# Full Stack Development

## Containers, Microservices and UI

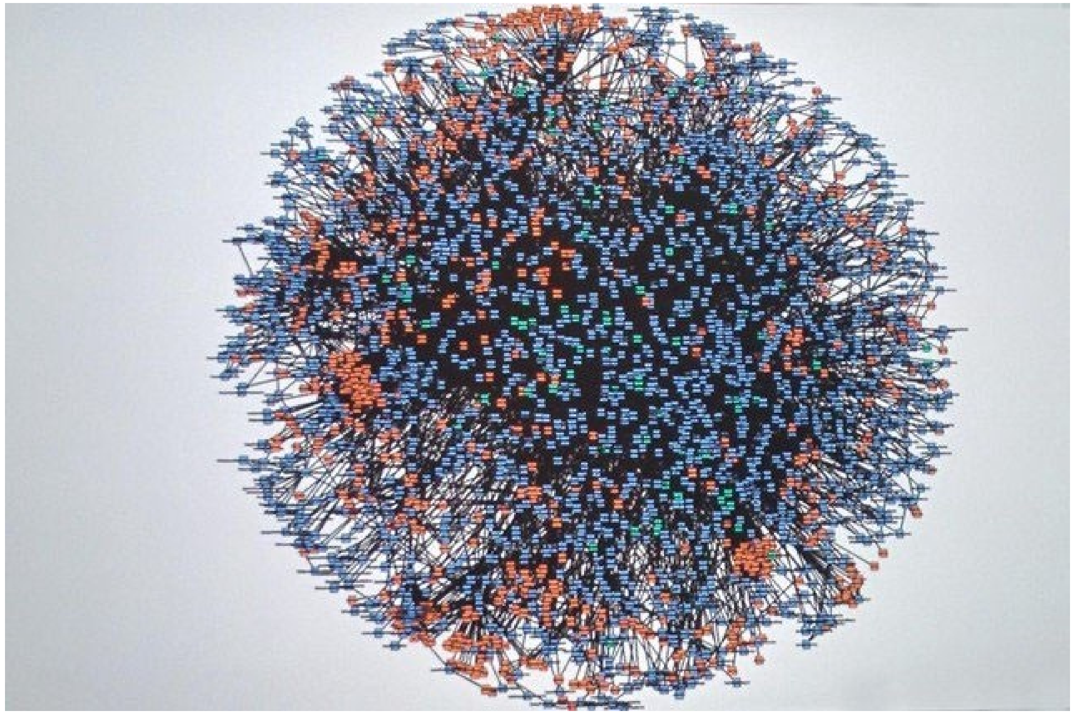
### 11. Kubernetes

# The Operations Challenge

- In a previous module, we looked at containerization from a development perspective
- In production, we have to be concerned with
  - Coordinating the activity of possibly thousand of containers that need to work together
  - Creating and maintaining connections between containers
  - Ensure the whole system operates well enough to meet Service Level Agreements (SLAs)
- We need to deal with non-functional requirements
  - Loading, throughput, stress, response times
  - Disaster recovery
  - Security
- The lack of an effective way to do this was a major impediment to the deployment of microservice based applications

# Site Reliability Engineering

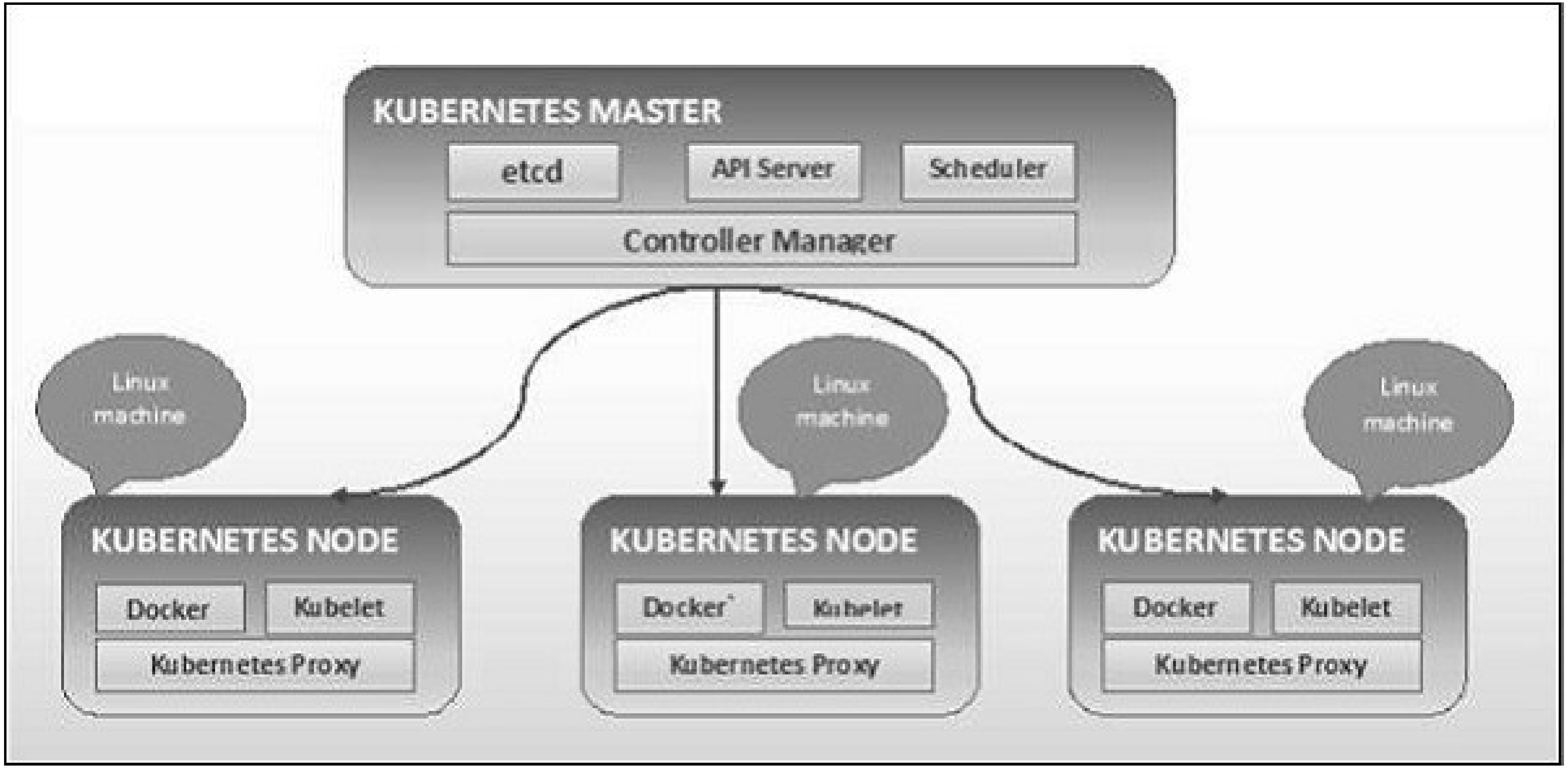
- Practices designed to ensure large systems are operational
- Continuously checking for potential problems
- Manages a set of mitigation responses to react to problems
- Recent examples
  - Rogers Canada 2022 network failure
  - Facebook October 2021 upgrade failure
  - Check out risks.org
- As applications scale, this becomes increasingly difficult



# Kubernetes

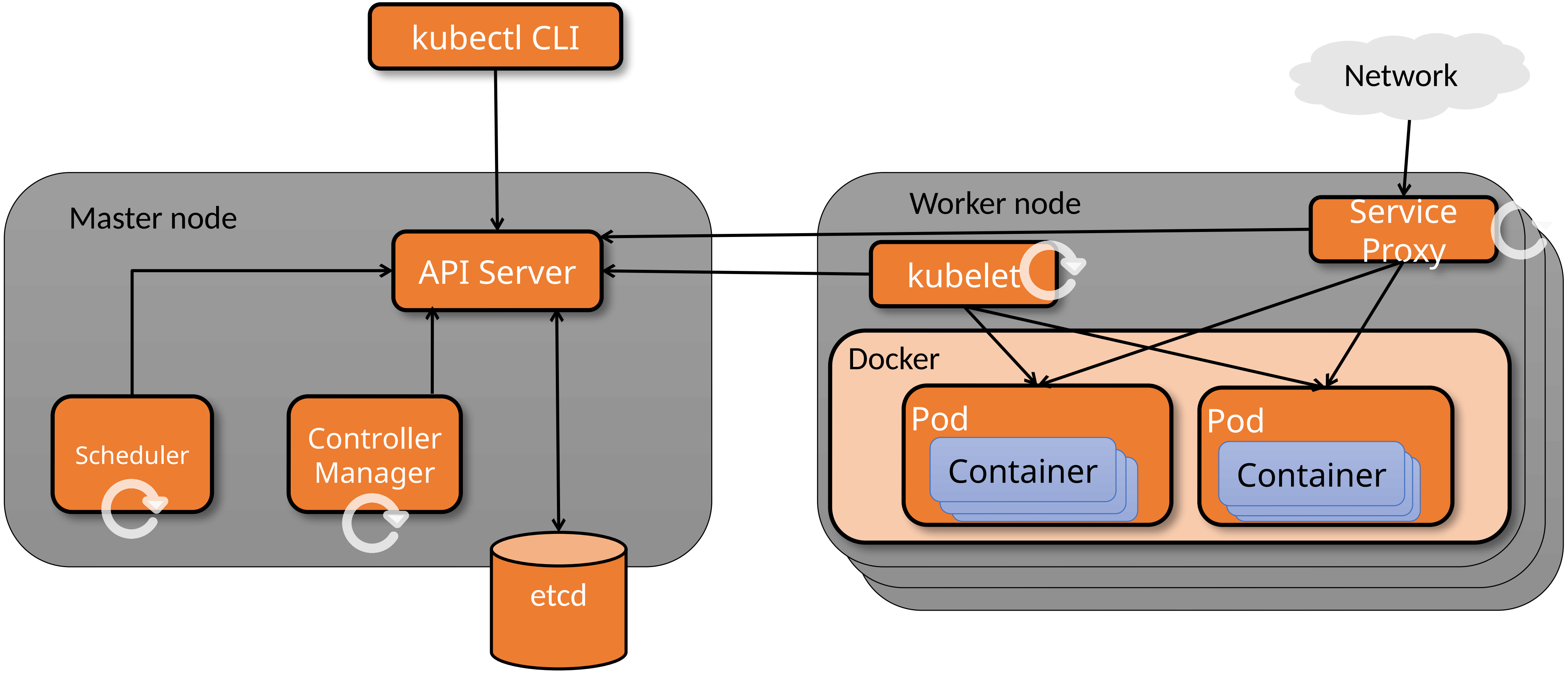
- Kubernetes is a container orchestration manager
  - Not the only manager
  - Docker Swarm does the same
  - Kubernetes is “industrial strength”
- Orchestration:
  - Manages “clusters” of containers
  - Provides service discovery
  - Manages scaling and failover
  - Works at the Ops level
  - Infrastructure as Code

# Kubernetes Cluster



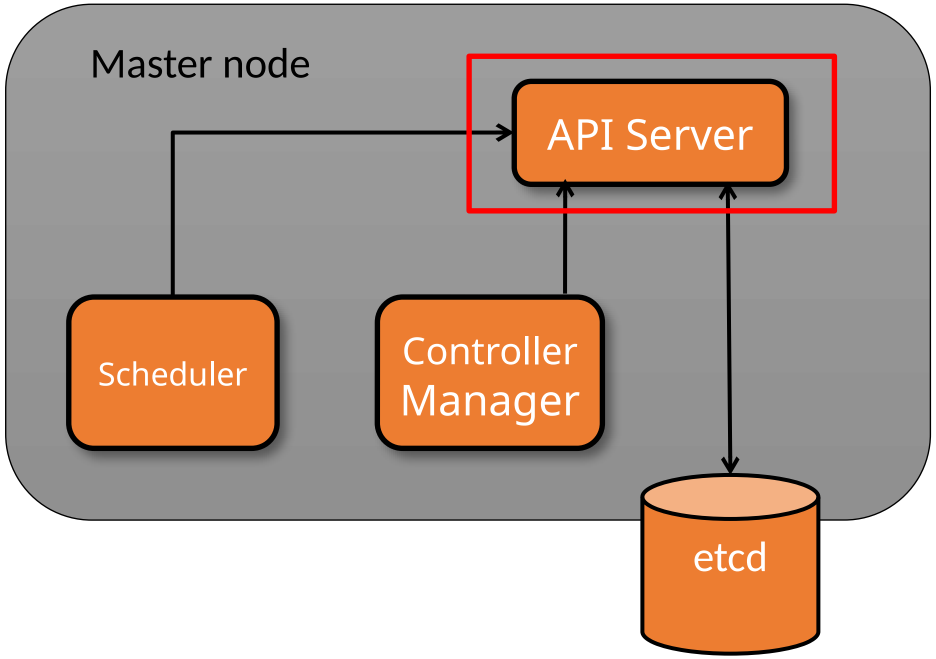
# Kubernetes Architecture

- Kubernetes nodes can be physical hosts or VM's running a container-friendly Linux (kernel > 3.10)



