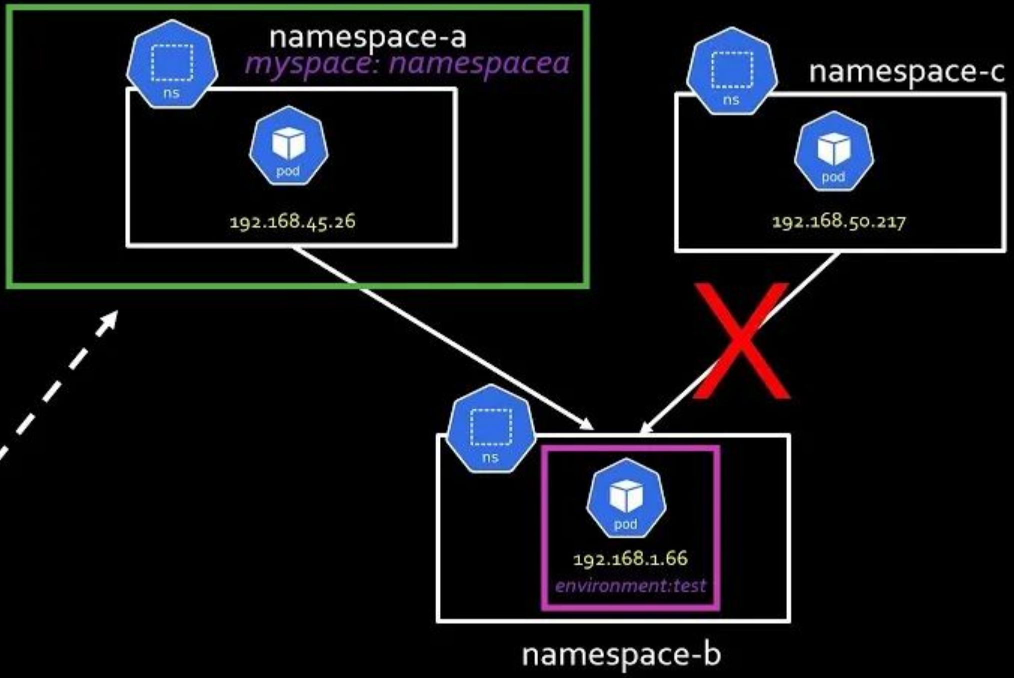
```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
    - host: my-app.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: myapp-service
                port:
                  number: 80
```


External Namespace
External App
External App
Backend Namespace 1
Backend Network Policy
Allow: Only Frontend Apps
Deny: All other
Backend Application
Frontend Namespace 2
Frontend App 2
Frontend App 2
Frontend App
External App
K8's Cluster
External User

NetworkPolicy

- Purpose: Controls network traffic (ingress and egress) to/from pods
- Default Behavior: Pods accept all traffic unless restricted by a NetworkPolicy
- Scope: Applies at the pod level, based on labels
- Enforcement: Requires a CNI (like Calico, Cilium, etc.) that supports NetworkPolicies

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: namespace-b
spec:
  podSelector:
    matchLabels:
      environment: test
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          myspace: namespacea
```



Advanced Kubernetes

StatefulSets

What are stateful applications?

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, automated rolling updates.

Which is a stateful application?

- aut.ac.ir front end app?
- MySQL?
- Websocket calculator?

StatefulSets cont.

- Stateful set in kubernetes provide non-interchangeable identities to pods.
  - *<sts-name>-<pod-index>*
- Persistent pod identities
  - Permits per pod volume attachment (v1 always attaches to p1)
  - Allows pod discovery with help of headless services
    - *<sts-name>-<pod-index>.<headless-service>.<namespace>.svc.cluster.local*
- Special graceful scaling operations (order of operations within is always the same)
  - Shutdown
  - Rolling update

Exercise

https://labs.iximiuz.com/challenges/start-pod-with-limited-resources

Other workload types

DaemonSets: A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them.

Example: DNS server, Log collection, etc

Jobs: obs represent one-off tasks that run to completion and then stop.

Example: Performing a custom backup job.

