Kubernetes

Antonio Brogi

Department of Computer Science
University of Pisa

Why K8s?

Containers

Containers provide a lightweight mechanism for isolating an application's environment

Container images can be executed reliably on any machine, providing us portability from development to deployment

Multiple workloads can be placed on the same physical machine, achieving higher resource (memory and CPU) utilization

Questions

What happens if your container dies?

What happens if the machine running your container fails?

What if you have multiple containers that need the communicate one another? How do you enable networking between containers?

If your production environment consists of multiple machines, how do you decide which machine to use to run your container?

Container orchestration

The director of an orchestra holds the **vision** for a musical performance and **communicates** with the musicians to **coordinate** their individual instrumental contributions to achieve this overall vision

The architect of a system simply **composes the music** (defines the containers to be run) and then hands over control to the orchestra director (container orchestration platform) to achieve that vision

Kubernetes: A container orchestration platform

K8s manages the entire lifecycle of individual containers, spinning up and shutting down resources as needed

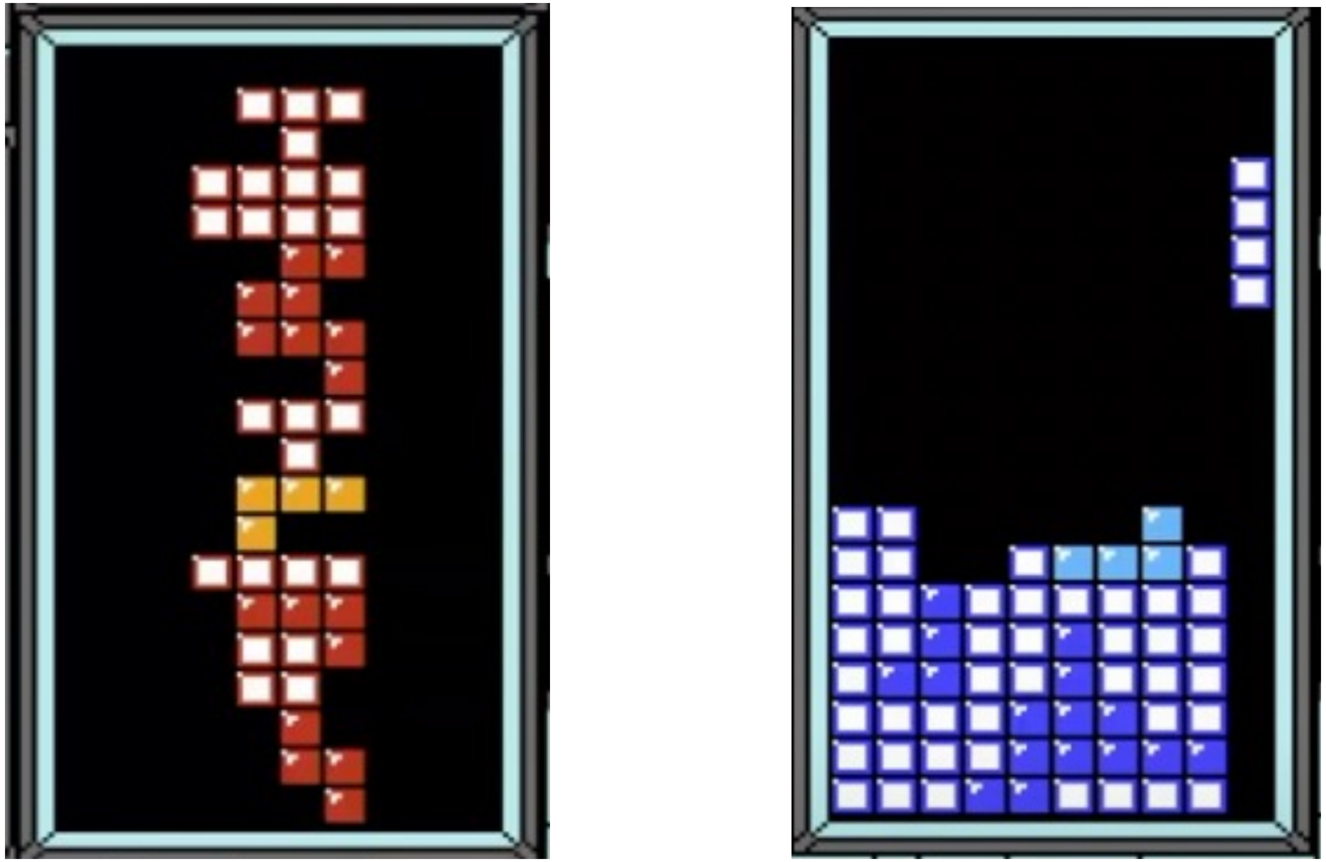
- if a container shuts down unexpectedly, K8s reacts by launching another container in its place

K8s provides a mechanism for applications to communicate with each other even as underlying individual containers are created and destroyed

Given a set of container workloads to run and a set of machines on a cluster, the container orchestrator examines each container and determines the optimal machine to schedule that workload

J. Jordan. An introduction to Kubernetes.

Automated deployment vs. container orchestration



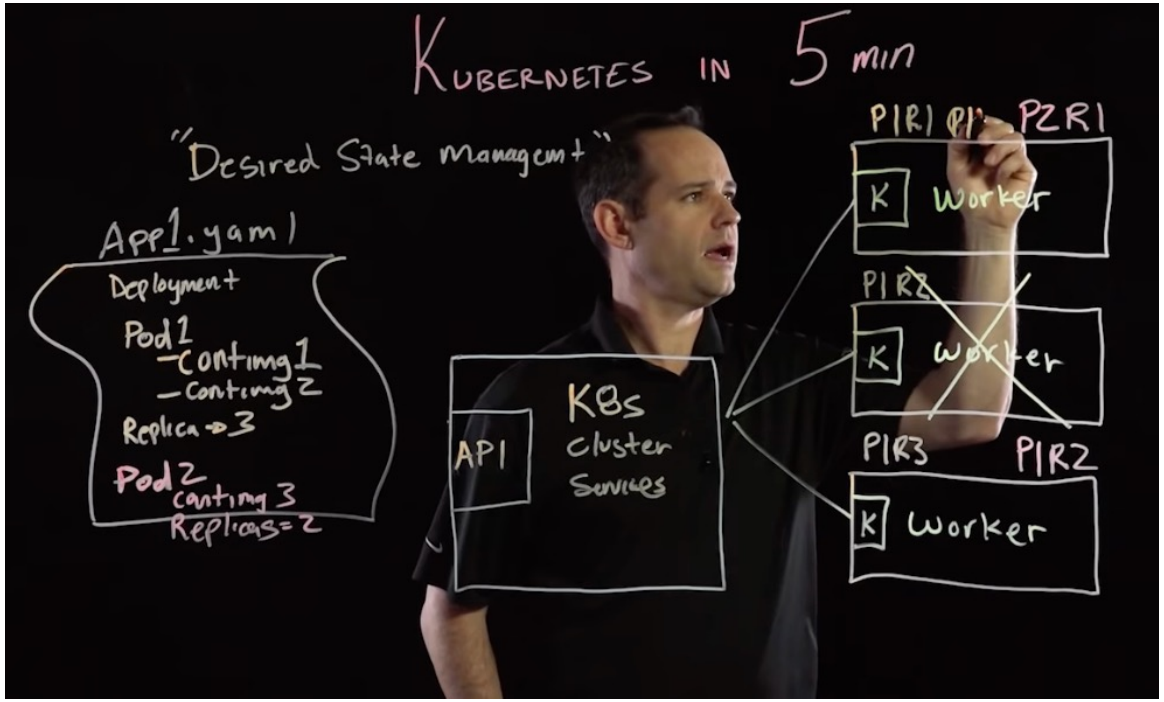
https://www.youtube.com/watch?v=u_iAXzy3xBA&t=1067s [17:47-20:55]

Why K8s?

