



# Running and Securing Kubernetes Apps



## WORKFORCE DEVELOPMENT



# Logistics



- **Class Hours:**
- Instructor will provide class start and end times.
- Breaks throughout class

- **Lunch:**
- 1 hour 15 minutes
- Extra time for email, phone, or to take a walk



- **Telecommunication:**
- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students

- **Miscellaneous**
- Courseware
- Bathroom

# Course Objectives



By the end of the course you will be able to:

- State the function and purpose of Kubernetes
- Describe the Kubernetes architecture:
  - Cluster components
  - Pods, Services and Deployments
  - Expose applications with Services
- Implement ConfigMaps and Secrets
- Use Helm to build and deploy containerized applications
- Implement Istio
  - Traffic management
  - Secure microservices
- Implement Role-Based Access Controls
- Manage Stateful applications
- More!

# Introductions

*Hello!*

- Name
- Job Role
- Your experience with (scale 1 - 5)
  - Docker/Containers
  - Kubernetes
  - Istio/Service Mesh
- Expectations for the course (please be specific)

# Data Center Evolution

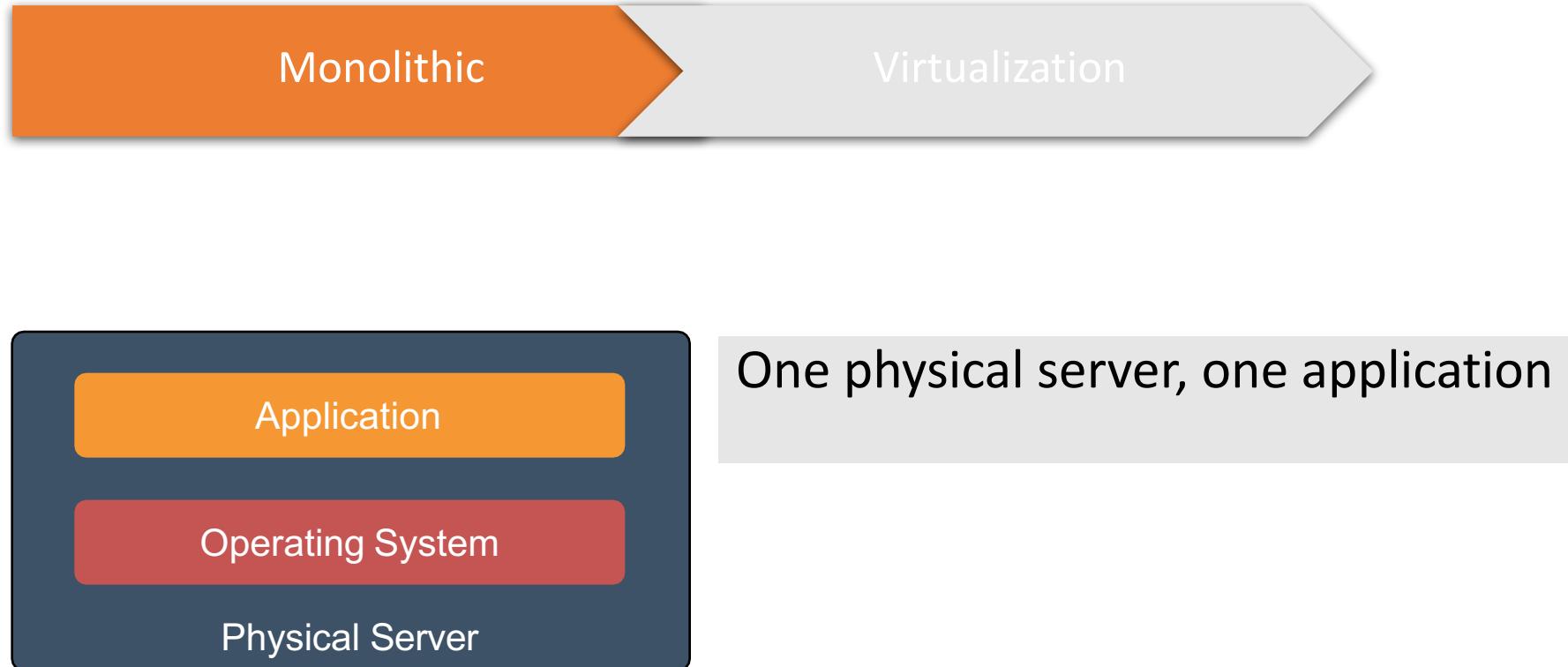


# Monolithic

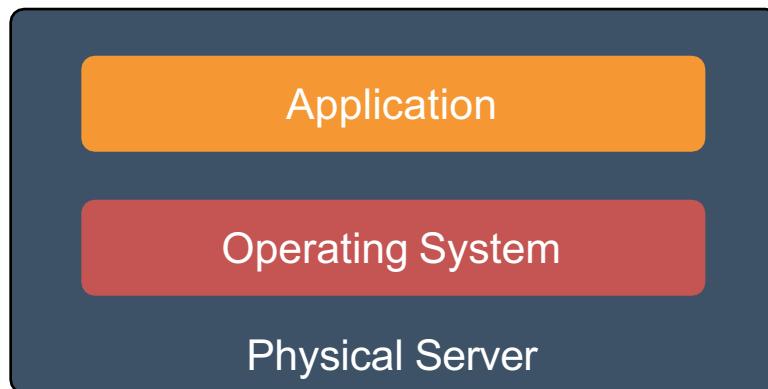
Monolithic

Virtualization

# Monolithic Server Architecture



# Monolithic Server Architecture



One physical server, one application

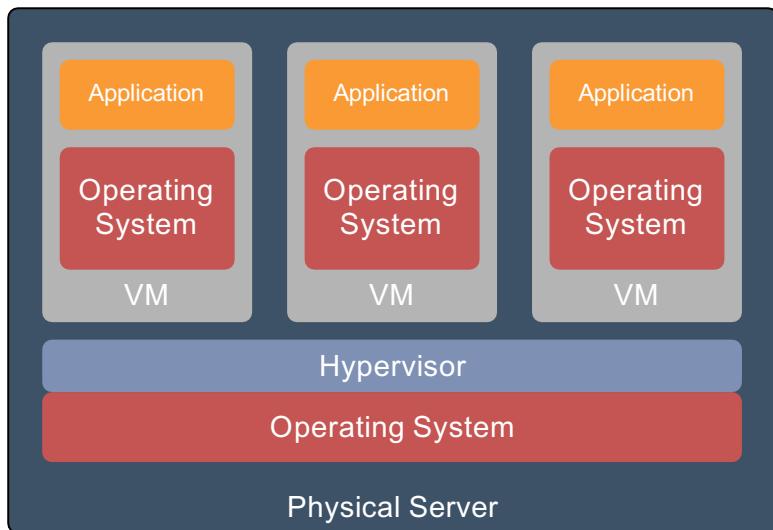
## Problems

- Slow deployment times
- Cost
- Wasted resources
- Difficult to scale
- Difficult to migrate

# Virtualized



# Virtualized Infrastructure

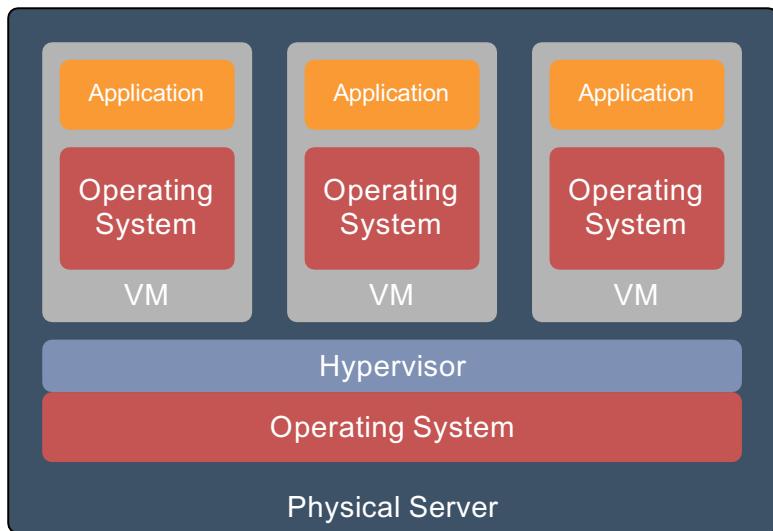


One physical server, multiple applications

# Discussion

What are some of the advantages and disadvantages of Virtual Machines?

# Virtualized Infrastructure - Advantages

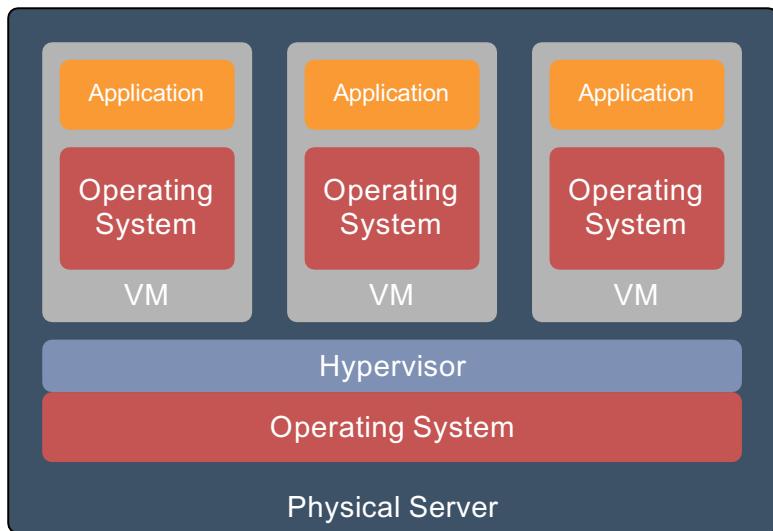


One physical server, multiple applications

## Advantages

- Better resource pooling
- Easier to Scale
- Enables Cloud/IaaS
  - Rapid elasticity
  - Pay as you go model

# Virtualized Infrastructure - Limitations



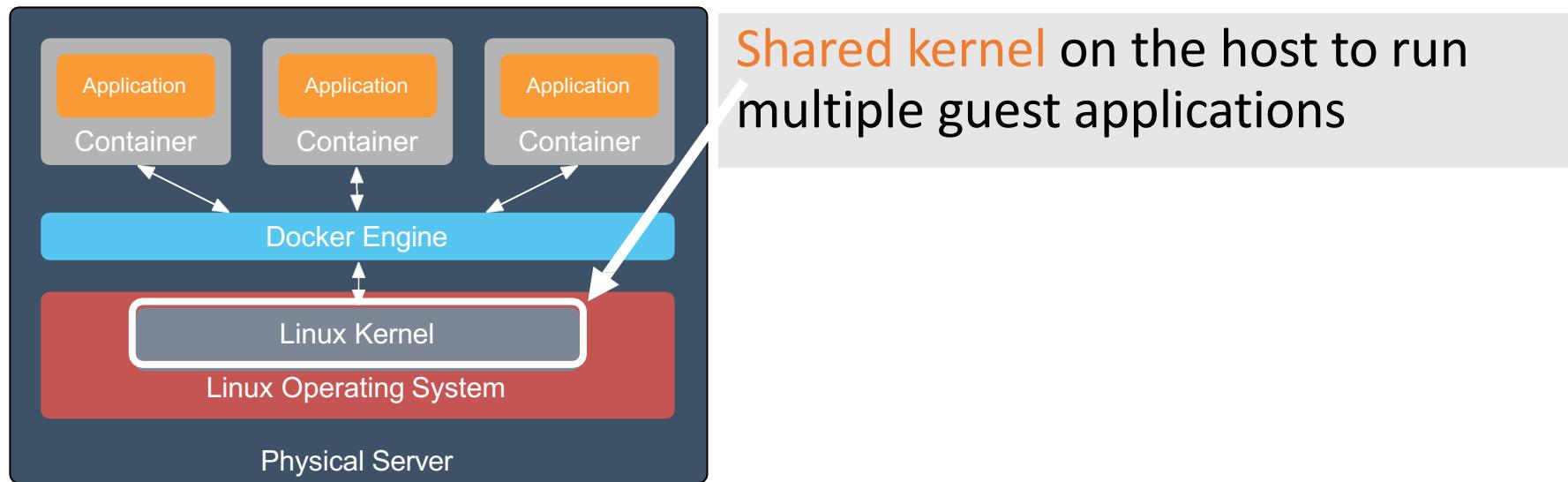
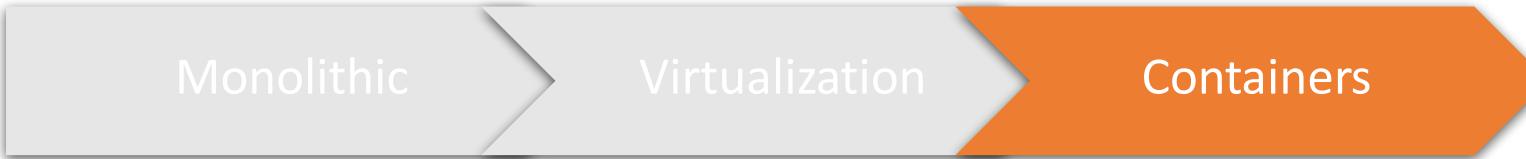
One physical server, multiple applications

## Limitations

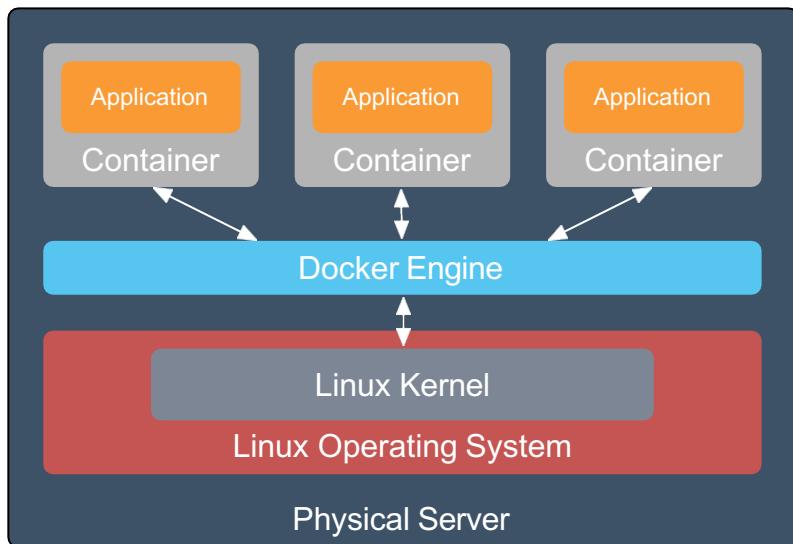
- Each VM requires:
  - CPU allocation
  - Storage
  - RAM
  - Guest Operating System
- More VMs, more wasted resources
- Application portability not guaranteed

# Containers

# Containers



# Containers - Advantages

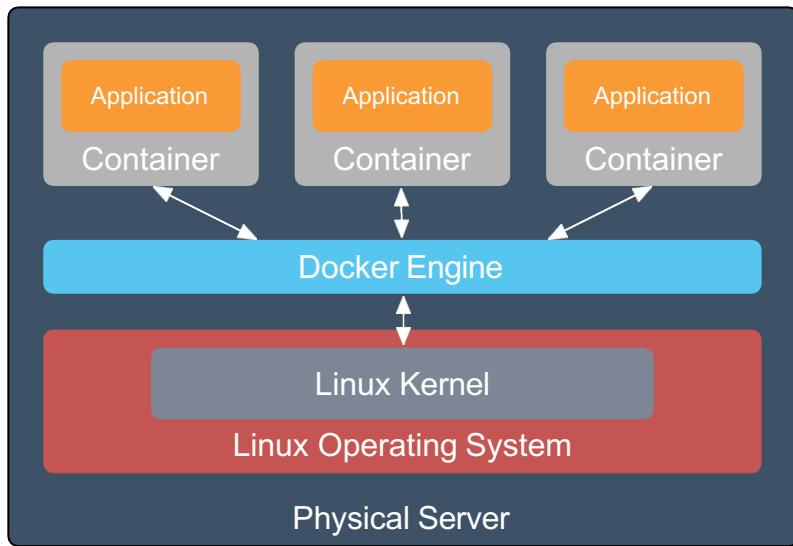
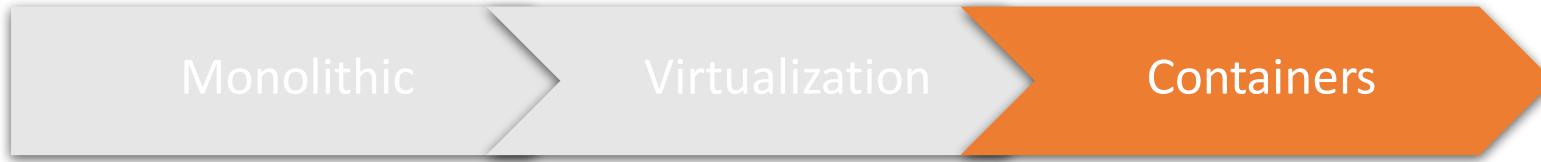


**Shared kernel** on the host to run multiple guest applications

## Advantages over VMs

- Containers are more lightweight
- No need to install a guest Operating System
- Less CPU, RAM, storage overhead
- More containers per machine
- Greater portability

# Containers - Challenges



**Shared kernel** on the host to run multiple guest applications

## Container Challenges

- Early Docker focused on single-node operations
- Up to user to cluster Docker hosts and manage deployment of containers on cluster
- User solves for automatic scale out of applications
- User solves for service discovery between application components (microservices)

# Container – Concept of Operations

# Container based virtualization

*Uses the kernel on the host operating system to run multiple guest instances*

- Each guest instance is a container
- Each container has its own

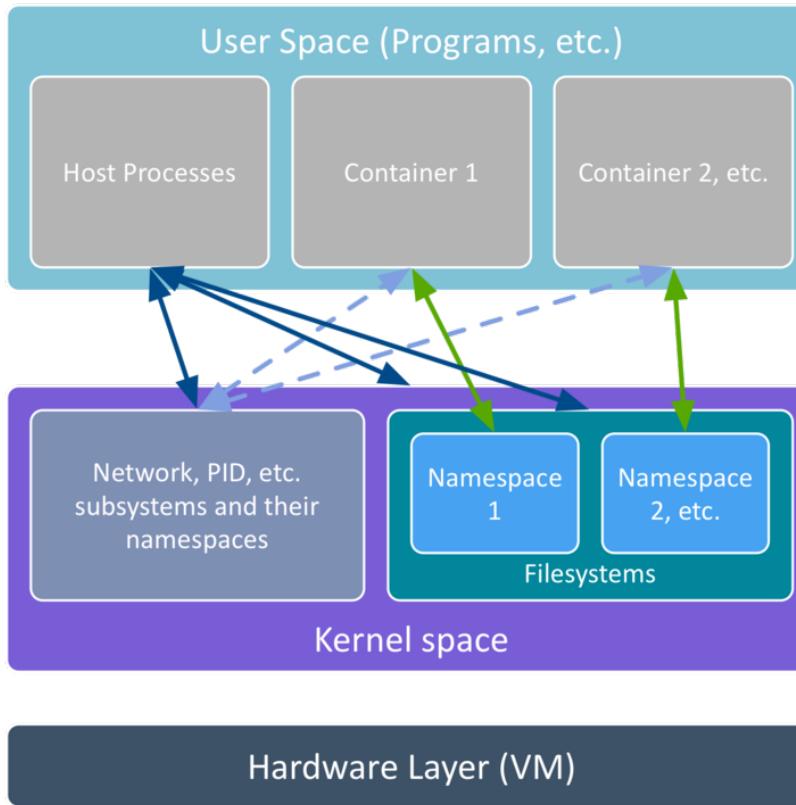
- Root filesystem
- Processes
- Memory
- Devices
- Network Ports



# Isolation with Namespaces

*Namespaces - Limits what a container can see (and therefore use)*

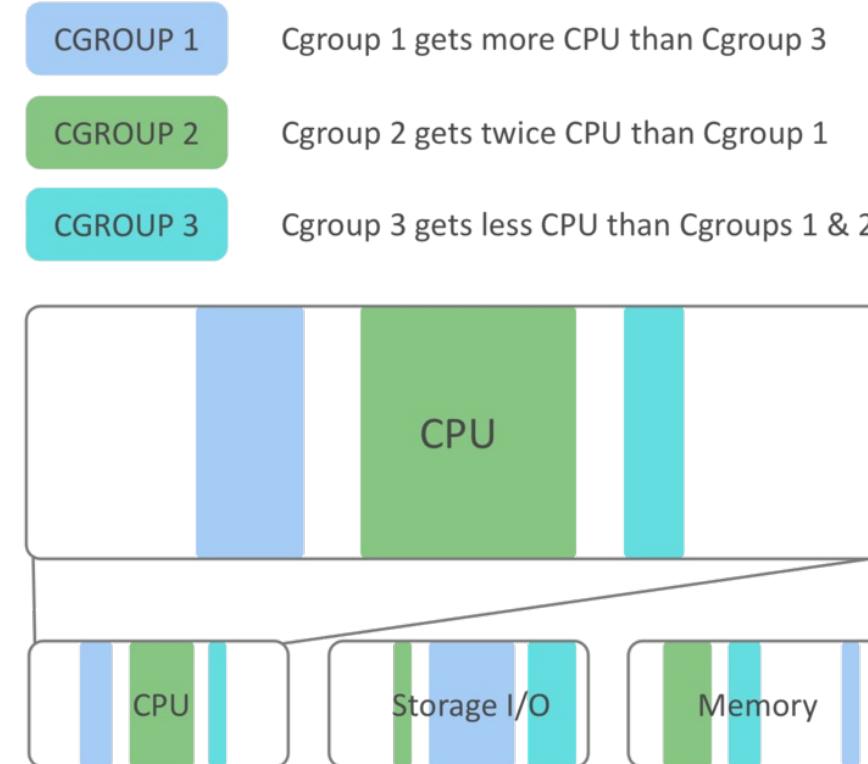
- Namespace wrap a global system resource in an abstraction layer
- Processes running in that namespace think they have their own, isolated resource
- Isolation includes:
  - Network stack
  - Process space
  - Filesystem mount points
  - etc.



