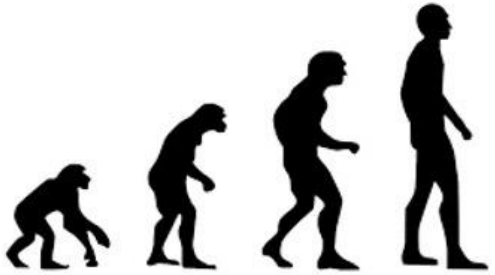
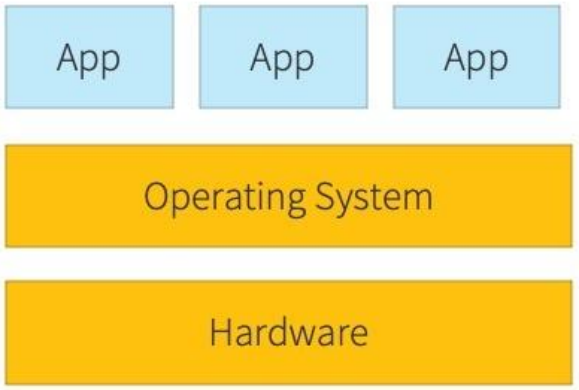
04 - Kubernetes

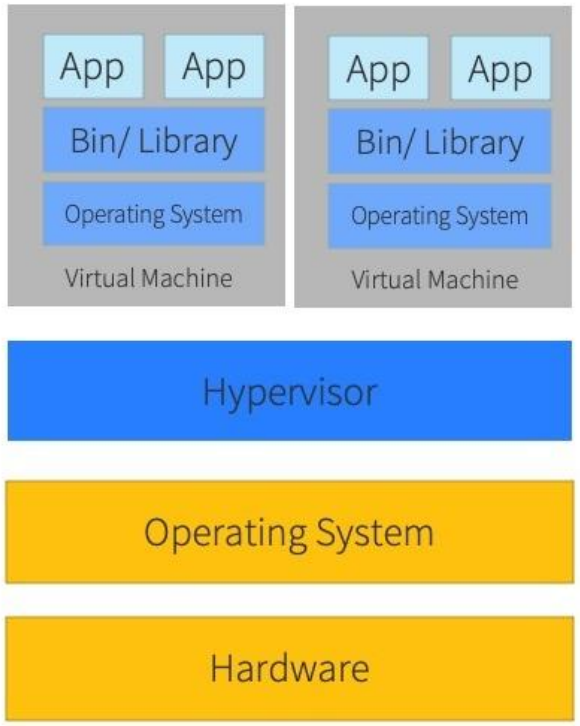
Department of Computing, (SETU)

App Deployment Roadmap

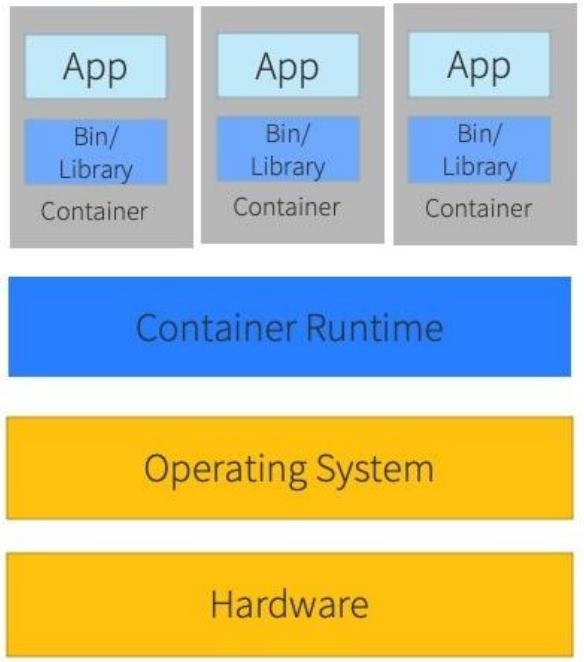




Traditional Deployment



Virtualized Deployment



Container Deployment

Introduction to Kubernetes <K8s>

K8s manages containers

K8s is an open-source platform for container management developed by Google and introduced in 2014. It has become the standard API for building cloud-native applications, present in nearly every public cloud.

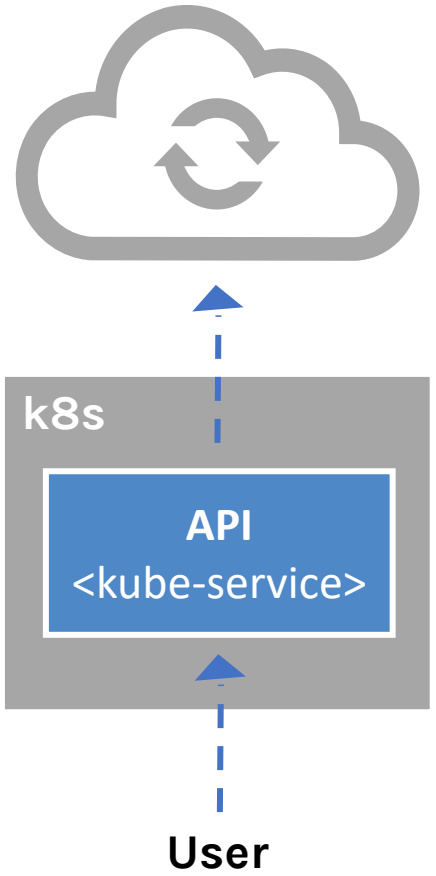
K8s users define rules for how container management should occur, and then K8s handles the rest!

> link to website <

Introduction to Kubernetes <cont>

There are many reasons why people come to use containers and container APIs like Kubernetes:

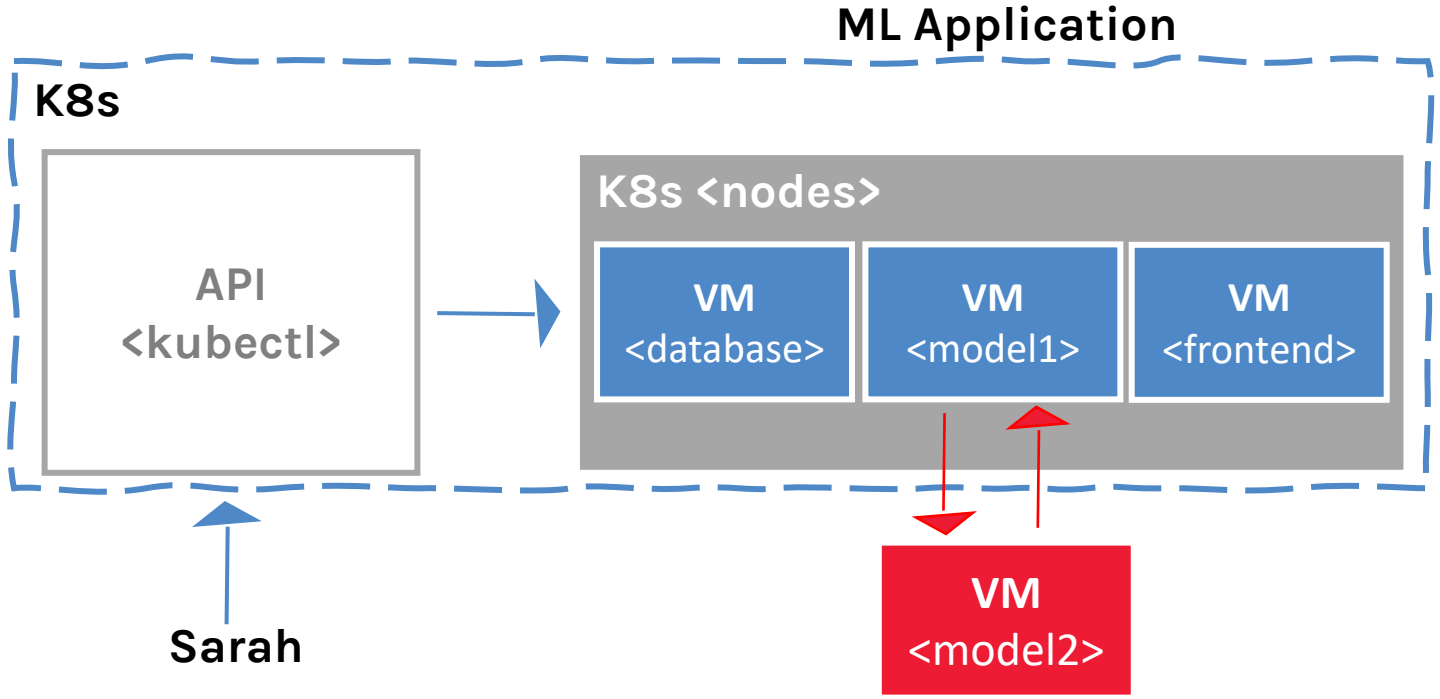
- Velocity
- Scaling (of both software and teams)
- Abstracting the infrastructure
- Efficiency



Velocity

It is the speed with which you can respond to innovations developed by others (e.g. change in software industry from shipping CDs to delivering over the network)

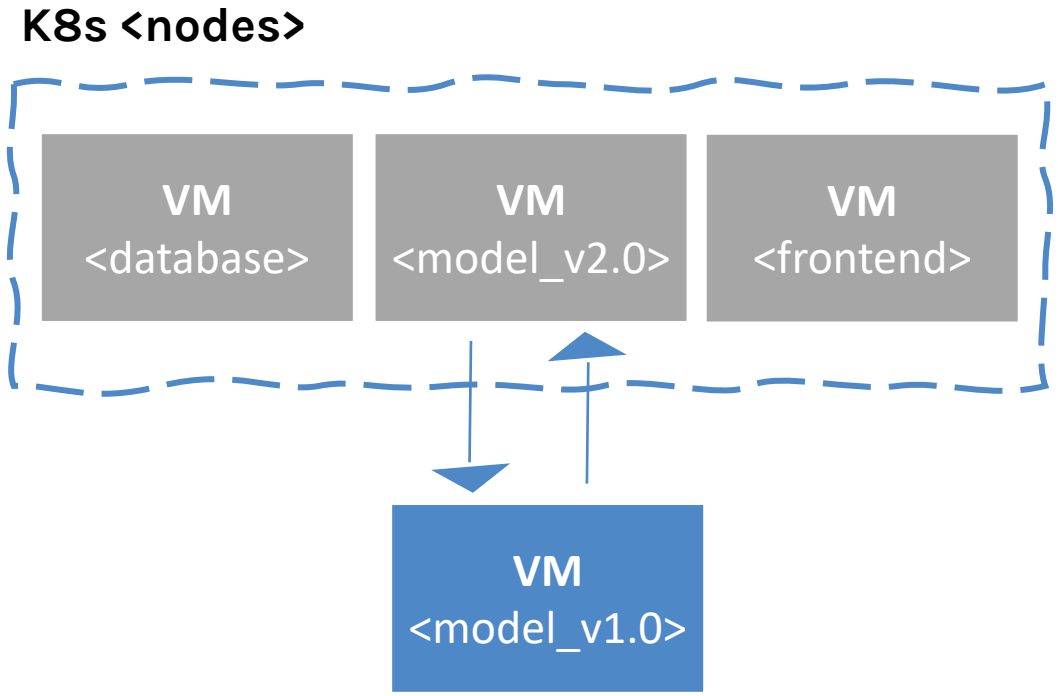
Velocity is measured not in terms of the number of things you can ship while maintaining a highly available service



Velocity <cont>

Velocity is enabled by:

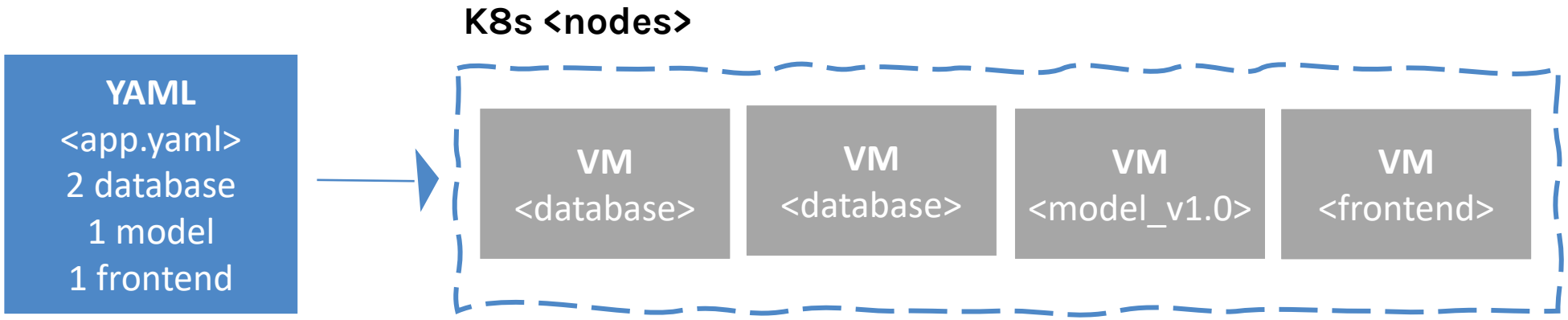
- **Immutable system:** you can't change running container, but you create a new one and replace it in case of failure (allows for keeping track of the history and load older images)



Velocity <cont>

Velocity is enabled by:

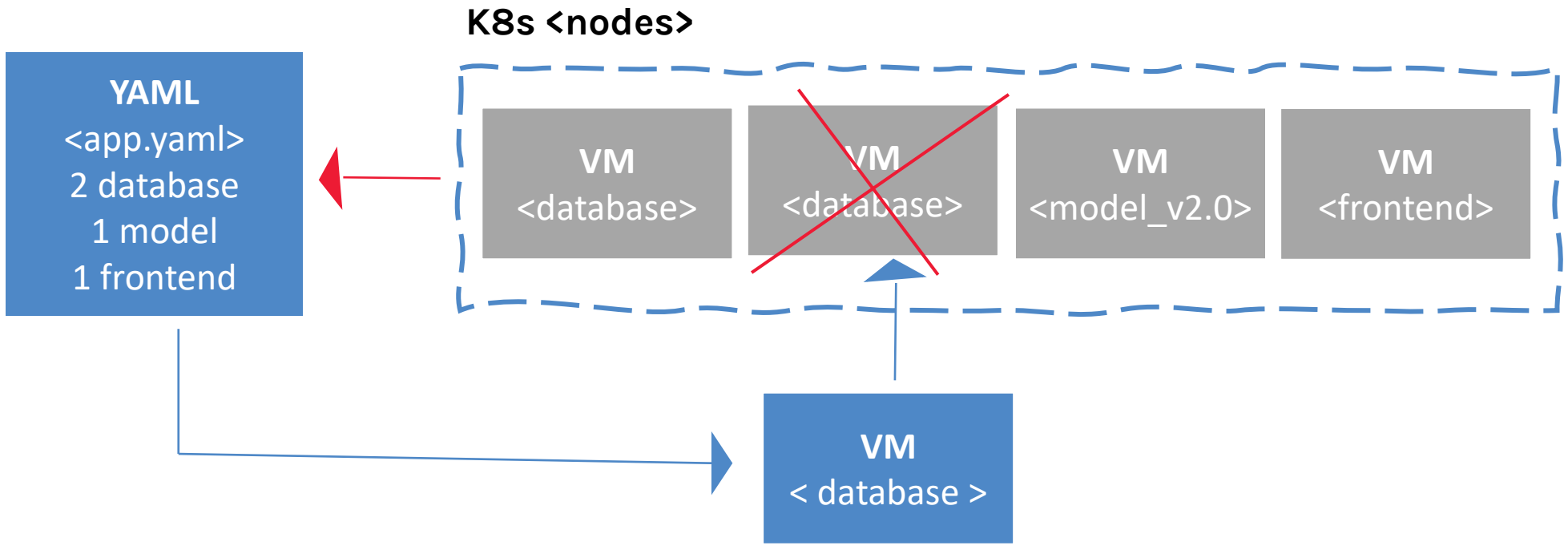
- **Declarative configuration:** you can define the desired state of the system restating the previous declarative state to go back. ***Imperative*** configuration are defined by the execution of a series of instructions, but not the other way around.



Velocity <cont>

Velocity is enabled by:

- **Online self-healing systems:** k8s takes actions to ensure that the current state matches the desired state (as opposed to an operator enacting the repair)



Velocity <recap>

Velocity is enabled by:

- Immutable system
- Declarative configuration
- Online self-healing systems

All these aspects relate to each other to speed up process that can reliably deploy software.