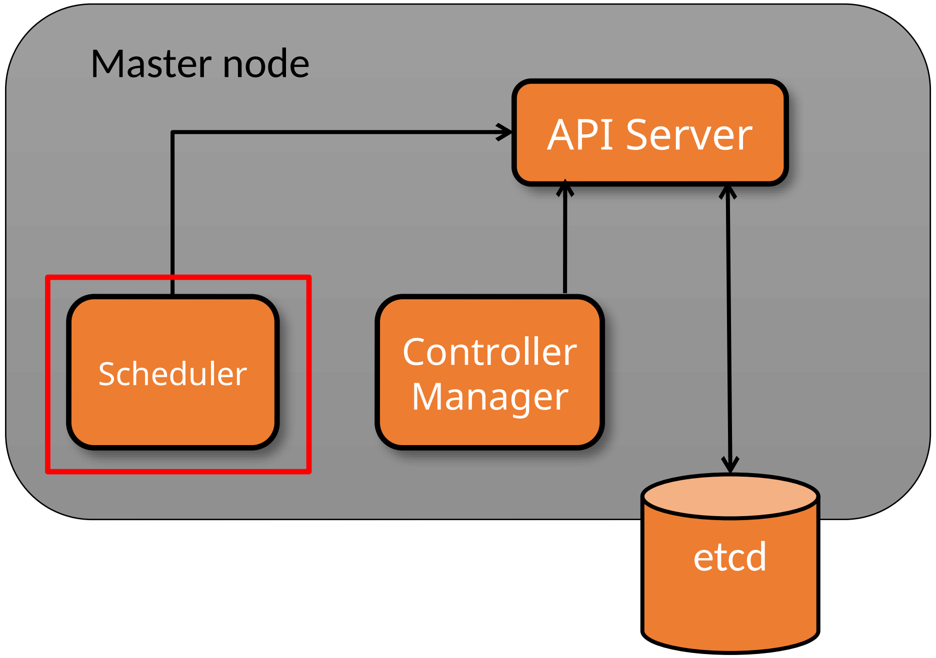
# Master Node Components

- API Server (kube-apiserver): exposes the Kubernetes REST API, and can be scaled horizontally



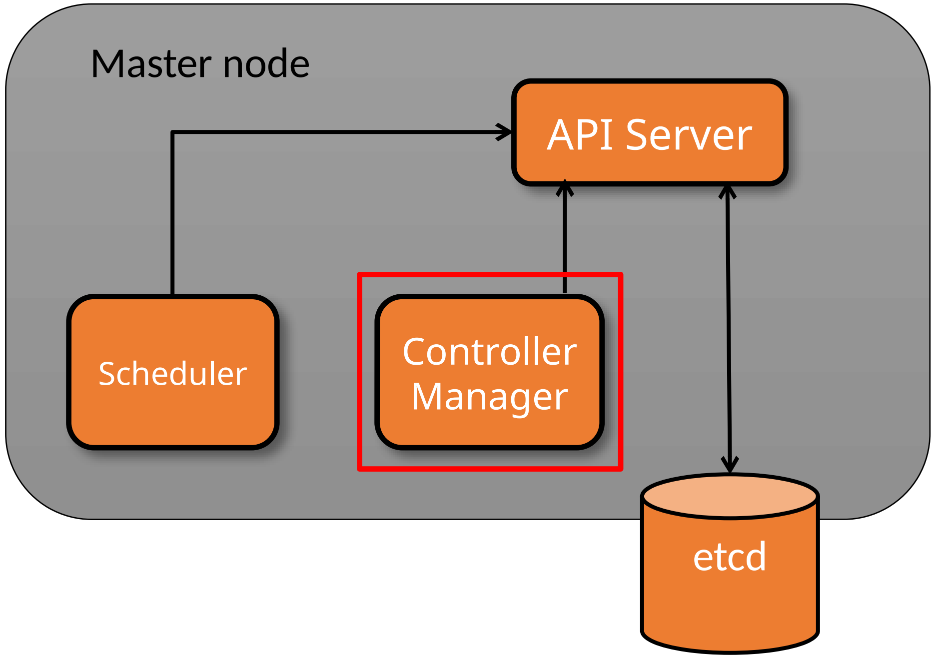
# Master Node Components

- Scheduler (kube-scheduler): selects nodes for newly created pods to run on



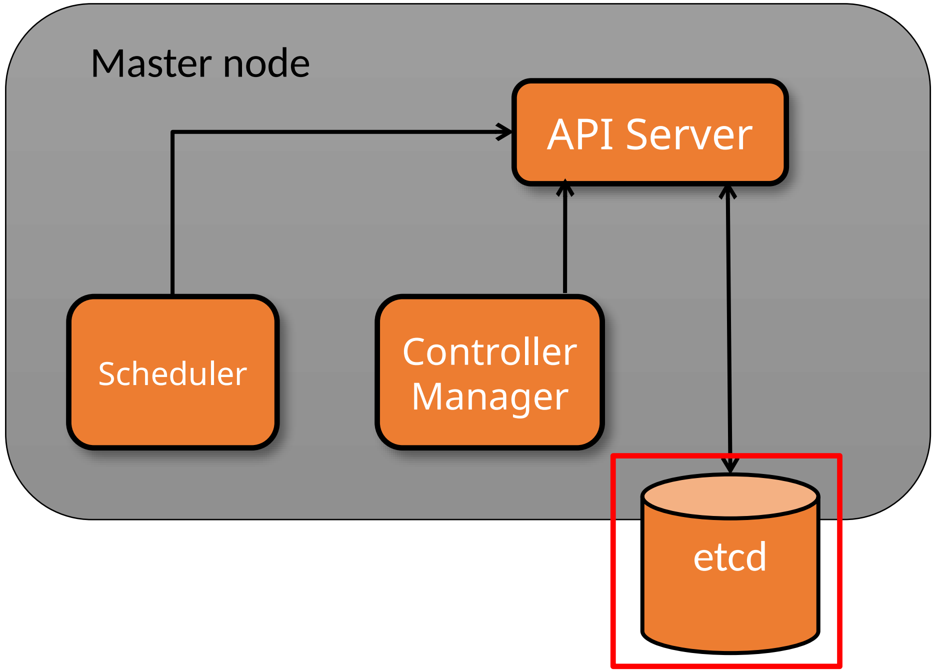
# Master Node Components

- Controller manager (kube-controller-manager): runs background controller processes for the system to enforce declared object states, e.g. Node Controller, Replication Controller,



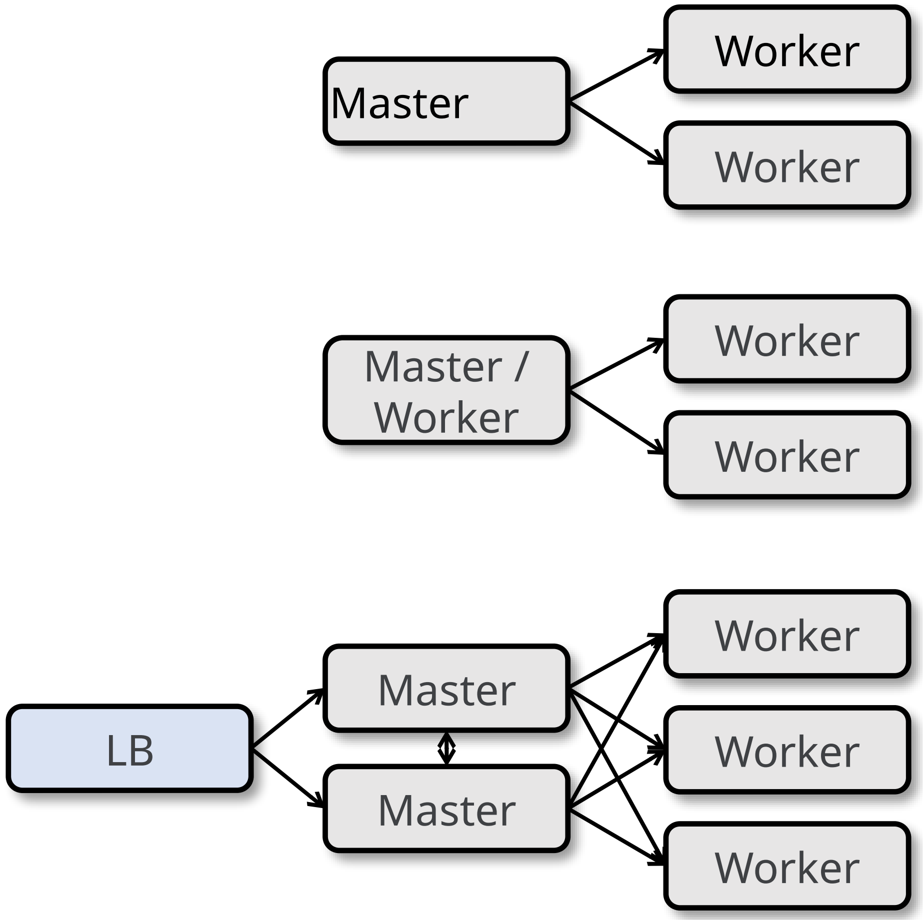
# Master Node Components

- 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



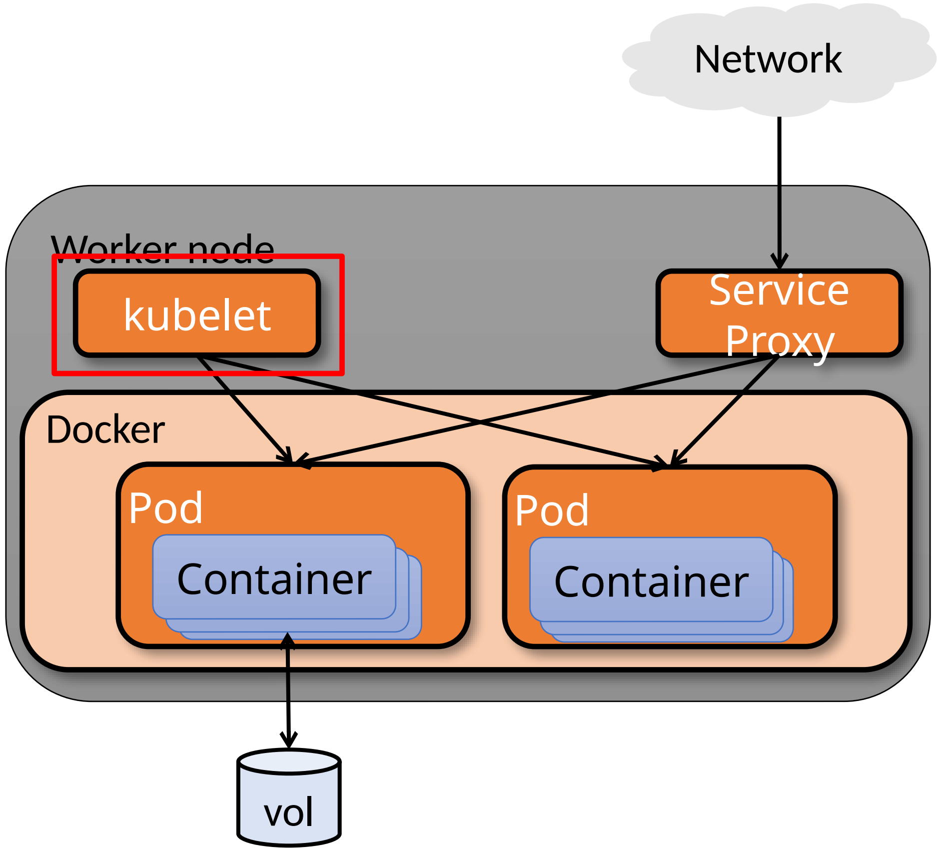
# Master Node Deployment Options

- Simple cluster has a single master node
  - At small scale, master may also be a worker node
- Cluster of master node replicas behind a loadbalancer
  - Kube-apiserver and etcd scale out horizontally
  - Kube-scheduler and kube-controller-manager use master election to run single instances at a time