# Isolation with Control group (Cgroups)

*Cgroups - Limits what a container can use*

- Resource metering and limiting
  - CPU
  - MEM
  - Block/I/O
  - Network
- Device node (`/dev/*`) access control



# Container Use Cases

# DevOps



Developers

Focus on applications inside the container

---



Operations

Focus on orchestrating and maintaining  
containers in production

# Container use-cases

## Development

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

# Container use-cases

## Development

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

## Test Environments

- Same container that developers run is the container that runs in test lab and production – includes all dependencies
- Well formed API allows for automated building and testing of new containers

# Container use-cases

## Development

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

## Test Environments

- Same container that developers run is the container that runs in test lab and production – includes all dependencies
- Well formed API allows for automated building and testing of new containers

## Microservices

- Design applications as suites of services, each written in the best language for the task
- Better resource allocation
- One container per microservice vs. one VM per microservice
- Can define all interdependencies of services with templates



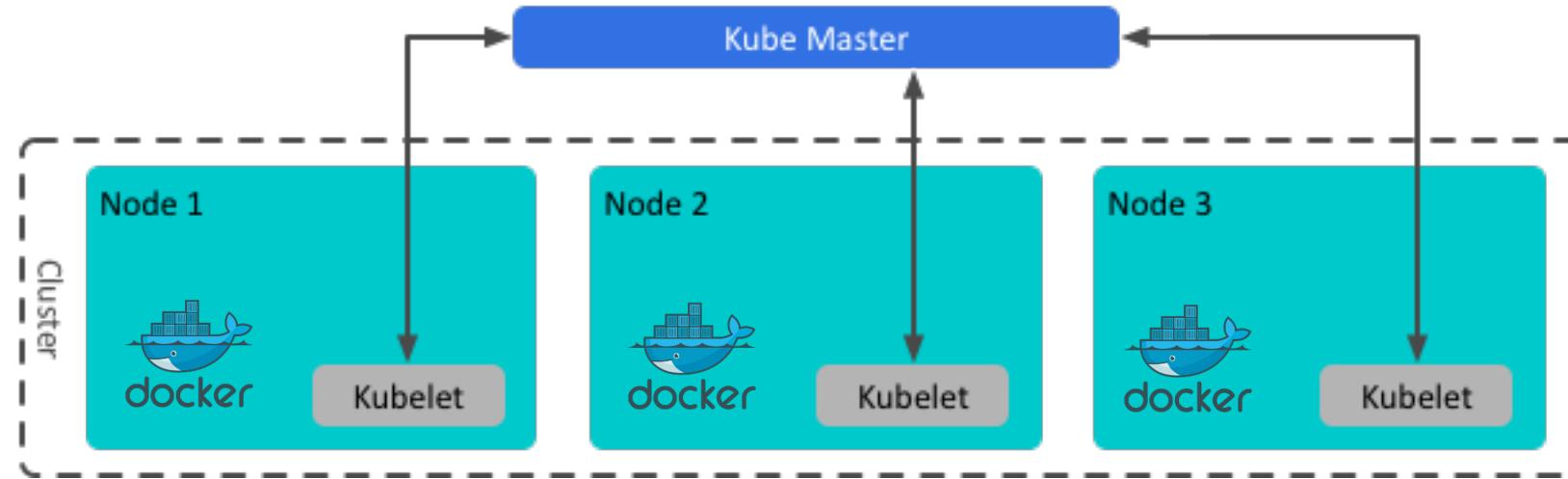
# Questions

# Kubernetes – Containers at scale



# Kubernetes: Containers at Scale

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



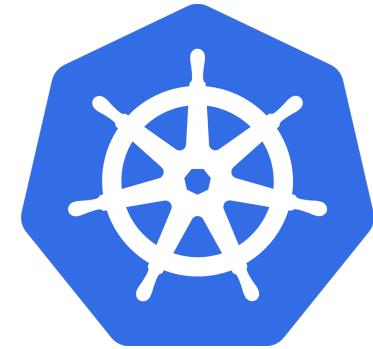
- Kubernetes resource objects provide key application management features, including
  - App elasticity and self-healing
  - Naming and discovery
  - Rolling updates
  - Request load balancing
  - Application health checking
  - Log access and resource monitoring

# 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
- All major Google services run on Borg
- Other cluster management frameworks, like Apache Mesos, inspired by Borg



Star Trek Borg Cube  
A hegemonizing swarm



Original project name:  
Seven of Nine  
(the friendly Borg)

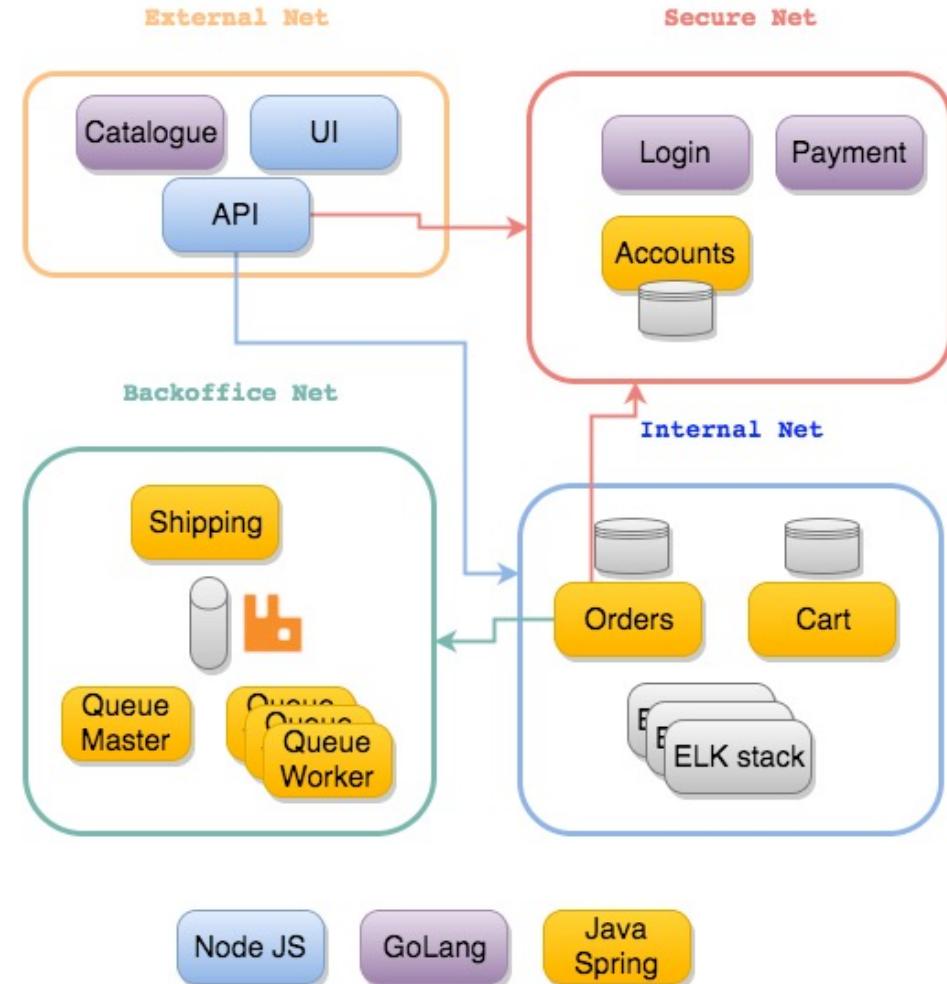
# Kubernetes Use Cases



# Kubernetes: Cloud-Native Application Deployment

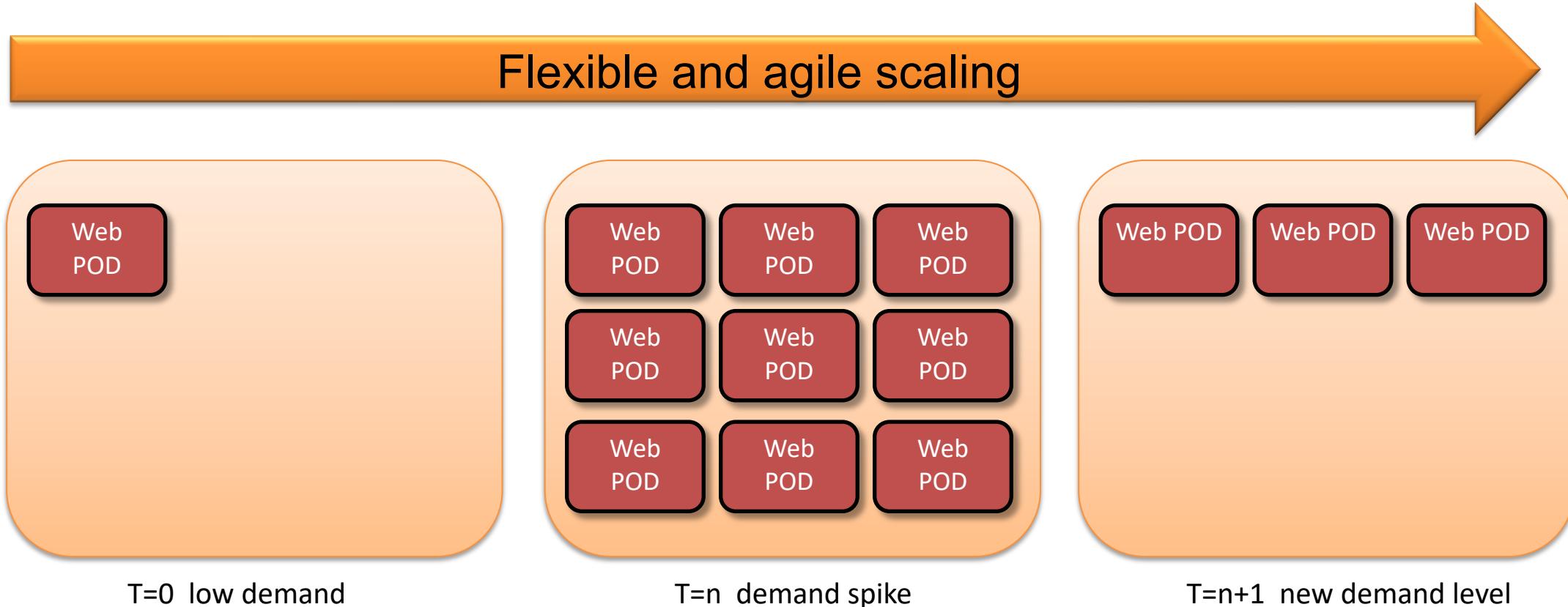
*Cloud-native applications, aka microservice-based or 12-factor apps*

- 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

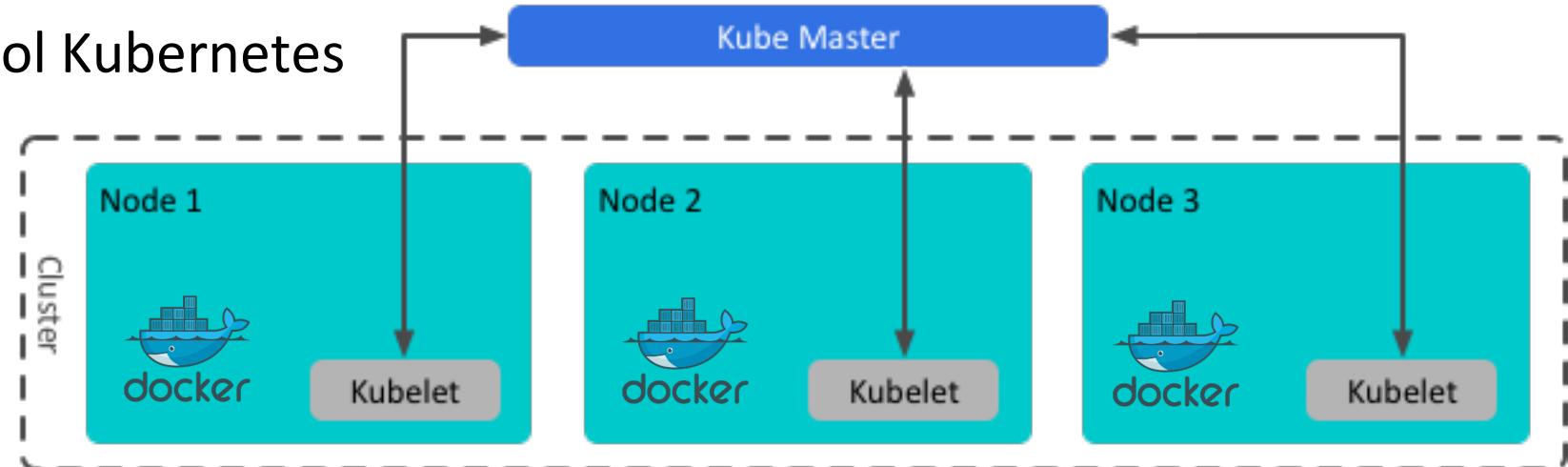


# Overview of Kubernetes Clusters



# 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 containerd hosts
- Kubernetes runs well in both private and public IaaS environments
- Users and admins control Kubernetes resources via REST API



# Kubernetes Components: Leader

## Main Components:

### Leader Node:

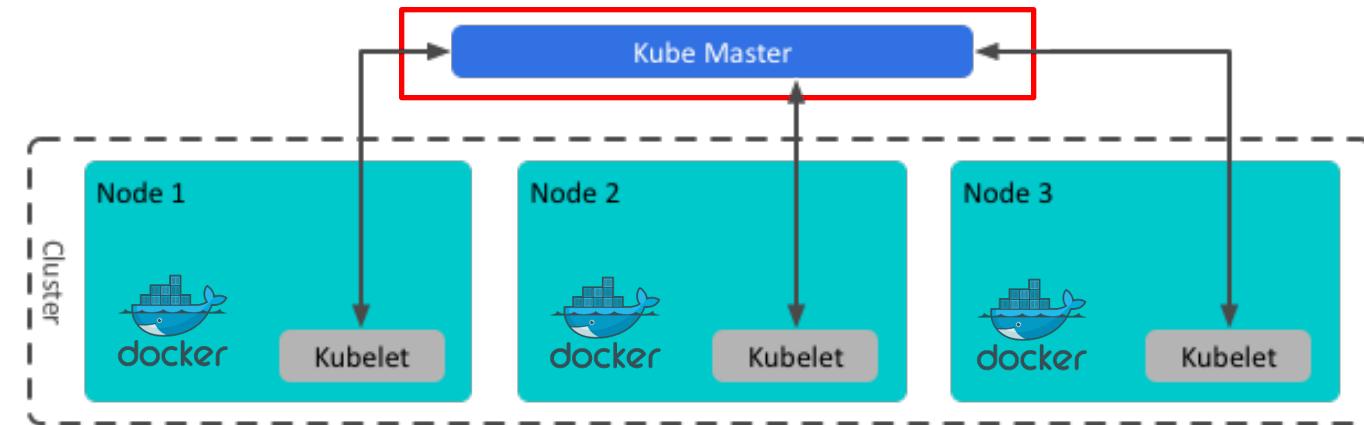
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

### Worker Nodes:

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

### Clusters:

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



# Kubernetes Components: Nodes

## Main Components:

### Leader Node:

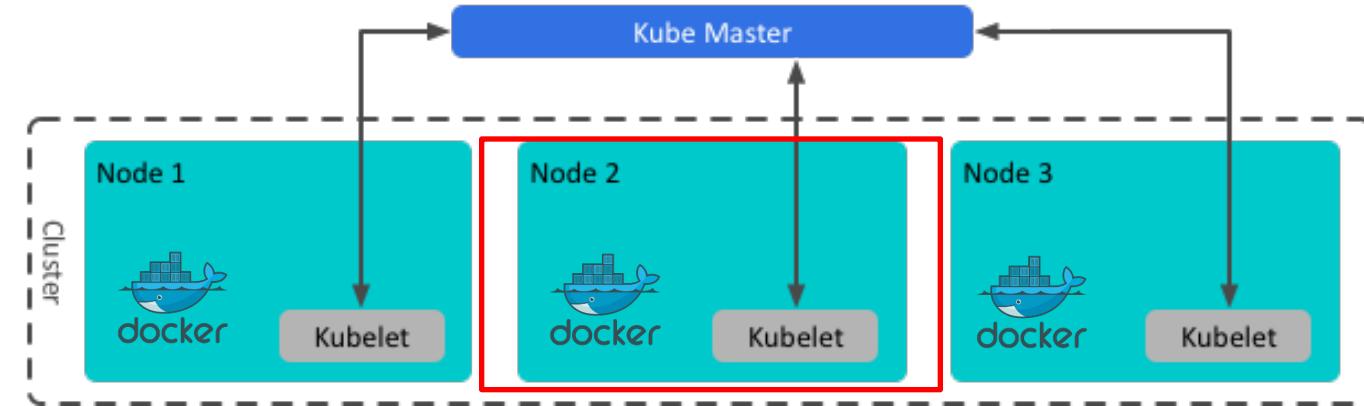
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

### Member Nodes:

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

### Clusters:

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



# Kubernetes Cluster Components

## Main Components:

### Leader Node:

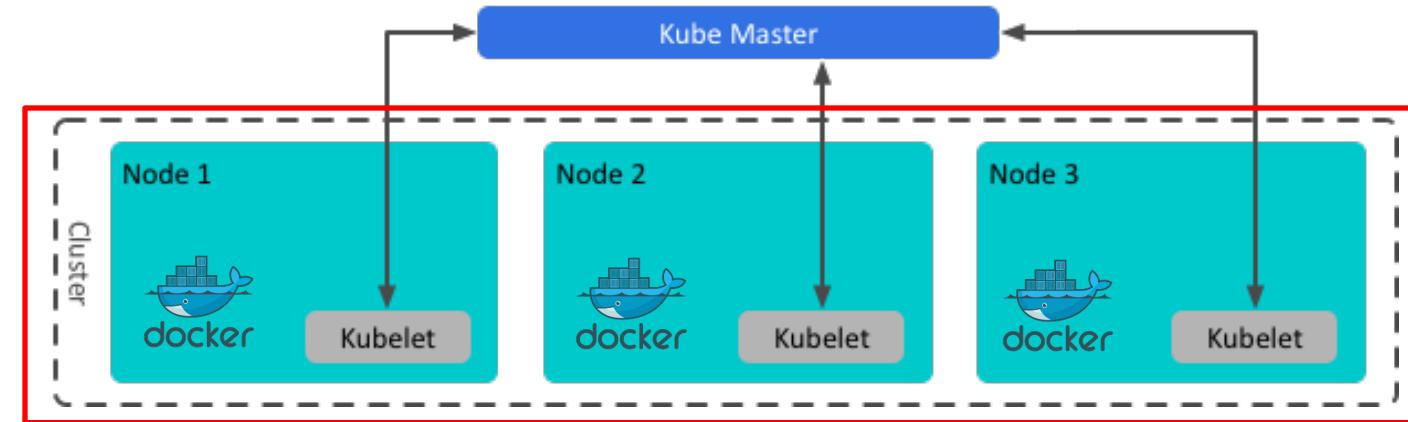
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

