TEKsystems Global Services | WORKFORCE DEVELOPMENT

Configuration Management and DevOps: Automate Infrastructure Deployments

PARTICIPANT GUIDE

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 content in any form or medium outside of the training program

4

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

Core Kubernetes

Logistics

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

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

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

- Miscellaneous
- Courseware
- Bathroom

Course Objectives

- Explain the benefits of using containers
- Build, run, and manage containers
- Configure container networks
- Store persistent data in volumes
- State the function and purpose of Kubernetes
- Describe the Kubernetes Architecture
  - Cluster components
  - Pods, Deployments, Services
- Build and deploy applications

The Training Dilemma

Hi!

Jason Smith

Cloud Consultant with a Linux sysadmin background.
Focused on cloud-native technologies: automation, containers & orchestration

LinkedIn
https://www.linkedin.com/in/jruels/

mail
jason@innovationinsoftware.com

github
https://github.com/jruels

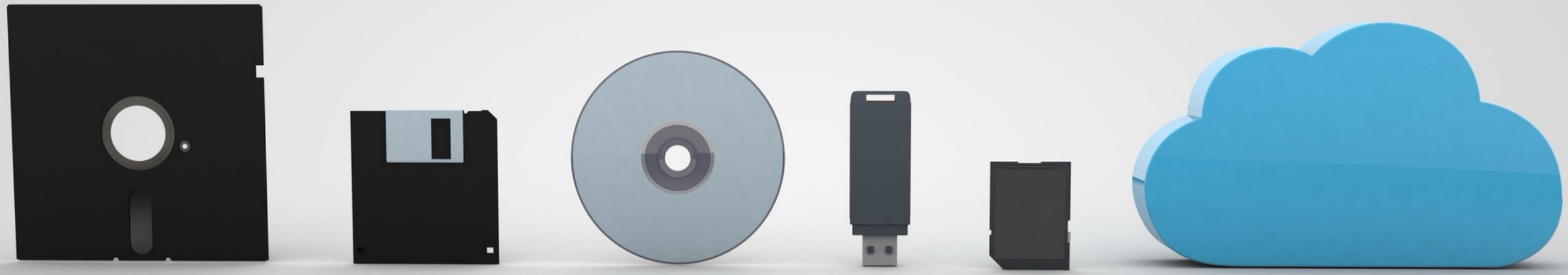
Expertise

- Cloud
- Automation
- CICD
- Docker
- Kubernetes

Introductions

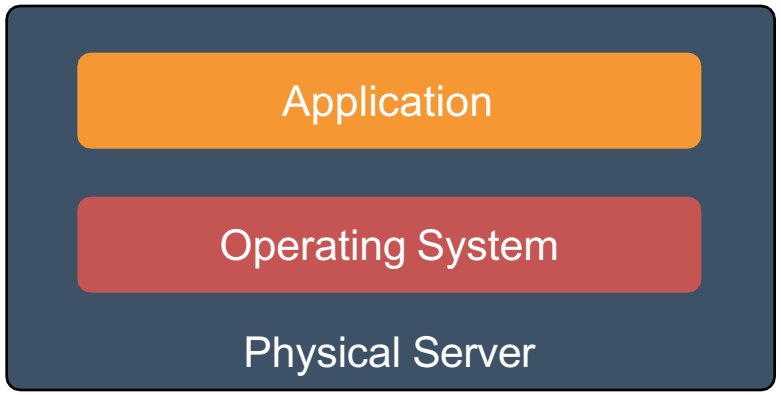
- Name
- Job Role
- Your experience with (scale 1 - 5)
  - Docker/Containers
  - Kubernetes
- Expectations for course (please be specific)

Data Center Evolution



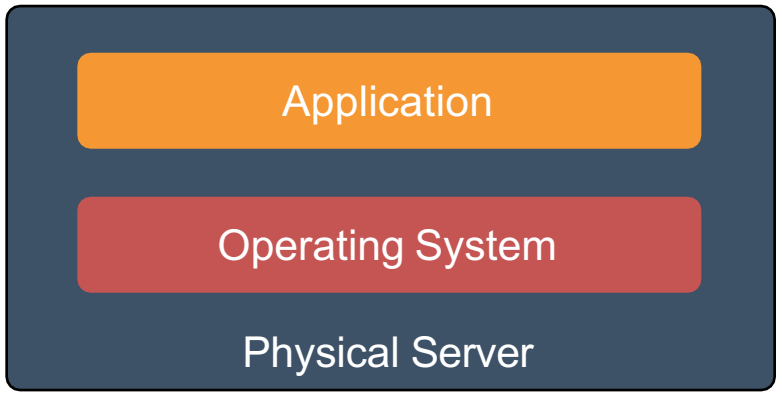
Monolithic

Monolithic Server Architecture



One physical server, one application

Monolithic Server Architecture



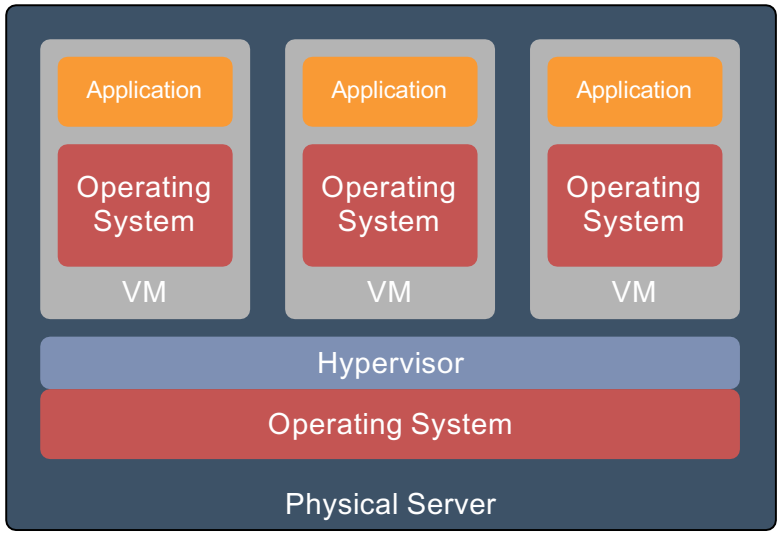
One physical server, one application

Problems

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

Virtualized

Virtualized Infrastructure

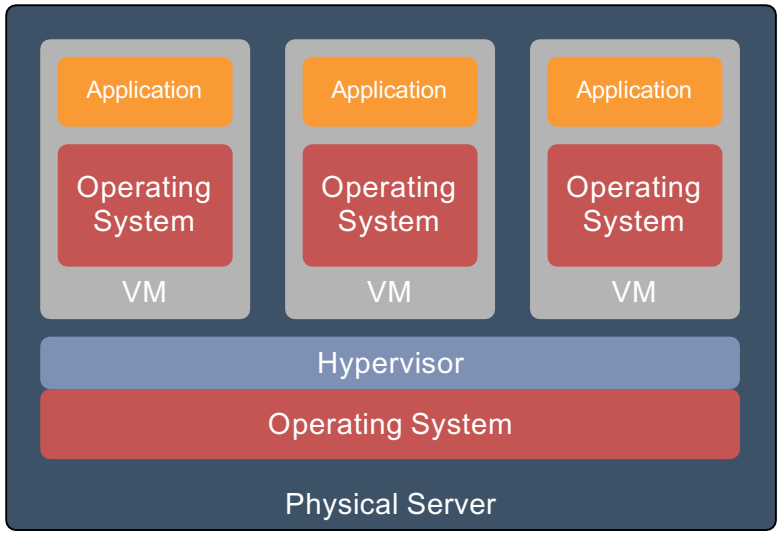


One physical server, multiple applications

Discussion

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

Virtualized Infrastructure - Advantages

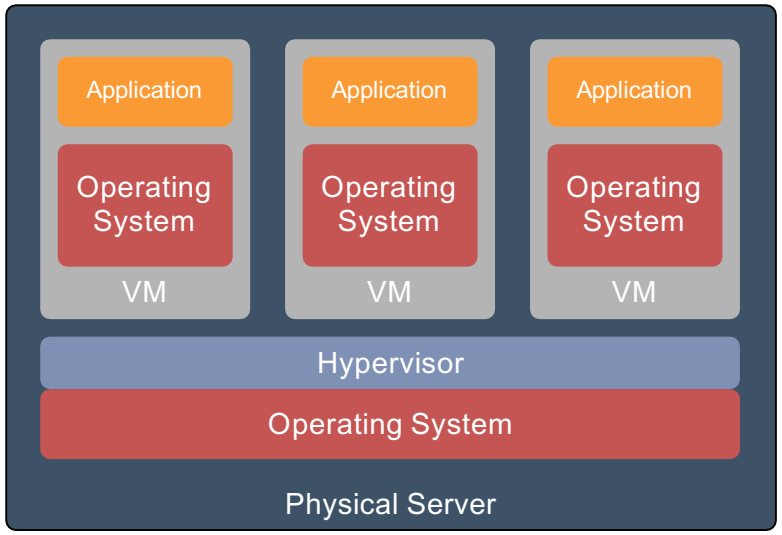


One physical server, multiple applications

Advantages

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

Virtualized Infrastructure - Limitations



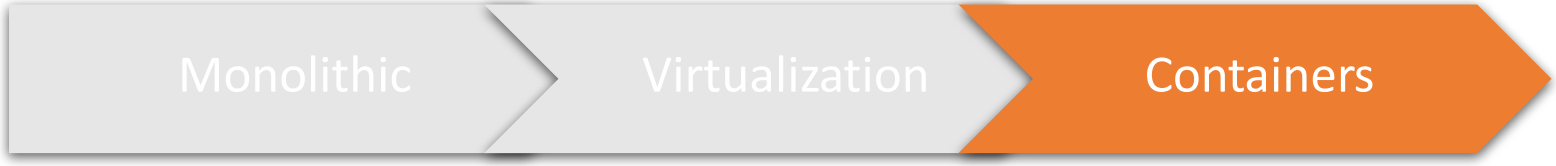
One physical server, multiple applications

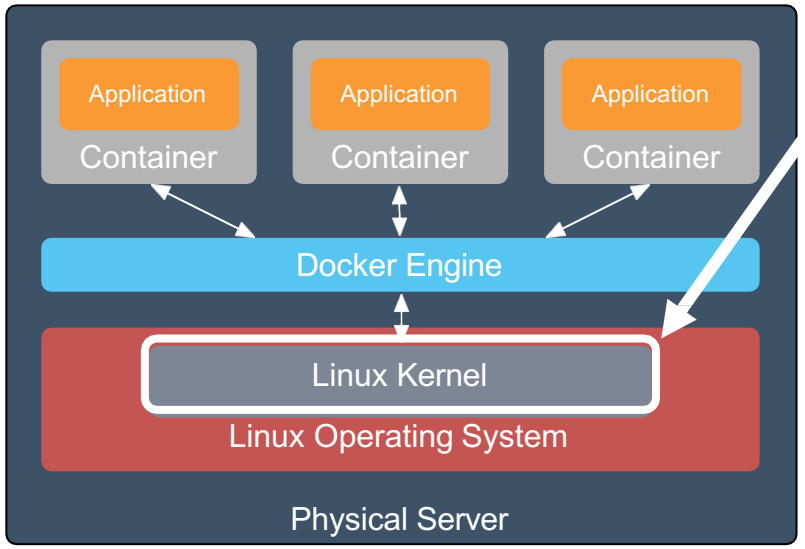
Limitations

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

Containers

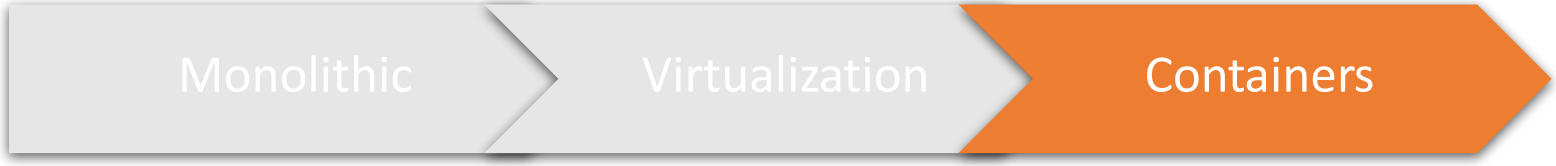
Containers

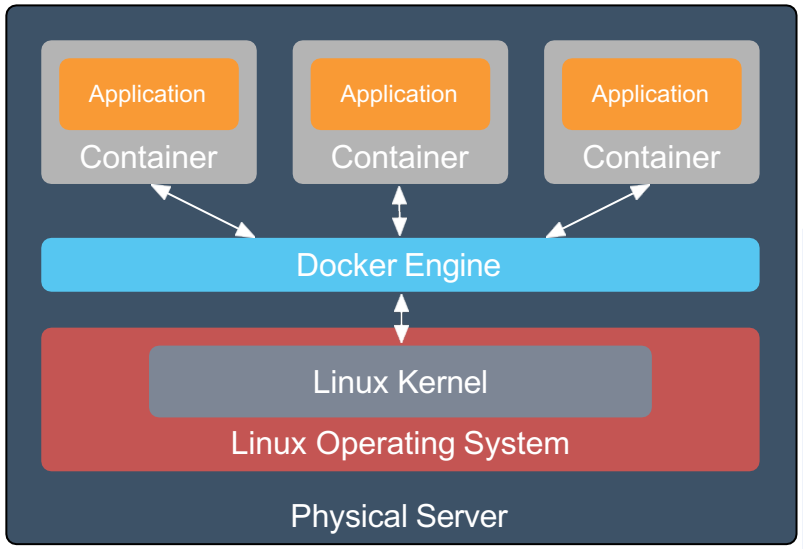




Shared kernel on the host to run multiple guest applications

Containers - Advantages



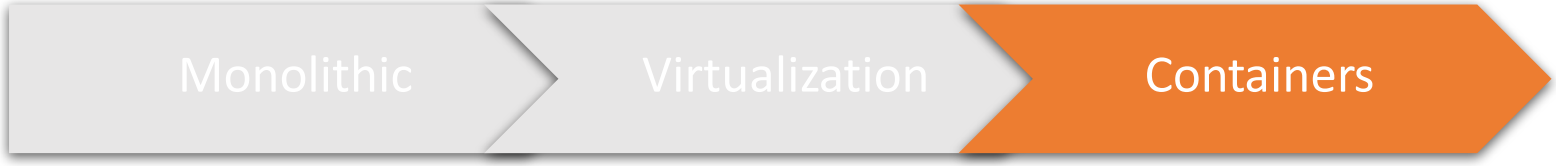


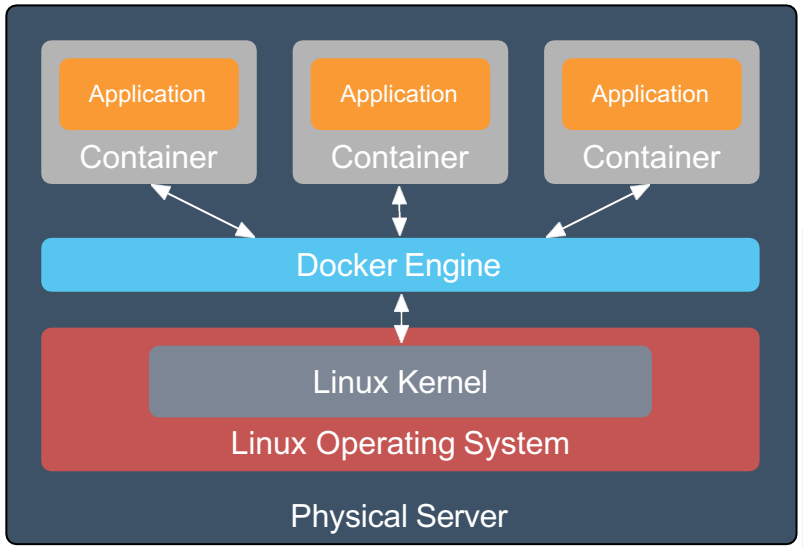
Shared kernel on the host to run multiple guest applications

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

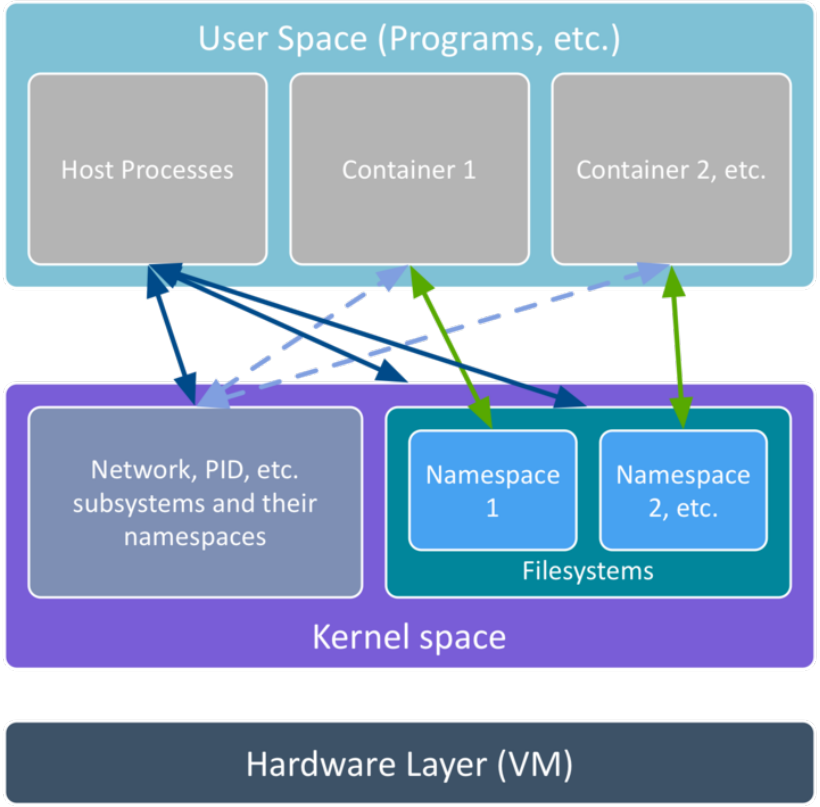
Uses the kernel on the host operating system to run multiple guest instances

- 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)

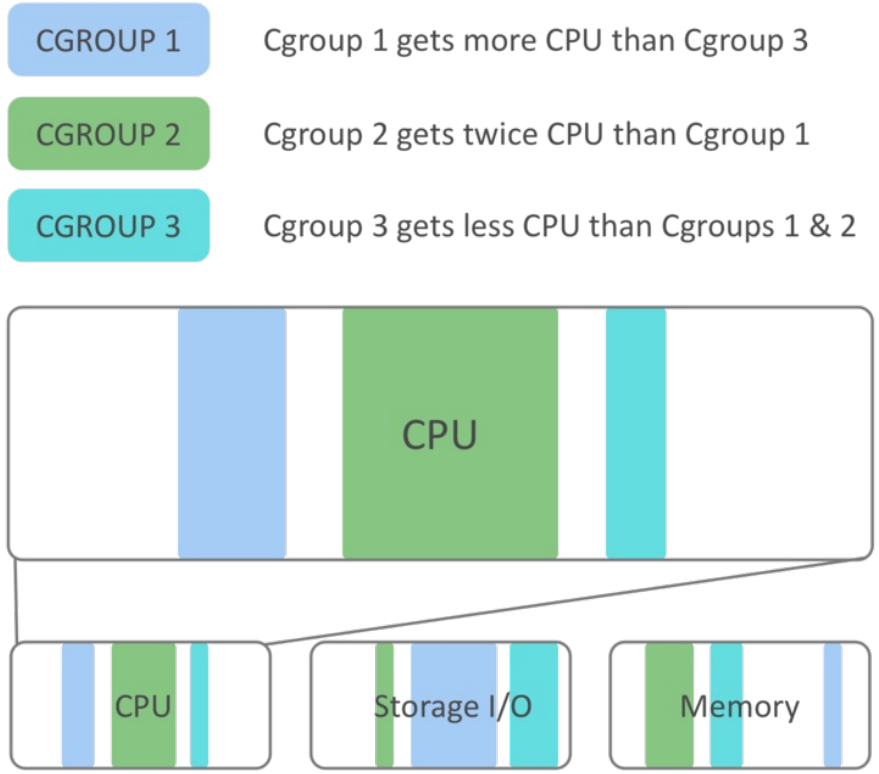
- Namespace wrap a global system resource in an abstraction layer
- Processes running in that namespace think they have their own, isolated resource
- Isolation includes:
  - Network stack
  - Process space
  - Filesystem mount points
  - etc.