Scaling

As your product grows, it’s inevitable that you will need to scale:

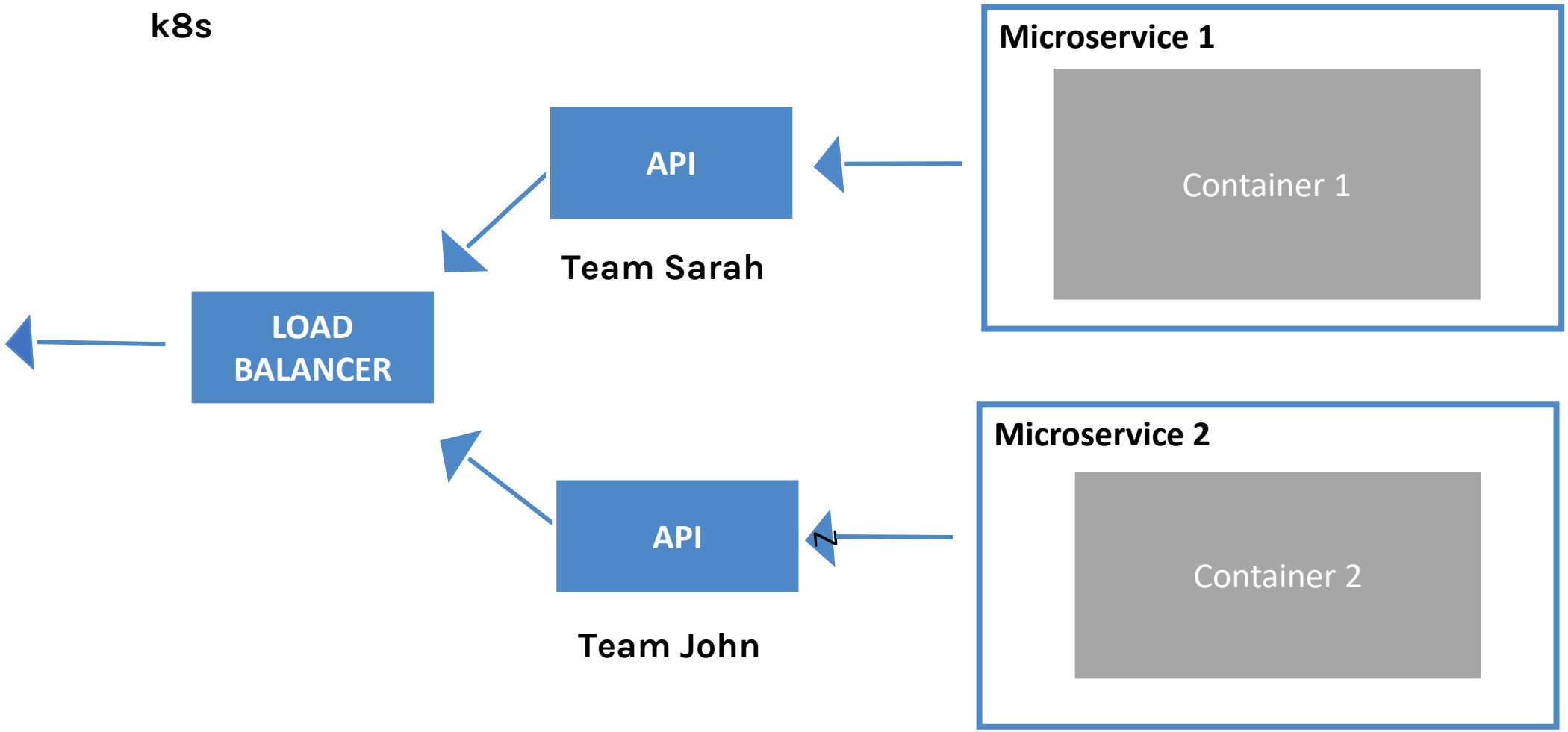
- Software
- Team/s that develop it

Scaling

Kubernetes provides numerous advantages to address scaling:

- **Decoupled architectures**: each component is separated from other components by defined APIs and *service load balancers*.
- **Easy scaling for applications and clusters**: simply changing a number in a configuration file, k8s takes care of the rest (part of declarative).
- **Scaling development teams with microservices**: small team is responsible for the design and delivery of a service that is consumed by other small teams (optimal group size: 2 pizzas team).

Scaling <cont>



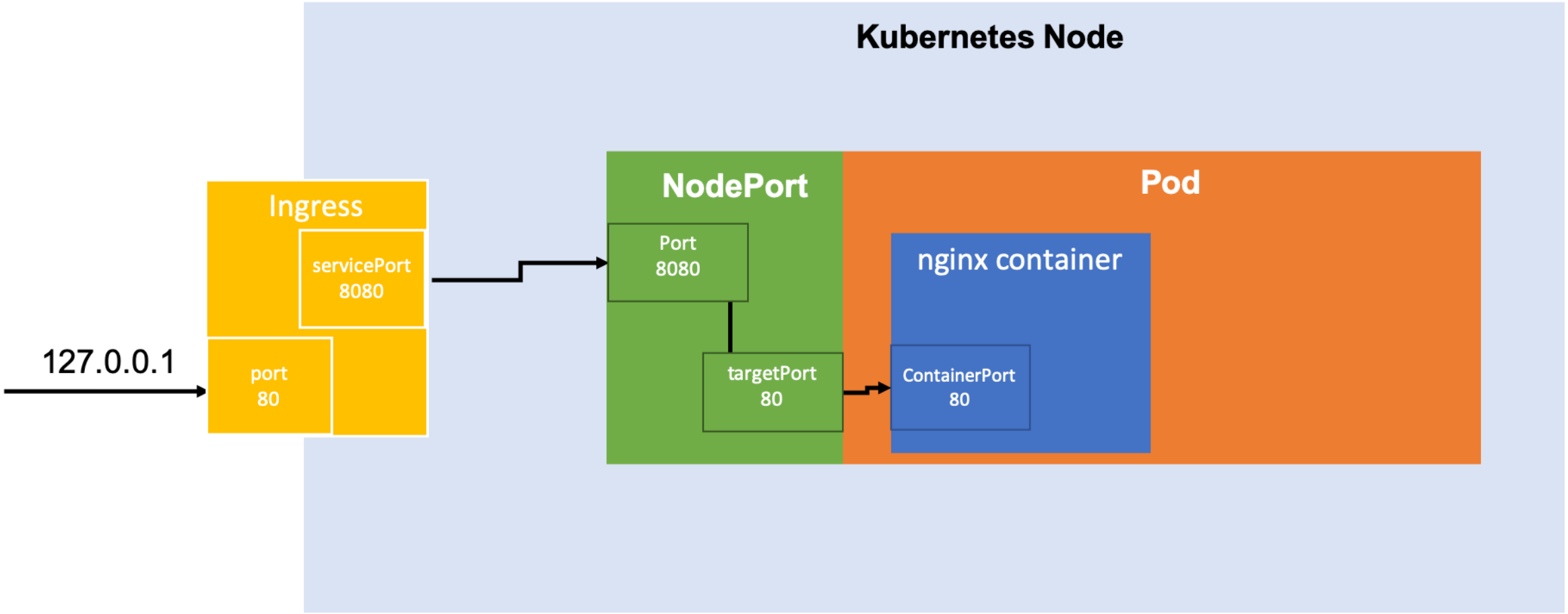
Scaling <cont>

Kubernetes provides numerous **abstractions** and APIs that help building these decoupled microservice architectures:

- **Pods** can group together container images developed by different teams into a single deployable unit (similar to docker-compose)
- **Other services to isolate** one microservice from another such (e.g. load balancing, naming, and discovery)
- **Namespaces** control the interaction among services
- **Ingress** combine multiple microservices into a single externalized API (easy-to-use frontend)

K8s provides full spectrum of solutions between doing it “the hard way” and a fully managed service

Scaling <cont>



Abstracting your infrastructure

Kubernetes allows to build, deploy, and manage your application in a way that is portable across a wide variety of environments. The move to application-oriented container APIs like Kubernetes has two concrete benefits:

- **separation**: developers from specific machines
- **portability**: simply a matter of sending the declarative config to a new cluster

Efficiency

There are concrete economic benefit to the abstraction because tasks from multiple users can be packed tightly onto fewer machines:

- **Consume less energy** (ratio of the useful to the total amount)
- **Limit costs of running a server** (power usage, cooling requirements, datacenter space, and raw compute power)
- **Create quickly a developer's test environment** as a set of containers
- **Reduce cost of development instances in your stack**, liberating resources to develop others that were cost-prohibitive

Monolithic Application

Recruitment Website

Job Applicants

Job Vacancies

Recruiters

Transition to Microservices

Recruitment Website

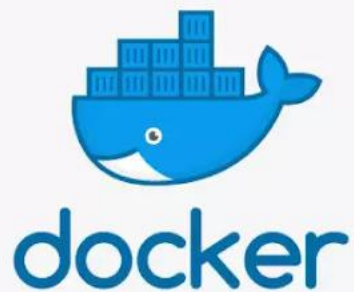
Recruiters

Job Applicants

Job Vacancies

Docker

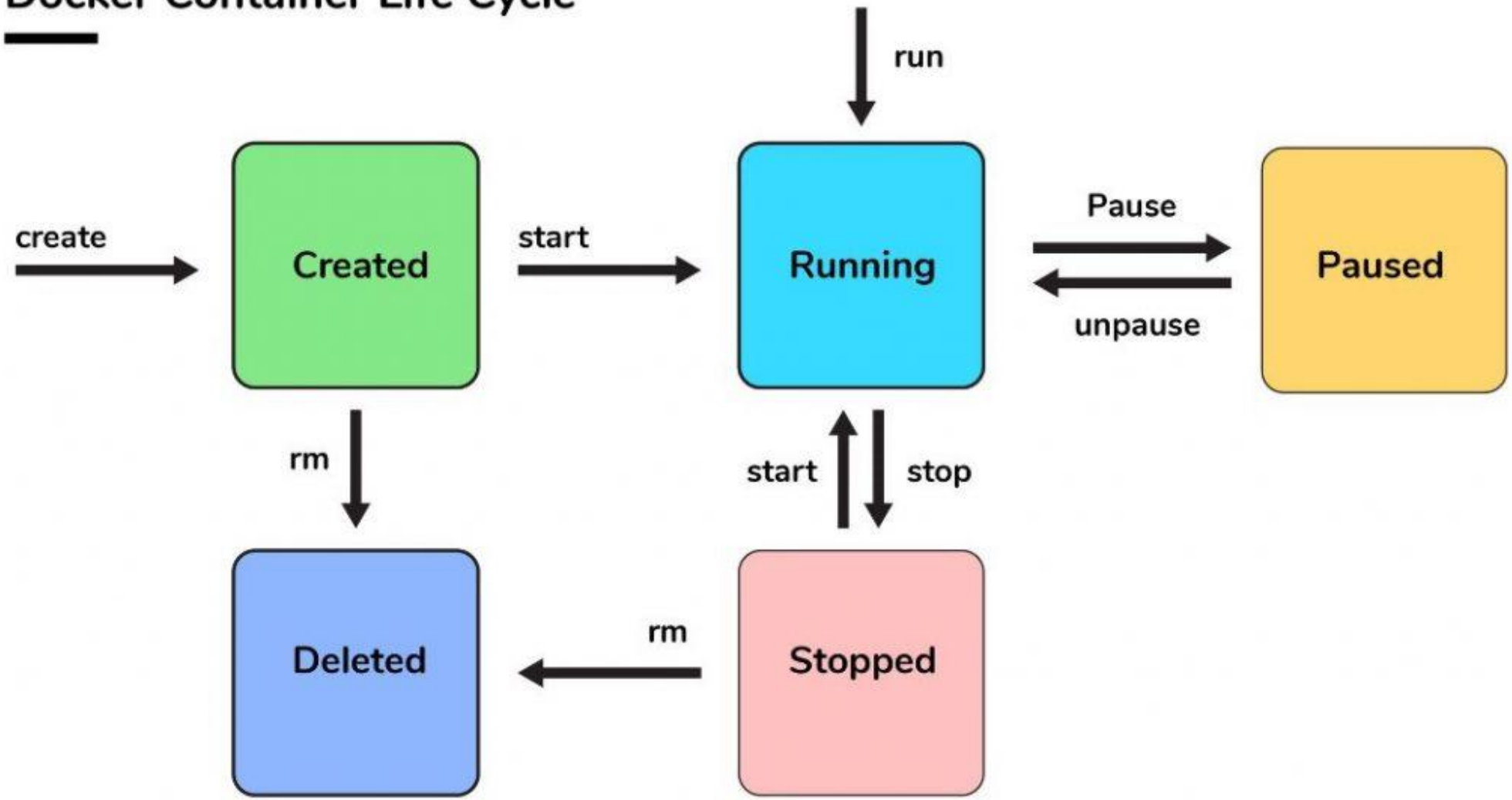
Create containers for your application



Kubernetes

Launch your containerised application in K8s

Docker Container Life Cycle



We need more than just packing and isolation

Scheduling: Where should my containers run?

Lifecycle and health: Keep my containers running despite failures

Discovery: Where are my containers now?

Monitoring: What's happening with my containers?

Auth{n,z}: Control who can do things to my containers

Aggregates: Compose sets of containers into jobs

Scaling: Making jobs bigger or smaller

...

Open Source Containers: Kubernetes

Greek for *"Helmsman"*; also the root of the word *"Governor"* and *"cybernetic"*

- Container orchestrator
- Builds on Docker containers
  - also supporting other container technologies
- Multiple cloud and bare-metal environments
- Supports existing OSS apps
  - cannot require apps becoming cloud-native
- Inspired and informed by Google's experiences and internal systems
- **100% Open source**, written in **Go**

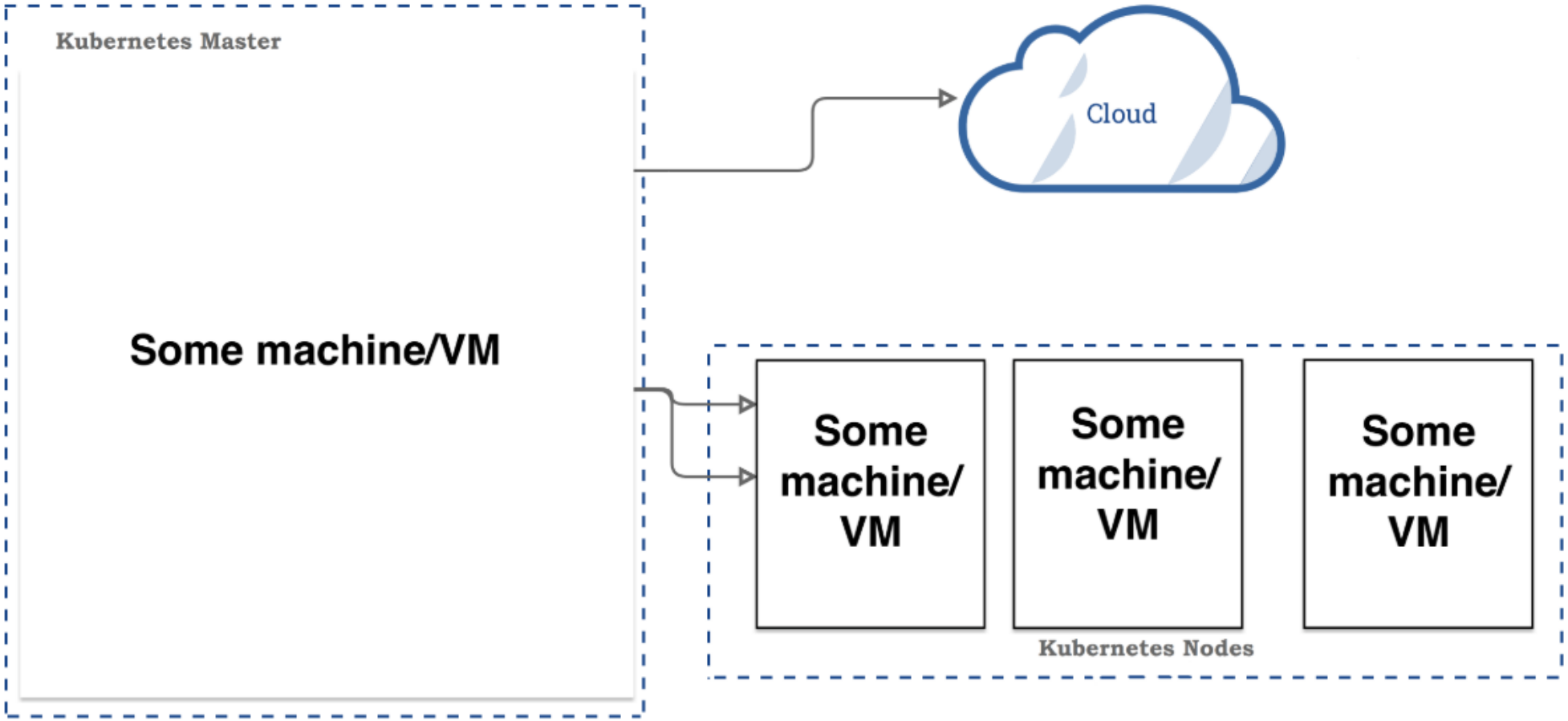
Let users manage **applications**, not machines