### Worker Nodes:

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

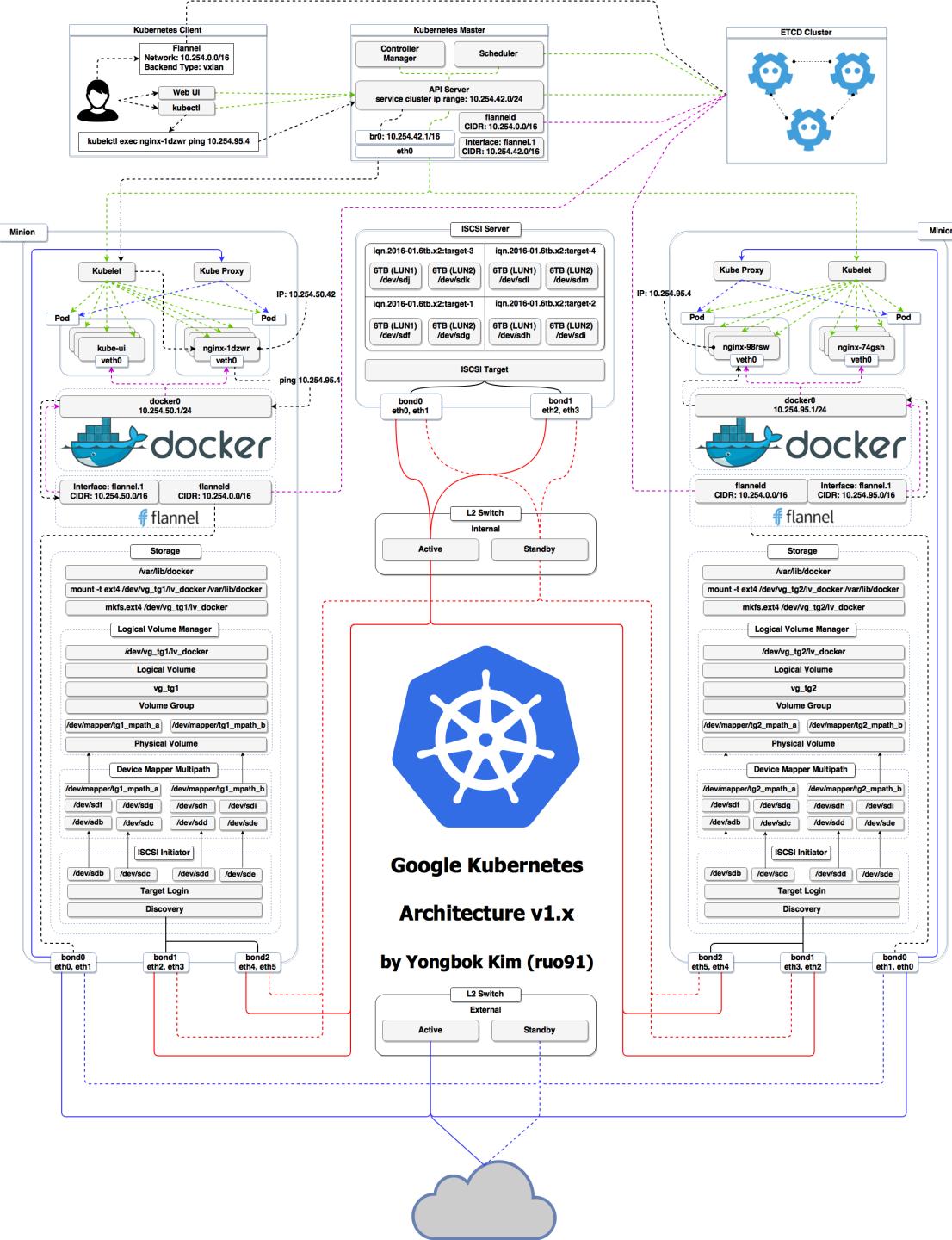
### Clusters:

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



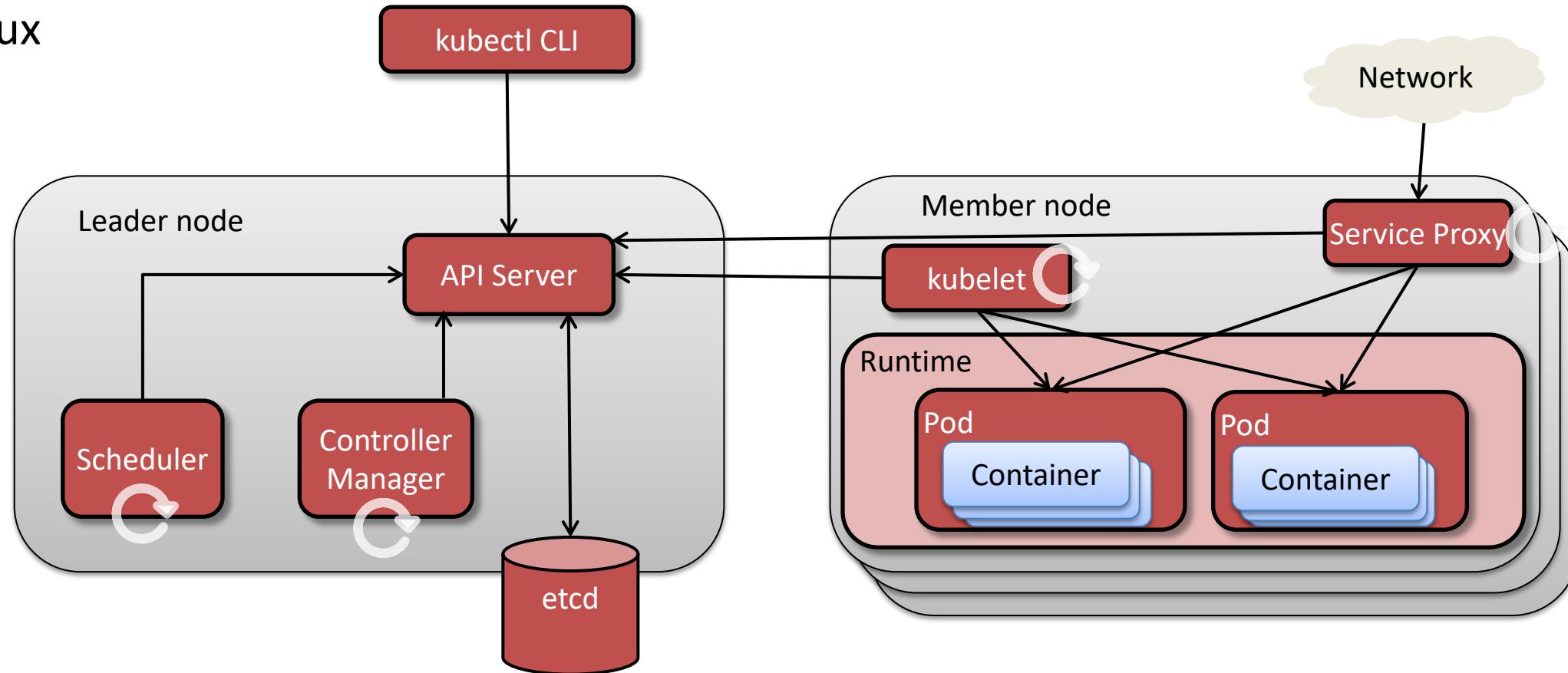
# Kubernetes Architecture





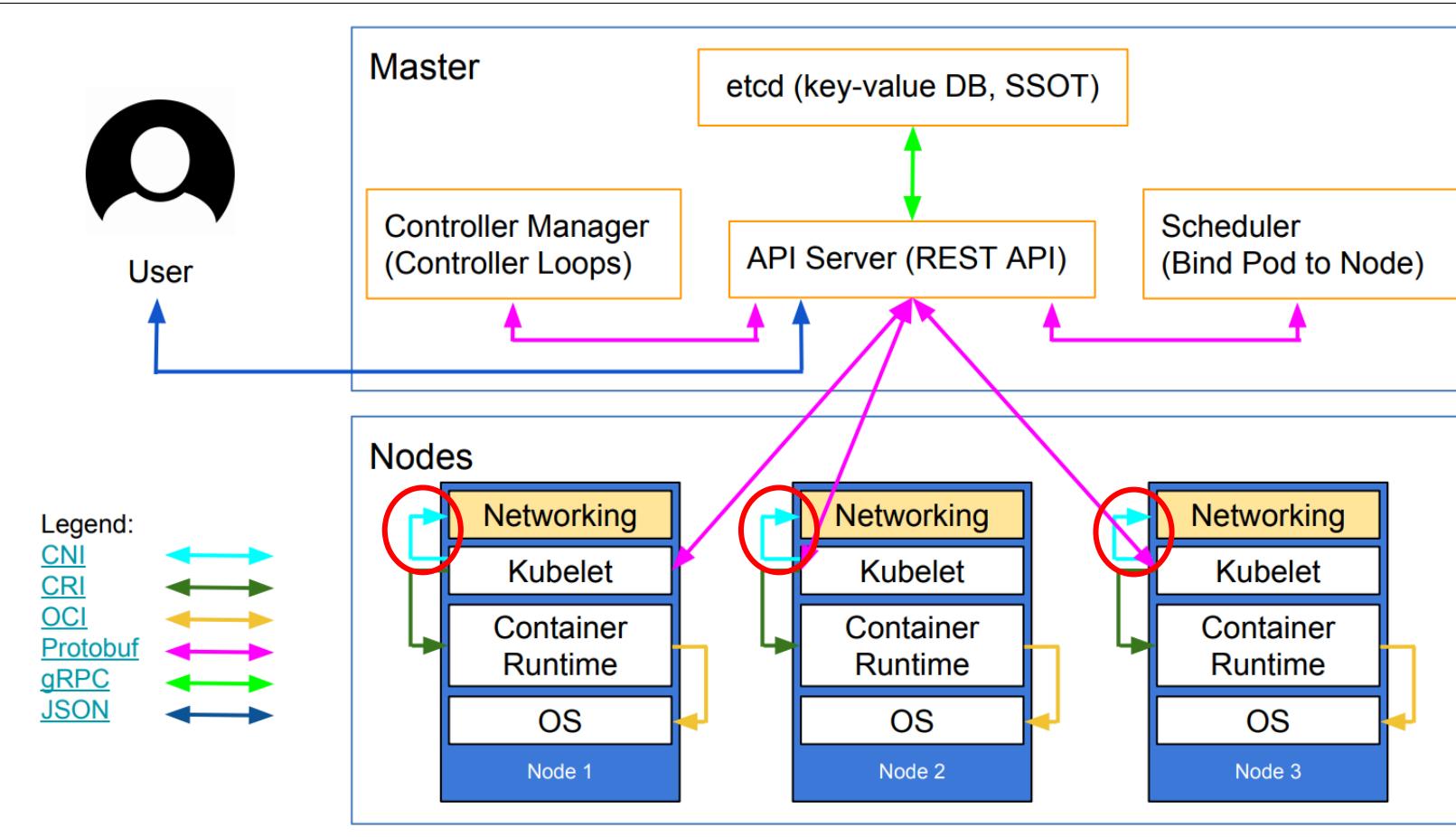
# Kubernetes Cluster Architecture

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

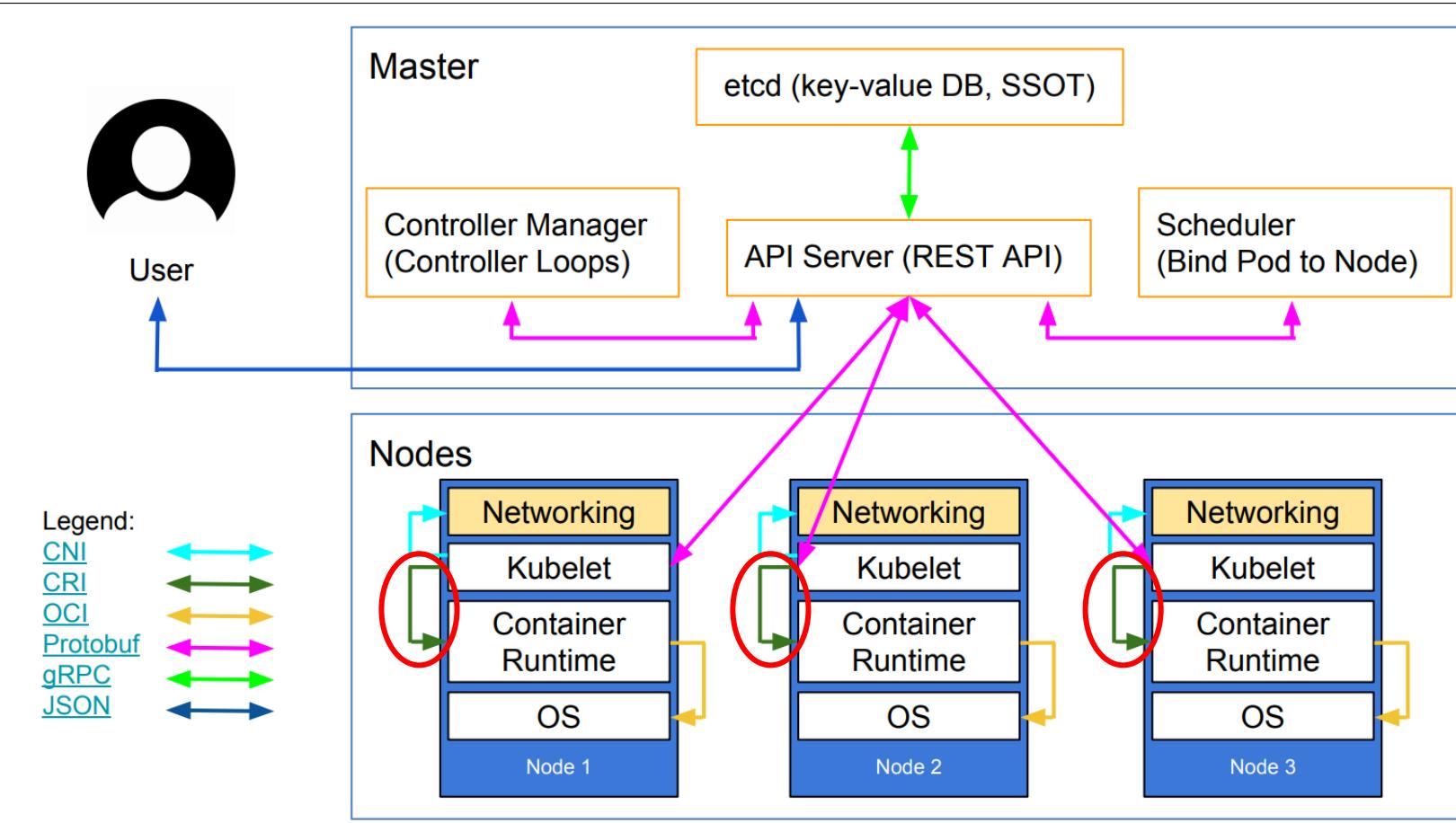


# Components Communication

- CNI = Container Network Interface

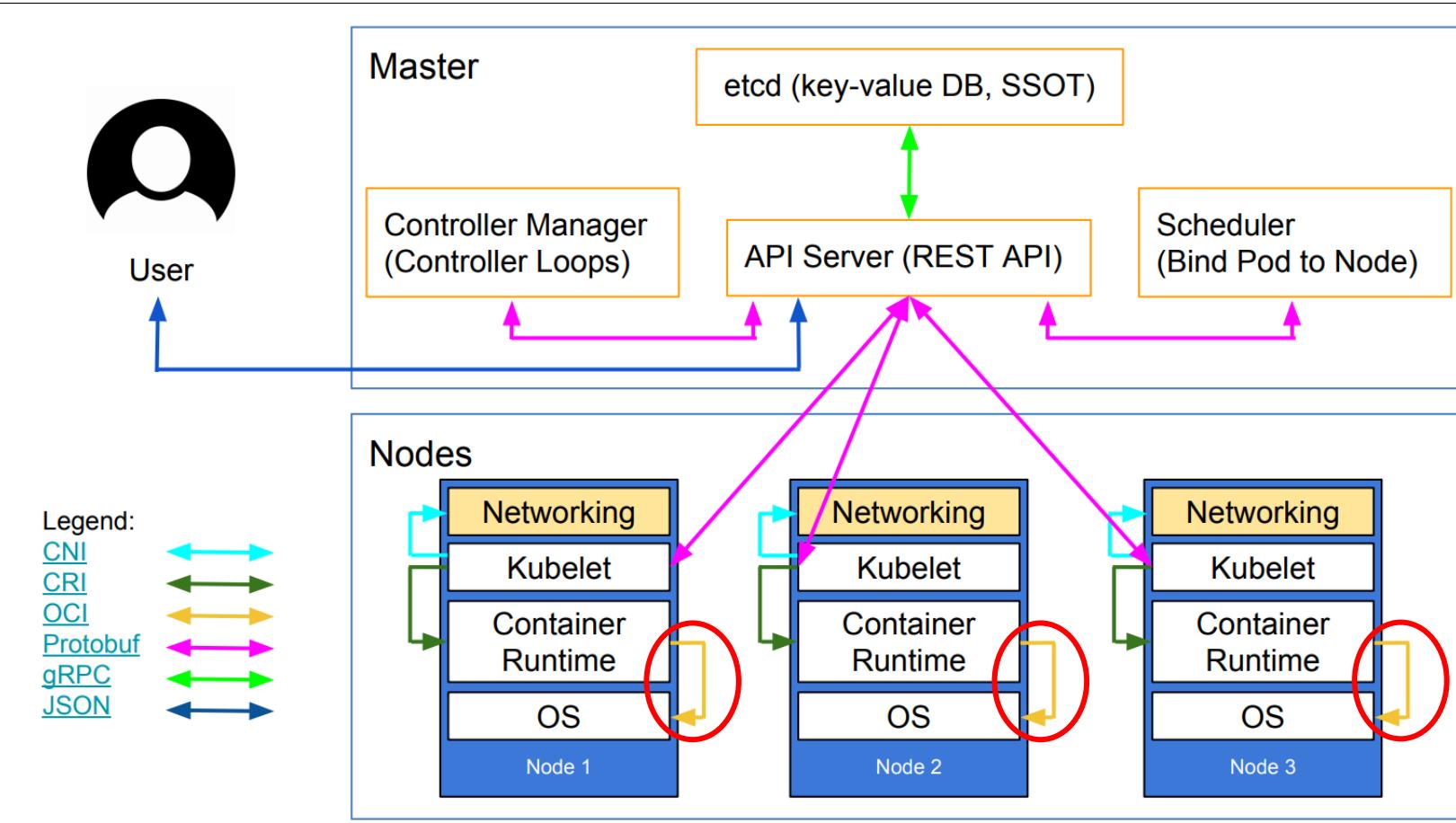


# Components Communication



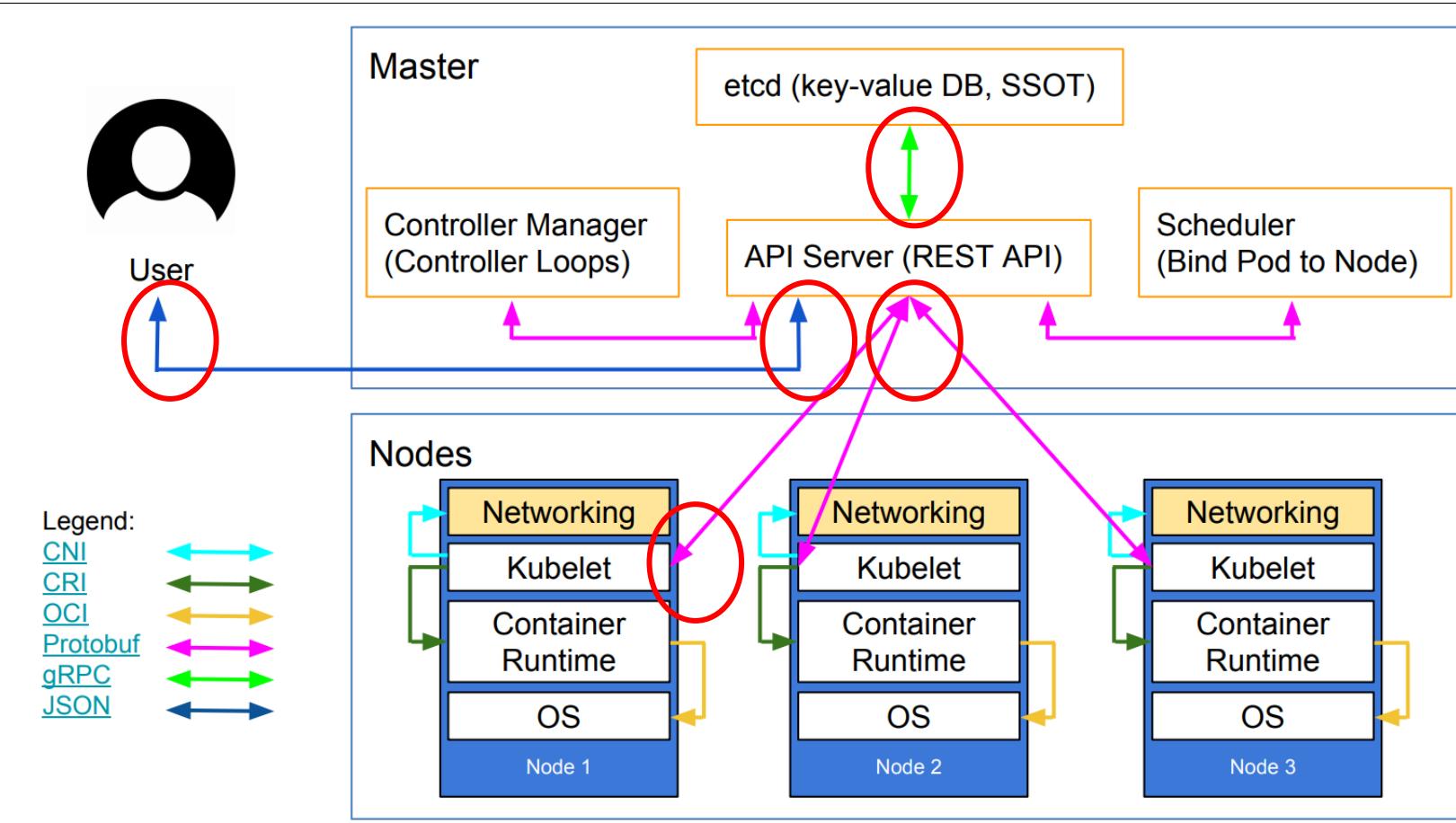
- CNI = Container Network Interface
- CRI = Container Runtime Interface
  - Docker, CRI-O, Containerd

