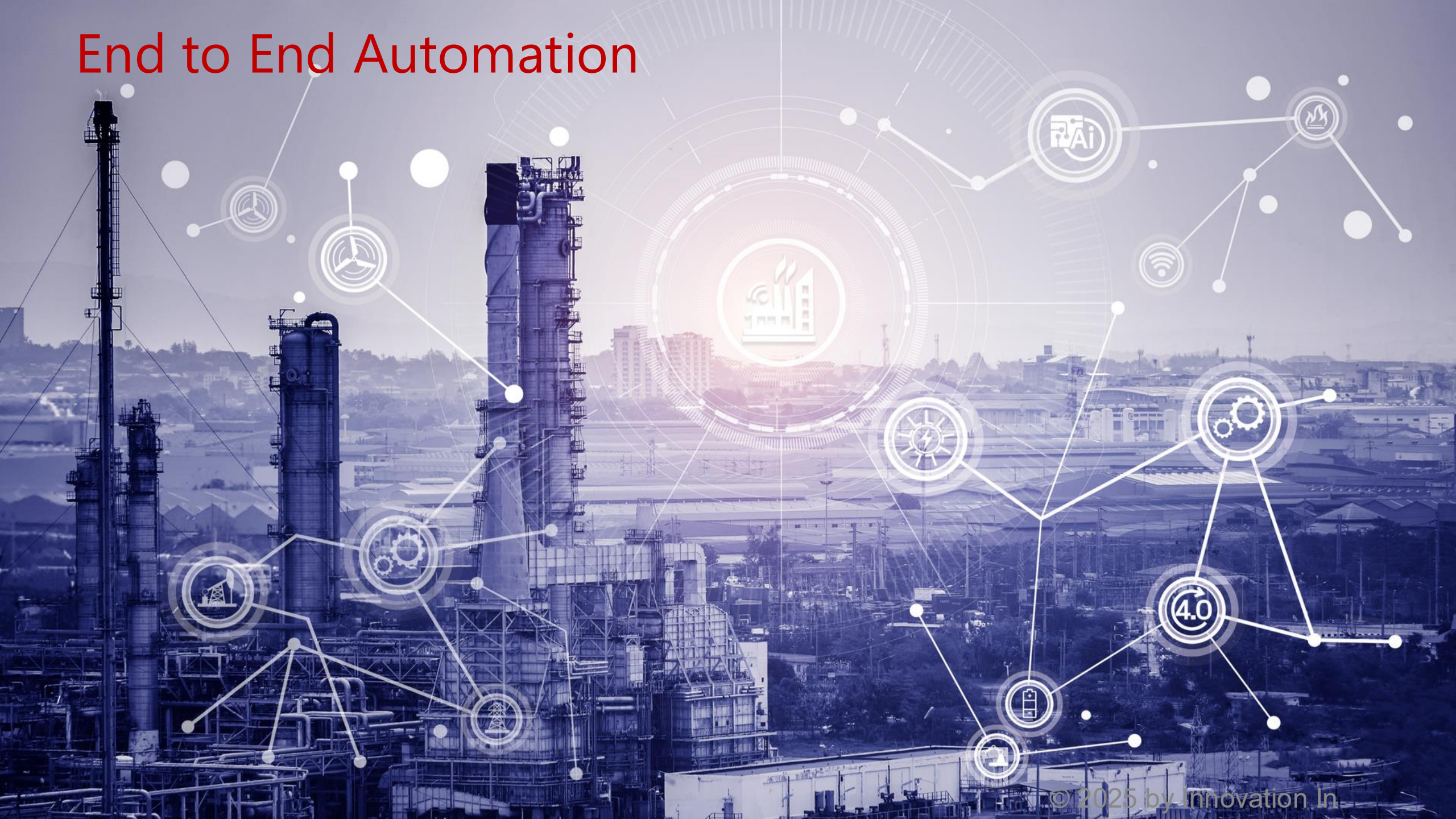
End to End Automation



TEKsystems Global Services

WORKFORCE DEVELOPMENT

Content Usage Parameters

Content refers to material including instructor guides, student guides, lab guides, lab or hands-on activities, computer programs, etc. designed for use in a training program

1

Content is subject to copyright protection

2

Content may only be leveraged by students enrolled in the training program

3

Students agree not to reproduce, make derivative works of, distribute, publicly perform and publicly display in any form or medium outside of the training program

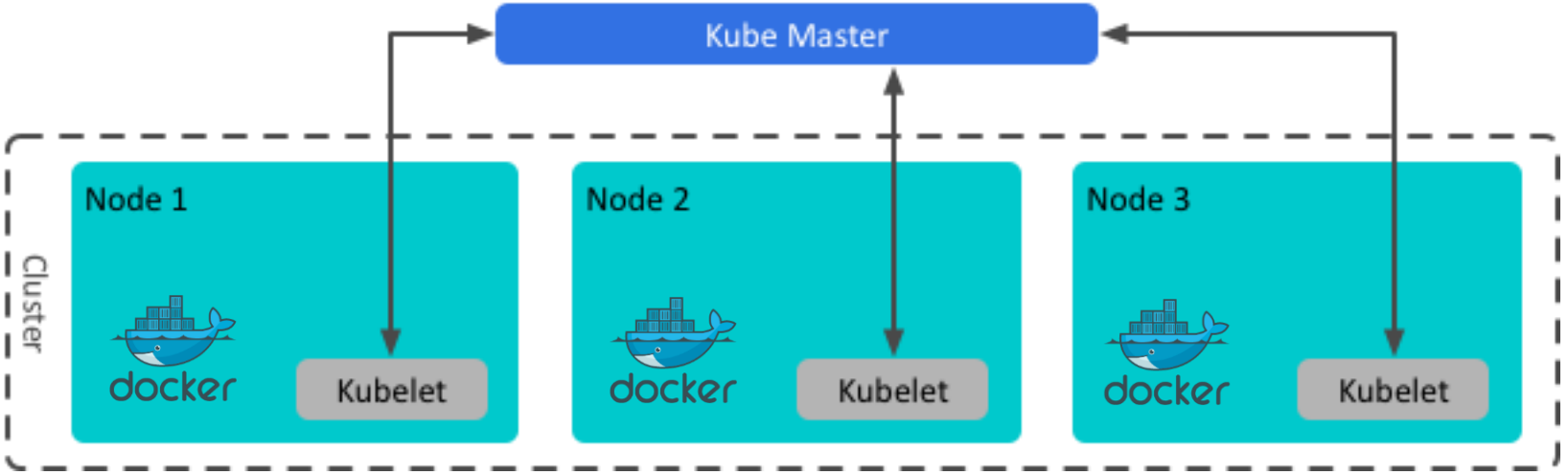
4

Content is intended as reference material only to supplement the instructor-led training

Kubernetes – Containers at scale

Kubernetes: Containers at Scale

- *Kubernetes provides the infrastructure for container-centric deployment and operation of applications*



- Request load balancing
- Application health checking
- Log access and resource monitoring

History of Kubernetes

- Google adopted containers as an application deployment standard over a decade ago
  - Contributed cgroups to Linux kernel in 2007
- Google developed generations of container management systems, scaling to thousands of hosts per cluster
  - First was Borg, treated as a trade secret until 2015*
  - Omega built on concepts of Borg, also Google-internal
  - Kubernetes inspired by observed needs of Google Cloud Platform customers, open source

Star Trek Borg Cube

A hegemonizing swarm

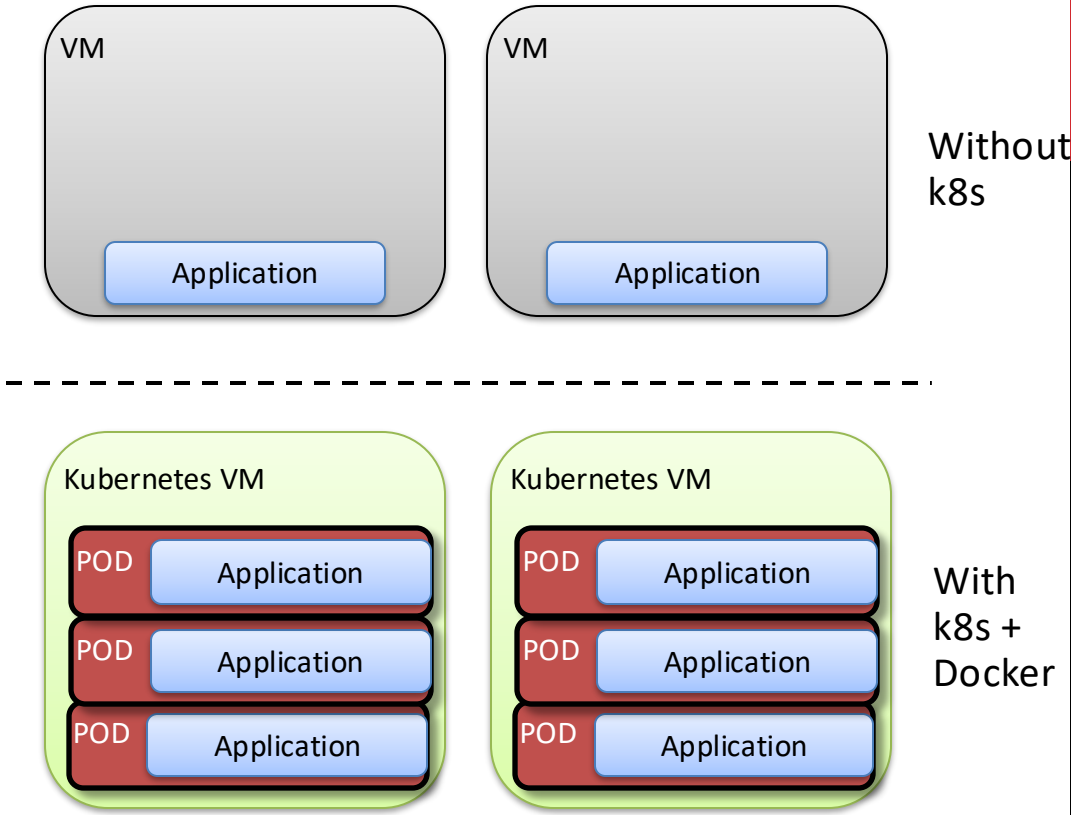
Original project name:

Seven of Nine

(the friendly Borg)

Kubernetes: Google Cloud Platform

- After launch of Google Compute Engine, utilization of VM's by customers was observed to be very low
  - Running apps in whole VM's leading to stranded resources
- GCE itself already running on Borg, which solved utilization through efficient scheduling of containerized applications
- Google engineers pushed to found Kubernetes, as open source project
  - K8s available to GCE customers
  - Not tied to Google or GCP infrastructure

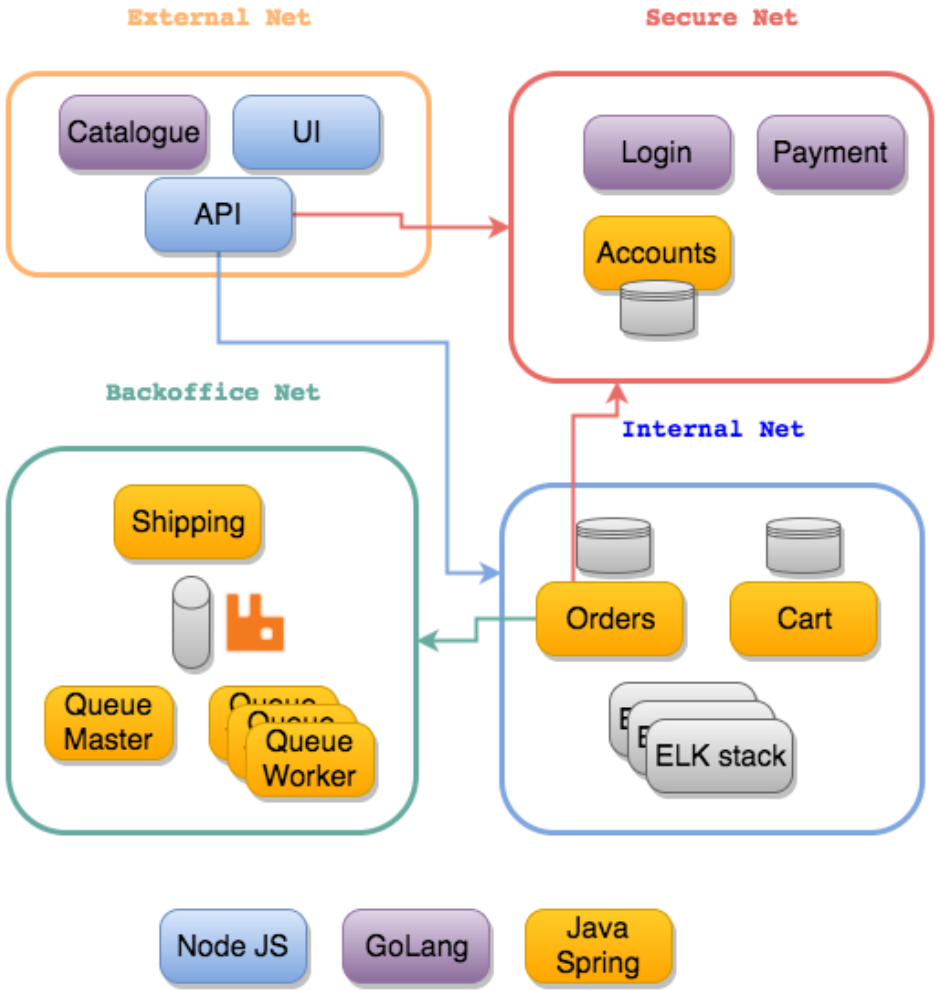


Kubernetes Use Cases

Kubernetes: Cloud-Native Application Deployment

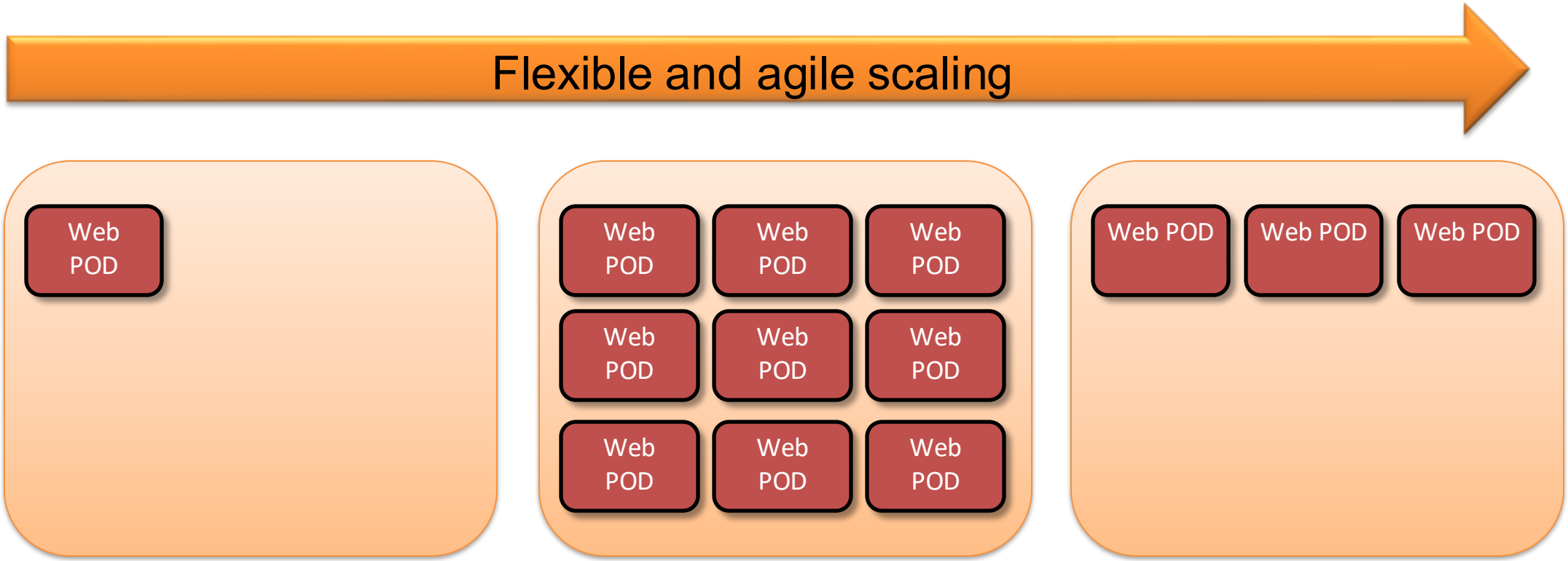
Cloud-native applications, aka microservice-based or 12-factor apps

- Cloud-native applications are composed of small, independently-deployable, loosely-coupled services
- Kubernetes makes it easy to deploy, update, and coordinate operations between multiple containerized service components
- Kubernetes project actively enhancing features to better support and manage stateful applications



Kubernetes: Elastic Services

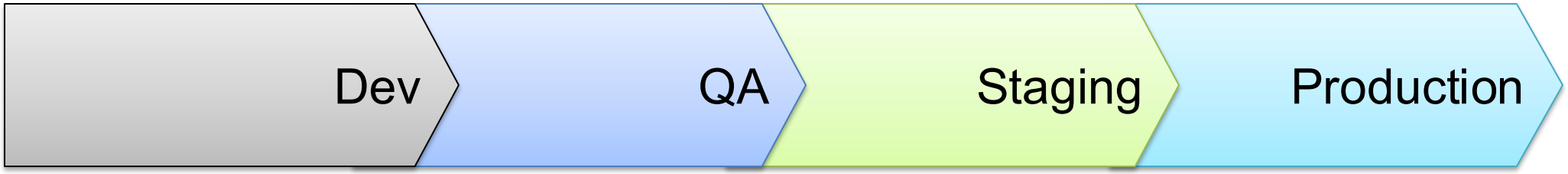
Kubernetes supports manual and automated scaling of application services based on demand for resources



Kubernetes: CI/CD Pipelines

Managing execution environments in a CI/CD pipeline

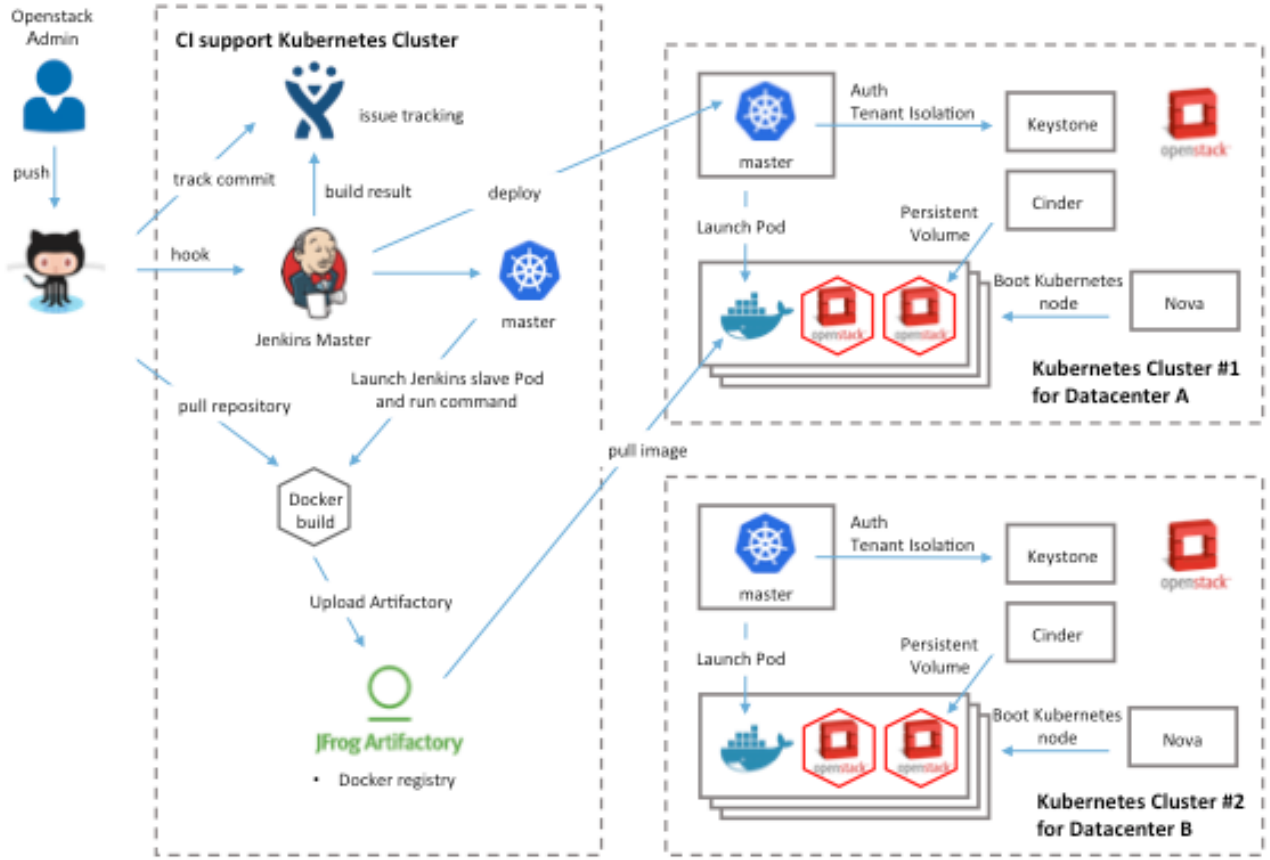
- Automated systems can push to logical environments in the same cluster, or to different clusters, using the same control plane API
- Kubernetes labels and annotations allow users to organize resources into separate environments without changing code or functionality
- Label selectors target specific sets of resources for control and exposure



Example: Kubernetes CI in Production at Scale

YAHOO! JAPAN

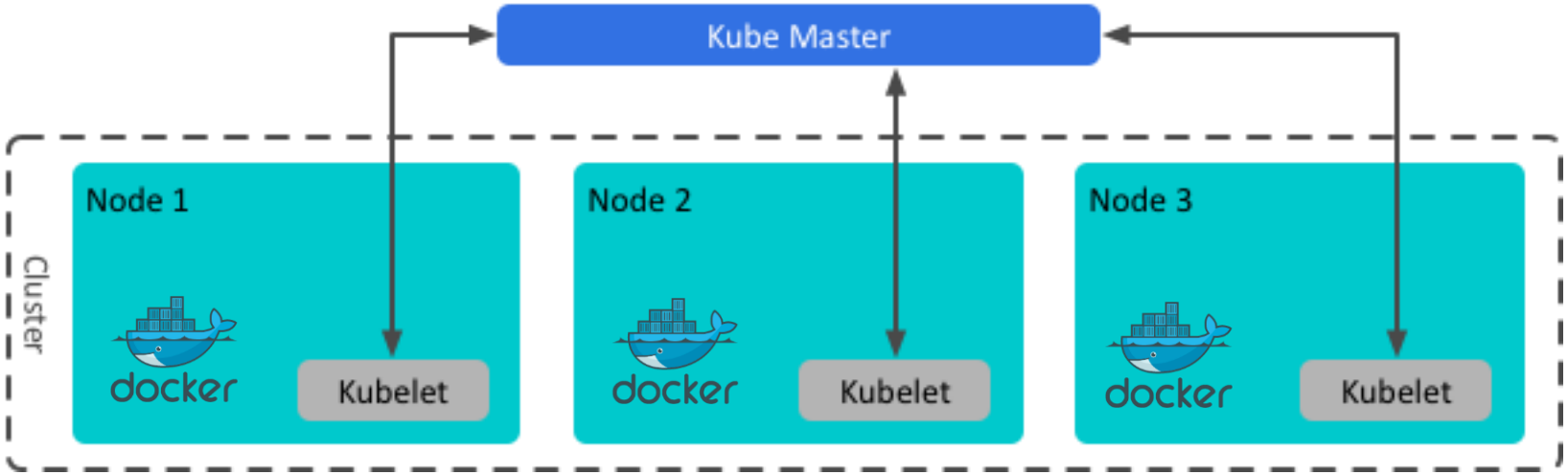
- Built automated system to build and deploy containerized applications on Kubernetes
- Kubernetes clusters built from VM's in private OpenStack environments
  - 1,000 nodes
  - 42,000 Containers
  - ~2,500 Applications



Overview of Kubernetes Clusters

Kubernetes: Cluster

- Kubernetes deployed on a set of physical or virtual hosts – K8s nodes
- Hosts run host OS that supports Linux containers, e.g. Docker or rkt hosts
- Kubernetes runs well in both private and public IaaS environments
- Users and admins control Kubernetes resources via REST API on K8s master



Kubernetes Components: Control Plane

Main Components:

Control Plane/Leader nodes:

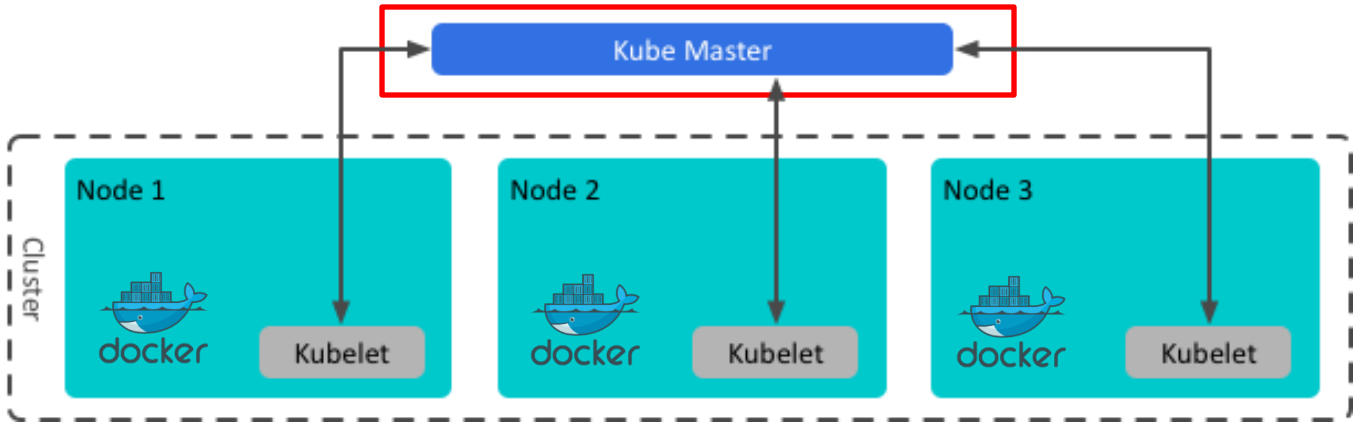
- This manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