Architecture Overview

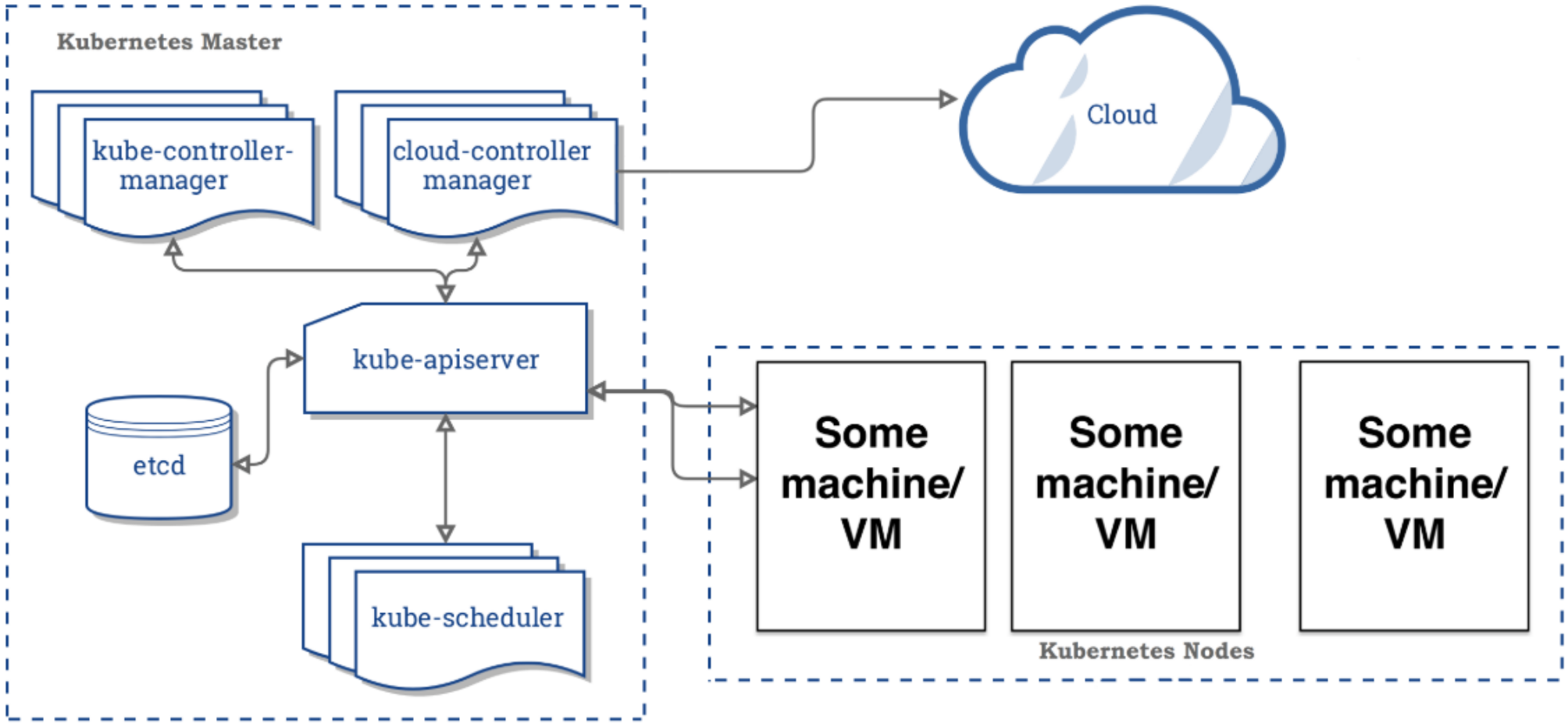
Anatomy of Kubernetes Cluster

- K8s works on a cluster of machines/nodes
- This could be VMs on your local machine or a group of machines through a cloud provider
- The cluster includes one master node and at least one worker node

Anatomy of Kubernetes Cluster <cont>



Anatomy of Kubernetes Cluster | Master Node



> to learn more on etcd <

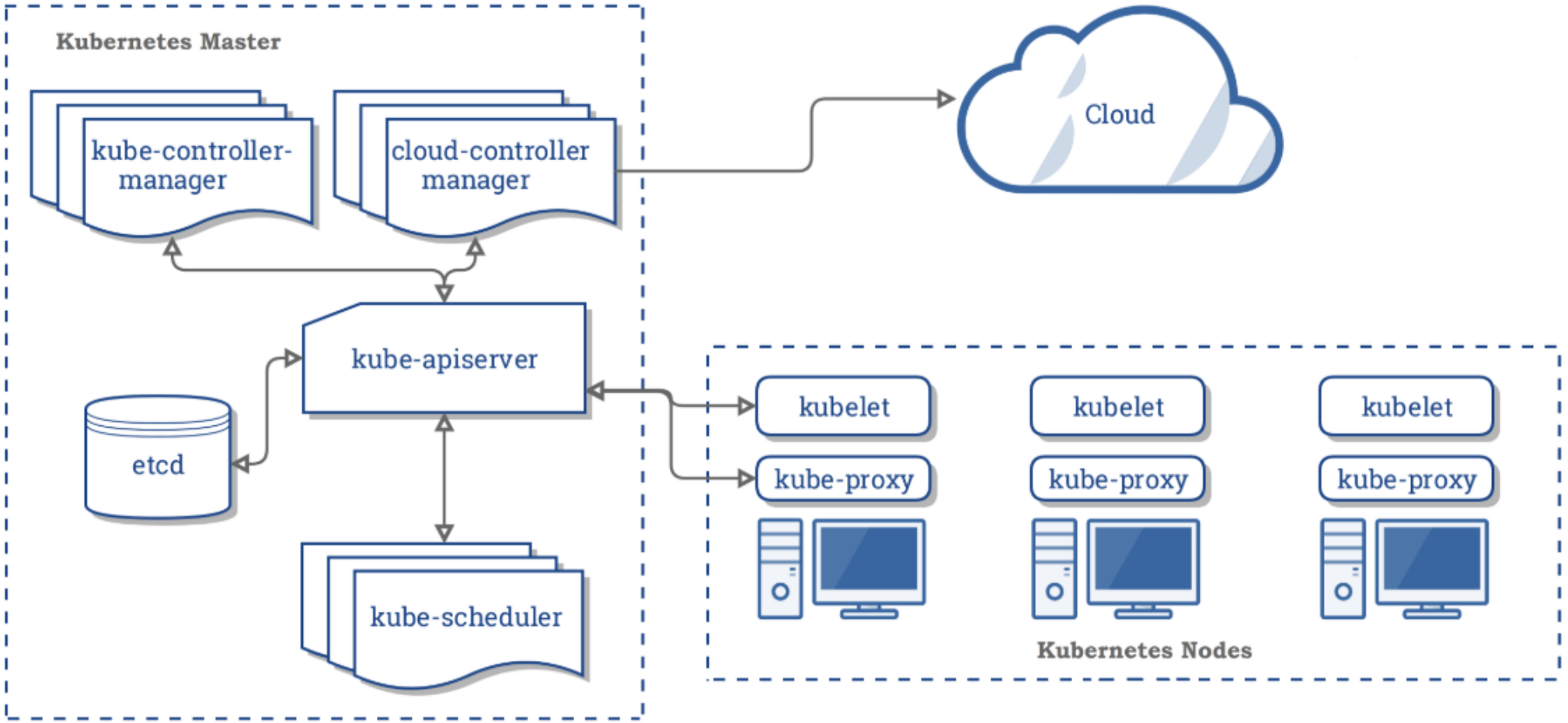
Anatomy of Kubernetes Cluster | Master Node

Master node main task is to manage the worker node(s) to run an application

The master node consists of:

1) **API server** contains various methods to directly access the Kubernetes
2) **Scheduler** assigns to each worker node an application
3) **Controller manager**
    3a) Keeps track of worker nodes
    3b) Handles node failures and replicates if needed
    3c) Provide endpoints to access the application from the outside world
4) **Cloud controller** communicates with cloud provide regarding resources such as nodes and IP addresses
5) **Etcd** works as backend for service discovery that stores the cluster's state and its configuration

Anatomy of Kubernetes Cluster | Worker Nodes



Anatomy of Kubernetes Cluster | Worker Nodes

A worker node consists of:

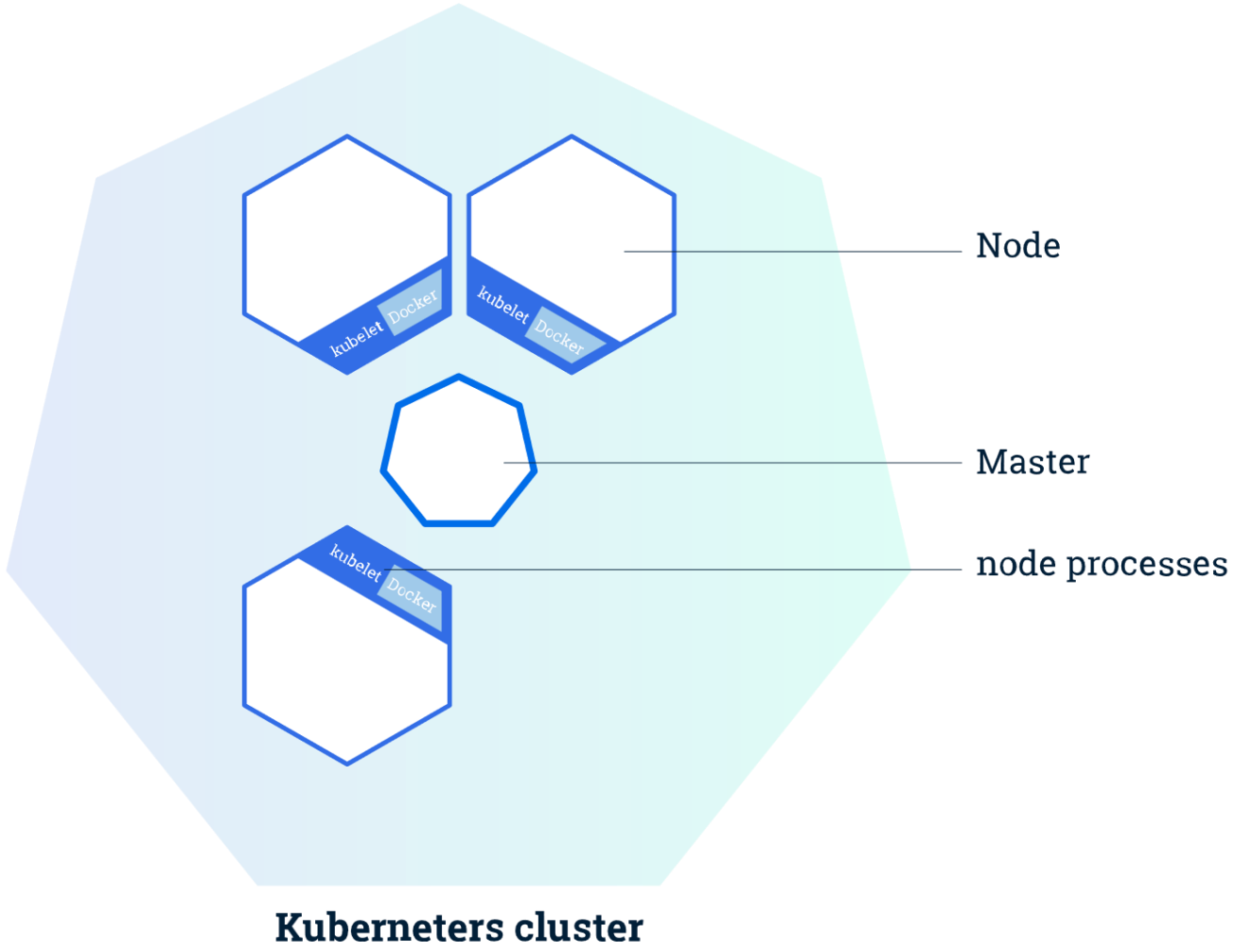
1) **Container runtime** that pulls a specified Docker image and deploys it on a worker node

2) **Kubelet** talks to the API server and manages containers on its node

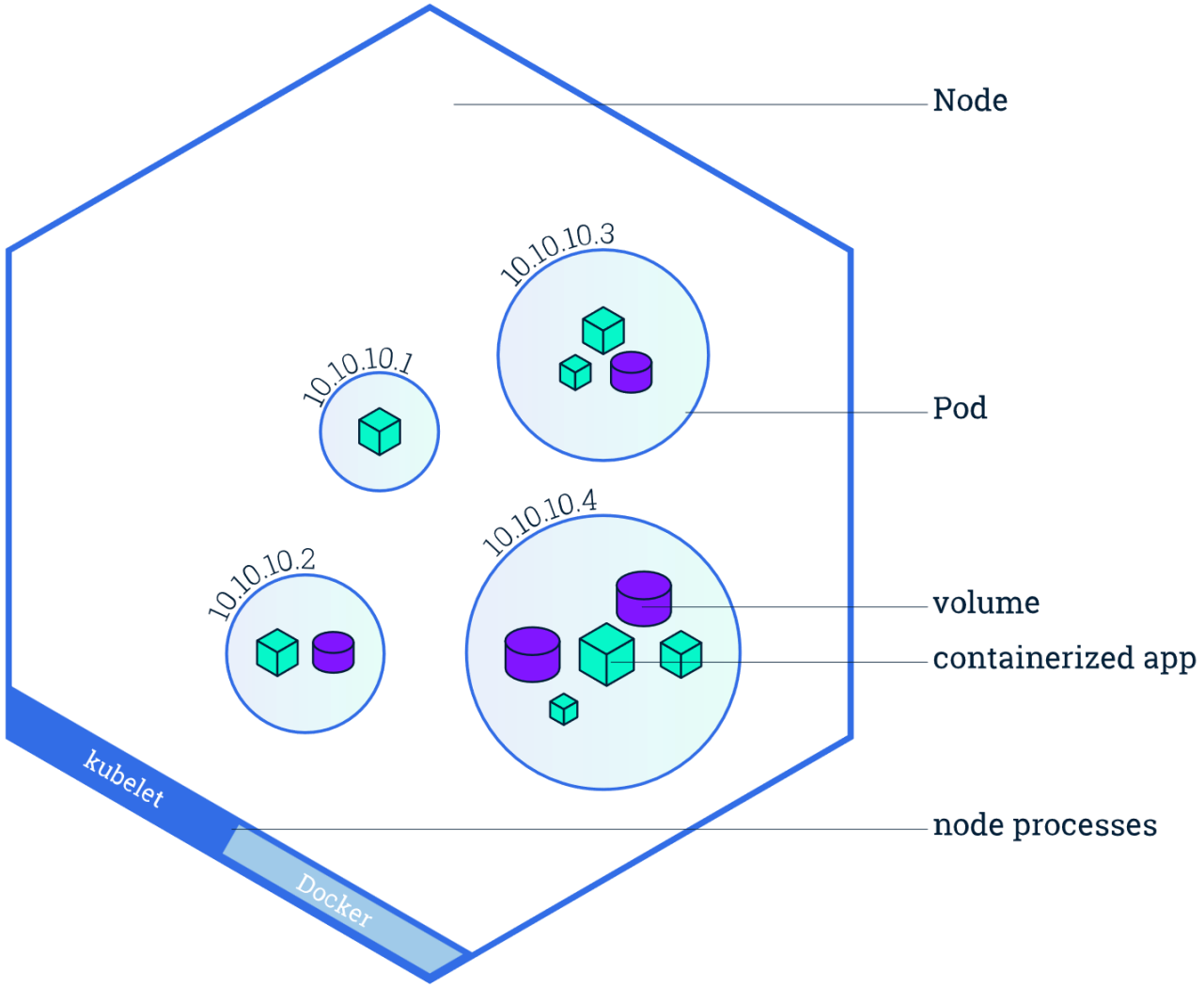
3) **Kube-proxy** load-balances network traffic between application components and the outside world