K8s in 5 minutes

K8s in 5 minutes

https://www.youtube.com/watch?v=PH-2FfFD2PU



“Desired state management”
deployment yaml file
Pods, images, replicas

K8s cluster services, API

Worker (container host), Kubelet process

What if we loose a worker?

Why K8s?

K8s in 5 minutes

K8s design principles

K8s design principles: Declarativeness

We simply define the **desired state** of our system

K8s

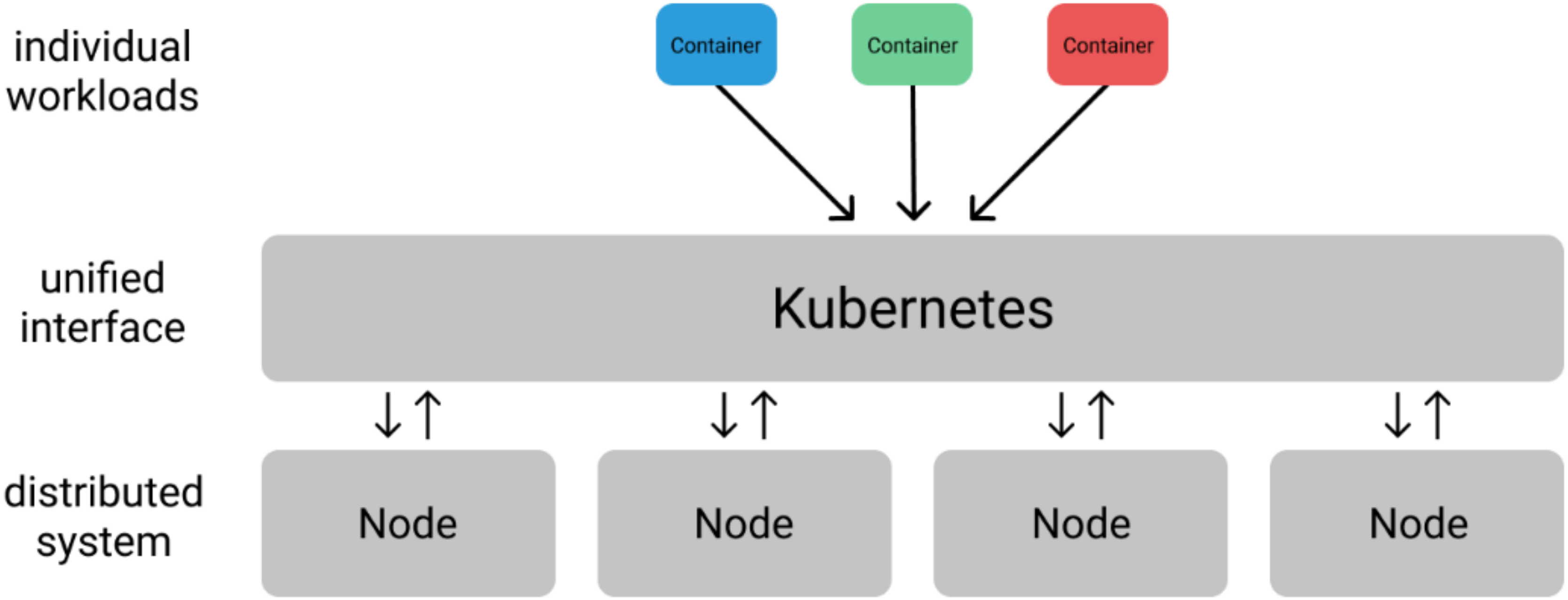
- will detect when the actual state of the system doesn't meet our expectations and
- it will intervene to fix the problem, making our system self-healing

Desired state defined by a collection of *objects*

- each object has a *specification* in which you provide the desired state and a *status* which reflects the current state of the object
- K8s constantly polls each object to ensure that its status is equal to the specification
  - if an object is unresponsive, K8s will spin up a new version to replace it
  - if a object's status has drifted from the specification, K8s will issue the necessary commands to drive that object back to its desired state

K8s design principles: Distribution

K8s provides a *unified* interface for interacting with a **cluster of machines**



We don't have to worry about communicating with each machine individually

K8s design principles: Decoupling

Containers should be developed with a *single concern* in mind

→ microservice-based architecture

K8s naturally supports the idea of **decoupled services** which can be scaled and updated independently

K8s design principles: Immutable infrastructure

To get the most from containers and container orchestration, we should be deploying **immutable infrastructure**

e.g. ~~log in to a container on a machine to update a library make changes~~ → build a new container image, deploy the new version, and terminate the older version

During the life-cycle of a project (development - testing - production) we should *use the same container image* (and only modify configurations external to the container image, e.g. by mounting a config file)

Containers are designed to be ephemeral, ready to be replaced by another container instance at any time

Maintaining immutable infrastructure makes it easier roll back applications to previous state (eg. if an error occurs) - we can simply update our configuration to use an older container image

Why K8s?