Isolation with Control group (Cgroups)

Cgroups - Limits what a container can use

- Resource metering and limiting
  - CPU
  - MEM
  - Block/IO
  - 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

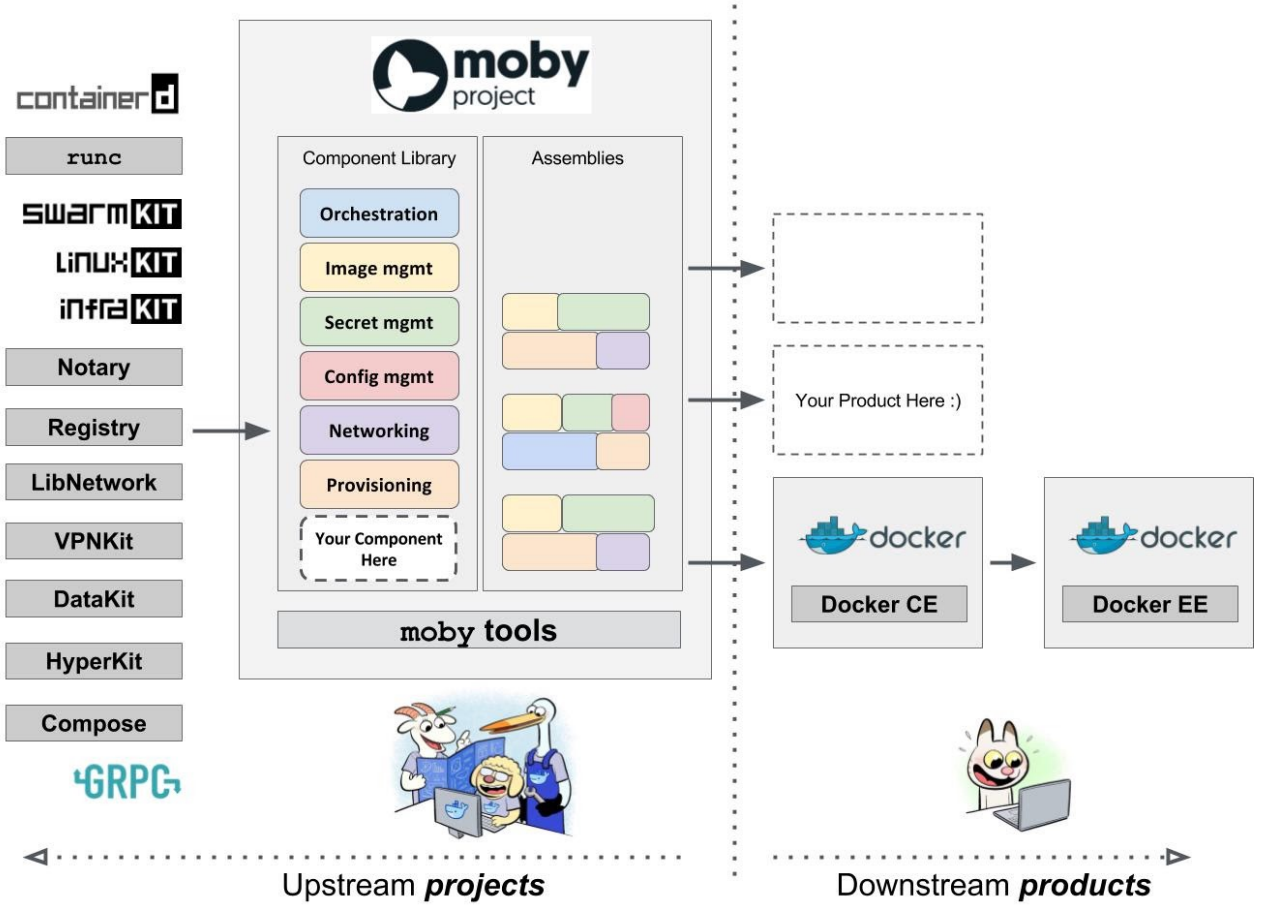
- Same container that developers run is the container that runs in test lab and production – includes all dependencies
- Well formed API allows for automated building and testing of new containers

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

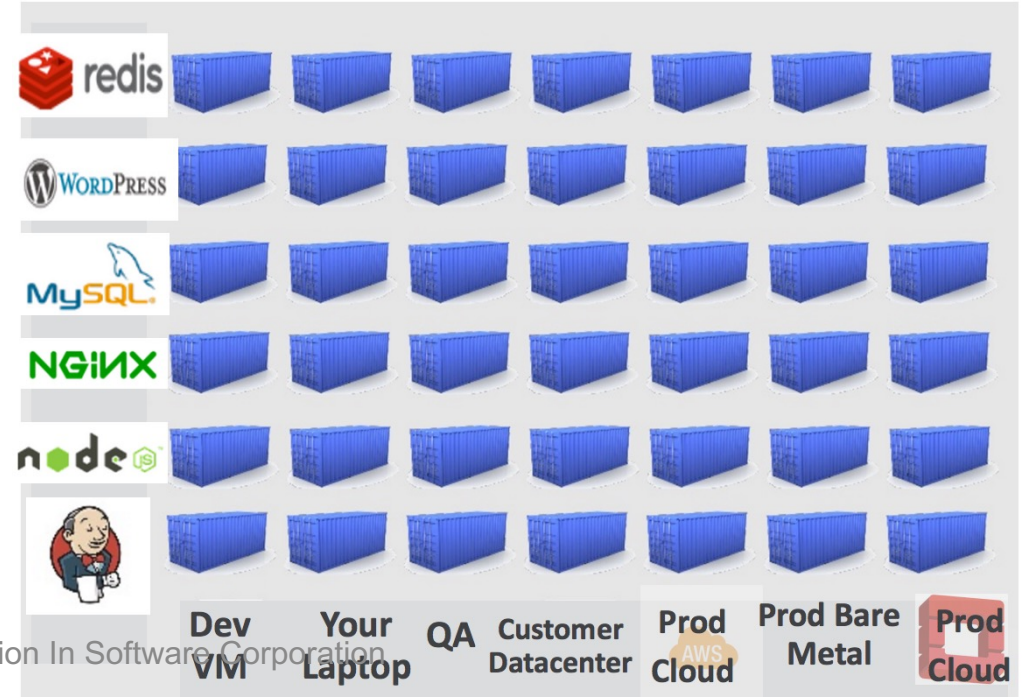
Docker Overview

What is Docker?



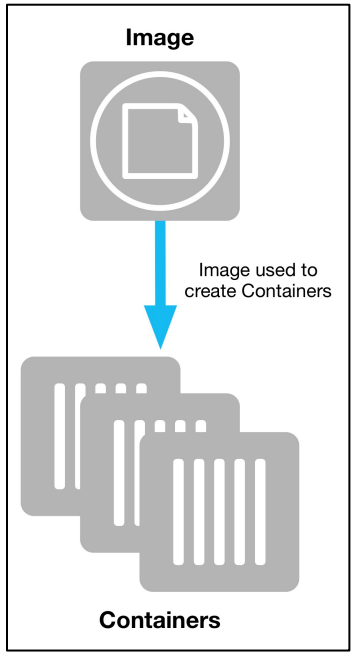
What is Docker?

Docker allows you to package an application with all of its dependencies into a standardized unit for software development.



Terminology

Image



Read only template used to create containers
Built by you or other Docker users
Stored in Docker Hub, Docker Trusted Registry or your own Registry

Terminology

Image

Read only template used to create containers
Built by you or other Docker users
Stored in Docker Hub, Docker Trusted Registry or your own Registry

Container

- Isolated application platform
- Contains everything needed to run your application
- Based on one or more images

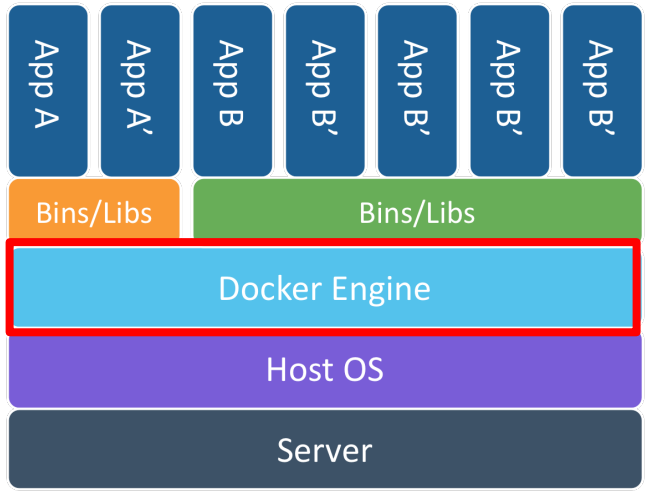
The Docker Platform Components

Docker Engine

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

> *Lightweight runtime program to **build, ship, and run Docker containers***

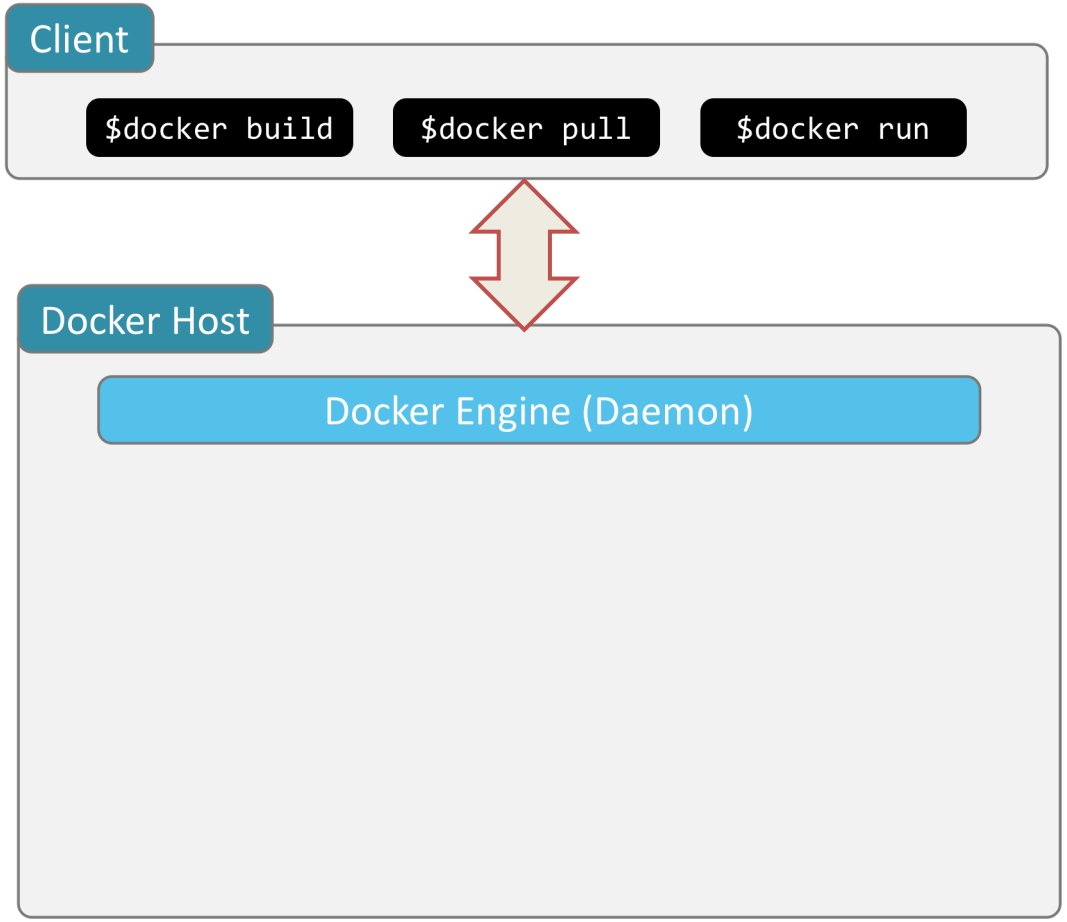
- Also known as **Docker Daemon**
- Uses Linux Kernel namespaces and control groups
  - **Linux Kernel (>= 3.10)**
- Namespaces provide an isolated workspace



Docker Engine

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

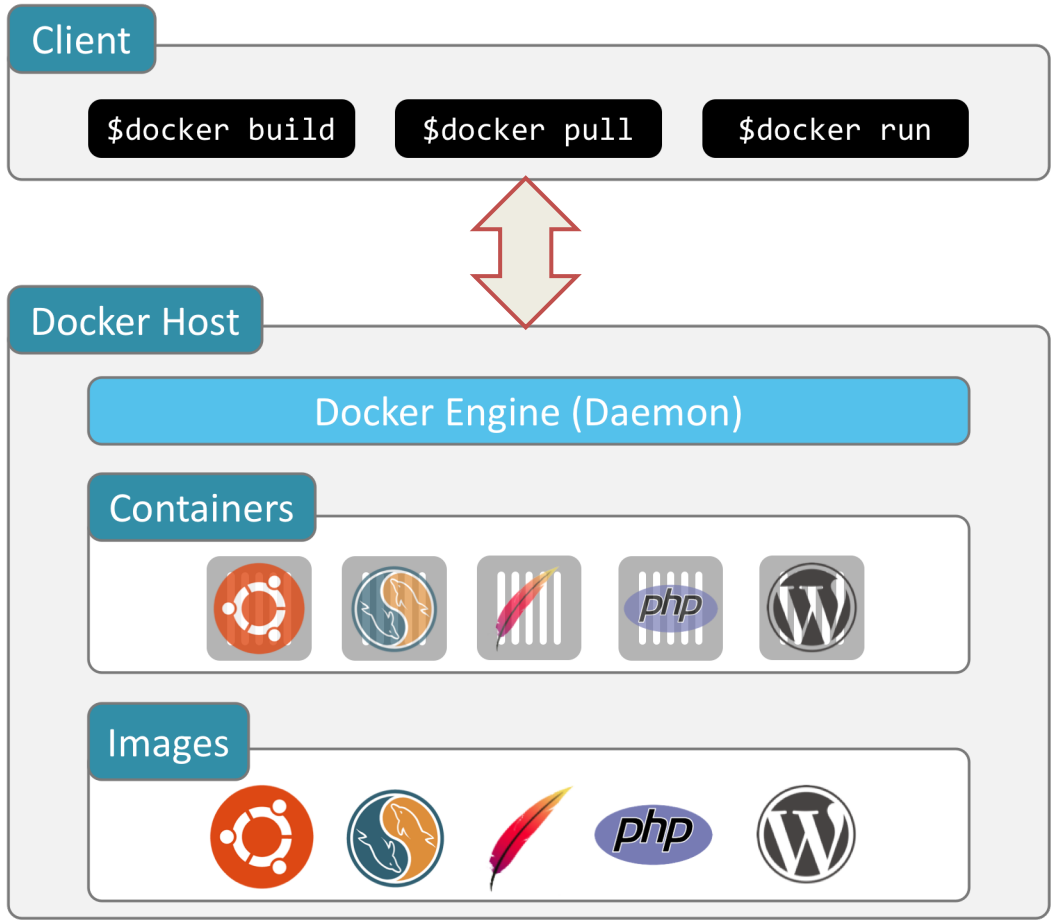
- The Docker Client is the `docker` binary
  - Primary interface to the Docker Host
- Accepts commands and communicates with the Docker Engine (Daemon)



Docker Engine

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

- Lives on a Docker host
- Creates and manages containers on the host



Docker Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

Image Storage & Retrieval System

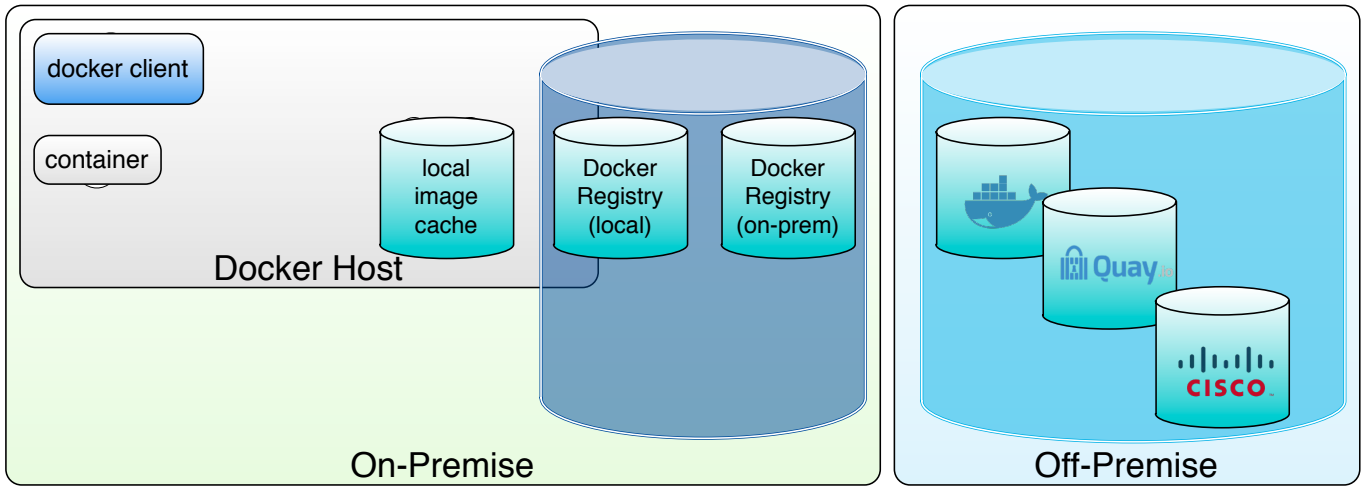
- Docker Engine **Pushes** Images to a Registry
- Version Control
- Docker Engine **Pulls** Images to Run