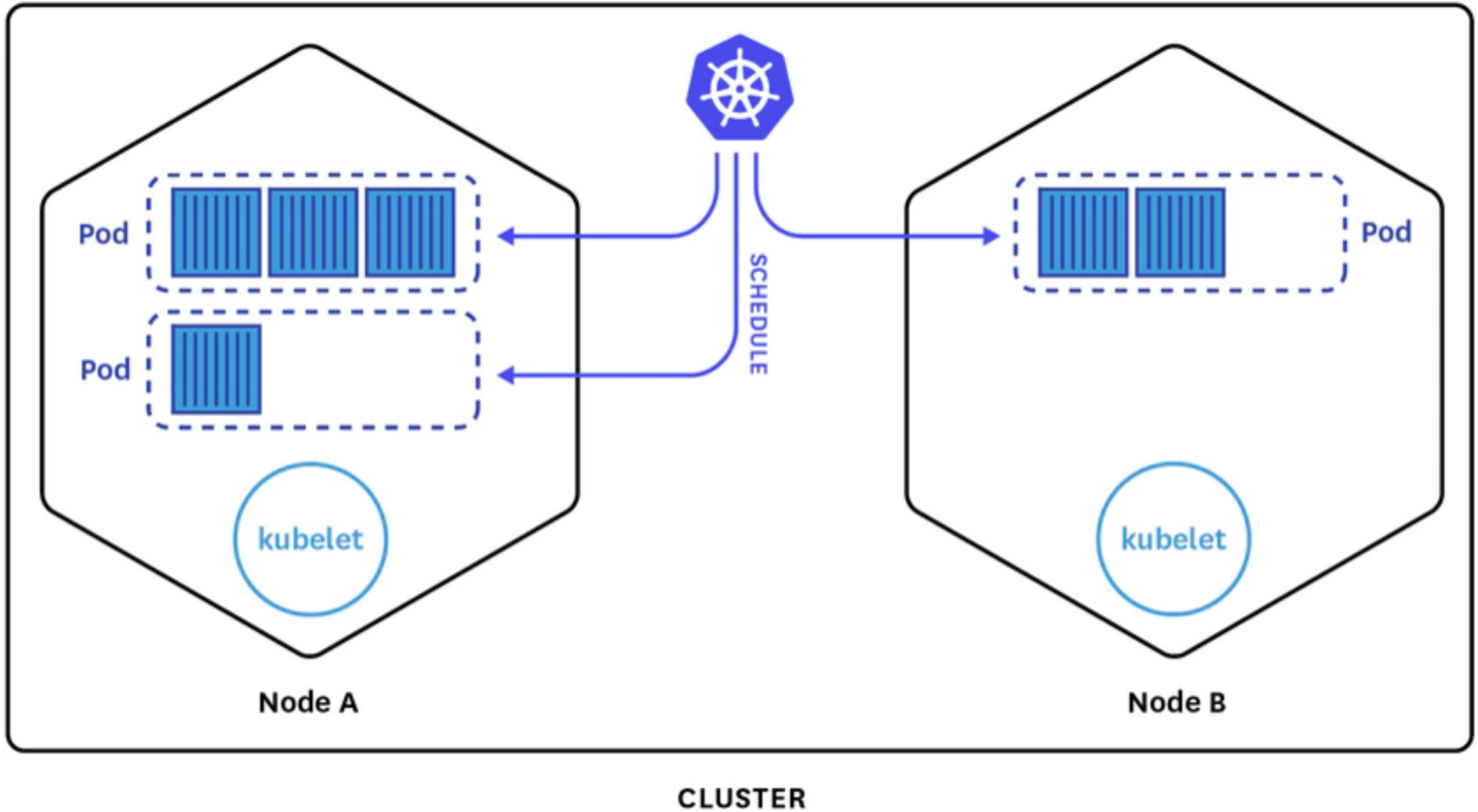
Kubernetes Cluster

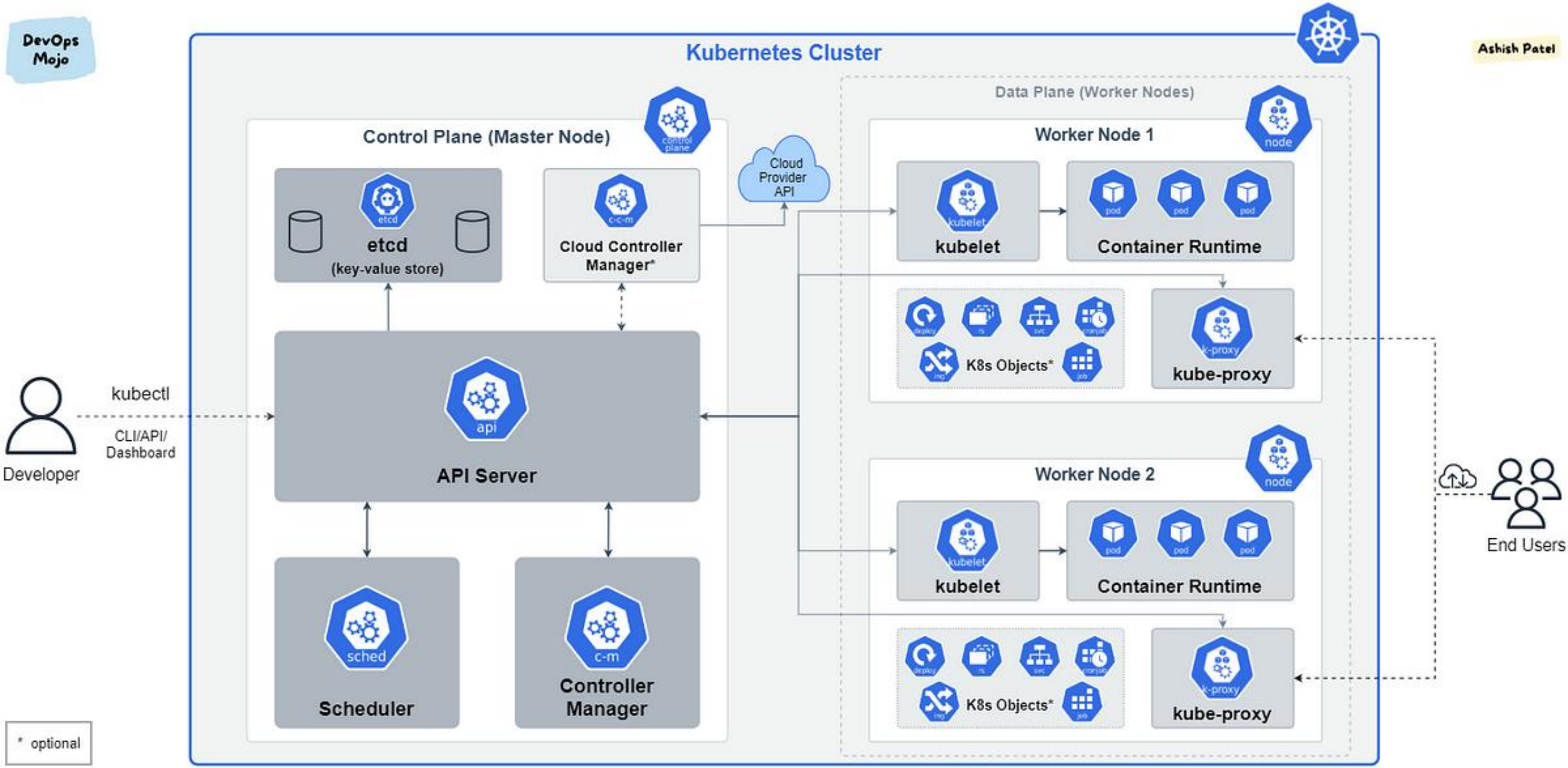
- The Master is responsible for managing the cluster.
- A node is a VM or a physical computer that is used as a worker machine in a Kubernetes cluster.

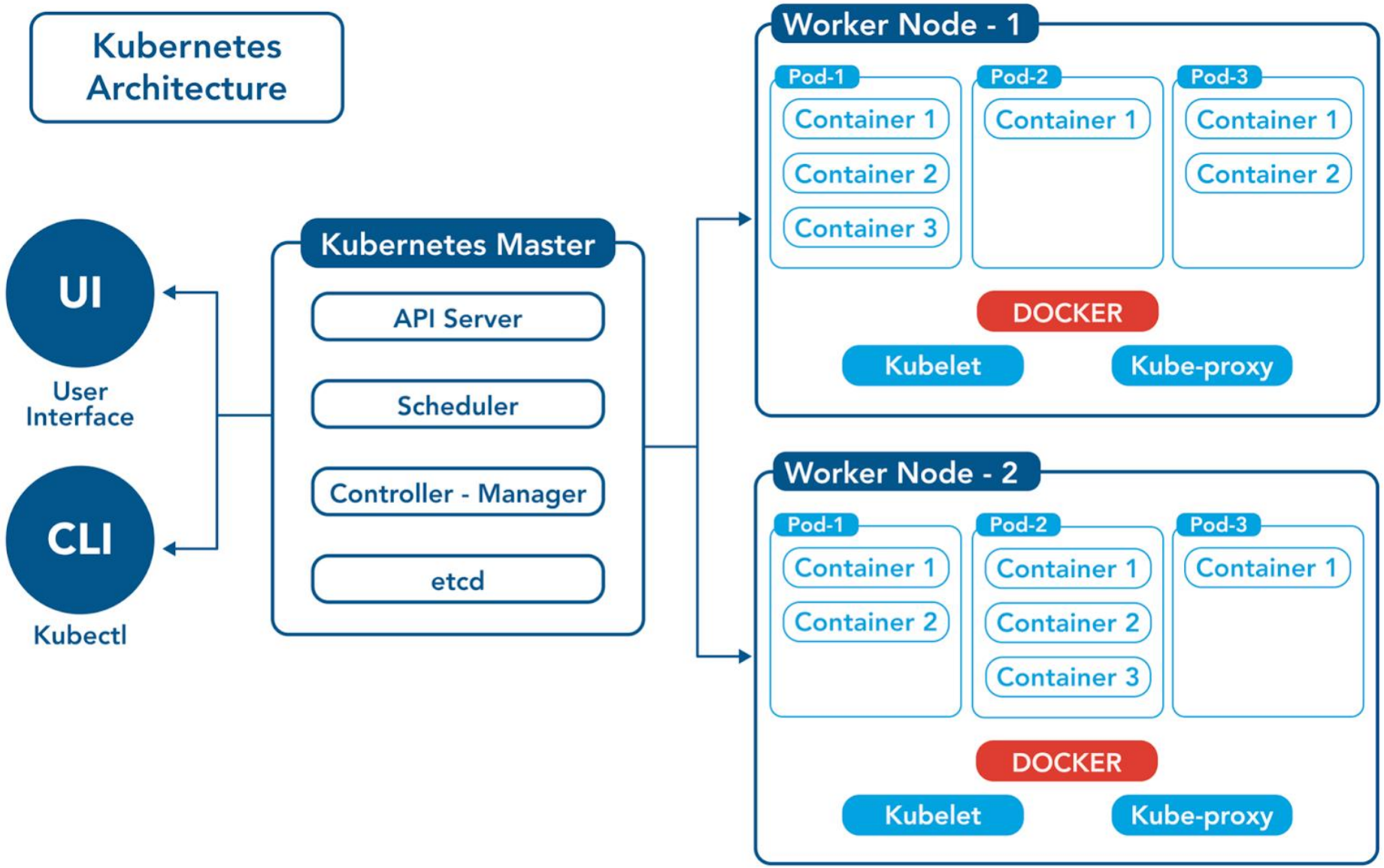


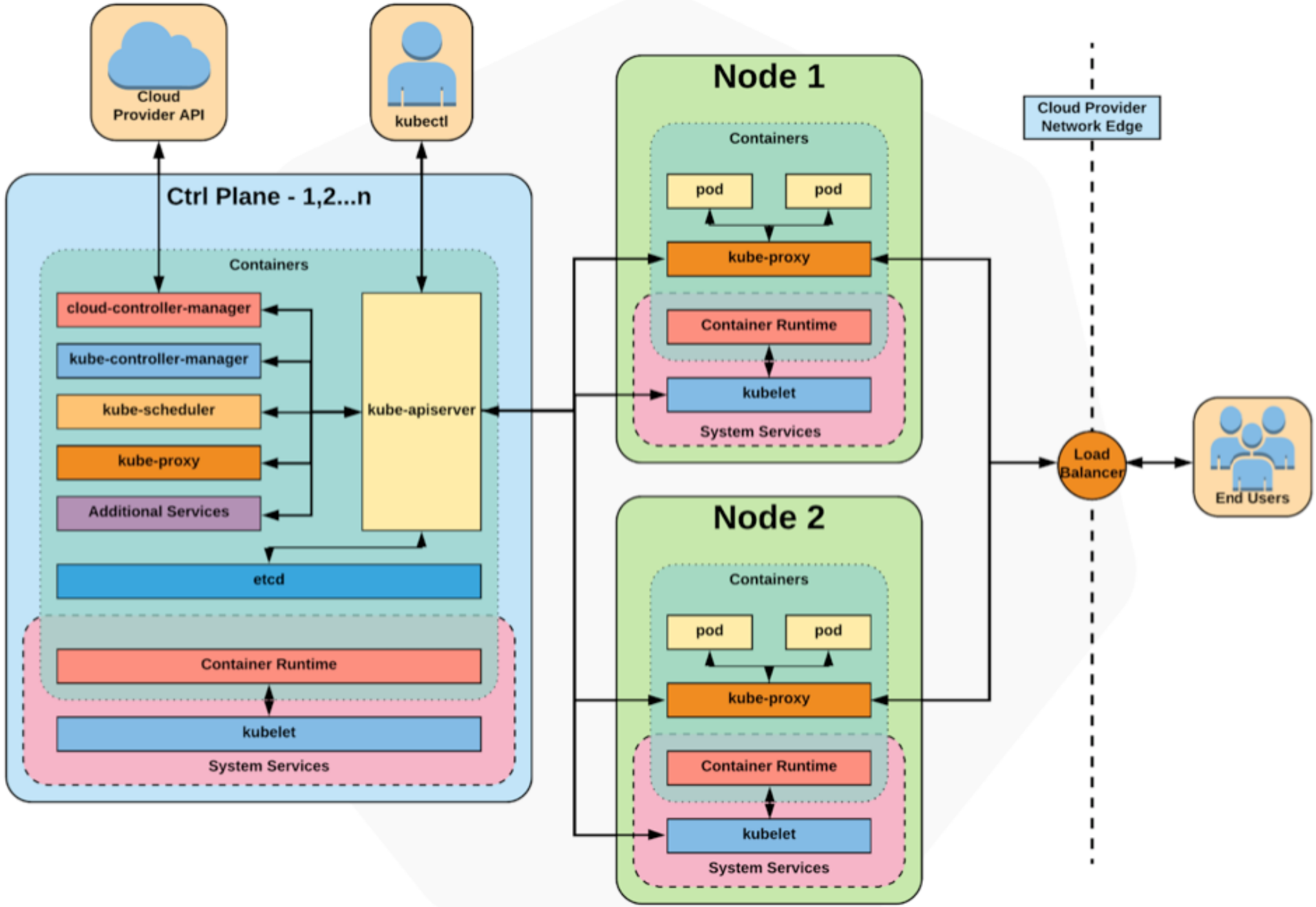
Node Overview




Pod
Pod
kubelet
Node A
SCHEDULE
Pod
kubelet
Node B
CLUSTER

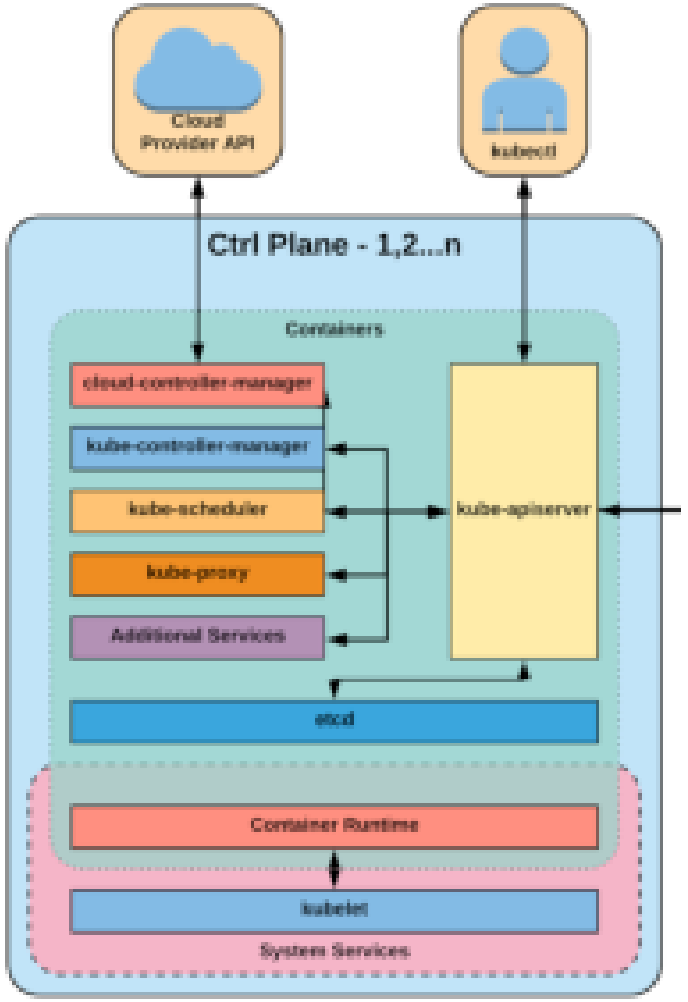

DevOps Mojo
Ashish Patel
Kubernetes Cluster
Control Plane (Master Node)
control plane
etcd
(key-value store)
c-c-m
Cloud Controller Manager*
Cloud Provider API
api
API Server
sched
Scheduler
c-m
Controller Manager
Data Plane (Worker Nodes)
Worker Node 1
node
kubelet
pod
Container Runtime
K8s Objects*
k-proxy
kube-proxy
Worker Node 2
Developer
kubectl
CLI/API/ Dashboard
End Users
* optional


Kubernetes Architecture
UI
User Interface
CLI
Kubectl
Kubernetes Master
API Server
Scheduler
Controller - Manager
etcd
Worker Node - 1
Pod-1
Container 1
Container 2
Container 3
Pod-2
Container 1
Pod-3
Container 1
Container 2
DOCKER
Kubelet
Kube-proxy
Worker Node - 2
Pod-1
Container 1
Container 2
Pod-2
Container 1
Container 2
Container 3
Pod-3
Container 1
DOCKER
Kubelet
Kube-proxy


Cloud Provider API
kubectl
Ctrl Plane - 1,2...n
Containers
cloud-controller-manager
kube-controller-manager
kube-scheduler
kube-proxy
Additional Services
kube-apiserver
etcd
Container Runtime
kubelet
System Services
Node 1
Containers
pod
pod
kube-proxy
Container Runtime
kubelet
System Services
Node 2
Containers
pod
pod
kube-proxy
Container Runtime
kubelet
System Services
Cloud Provider Network Edge
Load Balancer
End Users

Control Plane Components

- kube-apiserver
- etcd
- kube-controller-manager
- kube-scheduler
- cloud-controller-manager



kube-apiserver

- Provides a forward facing REST interface into the kubernetes control plane and datastore.
- All clients and other applications interact with kubernetes **strictly** through the API Server.
- Acts as the gatekeeper to the cluster by handling authentication and authorization, request validation, mutation, and admission control in addition to being the front-end to the backing datastore.

etcd

- etcd acts as the cluster datastore.
- Purpose in relation to Kubernetes is to provide a strong, consistent and highly available key-value store for persisting cluster state.
- Stores objects and config information.

etcd

Uses *"Raft Consensus"* among a quorum of systems to create a fault-tolerant consistent *"view"* of the cluster.