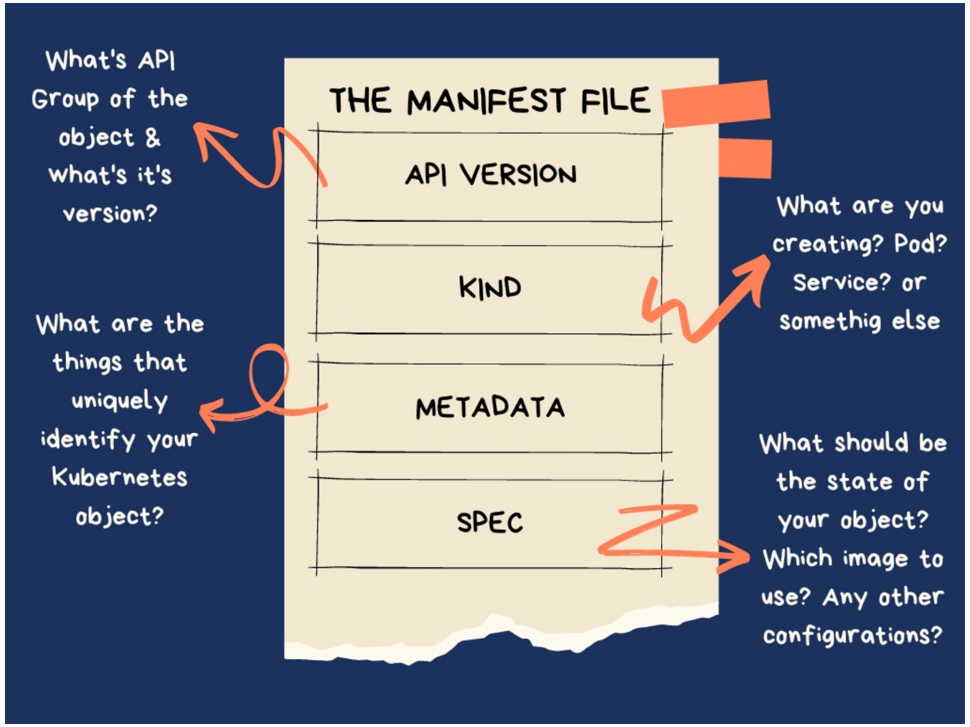
K8s in 5 minutes

K8s design principles

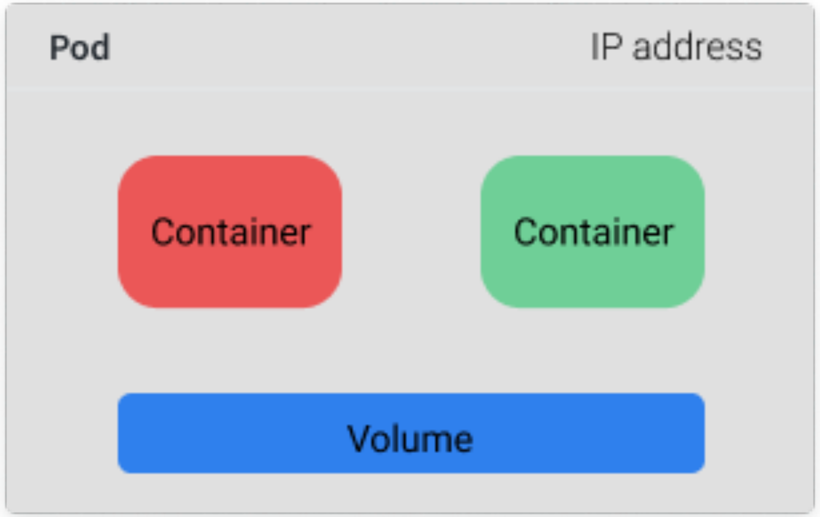
K8s objects

K8s objects

K8s objects can be defined in **manifests** (YAML or JSON files)


What's API Group of the object & what's it's version?
THE MANIFEST FILE
API VERSION
KIND
METADATA
SPEC
What are you creating? Pod? Service? or somethig else
What are the things that uniquely identify your Kubernetes object?
What should be the state of your object? Which image to use? Any other configurations?

K8s objects: Pod

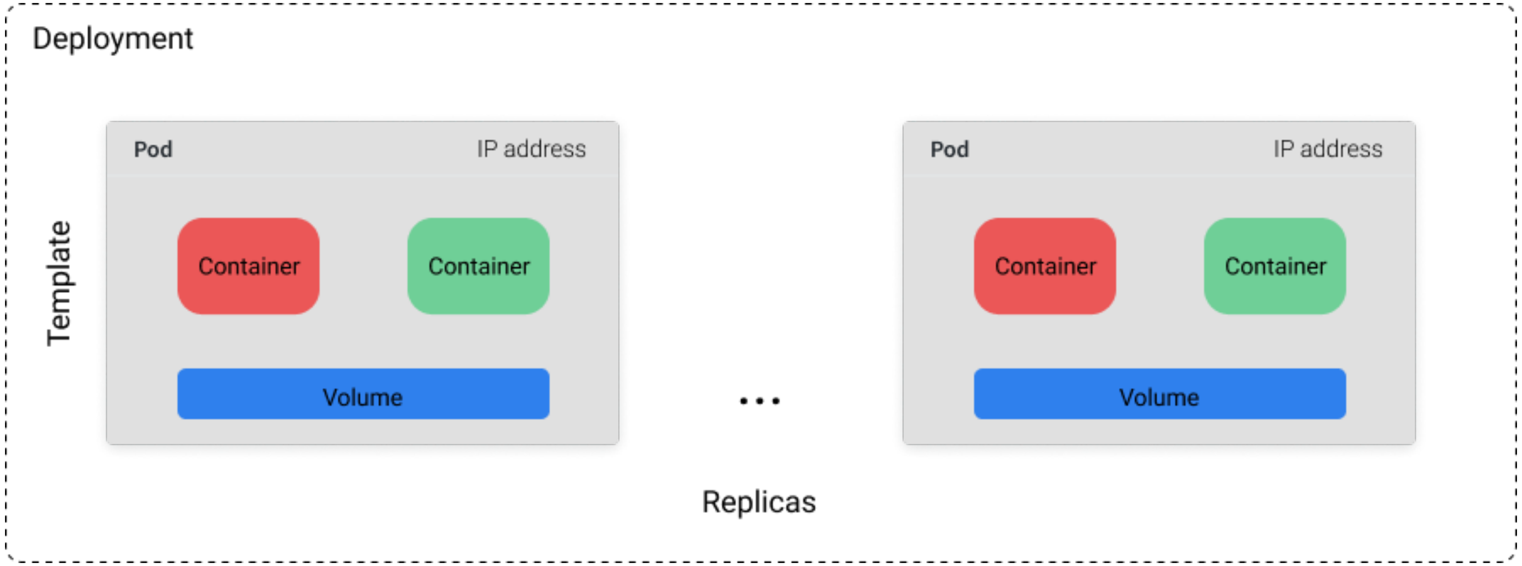


A **Pod** consists of

- one or more (tightly related) containers
- a shared networking layer
- shared filesystem volumes

K8s objects: Deployment

A **Deployment** object includes a collection of Pods defined by a template and a replica count (number n of how many copies of the template we want to run)