CronJobs: A CronJob starts one-time Jobs on a repeating schedule.

Example: Analytical reports, Periodic backups, etc.

Scheduling Node Selection

We can:

- **Restrict** pods to certain nodes → *.spec.nodeName, .spec.nodeSelector*
- **Prefer** pods on certain nodes. → Node Affinity Rules

Node/Pod affinity/anti affinity rules are **more expressive** → Cover more scheduling rules
based on **node labels** and **pods running on a node**.

There are two types:

- `requiredDuringSchedulingIgnoredDuringExecution` → **Restrict**
- `preferredDuringSchedulingIgnoredDuringExecution` → **Prefer**
    - Can use weighting to specify importance of each condition

Scheduling Node Selection cont.

Some Examples:

- https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes/
- https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity

Resource Management

There are two resource types: **CPU** and **Memory**

Resources management is done by setting **Limit** and **Request** constraints.

Request: Considered during scheduling by `kube-scheduler`

- `kubelet` reserves **at least** this amount on node.

Limit: Enforced by `kubelet` on containers.

- Memory enforced by OOM kill signal
- CPU enforced by throttling
- Possible sum(limits.[cpu/memory]) > host CPU/Memory

Resource Management cont.

CPU

- Measured in quantities of 1 physical or logical core
  - 1 → 1 core, 0.05 → 50 mili cores
  - 0.1 equals to same amount regardless of host CPU cores
- Container runtime configures cgroups
  - During each (cpu) scheduling interval, kernel checks if container has reached its limit.
  - In contention, container with higher **request** is allocated CPU time.

Memory

- Measured in bytes
  - Power of 2 → Ki, Mi, Gi
  - Power of 10 → K, M, G
- Container runtime configures cgroup
  - If a process reaches the cgroup limit, the kernel will handle the termination.
  - Memory backed emptyDir volume contributes to memory usage
  - If container reaches it's memory request and the host is out of memory → pod will be **evicted**

Exercise ?

https://kubernetes.io/docs/tutorials/stateful-application/basic-stateful-set/

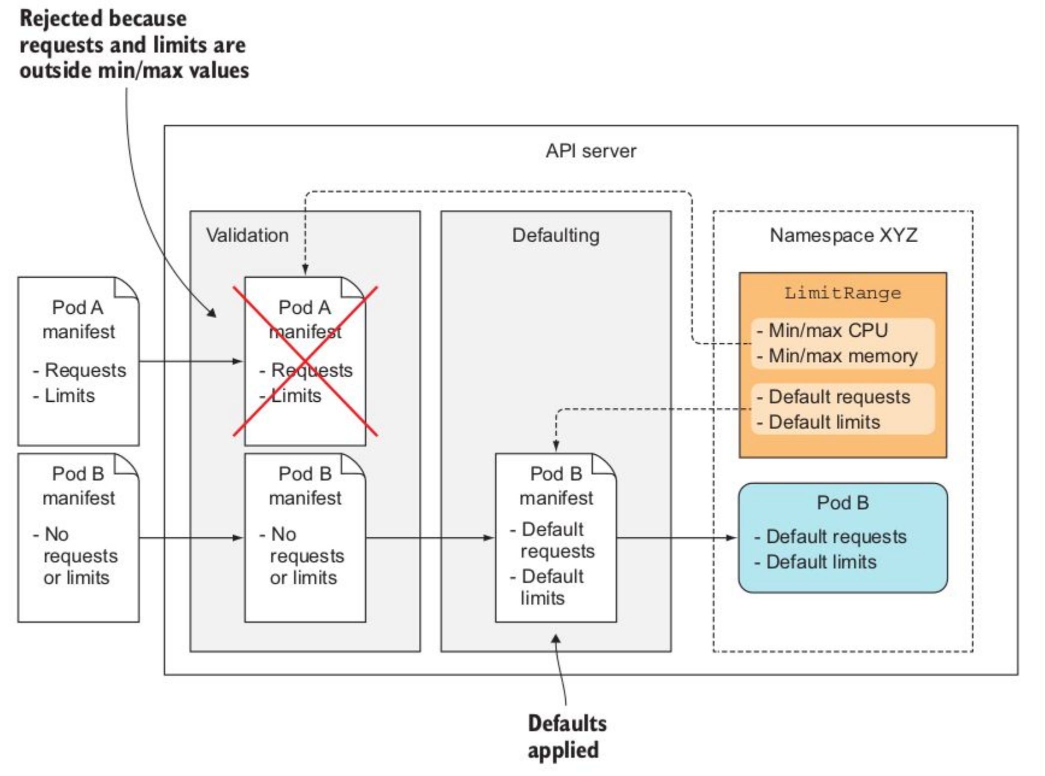
Resource Sharing

When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources. The solution for these scenarios is **`ResourceQuota`**

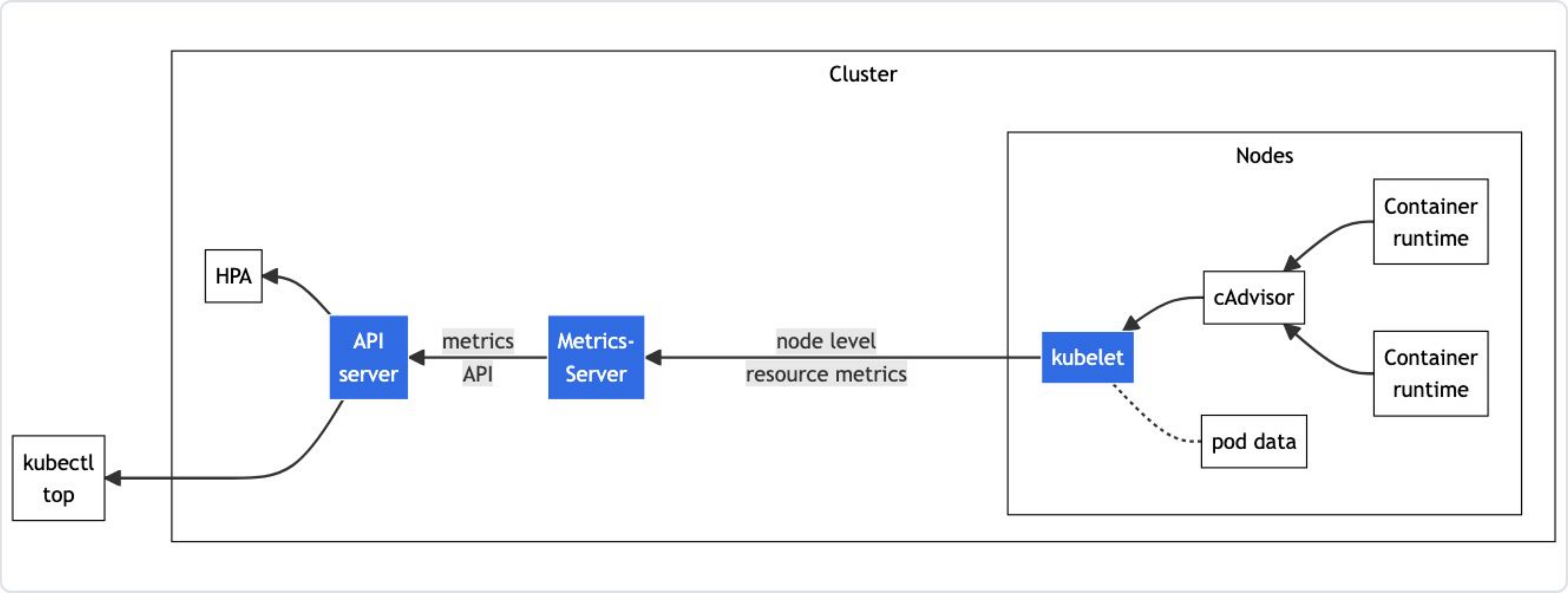
We can constraint resources **per namespace** with ResourceQuota.

We can set default values and limitation on resource values using **`LimitRanges`**

Resource Sharing cont.