https://raft.github.io/

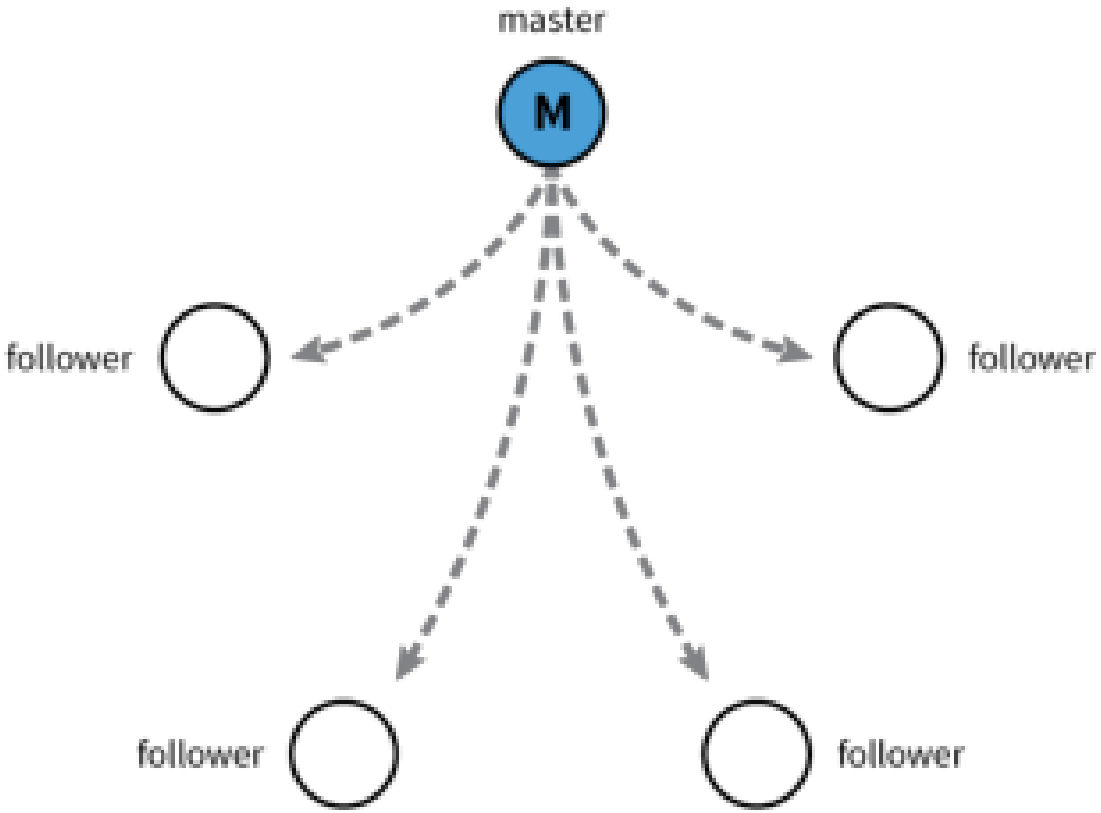


Image Source

kube-controller-manager

- Monitors the cluster state via the apiserver and **steers the cluster towards the desired state**.
- Node Controller: Responsible for noticing and responding when nodes go down.
- Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
- Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods).
- Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces.

kube-scheduler

- Component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on.
- Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

cloud-controller-manager

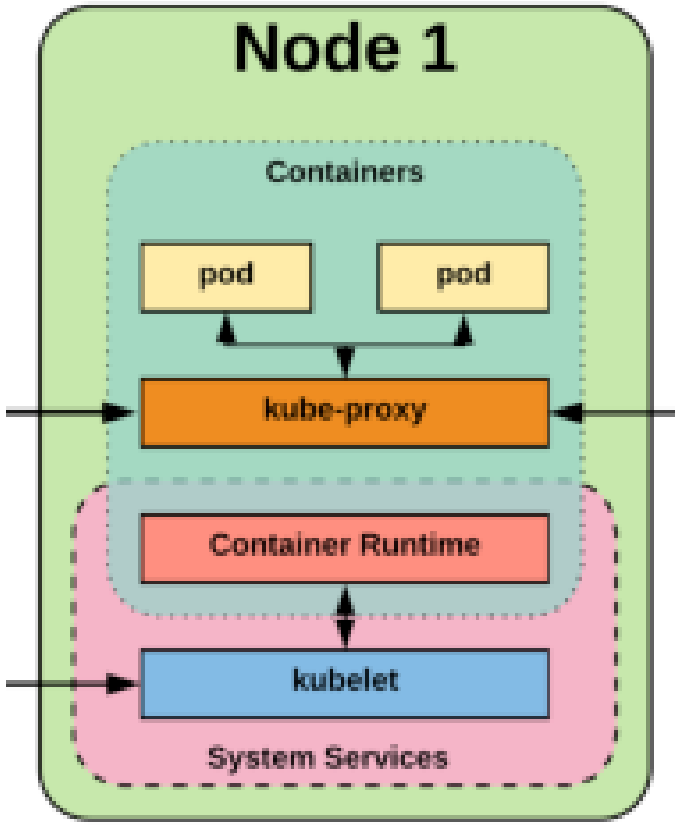
- Node Controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- Route Controller: For setting up routes in the underlying cloud infrastructure
- Service Controller: For creating, updating and deleting cloud provider load balancers
- Volume Controller: For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes

Node Components

Architecture Overview

Node Components

- kubelet
- kube-proxy
- Container Runtime Engine



kubelet

- An agent that runs on each node in the cluster. It makes sure that containers are running in a pod.
- The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy.

kube-proxy

- Manages the network rules on each node.
- Performs connection forwarding or load balancing for Kubernetes cluster services.

Container Runtime Engine

- A container runtime is a CRI (Container Runtime Interface) compatible application that executes and manages containers.
  - Containerd (docker)
  - Cri-o
  - Rkt
  - Kata (formerly clear and hyper)
  - Virtlet (VM CRI compatible runtime)

Primary concepts

Container: A sealed application package (Docker)

Pod: A small group of tightly coupled Containers

Labels: Identifying metadata attached to objects

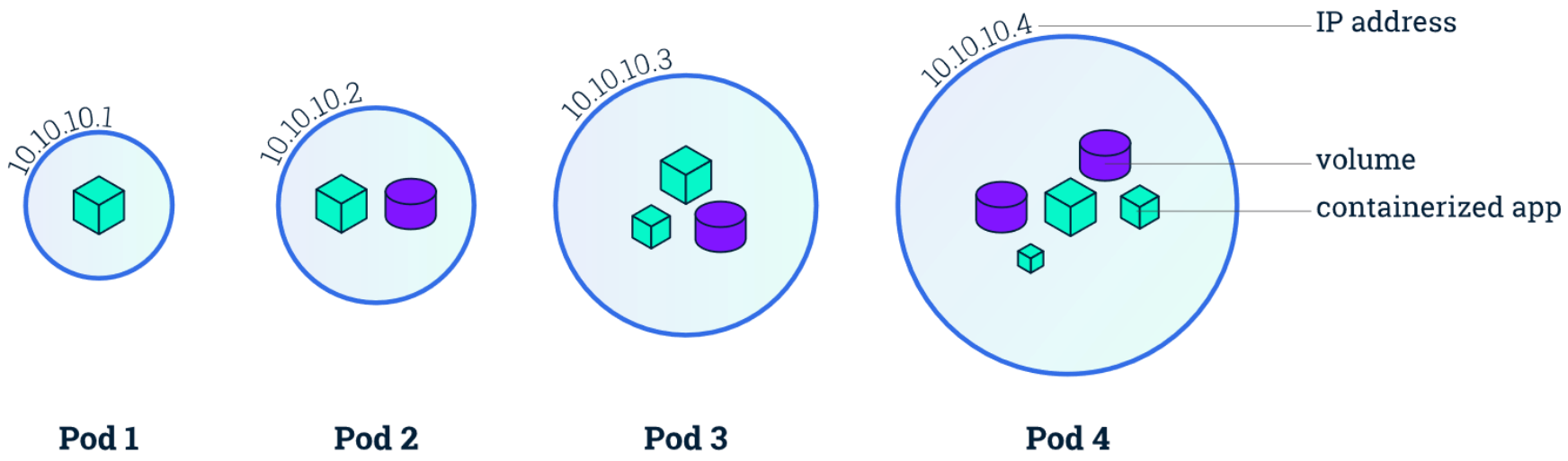
Selector: A query against labels, producing a set result

Controller: A reconciliation loop that drives current state towards desired state

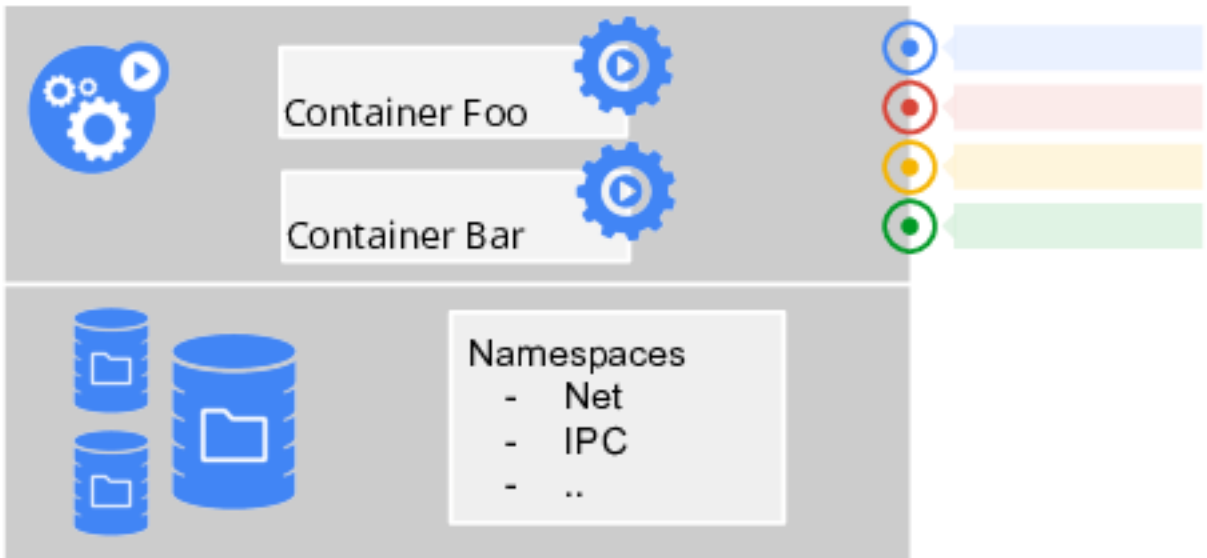
Service: A set of pods that work together

Pod

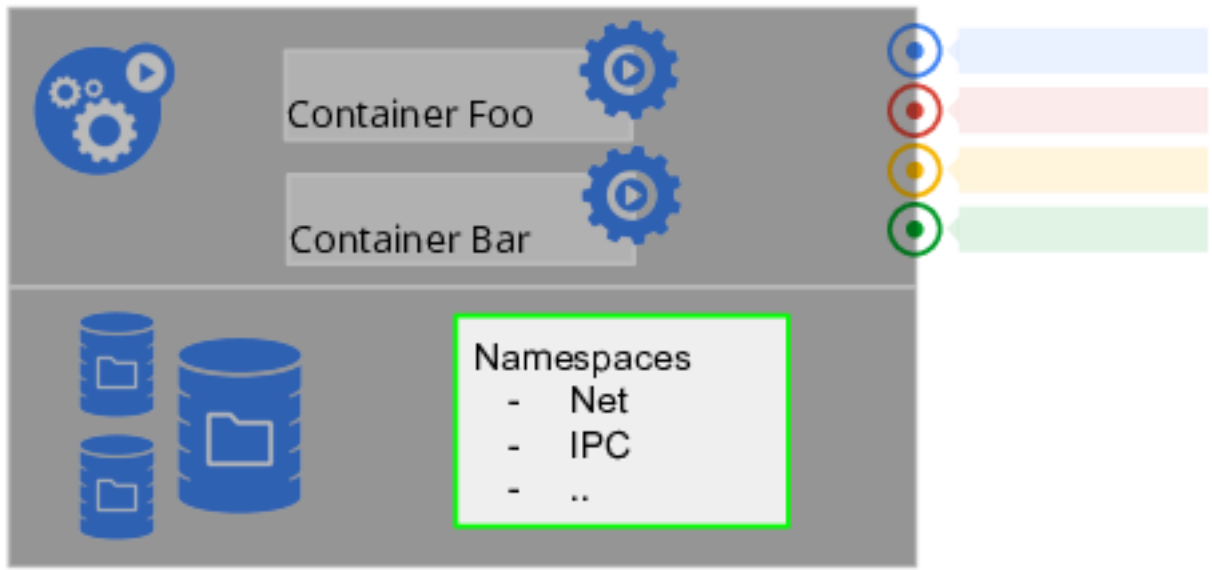
- A pod is a ground of one or more application containers (such as Docker) and includes shared storage (volumes), IP address and information about how to run them.



Pods: Grouping containers

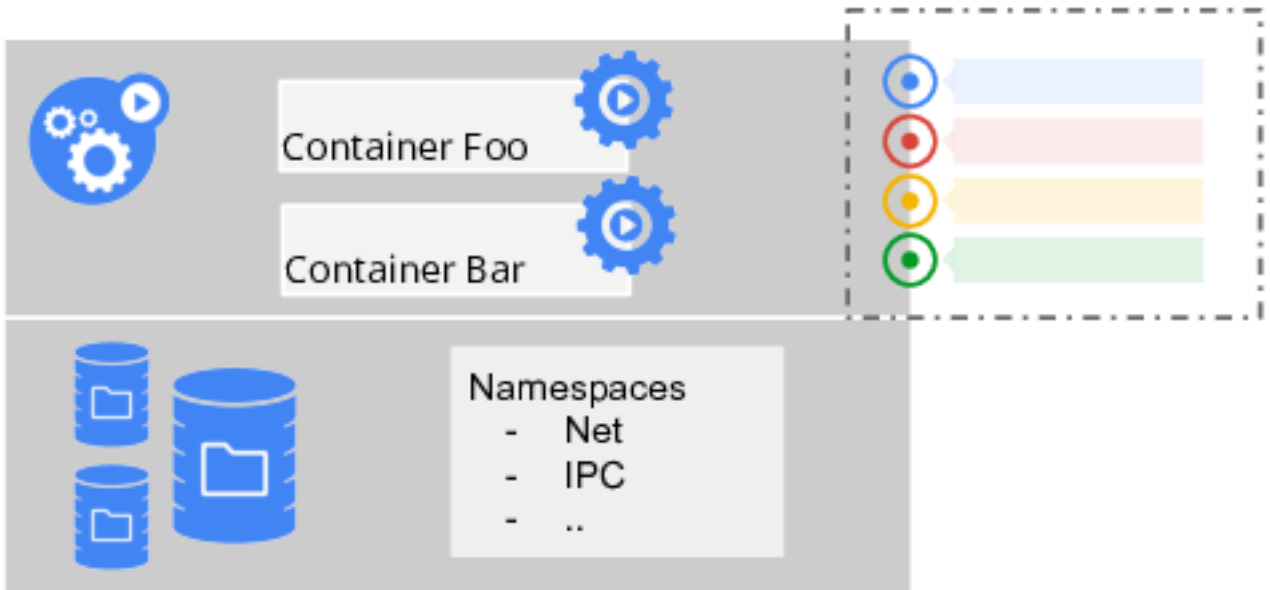


Pods: Networking



Pods: Volumes

Pods: Labels



Labels

Arbitrary metadata

Attached to any API object

Generally represent **identity**

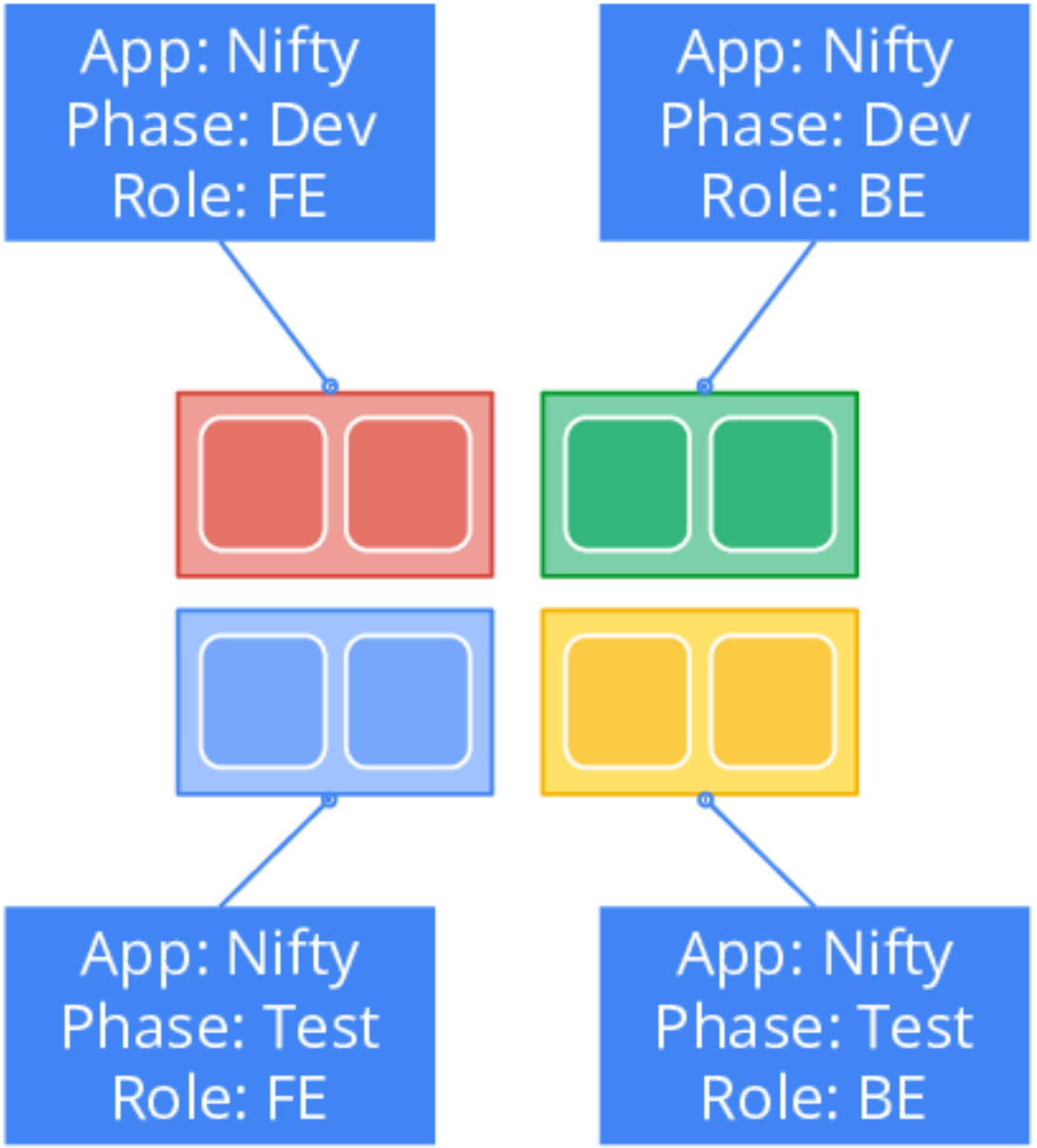
Queryable by **selectors**

- think SQL *'select ... where ...'*

The **only** grouping mechanism

Use to determine which objects to apply an operation to

- pods under a ReplicationController
- pods in a Service
- capabilities of a node (scheduling constraints)