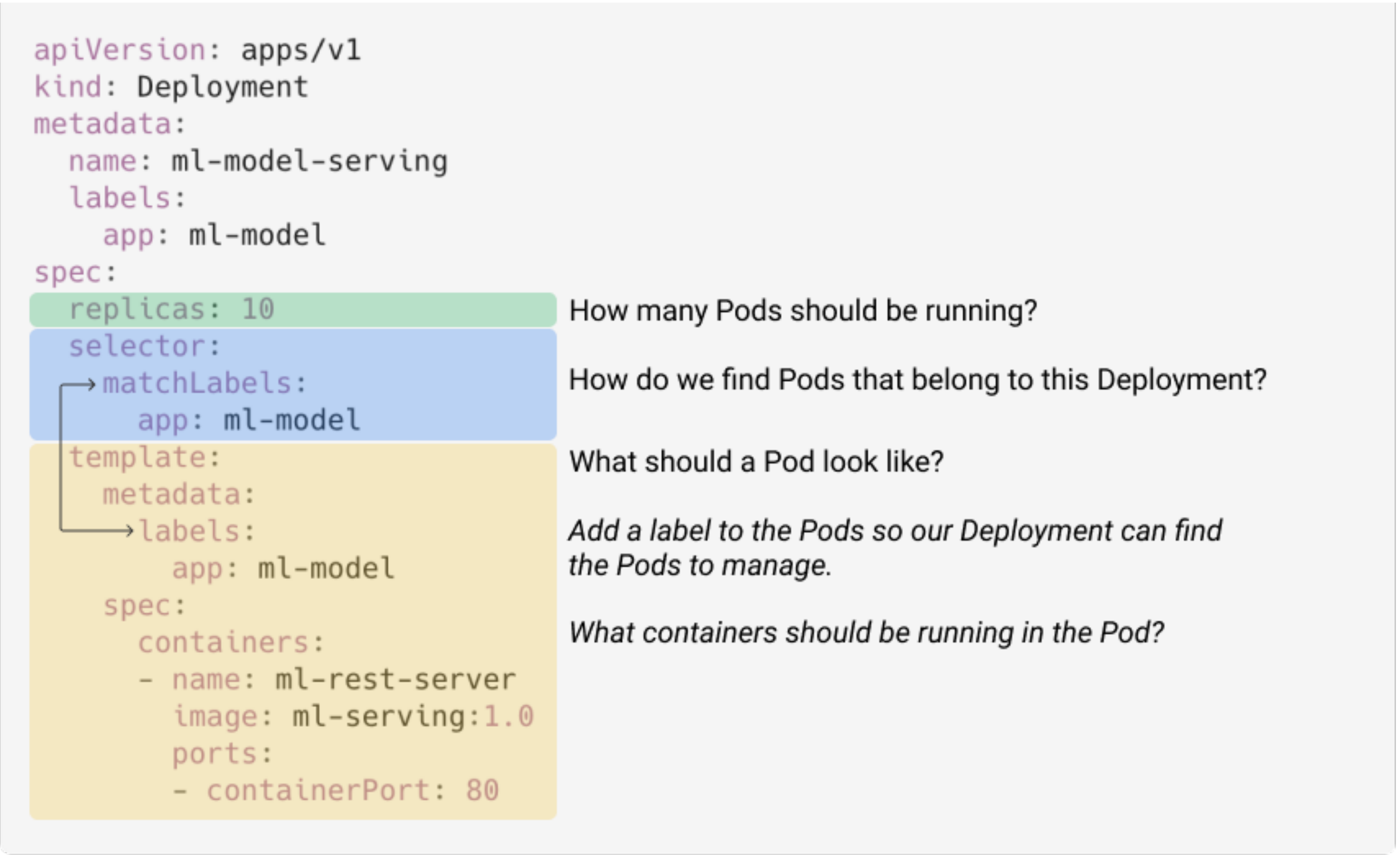


The cluster will always try to have n Pods available

- e.g. if we define a deployment with a replica count of 10 and 3 of those Pods crash, 3 more Pods will be scheduled to run on a different machine in the cluster

K8s objects: Deployment

Example: Deployment object to run 10 instances of a container which serves an ML model over a REST interface



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ml-model-serving
  labels:
    app: ml-model
spec:
  replicas: 10
  selector:
    matchLabels:
      app: ml-model
  template:
    metadata:
      labels:
        app: ml-model
    spec:
      containers:
      - name: ml-rest-server
        image: ml-serving:1.0
        ports:
        - containerPort: 80
```

How many Pods should be running?

How do we find Pods that belong to this Deployment?

What should a Pod look like?

Add a label to the Pods so our Deployment can find the Pods to manage.

What containers should be running in the Pod?

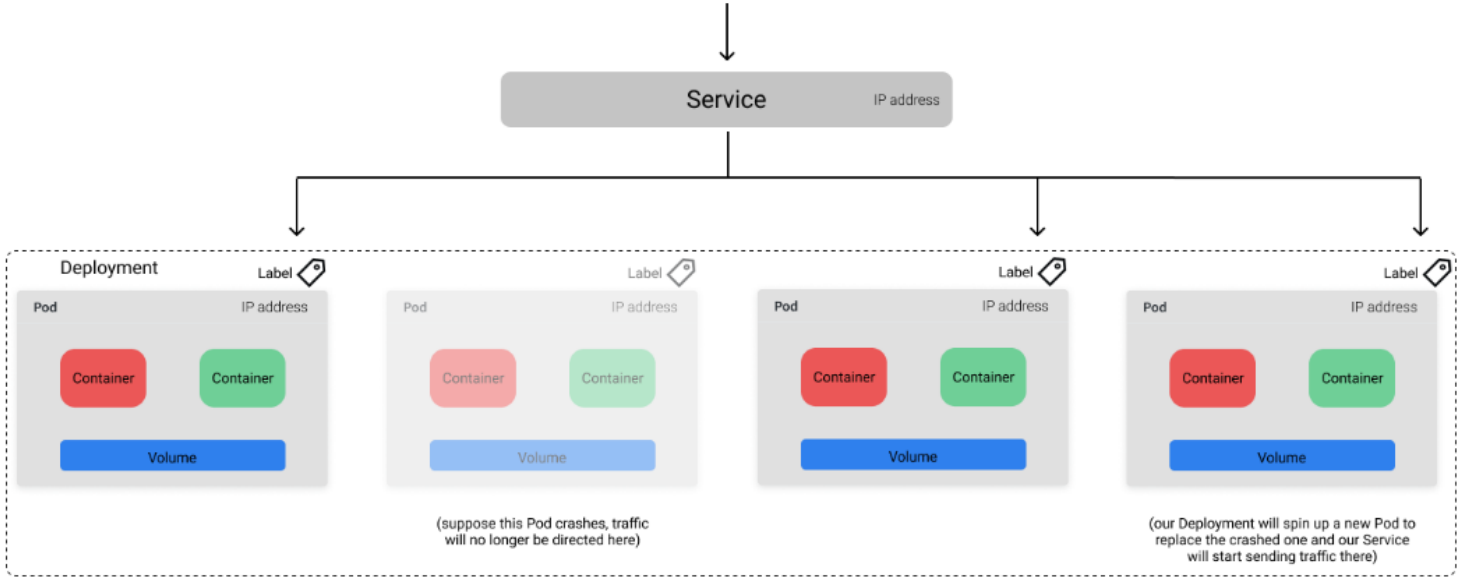
K8s objects: Service

Each Pod is assigned a unique IP address that we can use to communicate with it

Q: How to communicate with Pods if the set of Pods running as part of the Deployment can change at any time?

K8s **Service** provides a stable endpoint to direct traffic to the desired Pods even as the exact underlying Pods change due to updates/scaling/failures

Services know which Pods they should send traffic to based on *labels* (key-value pairs) which we define in the Pod metadata

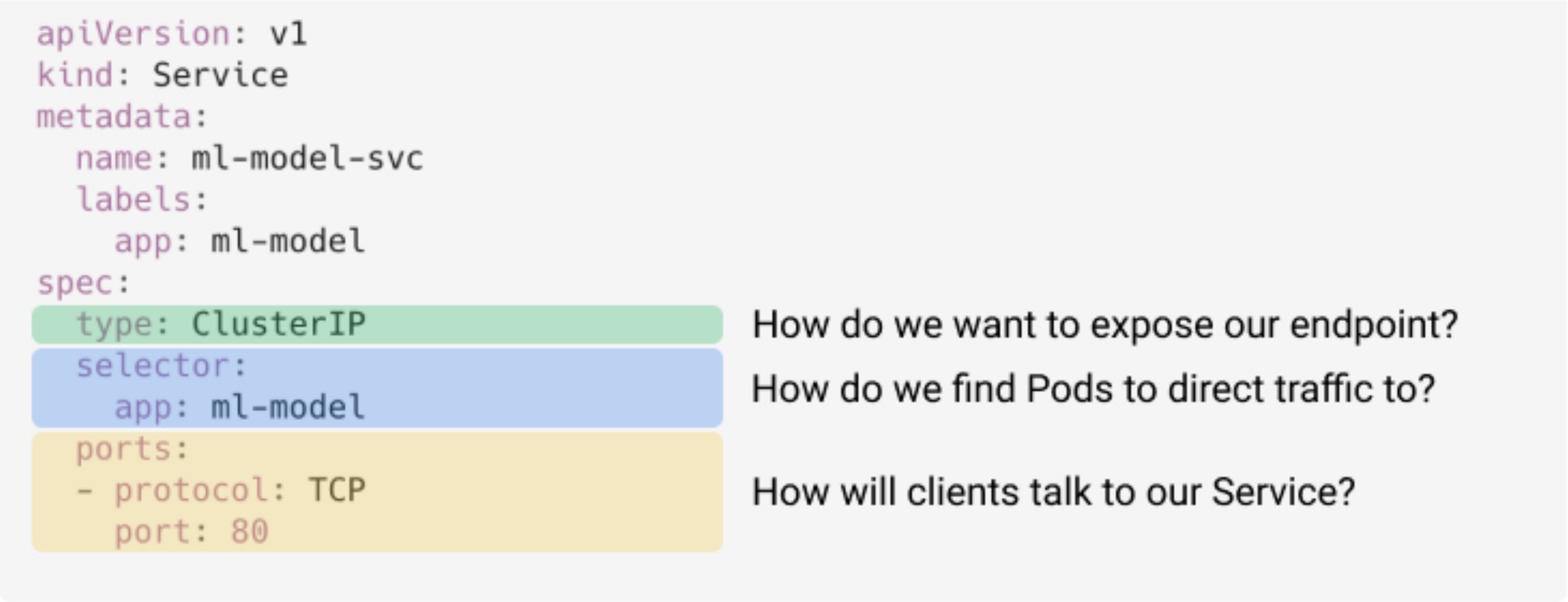


(suppose this Pod crashes, traffic will no longer be directed here)

(our Deployment will spin up a new Pod to replace the crashed one and our Service will start sending traffic there)

K8s objects: Service

Example: Service wrapped around the previous Deployment example