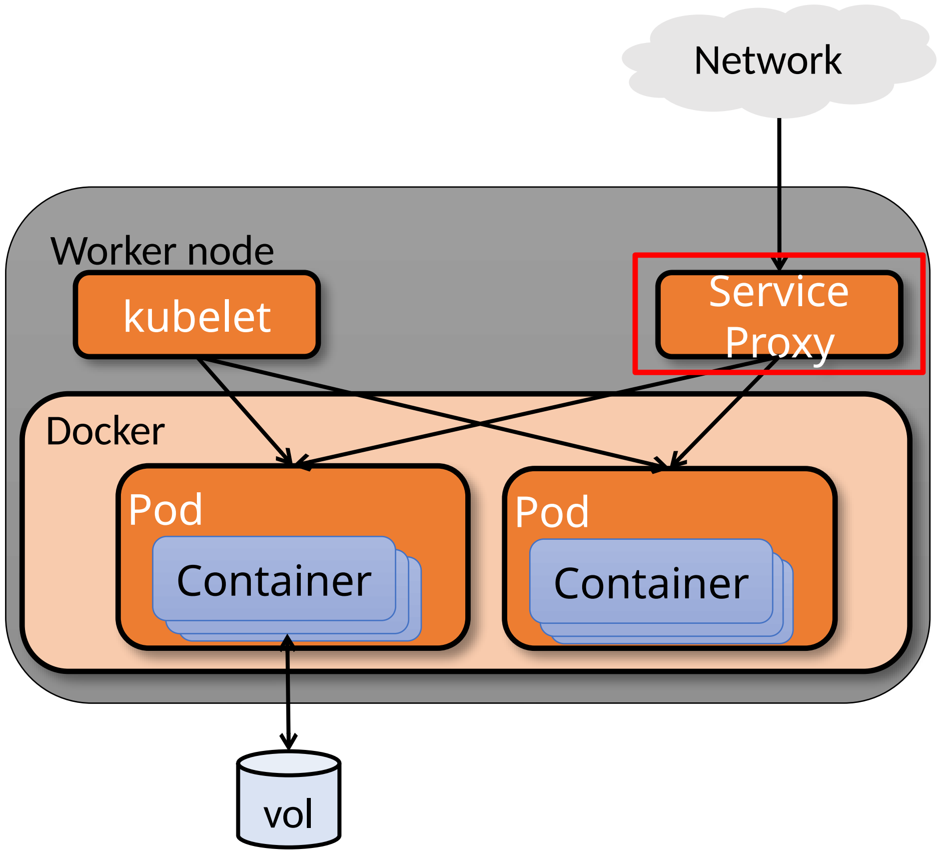
# Worker Node Components

- 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



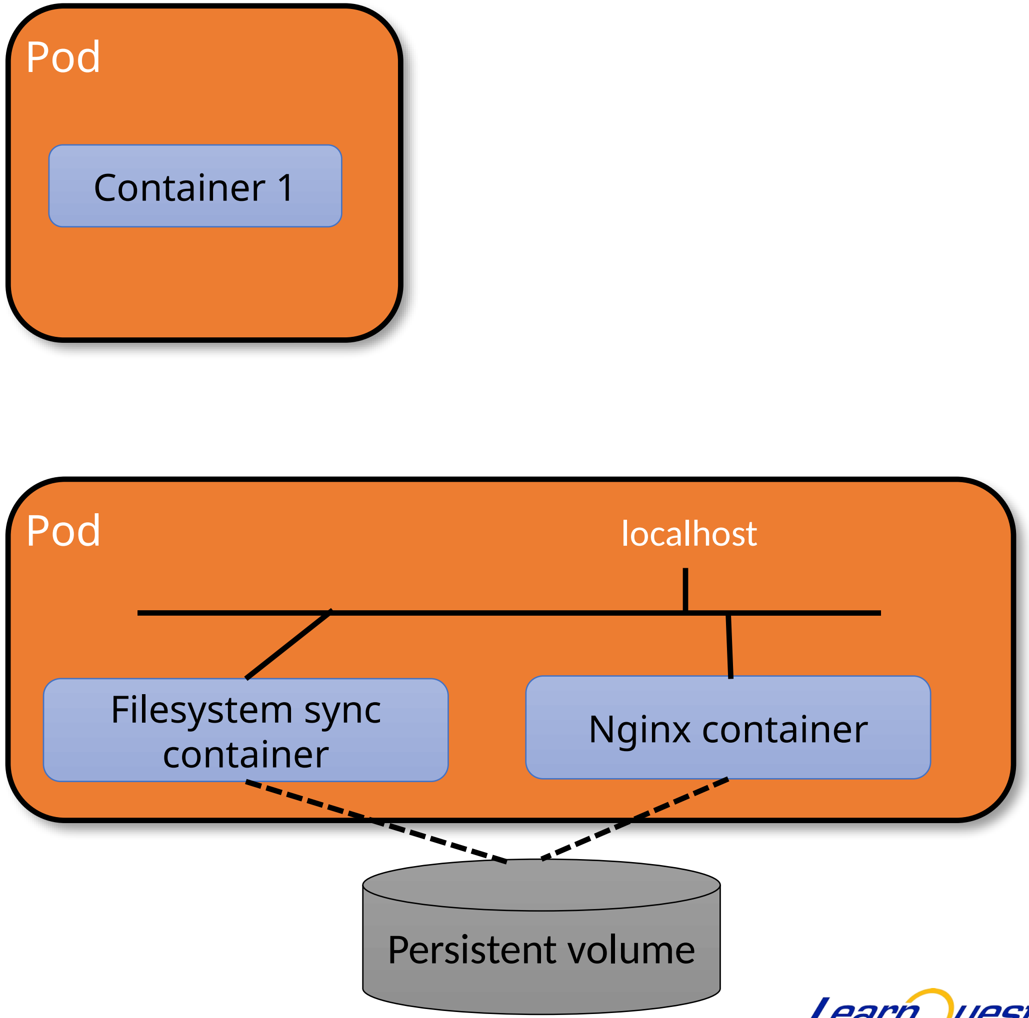
# Worker Node Components

- Service proxy (kube-proxy): enables K8s service abstractions by maintaining host network rules and forwarding connections
- Docker: runs the containers