Selectors

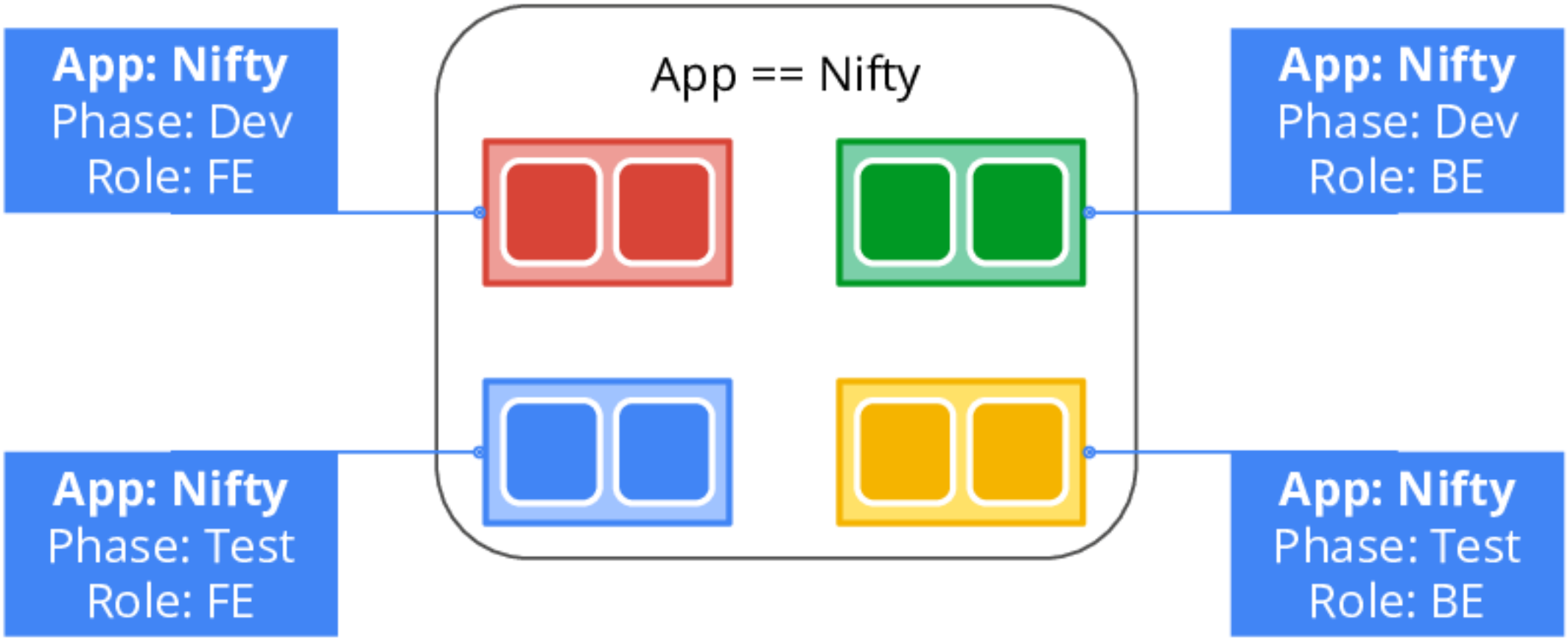
App: Nifty
Phase: Dev
Role: FE

App: Nifty
Phase: Dev
Role: BE

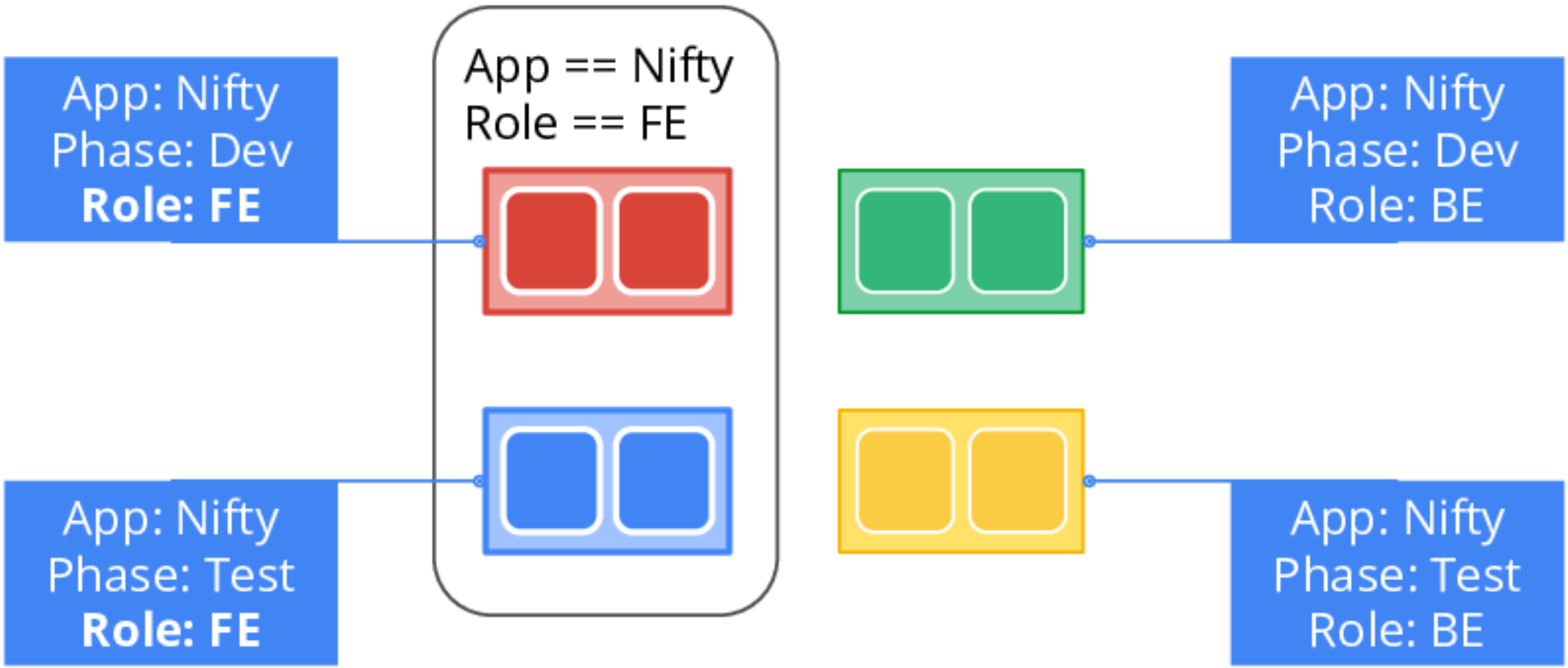
App: Nifty
Phase: Test
Role: FE

App: Nifty
Phase: Test
Role: BE

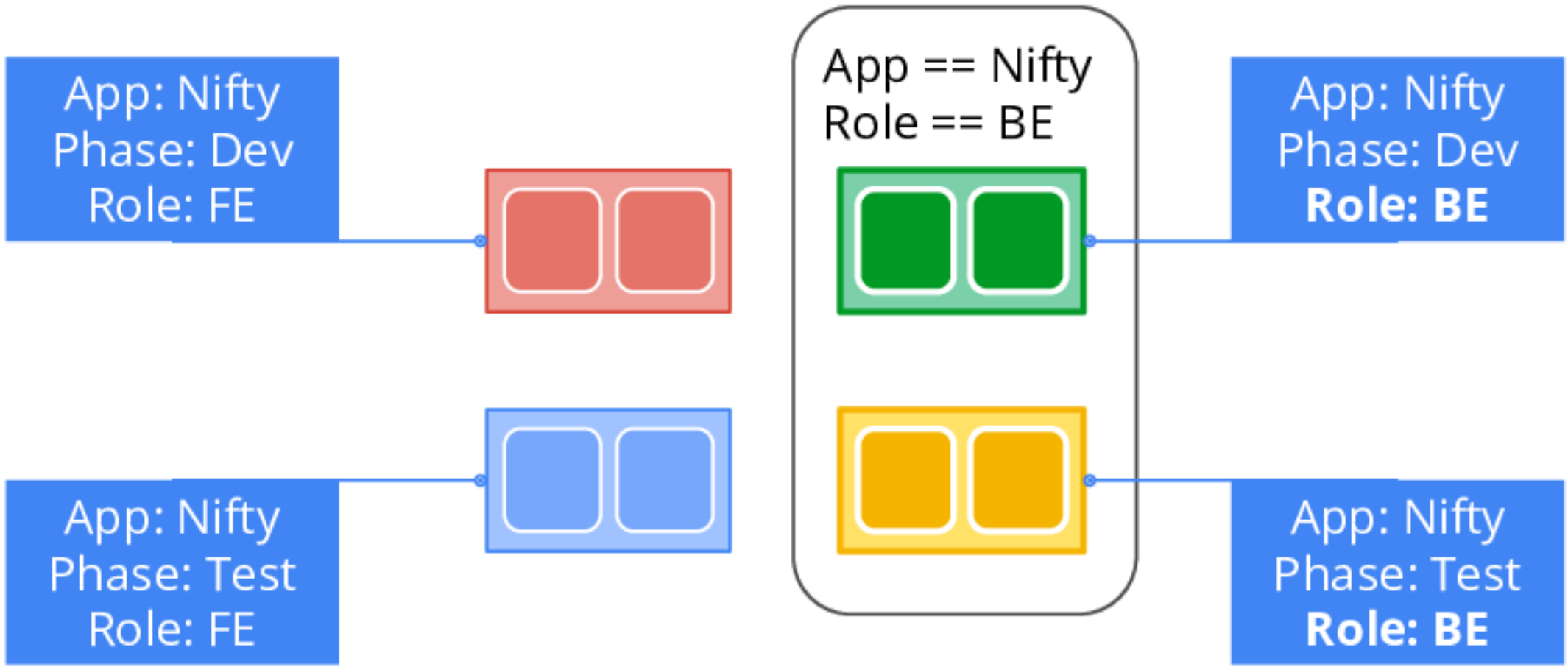
Selectors


App: Nifty
Phase: Dev
Role: FE
App == Nifty
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
Role: FE
App: Nifty
Phase: Test
Role: BE

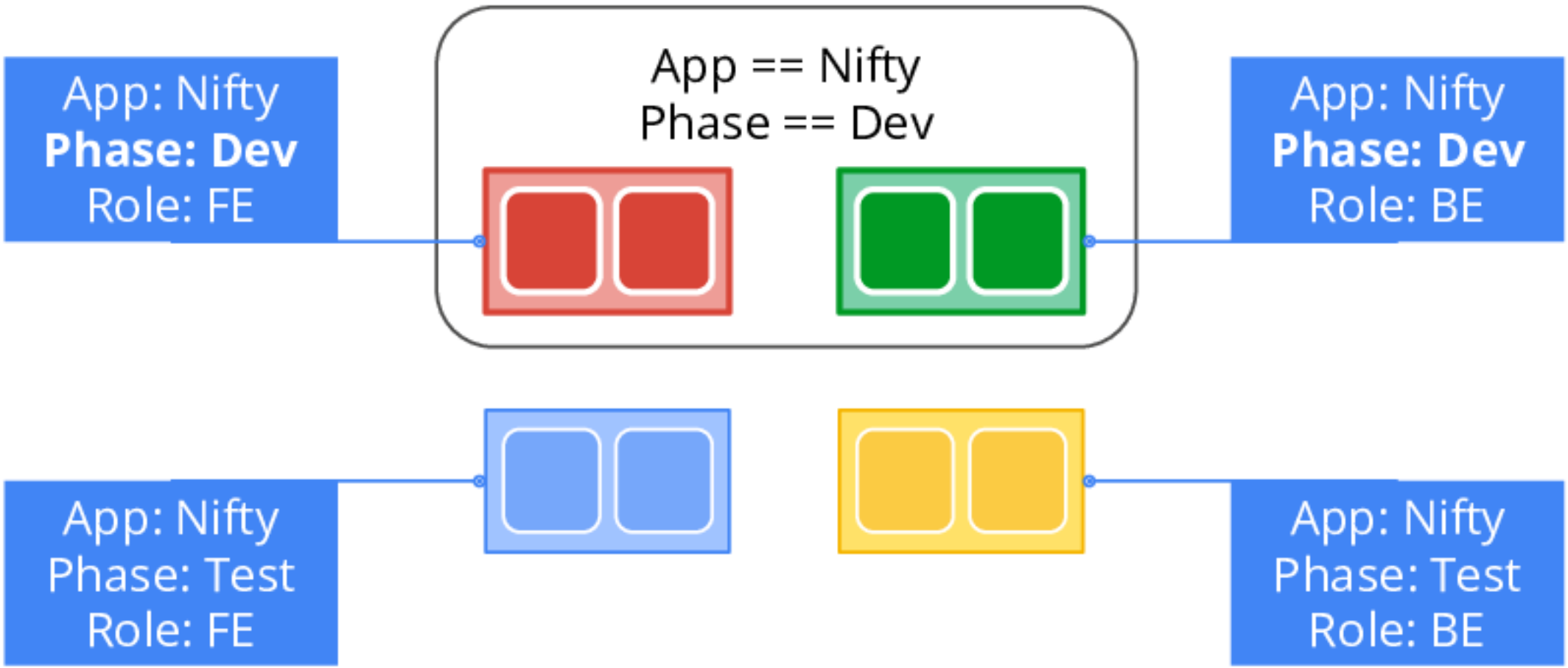
Selectors


App == Nifty
Role == FE
App: Nifty
Phase: Dev
Role: FE
App: Nifty
Phase: Test
Role: FE
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
Role: BE

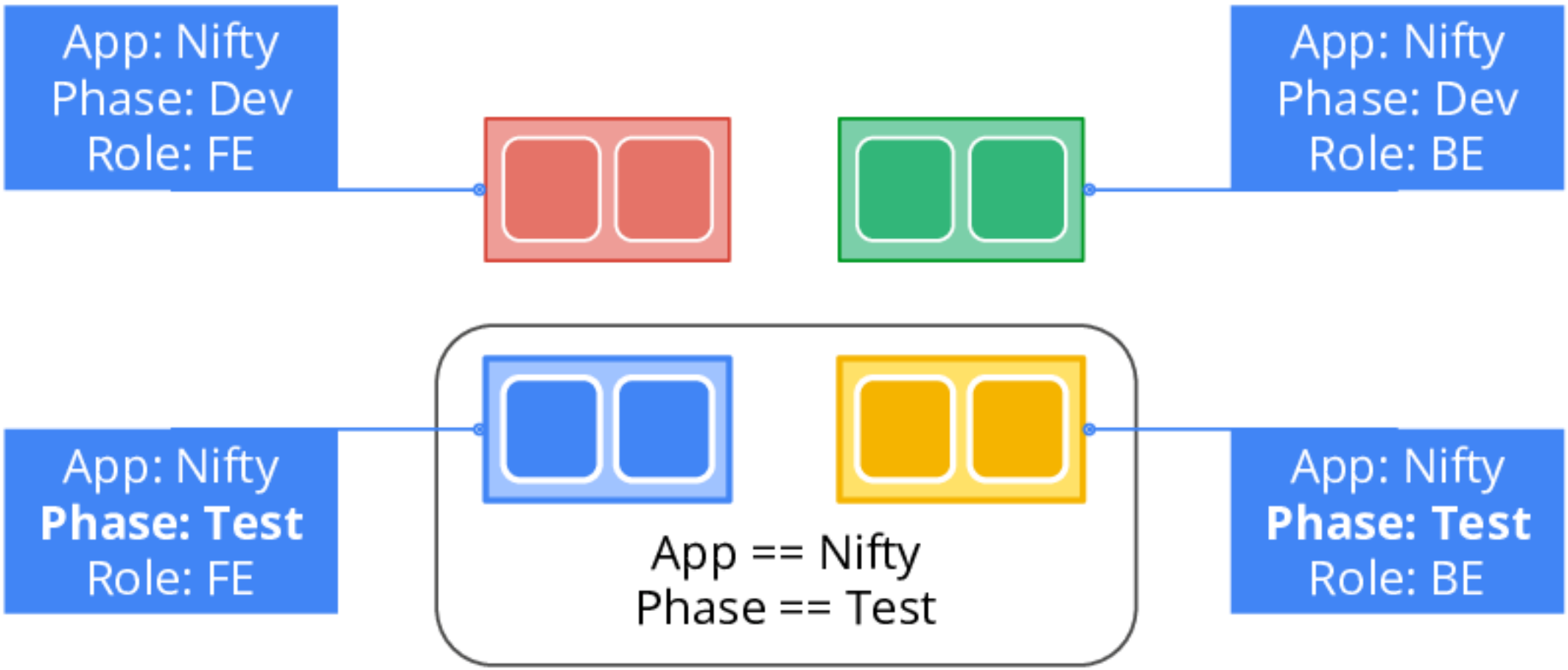
Selectors


App: Nifty
Phase: Dev
Role: FE
App: Nifty
Phase: Test
Role: FE
App == Nifty
Role == BE
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
Role: BE