```yaml
apiVersion: v1
kind: Service
metadata:
  name: ml-model-svc
  labels:
    app: ml-model
spec:
  type: ClusterIP
  selector:
    app: ml-model
  ports:
  - protocol: TCP
    port: 80
```

How do we want to expose our endpoint?

How do we find Pods to direct traffic to?

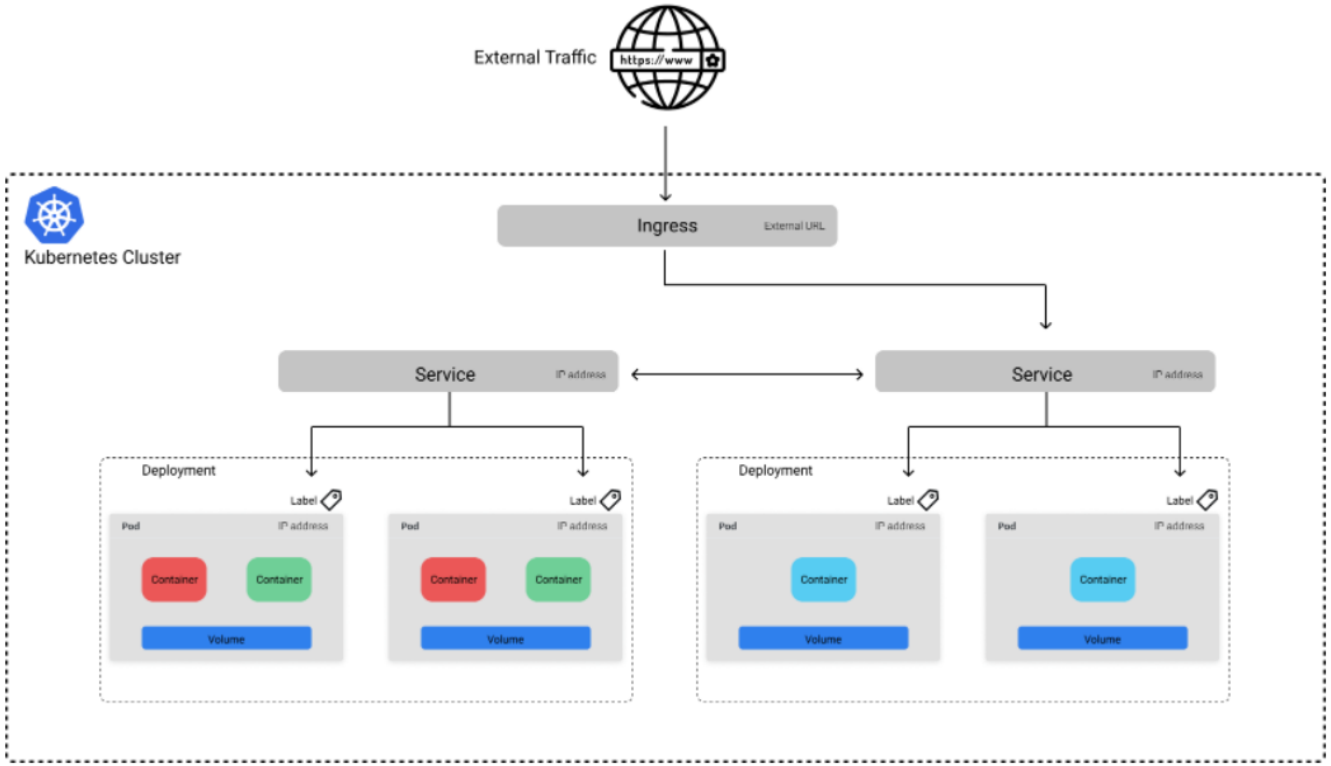
How will clients talk to our Service?

K8s objects: Ingress

Service allows us to expose applications behind a stable endpoint - only available to internal cluster traffic

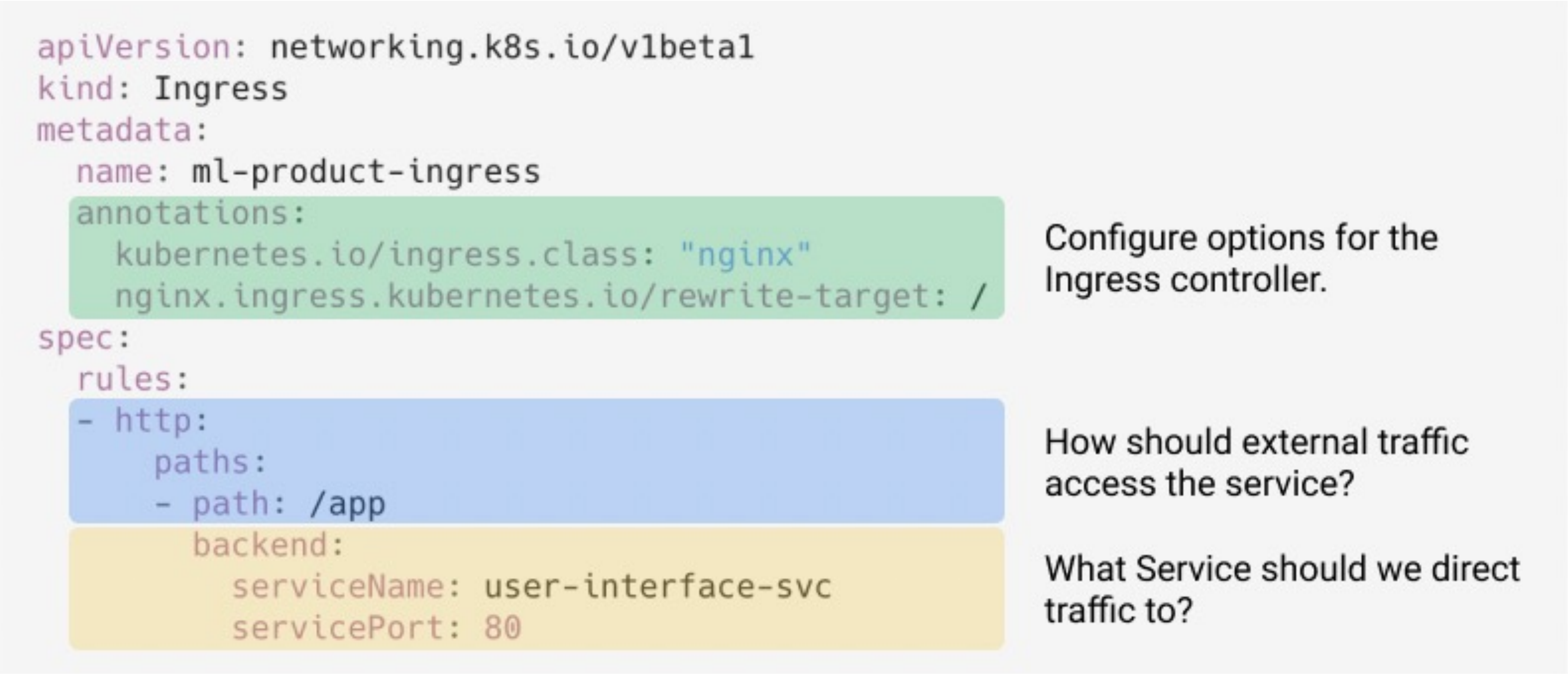
To to expose our application to traffic external to our cluster, we need to define an **Ingress** object

We can select which Services to make publicly available



K8s objects: Ingress

Example