Docker Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

Types of Docker Registries

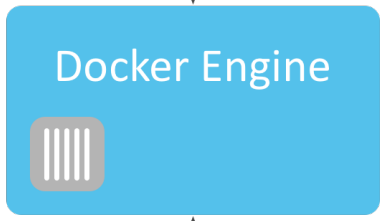
- Local Docker Registry
  (On Docker Host)
- Remote Docker Registry
  (On-Premise/Off-Premise)
- Docker Hub
  (Off-Premise)



Docker Engine and Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

Manage & Monitor



Push/Pull

Docker Engine:

- **Pushes** Images to a Registry
- **Pulls** Images to Run

Docker Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

The registry and engine both present APIs

- All of Docker's functionality will utilize these APIs

- RESTFUL API
- Commands presented with Docker's CLI tools can also be used with curl and other tools

Docker Compose

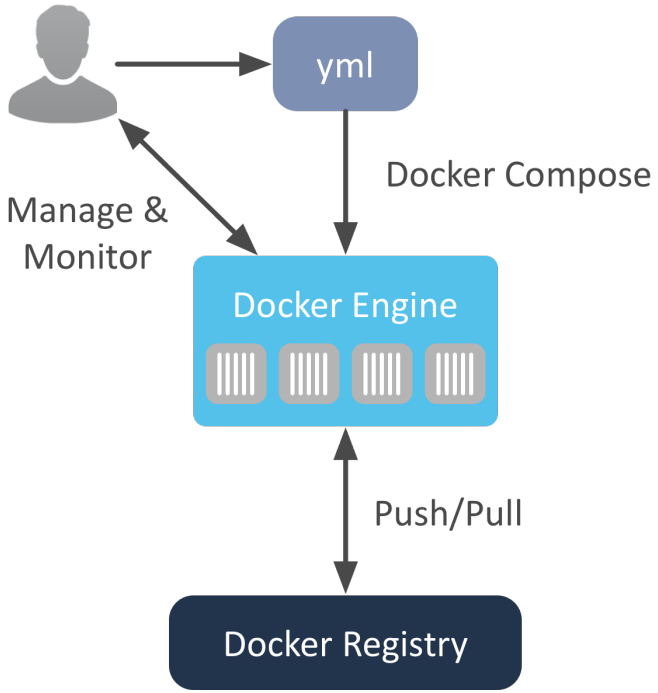
Docker Engine | *Docker Registry* | **Docker Compose** | Docker Swarm

Tool to create and manage multi-container applications

- Applications defined in a single file:

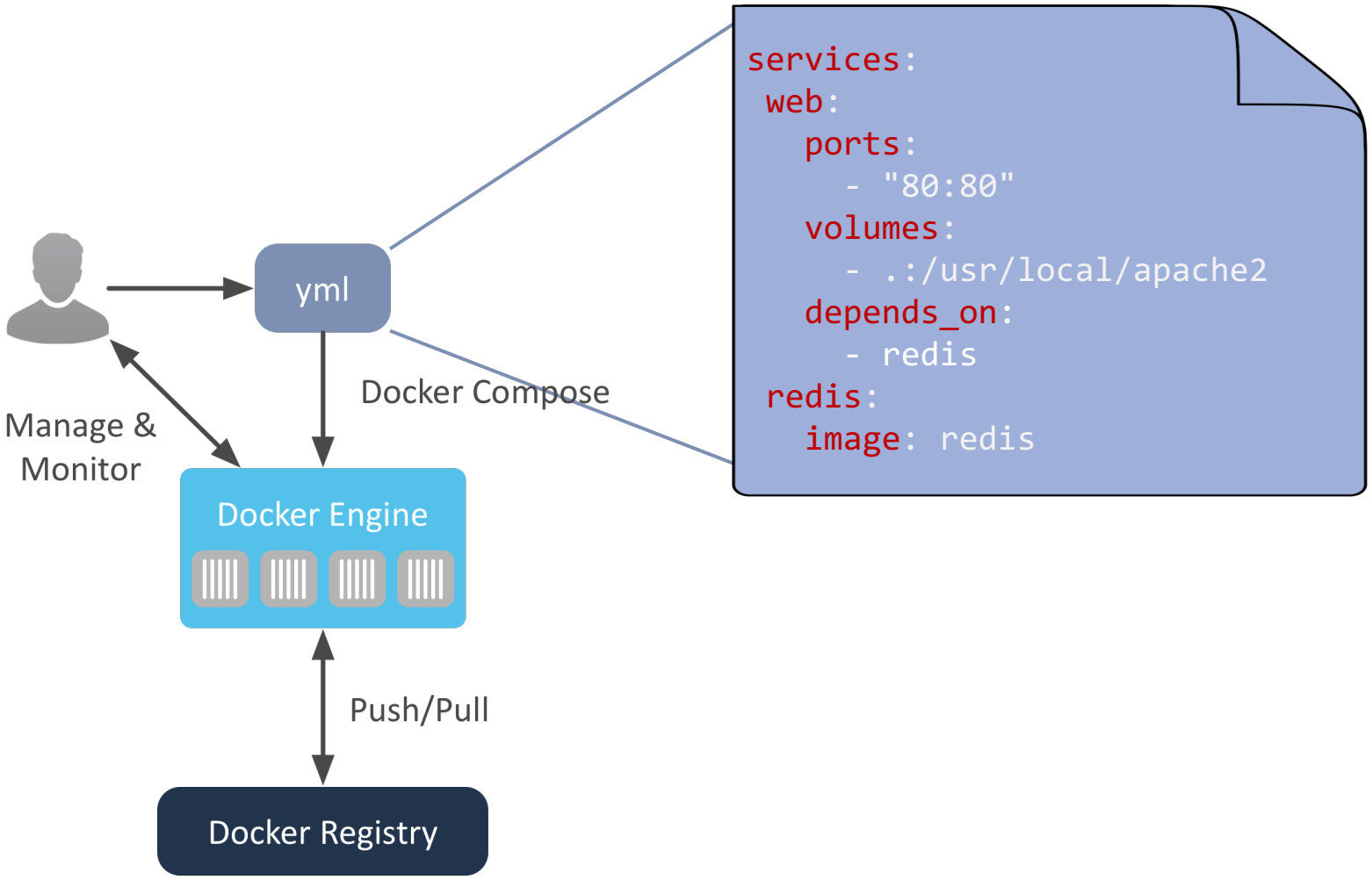
  docker-compose.yml

- Transforms applications into individual containers that are linked together
- Compose will start all containers in a single command

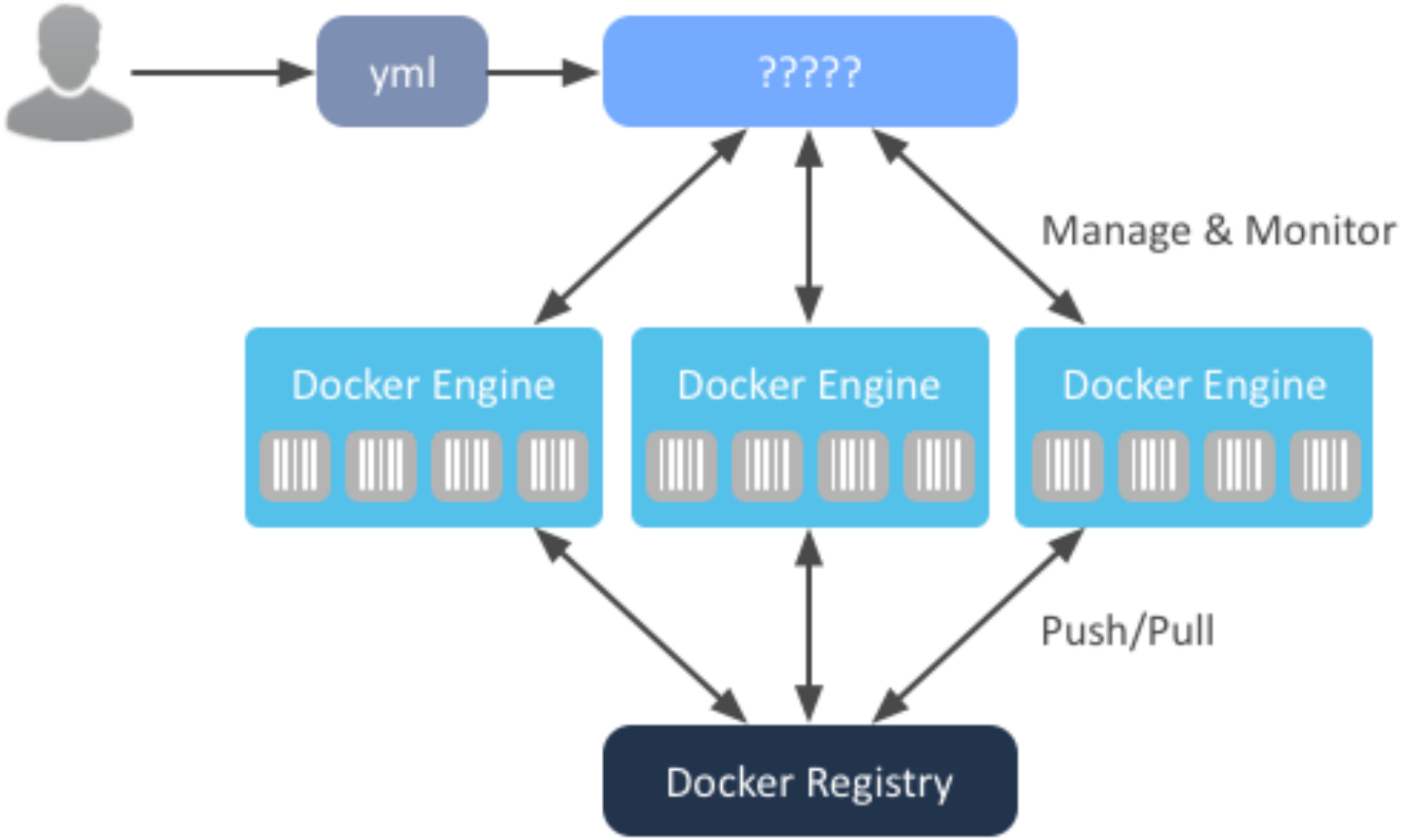


Docker Compose

Docker Engine | *Docker Registry* | **Docker Compose** | Docker Swarm



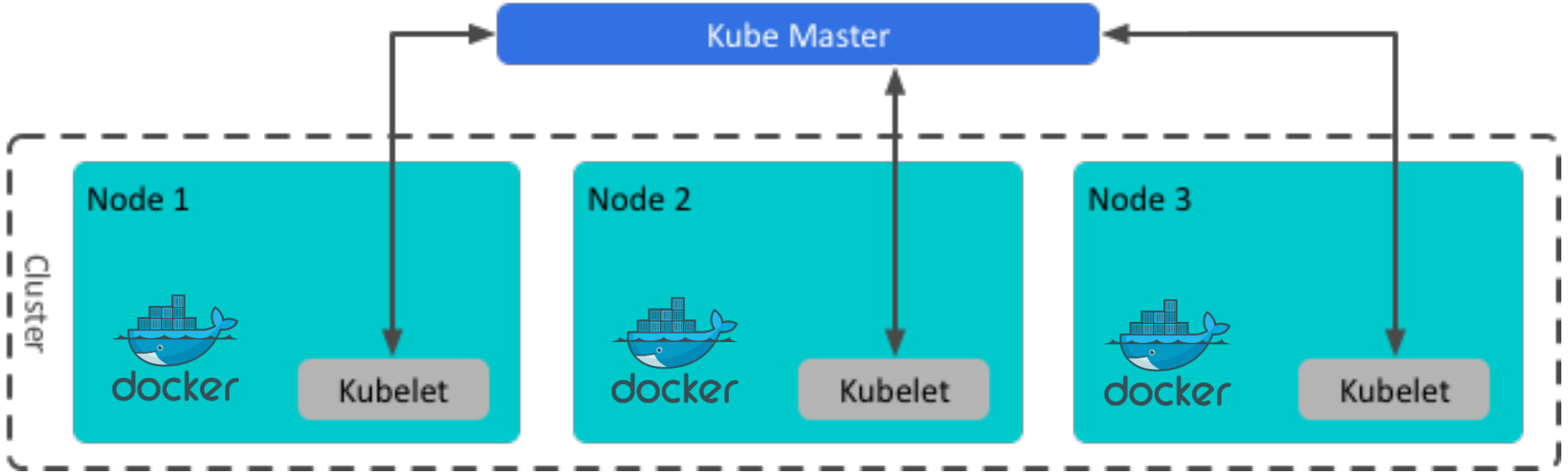
End Goal: Manage Cluster as Single Pool of Resources



Kubernetes – Containers at scale

Kubernetes: Containers at Scale

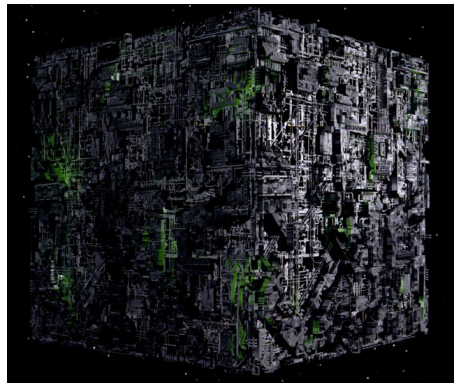
Kubernetes provides the infrastructure for container-centric deployment and operation of applications



- 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

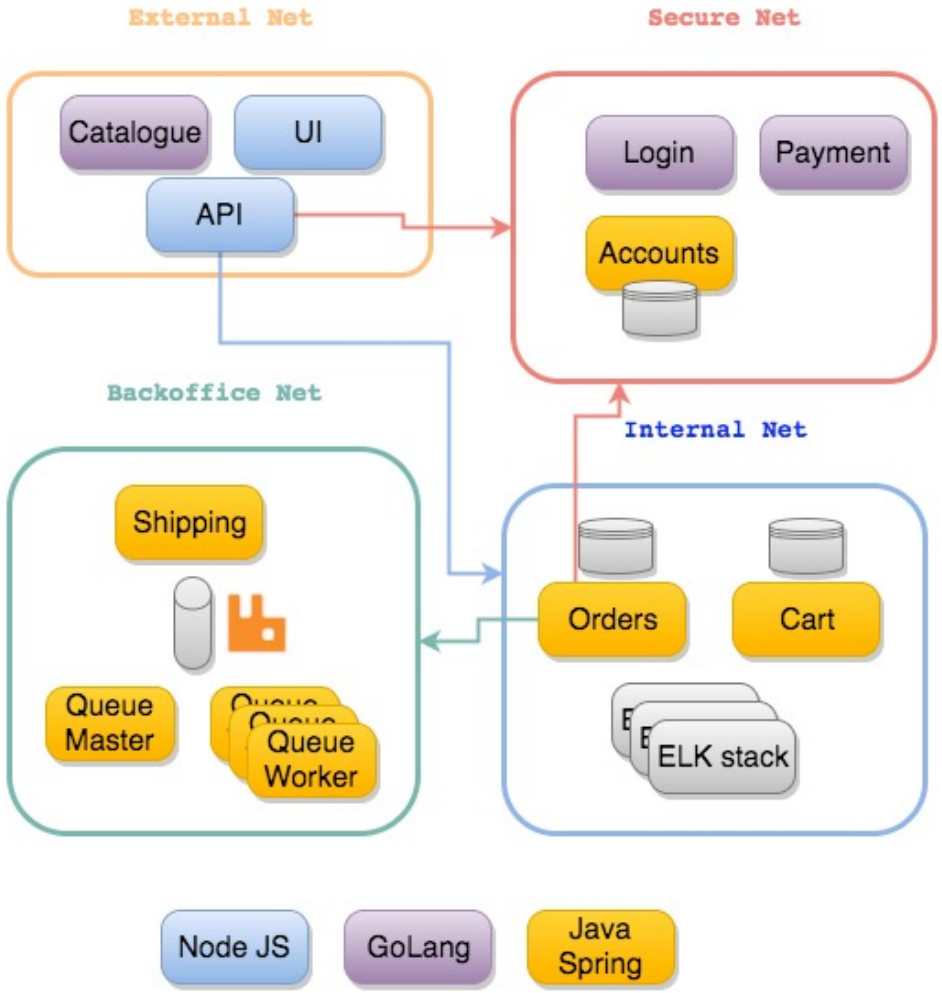
(the friendly Borg)

Kubernetes Use Cases

Kubernetes:Cloud-Native Application Deployment

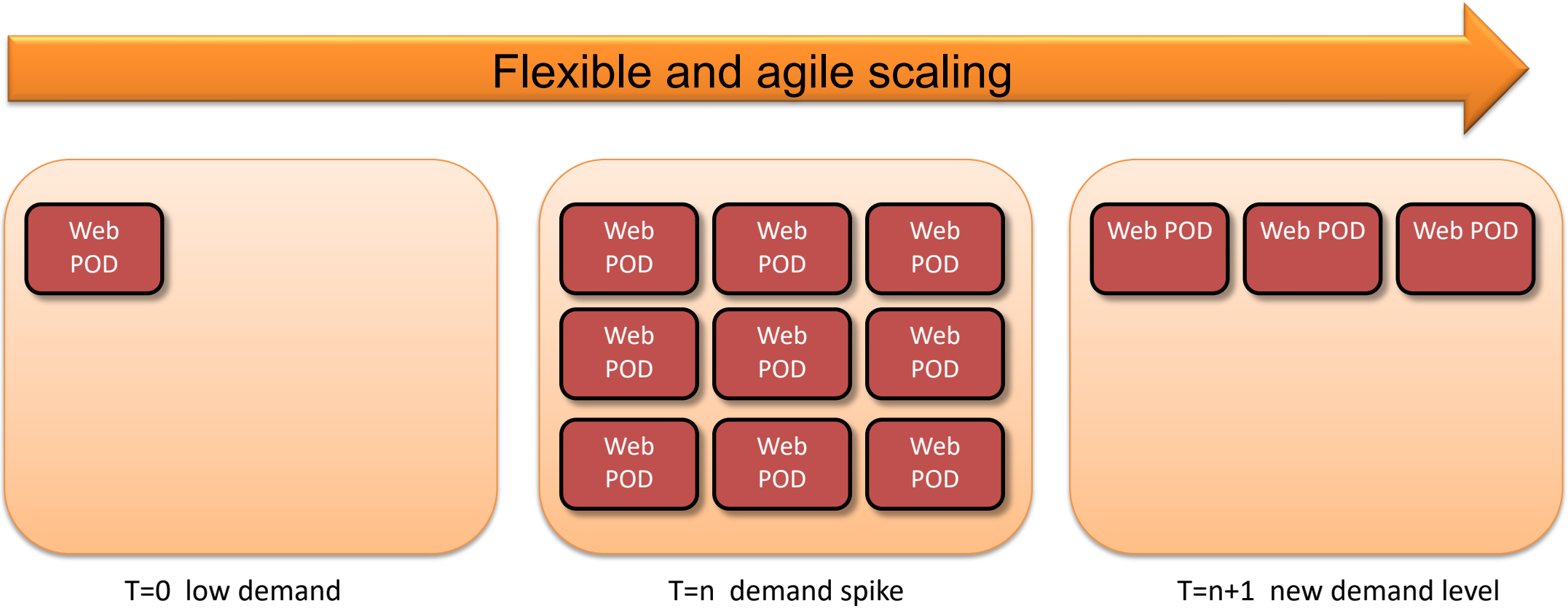
Cloud-native applications, aka microservice-based or 12-factor apps