Selectors


App == Nifty
Phase == Dev
App: Nifty
Phase: Dev
Role: FE
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
Role: FE
App: Nifty
Phase: Test
Role: BE

Selectors


App: Nifty
Phase: Dev
Role: FE
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
Role: FE
App: Nifty
Phase: Test
Role: BE
App == Nifty
Phase == Test

Replication Controllers

A type of *controller* (control loop)

Ensure N copies of a pod always running

- if too few, start new ones
- if too many, kill some
- group == selector

Cleanly layered on top of the core

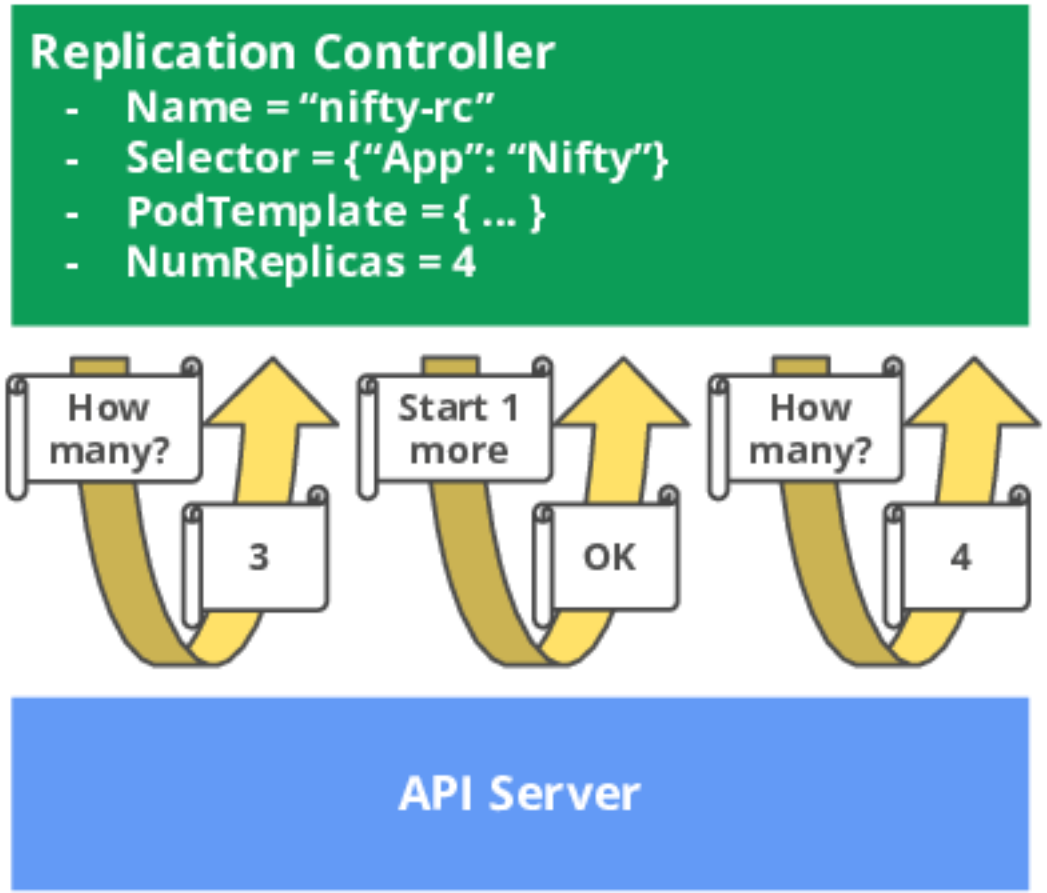
- all access is by public APIs

Replicated pods are fungible

- No implied ordinality or identity

Other kinds of controllers coming

- e.g. job controller for batch



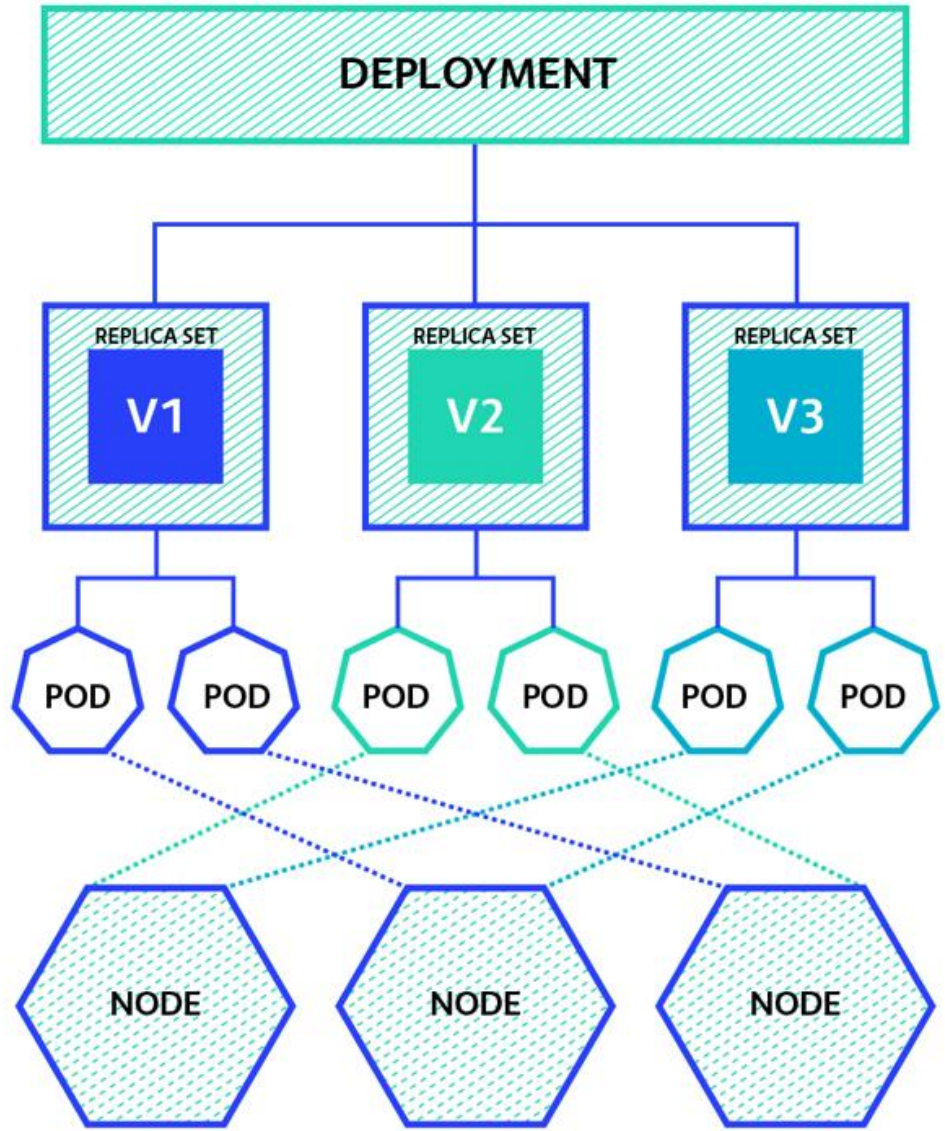
ReplicaSet

- A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time.
  - guarantee the availability of a specified number of identical Pods.
- Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features.

ReplicaSet

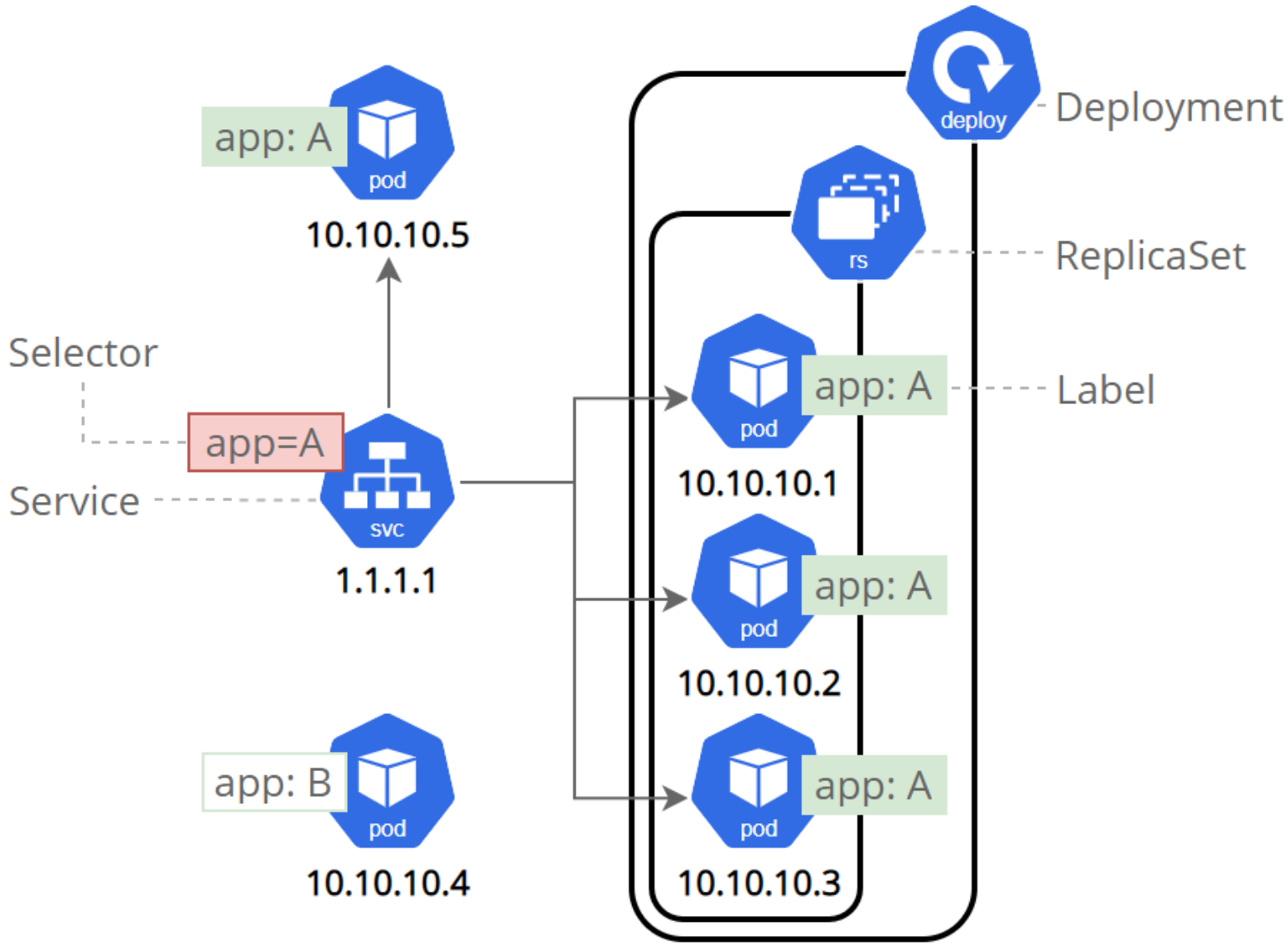
- kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml

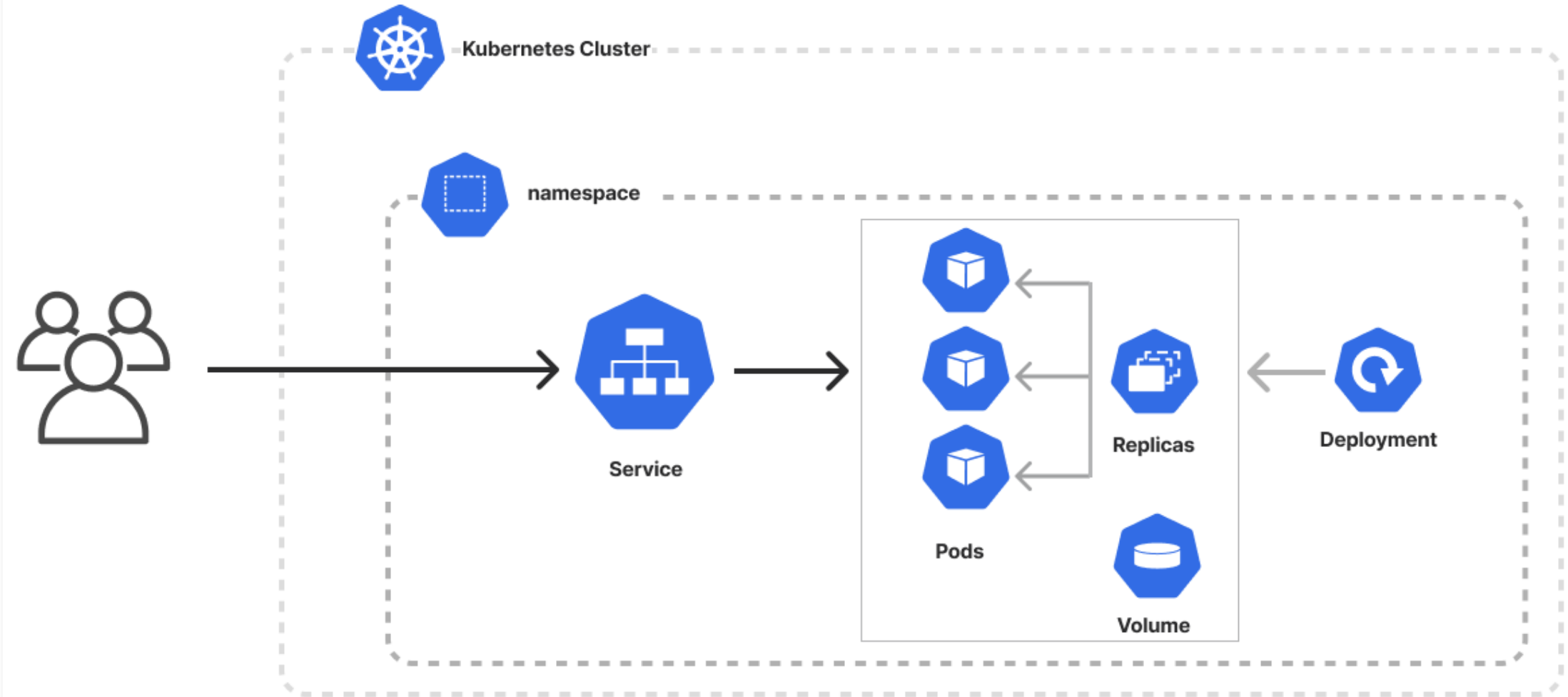
- submit it to a Kubernetes cluster will create the defined ReplicaSet and the Pods that it manages.
- kubectl get rs
- kubectl describe rs/frontend
- kubectl get pods
- kubectl get pods frontend-b2zdv -o yaml
  - replace frontend-b2zdv with your pod name