Worker Nodes:

- Run user workloads as directed by the K8s master
- The leader nodes may also serve as a worker nodes

Clusters:

- A collection of nodes bound to a Leader and managed as a single logical unit of capacity



Kubernetes Components: Nodes

Main Components:

Control Plane/Leader nodes:

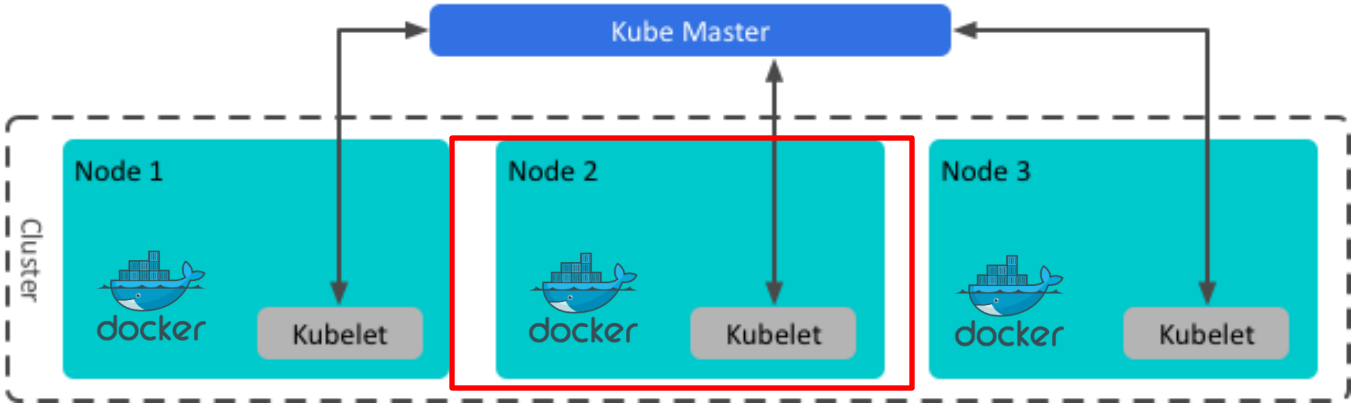
- This manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

Worker Nodes:

- Run user workloads as directed by the control plane
- The leader nodes may also serve as worker nodes

Clusters:

- A collection of nodes bound to a Leader and managed as a single logical unit of capacity



Kubernetes Cluster Components

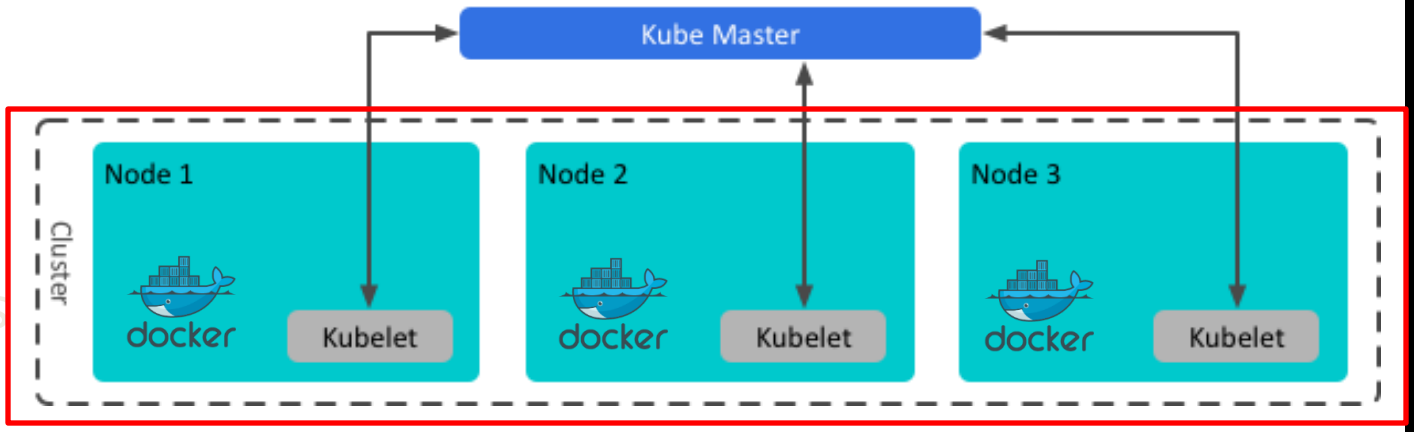
Main Components:

Control Plane/ Leader nodes:

- This manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

Worker Nodes:

- Run user workloads as directed by the control plane.
- The leader nodes may also serve as worker nodes



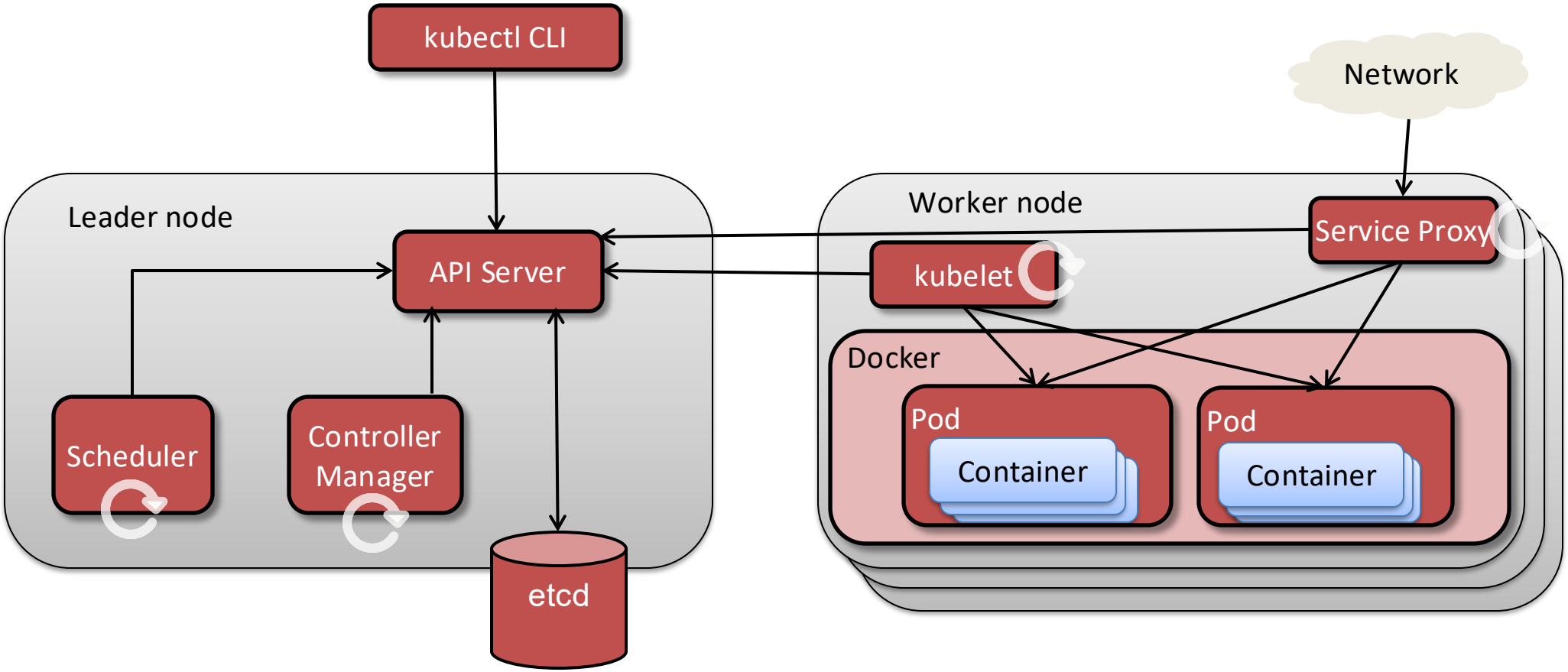
Clusters:

- A collection of nodes bound to a Leader and managed as a single logical unit of capacity

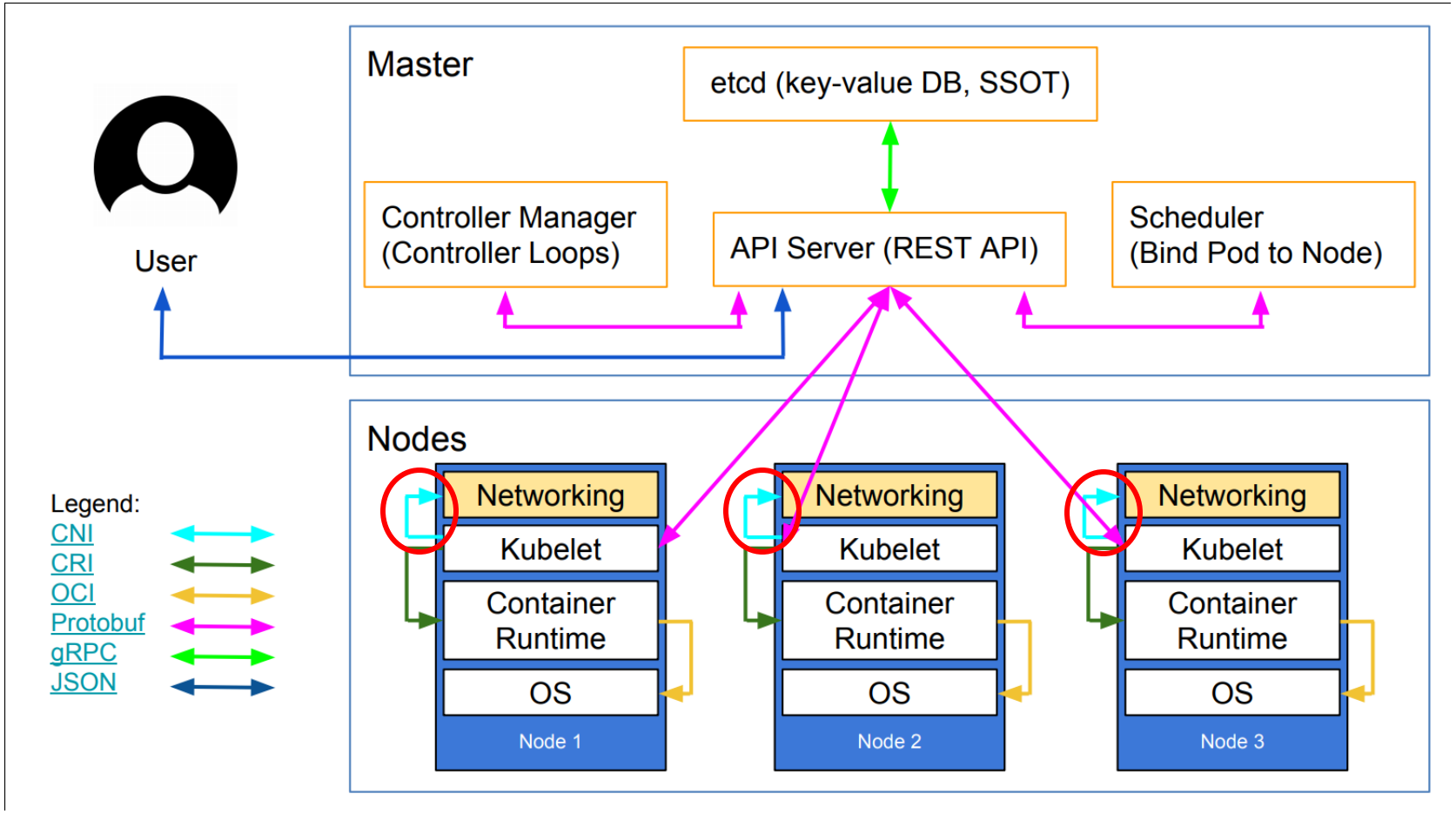
Kubernetes Architecture

Kubernetes Cluster Architecture

- Kubernetes nodes can be physical hosts or VM's running a container-friendly Linux kernel

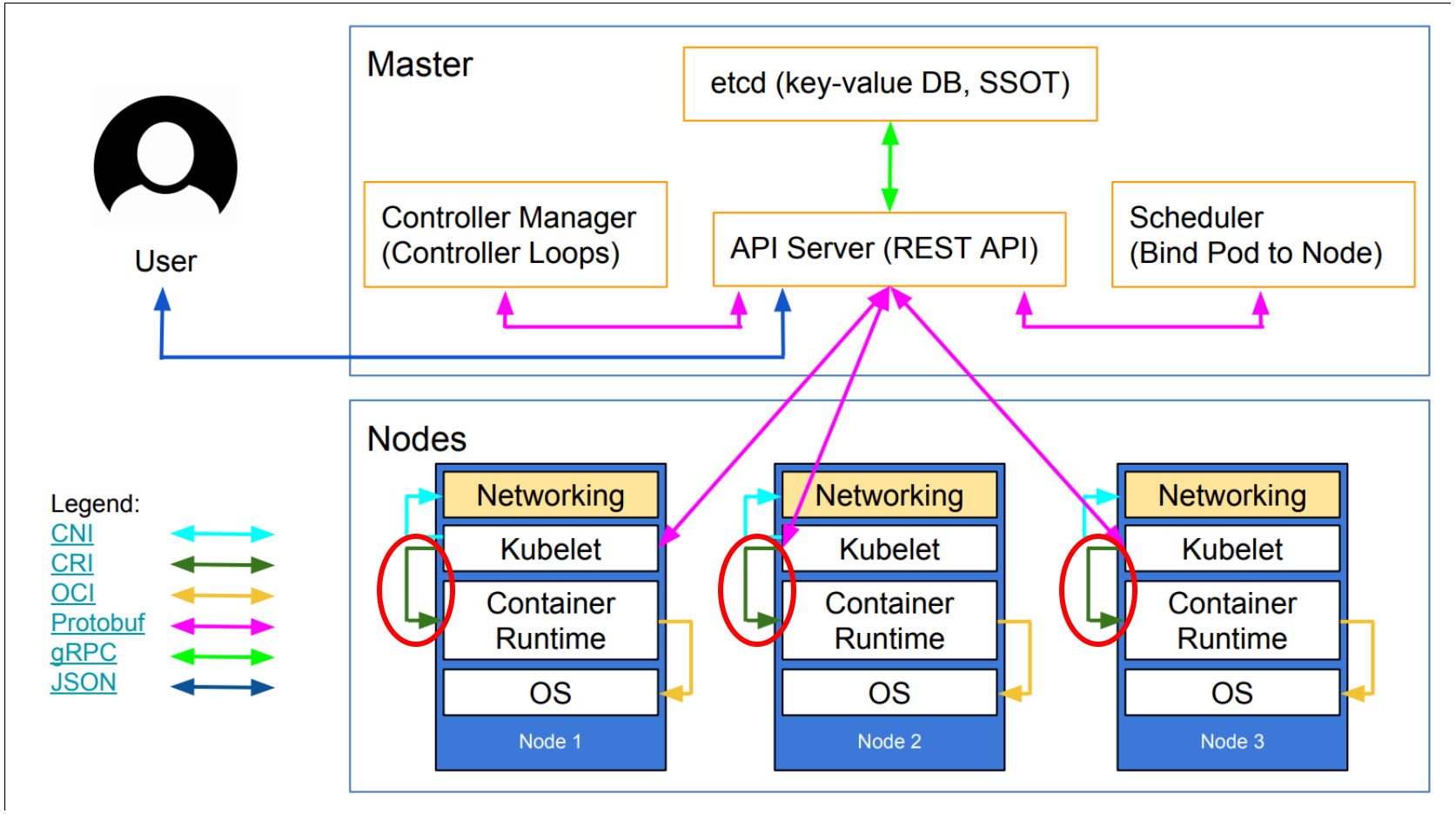


Components Communication



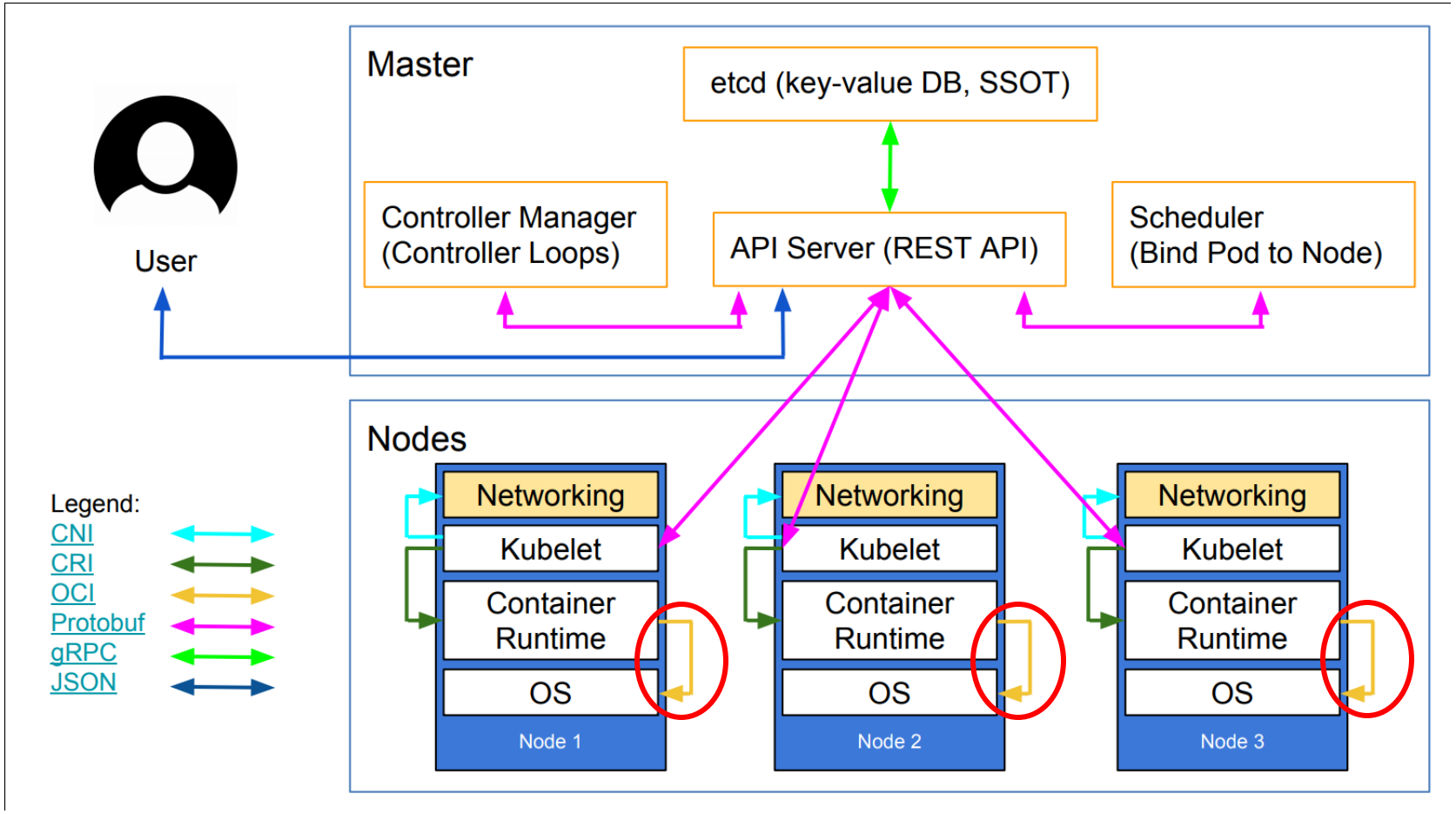
- CNI = Container Network Interface

Components Communication



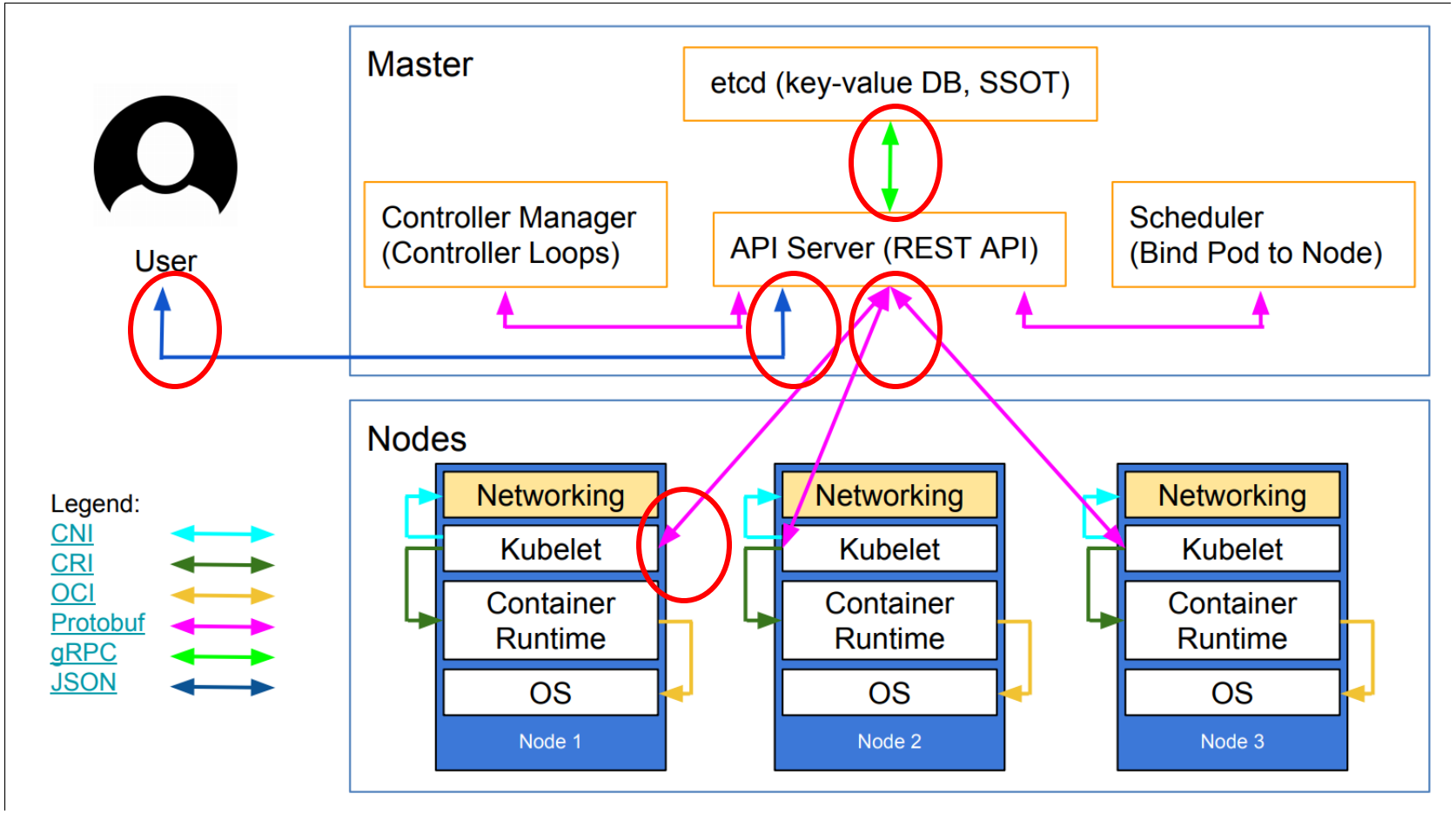
- CNI = Container Network Interface
- CRI = Container Runtime Interface
  - Docker, CRI-O, Containerd

Components Communication



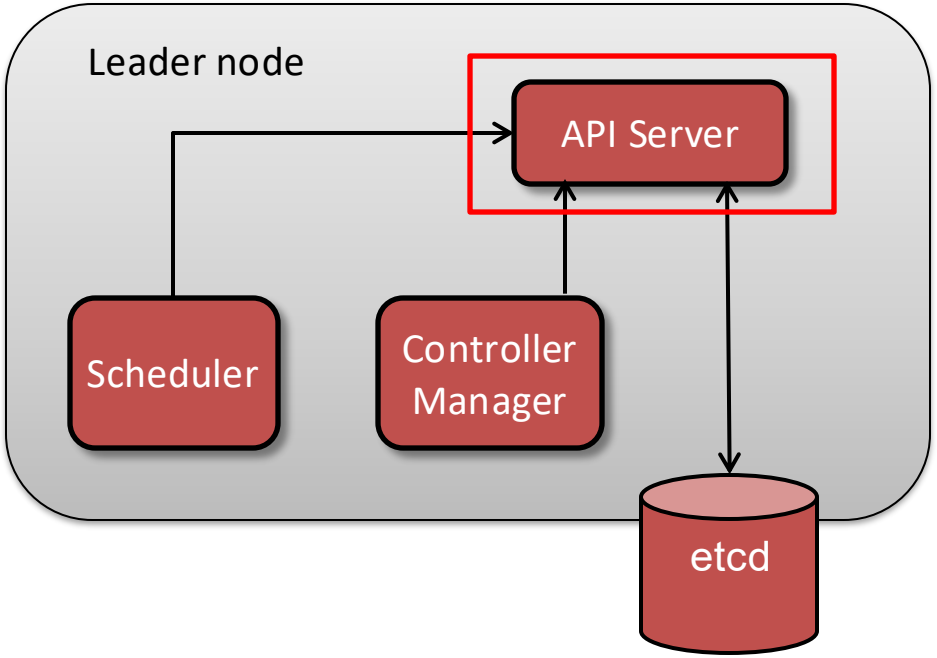
- CNI = Container Network Interface
- CRI = Container Runtime Interface
  - Docker, CRI-O, Containerd
- OCI = Open Container Initiative

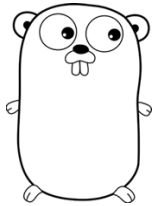
Components Communication



- CNI = Container Network Interface
- CRI = Container Runtime Interface
  - Docker, CRI-O, Containerd
- OCI = Open Container Initiative
- Protobuf
- gRPC
- JSON

Kubernetes Control Plane/Leader Components





K8s components written in Go (golang.org)

- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared object states, e.g. Node Controller, Replication Controller, …
- **Persistent data store (etcd)**: all K8s system data is stored in a distributed, reliable key-value store. etcd may run on separate nodes from the master

Kubernetes Objects and Resources

Kubernetes API Objects and Resources

- **Objects** are the persistent entities that users manage via the Kubernetes API
  - Objects track what's running and where, available system resources, and behavioral policies, e.g.