- 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

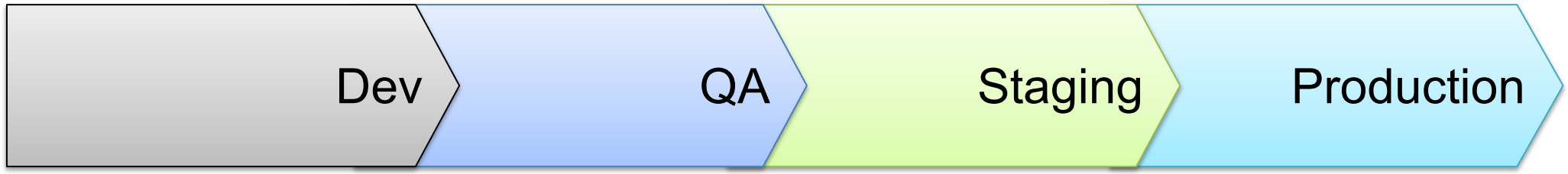
Kubernetes supports manual and automated scaling of application services based on demand for resources



Kubernetes: CI/CD Pipelines

Managing execution environments in a CI/CD pipeline

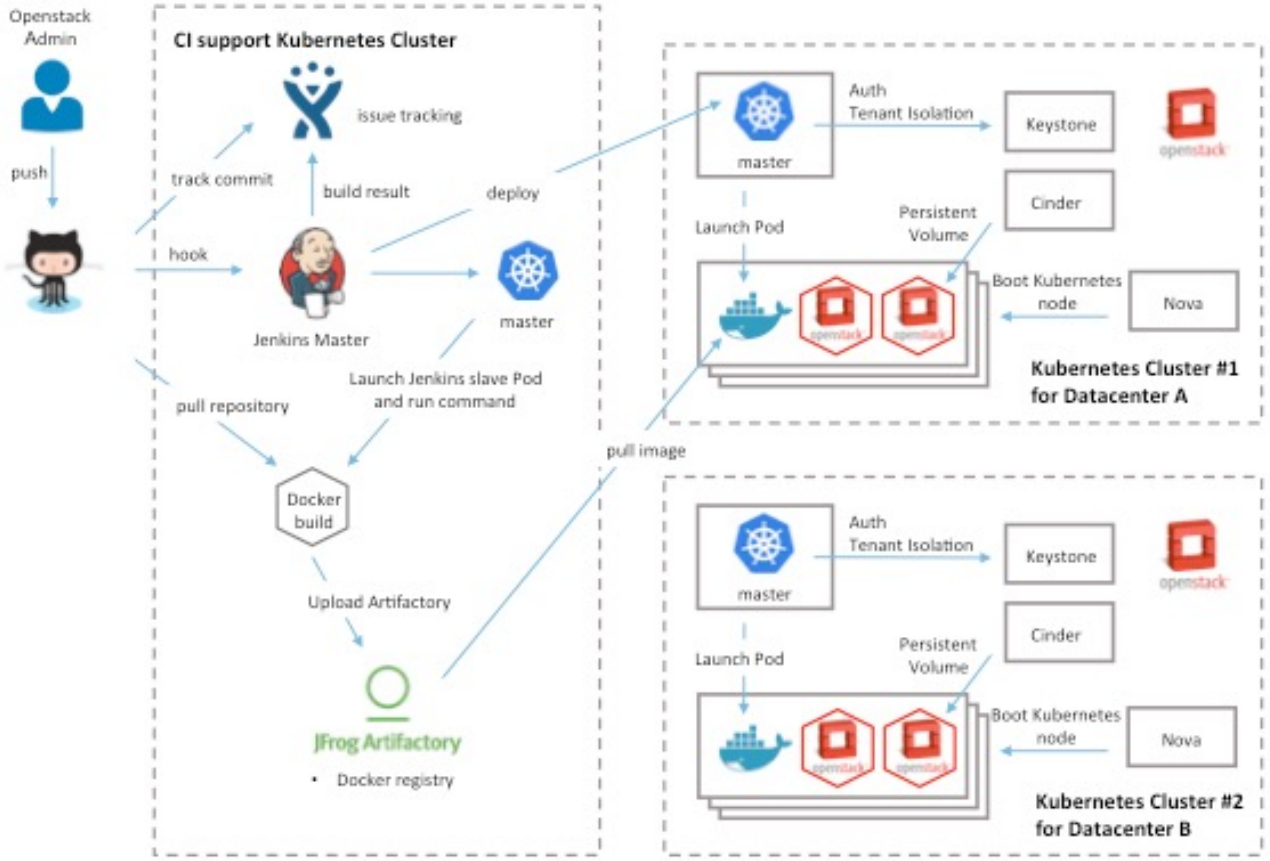
- Automated systems can push to logical environments in the same cluster, or to different clusters, using the same control plane API
- Kubernetes labels and annotations allow users to organize resources into separate environments without changing code or functionality
- Label selectors target specific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



Example: Kubernetes CI in Production at Scale

YAHOO! JAPAN

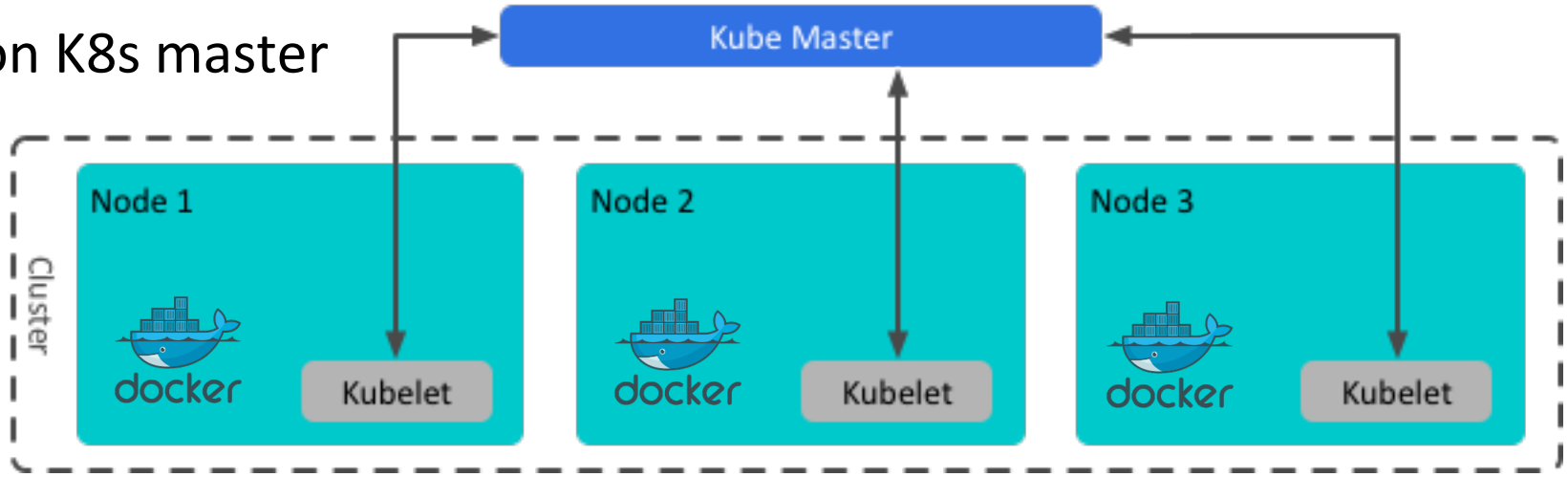
- Built automated system to build and deploy containerized applications on Kubernetes
- Kubernetes clusters built from VM's in private OpenStack environments
  - 1,000 nodes
  - 42,000 Containers
  - ~2,500 Applications



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: Leader

Main Components:

Leader Node:

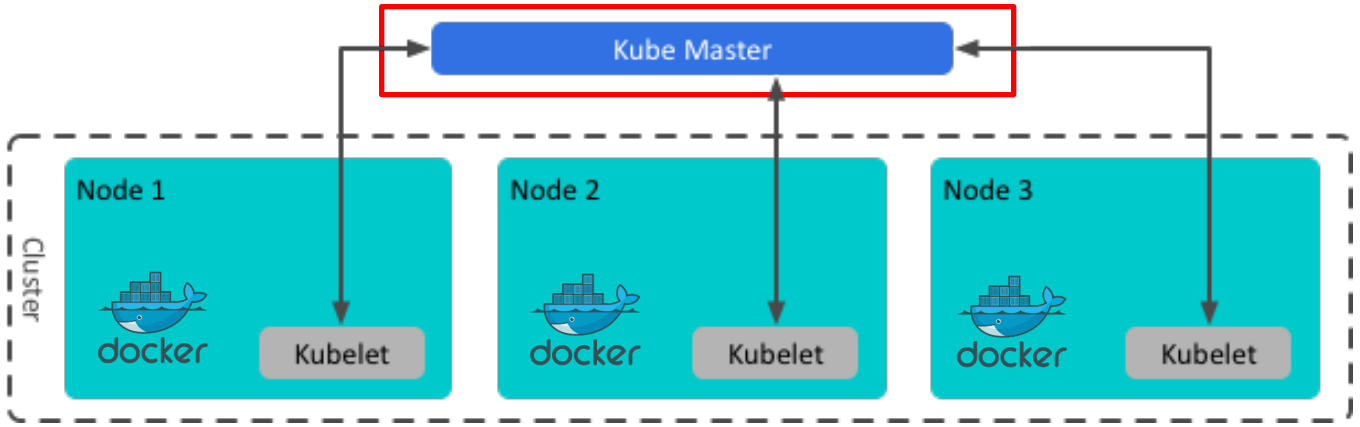
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

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:

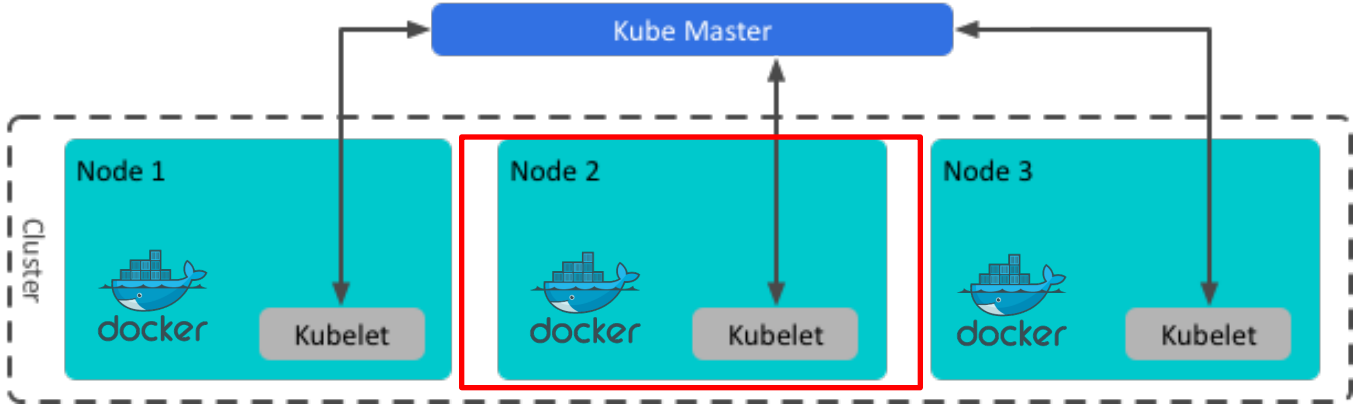
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

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:

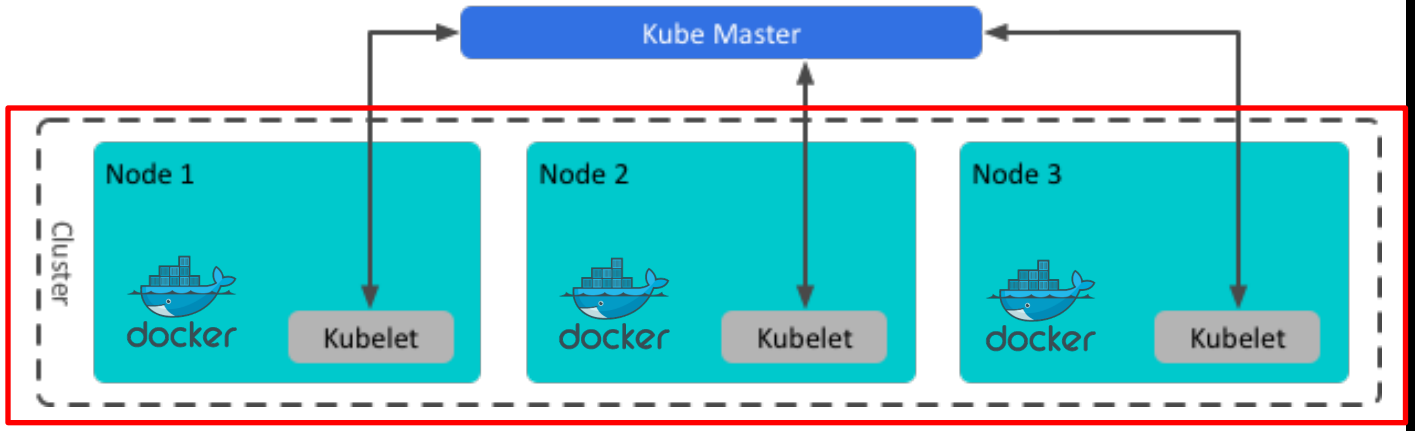
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

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



Azure Kubernetes Service (AKS)

Bare Metal K8s

Infrastructure: Deployed directly on physical servers, providing full control over hardware and software environments.

Customization: Complete control over the configuration and management of the Kubernetes cluster, allowing for maximum flexibility.

Scalability: Scaling requires manual effort in adding/removing servers and managing resources.

Maintenance: High operational overhead; administrators are responsible for all updates, patching, and monitoring.

Use Cases: Suitable for organizations needing complete control, specific hardware optimizations, or running highly sensitive workloads on-premises.

Advantages of AKS over Bare Metal

Infrastructure: Fully managed by Microsoft Azure, with simplified Kubernetes cluster provisioning and management.

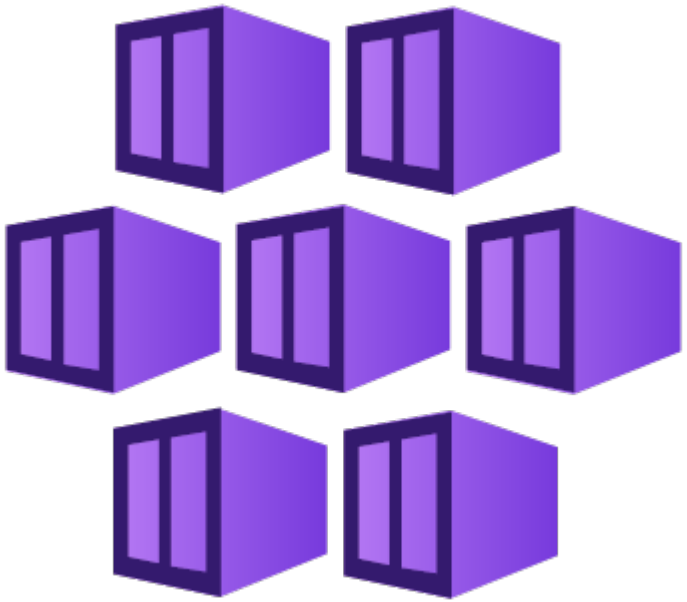
Managed Operations: Automated updates, scaling, patching, and monitoring by Azure, reducing operational overhead.

Scalability: Highly scalable with automatic node scaling based on demand; integrates easily with Azure services.

Cost: Pay only for the VMs and resources you use, with reduced costs compared to managing physical servers.

Use Cases: Ideal for cloud-native applications, reducing DevOps workload, and leveraging Azure's global infrastructure.

Azure Kubernetes Service (AKS)



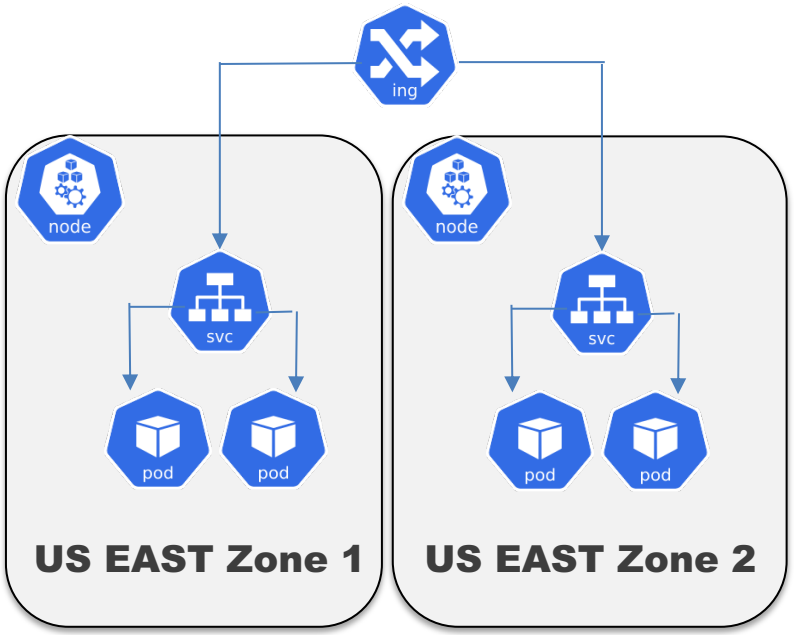
Managed Kubernetes: AKS is a fully managed Kubernetes service offered by Azure, simplifying the deployment, management, and scaling of containerized applications in the cloud.