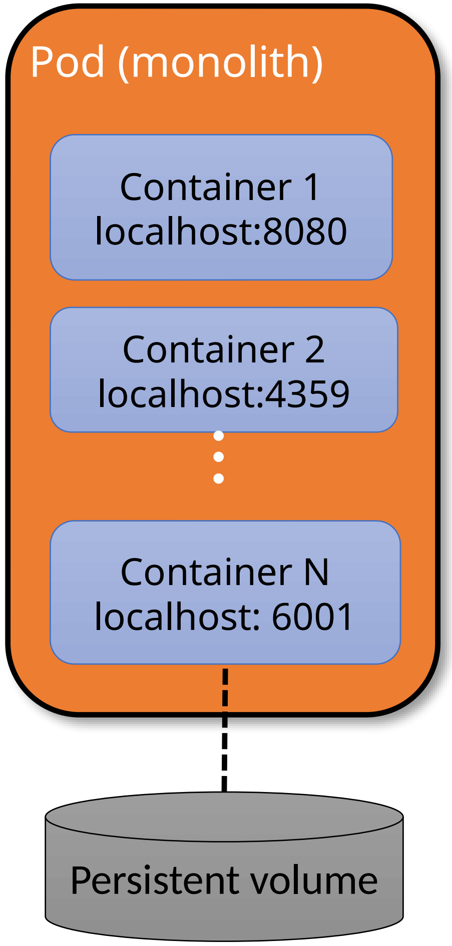
# Kubernetes Pod

- 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



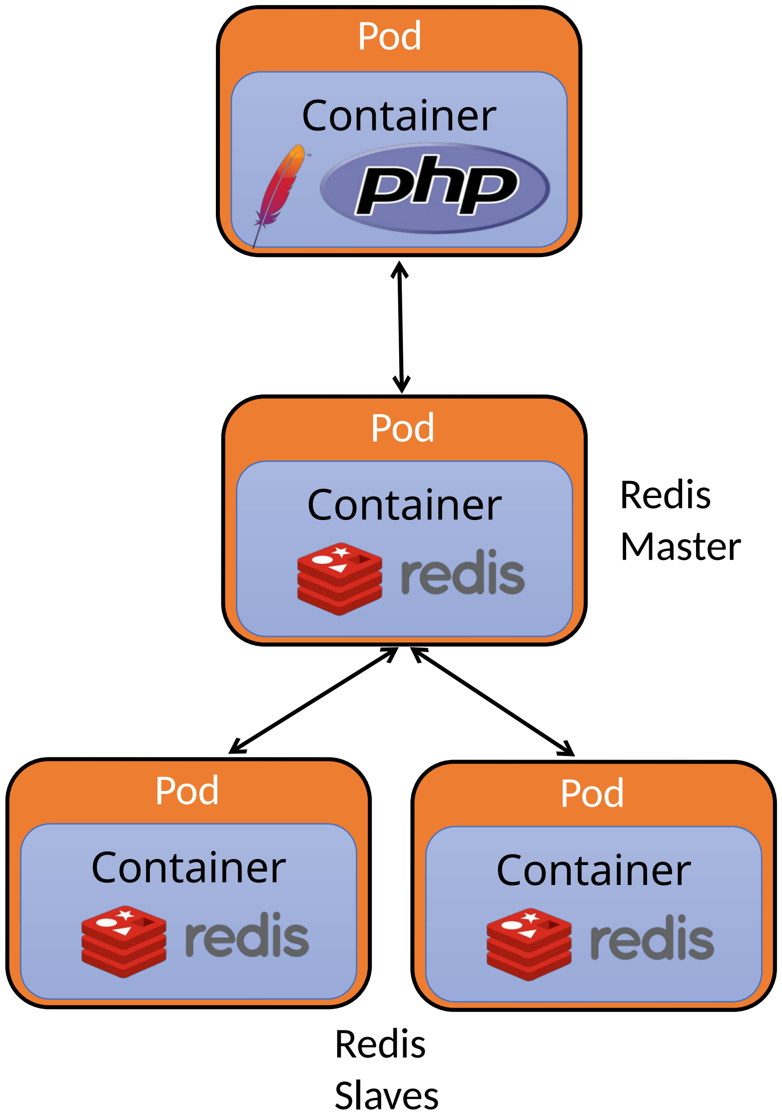
# Pod Deployment

- 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



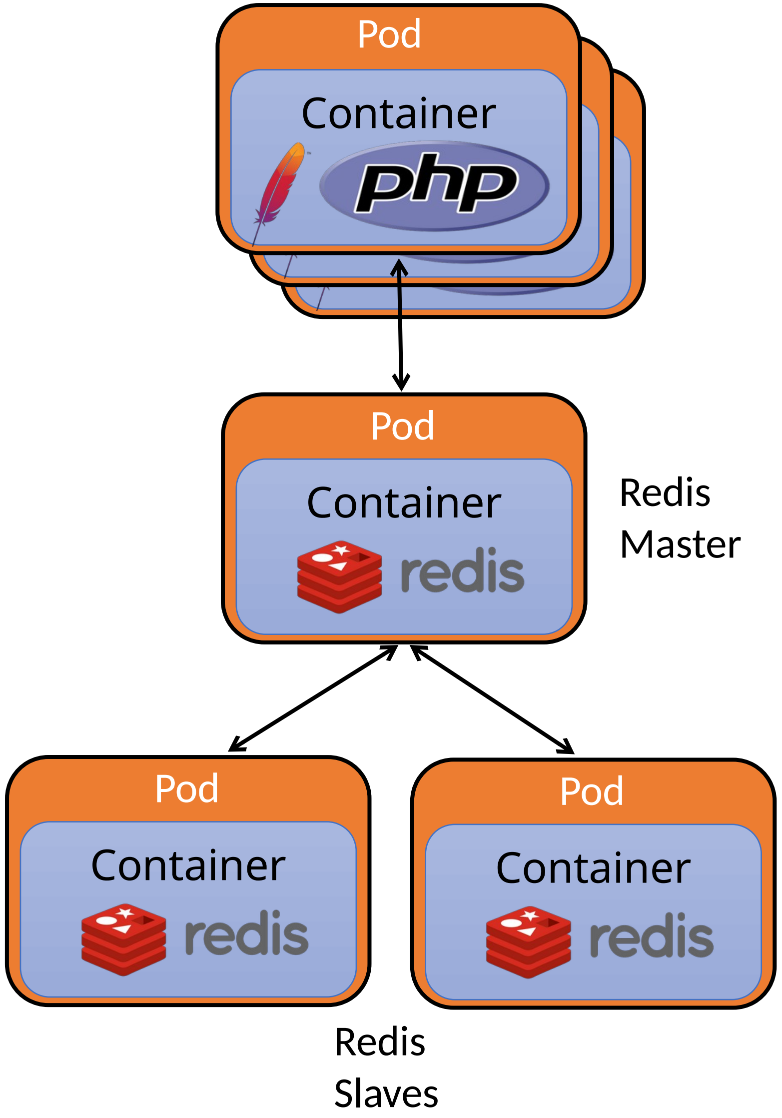
# Pod Deployment

- 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



# Pod Deployment

- 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

simplepod.yaml

```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 like creating Docker container directly

# Defining a Pod with Multiple Containers

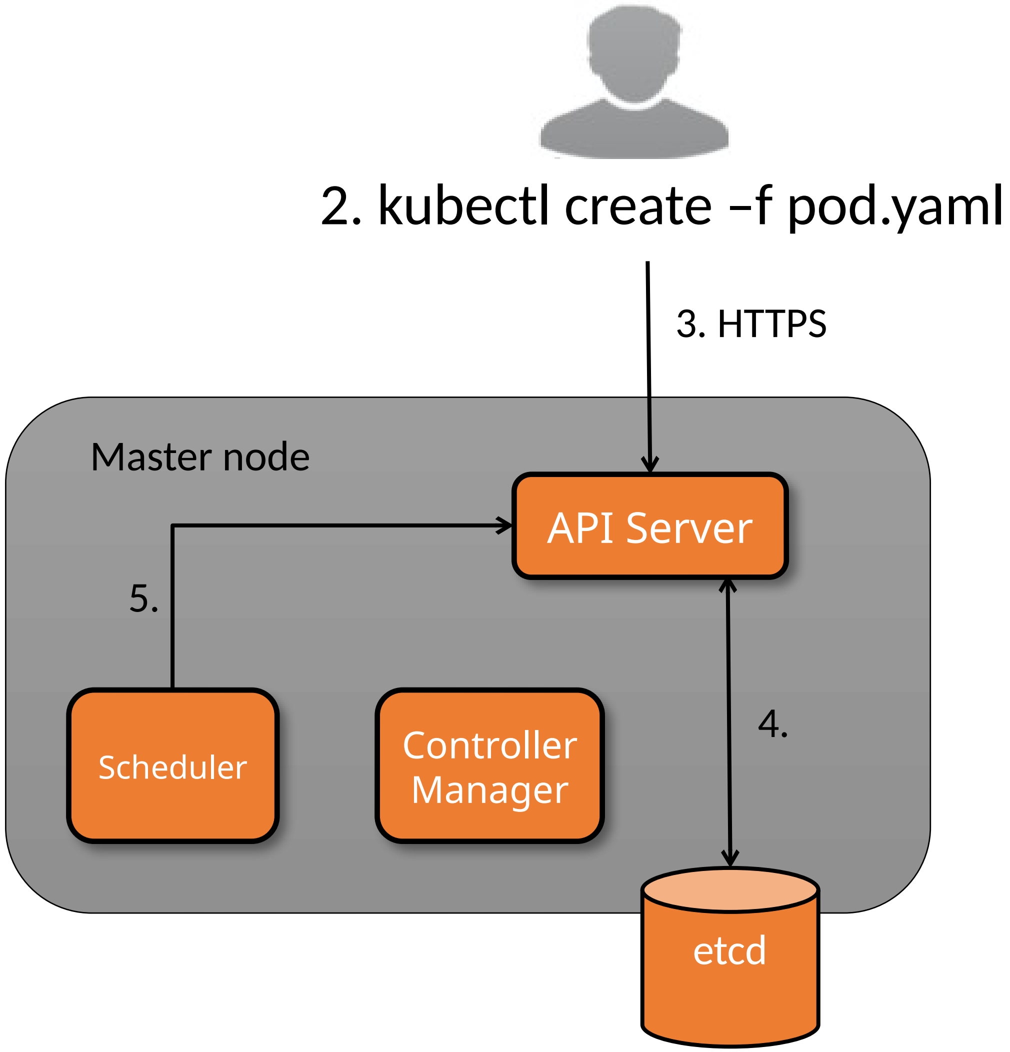
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
spec:
  containers:
  - image: tomcat
    name: test-tomcat
    ports:
    - containerPort: 8080
  - image: mysql
    name: test-mysql
    ports:
    - containerPort: 3306
```

→ multipod.yaml

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

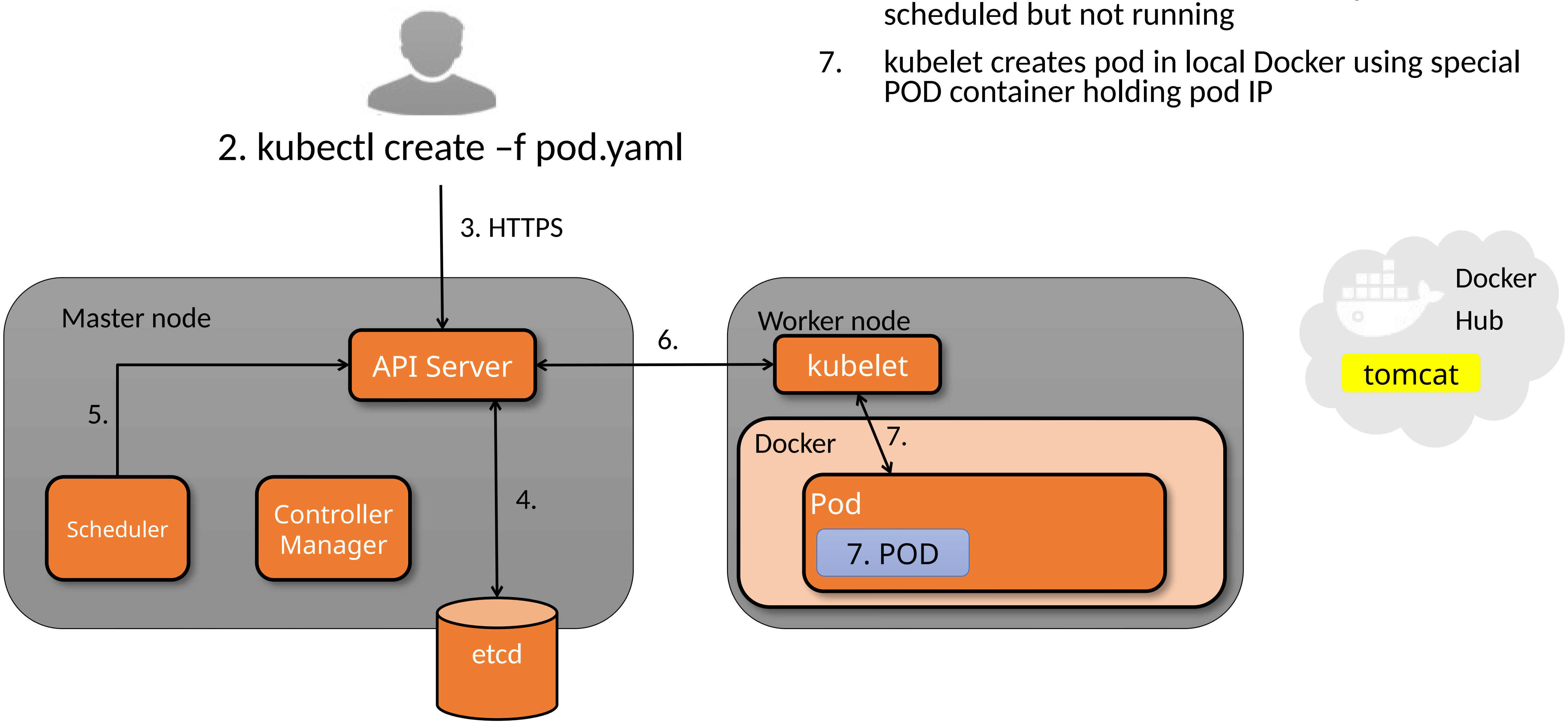
# Pod Creation Process

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
6. Selects node for pod to run on
7. Updates pod record via API with node assignment



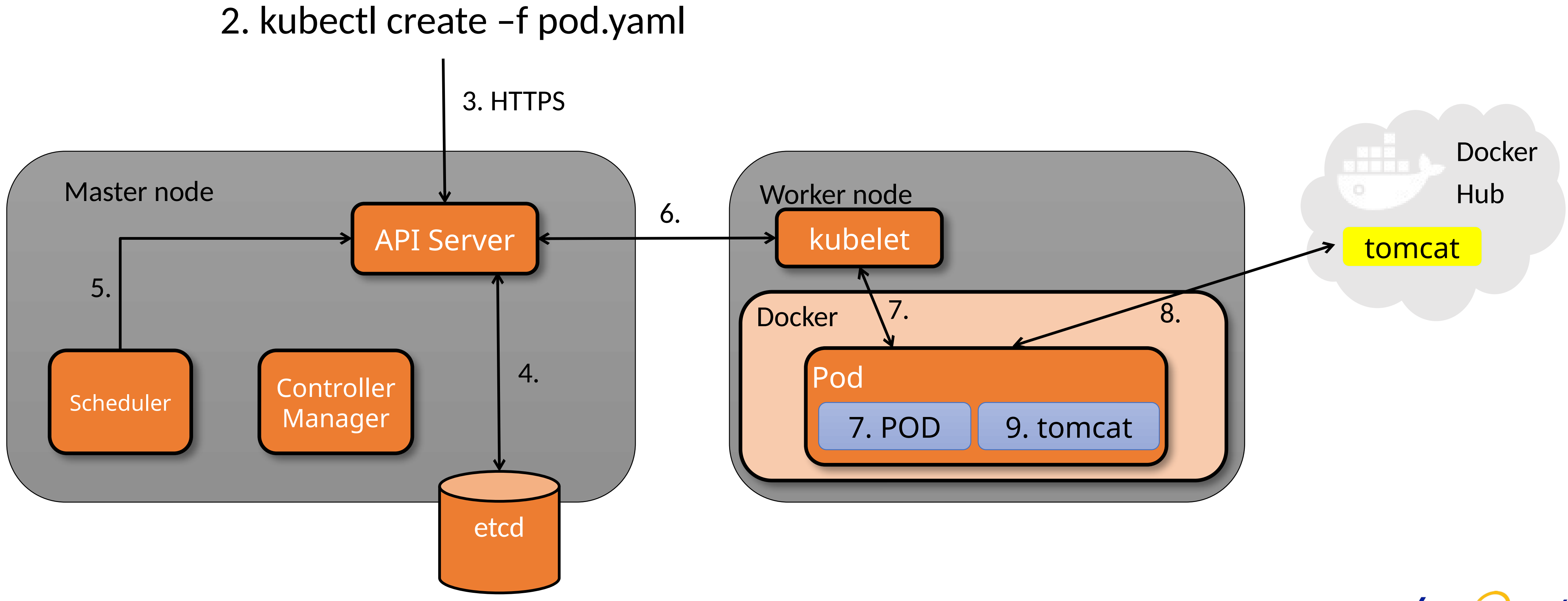
# Pod Creation Process

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

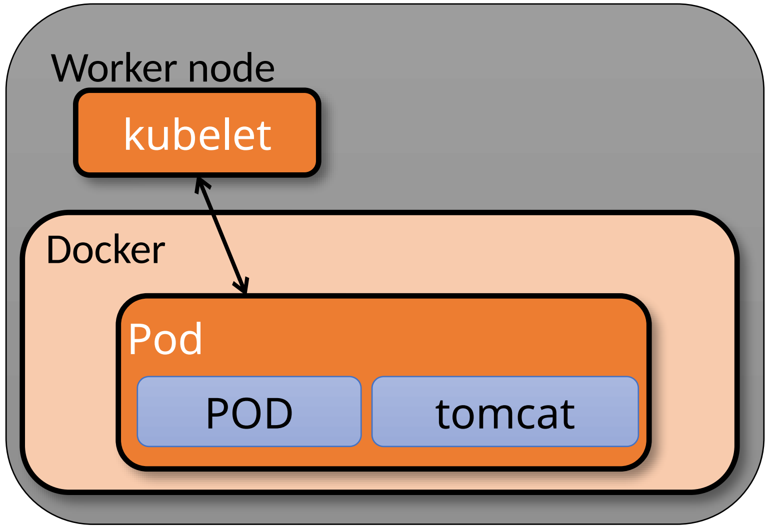
8. Kubelet pulls Docker image(s) for pod workload containers
9. Container(s) started and running on worker node