Workloads

- Pod (run)
- Service (expose)

Configuration

- Secret
- ConfigMap

Controllers

- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet
- Jobs / Cron Jobs

Workload Persistent Storage

- PersistentVolume
- PersistentVolumeClaim

Cluster Resources

- Node
- Namespace
- Cluster

Network Resources

- Ingress
- NetworkPolicy

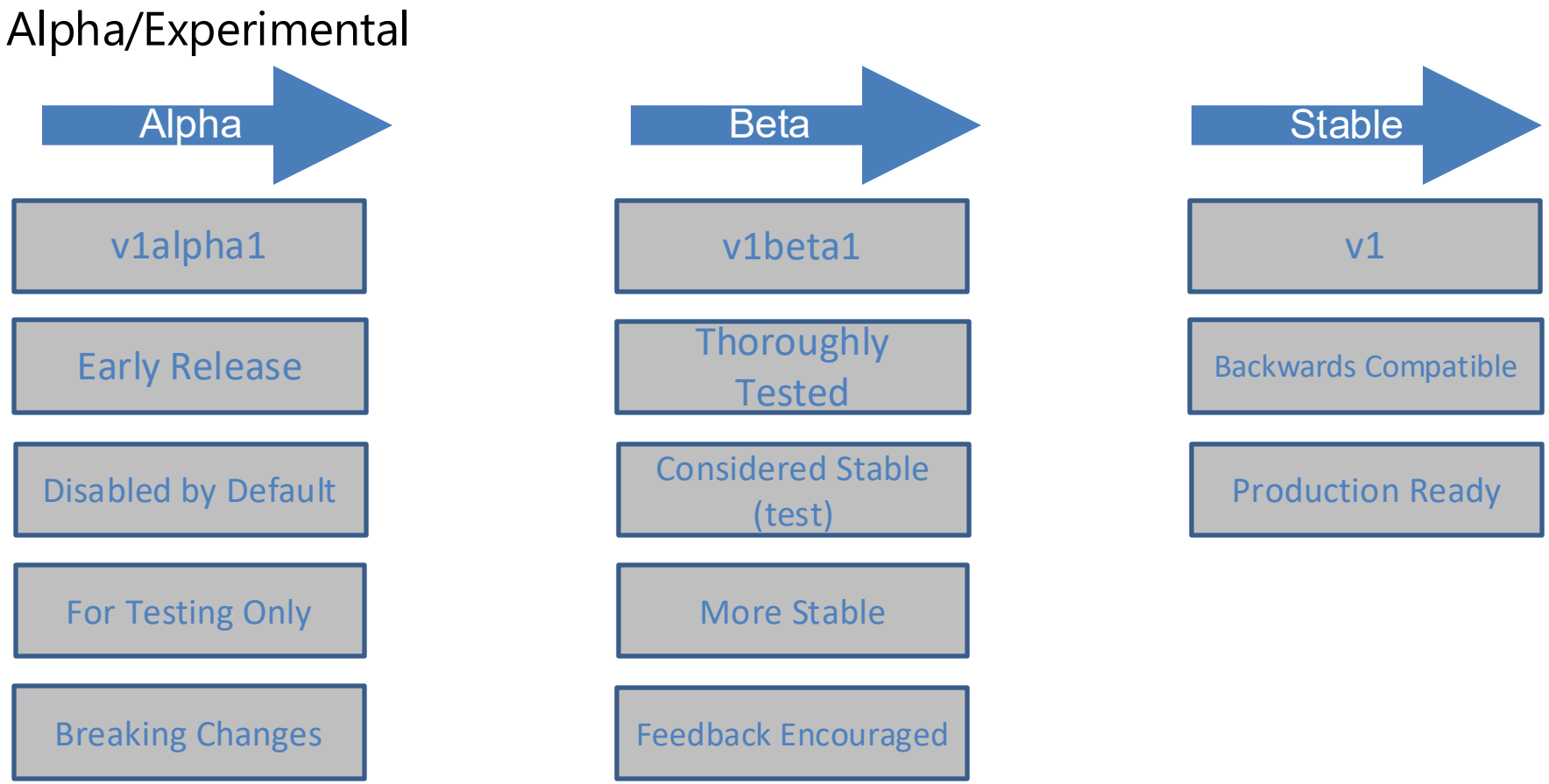
Kubernetes Resource Properties

- Every Kubernetes object has
  - **apiVersion:** object schema version
  - **kind**: type of resource
  - **metadata:** resource name and labels
  - **spec:** description of object's desired state
- Kubernetes will actively manage the state of an object to match its spec
  - spec is a 'record of intent'
- Object **status** is description of current state of the object as known to K8s

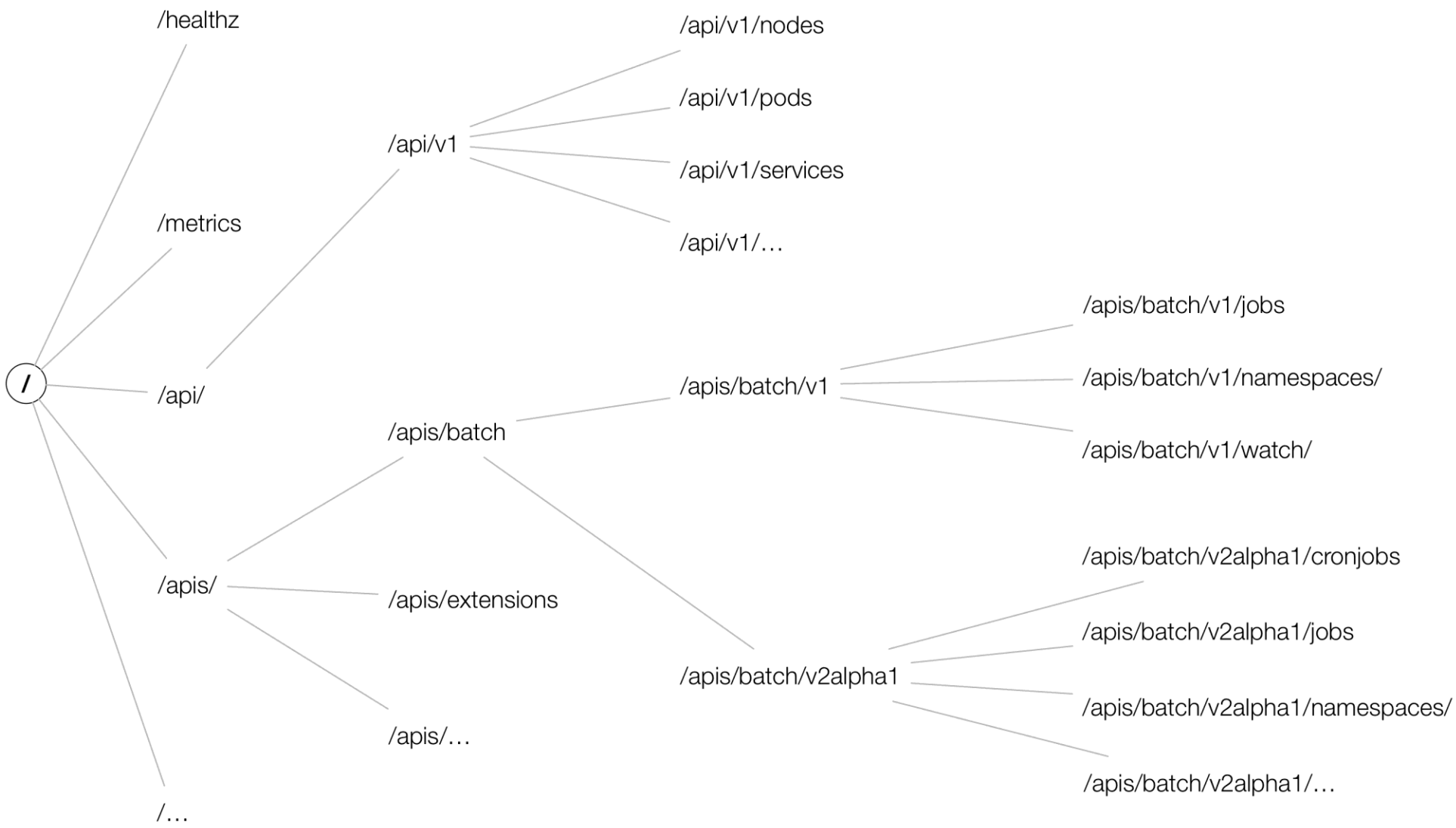
```
apiVersion: v1          # schema version
kind: Pod               # type of object
metadata:
  name: nginx           # object name
  labels:               # user-defined labels
    app: website
    tier: frontend
spec:                   # object spec values
    containers:
    - image: nginx:1.7.9
      name: nginx
      ports:
      - containerPort: 80
```

simple_pod.yaml

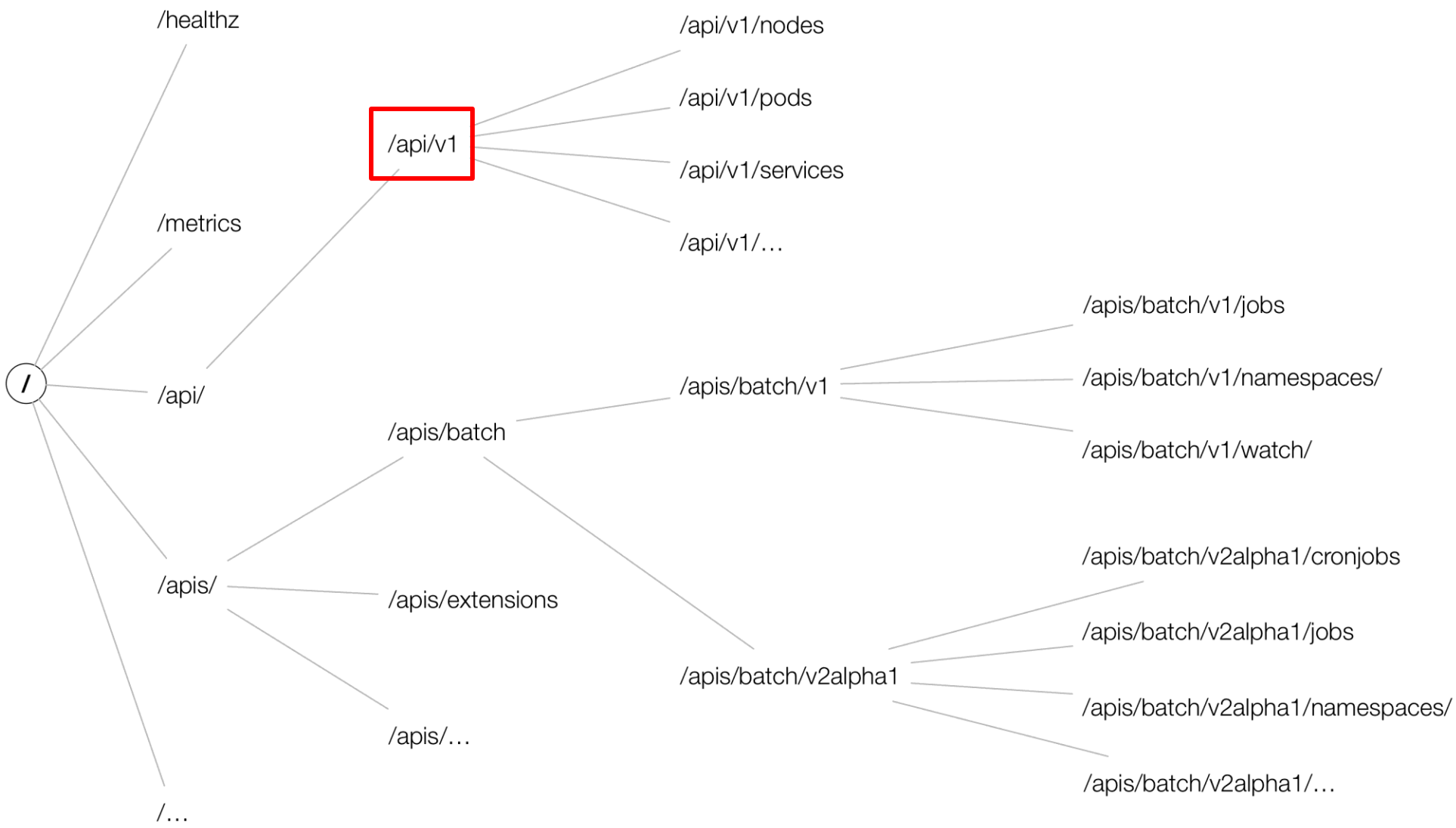
API Versioning



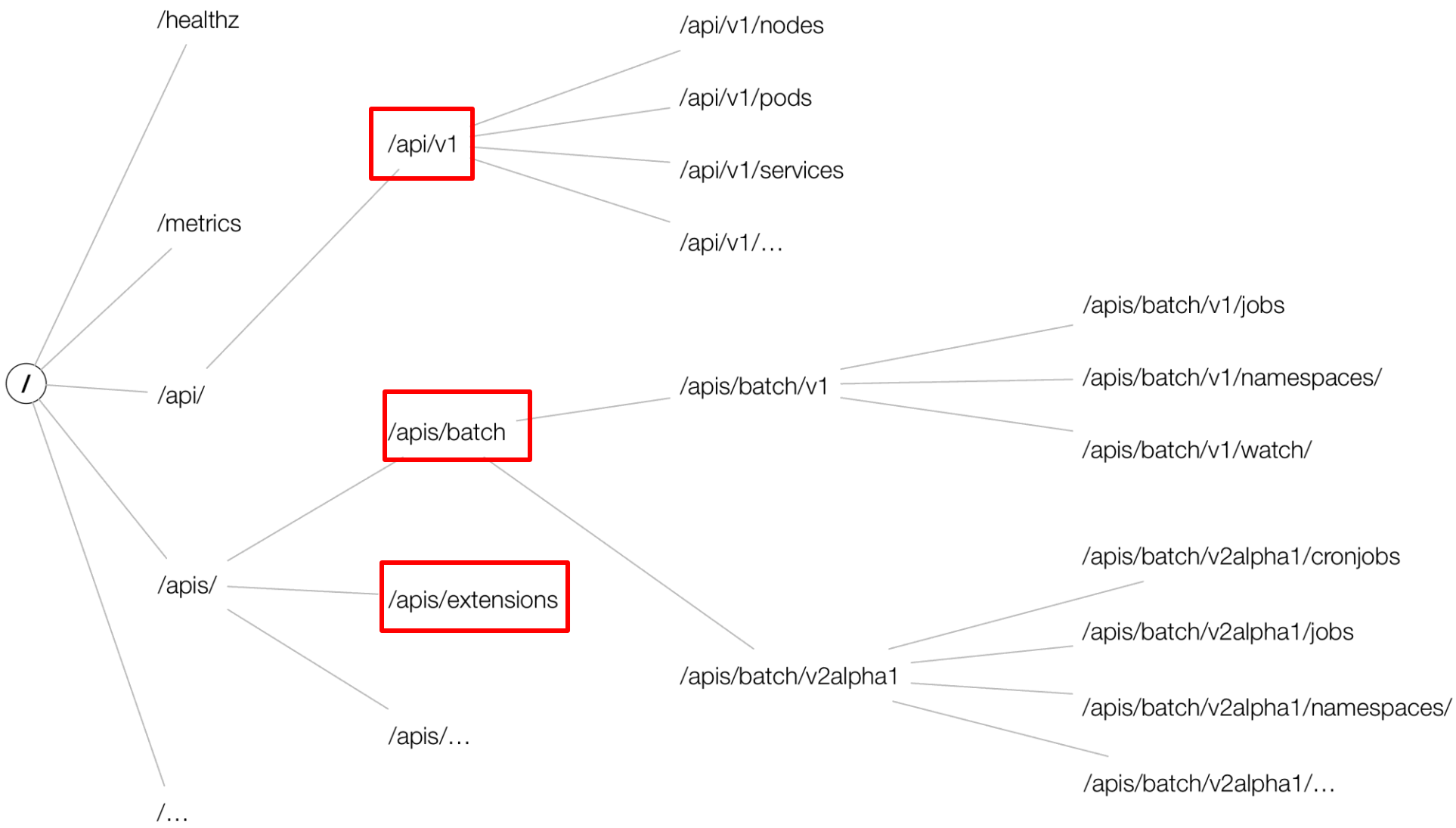
API


/
/healthz
/metrics
/api/
/api/v1
/api/v1/nodes
/api/v1/pods
/api/v1/services
/api/v1/…
/apis/
/apis/batch
/apis/batch/v1
/apis/batch/v1/jobs
/apis/batch/v1/namespaces/
/apis/batch/v1/watch/
/apis/batch/v2alpha1
/apis/batch/v2alpha1/cronjobs
/apis/batch/v2alpha1/jobs
/apis/batch/v2alpha1/namespaces/
/apis/batch/v2alpha1/…
/apis/extensions
/apis/…
/…

API Groups



API Groups



API Group

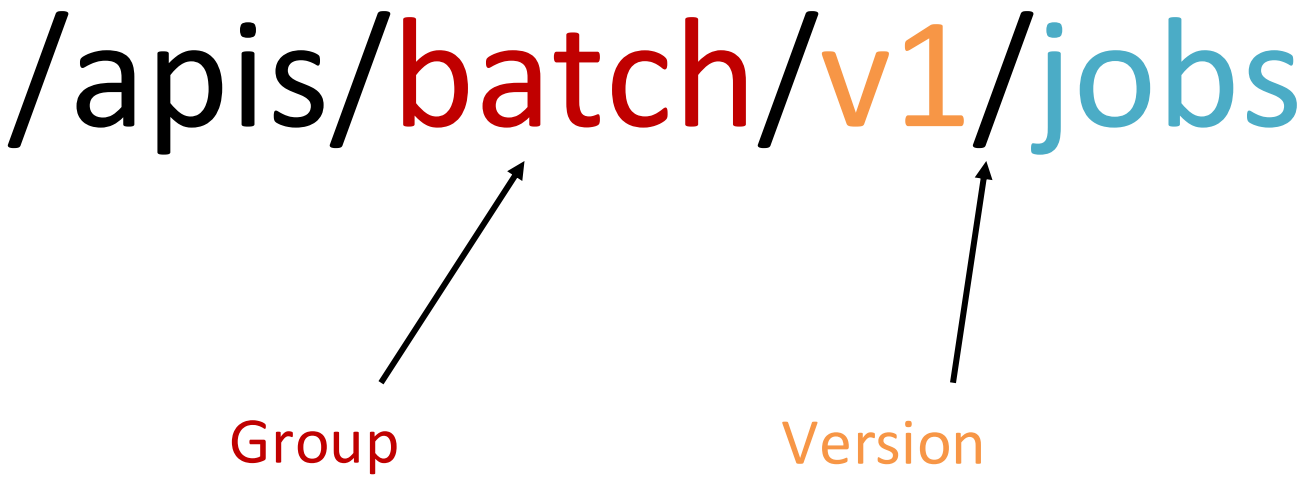
- Collection of Kinds that are logically related
  - Job, ScheduledJob in batch API Group

/apis/batch/v1/jobs

Group

API Version

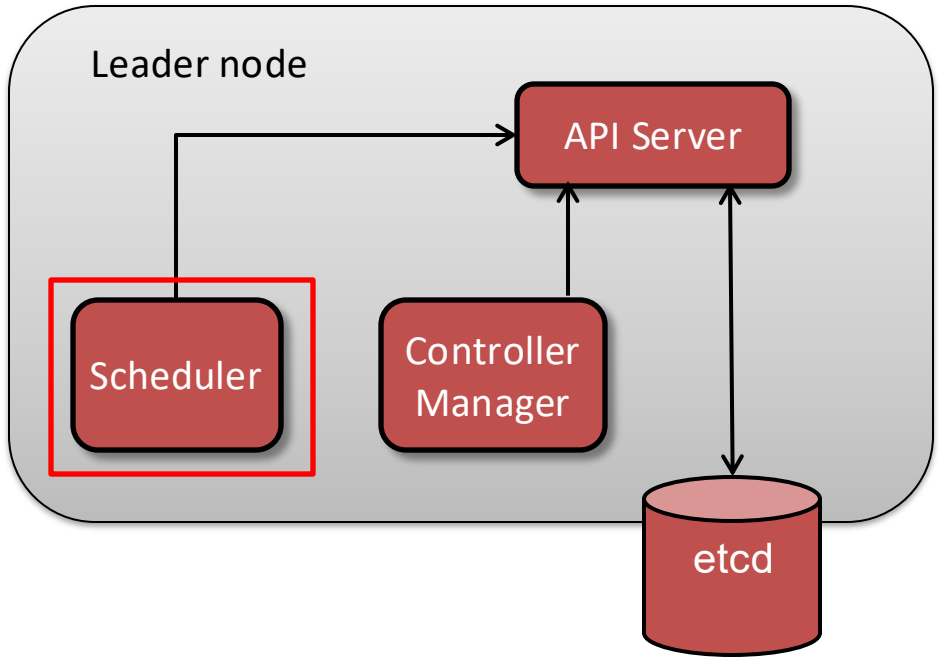
- Each API Group can be part of multiple versions
  - v1alpha1 -> v1beta1 -> v1

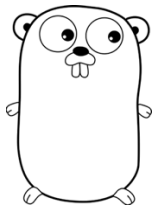


API Resource

- System entity being manipulated as JSON over HTTP

Kubernetes Control Plane/Leader Components

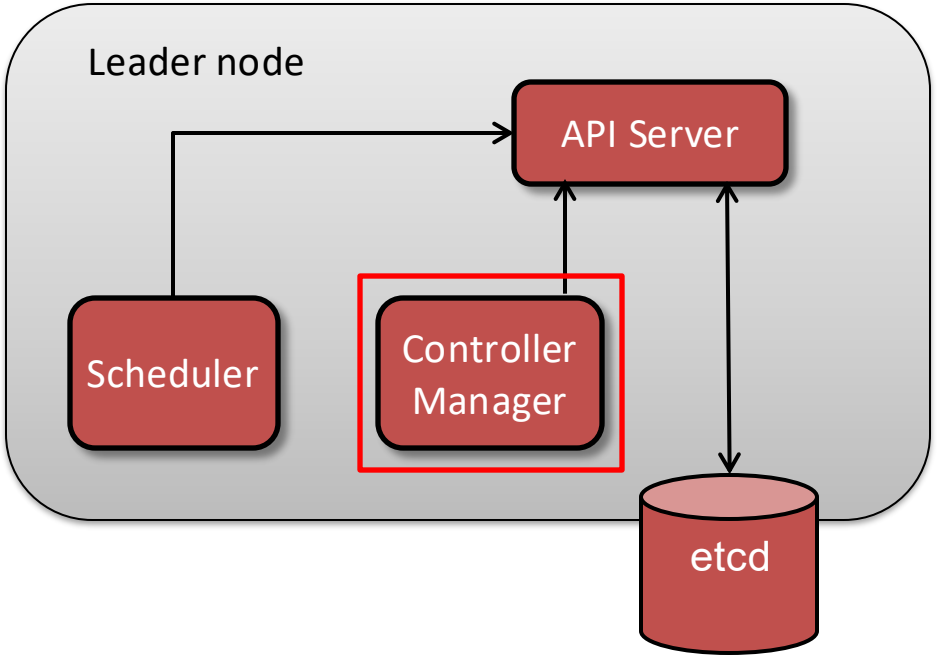


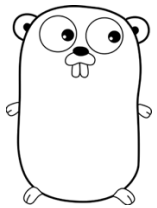


K8s components written in Go (golang.org)

- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared object states, e.g. Node Controller, Replication Controller, …
- **Persistent data store (etcd)**: all K8s system data is stored in a distributed, reliable key-value store. etcd may run on separate nodes from the master

Kubernetes Control Plane/Leader Components

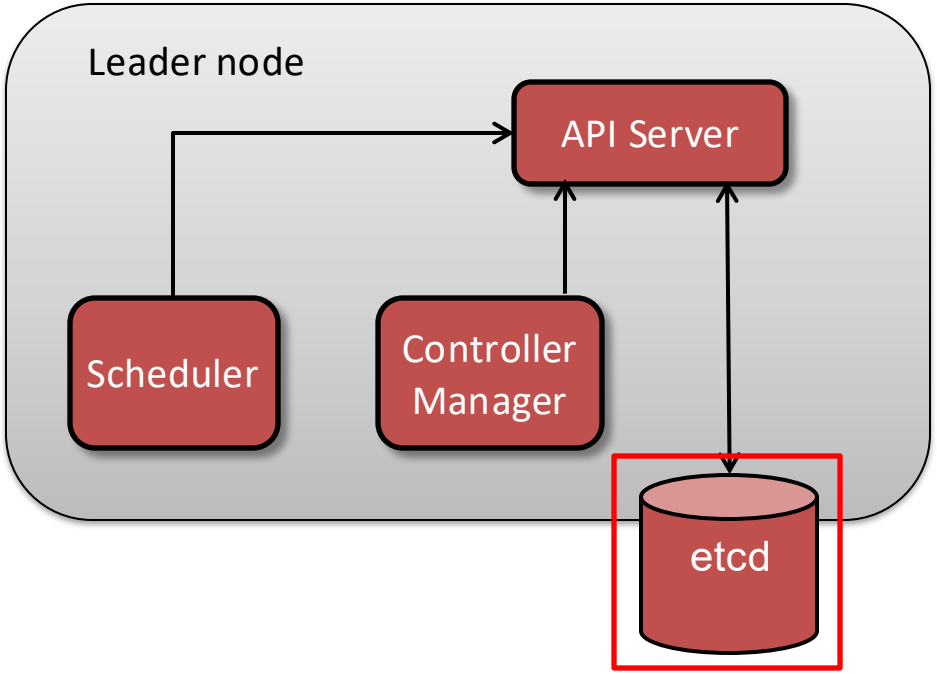


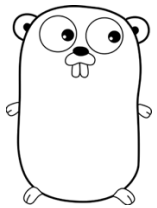


K8s components written in Go (golang.org)

- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared object states, e.g. Node Controller, Replication Controller, …
- **Persistent data store (etcd)**: all K8s system data is stored in a distributed, reliable key-value store. etcd may run on separate nodes from the master

Kubernetes Control Plane/Leader Components



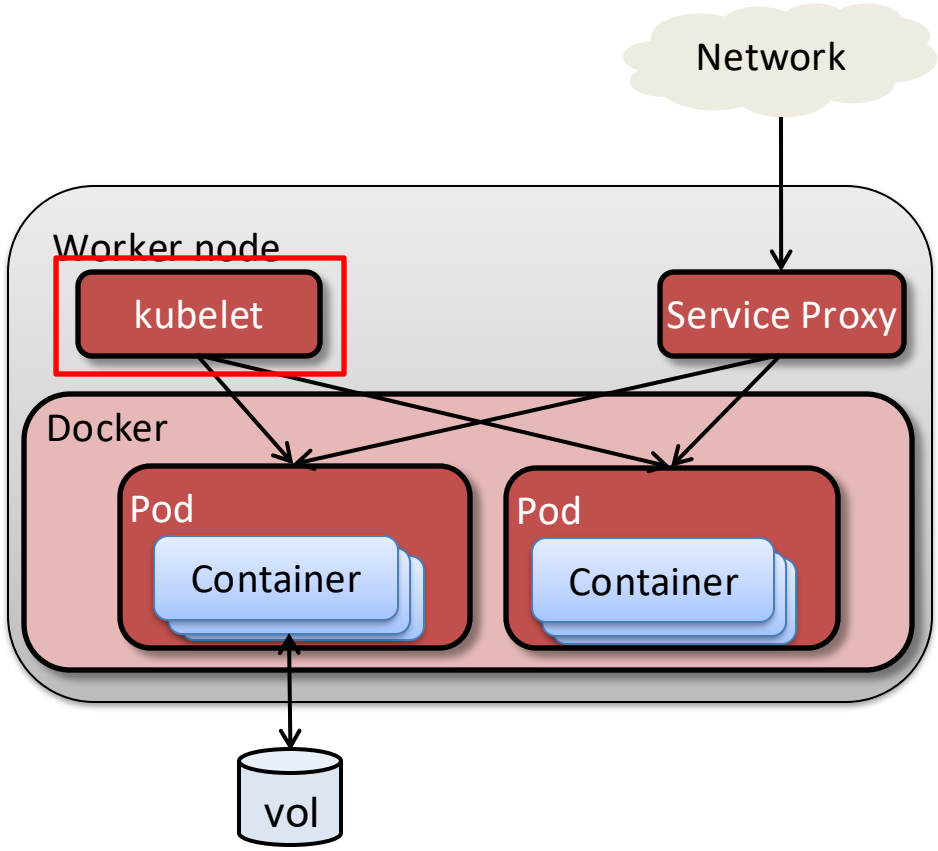


K8s components written in Go (golang.org)

- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared object states, e.g. Node Controller, Replication Controller, …
- **Persistent data store (etcd)**: all K8s system data is stored in a distributed, reliable key-value store. etcd may run on separate nodes from the master

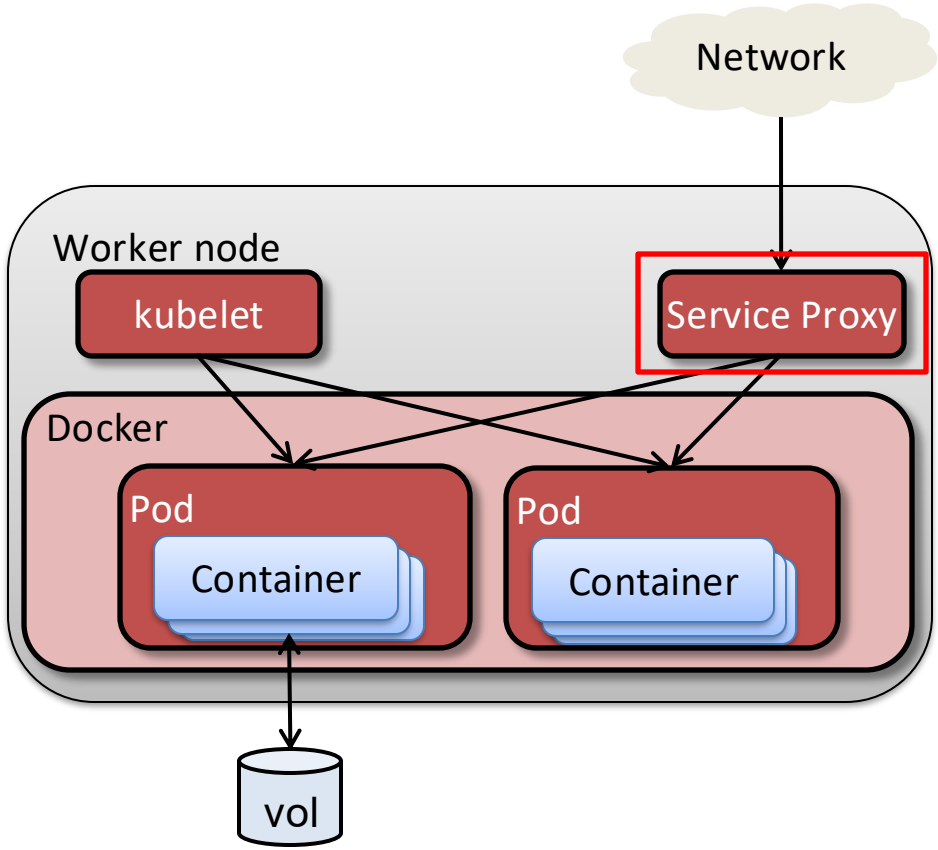
Kubernetes Worker Node Components

- **kubelet**: local K8s agent that is responsible for operations on the node, including
  - Watching for pod assignments
  - Mounting pod required volumes
  - Running a pod's containers
  - Executing container liveness probes
  - Reporting pod status to system
  - Reporting node status to system
- **Service proxy (kube-proxy):** enables K8s service abstractions by maintaining host network rules and forwarding connections
- **Docker:** runs the containers



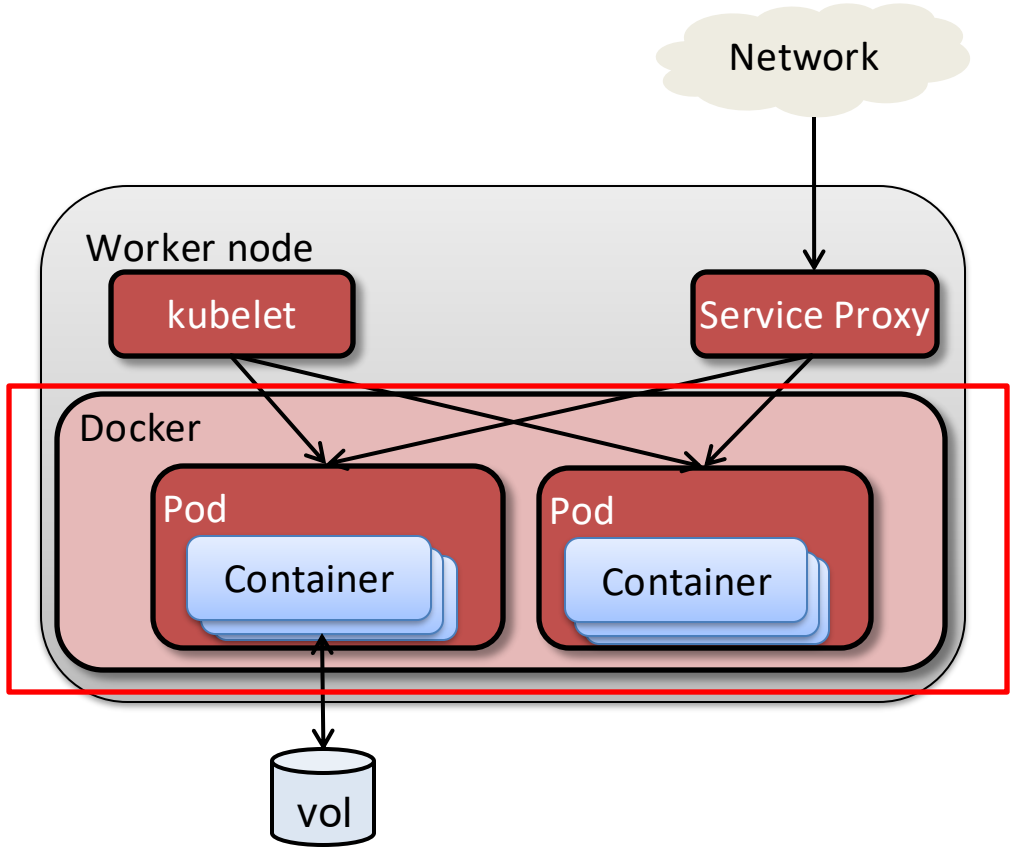
Kubernetes Worker Node Components

- **kubelet**: local K8s agent that is responsible for operations on the node, including
  - Watching for pod assignments
  - Mounting pod required volumes
  - Running a pod's containers
  - Executing container liveness probes
  - Reporting pod status to system
  - Reporting node status to system
- **Service proxy (kube-proxy):** enables K8s service abstractions by maintaining host network rules and forwarding connections
- **Docker:** runs the containers



Kubernetes Worker Node Components

- **kubelet**: local K8s agent that is responsible for operations on the node, including
  - Watching for pod assignments
  - Mounting pod required volumes
  - Running a pod's containers
  - Executing container liveness probes
  - Reporting pod status to system
  - Reporting node status to system
- **Service proxy (kube-proxy):** enables K8s service abstractions by maintaining host network rules and forwarding connections
- **Docker:** container runtime

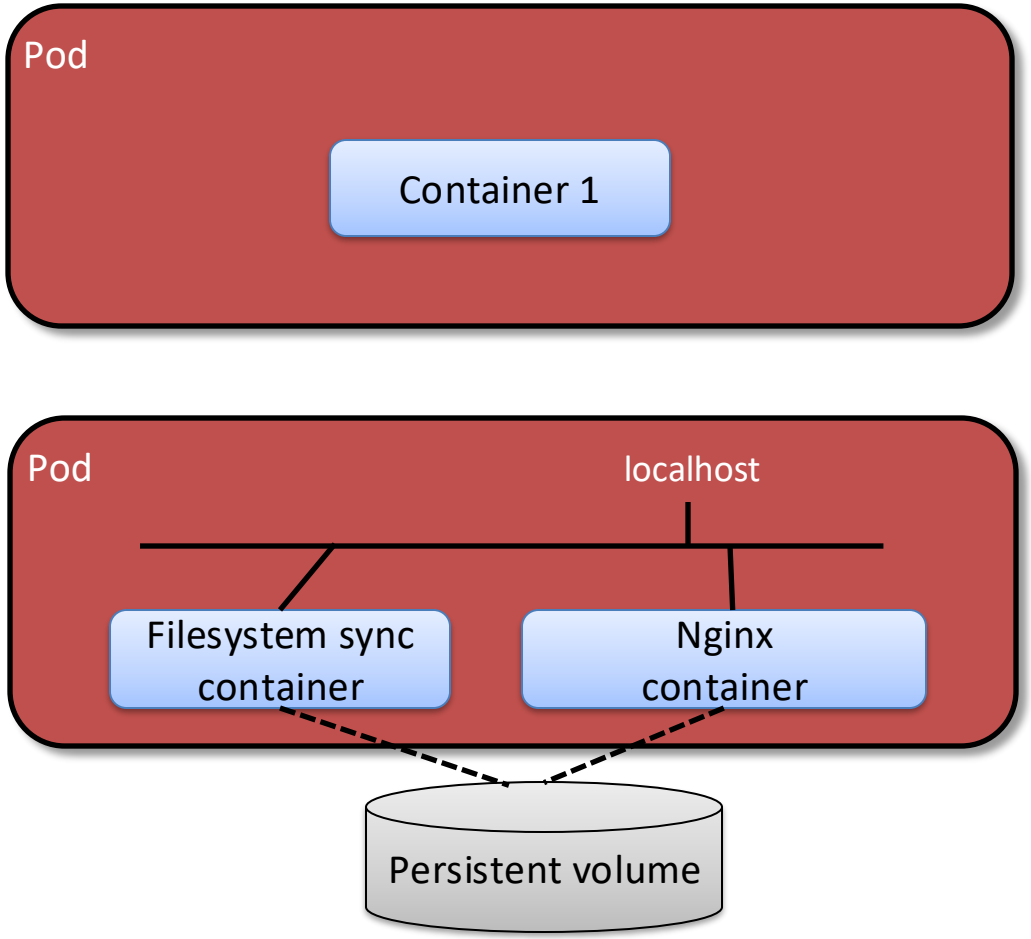


Kubernetes Pods

What is a Kubernetes Pod?

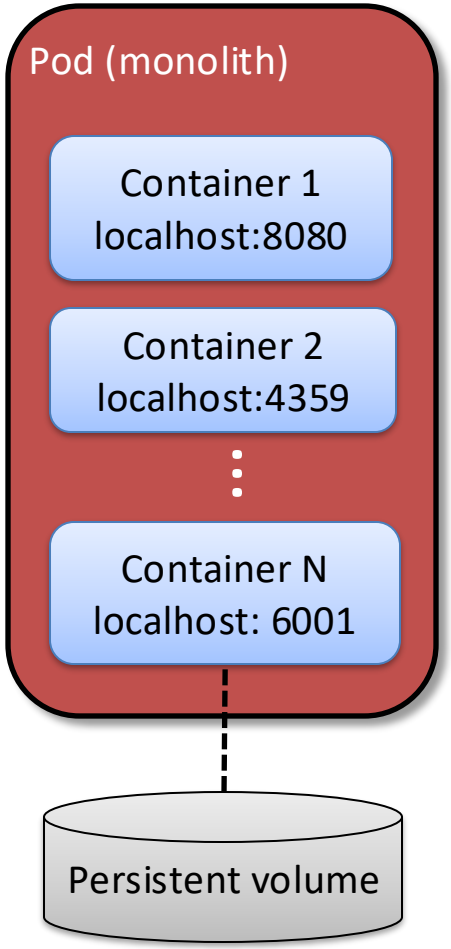
Kubernetes design intention: Pod == application instance

- Basic unit of deployment is the **pod,** a set of co-scheduled containers and shared resources
- Pods can include more than one container, for tightly-coupled application components, e.g.
  - Sidecar containers : nginx + filesystem synchronizer to update www from git
  - Content adapter: transform data to common output standard
- Containers in a pod share network namespaces and mounted volumes



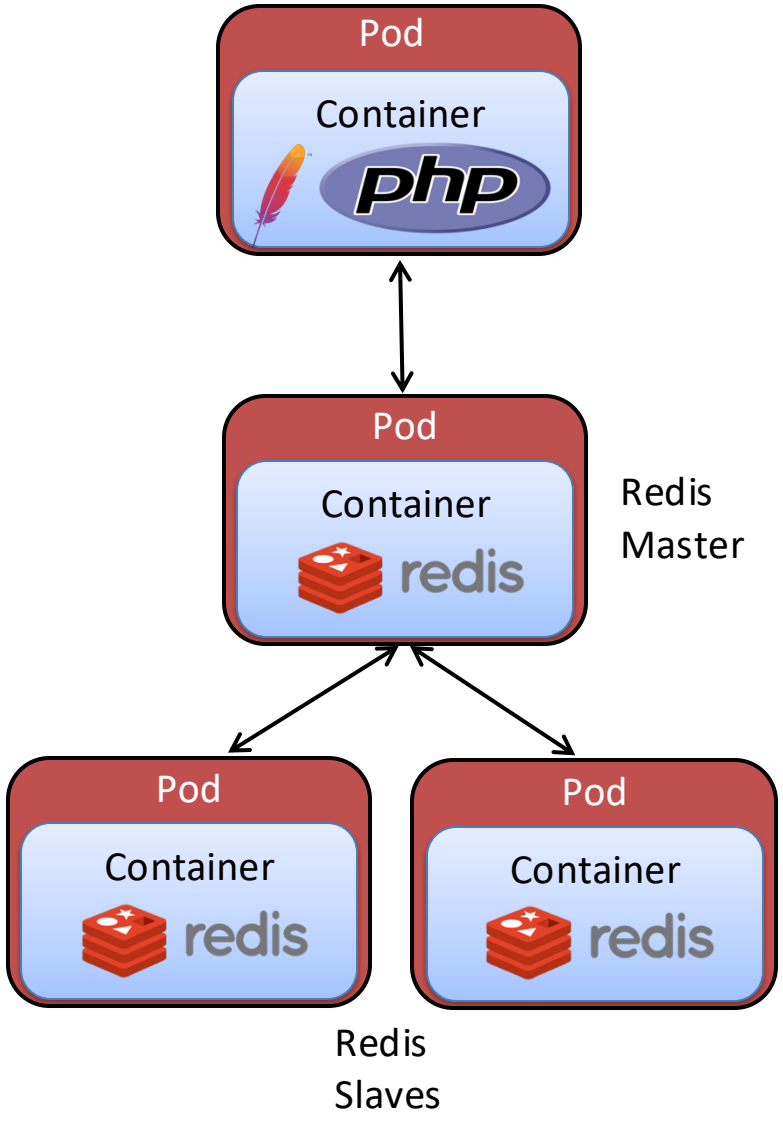
Pods Enable Deployment Flexibility

- Possible to use a single pod to run a monolithic application
  - Each application process can be built as a container
  - All containers can access each other's ports on localhost
- More advanced features of the K8s system available if application built instead from assemblages of pods, e.g.
  - Web tier: Apache pods
  - Data tier: Redis master/slave pods
- Pods provide scale and elasticity via replication – not possible in the monolith
- Best practice: assume every pod is mortal



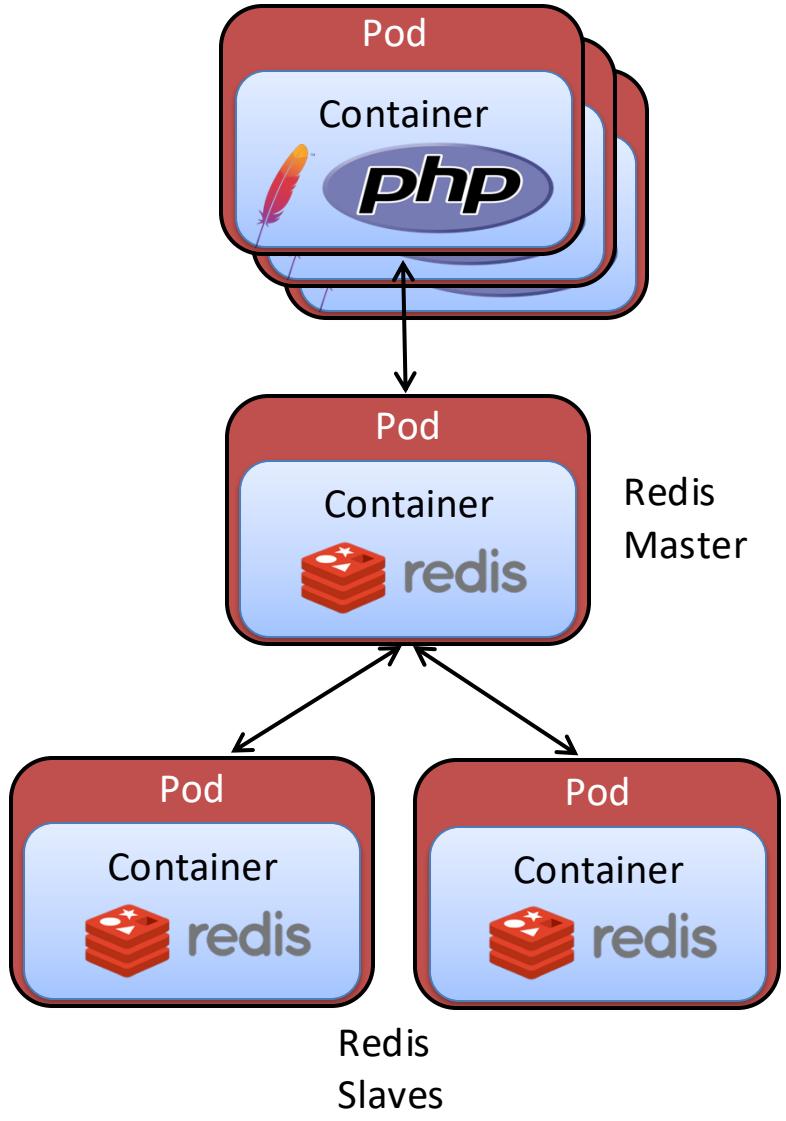
Pods Enable Deployment Flexibility

- Possible to use a single pod to run a monolithic application
  - Each application process can be built as a container
  - All containers can access each other's ports on localhost
- More advanced features of the K8s system available if application built instead from assemblages of pods, e.g.
  - Web tier: Apache pods
  - Data tier: Redis master/slave pods
- Pods provide scale and elasticity via replication – not possible in the monolith
- Best practice: assume every pod is mortal



Pods Enable Deployment Flexibility

- Possible to use a single pod to run a monolithic application
  - Each application process can be built as a container
  - All containers can access each other's ports on localhost
- More advanced features of the K8s system available if application built instead from assemblages of pods, e.g.
  - Web tier: Apache pods
  - Data tier: Redis master/slave pods
- Pods provide scale and elasticity via replication – not possible in the monolith
- Best practice: assume every pod is mortal



Defining a Pod via a Manifest File

Like other K8s objects, pods can be defined in YAML or JSON files

- K8s API accepts object definitions in JSON, but manifests often in YAML
- YAML format used by a variety of other tools, e.g. Docker Compose, Ansible, etc.
- **kind** field value is 'Pod'
- **metadata** includes
  - **name** to assign to pod
  - **label** values
- **spec** includes specifics of container images, ports, and other resources

```
apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
  labels:
    app: web
spec:
  containers:
  - image: tomcat
    name: test-tomcat
    ports:
    - containerPort: 8080
```