# Components Communication



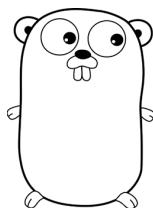
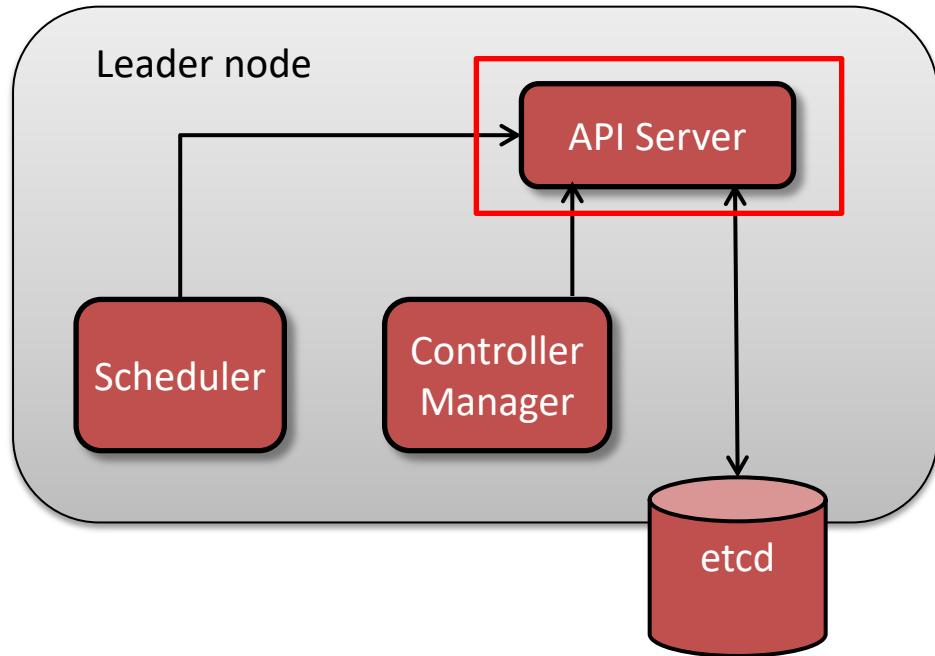
- CNI = Container Network Interface
- CRI = Container Runtime Interface
  - Docker, CRI-O, Containerd
- OCI = Open Container Initiative

# Components Communication



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

# Kubernetes Leader Node Components



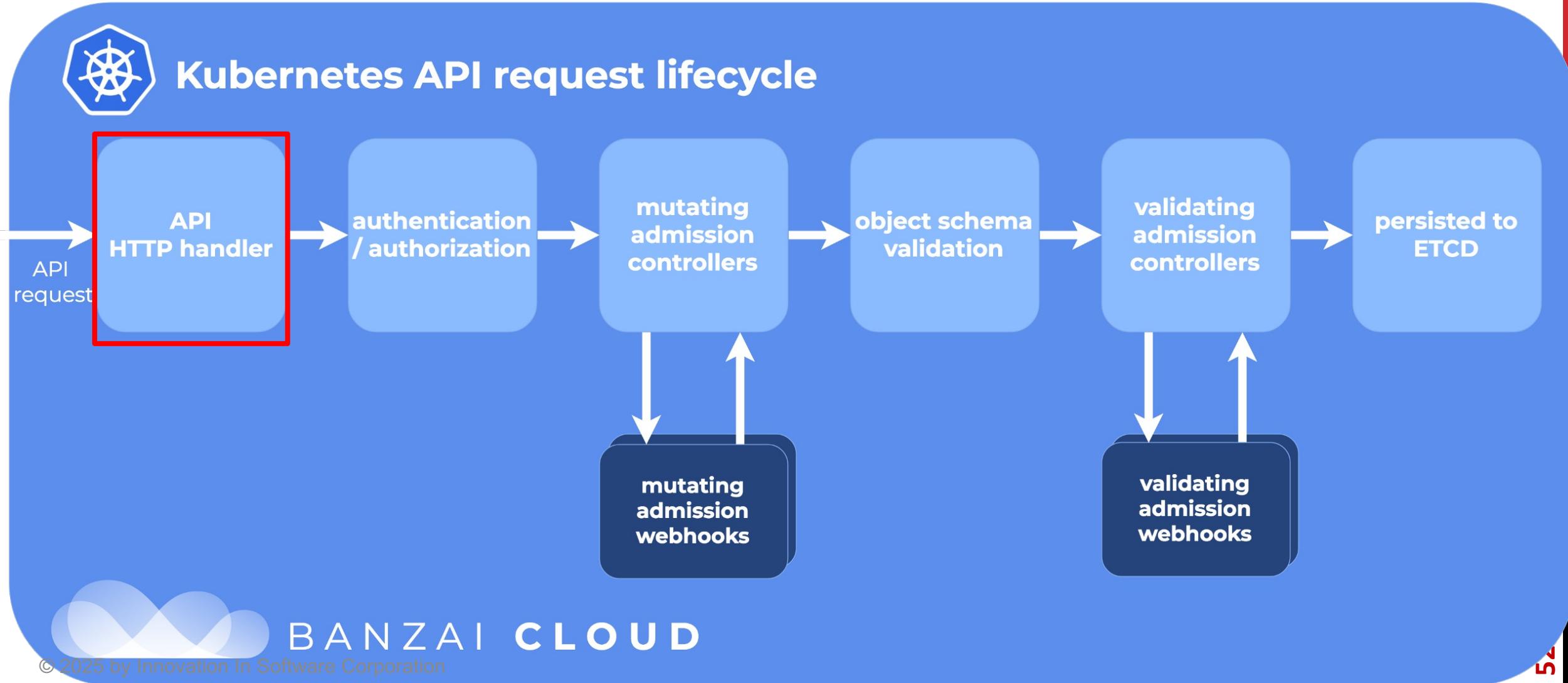
K8s components  
written in Go  
([golang.org](https://golang.org))