```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ml-product-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /app
        backend:
          serviceName: user-interface-svc
          servicePort: 80
```

Configure options for the Ingress controller.

How should external traffic access the service?

What Service should we direct traffic to?

K8s objects

Many more:

- Job
- Volume
- Secret
- Namespace
- ConfigMap
- HorizontalPodAutoscaler
- StatefulSet
- …

Why K8s?

K8s in 5 minutes

K8s design principles

K8s objects

K8s control plane

K8s control plane

Two types of machines in a cluster:

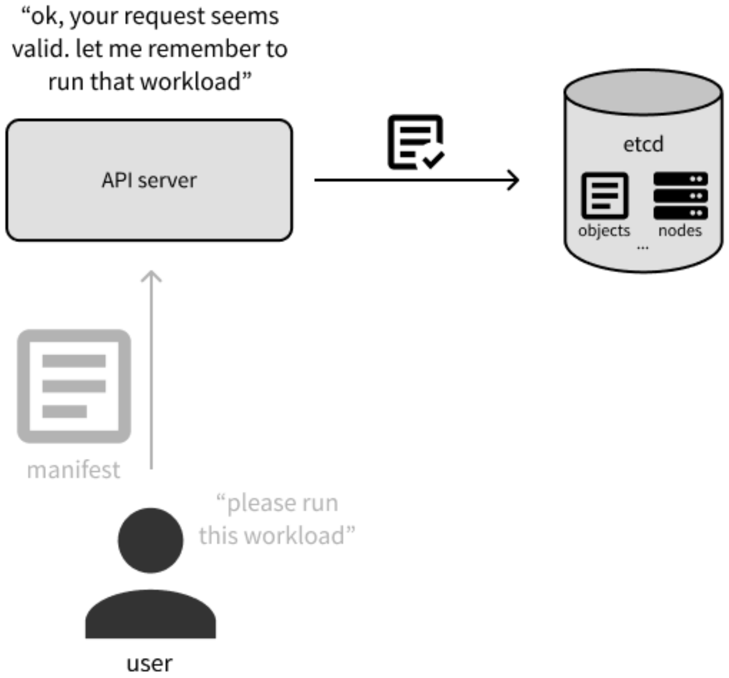
- **master node** - (often single) machine that contains most of the control plane components
- **worker node** - machine that runs the application workloads

K8s control plane (master node)

User provides new/updated object specification to **API server** of master node

- API server validates update requests and acts as unified interface for questions about cluster's current state
- State of cluster* is stored in a distributed key-value store **etcd**, a distributed key-value store

(*cluster configuration, object specifications, object statuses, nodes on the cluster, object-node assignments etc.)

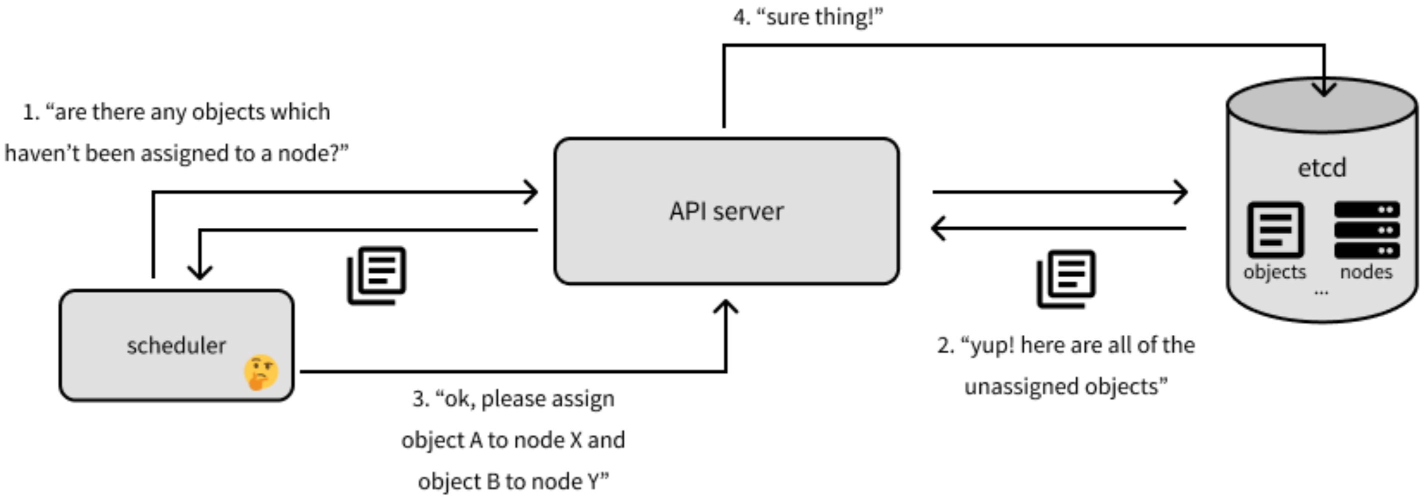

"ok, your request seems valid. let me remember to run that workload"
API server