Looking at a Pod Manifest File

simplepod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
  labels:
    app: web
spec:
  containers:
  - image: tomcat
    name: test-tomcat
    ports:
    - containerPort: 8080
```

Type of this K8s object (service, pod, deployment, …)

User defined labels on pod

Set of containers, each With name, Docker image, and resources

Port configuration for this container

- Configuration options similar to creating Docker container directly

Defining a Pod with Multiple Containers

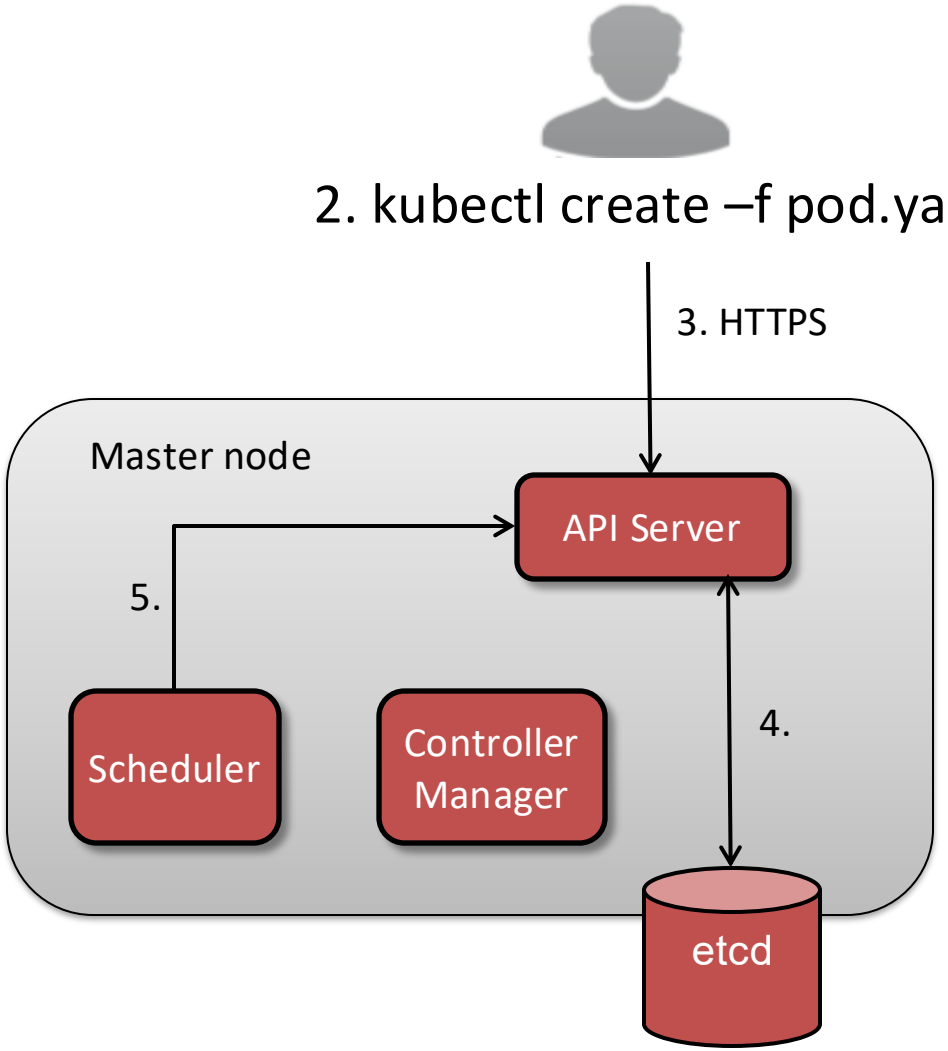
```
apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
spec:
  containers:
  - image: tomcat
    name: test-tomcat
    ports:
    - containerPort: 8080
  - image: mysql
    name: test-mysql
    ports:
    - containerPort: 3306
```


multipod.yaml

- Pod spec can contain multiple containers from different images
- Containers in pod share local network context and cluster IP for pod

Pod Creation and Management

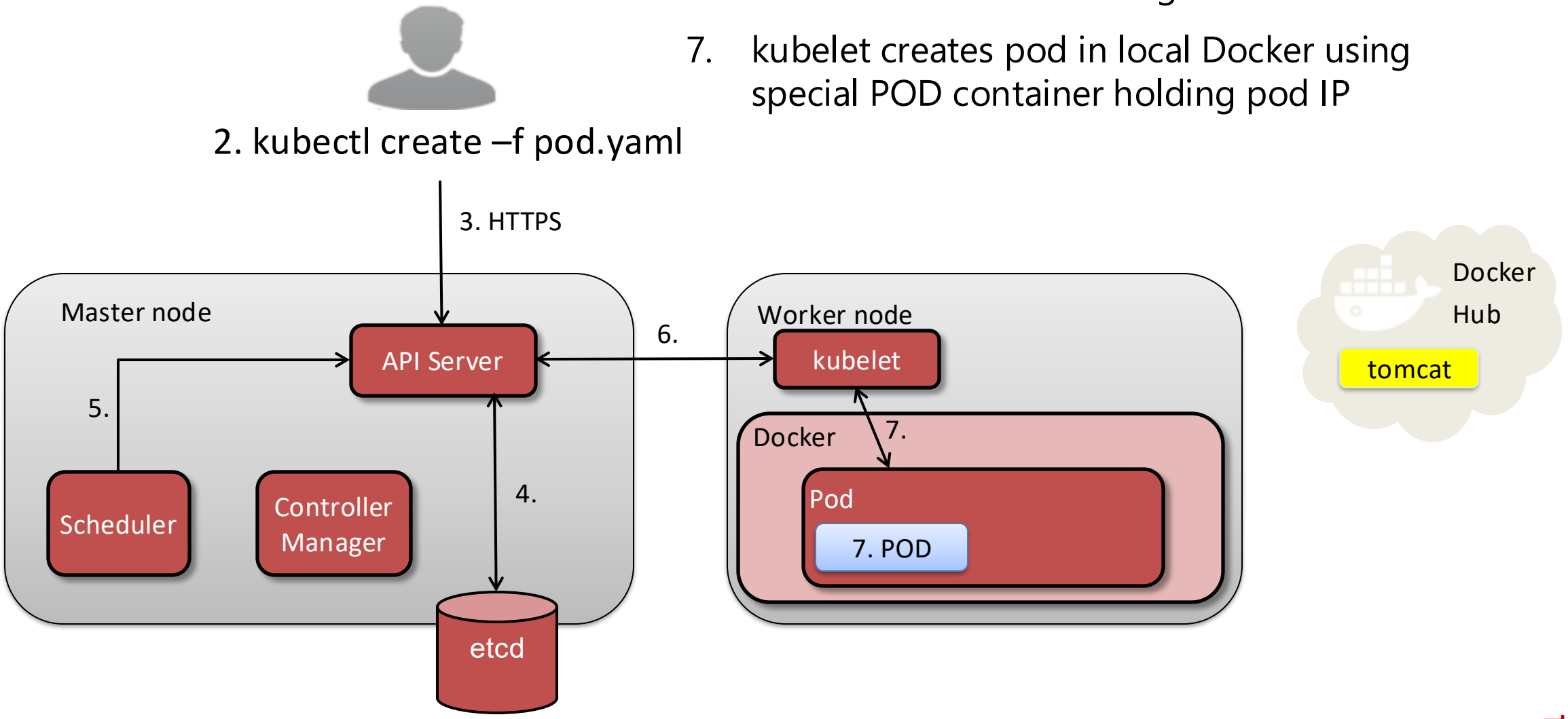
Pod Creation Process



1. User writes a pod manifest file
2. User requests creation of pod from manifest via CLI
3. CLI tool marshals parameters into K8s RESTful API request (HTTP POST)
4. kube-apiserver creates new pod object record in etcd, with no node assignment
5. kube-scheduler notes new pod via API
   a. Selects node for pod to run on
   b. Updates pod record via API with node assignment

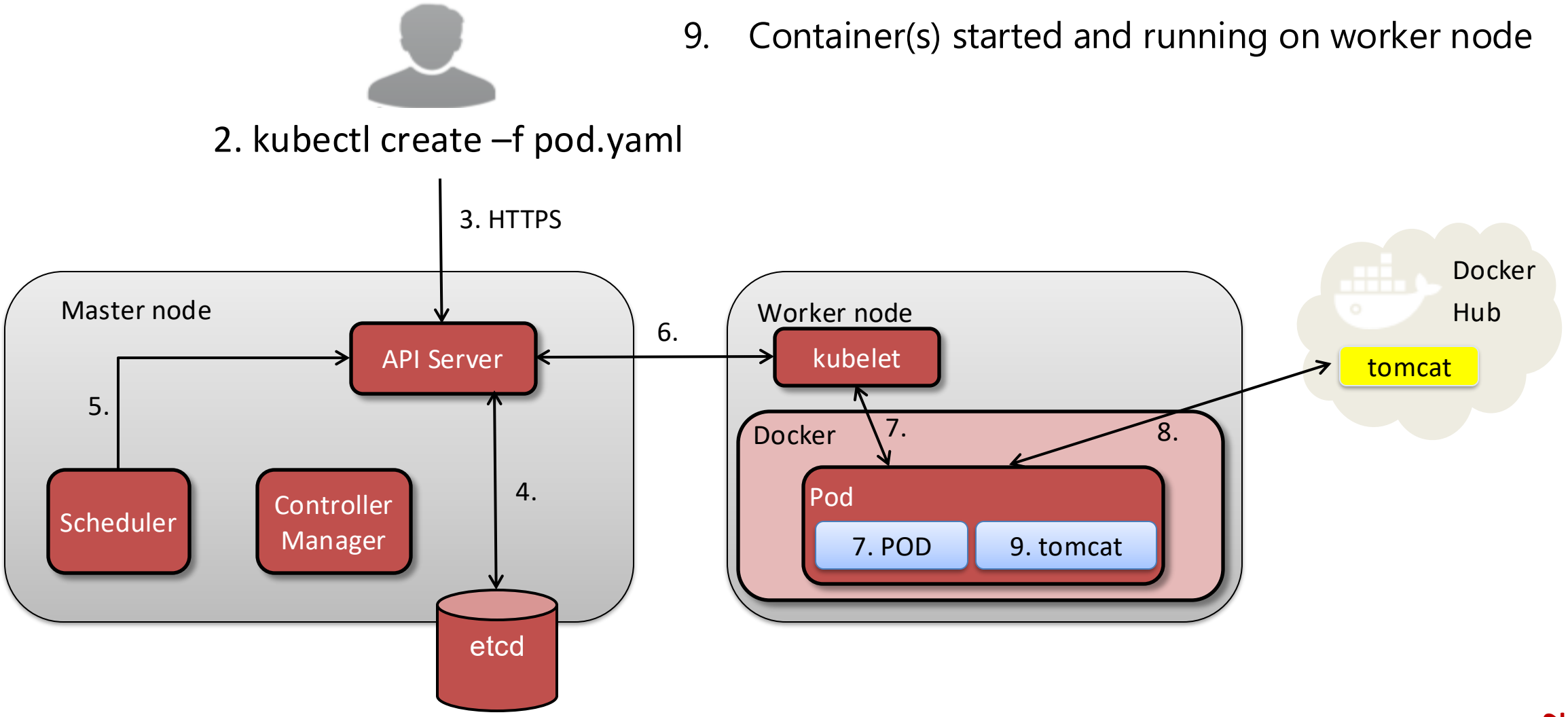
Pod Creation Process

6. Kubelet on worker node notes new pod scheduled but not running
7. kubelet creates pod in local Docker using special POD container holding pod IP



Pod Creation Process

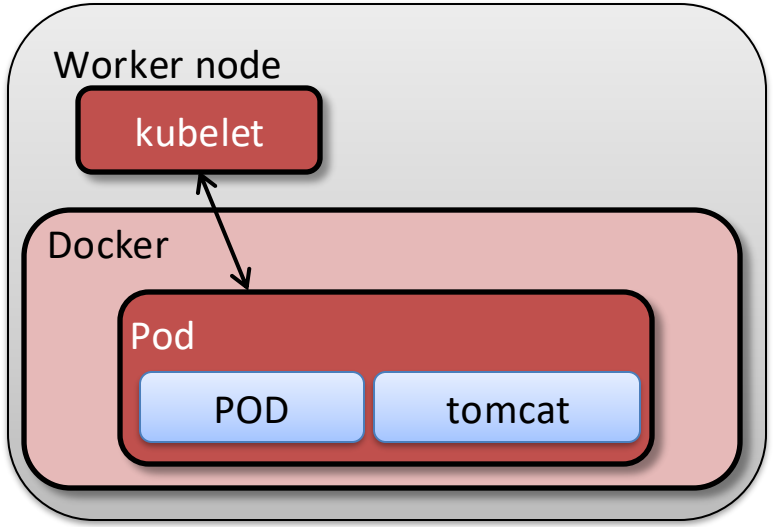
8. Kubelet pulls Docker image(s) for pod workload containers
9. Container(s) started and running on worker node



Pod Lifecycles

- By default, K8s Pods have an indefinite lifetime, which is not immortality
  - **restartPolicy** of Always by default
  - **restartPolicy** of Never or OnFailure also available for terminating jobs
- Node's kubelet will create and keep running containers for pods assigned to node, per the pod specs
- If a Pod container fails to start, or unexpectedly exits, kubelet will restart it
  - Can see container lifecycle events via 'kubectl describe pod <PODNAME>'
- If node is lost, its Pods are also lost – K8s will not rebind Pods to another node

```
apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
  labels:
    app: web
spec:
  containers:
  - image: tomcat
    name: test-tomcat
    ports:
    - containerPort: 8080
```



Modifying a Pod

Change the container version

- You can make changes to the desired state of a pod via updating the manifest file
- Changes can then be applied to the pod via the command
  - `kubectl apply –f <manifest.yaml>`
- Changing a container image as shown will result in K8s automatically killing and recreating the pod's workload container

```
$ vi simplepod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
spec:
  containers:
  - image: tomcat:8.5.5
    name: test-tomcat
    ports:
    - containerPort: 8080
```

New image version

Modifying a Pod

```
$ kubectl apply -f simplepod.yaml
pod "test-tomcat-pod" configured

$ kubectl describe pod test-tomcat-pod
Name:           test-tomcat-pod
Namespace:      default
...
Labels:         tier=frontend
Status:         Running
...
Containers:
  test-tomcat:
    Image:              tomcat:8.5.5
    Image ID:
    Port:               8080/TCP
    State:                      Running
...
```

New version running

Labeling Pods

User-defined labels help organize K8s resources

- Labels are key/value pairs that users can assign and update on any K8s resources, including pods
- Other K8s objects, like controllers, use labels to select pods to govern
- Labels can also be used to filter data queries with *kubectl,* e.g.
  - `kubectl get pods -l <label=value>`
- Labels can be used to distinguish pods on any criteria, such as
  - Application, application tier, version, environment state, etc.
- K8s system does not require specific labels to be used – all user-defined

Labeling a Pod

```
$ kubectl label pod test-tomcat-pod tier=frontend
pod "test-tomcat-pod" labeled


$ kubectl describe pods test-tomcat-pod
Name:           test-tomcat-pod
...
Labels:         tier=frontend


$ kubectl get pods -l tier=frontend
NAME              READY     STATUS    RESTARTS   AGE
test-tomcat-pod   1/1       Running   0          1d
```

Reviewing Labels on Pods

Changing kubectl output

- You can display pod labels via a flag on the *kubectl* command

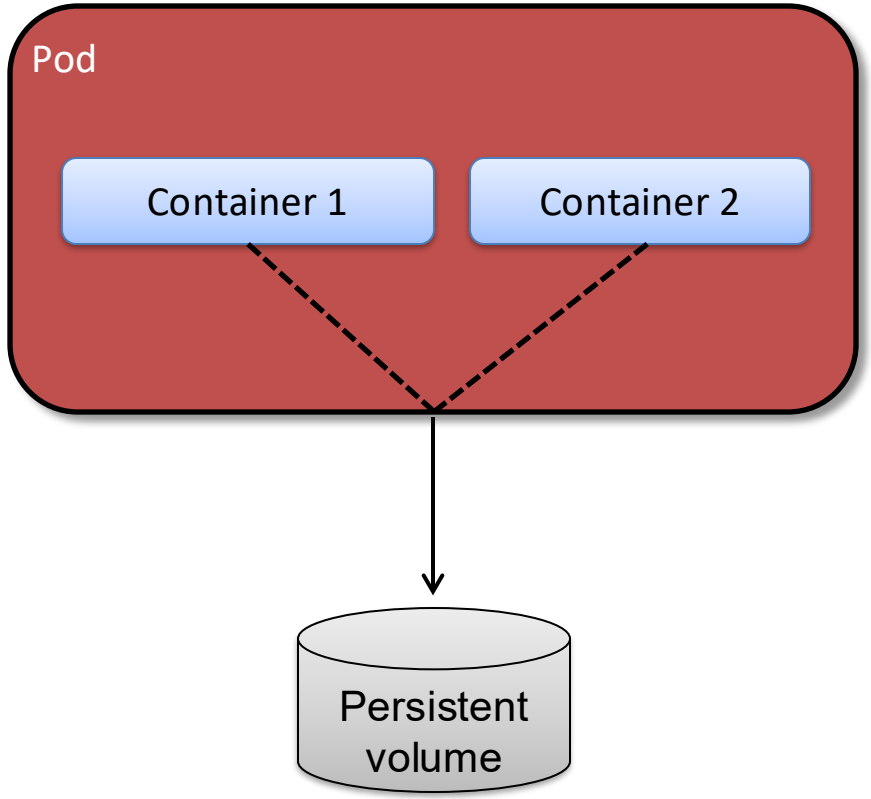
```
$ kubectl get pods --show-labels
NAME            READY   STATUS   RESTARTS AGE   LABELS
test-tomcat-pod 1/1     Running  1         1d   tier=frontend

$ kubectl get pods --show-labels --namespace=kube-system
NAME                          READY     STATUS    RESTARTS   AGE       LABELS
kube-addon-manager-minikube   1/1       Running   3          8d        component=kube-addon-
manager,kubernetes.io/minikube-addons=addon-manager,version=v6.4-alpha.1
kube-dns-v20-mm0zl            3/3       Running   9          8d        k8s-app=kube-
dns,version=v20
kubernetes-dashboard-kc9rk    1/1       Running   3          8d        app=kubernetes-
dashboard,kubernetes.io/cluster-service=true,version=v1.6.0
```

Deleting Pods

Pod deletion will discard all local pod resources

- When deleting a Pod, its containers will be removed and pod IP relinquished
- If an application needs to persist data, its pods must be configured to use persistent volumes for storage
- If a node dies, its local pods are also gone
- Best practice: use controller resources instead of managing pods directly
- Best practice: use service resources to build reliable abstraction layers for clients



Lab: Pods

How Kubernetes Runs Workloads

All Workloads are Containerized

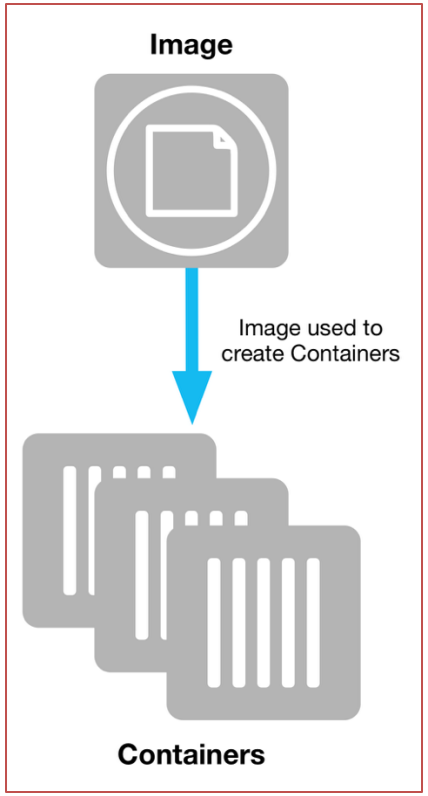
Docker allows you to package an application with its dependencies into a standardized unit for software development and deployment

Image

- Read-only template used to create containers
- Includes all dependencies for a given application
- Built by you or other Docker users
- Stored in an image registry (e.g. Docker Hub)

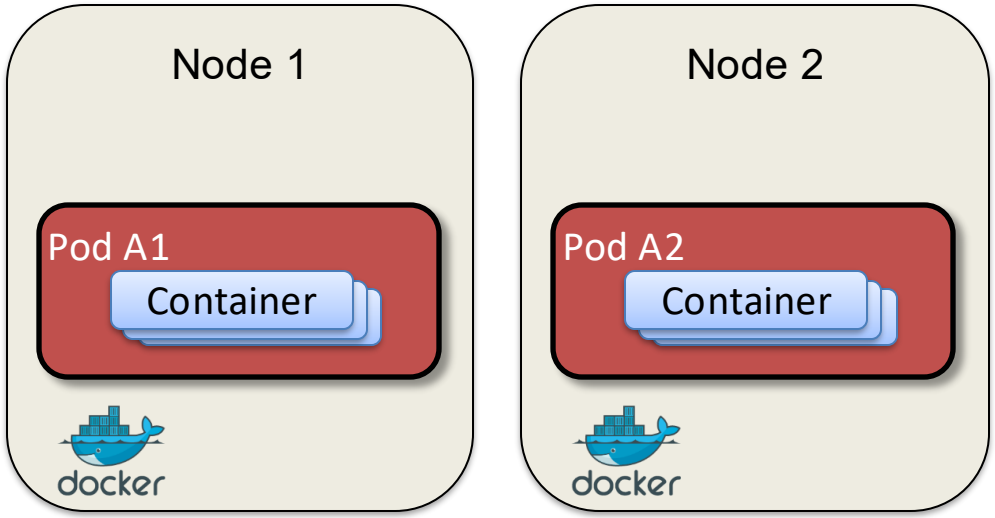
Container

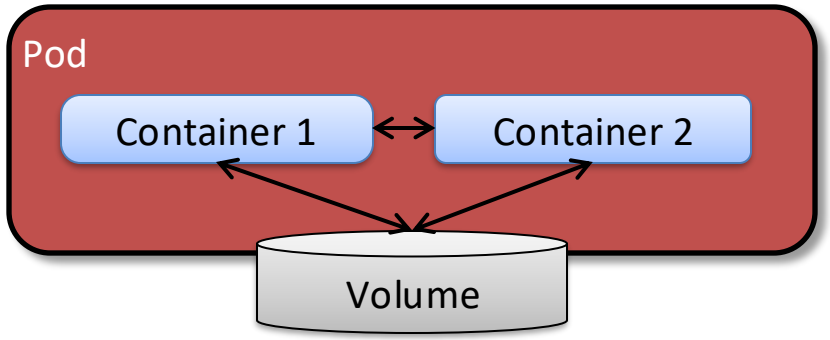
- Isolated application instance
- Created from a Docker image
- Based on Linux kernel primitives
  - Namespaces (resource visibility)
  - Control groups/cgroups (resource limits)



Kubernetes Pods

- Smallest K8s workload unit is the **Pod,** a set of co-scheduled containers
- A Pod == an application instance
- Pods can include more than one container, for tightly-coupled application components
- Containers in the same Pod share networking and storage resources
- Kubernetes handles efficient placement of Pods across available Nodes
- Pods and other K8s objects carry user-defined labels

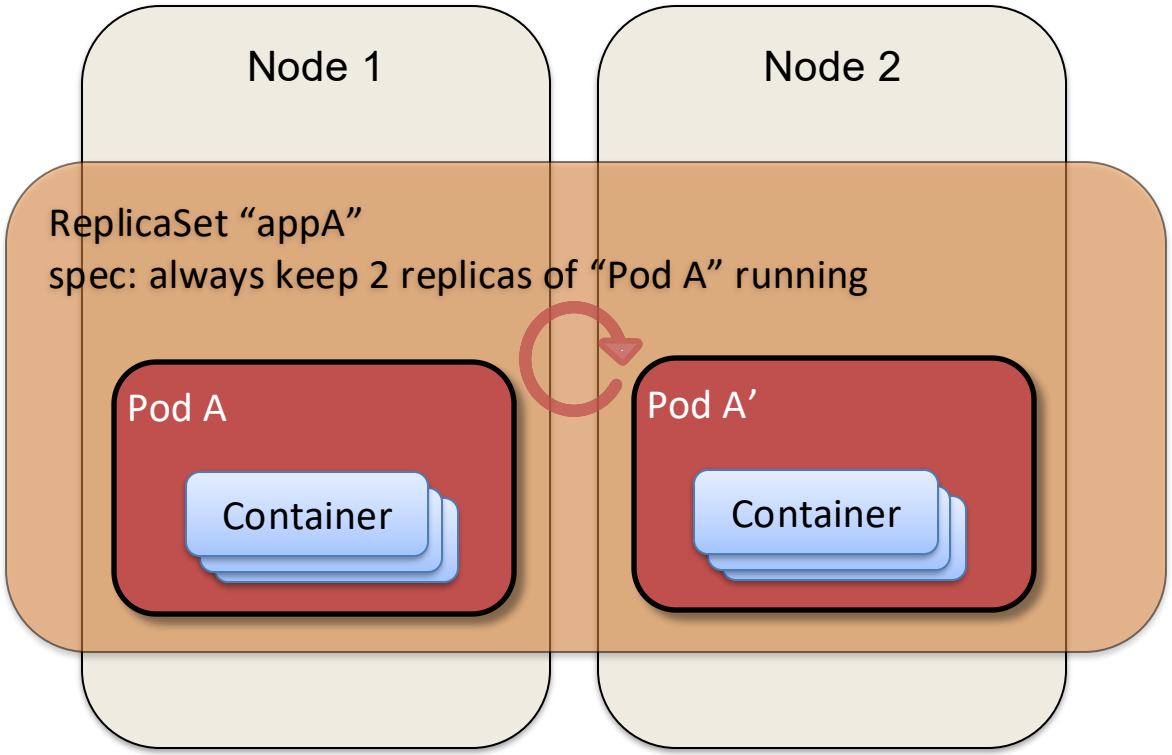




Controllers for Different Application Patterns

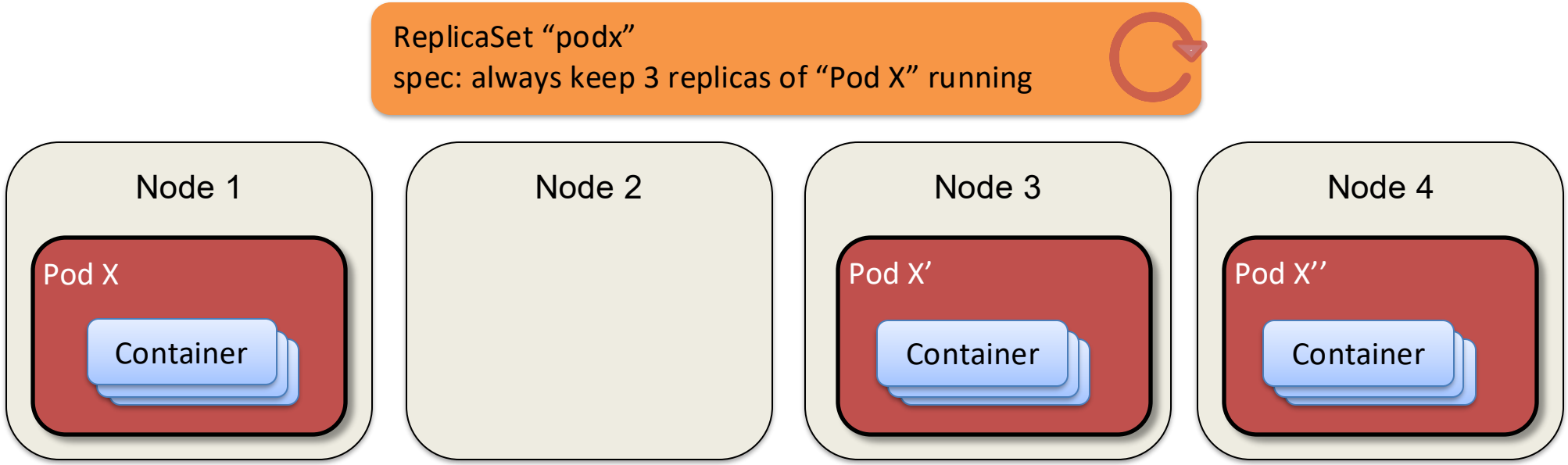
- K8s controller objects used to create and manage Pods according to different application patterns => control loops
- **ReplicaSets** manage sets of replicas of stateless workloads to ensure availability
- **StatefulSets** manage stateful workloads on stable storage to ensure consistency
- **DaemonSets** manage workloads that must run on every node, or set of nodes
- **Jobs** manage parallel batch processing workloads

Controller example: ReplicaSet



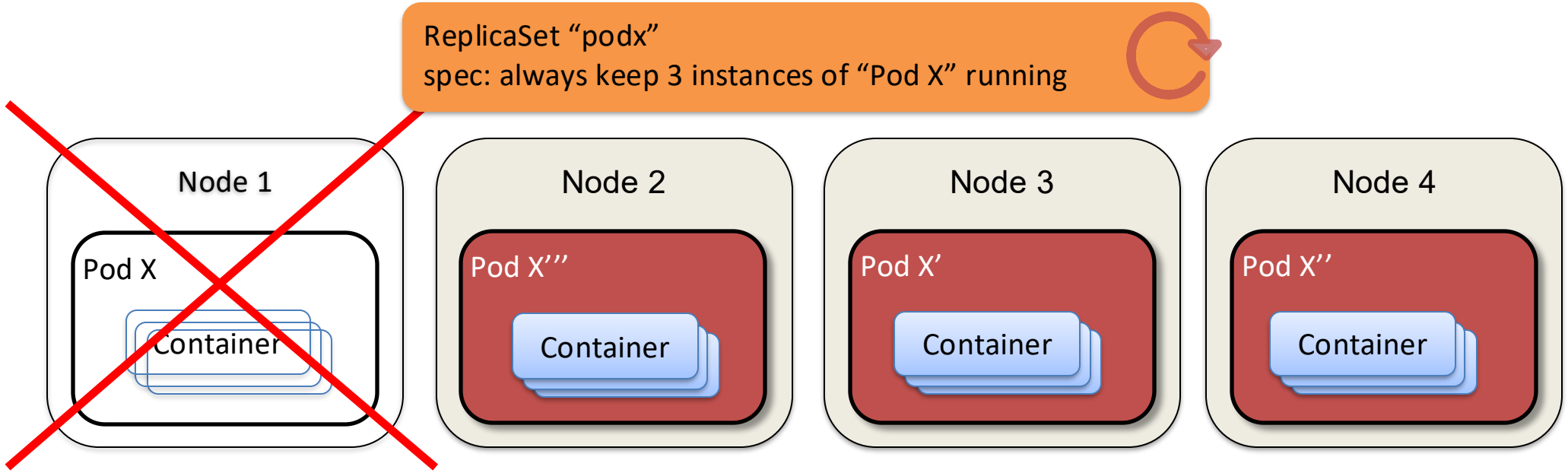
Controller Example: ReplicaSets

- ReplicaSet configuration specifies how many instances of given Pod exist
- Configuration includes Pod template to define managed Pod configuration
- ReplicaSet used for web applications, mobile back-ends, API's
  - Usually managed by Deployment controllers



Replication Ensures Application Availability

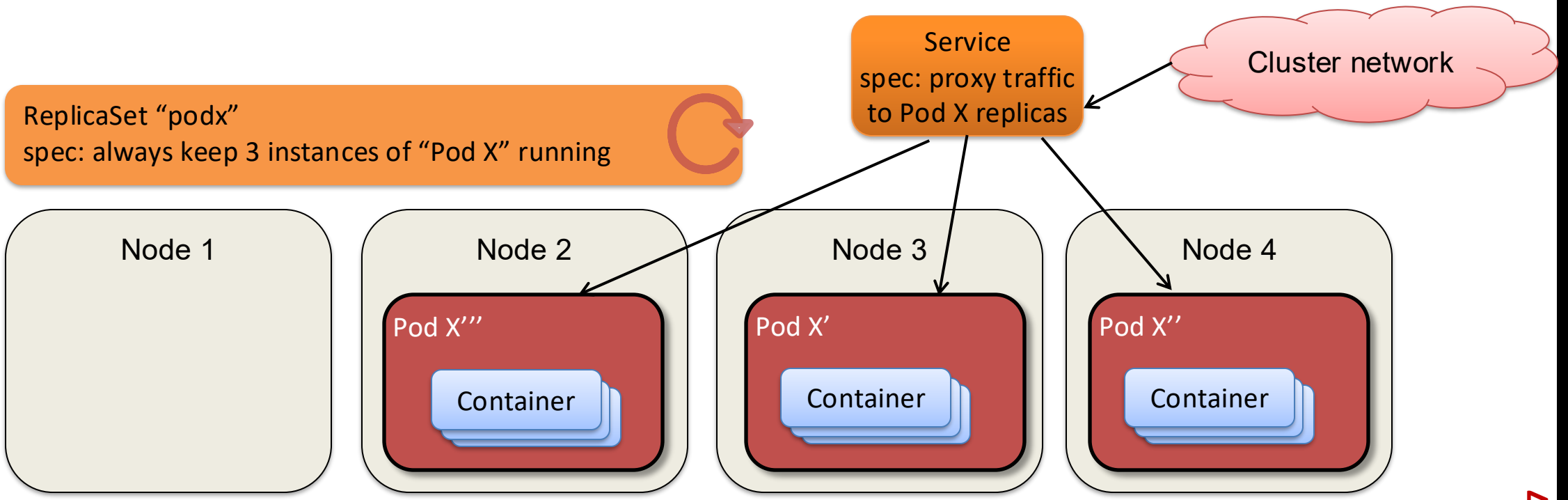
- When a Node fails, its Pods are lost
- K8s system manages the state of the ReplicaSet back to the declared configuration
- Changing the configuration will result in management to new state, e.g. scale out



Kubernetes Services Expose Applications

Services are named load balancers for application endpoints

- Service supports several different types of methods to expose an application
- Service defines stable IP and ports for application



Kubernetes Services Overview

Kubernetes Service Objects:Microservices

Services provide abstraction between layers of an application

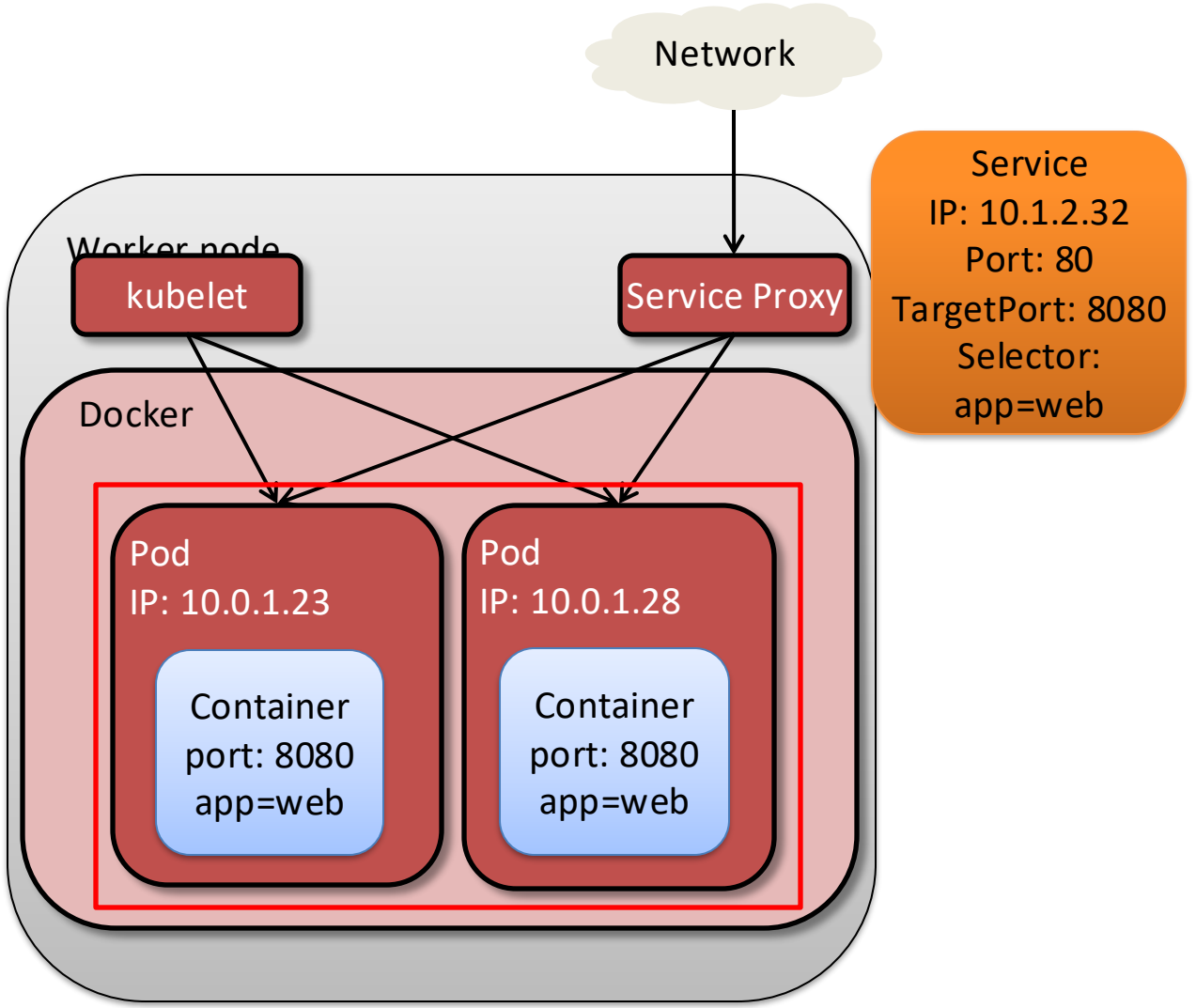
- **Service** object provides a stable IP for a collection of Pods
- Services use a label **selector** to target a specific set of pods as endpoints to receive proxied traffic
- Clients can reliably connect via the service IP and port(s), even as individual endpoint pods are dynamically created & destroyed
- Can model other types of backends using services without selectors

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: wordpress
    tier: frontend
  type: LoadBalancer
```