- **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

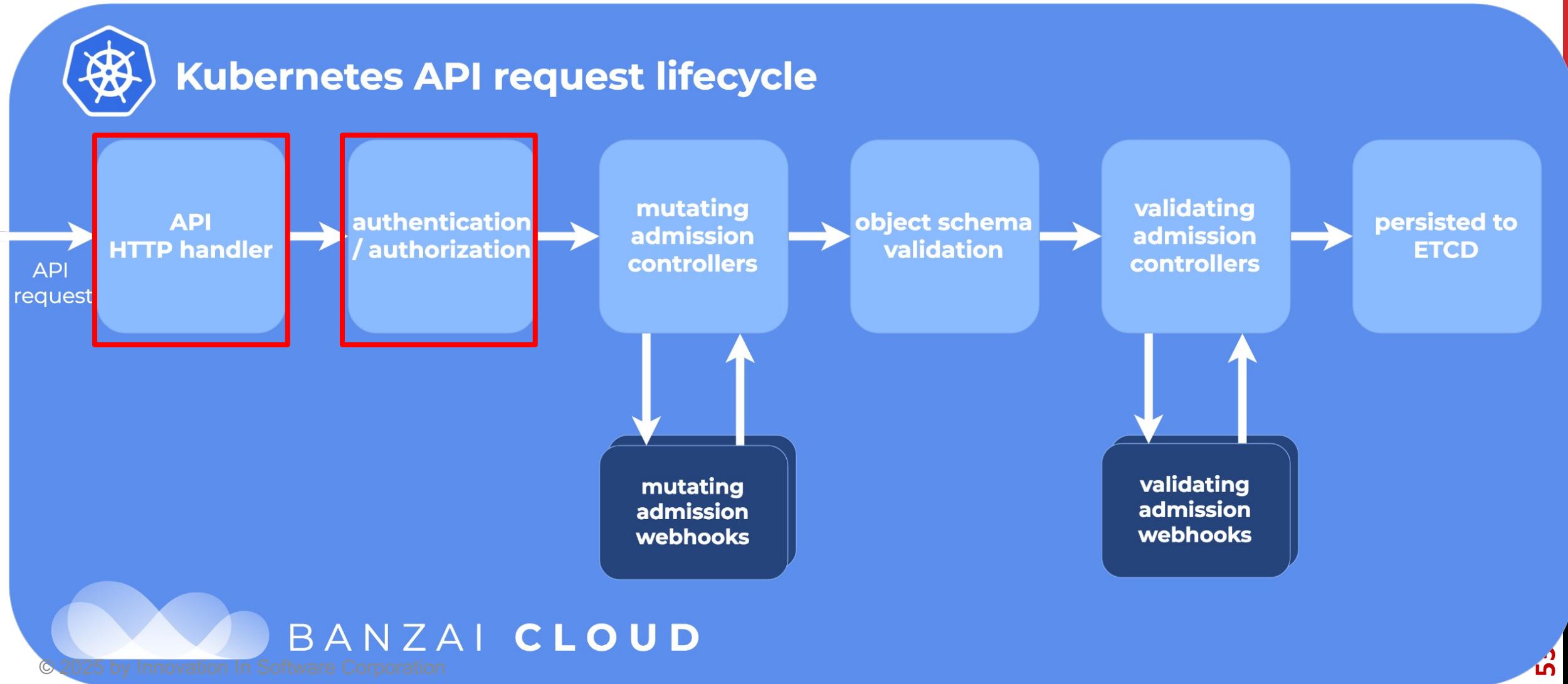
# Kubernetes API Deep Dive



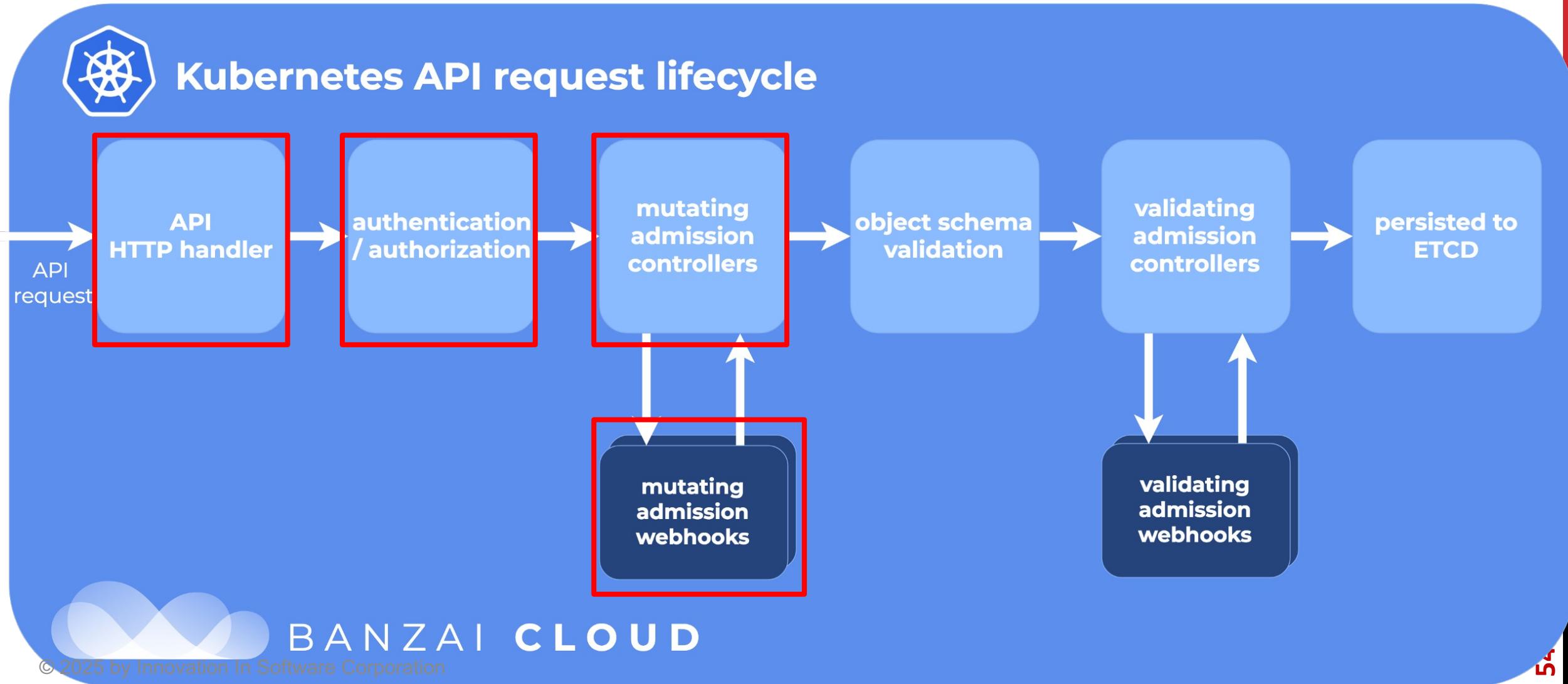
# Kubernetes lifecycle



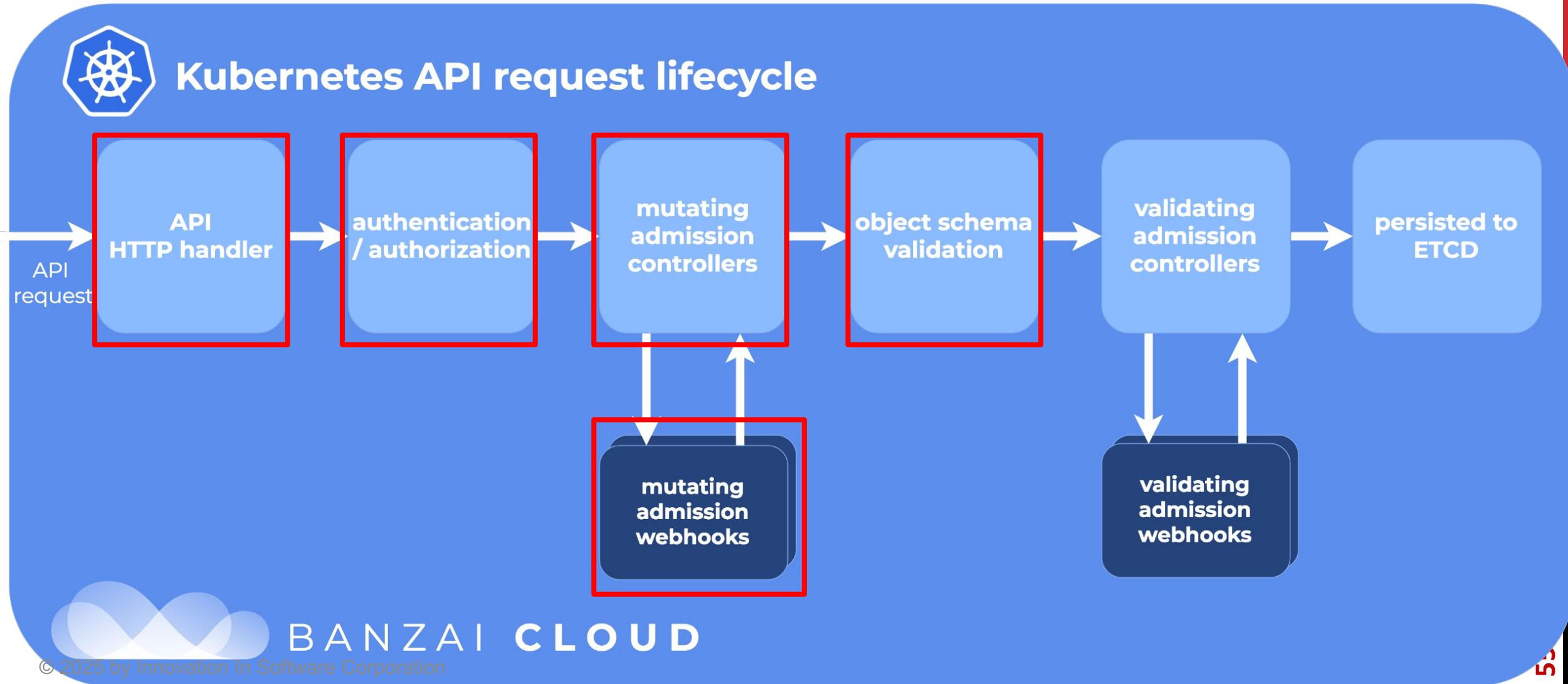
# Kubernetes lifecycle



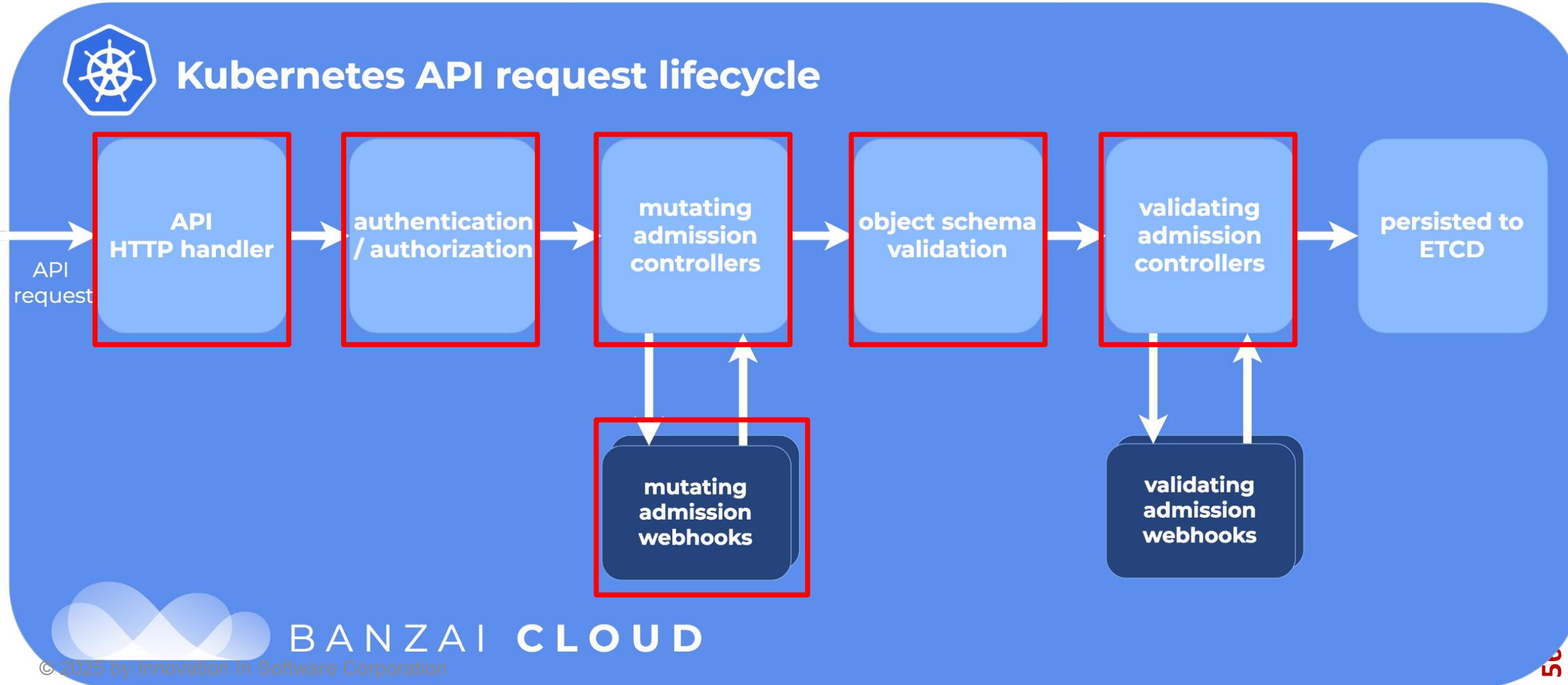
# Kubernetes lifecycle



# Kubernetes lifecycle

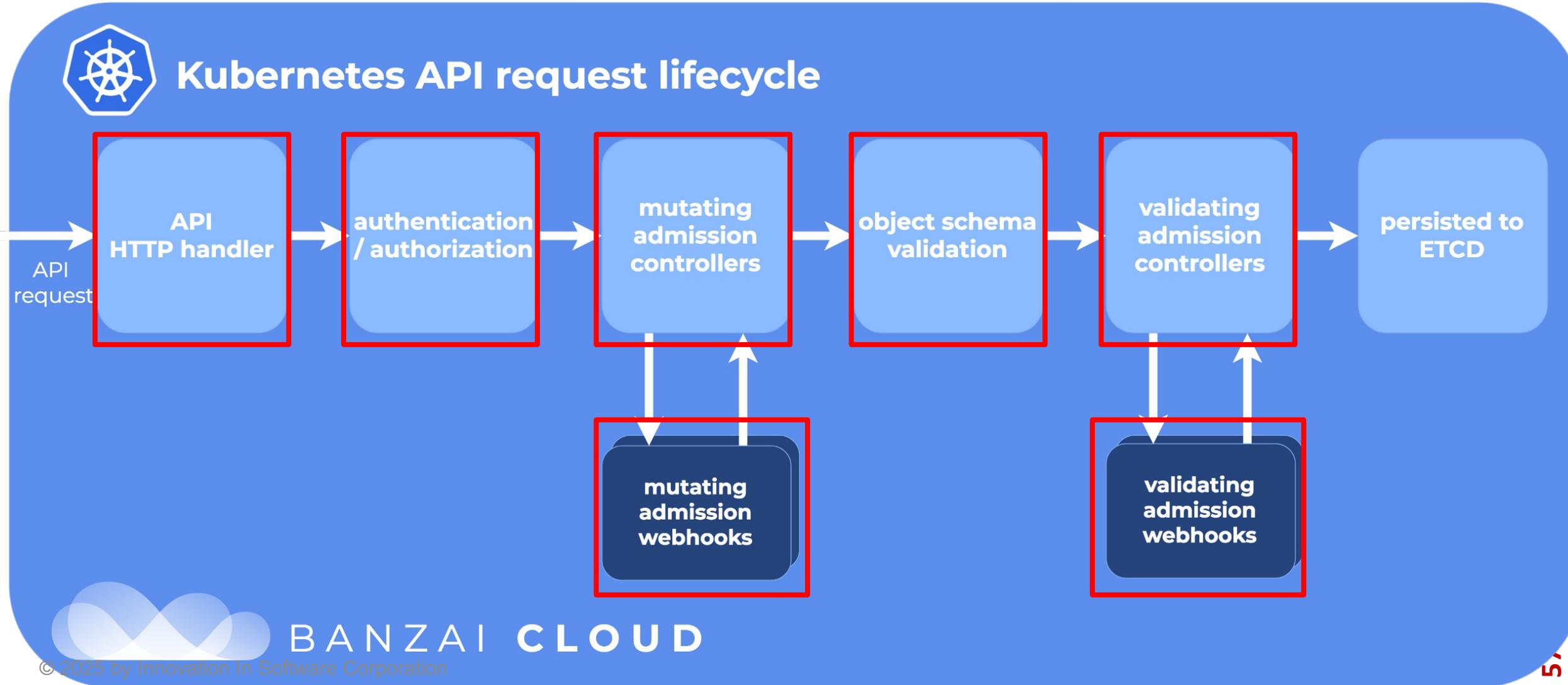


# Kubernetes lifecycle

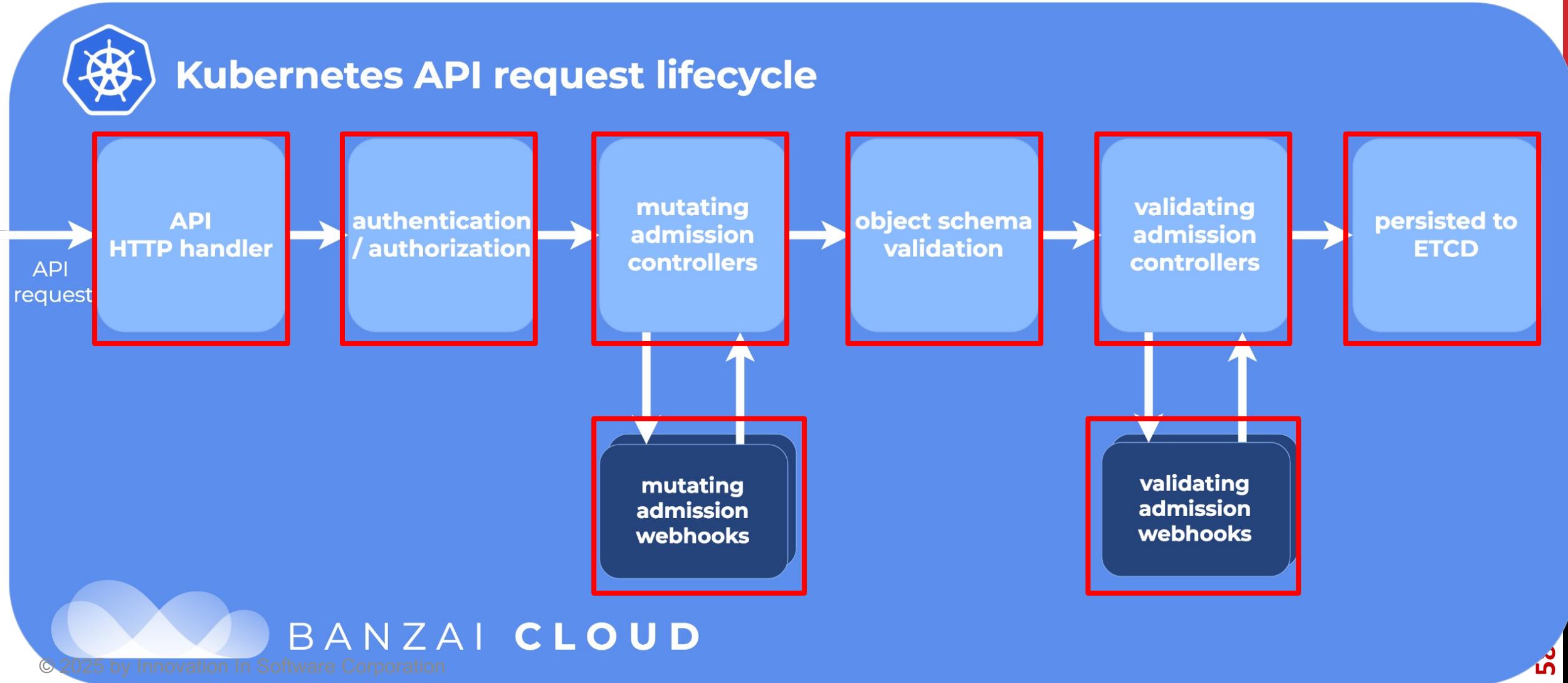


BANZAI CLOUD

# Kubernetes lifecycle



# Kubernetes lifecycle



# 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

Alpha/Experimental



v1alpha1

Early Release

Disabled by Default

For Testing Only

Breaking Changes



v1beta1

Thoroughly Tested

Considered Stable  
(test)

More Stable

Feedback Encouraged

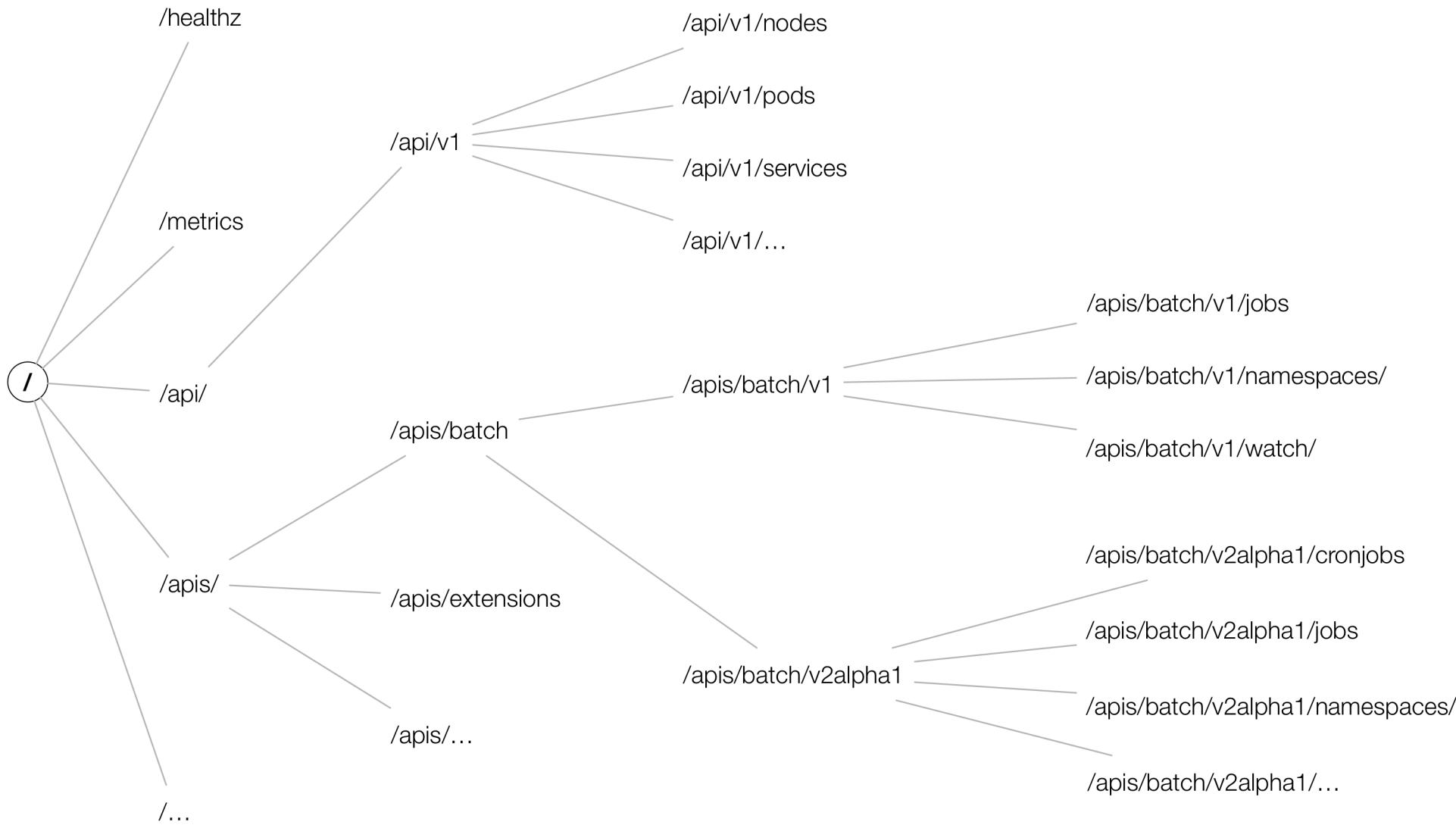


v1

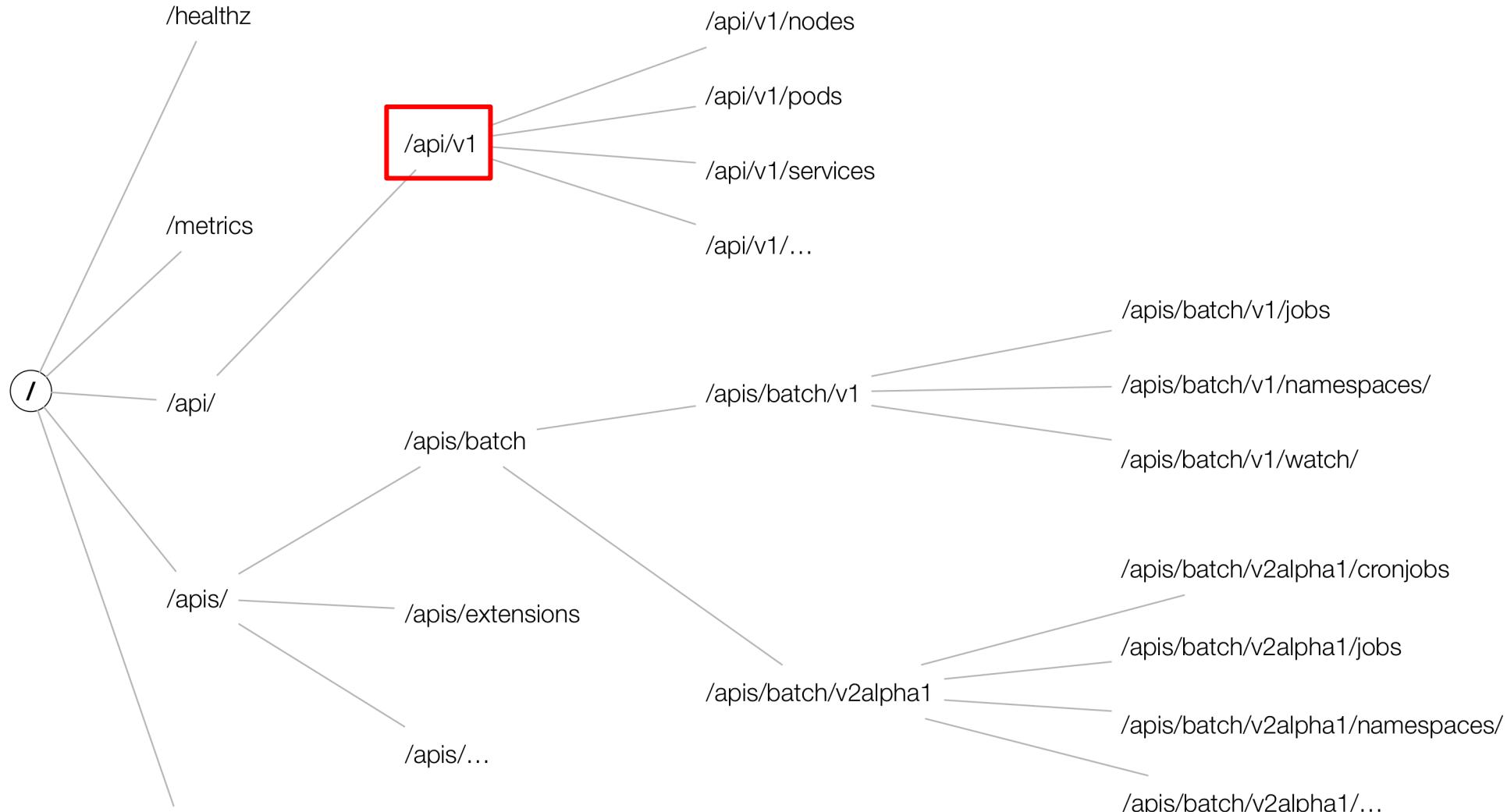
Backwards Compatible

Production Ready

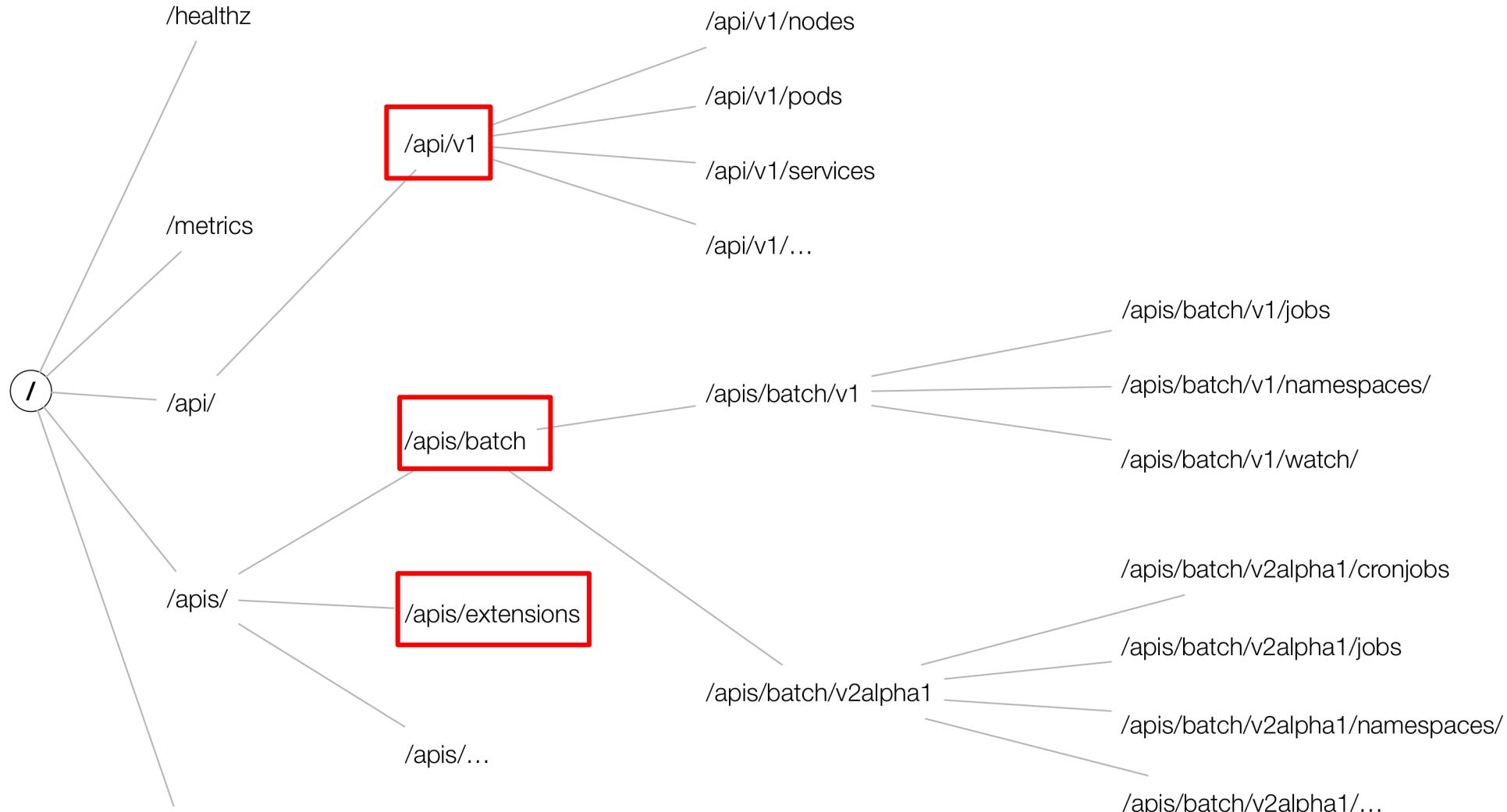
# API



# API Groups



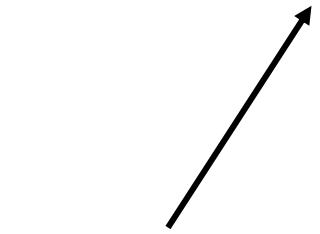
# API Groups



# API Group

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

/apis/**batch**/v1/jobs

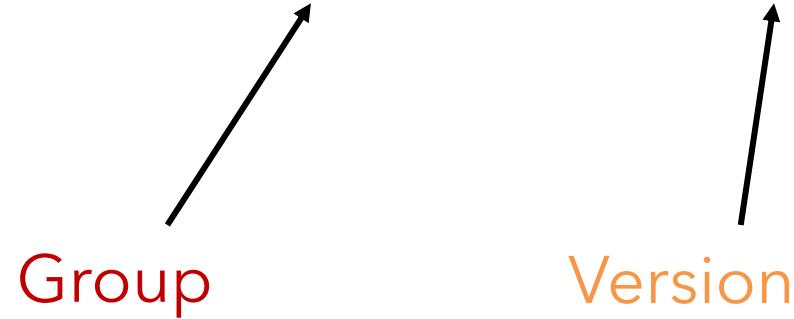


Group

# API Version

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

/apis/**batch**/**v1**/jobs



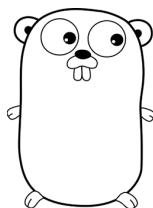
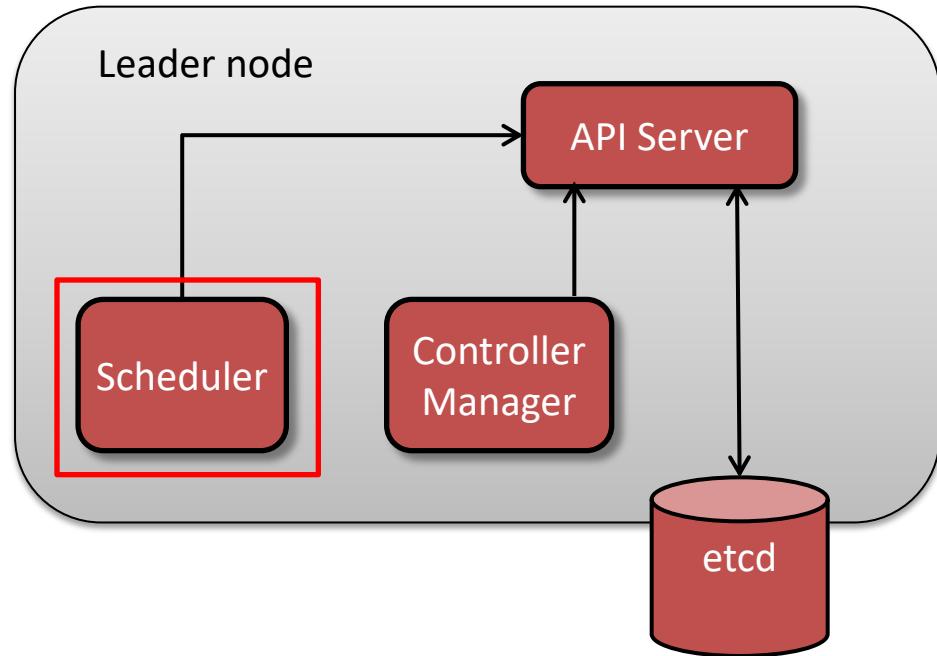
# API Resource

- System entity being manipulated as JSON over HTTP

/apis/**batch**/**v1**/**jobs**

The URL is divided into three main components: 'Group' (batch), 'Version' (v1), and 'Resource' (jobs). Each component is labeled with its name below it and has an arrow pointing to its corresponding part in the URL.