- **Seamless Integration:** It integrates natively with other Azure services like Azure Monitor for monitoring, Azure Active Directory for role-based access control, and Azure DevOps for CI/CD pipelines.

Key Benefits:

- **Automatic Scaling:** Easily scale applications based on demand.
- **Security:** Built-in security features, including network isolation and identity integration.
- **Cost-Effective:** Pay only for the resources you use, with the ability to quickly scale up or down.
- **Use Cases:** Ideal for microservices, CI/CD workflows, and running complex applications with high availability in cloud environments.

High Availability

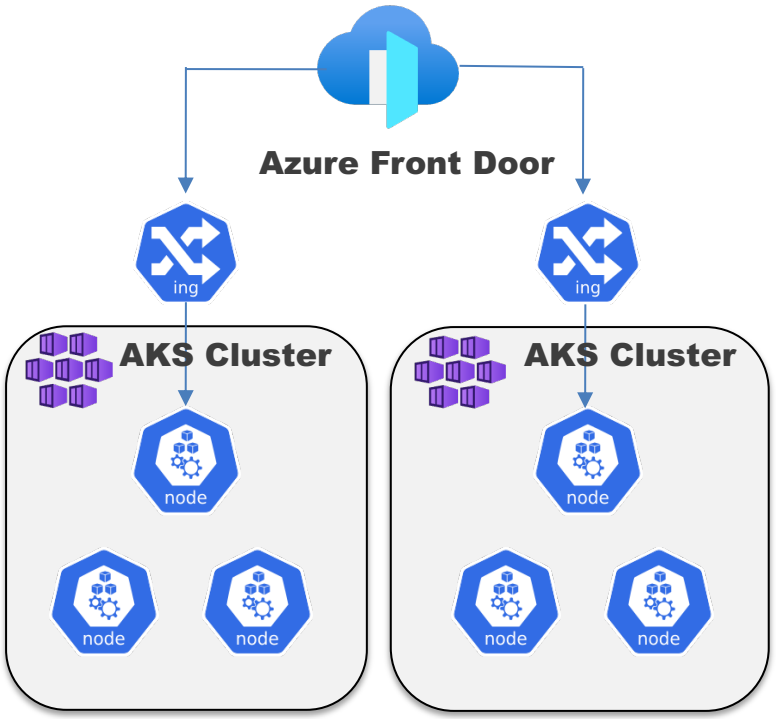


Azure Kubernetes Service (AKS) provides built-in high availability (HA) options for enterprise-grade resilience:

- **Multi-Zone Availability:** AKS supports deployment across multiple availability zones (AZs) within a region. This ensures redundancy for both control plane and worker nodes, so if one zone experiences issues, others can continue running.
- **Node Pool Redundancy:** By configuring multiple node pools, you can run workloads across various VM sizes or regions, spreading the risk and ensuring availability in the event of node failure.
- **Cluster Autoscaler:** Automatically adjusts the number of nodes in a cluster based on the workload demands, improving resource availability when traffic spikes and reducing costs when resources are idle.
- **Load Balancing:** AKS integrates with Azure Load Balancer to distribute incoming traffic across multiple replicas of services running on different nodes, ensuring balanced traffic and failover support.

These features ensure that AKS environments can maintain high availability, minimize downtime, and provide fault tolerance for mission-critical applications.

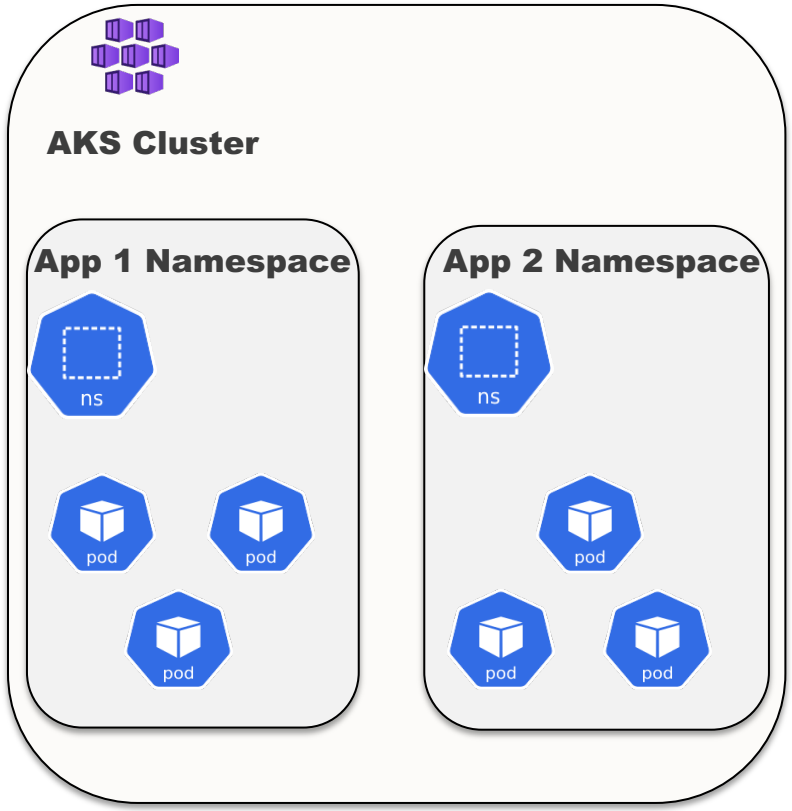
Multi-region Failover



Azure Kubernetes Service (AKS) does not natively support multi-region failover in a single cluster, but several strategies can be employed to achieve this using external services and tools:

- **Multiple AKS Clusters:** One common approach is to deploy multiple AKS clusters in different Azure regions. Each cluster operates independently, but you can distribute workloads across regions to enhance availability.
- **Global Load Balancing:** Services like Azure Traffic Manager or Azure Front Door can be used for traffic routing. These services distribute incoming requests between AKS clusters based on factors like latency, geographic proximity, or health checks, allowing for automatic failover between regions in case of an outage.
- **Active-Passive Failover:** In this setup, the primary AKS cluster in one region actively handles the traffic, while a secondary AKS cluster in a different region remains on standby. When the primary region fails, the system automatically switches to the secondary cluster.

Kubernetes Multi-Tenancy

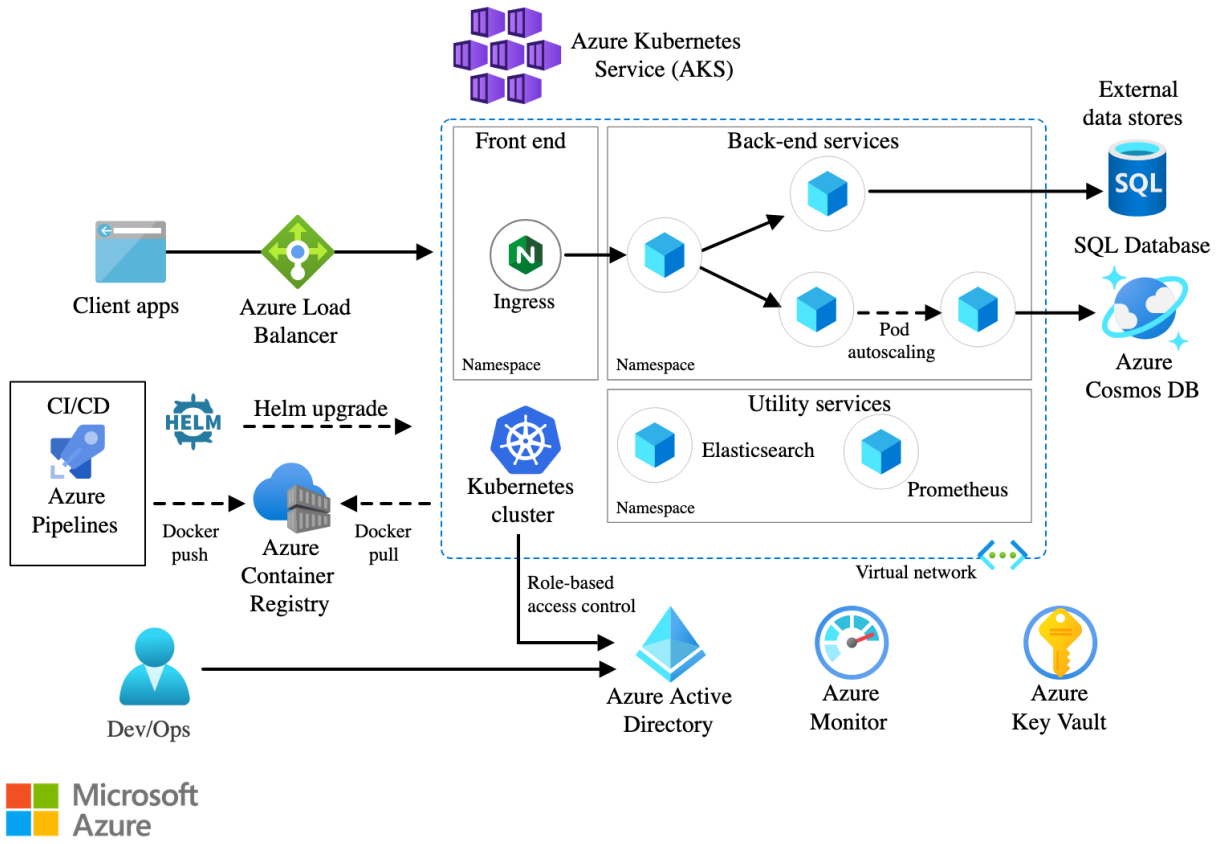


Multi-tenancy in Kubernetes allows multiple teams, applications, or customers to share the same cluster while keeping their resources, workloads, and data isolated from each other. This isolation can be achieved at different levels:

- **Namespaces:** Logical isolation, where each tenant is assigned a namespace to segregate resources.
- **RBAC (Role-Based Access Control):** Ensures different levels of access and permissions for different tenants, controlling what resources they can interact with.
- **Network Policies:** Define how pods communicate within and across namespaces, ensuring secure and controlled communication between tenants.
- **Resource Quotas:** Limits CPU, memory, and storage usage per tenant to avoid resource contention between tenants.

Multi-tenancy in Kubernetes helps organizations efficiently share infrastructure while maintaining security, performance, and compliance.

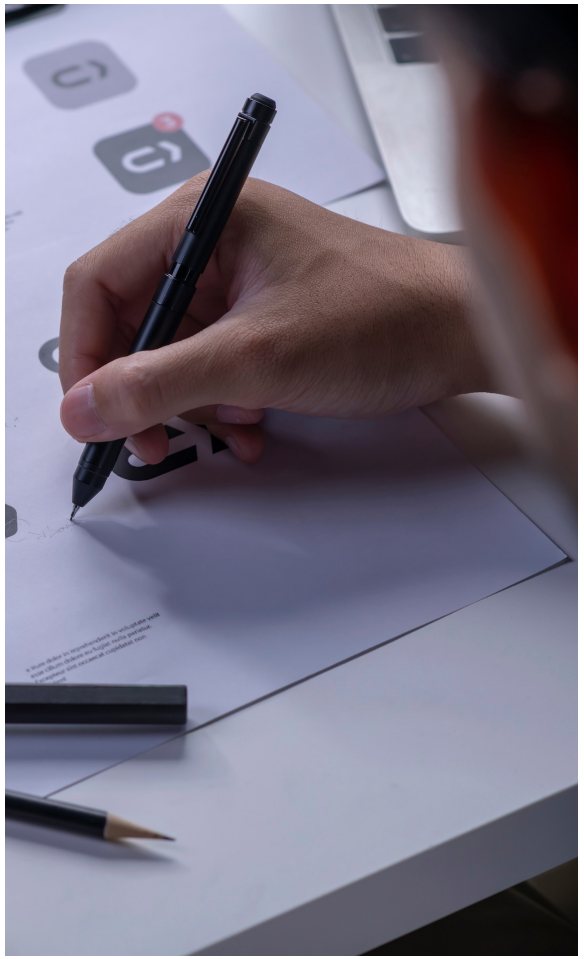
AKS in Production



AKS in production environments involves a range of strategies and tools to meet specific customer requirements. These can include high availability setups, scaling mechanisms, monitoring tools, and deployment automation.

To ensure resilience and performance, production Kubernetes often incorporates services like load balancing, persistent storage solutions, and network security. Depending on the workload, organizations might use approaches like multi-cluster management, service meshes, and CI/CD pipelines, all tailored to meet operational and customer-driven demands.

AKS Creation Options

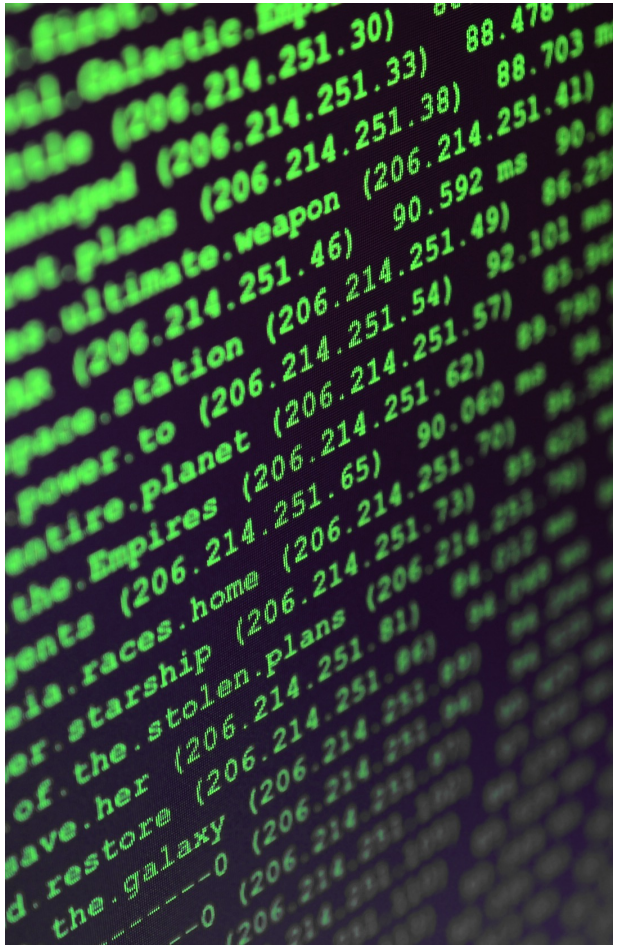


For users who prefer a visual approach, the Azure Portal provides a straightforward, manual interface that is useful for smaller applications or proof-of-concept deployments.

For more automated approaches, the Azure CLI offers a command-line tool to efficiently script AKS cluster creation and management. This is especially useful for integrating into CI/CD pipelines, allowing for easier management of infrastructure. However, for larger or more complex environments, Infrastructure as Code (IaC) tools like Terraform or ARM templates are better suited. These tools enable scalable and repeatable deployments across different environments.

Additionally, the Azure REST API offers programmatic access to create and manage AKS clusters, giving flexibility for developers needing custom integration or advanced automation.
In this course, the focus will be on using the Azure CLI.

Creating an AKS cluster (az cli)



To create an Azure Kubernetes Service (AKS) cluster using the Azure CLI, you need to specify parameters such as the resource group, cluster name, and node count. This command also enables monitoring addons and generates SSH keys, making the cluster ready for deploying and managing containerized applications programmatically or via automation.

Create an AKS Cluster

```
az aks create \
  --resource-group myResourceGroup \
  --name myAKSCluster \
  --node-count 2 \
  --enable-addons monitoring \
  --generate-ssh-keys
```

- **Note:** This command can take up to 10 minutes to complete.

Authentication to the Cluster

The 'az aks get-credentials' command downloads the access credentials and configuration details needed to interact with the AKS cluster. The credentials are stored in the default kubeconfig file, typically located at `~/.kube/config` on your local machine. The identity associated with these credentials is the Azure Active Directory (AAD) identity used to authenticate with the Azure CLI. This allows `kubectl` to manage and deploy applications on the AKS cluster.

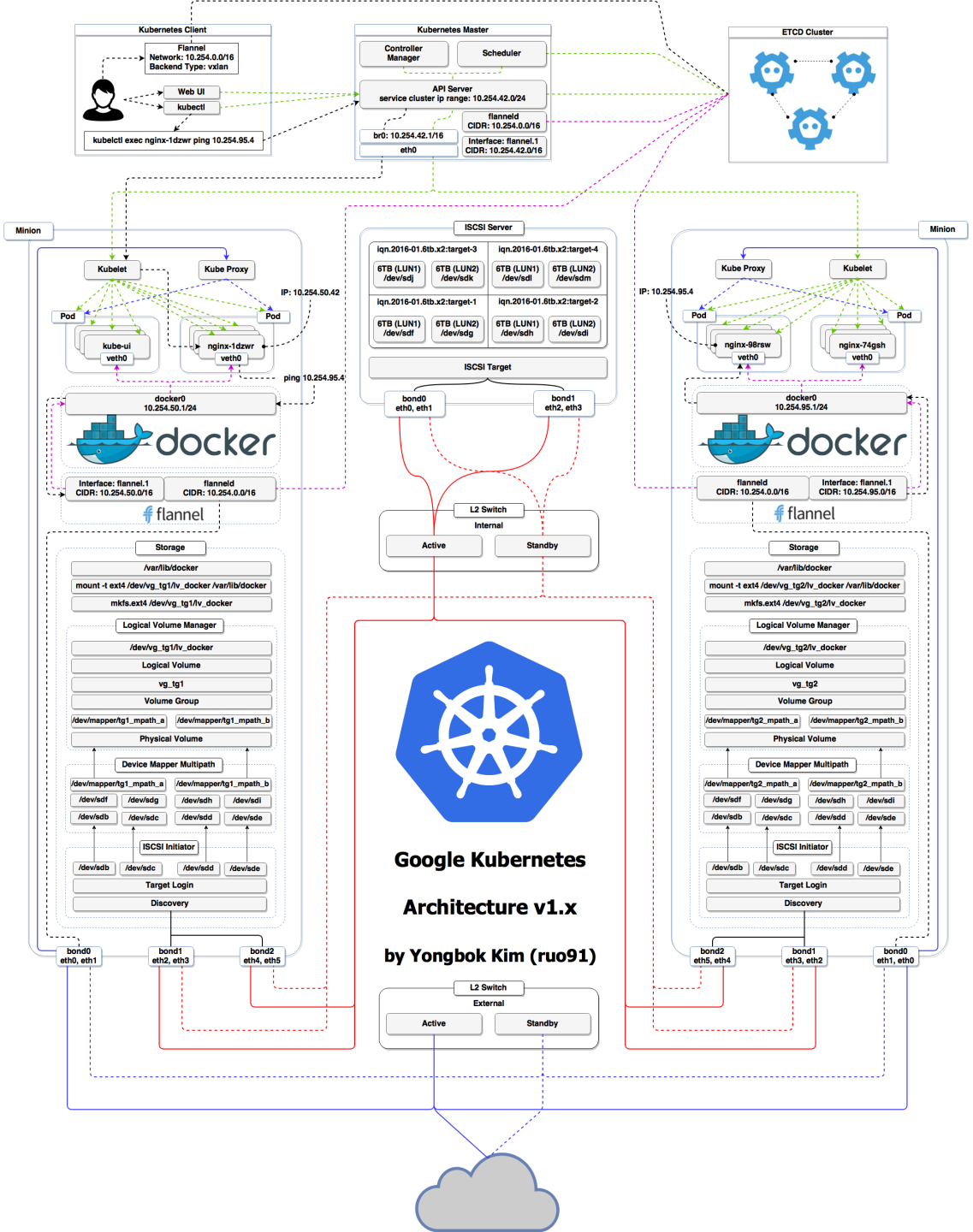
Authenticate to the AKS Cluster

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

- This command configures `kubectl` to use your credentials for the AKS cluster.