Resource Monitoring



Autoscaling

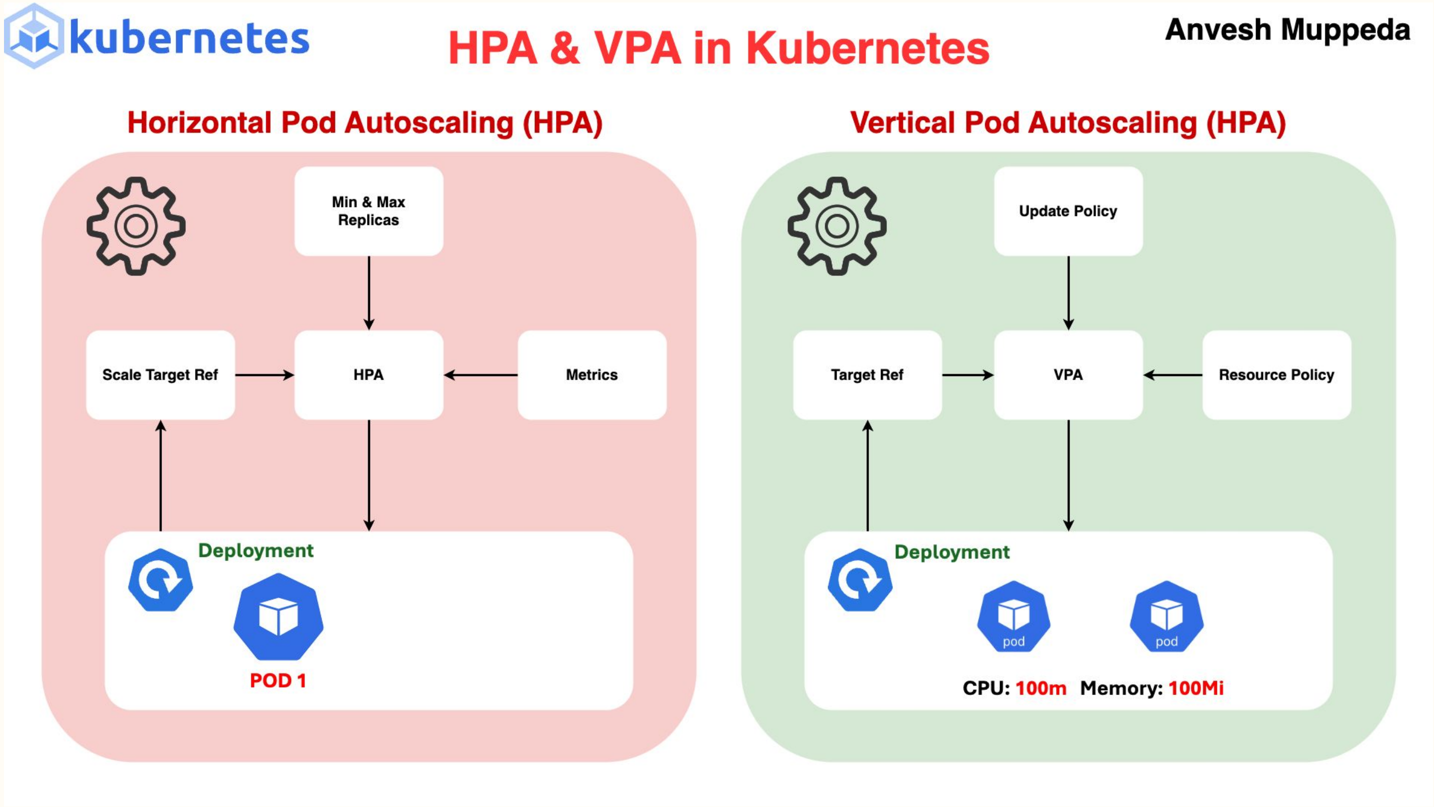
With autoscaling, you can automatically update your workloads in one way or another. This allows your cluster to react to changes in resource demand more elastically and efficiently.

Autoscaling is either **Horizontal** or **Vertical**

Vertical scaling: Resizing CPU and memory resources assigned to containers

Horizontal scaling: Running multiple instances of your app

Autoscaling



Demo

https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/

https://github.com/collabnix/kubelabs/blob/master/Autoscaler101/autoscaler-lab.md