DEPLOYMENT
REPLICA SET
V1
REPLICA SET
V2
REPLICA SET
V3
POD
POD
POD
POD
POD
POD
NODE
NODE
NODE

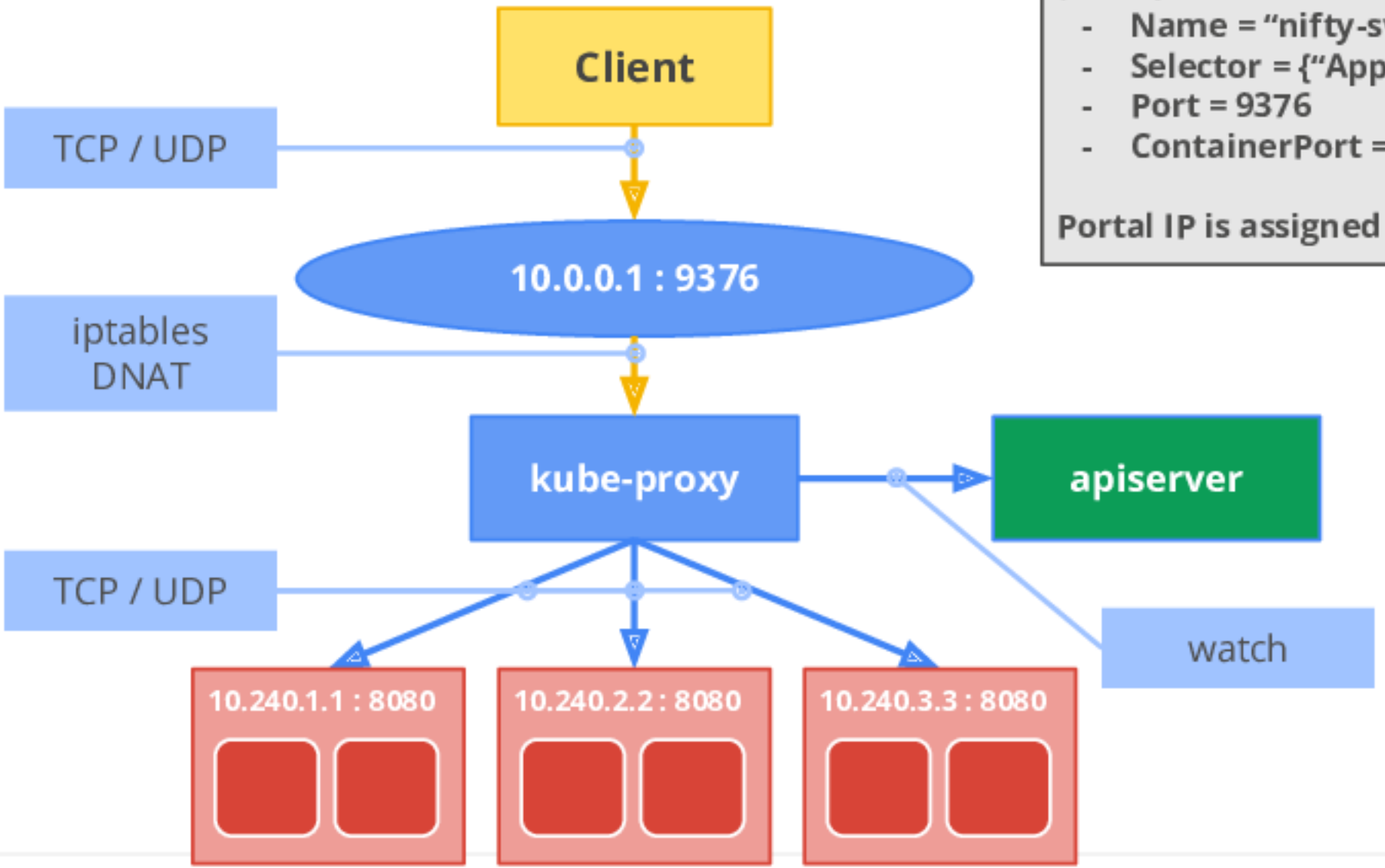
Deployment

- A Kubernetes Deployment tells Kubernetes how to create or modify instances of the pods that hold a containerized application.
- Deployments can help to efficiently scale the number of replica pods, enable the rollout of updated code in a controlled manner, or roll back to an earlier deployment version if necessary.


app: A
pod
10.10.10.5
Selector
app=A
Service
svc
1.1.1.1
app: B
pod
10.10.10.4
deploy
Deployment
rs
ReplicaSet
pod
app: A
Label
10.10.10.1
pod
app: A
10.10.10.2
pod
app: A
10.10.10.3


Kubernetes Cluster
namespace
Service
Pods
Replicas
Deployment
Volume

Services

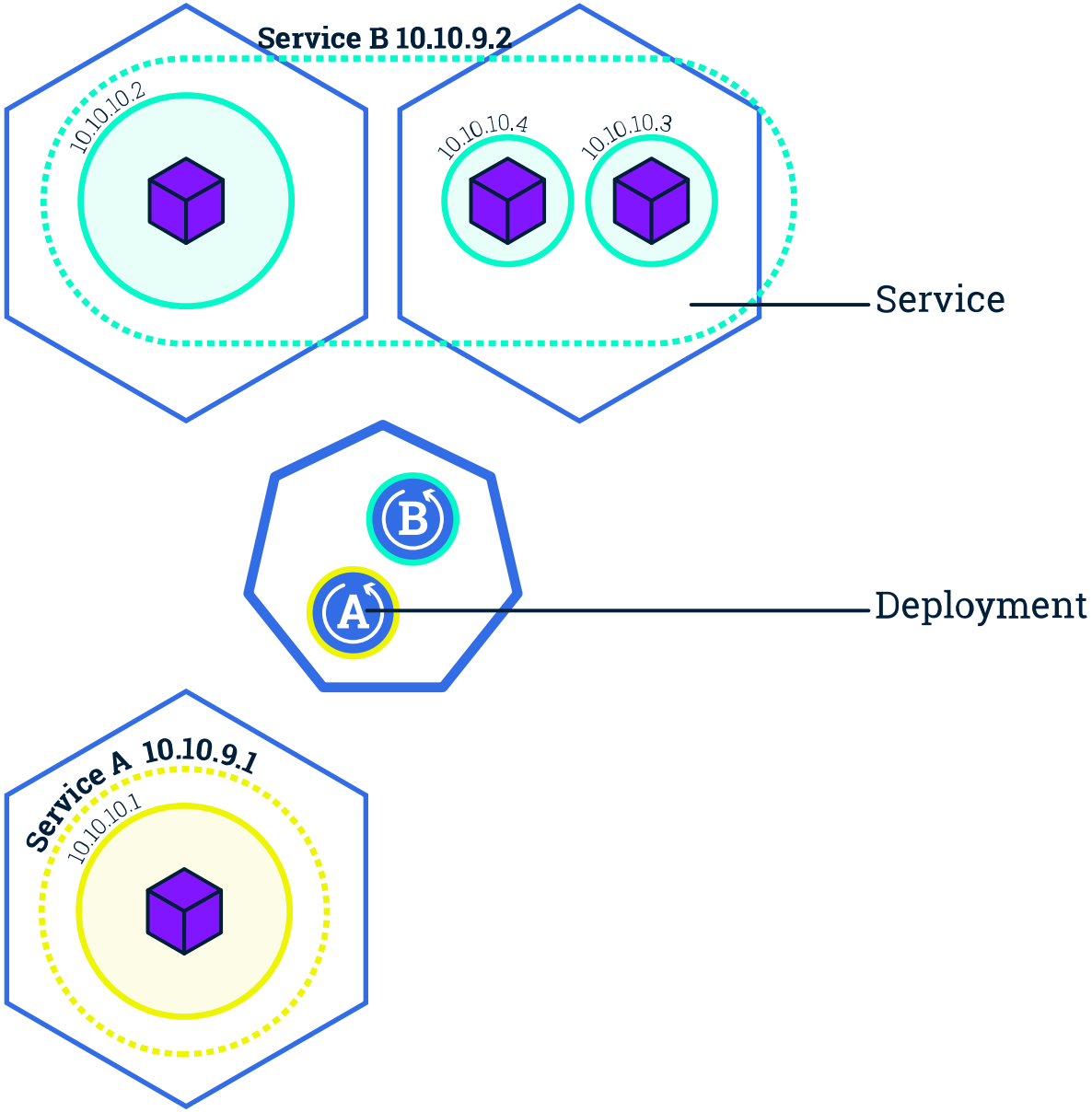


Service
- Name = “nifty-svc”
- Selector = {“App”: “Nifty”}
- Port = 9376
- ContainerPort = 8080

Portal IP is assigned

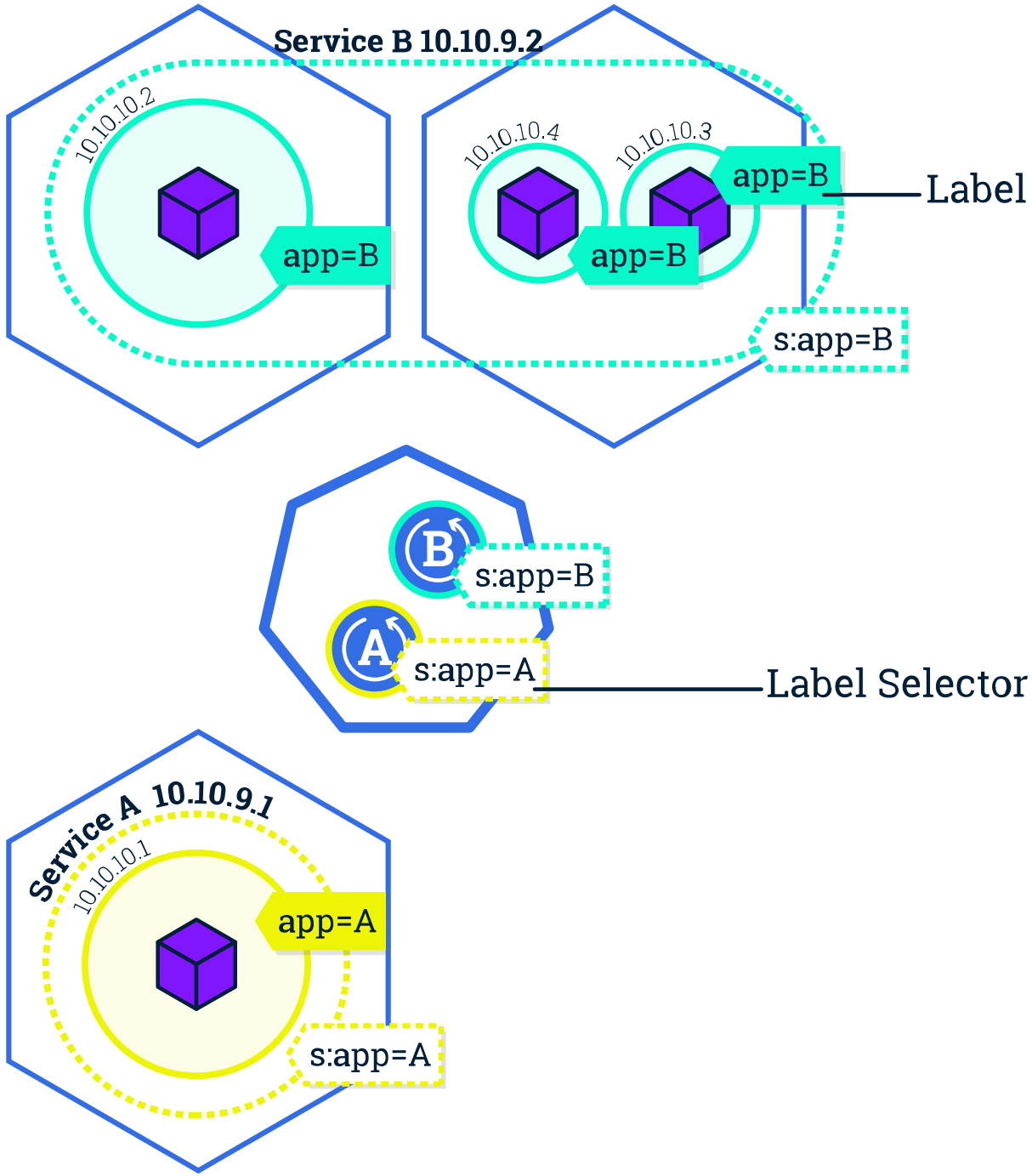
Services

- Load balancing of traffic across the contained set of Pods.
  - Useful when a service is created to group all Pods from a specific Deployment.
- Service discovery within the cluster. This will for example allow a frontend service (like a webserver) to receive traffic from a backend service (like a database) without worrying about Pods.
- Services match a set of Pods using Label Selectors, a grouping primitive that allows logical operation on Labels.



Labels

- Labels can be attached to objects at the creation time or later and can be modified at any time.
- The kubectl run command sets some default Labels/Label Selectors on the new Pods/ Deployment.
- The link between Labels and Label Selectors defines the relationship between the Deployment and the Pods it creates.



ClusterIP

- *ClusterIP* is the default service type in Kubernetes, and it provides internal connectivity between different components of our application.
- **Kubernetes assigns a virtual IP address to a *ClusterIP* service that can solely be accessed from within the cluster during its creation**. This IP address is stable and doesn't change even if the pods behind the service are rescheduled or replaced.

ClusterIP

- excellent choice for internal communication between different components of our application that don't need to be exposed to the outside world.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: backend
  ports:
  - name: http
    port: 80
    targetPort: 8080
```

ClusterIP

- we define a service named *backend* with a selector that targets pods labeled with *app: backend.*
- The service exposes port *80*, which is the port used by clients to access the service, and forwards the traffic to the pods' port *8080*, which is where the backend application is running.

NodePort

- *NodePort* services extend the functionality of *ClusterIP* services by enabling external connectivity to our application.
- When we create a *NodePort* service on any node within the cluster that meets the defined criteria, **Kubernetes opens up a designated port that forwards traffic to the corresponding *ClusterIP* service running on the node.**

NodePort

- These services are ideal for applications that need to be accessible from outside the cluster, such as web applications or APIs.
- With *NodePort* services, we can access our application using the node's IP address and the port number assigned to the service.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  selector:
    app: frontend
  type: NodePort
  ports:
    - name: http
      port: 80
      targetPort: 8080
```

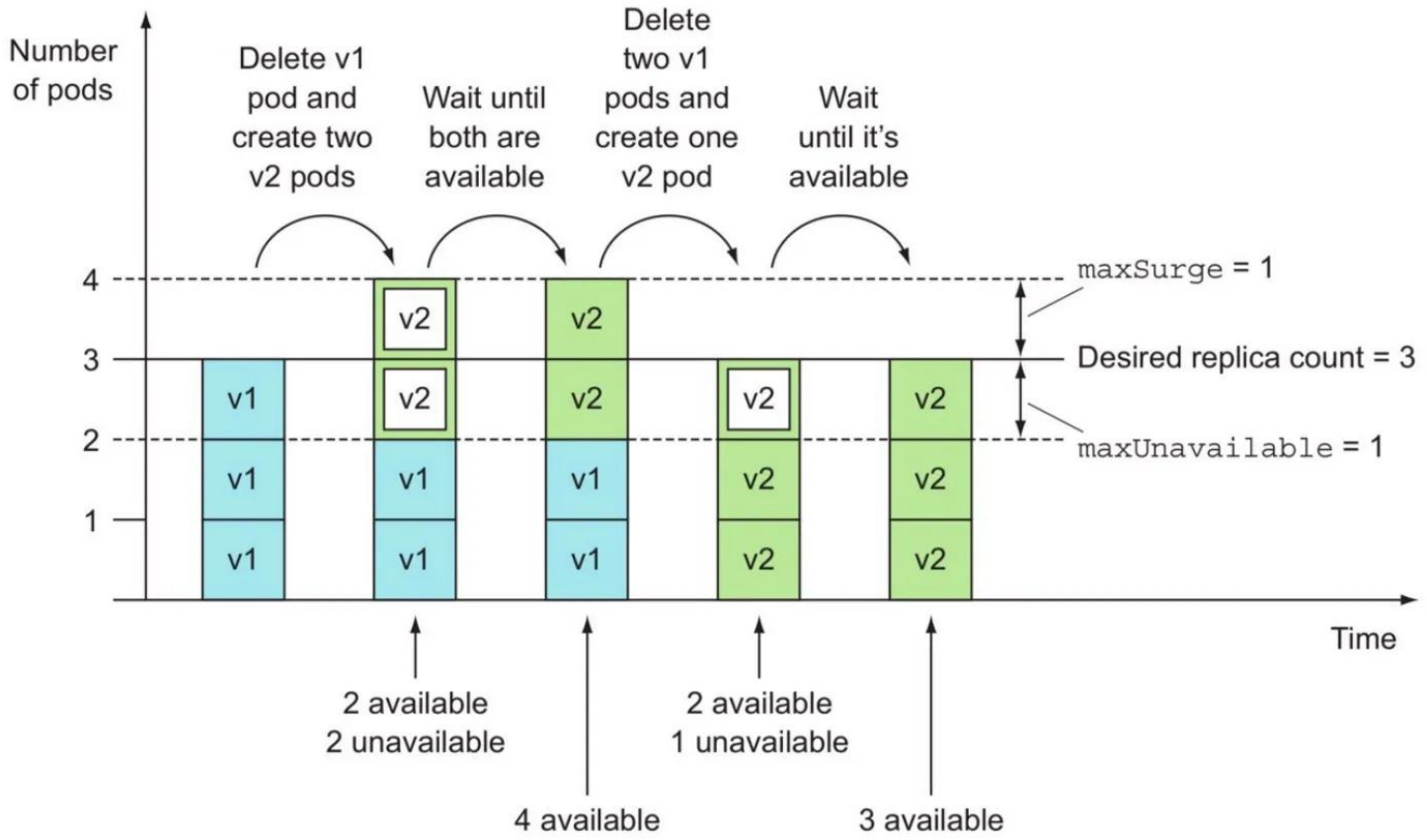
NodePort

- define a service named *frontend* that targets pods labeled with *app: frontend* by setting a selector.
- The service exposes port *80* and forwards the traffic to the pods' port *8080*.
- We set the service type to *NodePort*, and Kubernetes exposes the service on a specific port on a qualifying node within the cluster.

NodePort Service

- When we create a *NodePort* service, Kubernetes assigns a port number from a predefined range of *30000-32767*. Additionally, we can specify a custom port number by adding the *nodePort* field to the service definition:

Rolling Update



Common kubectl Commands

- Useful commands to complete the exercise:

```
$ kubectl create -f app-db-deploymnet.yaml
$ kubectl get deployment
$ kubectl get pods
$ kubectl get pods /
  -o=custom-columns=NAME:.metadata.name,IP:.status.podIP
$ kubectl create -f app-server-deploymnet.yaml
$ kubectl expose deployment /
app-deployment --type=LoadBalancer --port=8080
$ kubectl get services
$ kubectl delete service app-deployment
$ kubectl delete deployment app-server-deployment
$ kubectl delete deployment app-db-deployment
```