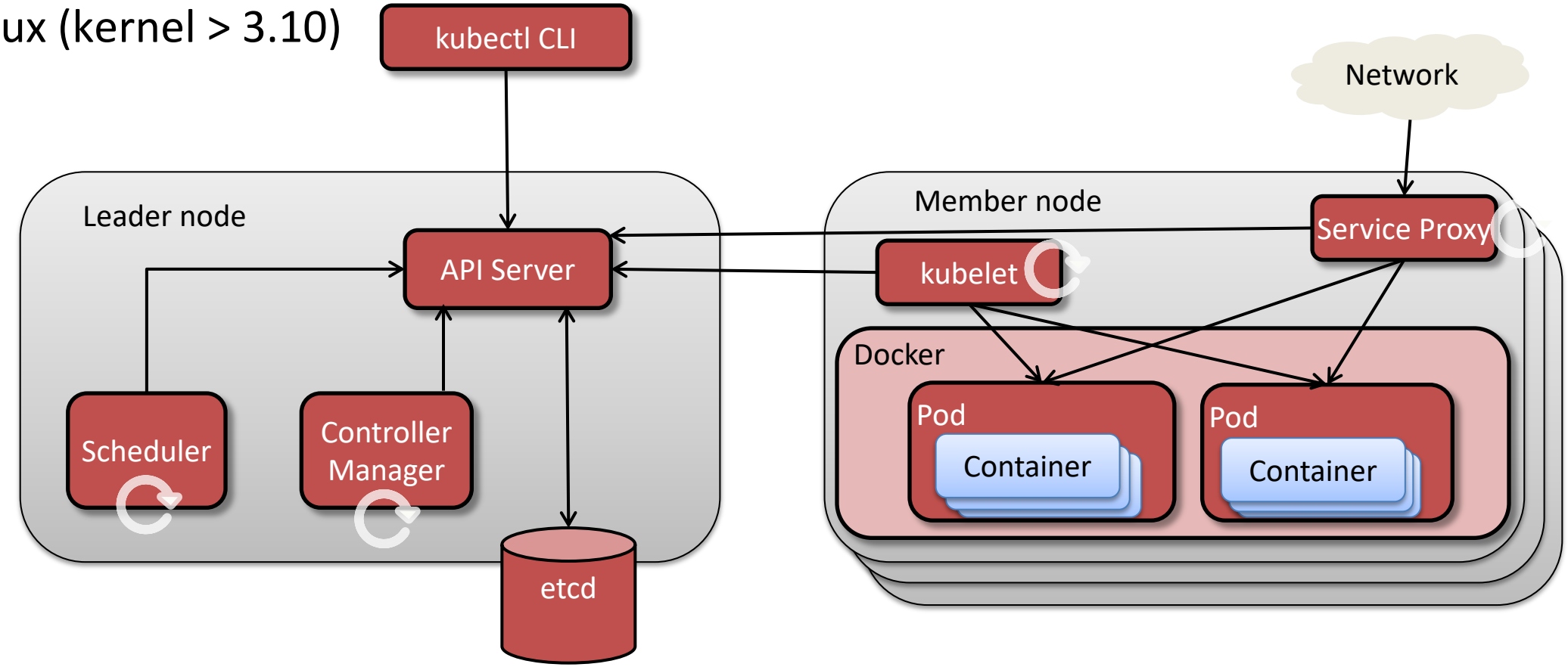
Create cluster

Kubernetes Architecture

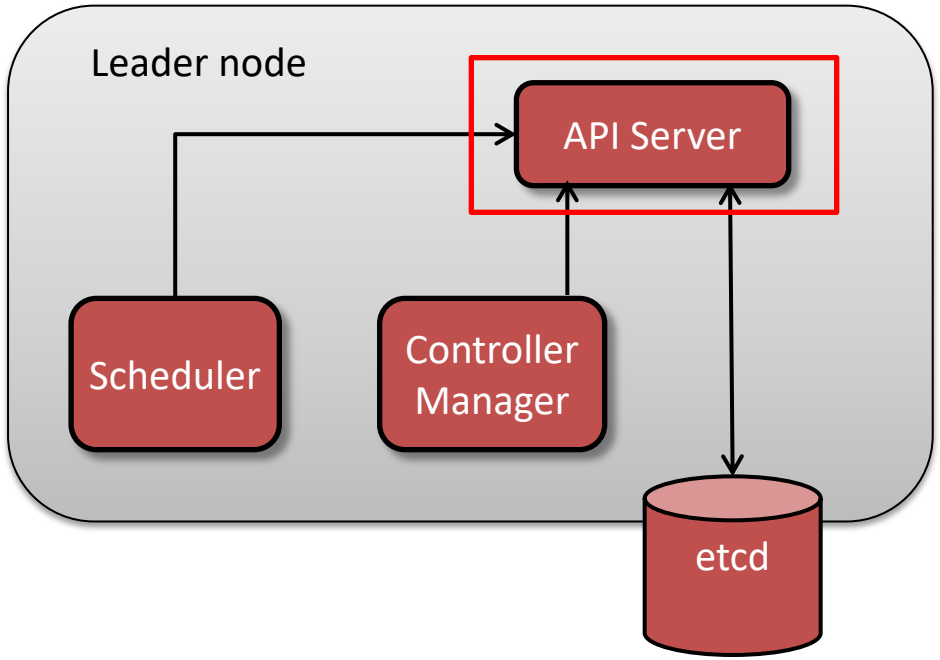


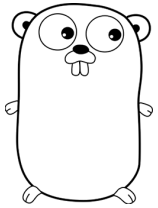
Kubernetes Cluster Architecture

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



Kubernetes Leader Node 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

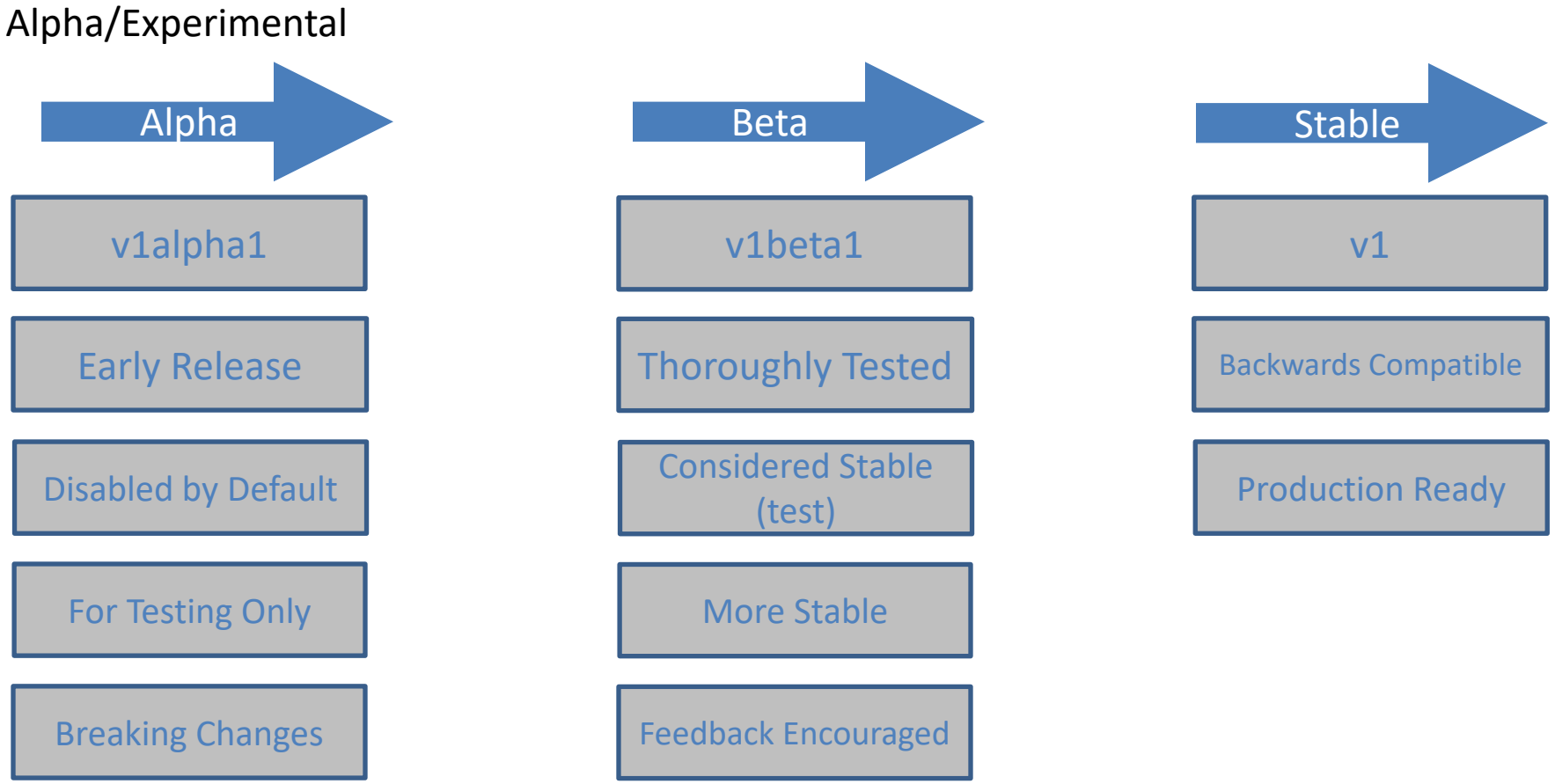
Kubernetes Resource Properties

- 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

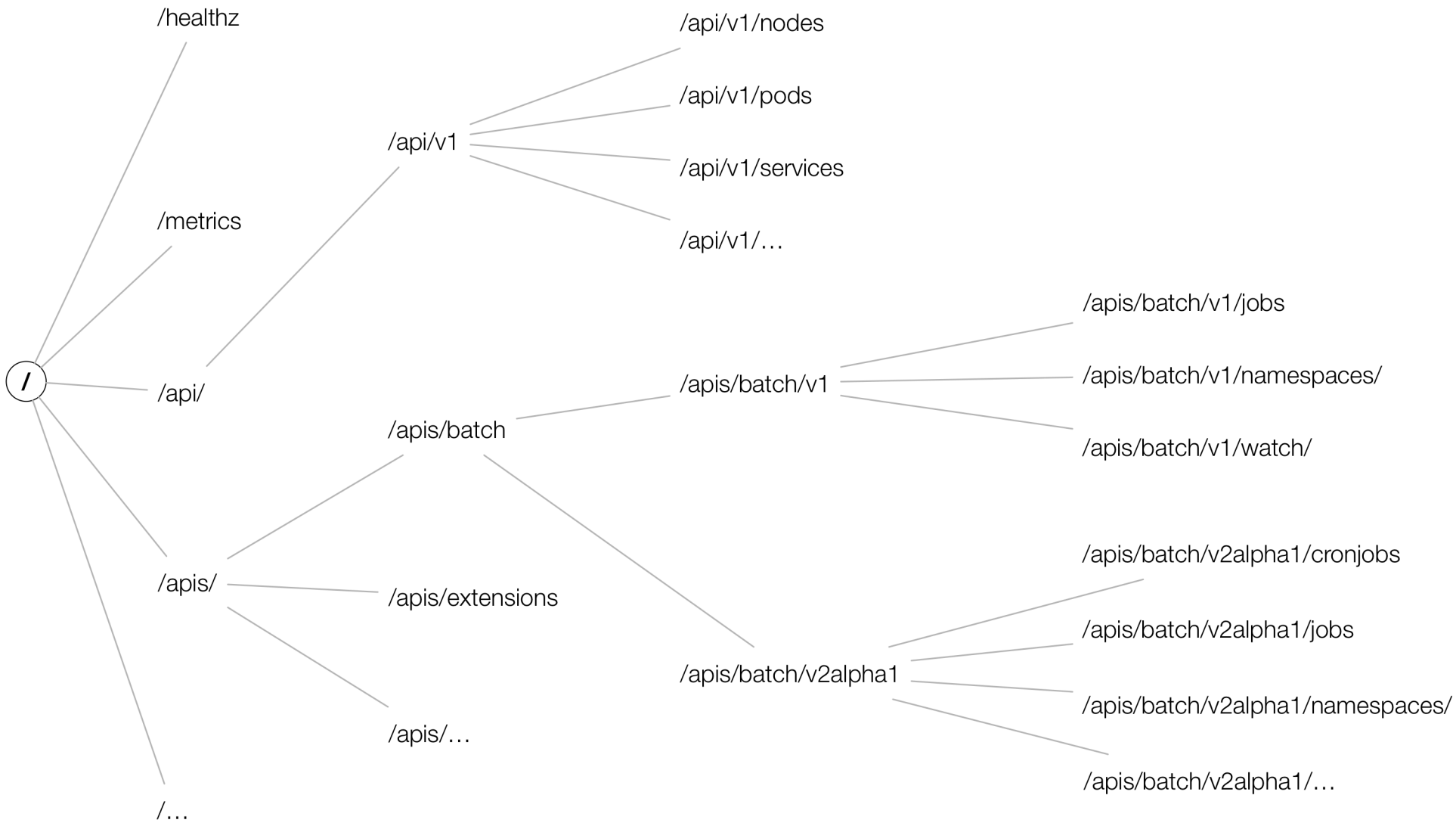
```yaml
apiVersion: v1       # schema version
kind: Pod            # type of object
metadata:
  name: nginx        # object name
  labels:            # user-defined labels
    app: website
    tier: frontend
spec:                # object spec values
    containers:
    - image: nginx:1.7.9
      name: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

simple_pod.yaml

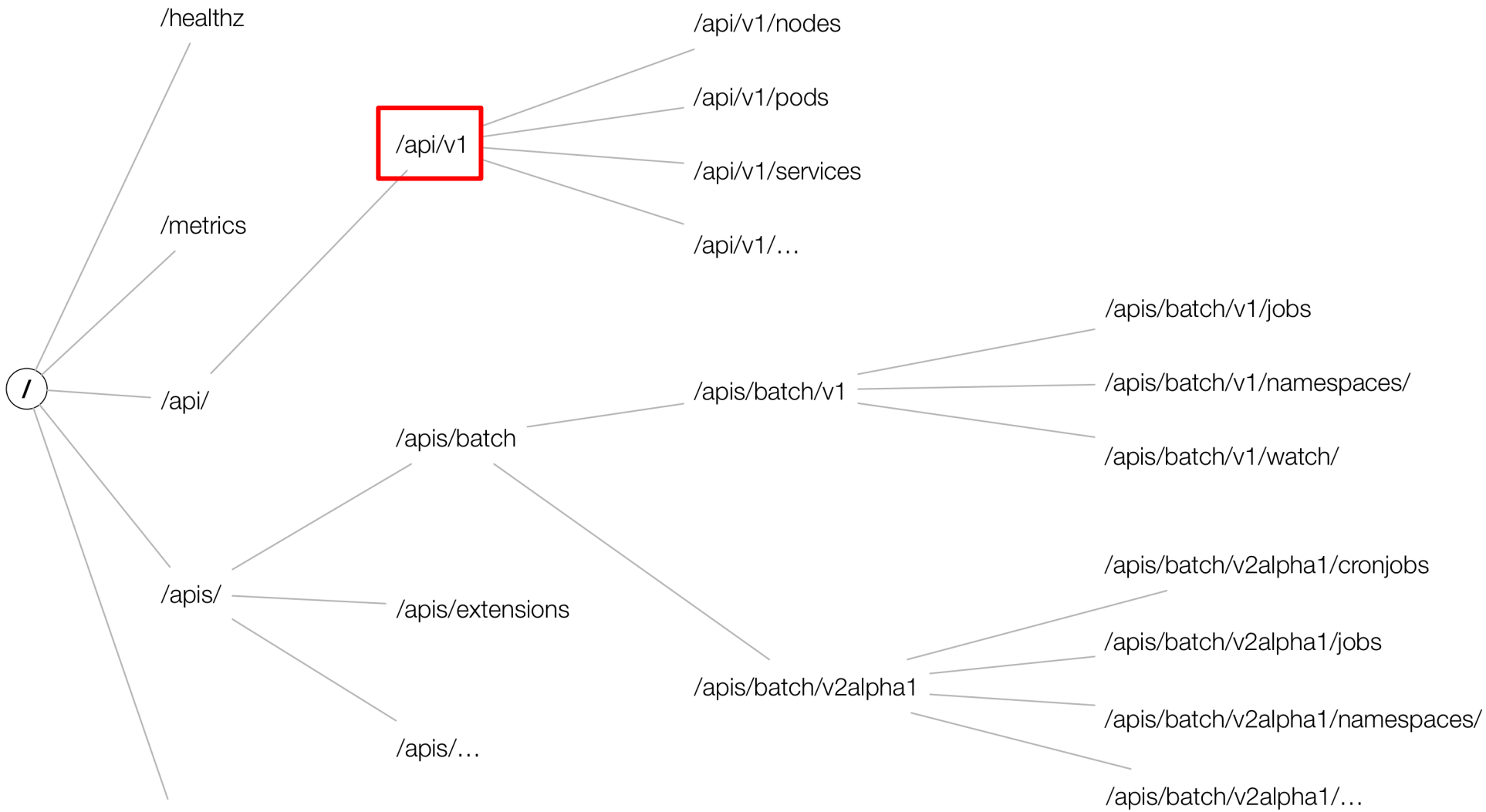
API Versioning



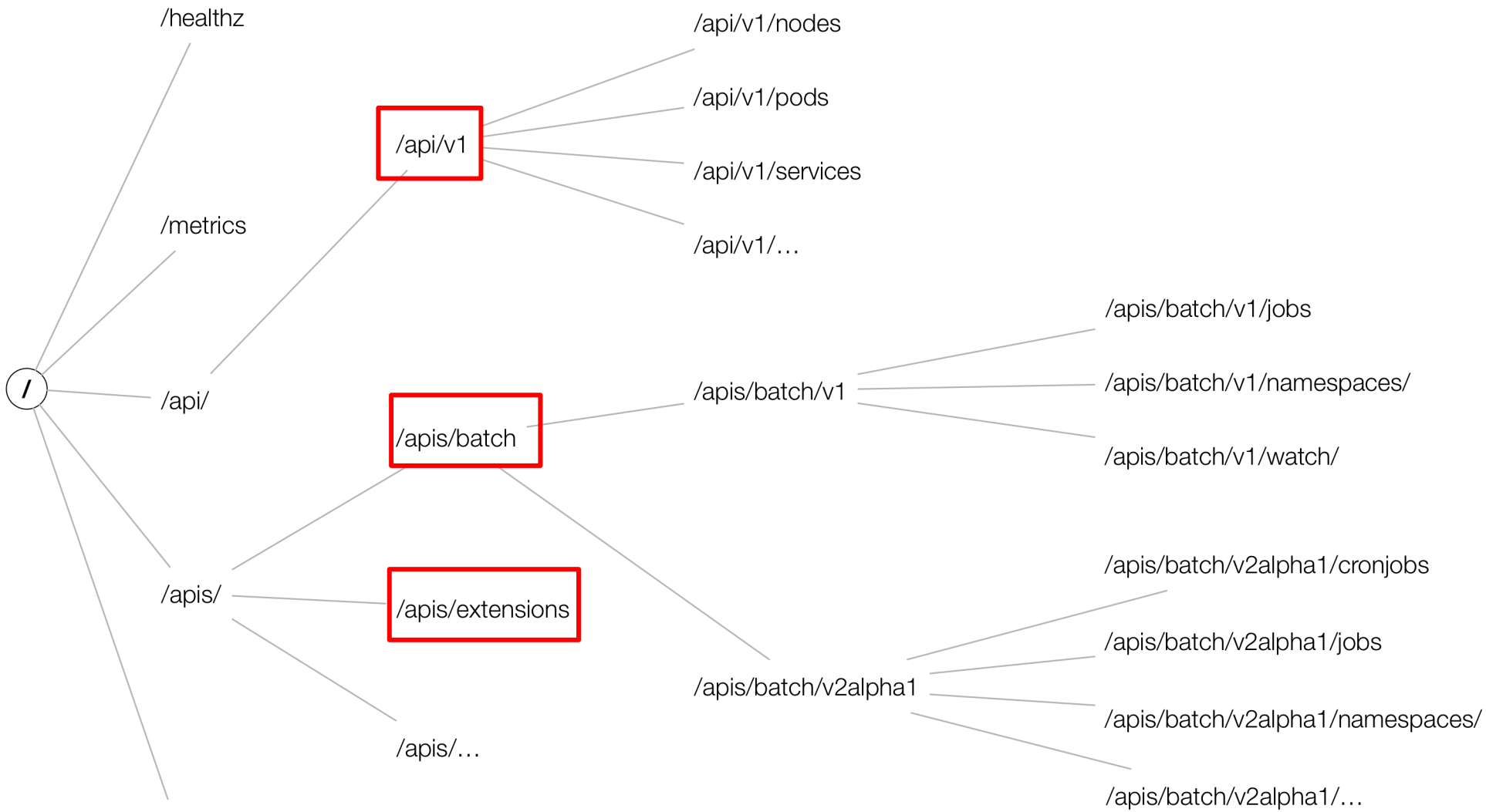
API


/
/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/extensions
/apis/…
/apis/batch/v2alpha1
/apis/batch/v2alpha1/cronjobs
/apis/batch/v2alpha1/jobs
/apis/batch/v2alpha1/namespaces/
/apis/batch/v2alpha1/…
/…

API Groups



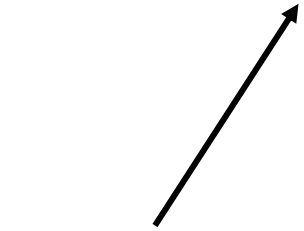
API Groups



API Group

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

/apis/batch/v1/jobs