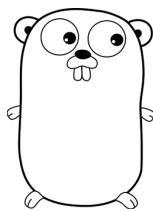
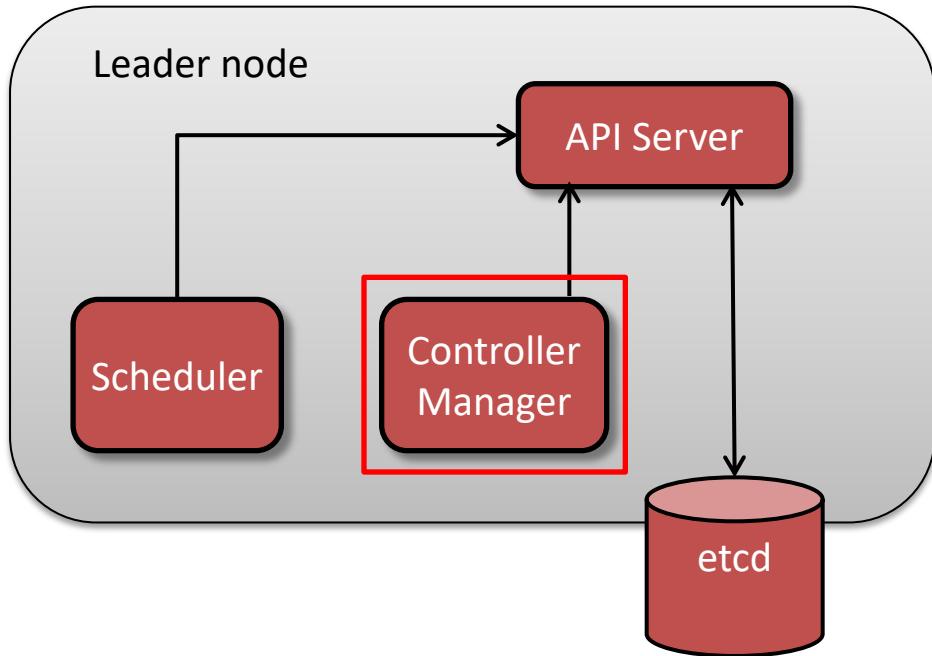
# Kubernetes Leader Node Components



K8s components  
written in Go  
([golang.org](https://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

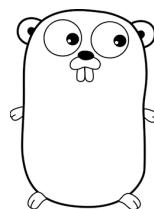
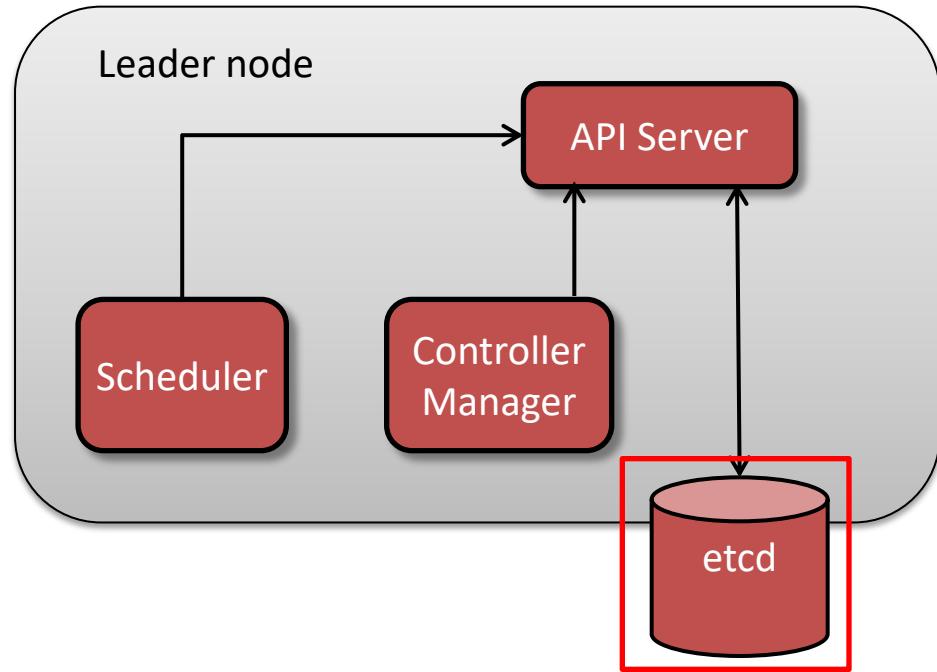
# Kubernetes Leader Node Components



K8s components  
written in Go  
([golang.org](https://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

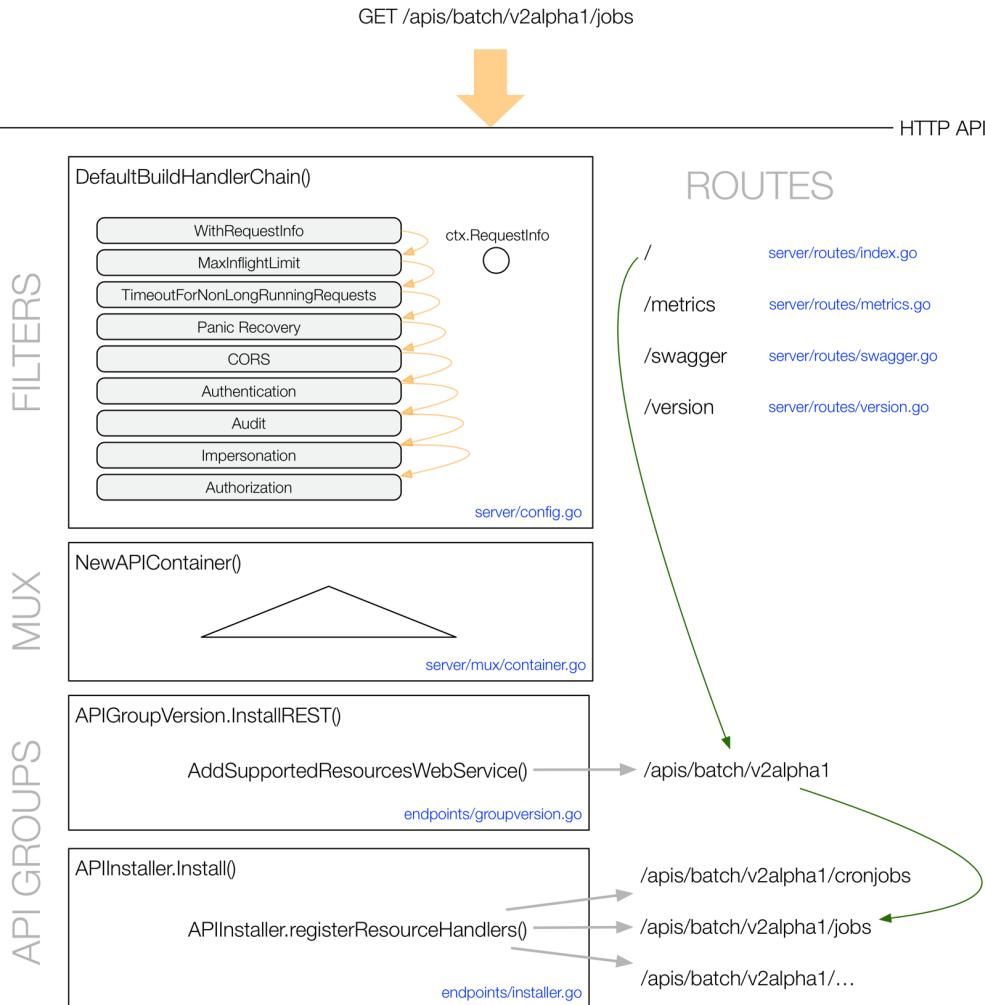
# Kubernetes Leader Node Components



K8s components  
written in Go  
([golang.org](https://golang.org))

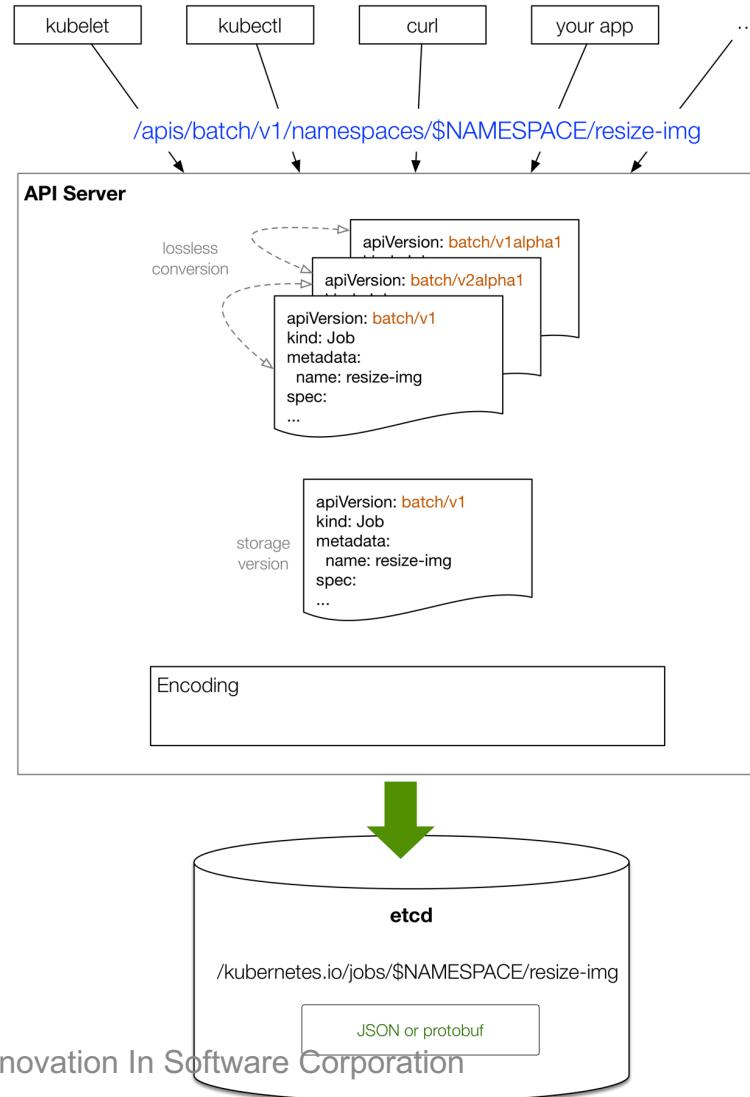
- **API Server (`kube-apiserver`)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (`kube-scheduler`)**: selects nodes for newly created pods to run on
- **Controller manager (`kube-controller-manager`)**: runs background controller processes for the system to enforce 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

# Request Flow and Processing



1. HTTP request is processed
2. Multiplexer routes the HTTP request to handler depending on path
3. Routes connect handlers with HTTP paths
4. The handler, registered per API Group takes the HTTP request context (user, rights etc.) delivers the requested object from storage (etcd)

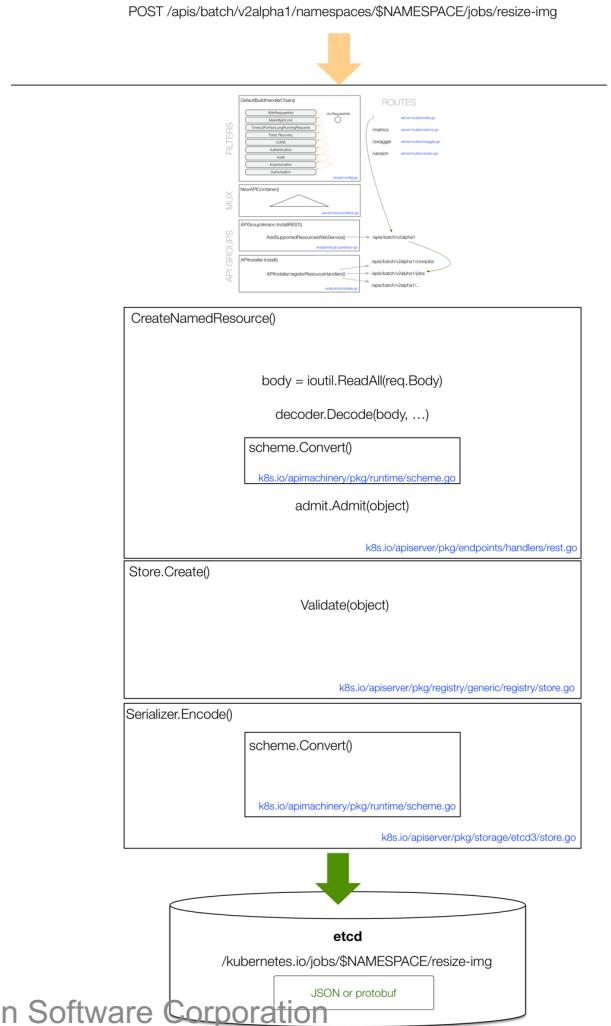
# Etcd flow



- **Etcd flow:**

- client provides desired object state in YAML or JSON
- kubectl converts YAML to JSON and sends it to API
- API server turns input object state into canonical storage version
- storage process in etcd, at a certain key, into a value with the encoding to JSON or protobuf.

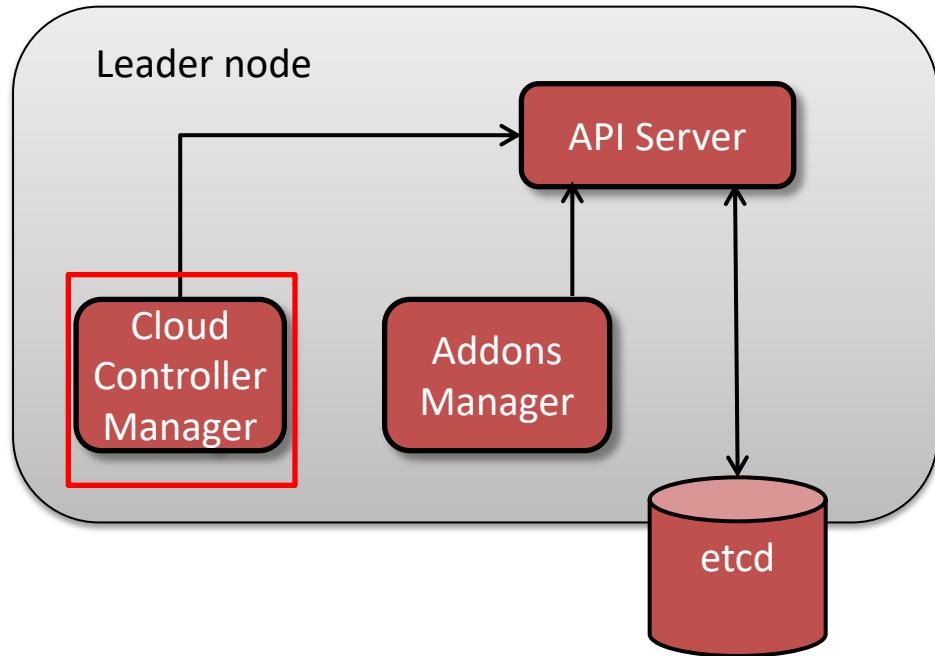
# Serialization of State Flow



- **Serialization flow:**

- API Server keeps all known k8s object kinds in a Go registry (Scheme).
- Version of kinds defined along with how they can be:
  - converted
  - new objects created
  - objects encoded/decoded to JSON/protobuf

# Leader Node Additional Components

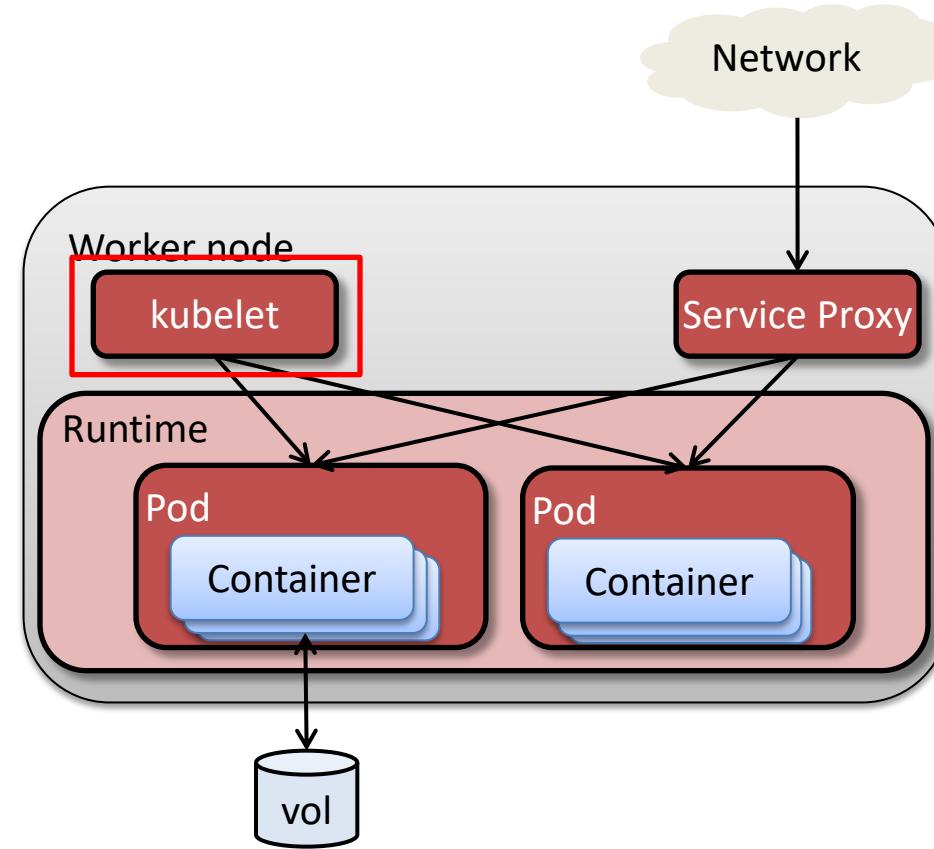


K8s components  
written in Go  
([golang.org](https://golang.org))

- **cloud-controller-manager**: runs controllers interacting with underlying IaaS providers – Allows cloud vendor-specific code to be separate from main K8s system components
  - **addons-manager**: creates and maintains cluster addon resources in ‘kube-system’ namespace, e.g.
- **Kubernetes Dashboard**: general web UI for application and cluster management
- **kube-dns**: serves DNS records for K8s services and resources
- Container resource monitoring and cluster-level logging

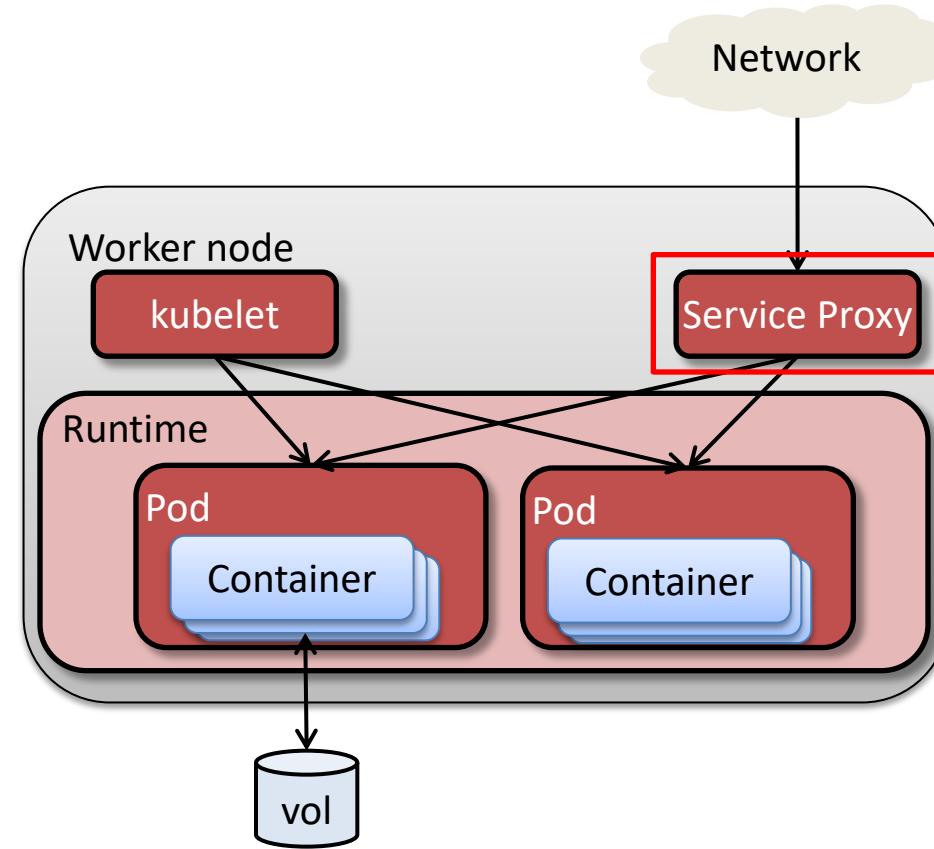
# Kubernetes Member Node Components

- **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



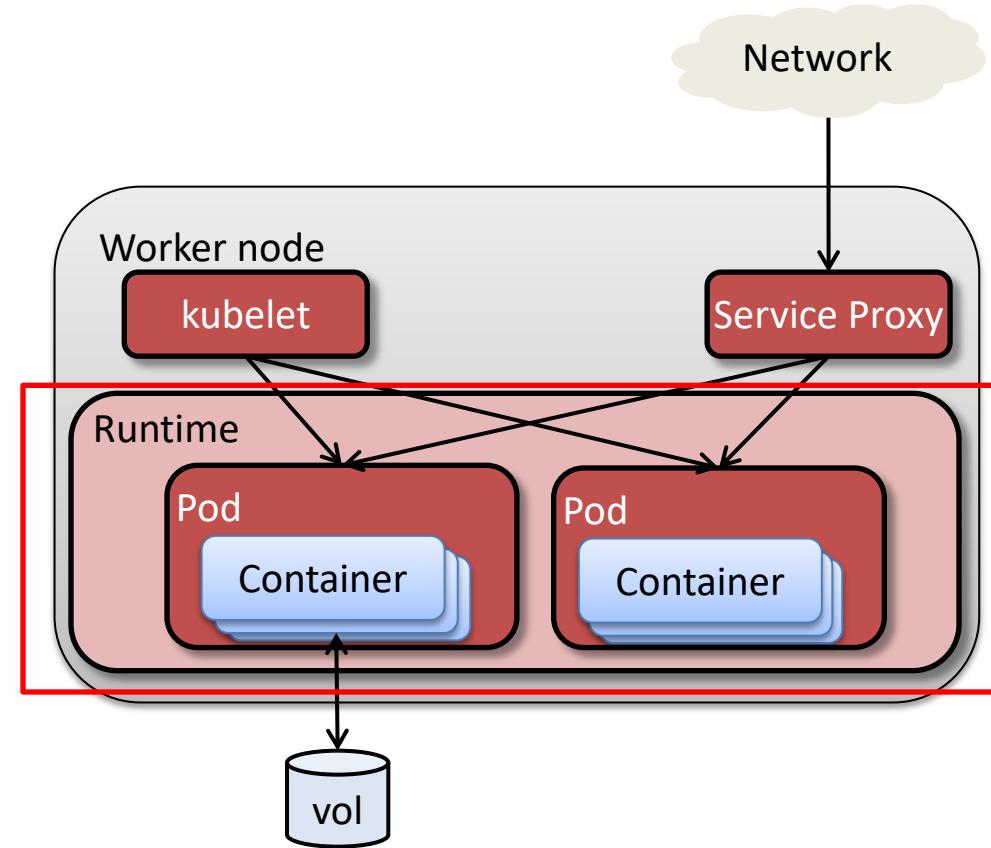
# Kubernetes Member Node Components

- **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 Member 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
- **Container runtime**