# Pod Lifecycles

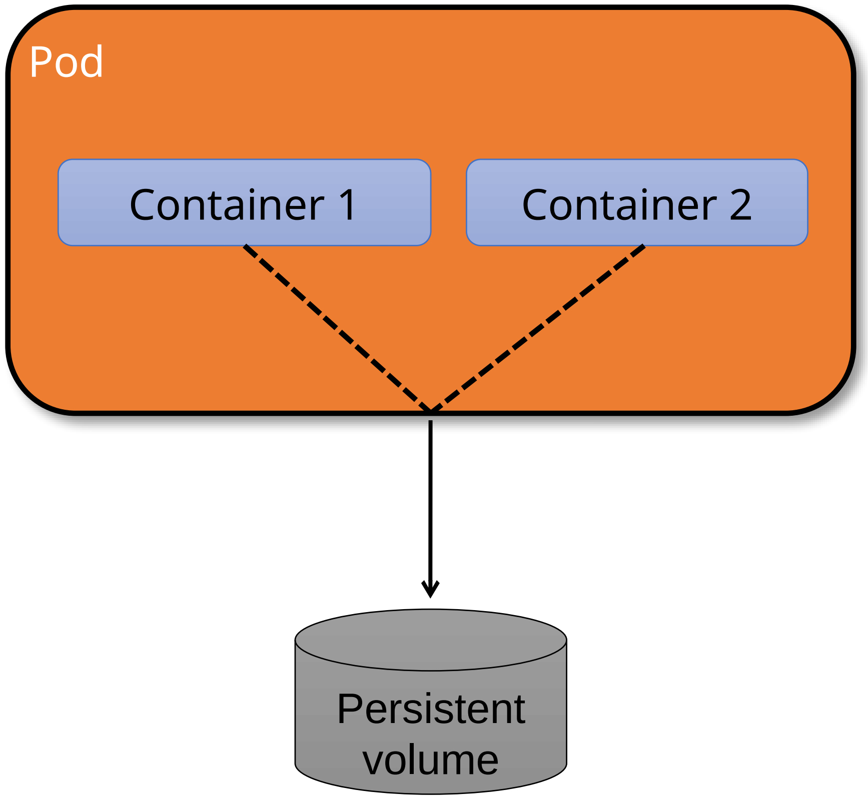
- By default, K8s Pods have an indefinite lifetime, which is not immortality
  - restartPolicy of Always by default
  - restartPolicy of Never or OnFailure also available for terminating jobs
- Node's kubelet will create and keep running containers for pods assigned to node, per the pod specs
- If a 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
```



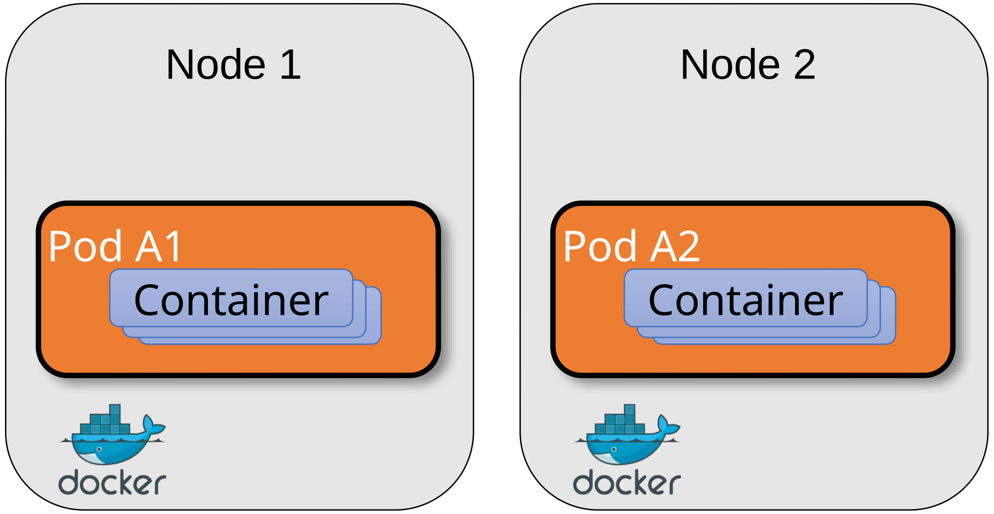
# Deleting Pods

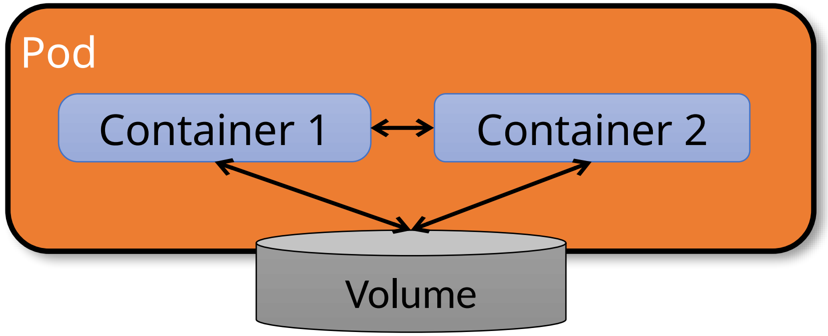
- When deleting a Pod, its containers will be removed and pod IP relinquished
- If an application needs to persist data, its pods must be configured to use persistent volumes for storage
- If a node dies, its local pods are also gone
- Best practice: use controller resources instead of managing pods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