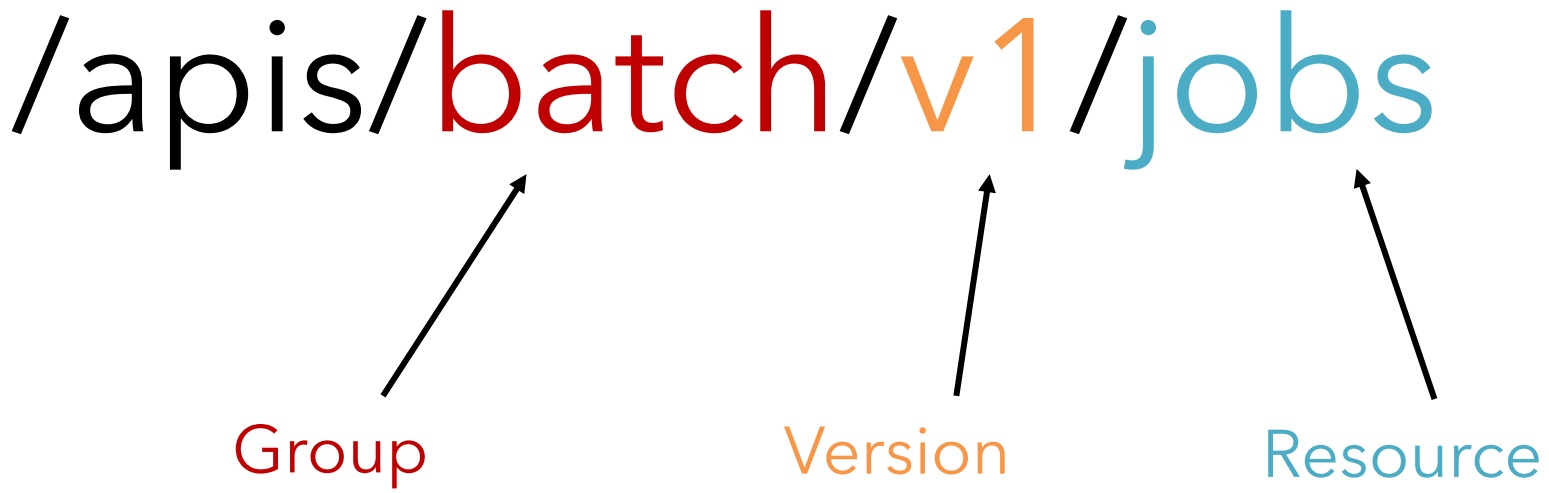
Group

API Version

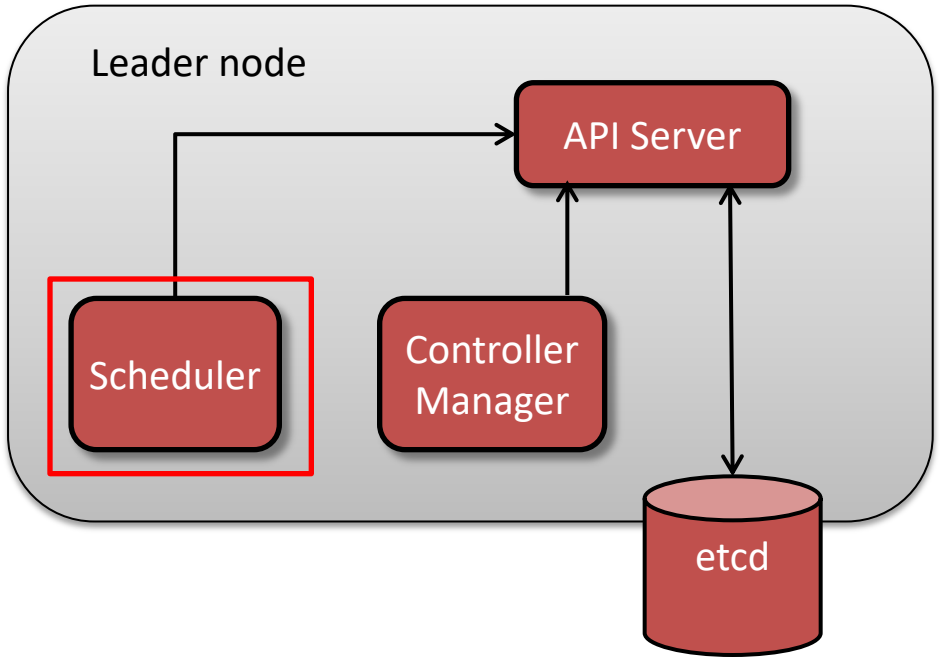
- Each API Group can be part of multiple versions
  - v1alpha1 -> v1beta1 -> v1

API Resource

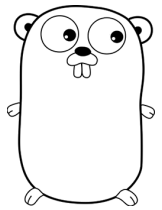
- System entity being manipulated as JSON over HTTP



Kubernetes Leader Node Components

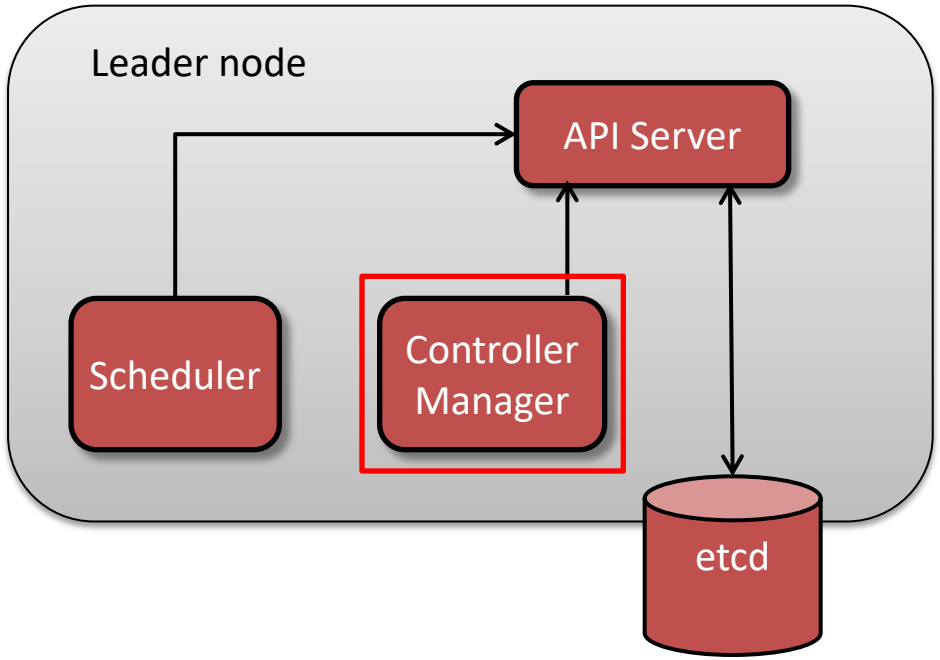


- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared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

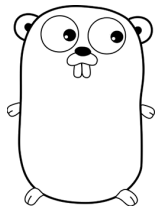


K8s components written in Go (golang.org)

Kubernetes Leader Node Components

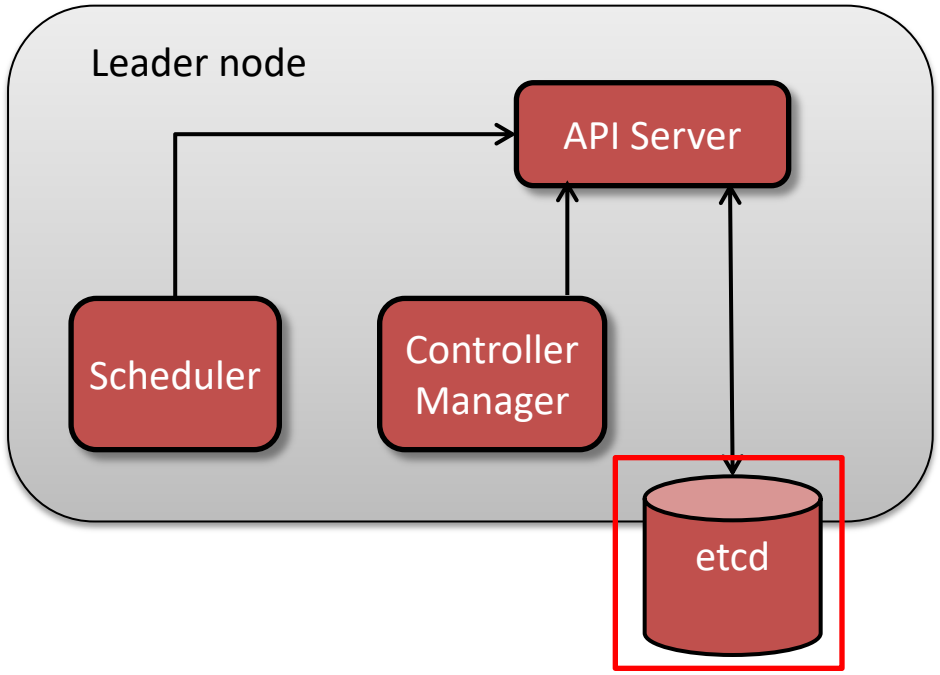


- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared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

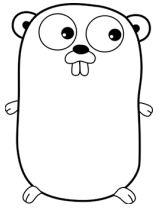


K8s components written in Go (golang.org)

Kubernetes Leader Node Components

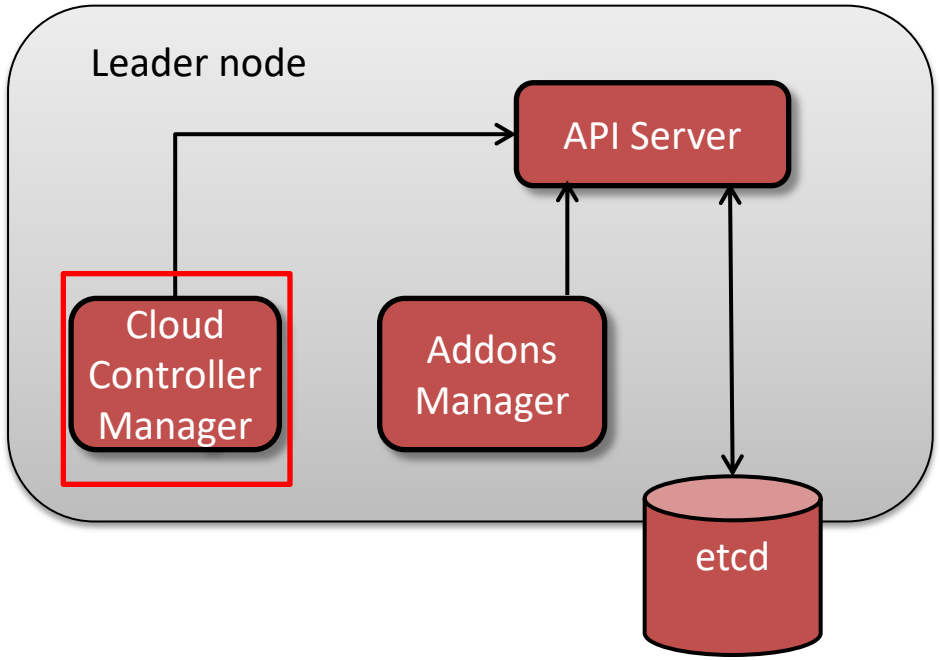


- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared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



K8s components written in Go (golang.org)

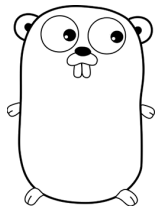
Leader Node Additional Components



- **cloud-controller-manager**: runs controllers interacting with underlying IaaS providers – alpha feature

  Allows cloud vendor-specific code to be separate from main K8s system components