sampleservice.yaml

Services Provide Abstraction Layer for Applications

- Services can be used for communications between application tiers
- Services can also be used to expose applications outside the K8s cluster
- Services distribute requests over the set of Pods matching the service's selector
  - Service functions as TCP and UDP proxy for traffic to Pods
  - Service maps its defined ports to listening ports on Pod endpoints



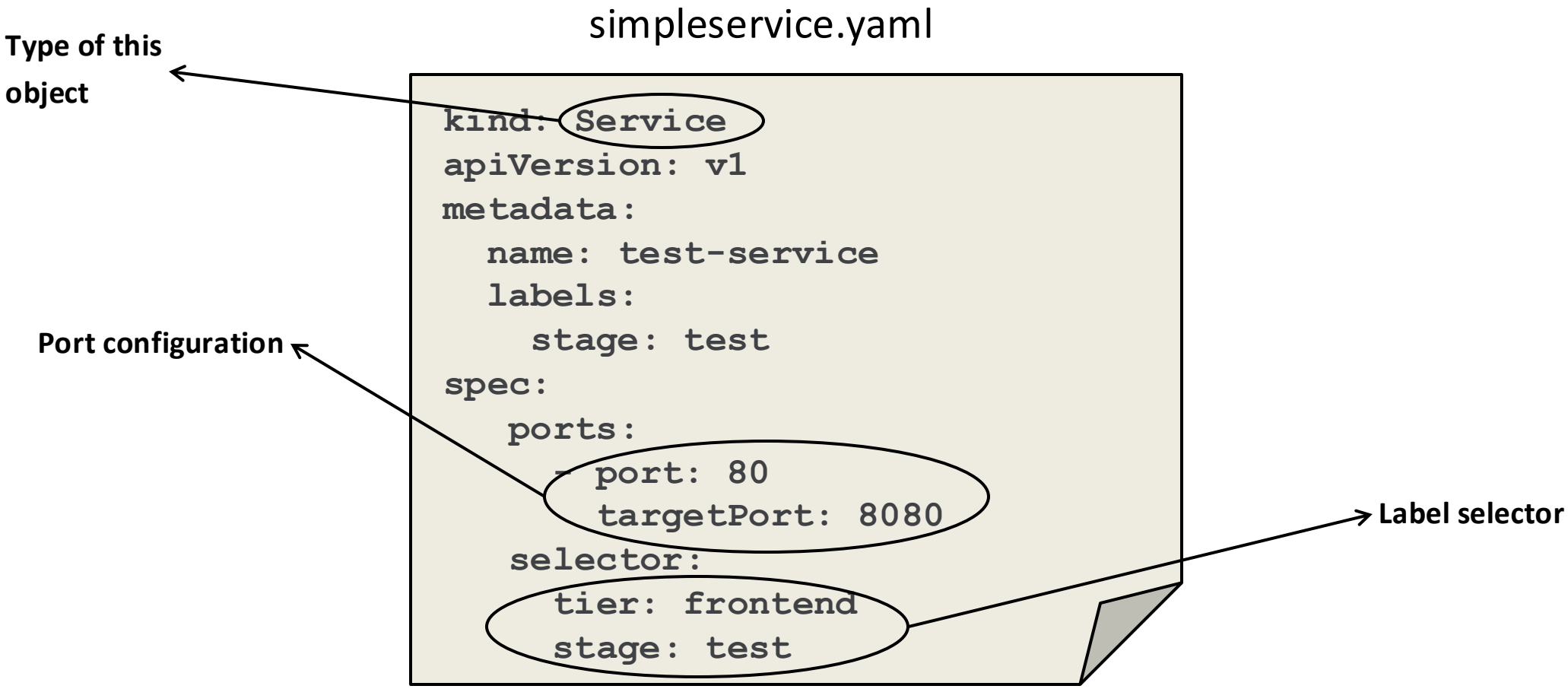
Defining a Service Using a Manifest File

Services can defined in YAML or JSON, like other K8s resources

- **kind** field value is 'Service'
- **metadata** includes
  - **name** to assign to Service
- **spec** includes the ports associated with the Service
  - **port** is the Service's port value
  - t**argetPort** is connection port on selected pods (default: **port** value)
- **selector** specifies a set of label KV pairs to identify the endpoint pods for the Service

```
kind: Service
apiVersion: v1
metadata:
  name: test-service
  labels:
    stage: test
spec:
   ports:
     - port: 80
       targetPort: 8080
   selector:
     tier: frontend
     stage: test
```

Reviewing a Service Manifest File



ServiceTypes and Exposing Applications

- By default, a Service is assigned a cluster-internal IP – good for back-ends
  - **ServiceType** ClusterIP
- To make a front-end Service accessible outside the cluster, there are other ServiceTypes available
  - **ServiceType** NodePort exposes Service on each Node's IP on a static port
  - **ServiceType** LoadBalancer exposes the Service externally using cloud provider's load balancer
- Can also use a Service to expose an external resource via **ServiceType** ExternalName

```
kind: Service
apiVersion: v1
metadata:
  name: test-service
  labels:
    stage: test
spec:
   ports:
     - port: 80
       targetPort: 8080
       nodePort: 30080
   selector:
     tier: frontend
   type: NodePort
```

Exposing Services at L7 through Ingresses

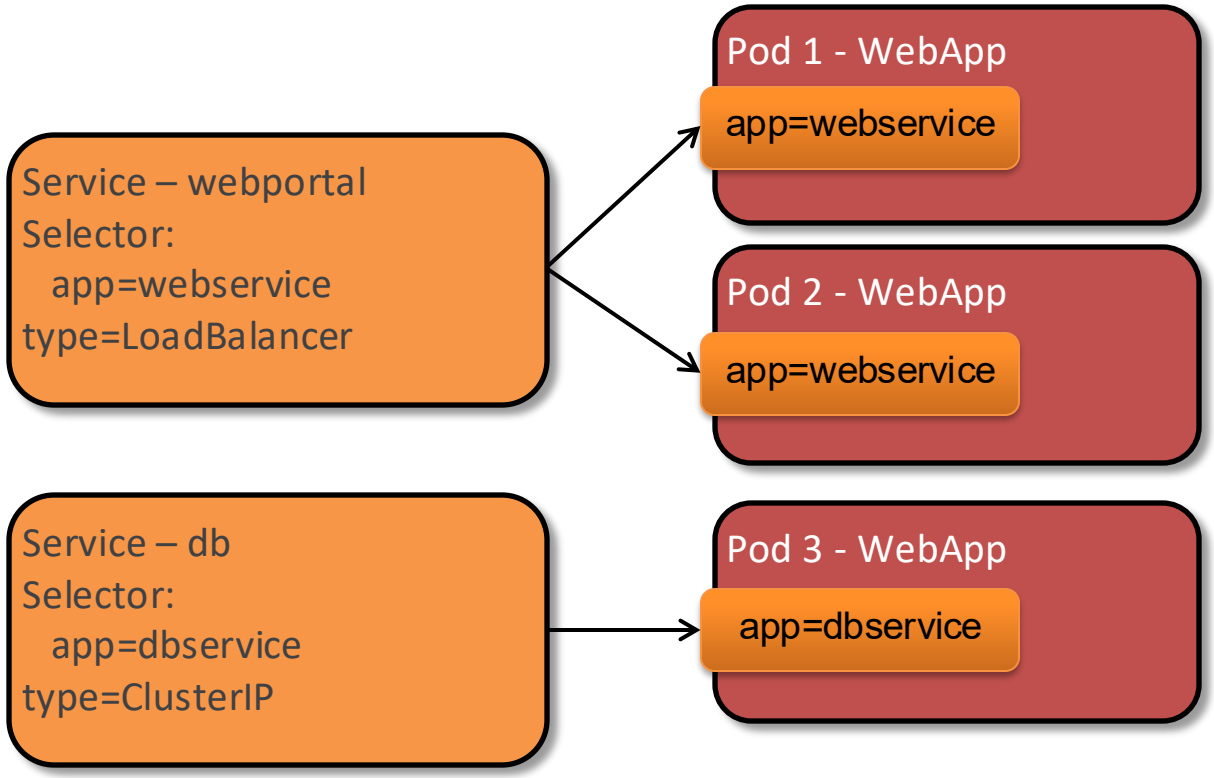
- Kubernetes also provides the facility to define **Ingress** resource to configure an external loadbalancer at L7
- Spec of Ingress resource is a set of rules matching HTTP host/url paths to specific Service backends
- Ingresses require the cluster to be running an appropriately configured Ingress controller to function (e.g. nginx)
- Useful for implementing fanout, Service backends for virtual hosts, etc.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  rules:
  -host: bar.foo.com
   http:
      paths:
      - path: /first
        backend:
          serviceName:
firstservice
          servicePort: 80
      - path: /second
        backend:
          serviceName:
secondservice
          servicePort: 80
```

Selecting Pods as Service Endpoints

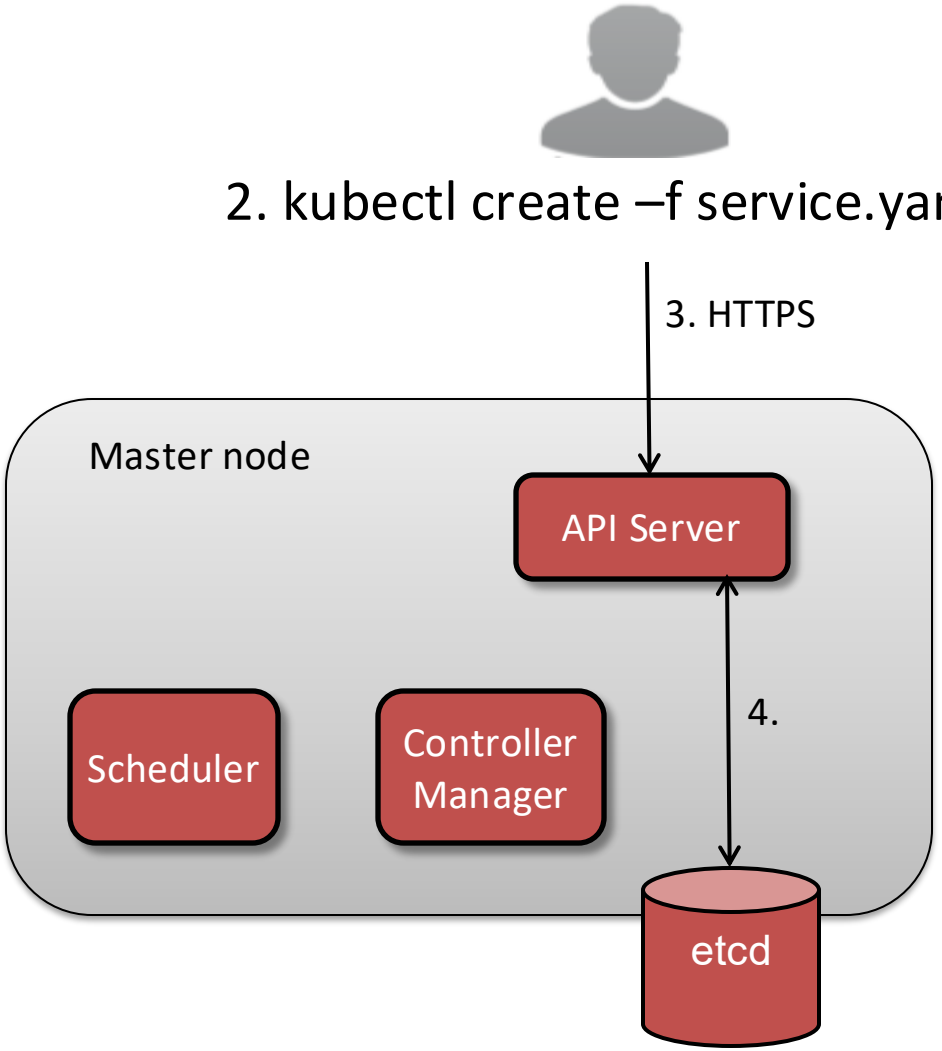
Service's pod selector based on labels

- Multiple pods can have the same label, unlike pod names which are unique in the namespace
- K8s system re-evaluates Service's selector continuously
- K8s maintains **endpoints** object of same name with list of pod IP:port's matching Service's selector



Services Management

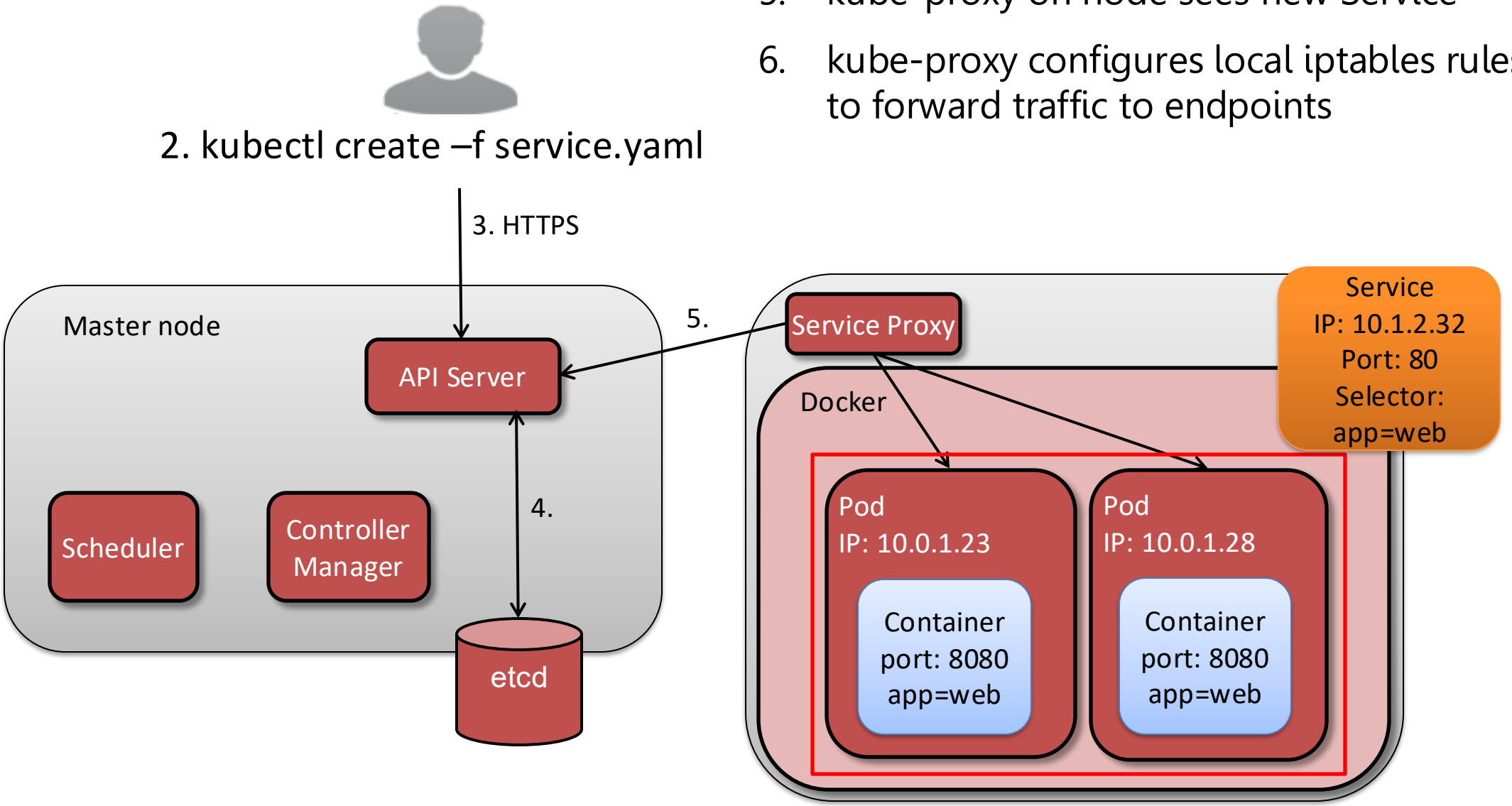
Service Creation Process



1. User writes a Service manifest file
2. User requests creation of Service from manifest via CLI
3. CLI tool marshals parameters into K8s RESTful API request (HTTP POST)
4. kube-apiserver creates new Service object record in etcd

Service Creation Process

5. kube-proxy on node sees new Service
6. kube-proxy configures local iptables rules to forward traffic to endpoints



Accessing Services Externally

Checking the service external IP

- Configured as type LoadBalancer, a configured cluster will provide an externally accessible IP for your Service

```
$ kubectl get services test-service
NAME         CLUSTER-IP   EXTERNAL-IP     PORT(S)    AGE
test-service 10.3.247.123 104.154.105.198 8080/TCP   4h
```

Deleting Services

Services can be deleted anytime

- Service deletion does not affect pods targeted by the Service's selector
- Can be done referencing the Service name or a manifest for the resource

```
$ kubectl delete -f simpleservice.yaml
```

```
$ kubectl delete service test-service
```

Service Discovery via DNS

Kubernetes advertises services via cluster DNS

- Kubernetes uses a cluster-internal DNS add-on to create and manage records for all Services in the cluster
- Pod's DNS search list includes its own namespace and cluster default domain by default