# Kubernetes Pods

- Pod is 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

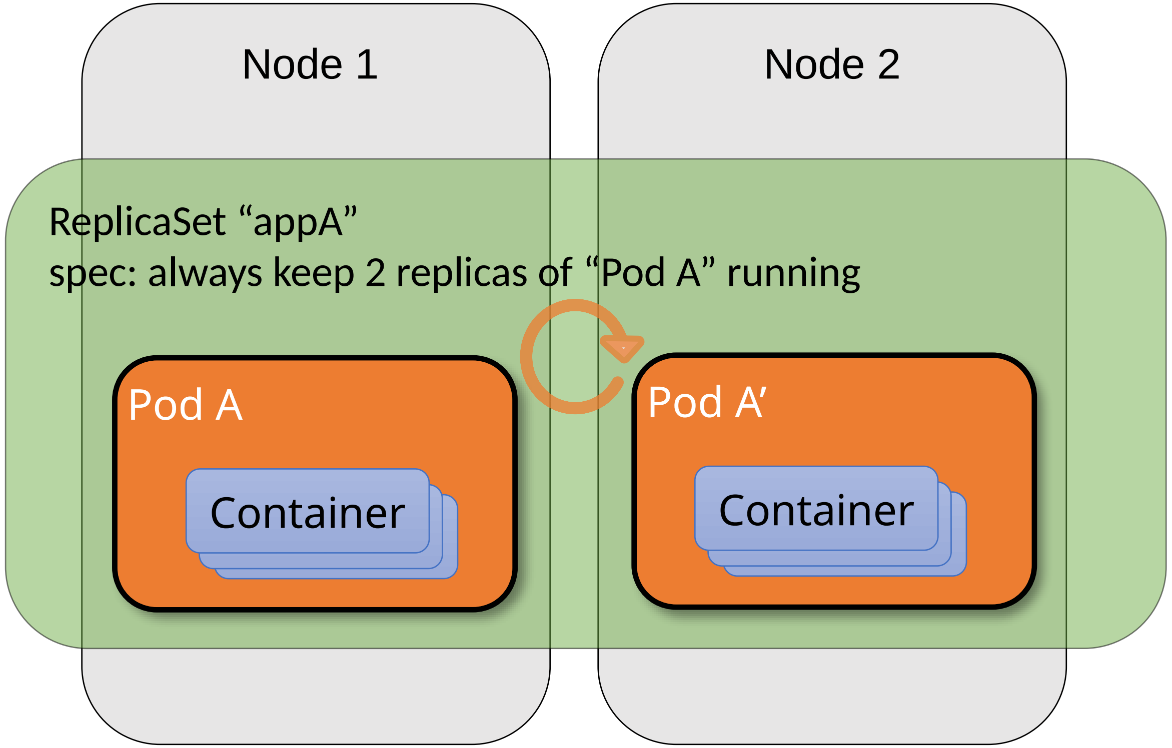


# Application Patterns

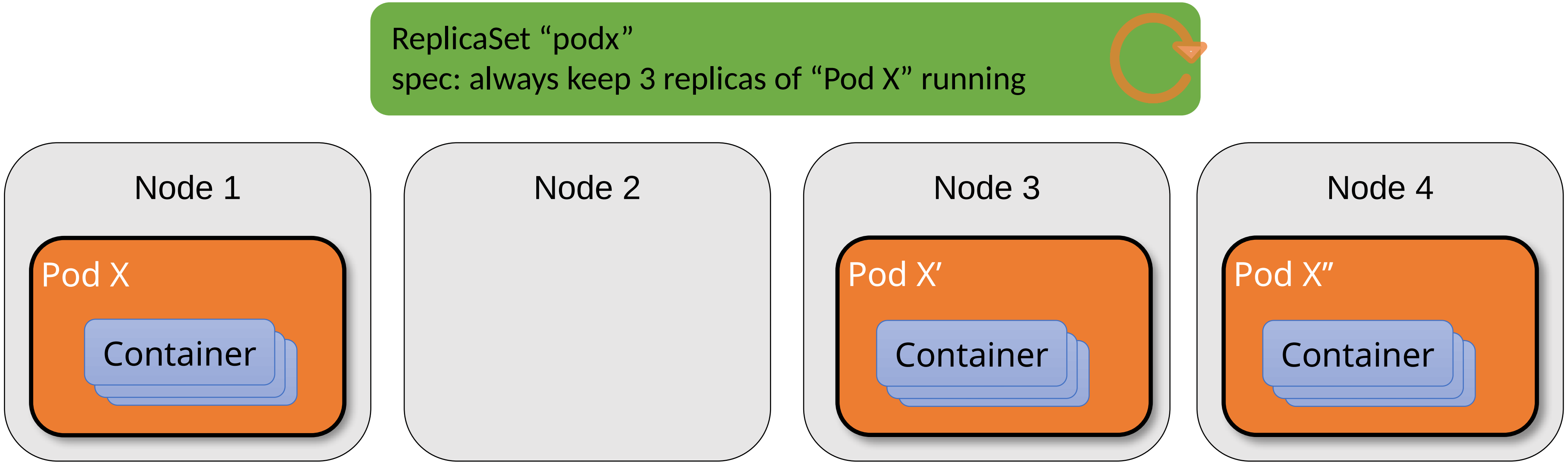
- K8s controller creates and manages Pods according to different application
  - ReplicaSets manage sets of replicas of stateless workloads to ensure availability
  - StatefulSets manage stateful workloads on stable storage to ensure consistency
  - DaemonSets manage workloads that must run on every node, or set of nodes
- Jobs manage parallel batch processing workloads

Controller example: ReplicaSet



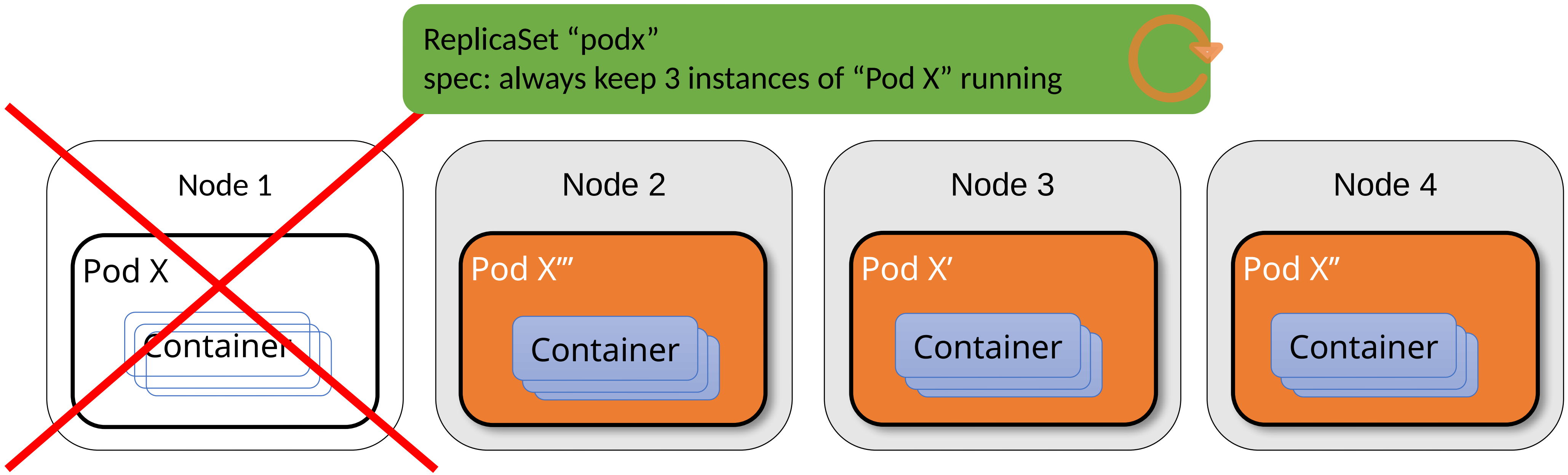
# ReplicaSets

- ReplicaSet configuration specifies how many instances of given Pod exist
- ReplicaSet used for web applications, mobile back-ends, API's