etcd
objects
nodes
...
manifest
"please run this workload"
user

K8s control plane (master node)

The **scheduler** determines where objects should be run

The scheduler

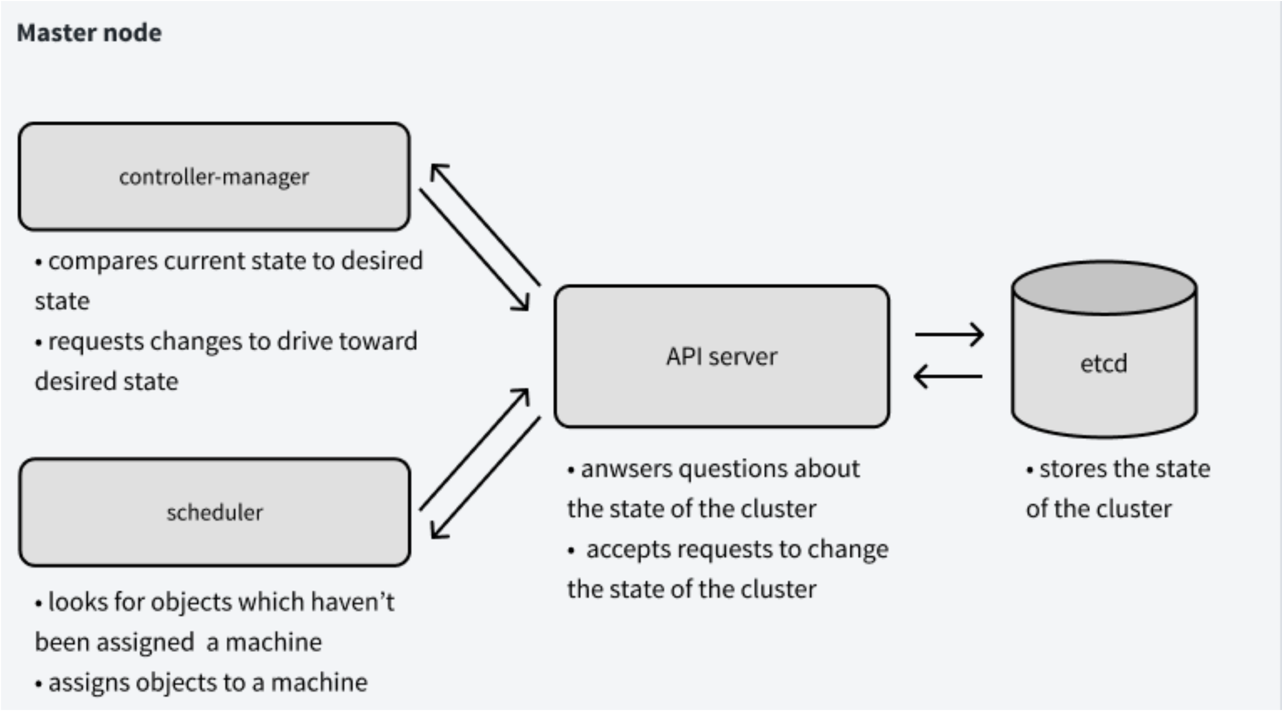
- asks the API server which objects haven't been assigned to a machine
- determines which machines those objects should be assigned to
- replies back to the API server to reflect this assignment


4. "sure thing!"
1. "are there any objects which haven't been assigned to a node?"
API server
etcd
objects
nodes
...
scheduler
2. "yup! here are all of the unassigned objects"
3. "ok, please assign object A to node X and object B to node Y"

K8s control plane (master node)

The **controller-manager** monitors cluster state through the API server

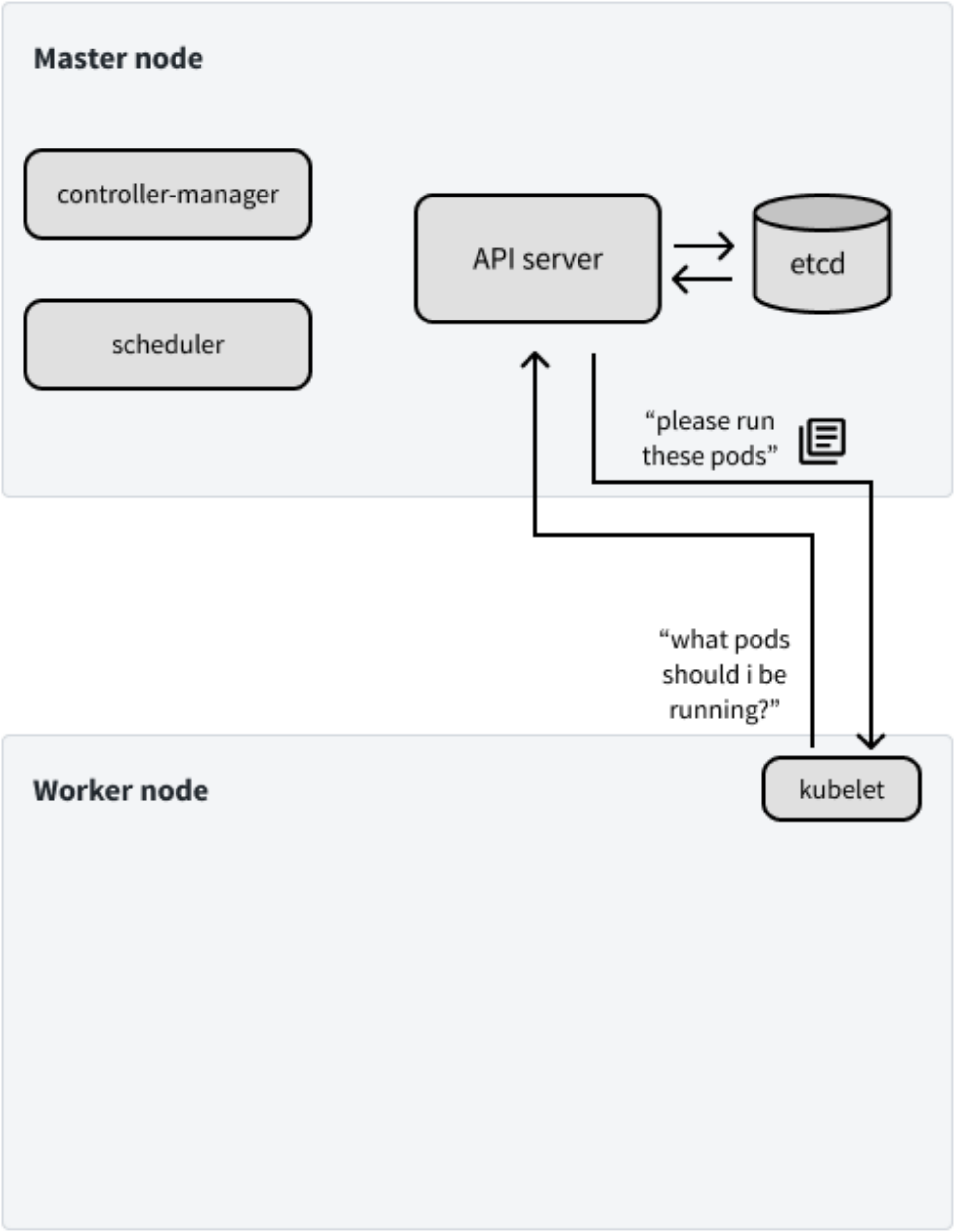
If actual state differs from desired state, the controller-manager will make changes via the API server to drive the cluster towards the desired state


Master node
controller-manager
• compares current state to desired state
• requests changes to drive toward desired state
scheduler
• looks for objects which haven't been assigned a machine
• assigns objects to a machine
API server
• anwsers questions about the state of the cluster
• accepts requests to change the state of the cluster
etcd
• stores the state of the cluster

K8s control plane (worker node)