```
$ kubectl get svc
NAME           CLUSTER-IP    EXTERNAL-IP   PORT(S)          AGE
kubernetes     10.0.0.1      <none>        443/TCP          11d
test-service   10.0.0.102    <nodes>       8080:30464/TCP   2d

kubectl exec -ti busybox1 -- nslookup test-service.default
Server:    10.0.0.10
Address 1: 10.0.0.10 coredns.kube-system.svc.cluster.local

Name:      test-service.default
Address 1: 10.0.0.102 test-service.default.svc.cluster.local
```

Deployments

What is a Deployment?

Kubernetes controller optimal for stateless applications

- Deployments allow you to declaratively manage pods, including replication
- Deployments support
  - Creating, rolling out, and rolling back changes to homogeneous set of pods
  - Scaling set of pods out and back declaratively
- Deployments include
  - Implicit Replica Set controller to handle pod replicas
  - Template spec of pods to be created and managed – no need to separately create pods
- Deployments used for web applications, mobile back-ends, API's

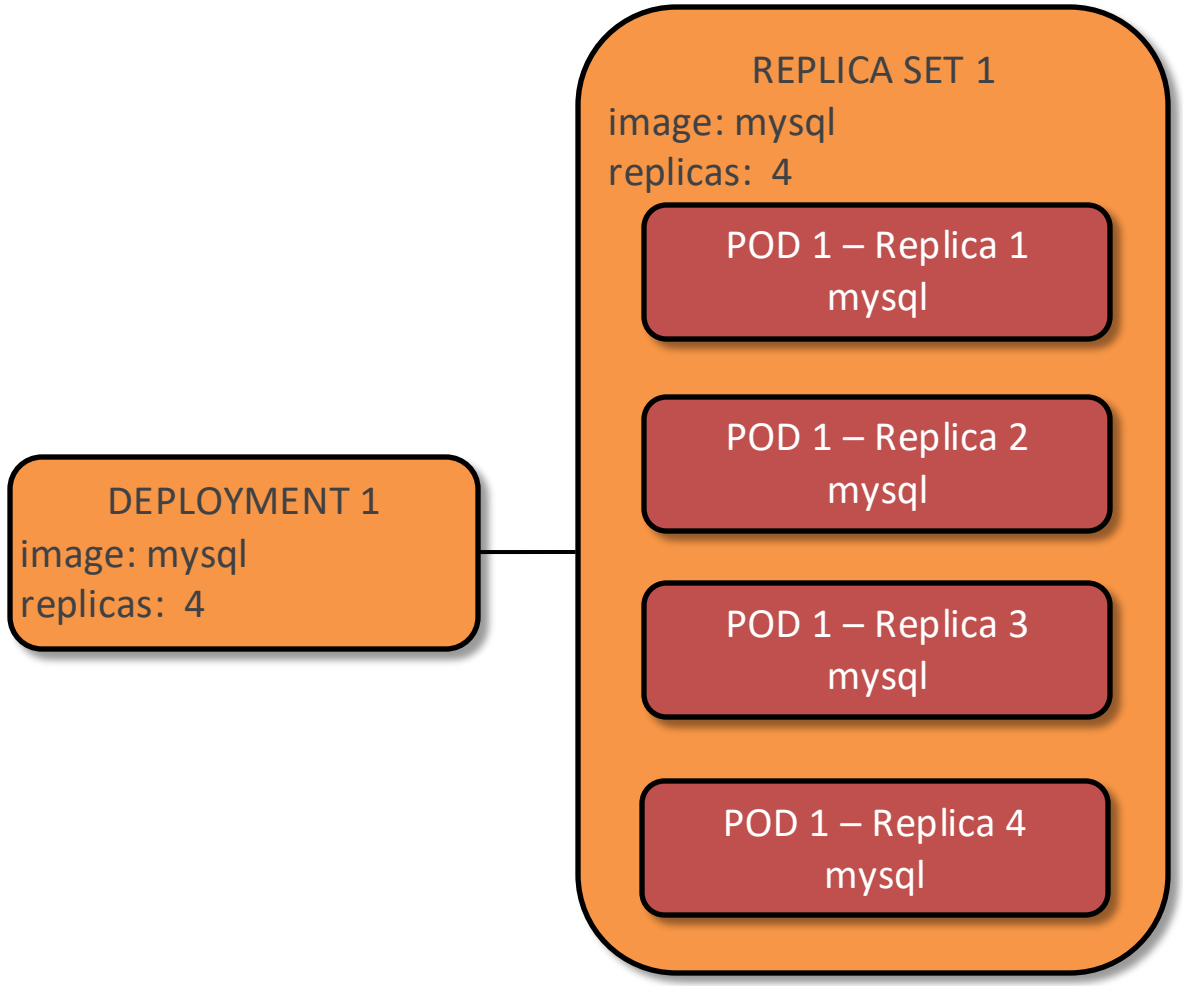
What if my Application isn't Stateless?

- Kubernetes provides other controller objects for applications that need different deployment schemes
- **StatefulSets** (previously PetSets) control deployment of pods for applications that need more stable deployment contexts
  - Pods in StatefulSets have unique ordinal, stable network identity and stable storage using persistent volumes
  - When pods are deployed, they are created in sequence of ordinals 1..N
  - Pod N must be running and ready before Pod N+1 is deployed
  - When pods are destroyed, they are terminated in reverse sequence N..1
- **DaemonSets** ensure that a replica of a specified pod is running on every node (or every selected node) in the cluster
- **Jobs** manage sets of pods where N must run to successful completion

Deployments Control ReplicaSet Controllers

Definition of how many replicated Pods should exist

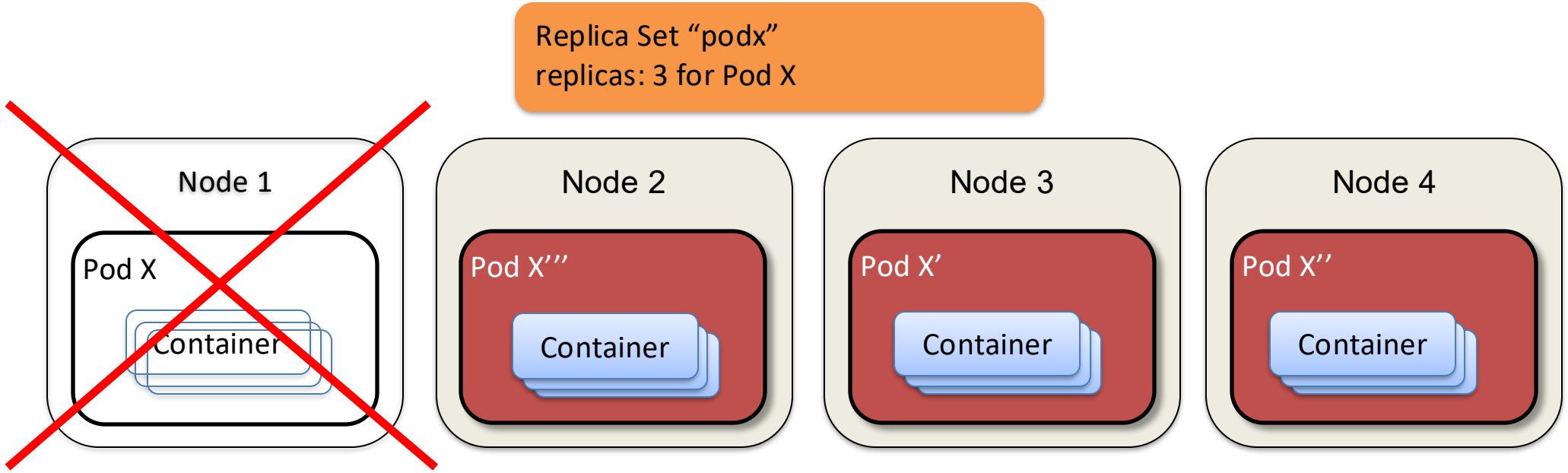
- Deployment creates and manages a Replica Set that manages a set of pods
- Replica count can be adjusted as needed to scale the Replica Set out and back
- Replica Set successor to the ReplicationController object



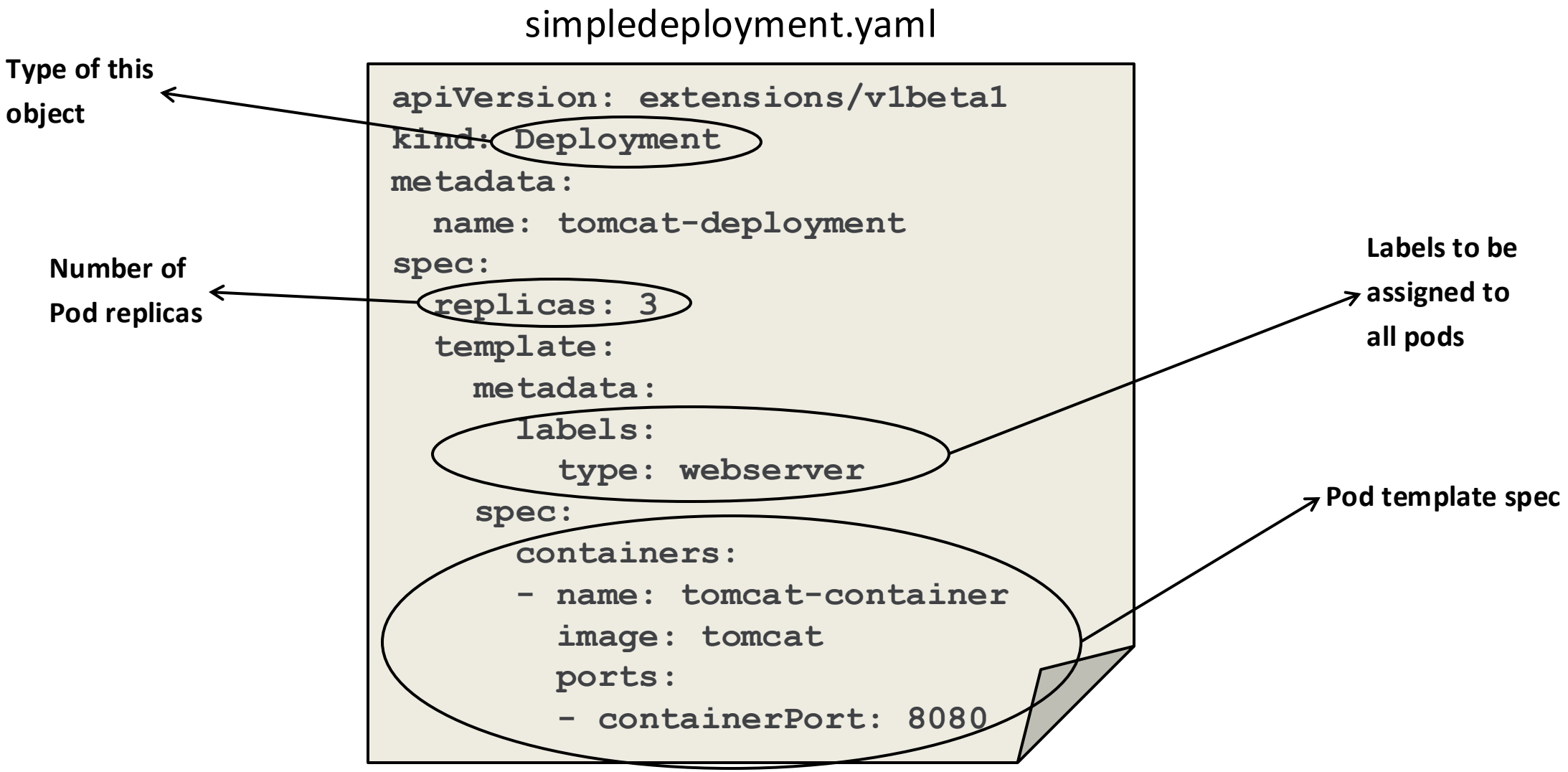
What is a Replica Set?

Provides scaling and high availability

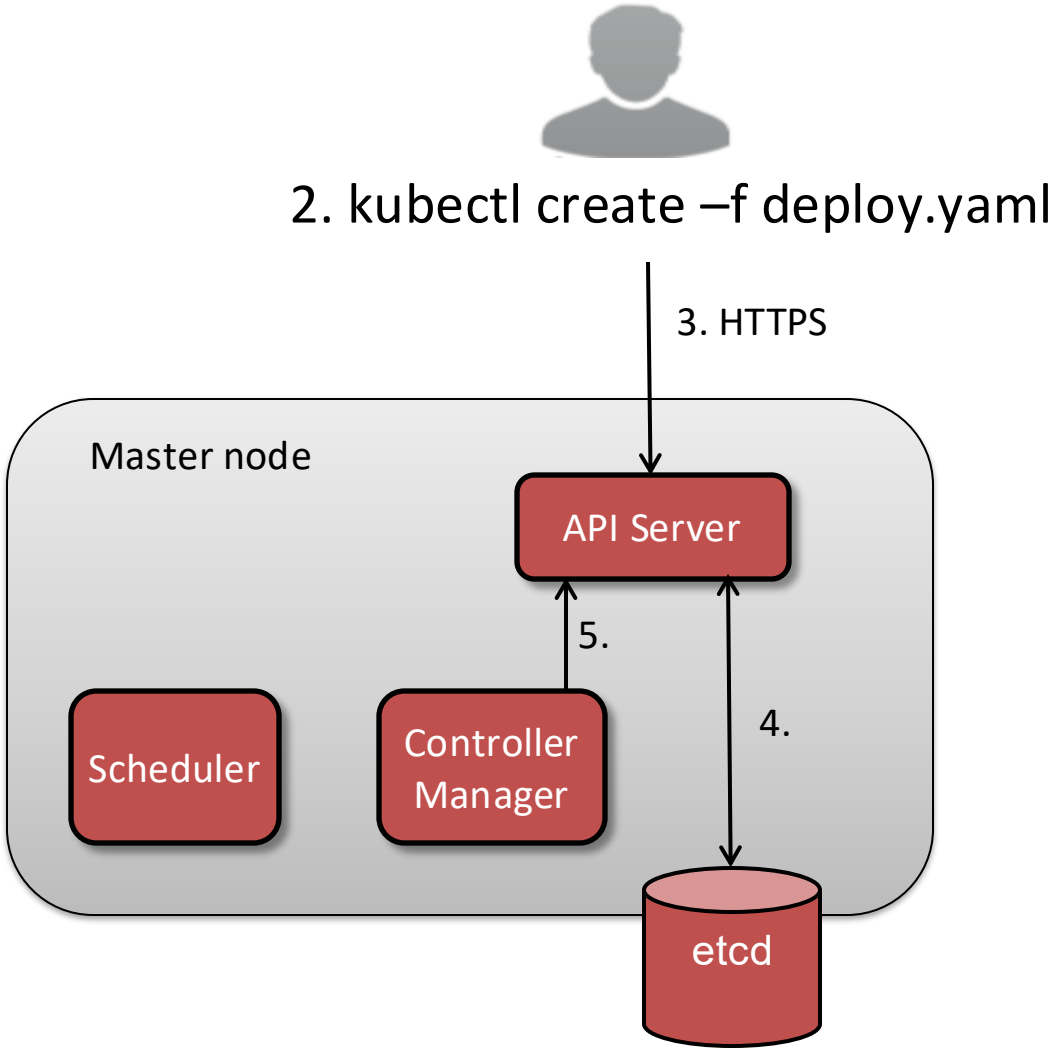
- Replica count can be changed to provide scaling on demand as needed
- If the node hosting a pod fails, the Kubernetes cluster will recreate the pod elsewhere to achieve the target number of replicas



Examining a Deployment Manifest File



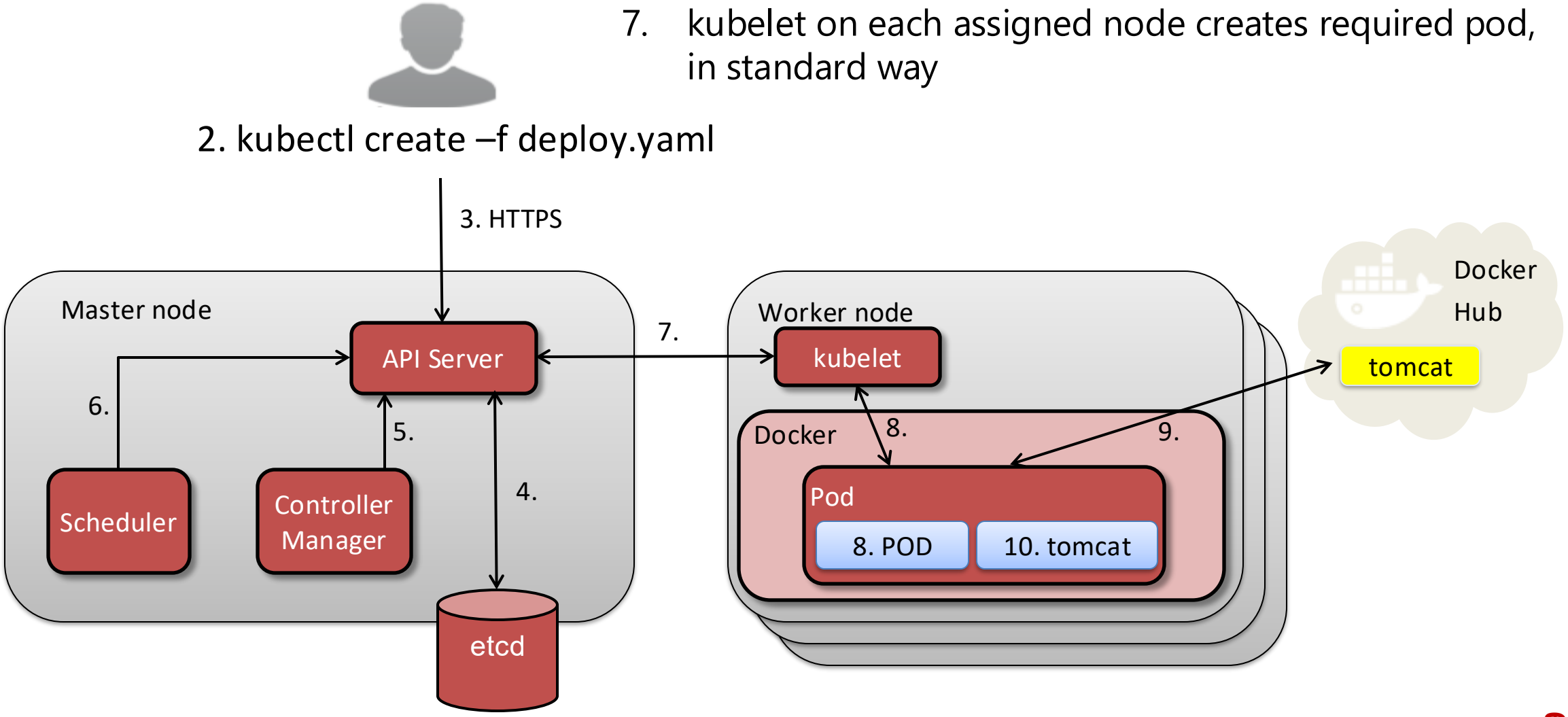
Deployment Creation Process



1. User writes a deployment manifest file
2. User requests creation of deployment from manifest via CLI
3. CLI tool marshals parameters into K8s RESTful API request (HTTP POST)
4. kube-apiserver creates new deployment object record in etcd, and new Replica Set object
5. kube-controller-manager sees new Replica Set and
   a. Evaluates state of existing vs. required replicas
   b. Submits pod creation requests to API to create required number of replicas

Deployment Creation Process

6. Scheduler sees new pod objects created and selects node assignments
7. kubelet on each assigned node creates required pod, in standard way



Deployment Strategies Overview

What is a Deployment Strategy?

Approaches to manage risks on updating Deployments

- On each Deployment update/change, all pods in the deployment will be deleted and recreated
- Recreation process can have service impacts, especially for large Deployments
- A Deployment strategy defines how this rebuild process is done, to minimize downtime due to application failures or malfunctions

Types of Deployment Strategies

Kubernetes supports two basic strategies, but users can also leverage multiple Deployments when applying changes

- Strategies for single Deployments
  - Recreate
  - RollingUpdate
- Strategic approaches using two Deployments with a Service
  - Canary deployments
  - Blue/Green deployments

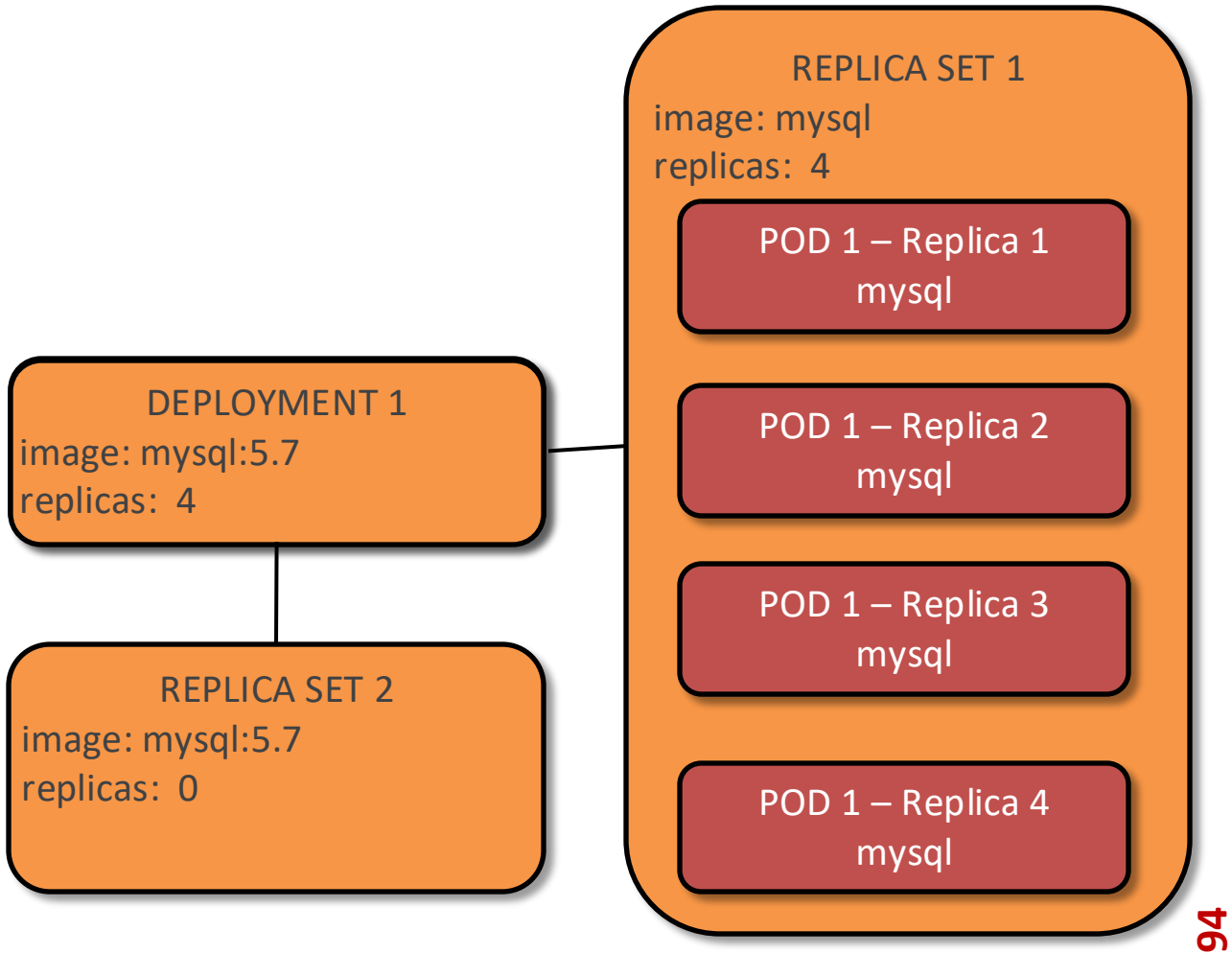
Each approach has a specific behavior and advantages/disadvantages.

Deployment Strategy: Recreate

Deployment Strategy: Recreate

Simplest strategy for deployments

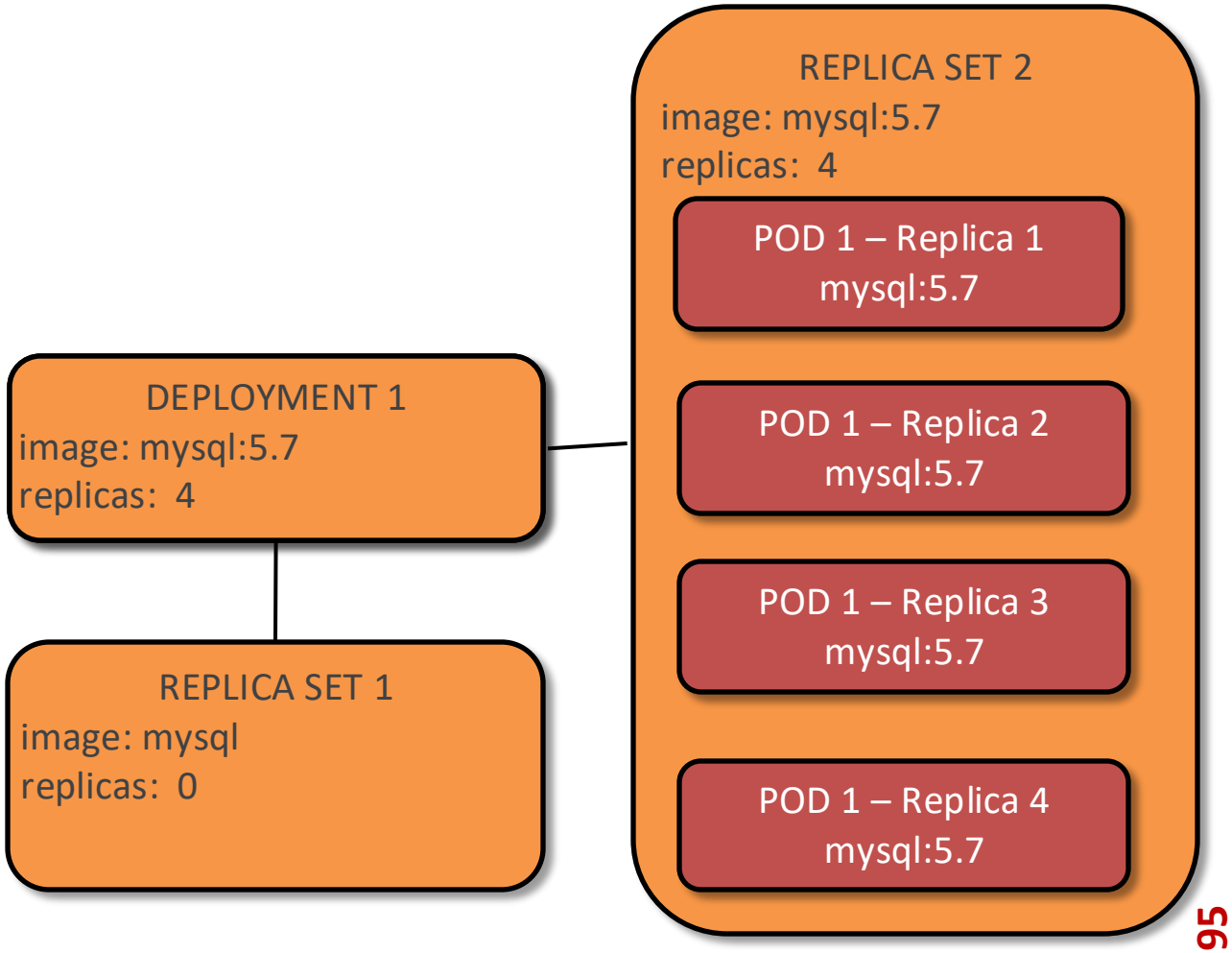
- When a change is made to a Deployment's spec, all Pods are removed and then recreated
  - Old Replica Set pods are killed
  - Then, new Replica Set starts pods
- May lead to downtime during the process while new pods are started



Deployment Strategy: Recreate

Simplest strategy for deployment updates

- When a change is made to a Deployment's template, all Pods are removed and then recreated
  - Old Replica Set pods are killed
  - New Replica Set starts pods
- May cause downtime due to delay between old pods terminating and new pods becoming available



Deployment Strategy: Recreate

Strategies are defined in the spec of a Deployment

- **strategy** parameter in Deployment spec sets the strategy to be used for updates
- If no parameter value is set, the default is **RollingUpdate**

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: tomcat-deployment
spec:
  replicas: 3
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        type: webserver
    spec:
      containers:
      - name: tomcat-container
        image: tomcat
        ports:
        - containerPort: 8080
```