# 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



# 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)

### Controllers

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

### Cluster Resources

- Node
- Namespace
- Cluster

### Configuration

- Secret
- ConfigMap

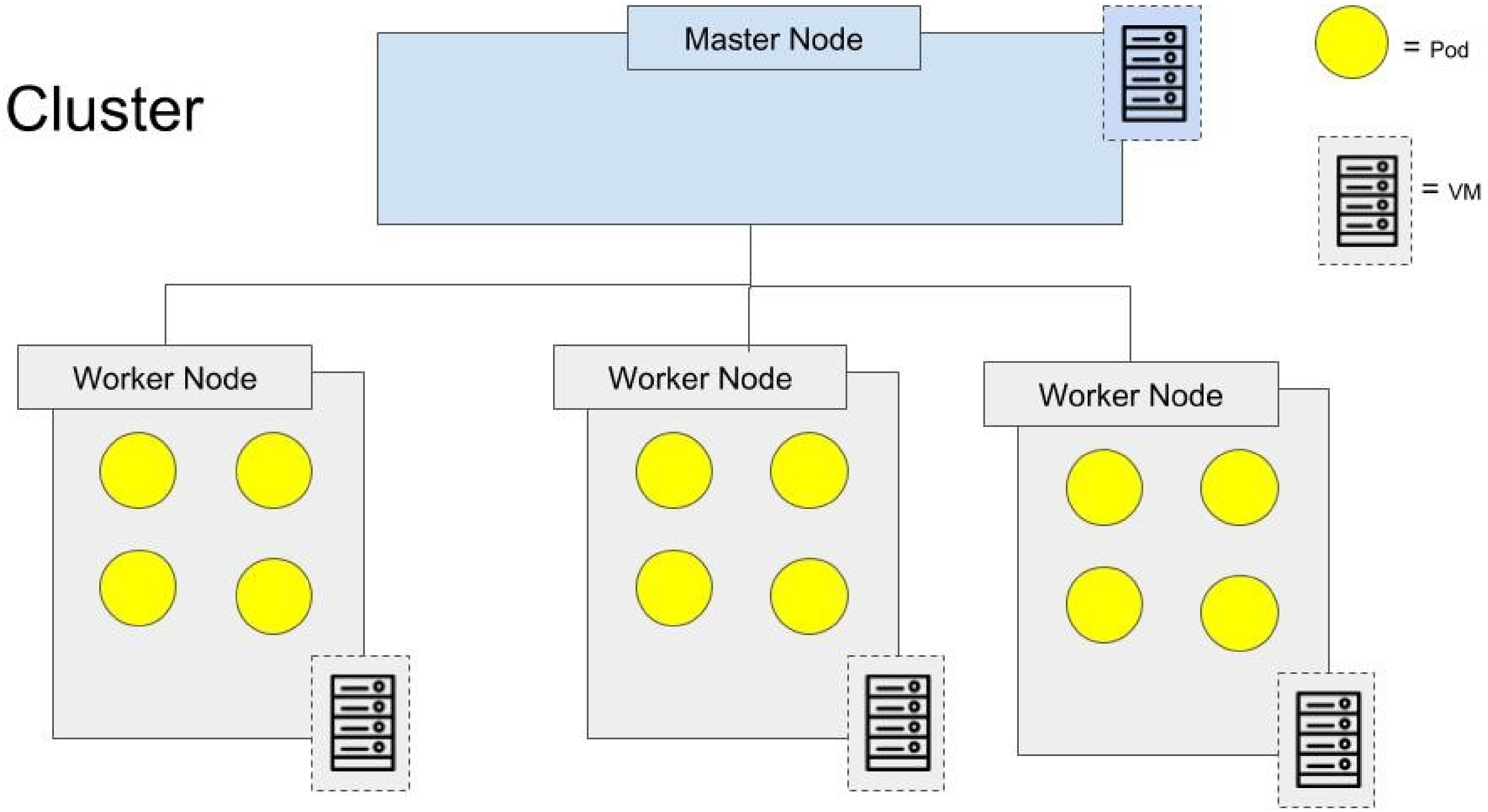
### Workload Persistent Storage

- PersistentVolume
- PersistentVolumeClaim

### Network Resources

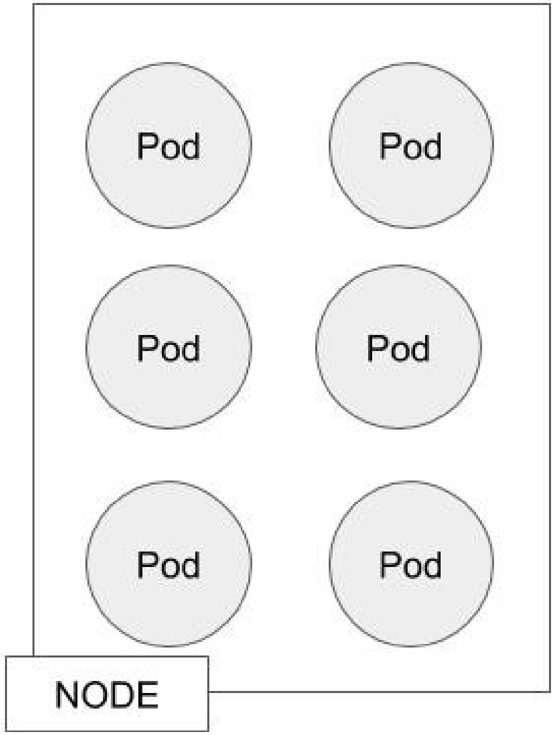
- Ingress
- NetworkPolicy

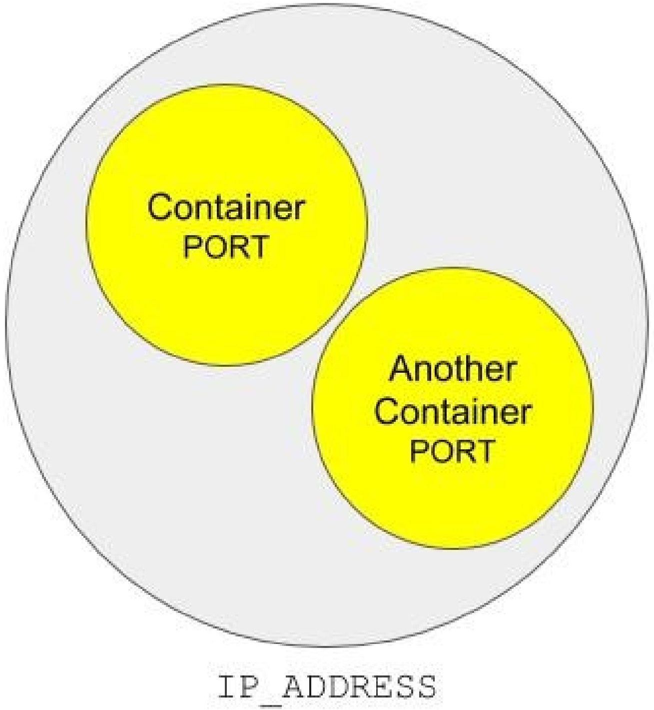
# The Essential Architecture Review



# The Essential Architecture Review

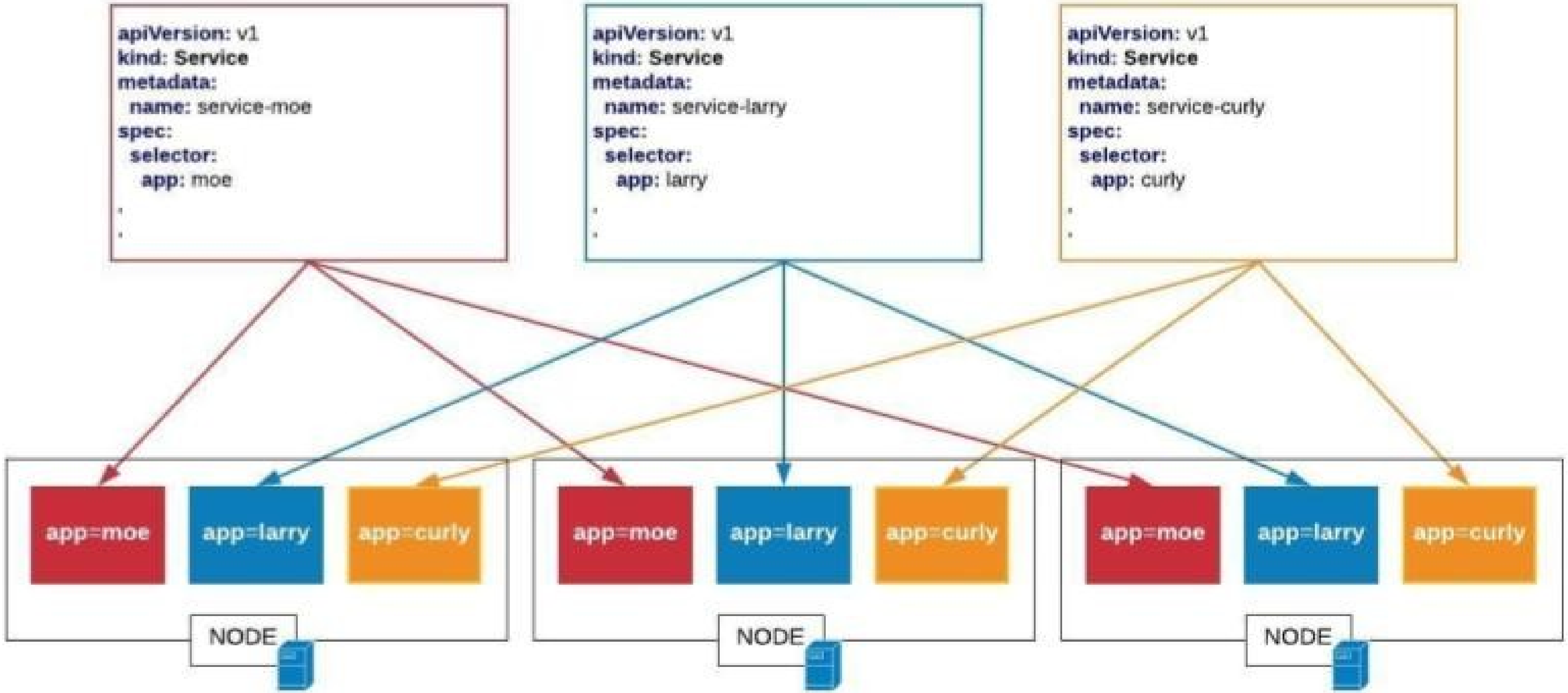
Pod



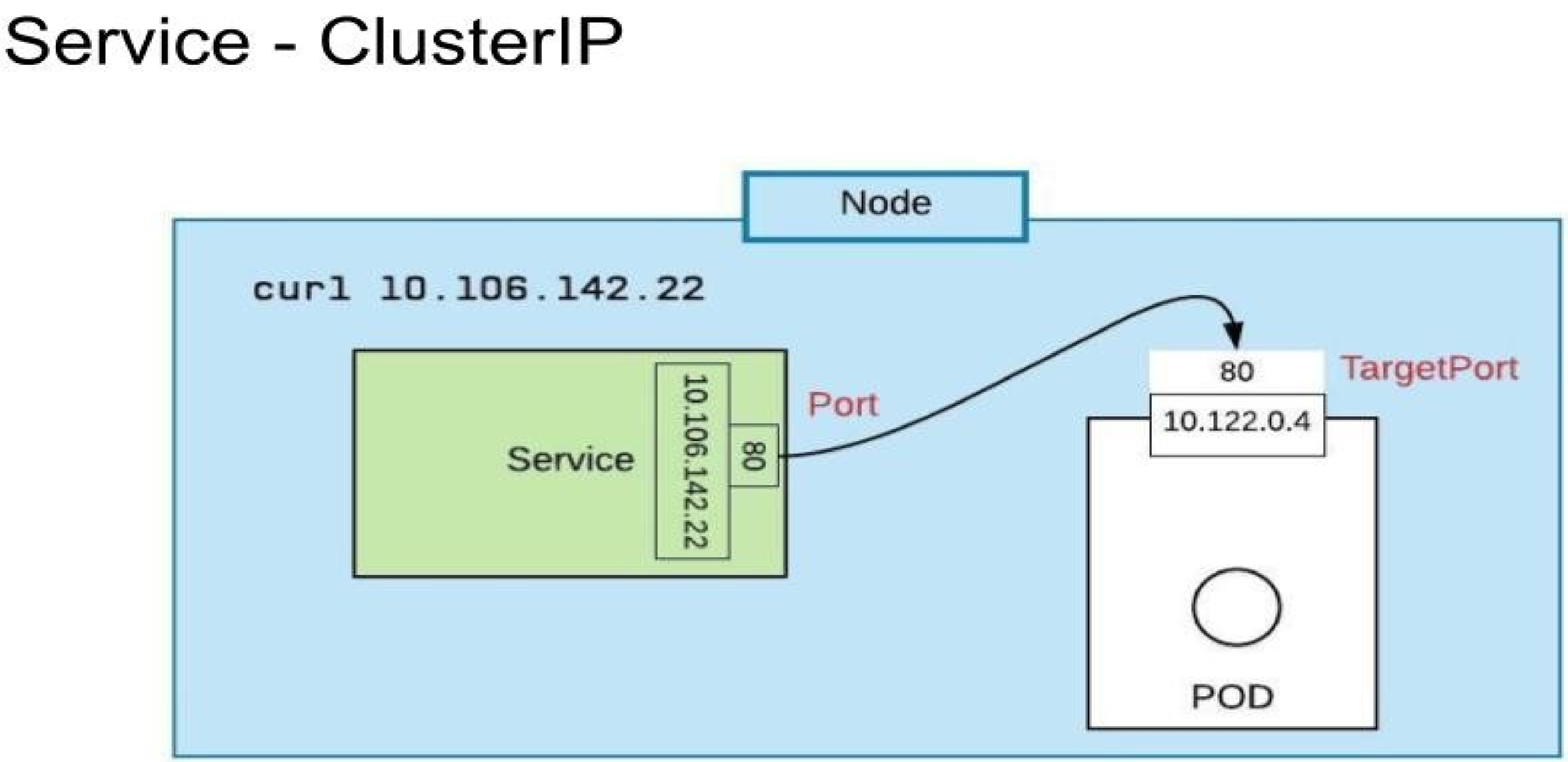



# Services and Service Binding

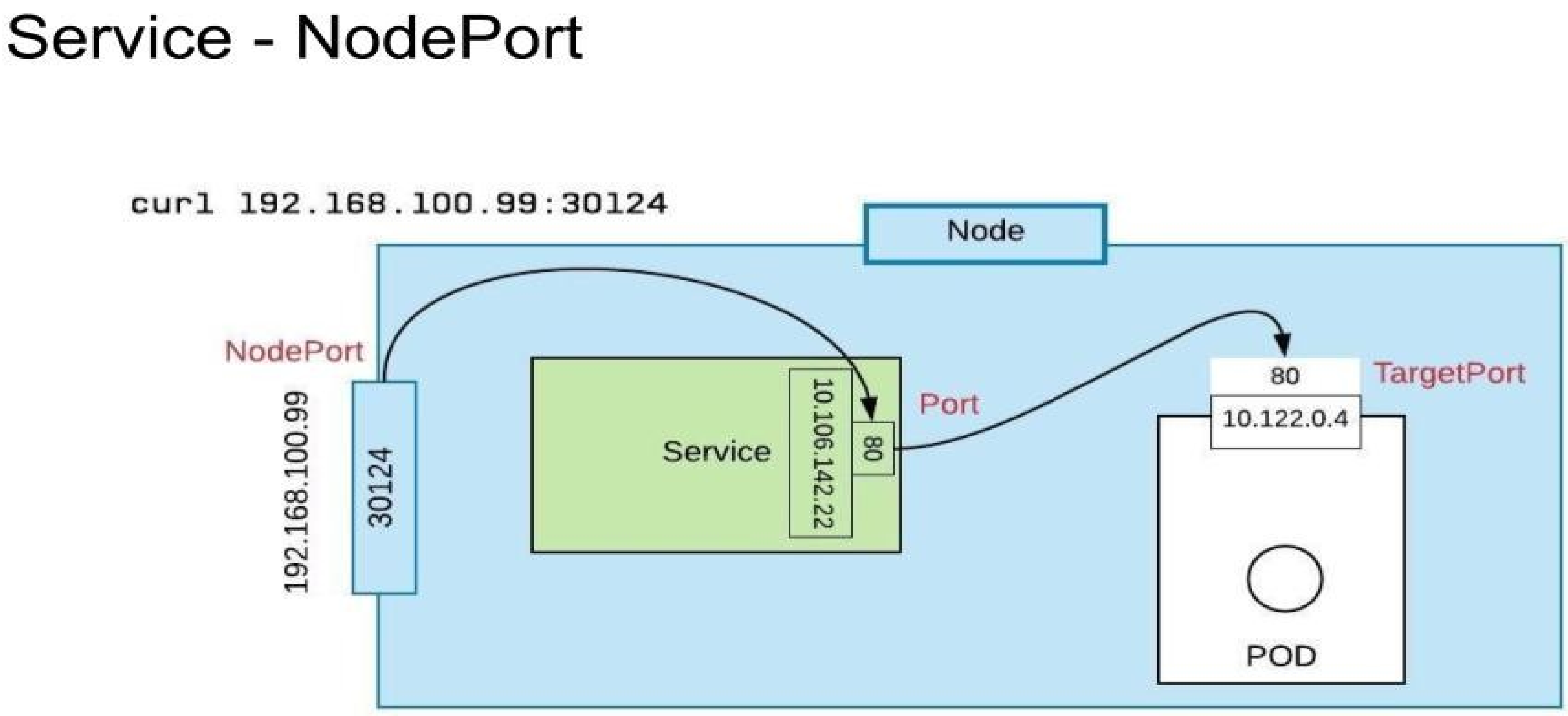
## Service



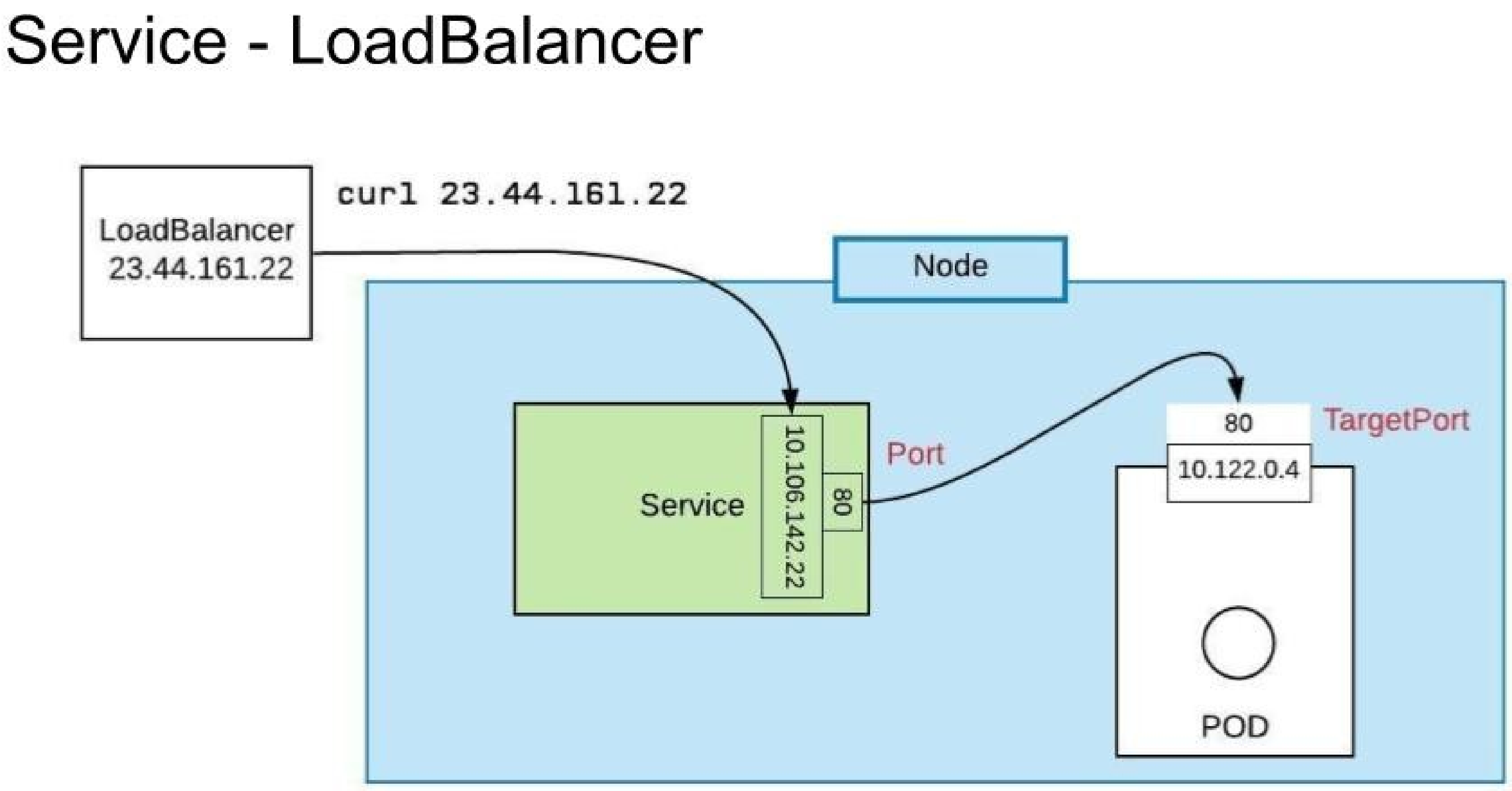