# Questions

# Lab: Create cluster



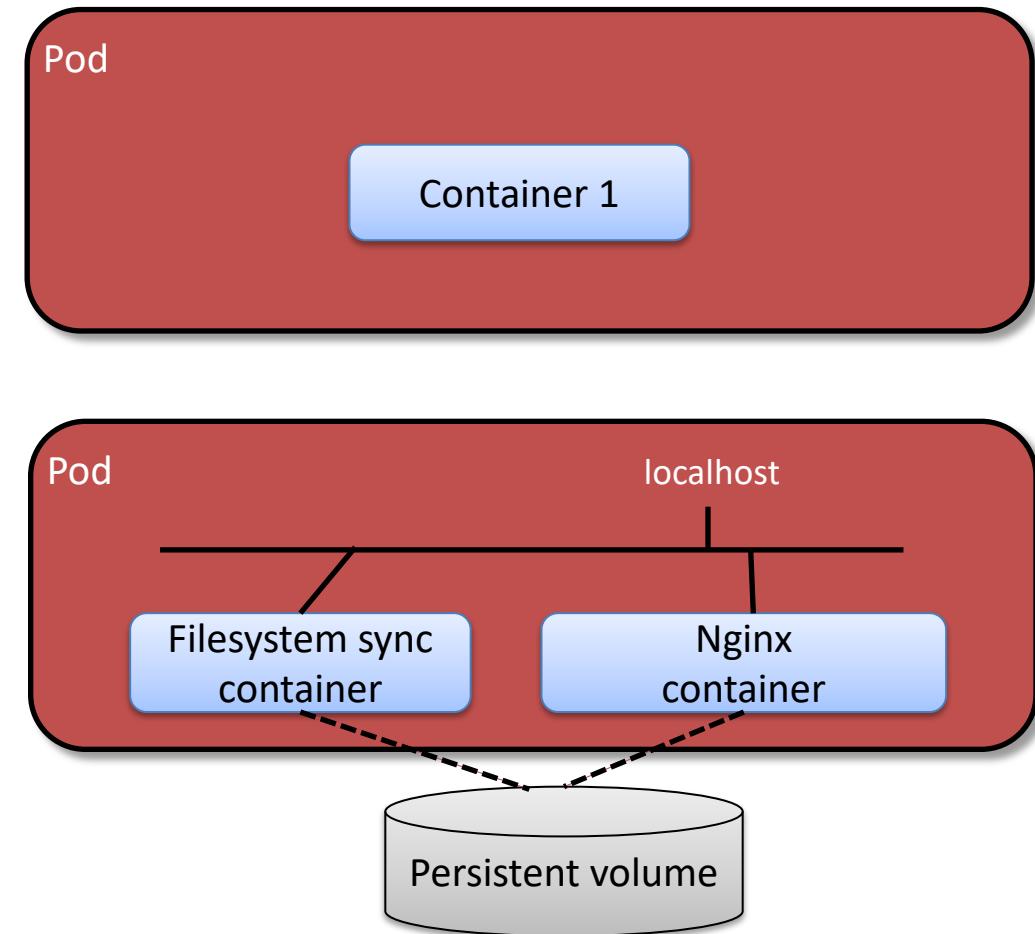
# Kubernetes Pods



# What is a Kubernetes Pod?

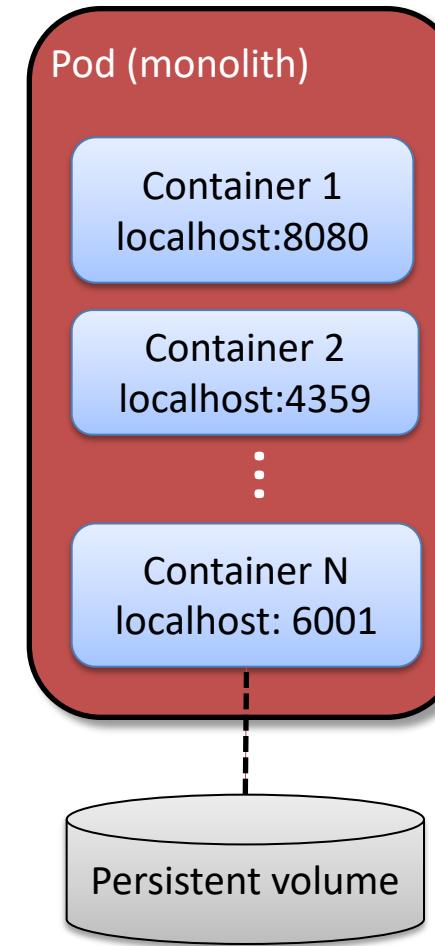
*Kubernetes design intention: Pod == application instance*