Deployment Strategy: RollingUpdate

Deployment Strategy: RollingUpdate

RollingUpdate is DEFAULT strategy for Deployments

- When a change is made to the Deployment, the old Replica Set pods are scaled down as new pods are created by the new Replica Set
- A minimum number of running Pods is specified, so the Deployment will never be totally out of Pods to respond to service requests
- During the update process, the requested replica count may be temporarily exceeded

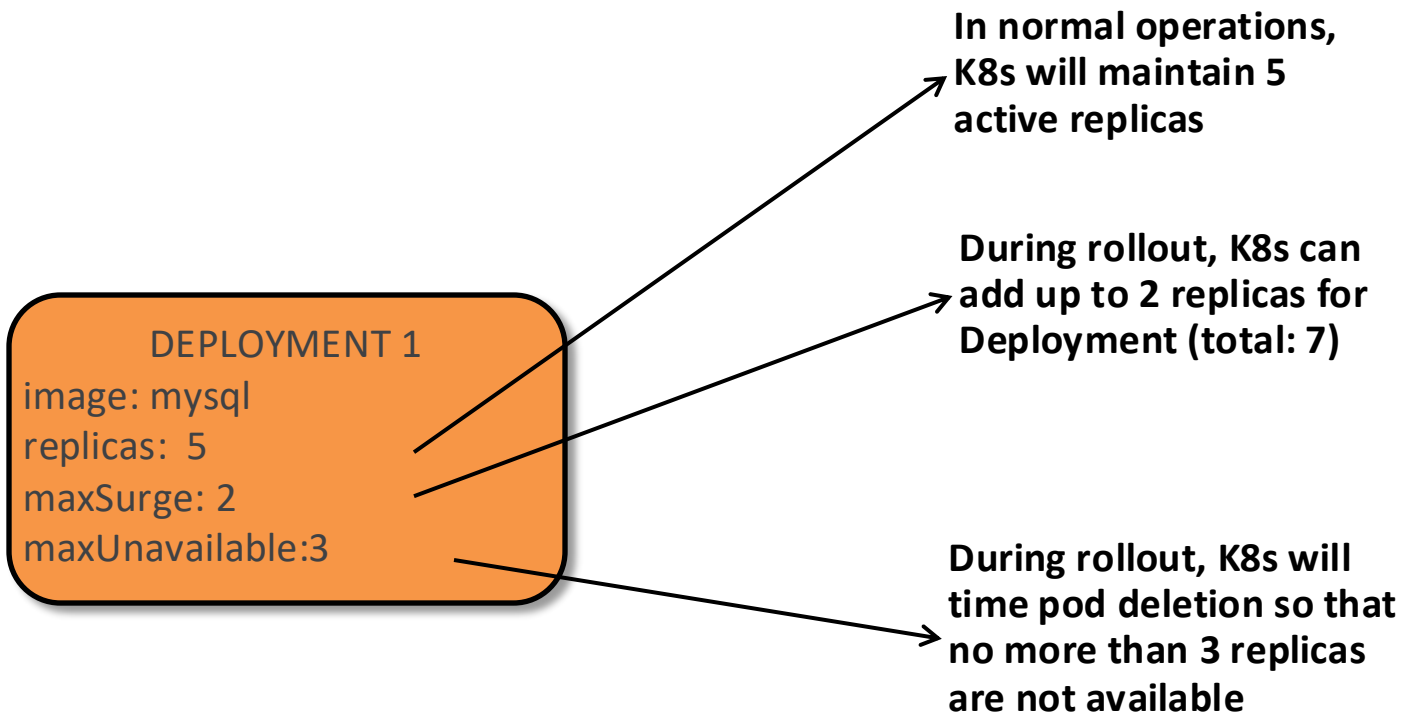
Deployment Strategy: RollingUpdate

Configure parameters to control the update process

```
...
metadata:
  name: tomcat-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable:10%
template:
    metadata:
      labels:
        type: webserver
...
```

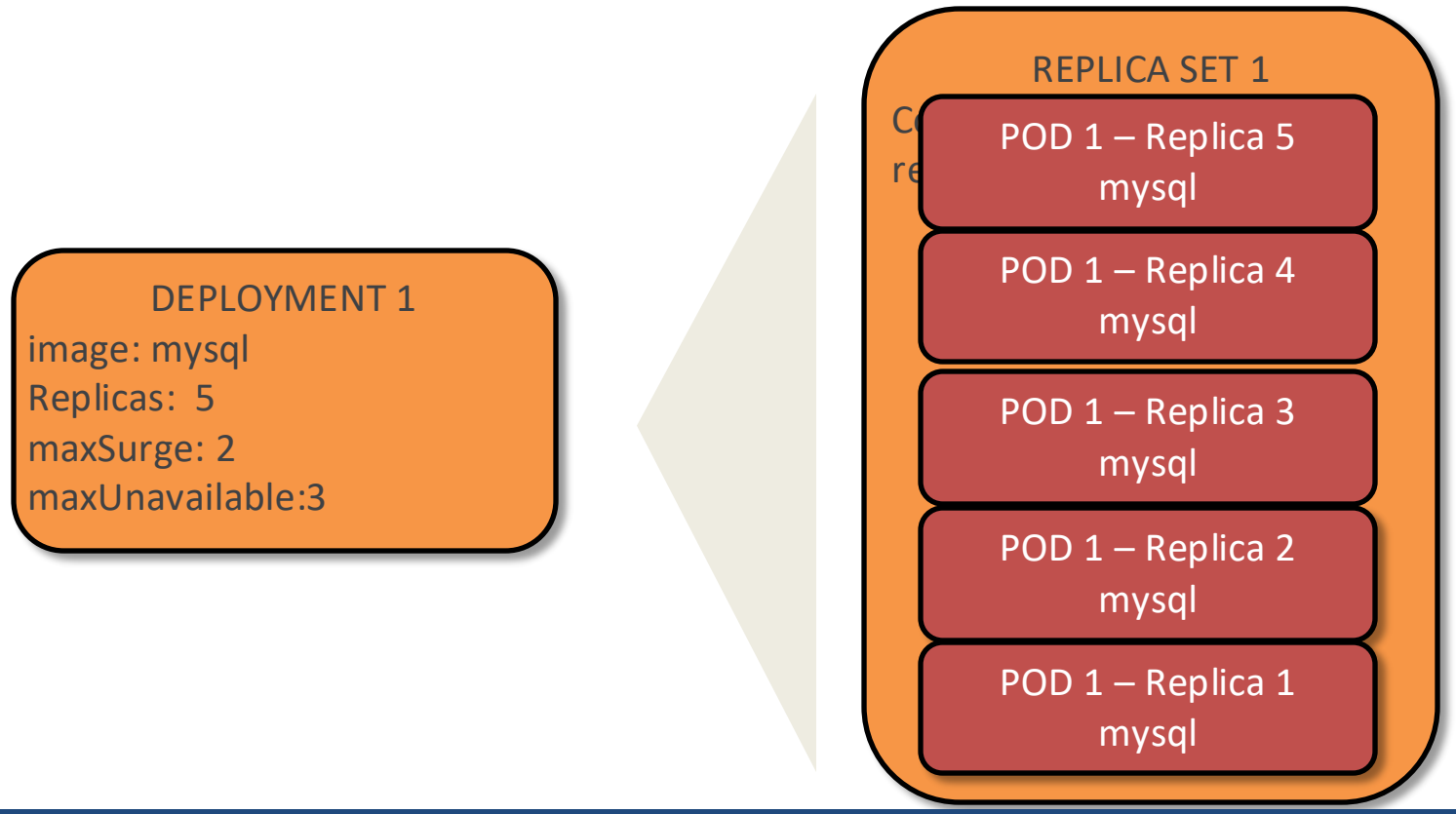
- **maxSurge**: number or percentage of additional Pods that can be created exceeding the replica count during update
  - Default value of 25%
- **maxUnavailable**: number of Pods that can be unavailable during the update
  - Default value of 25%

Deployment Strategy: RollingUpdate



Deployment Strategy: RollingUpdate

Initial State

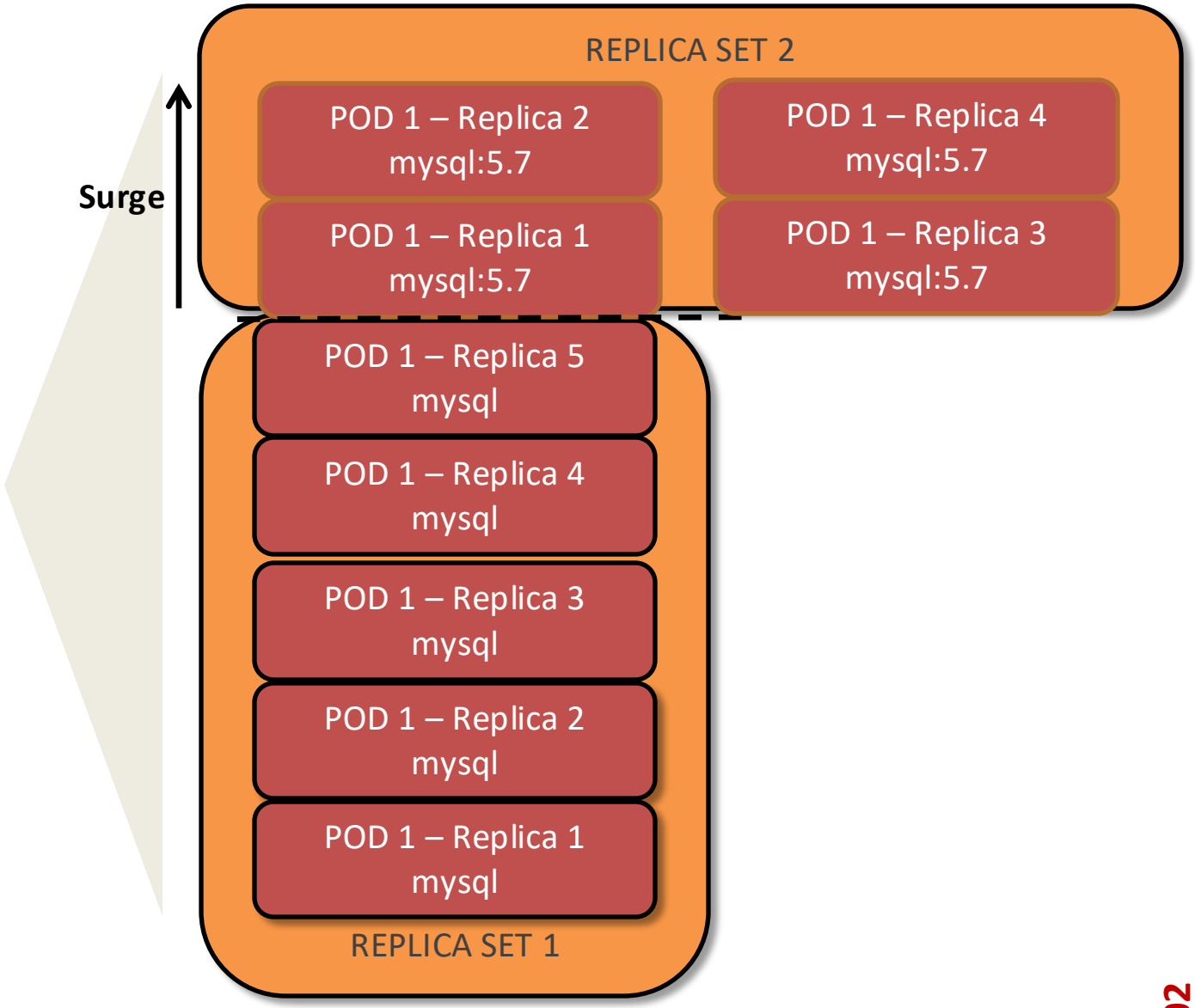



```
$ vi simpledeployment.yaml
...
    image: mysql:5.7
...
$ kubectl apply –f simpledeployment.yaml
```

Deployment Strategy: RollingUpdate

Rollout in progress

DEPLOYMENT 1
image: mysql:5.7
Replicas: 5
maxSurge: 2
maxUnavailable:3



- Initial surge of new pods on new Replica Set
- Original Replica Set scaled back as new RS scaled out

Deployment Strategy: RollingUpdate

Rollout complete

DEPLOYMENT 1
image: mysql:5.7
replicas: 5
maxSurge: 2
maxUnavailable:3

REPLICA SET 2
image: mysql:5.7
replicas: 5

- POD 1 – Replica 5 mysql:5.7
- POD 1 – Replica 4 mysql:5.7
- POD 1 – Replica 3 mysql:5.7
- POD 1 – Replica 2 mysql:5.7
- POD 1 – Replica 1 mysql:5.7

REPLICA SET 1
image: mysql
replicas: 0

- By default, old, inactive Replica Set saved – previous version of the Deployment

Updating Using Multiple Deployments

RollingUpdate using Multiple Deployments

Controlled testing of new versions in production

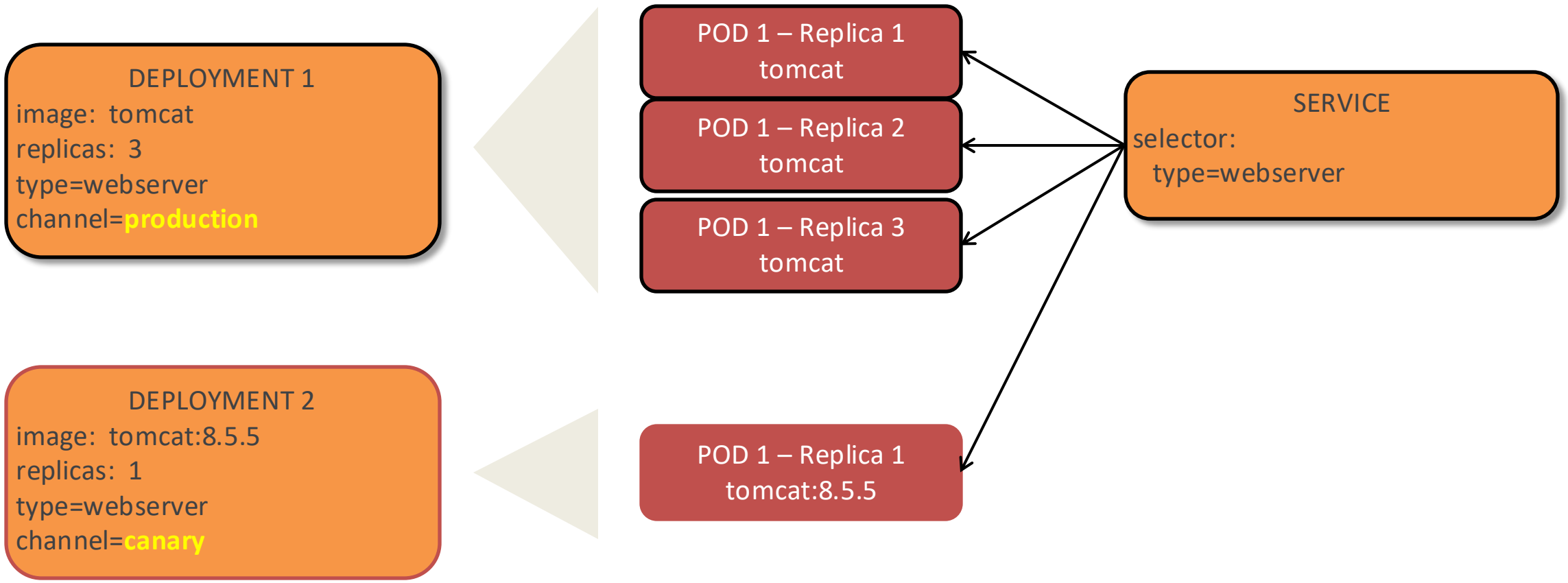
- Assume an application running as a Deployment, exposed as a Service
- To apply a new application version in production, a second Deployment can be used using labels in common with the first Deployment
  - Canary deployment allows for limited testing of new version in production
  - Blue/green deployment

Strategic Approach: Canary Deployment

Controlled testing of the update on production

- Consider a Service selecting pods from a Deployment of application pods
- In a canary deployment, a second Deployment (Canary Deployment) is created with pods for the new version, with labels matching the Service's selector
- Service directs some requests to pods on the Canary, allowing testing of changes in production
- If a malfunction is detected, it will only impact a small portion of the Pods and can be undone.

Strategic Approach: Canary Deployment



Strategic Approach: Canary Deployment

Decisions after running the canary Deployment in production

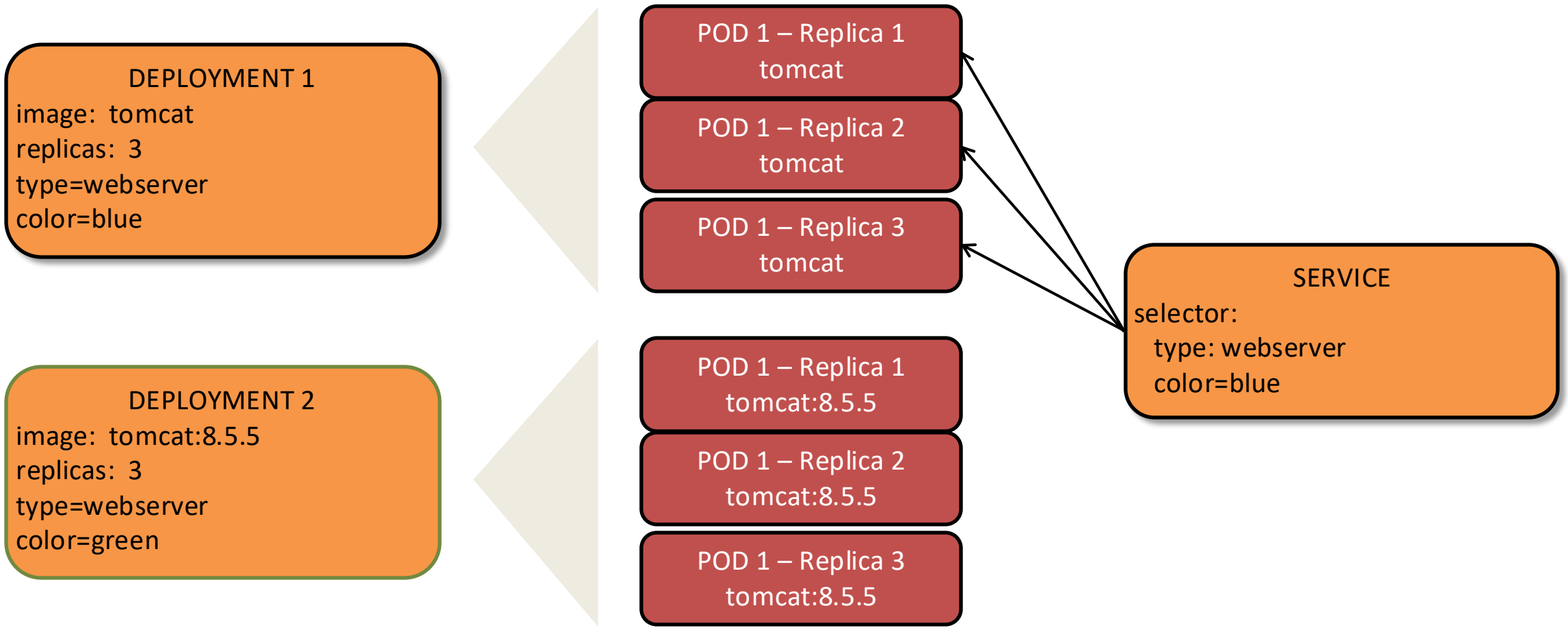
- If the application error rate is not increasing and Canary Deployment is stable:
  - The main Deployment can be updated to the newer version (using Rolling Update for example) and then the Canary can be discarded; OR
  - The Canary can be scaled up and reconfigured and the old Deployment can be discarded.
- If the test results in failure, the Canary deployment can be deleted

Strategic Approach: Blue/Green Deployment

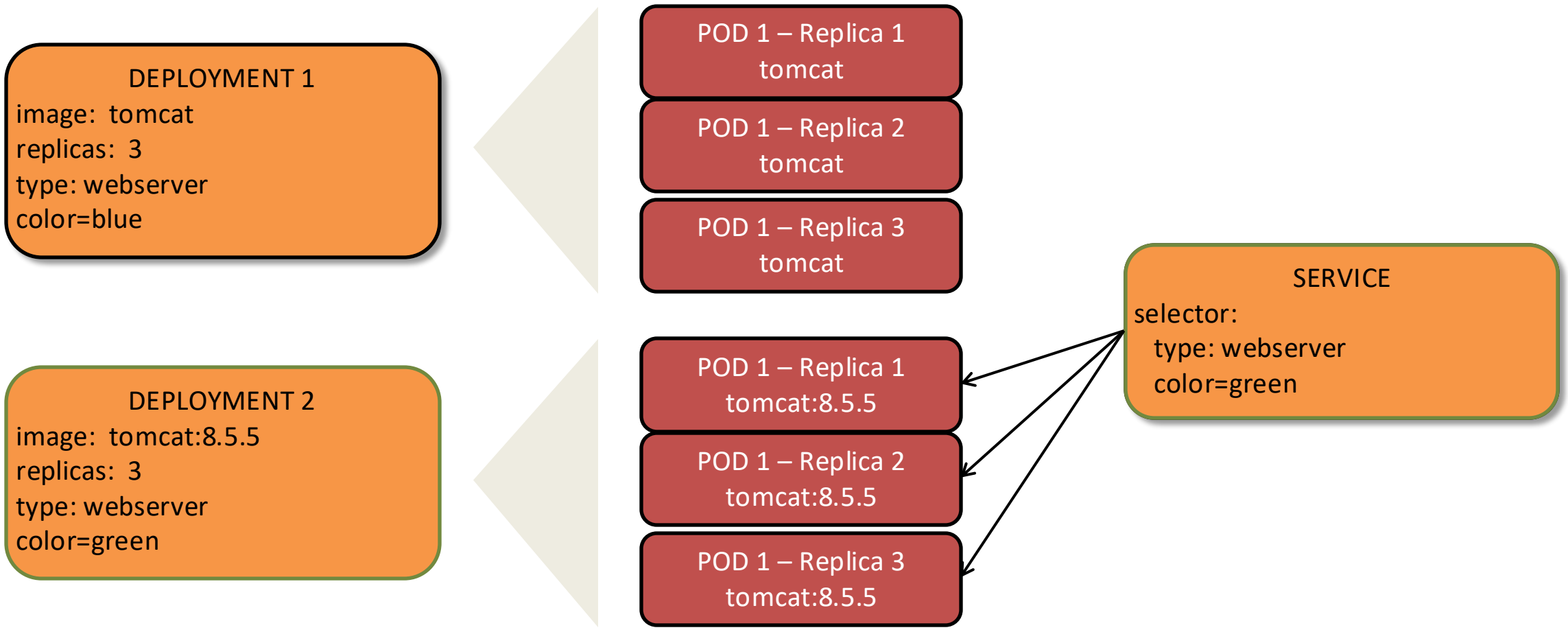
Complete environment switch from one version to another

- With a Blue/Green deployment, you create a new full-scale Deployment in addition to the current production Deployment
- Reconfiguring the pod label selector on the application's Service allows choice of directing requests to old Deployment or new Deployment
- Similar to effect of Replace strategy without application downtime

Strategic Approach: Blue/Green Deployment



Strategic Approach: Blue/Green Deployment



Lab: Deployments