# Services and Service Binding



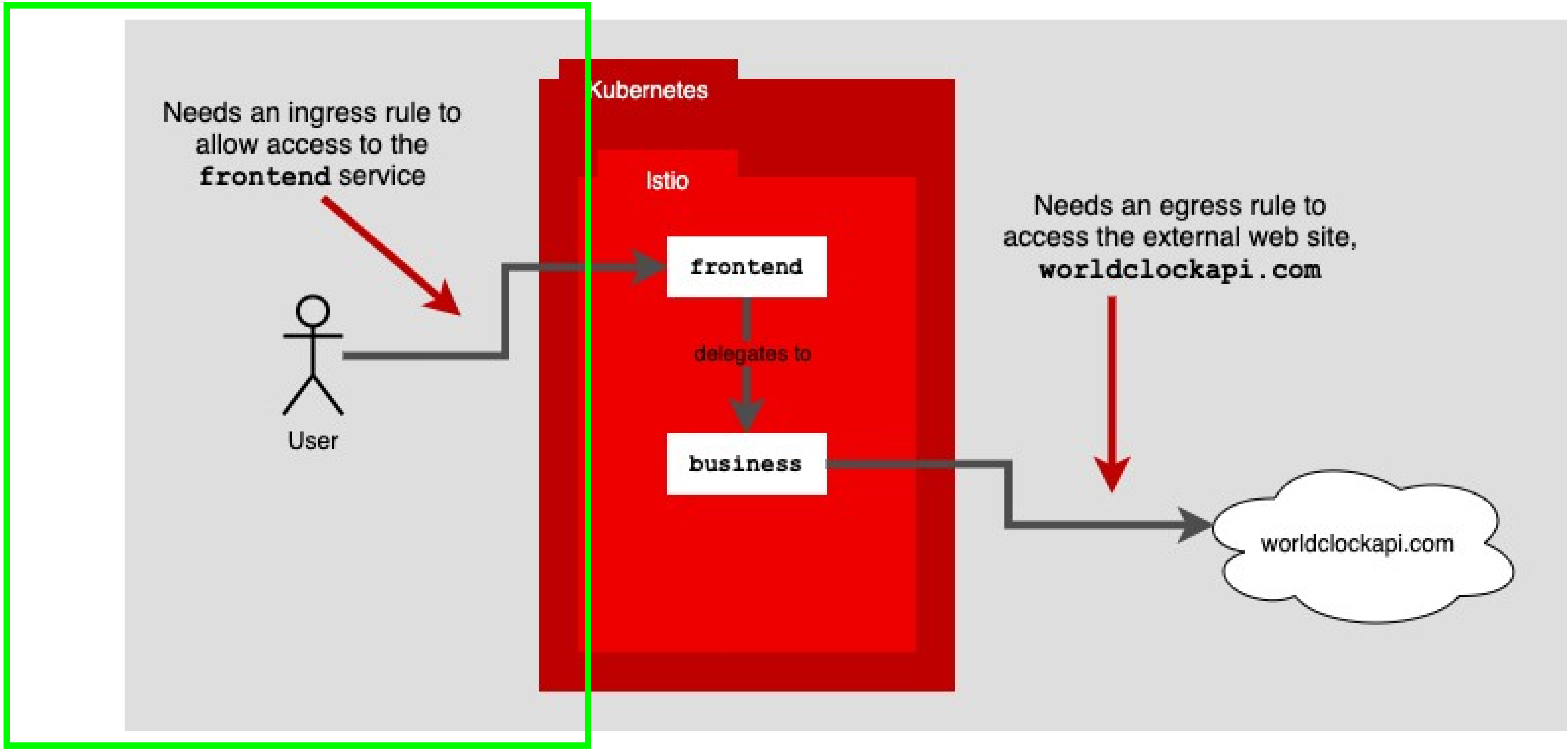
# Services and Service Binding



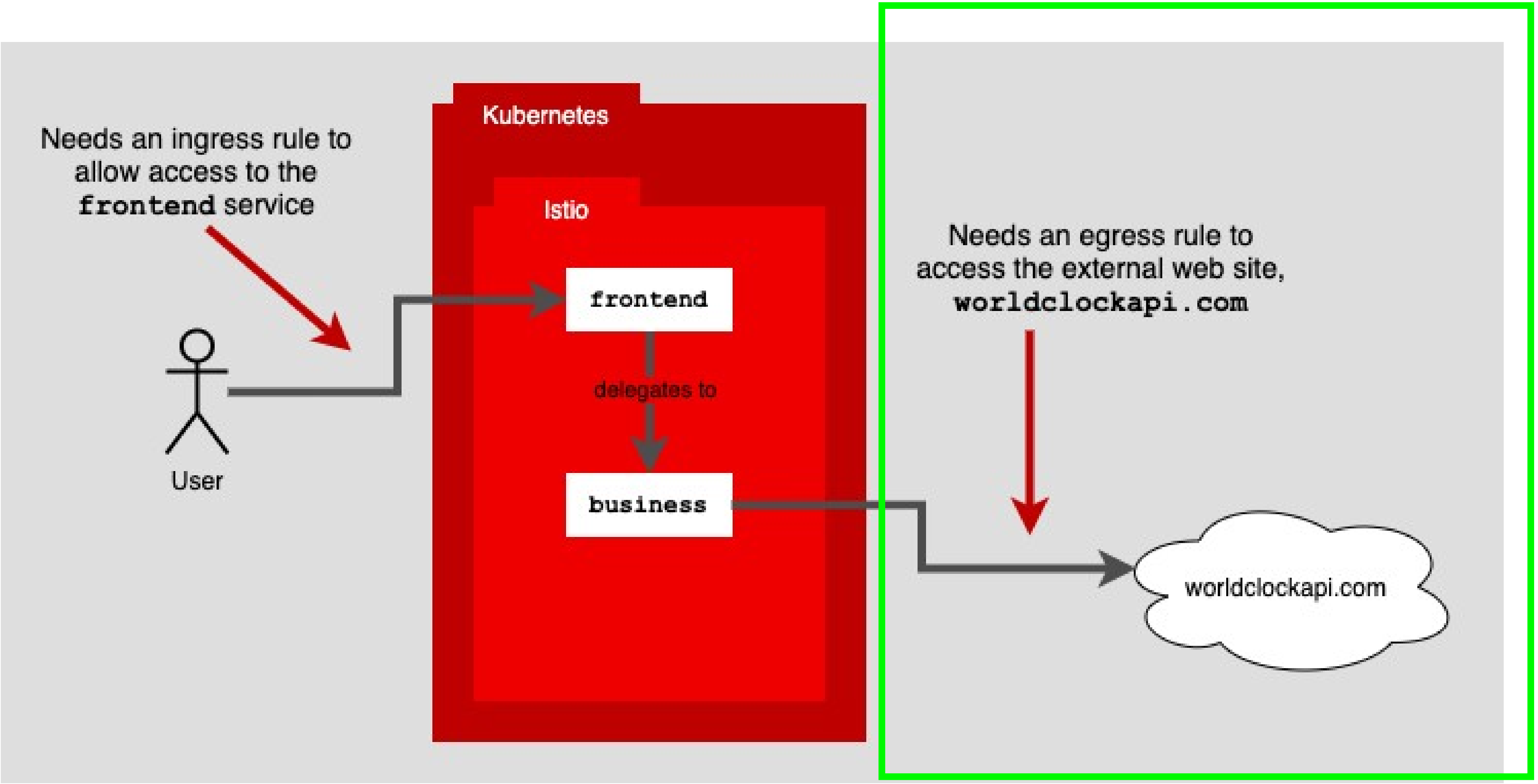
# Services and Service Binding



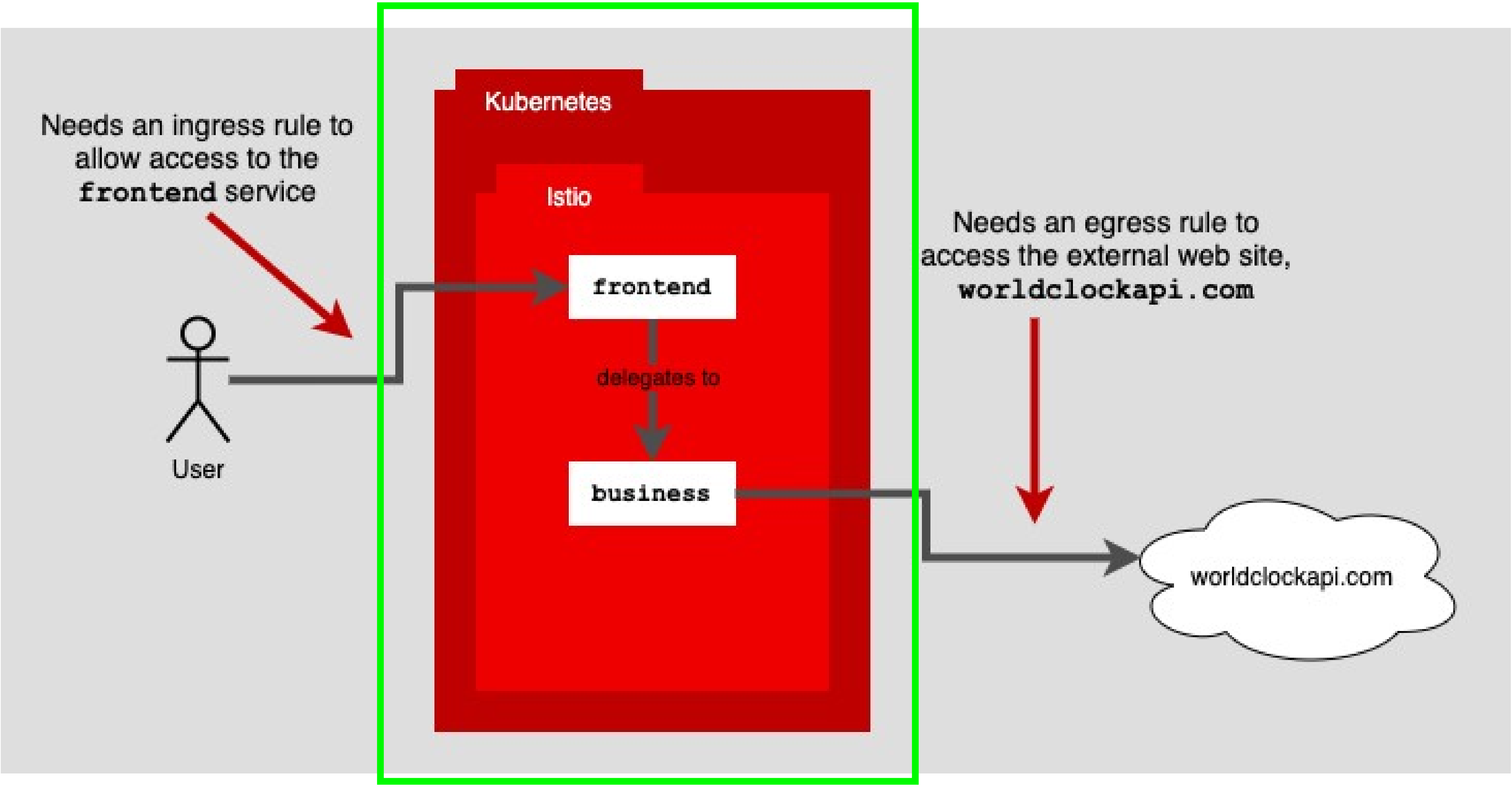
# Kubernetes: Ingress



# Kubernetes: Egress



# Service Mesh



# Working with a Service Mesh

- What does a Service Mesh do?
  - Enforces routing rules and restrictions
  - Binds service to implementation instance
  - Determines optimal instance
  - Keep track of  performance aspects, e.g., latency
  - Coordinates retries and failures
  - Ejects failing instances from the load balancing pool
  - Provides monitoring, e.g. tracing and performance metrics

# Sampling of Products

LINKERD

HashiCorp Consul

Istio

dapr

Service Fabric Mesh!

# Questions?

# End Module