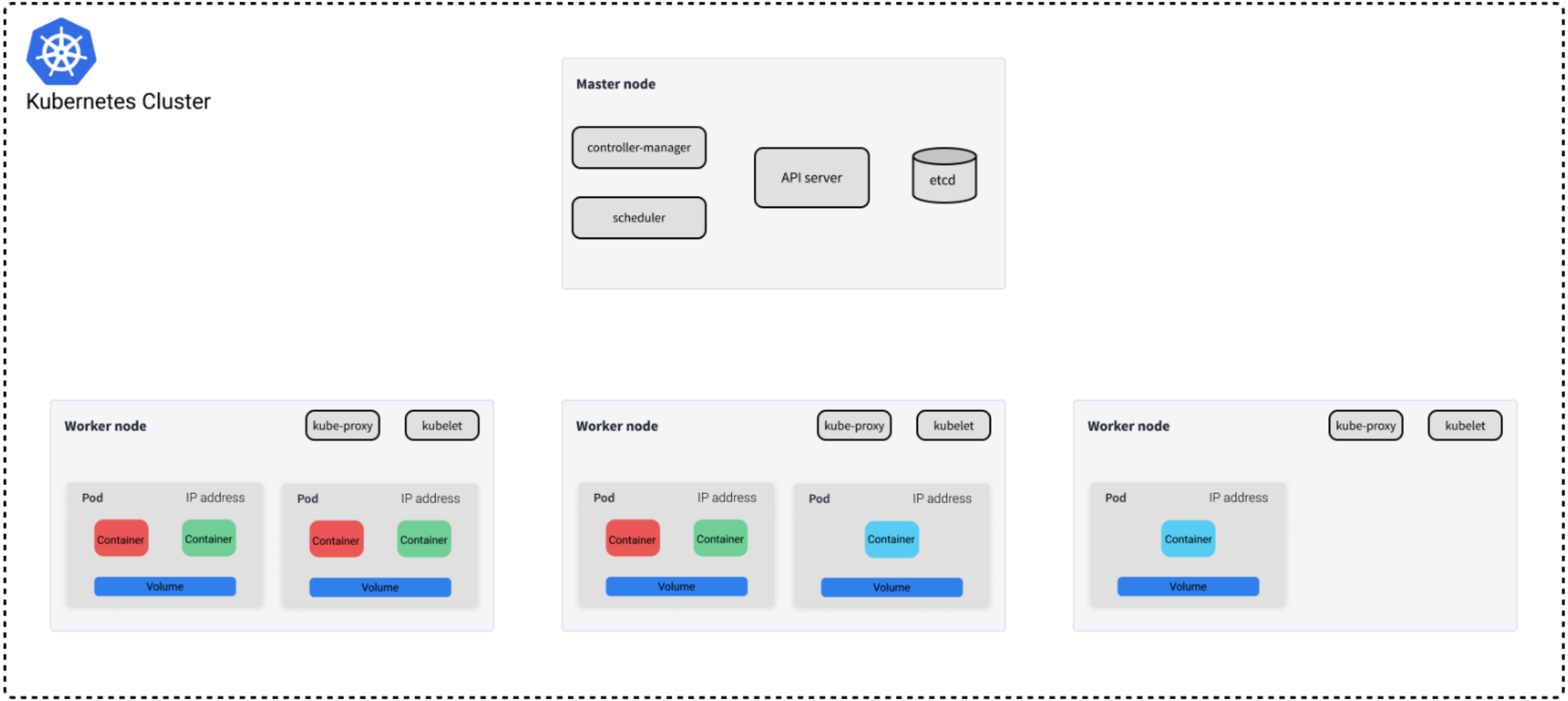
kubelet

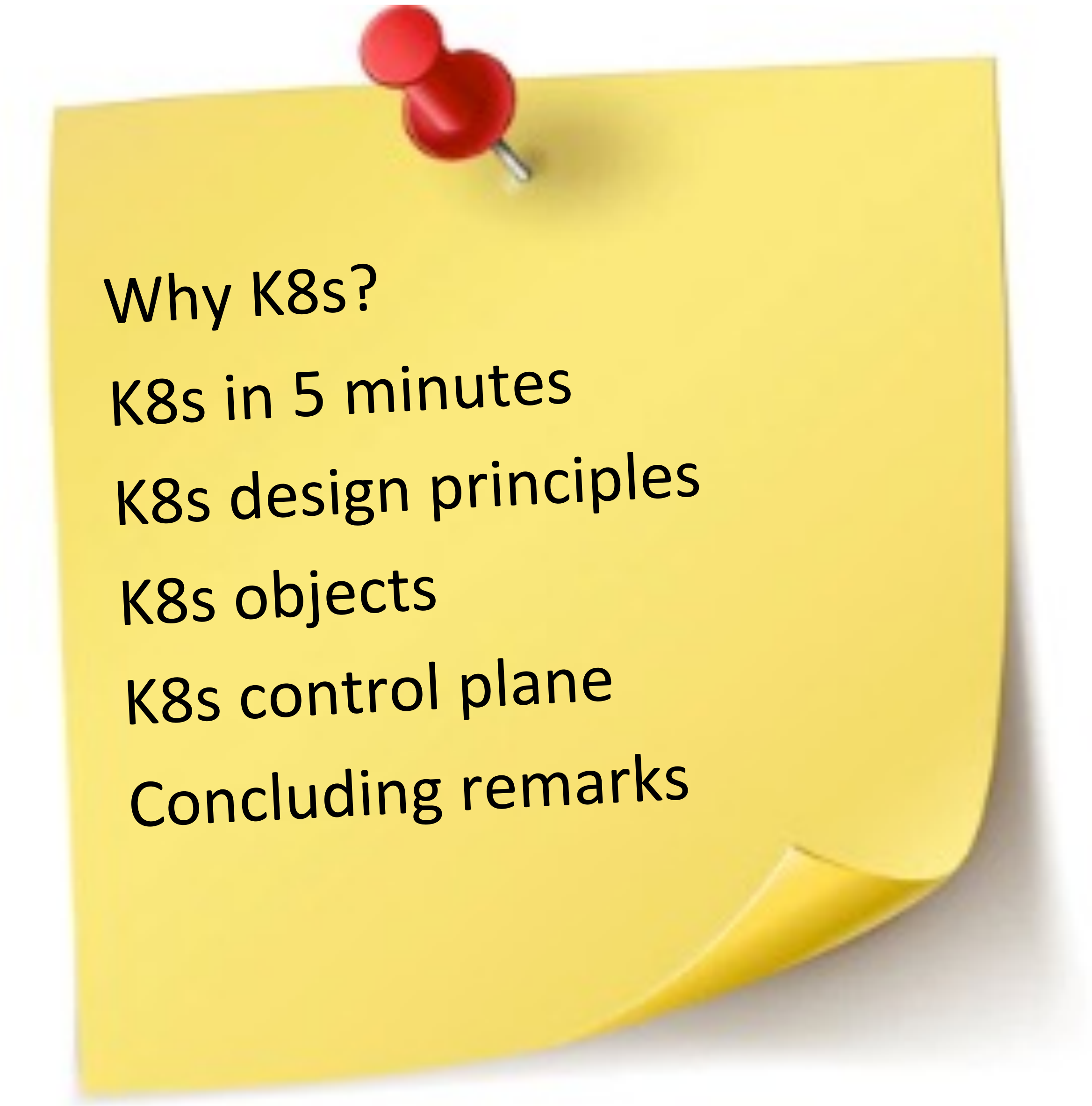
- acts as a node's "agent" which communicates with the API server to see which container workloads have been assigned to the node
- responsible for spinning up pods to run these assigned workloads
- when a node first joins the cluster, kubelet announces the node's existence to the API server so the scheduler can assign pods to it



K8s control plane (worker node)

kube-proxy enables containers to communicate with each other across the various nodes on the cluster





Why K8s?

K8s in 5 minutes

K8s design principles

K8s objects

K8s control plane

Concluding remarks

Concluding remarks

When should you NOT use K8s?

- If you can run your workload on a single machine
- If your compute needs are light
- If you don't need high availability and can tolerate downtime
- If you don't envision making a lot of changes to your deployed services
- If you have a monolith and don't plan to break it into microservices

K8s vs. Docker Swarm

Docker Swarm	Kubernetes
▪ Simpler to install ▪ Softer learning curve	▪ Features auto-scaling ▪ Higher fault tolerance ▪ Huge community ▪ Backed by Cloud Native Computing Foundation (CNCF)
Preferred in environments where simplicity and fast development is favored	Preferred for environments where medium to large clusters are running complex applications