- 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



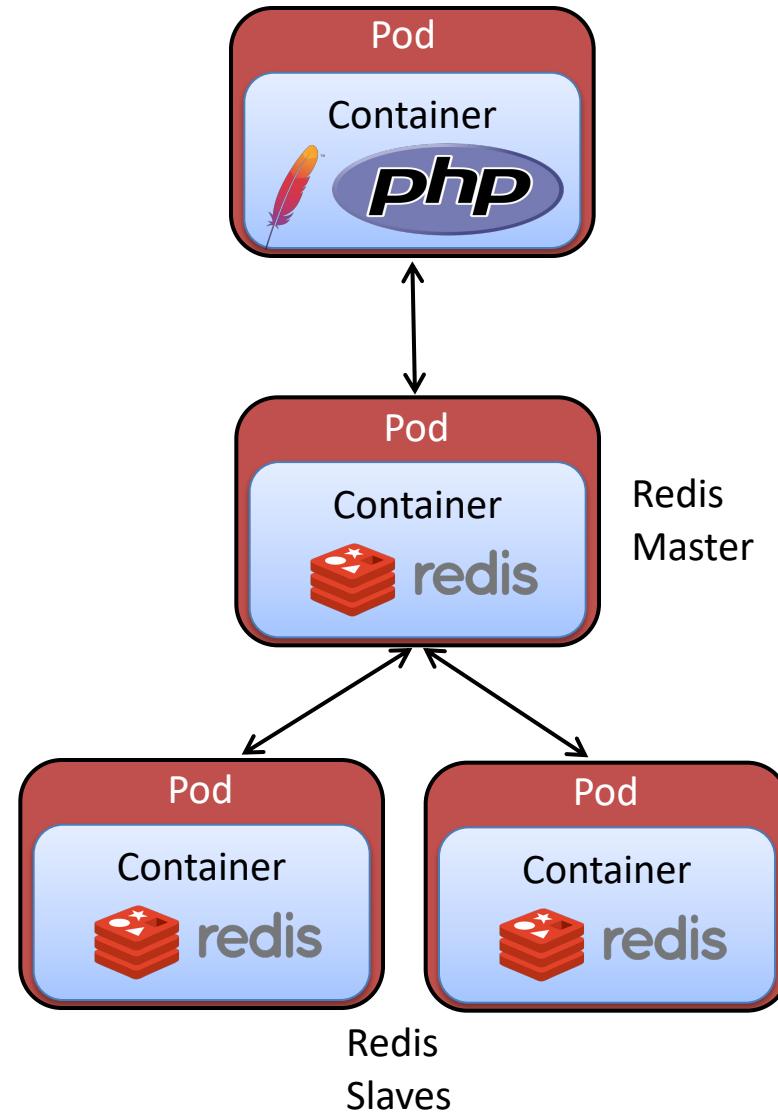
# Pods Enable Deployment Flexibility

- 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



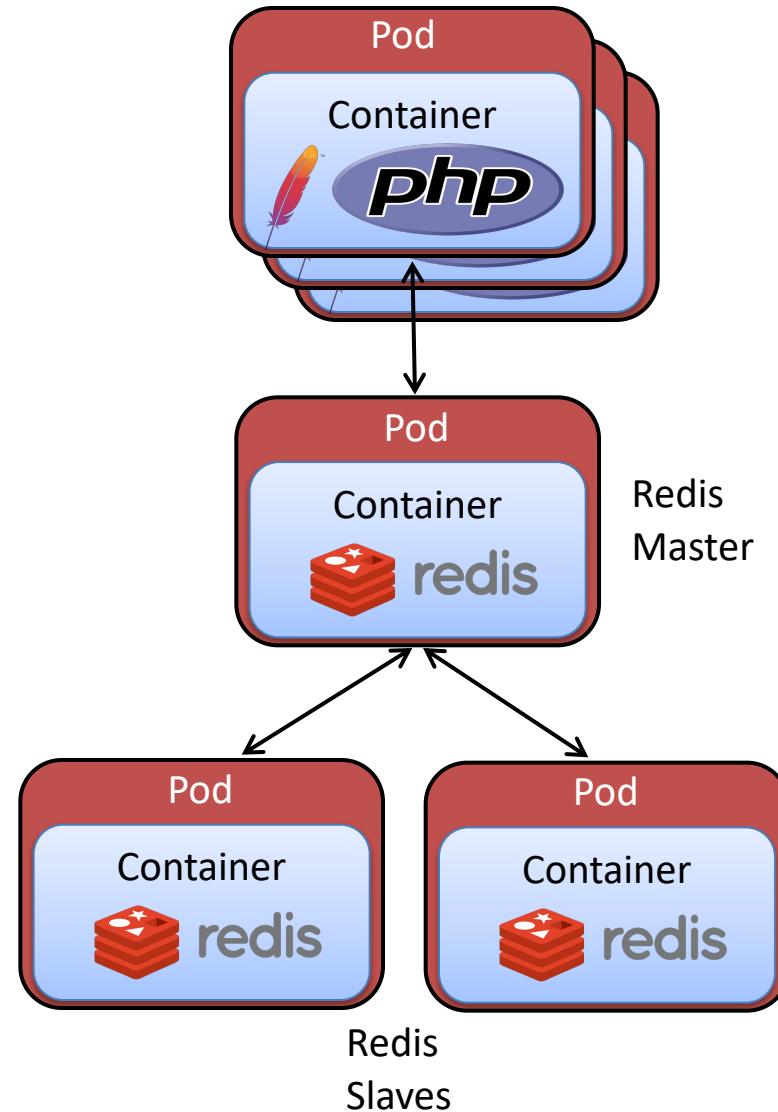
# Pods Enable Deployment Flexibility

- 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



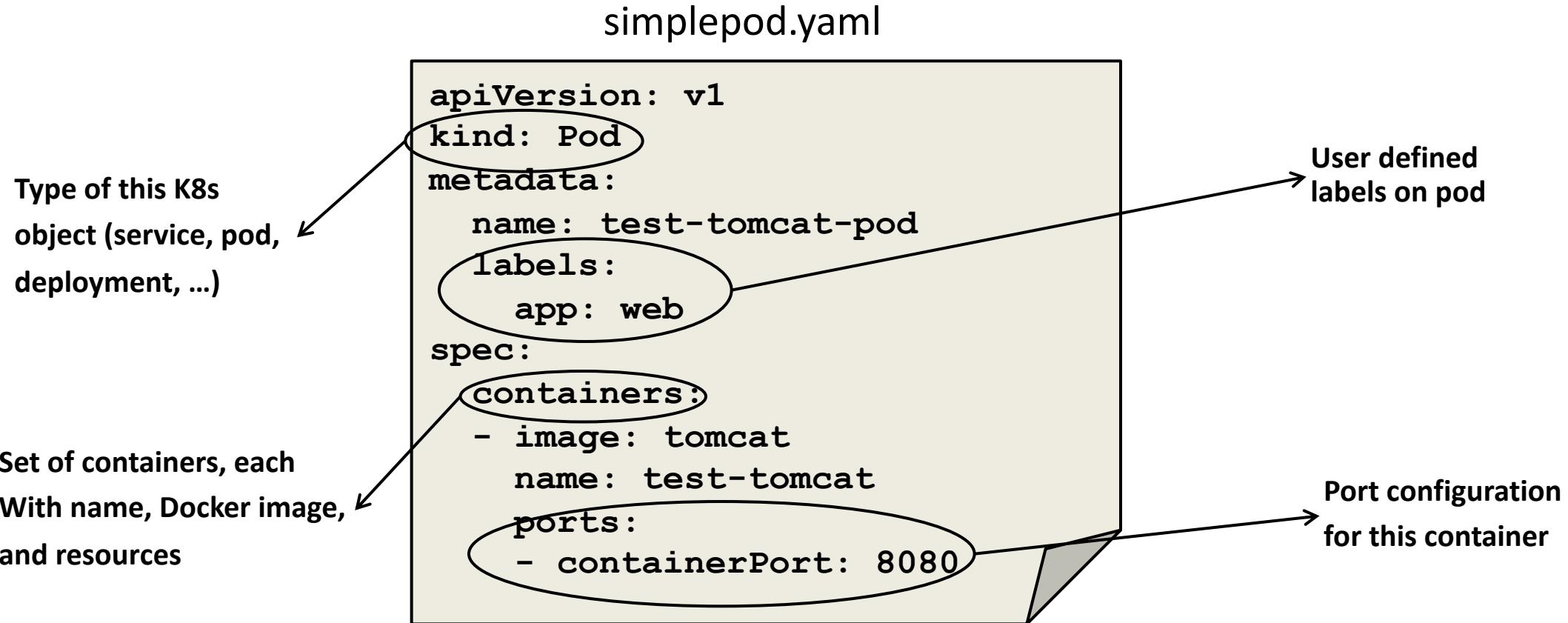
# Defining a Pod via a Manifest File

*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



- 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
```

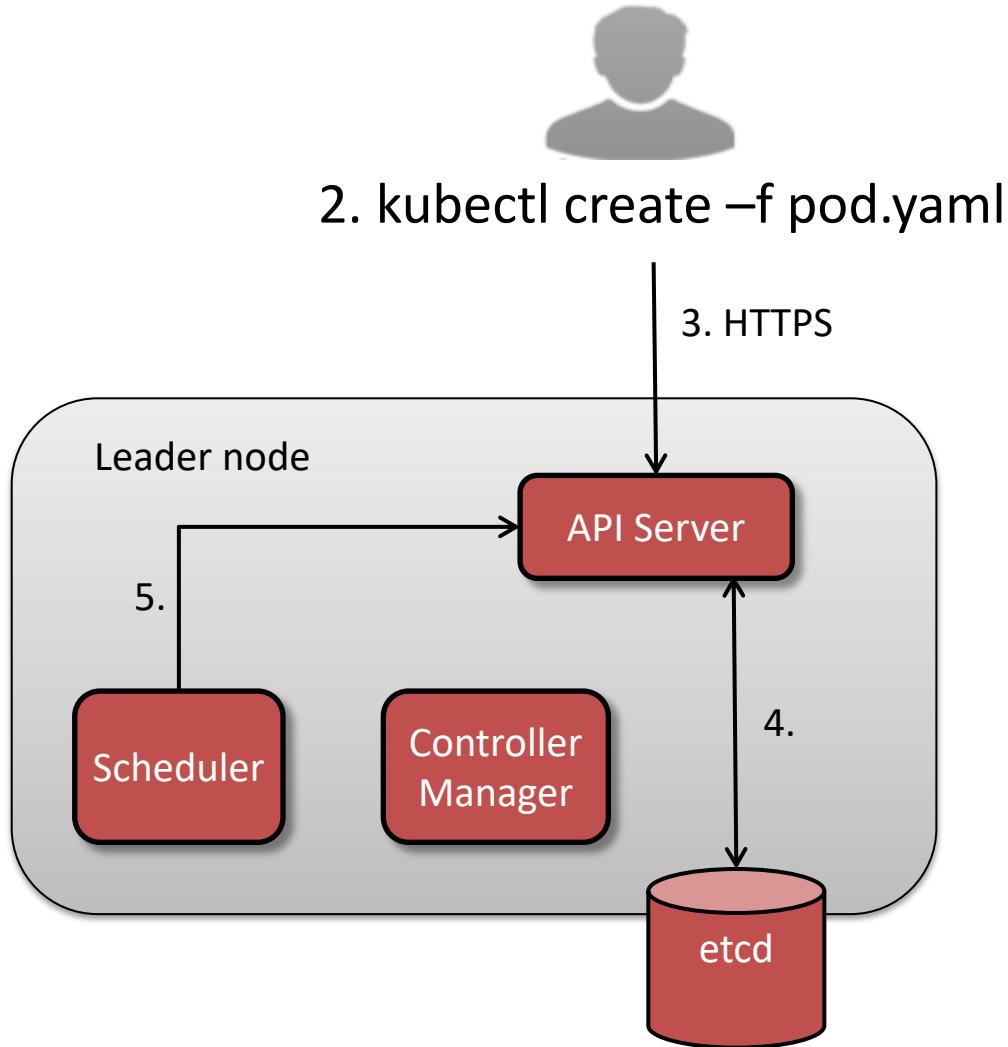
→ multipod.yaml

- 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

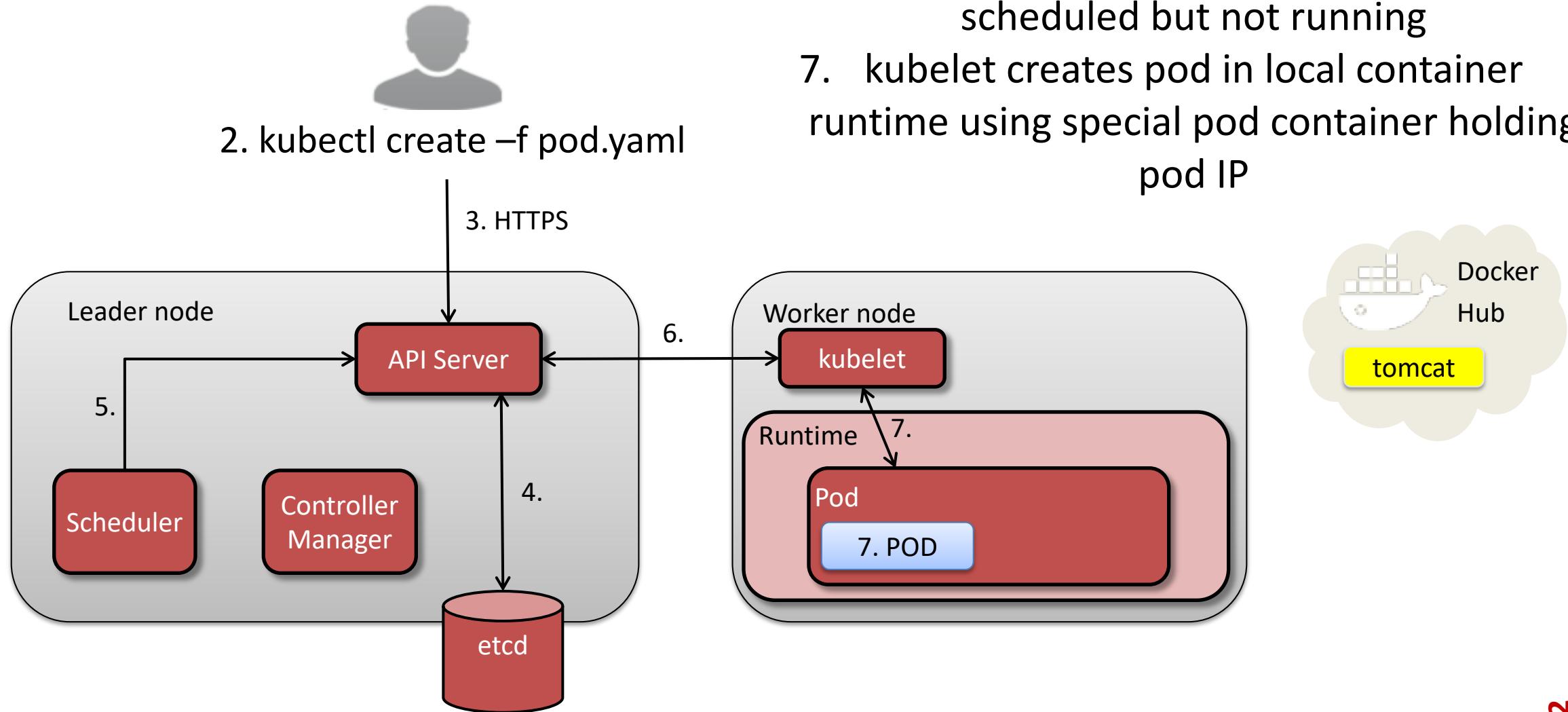


# Pod Creation Process

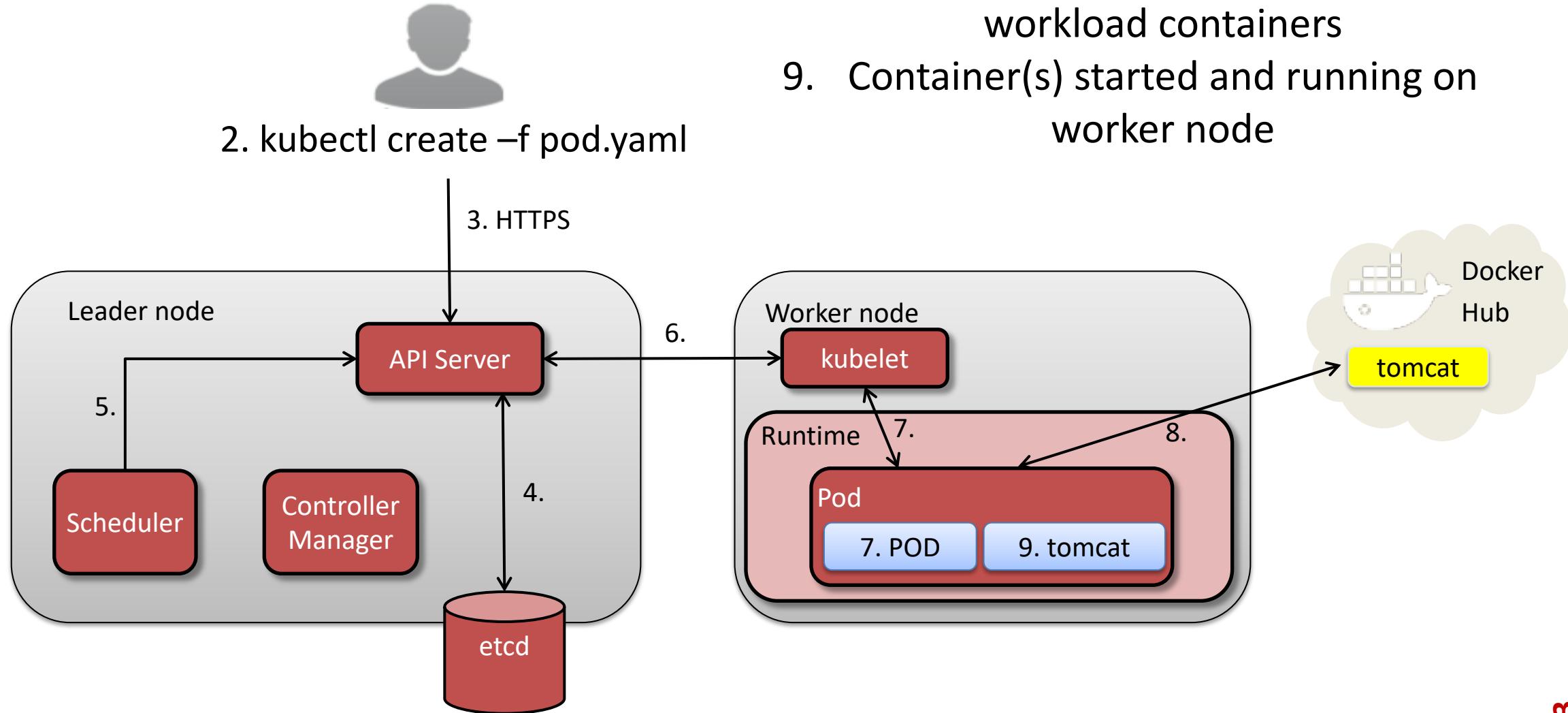


1. User writes a pod manifest file
2. User requests creation of pod from manifest via CLI
3. CLI tool marshals parameters into K8s RESTful API request (HTTP POST)
4. kube-apiserver creates new pod object record in etcd, with no node assignment
5. kube-scheduler notes new pod via API
  - a. Selects node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

# Pod Creation Process



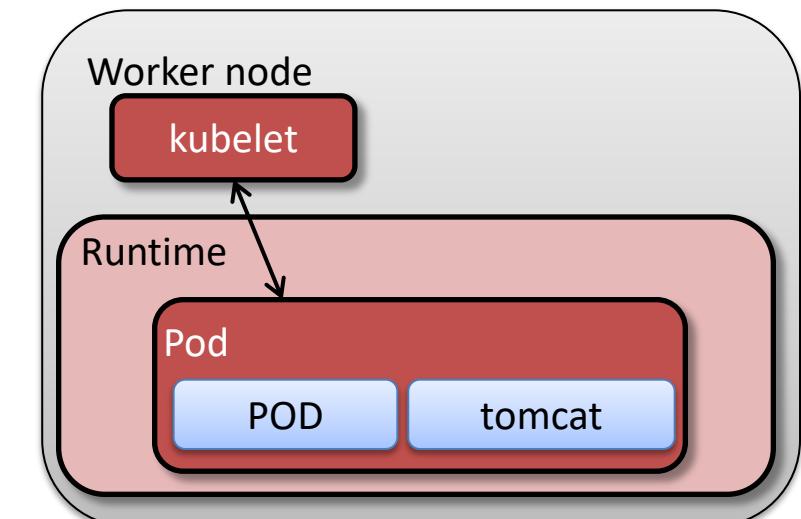
# Pod Creation Process



# 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:       8d933333333333333333333333333333
    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

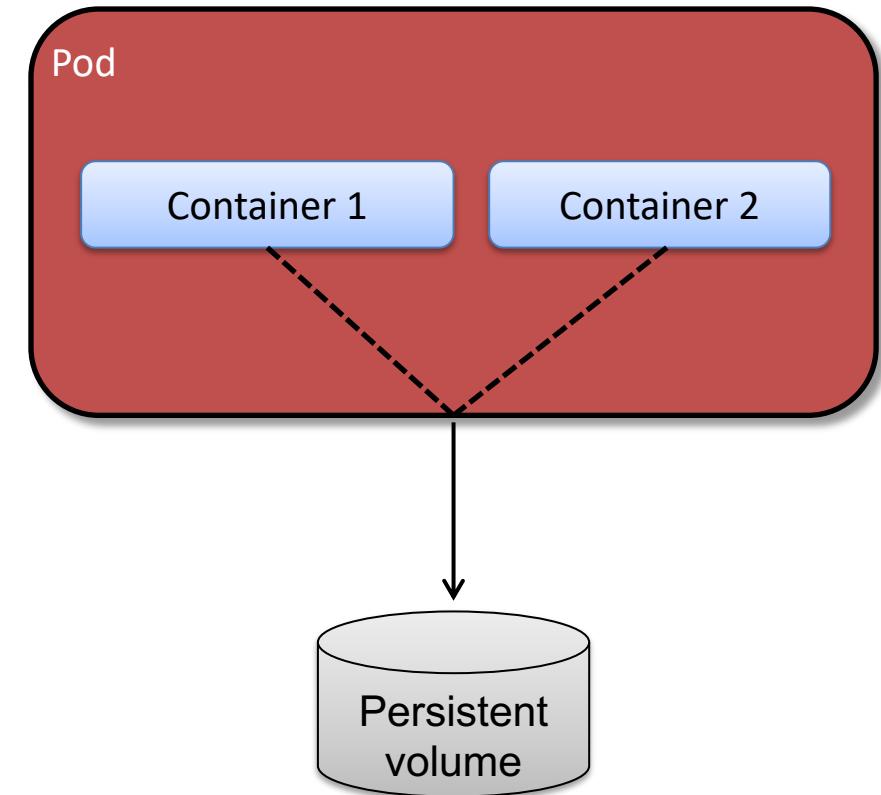
```
$ kubectl get pods --show-labels --namespace=kube-system
```

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



# How Kubernetes Runs Workloads



# All Workloads are Containerized



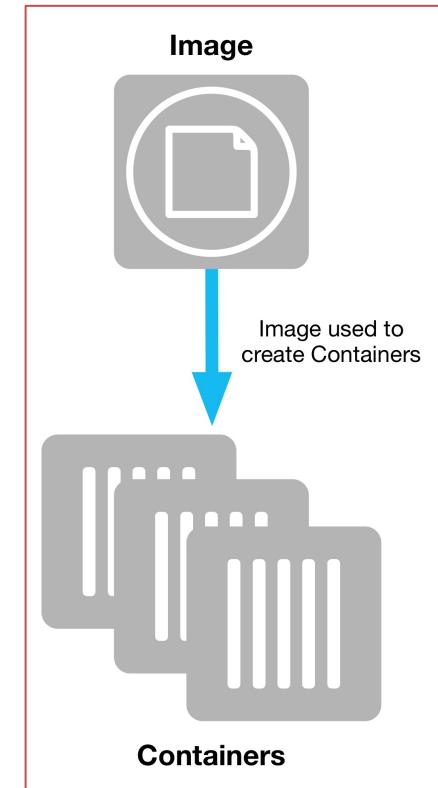
Docker allows you to package an application with its dependencies into a standardized unit for software development and deployment

## 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)

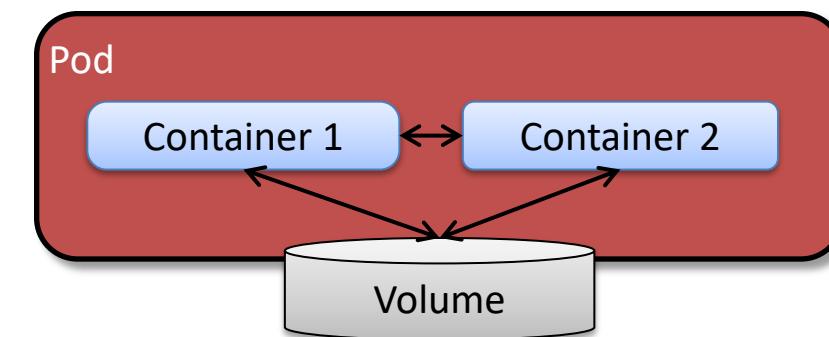
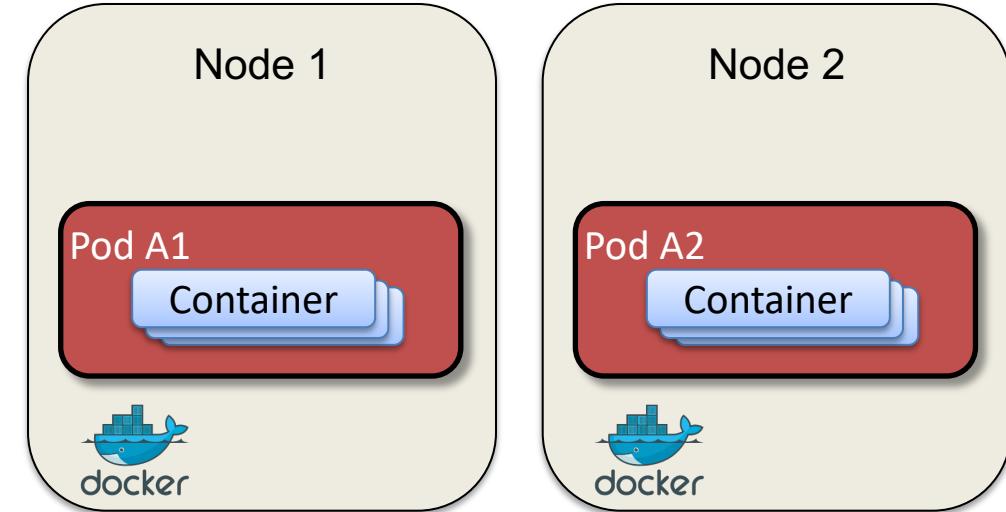
## Container

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



# Kubernetes Pods

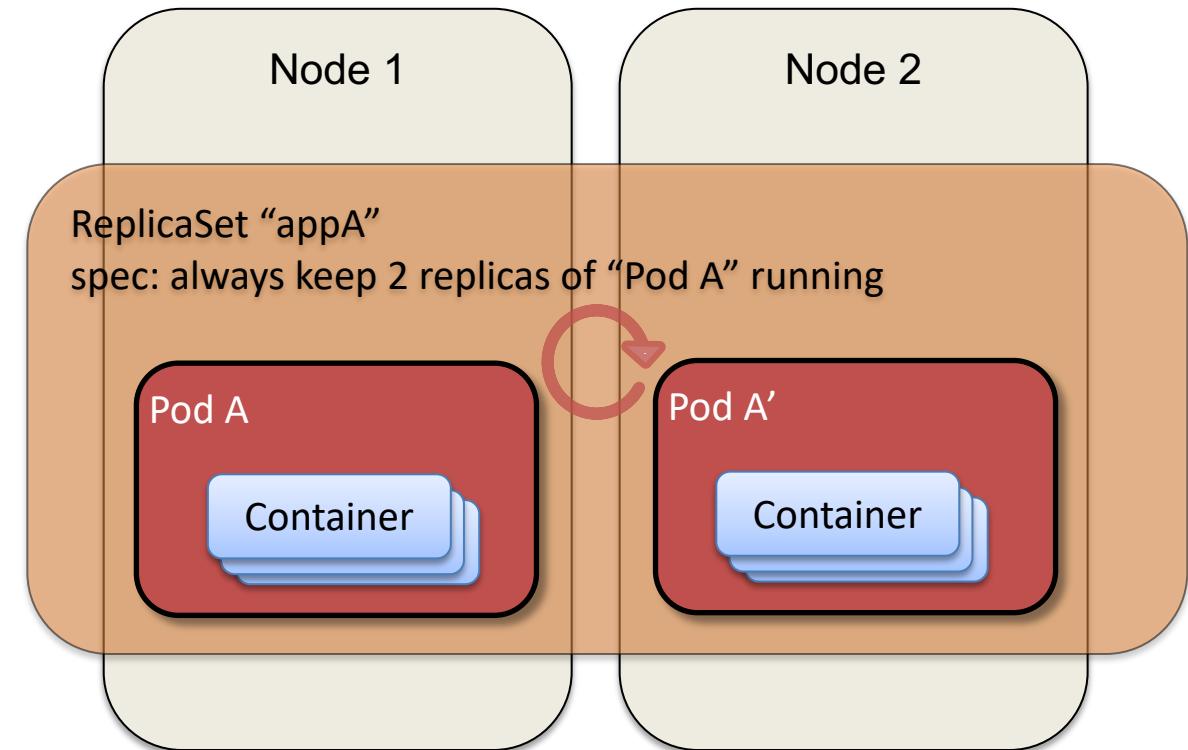
- Smallest K8s workload unit is the **Pod**, a set of co-scheduled containers
- A Pod == an application instance
- Pods can include more than one container, for tightly-coupled application components
- Containers in the same Pod share networking and storage resources
- Kubernetes handles efficient placement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



# Controllers for Different Application Patterns

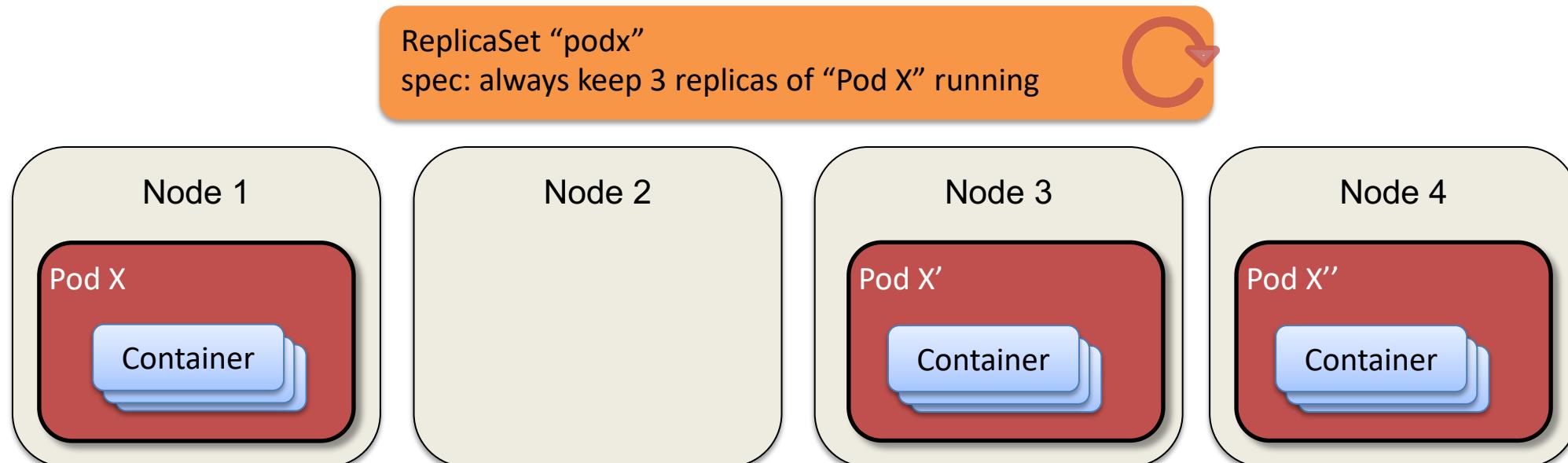
- K8s controller objects used to create and manage Pods according to different application patterns => control loops
- **ReplicaSets** manage sets of replicas of stateless workloads to ensure availability
- **StatefulSets** manage stateful workloads on stable storage to ensure consistency
- **DaemonSets**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

Controller example: ReplicaSet



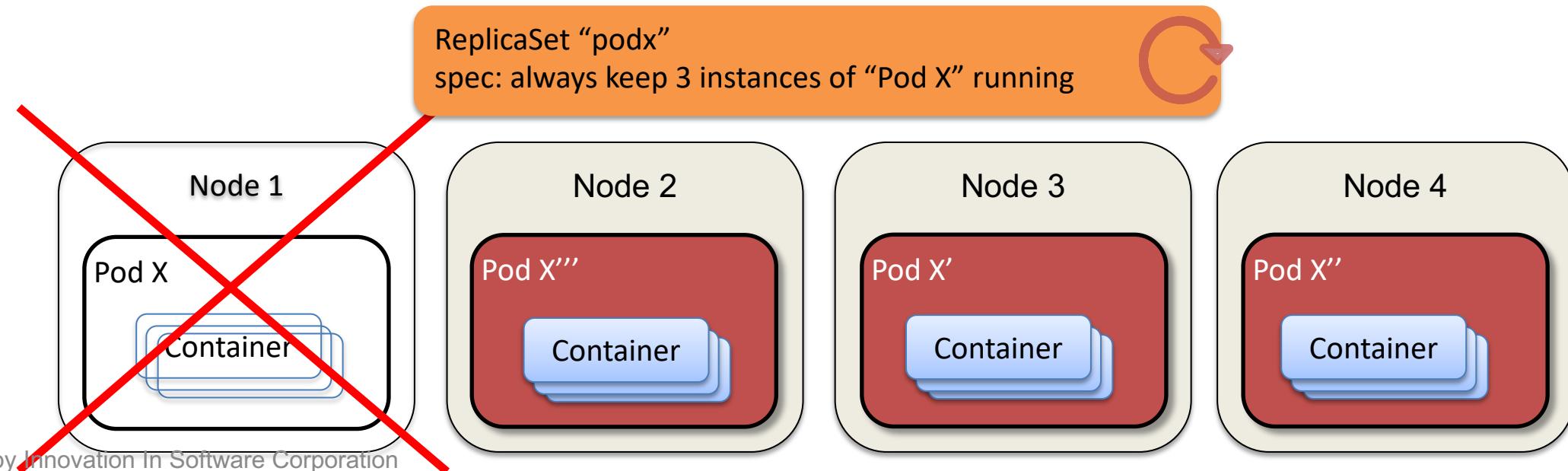
# Controller Example: ReplicaSets

- 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

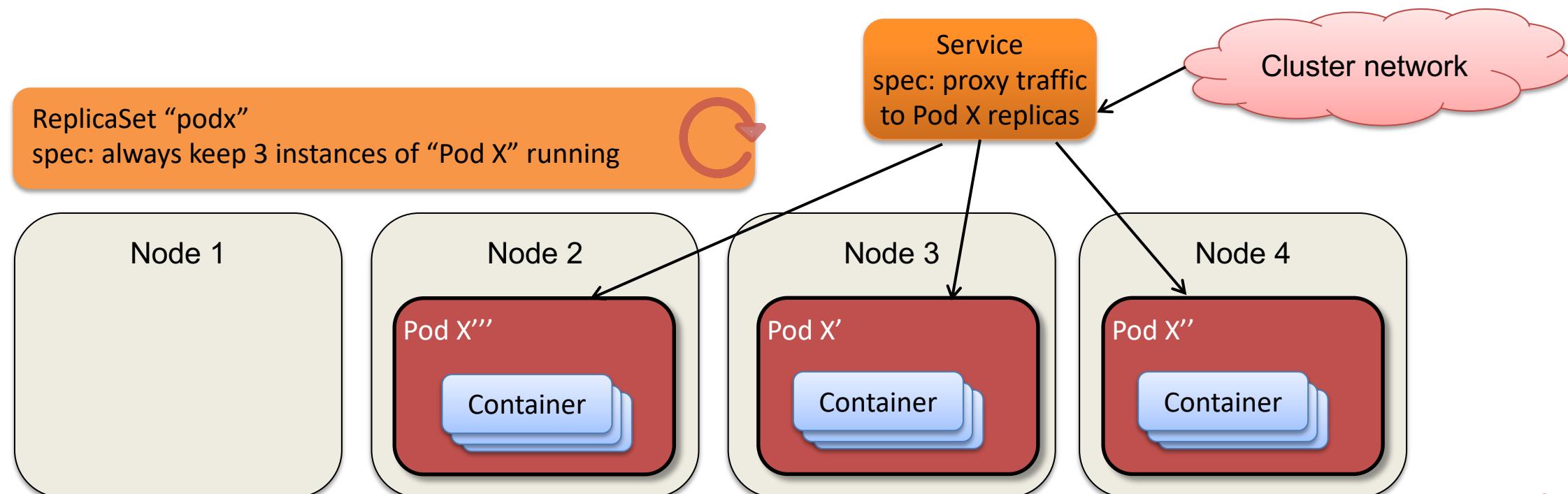
- When a Node fails, its Pods are lost
- K8s system manages the state of the ReplicaSet back to the declared configuration
- Changing the configuration will result in management to new state, e.g. scale out



# 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



# Lab: Kubernetes Pods