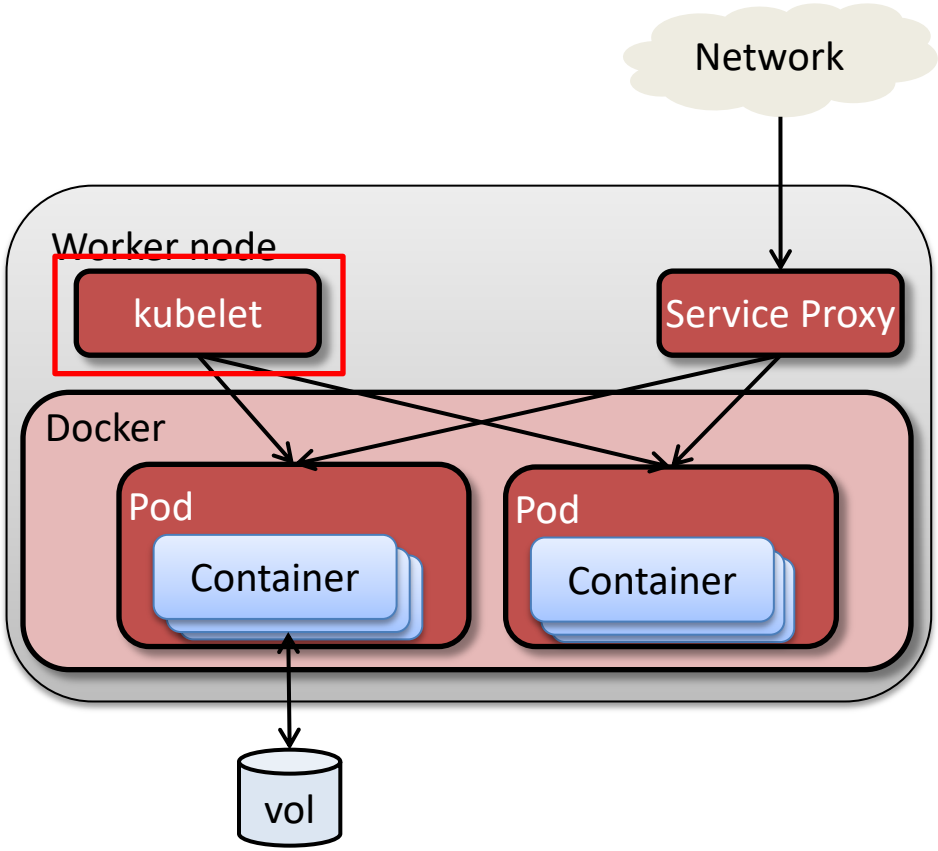
  - **addons-manager**: creates and maintains cluster addon resources in 'kube-system' namespace, e.g.
  - **Kubernetes Dashboard**: general web UI for application and cluster management
  - **kube-dns**: serves DNS records for K8s services and resources
  - Container resource monitoring and cluster-level logging



K8s components written in Go (golang.org)

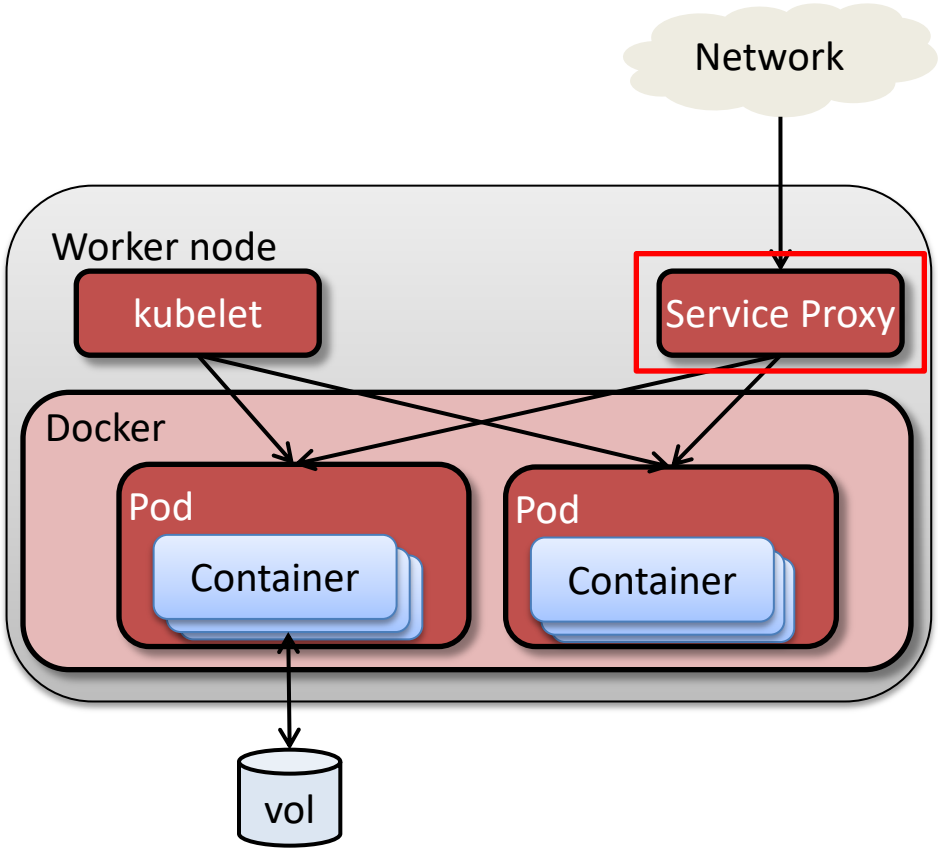
Kubernetes Member Node Components

- **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



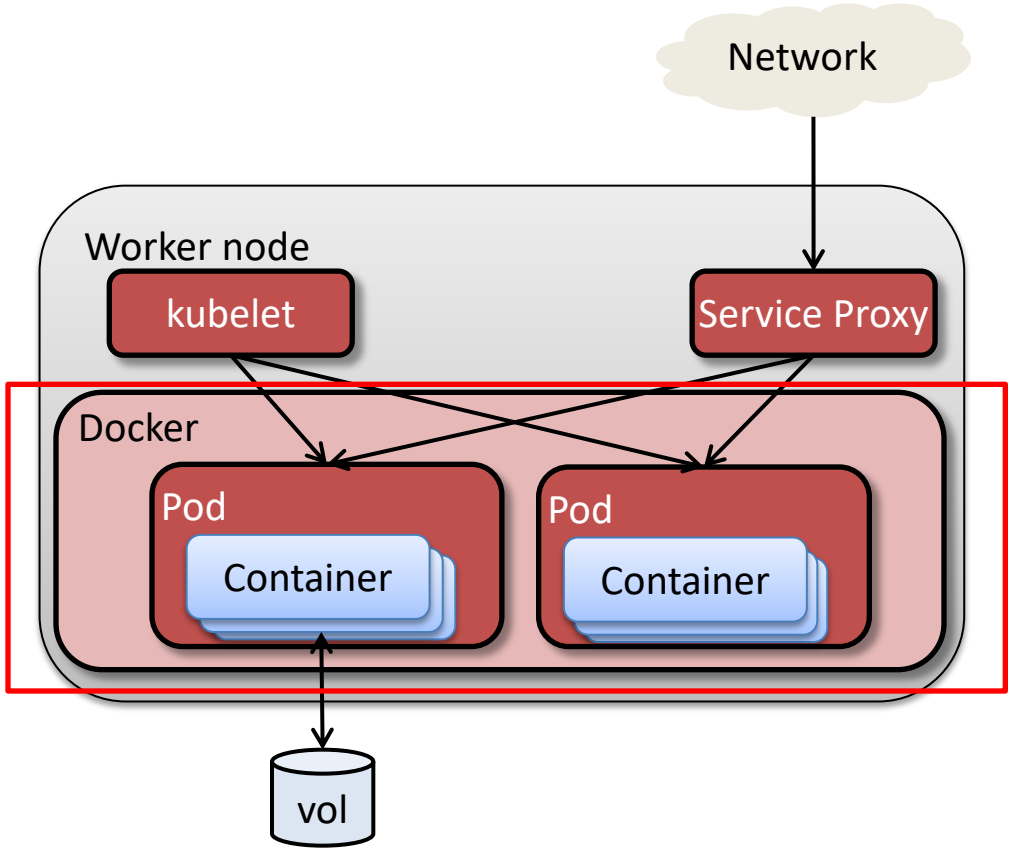
Kubernetes Member Node Components

- **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
- **Docker:** container runtime

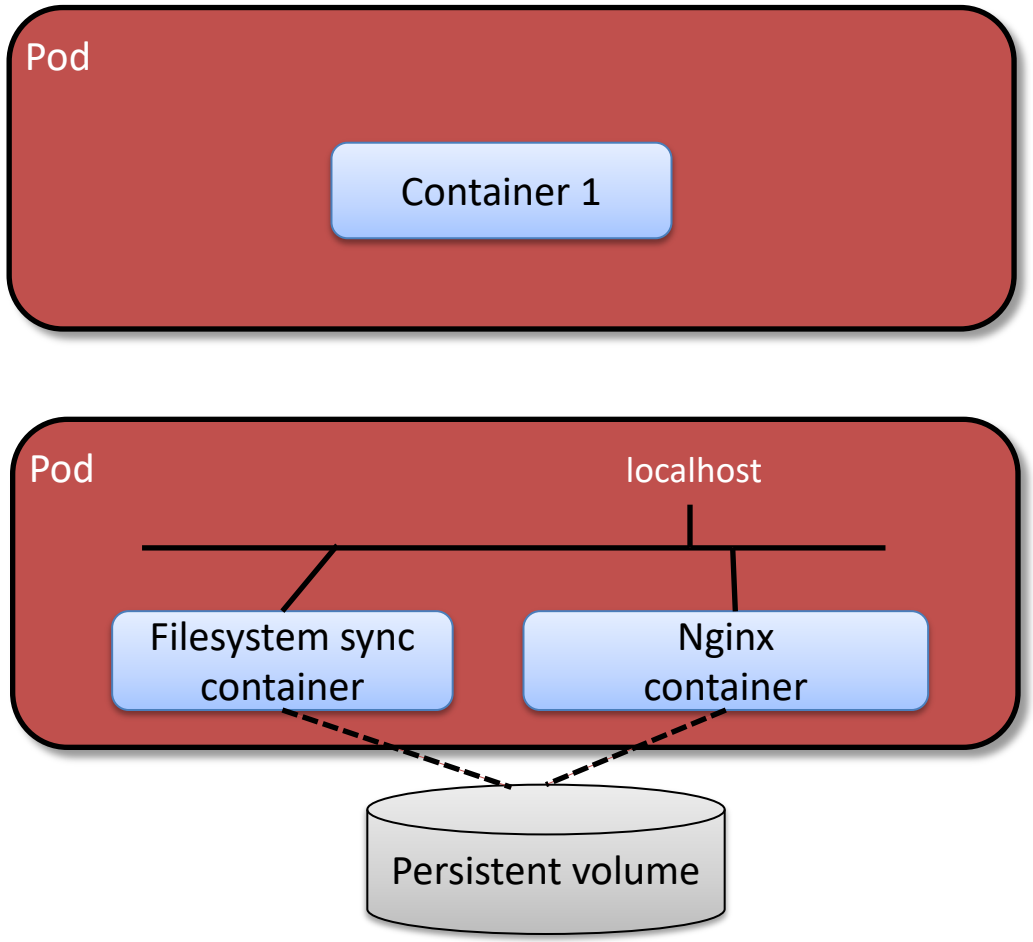


Kubernetes Pods

What is a Kubernetes Pod?

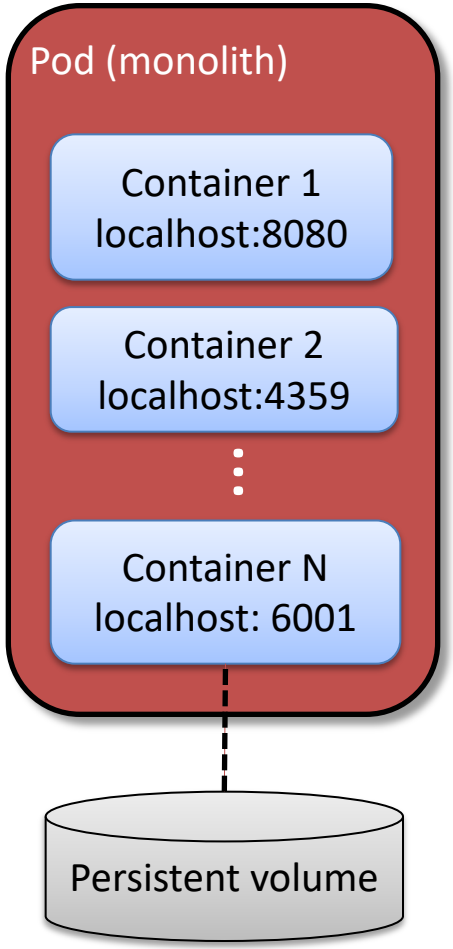
Kubernetes design intention: Pod == application instance

- 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



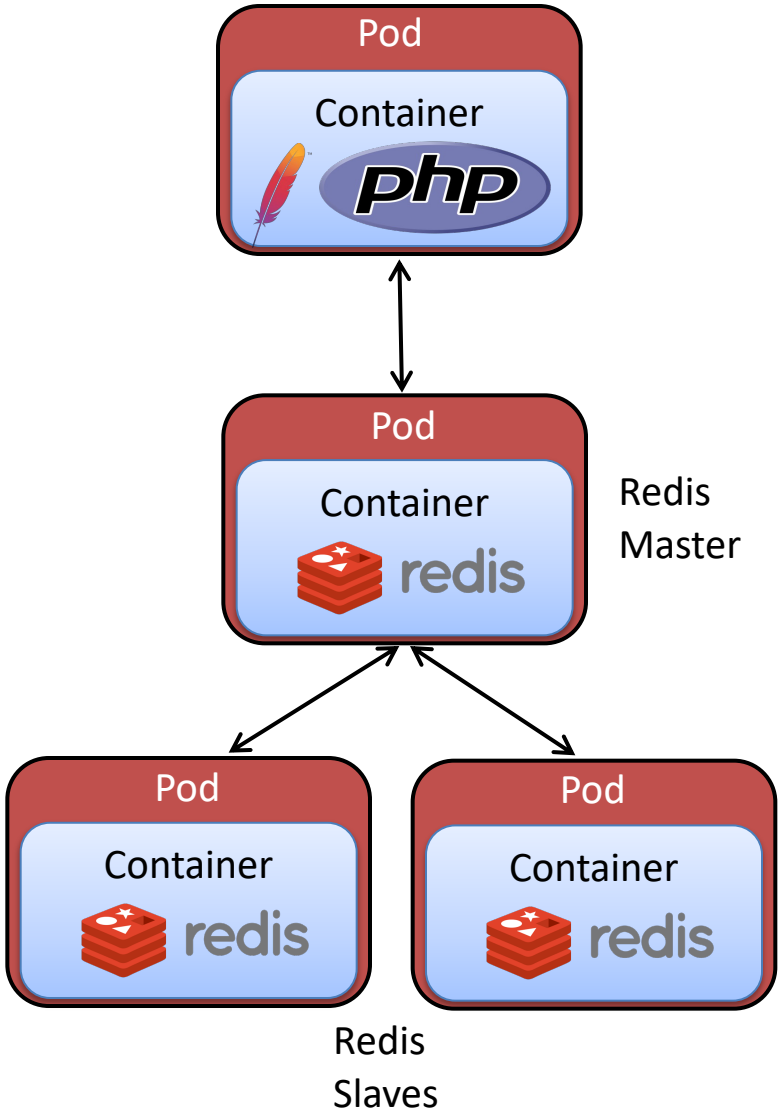
Pods Enable Deployment Flexibility

- 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



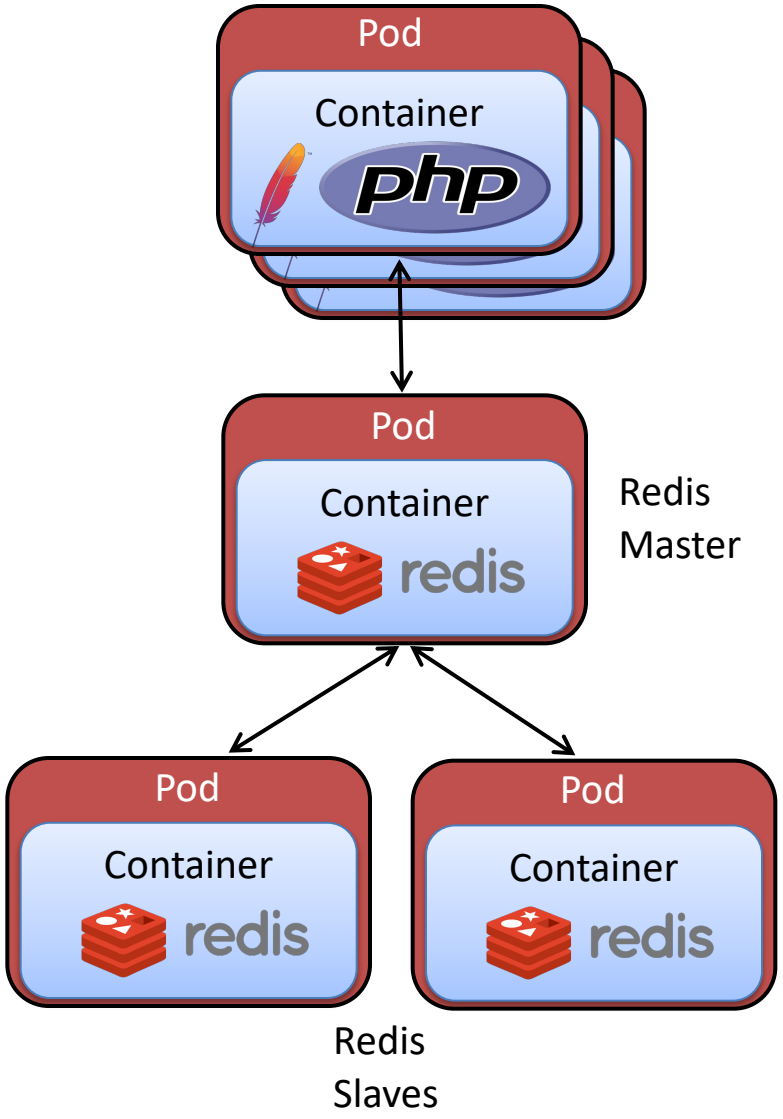
Pods Enable Deployment Flexibility

- 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

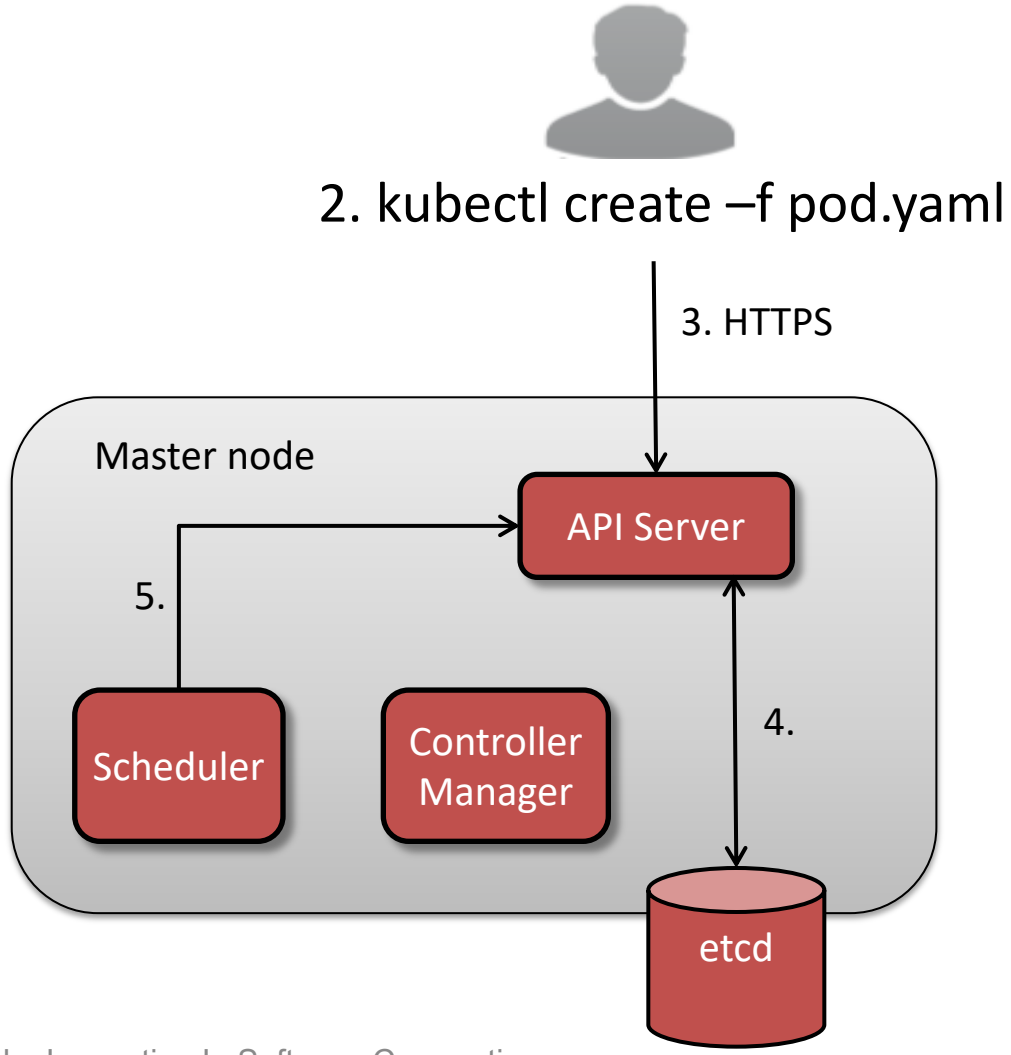
```
apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
spec:
  containers:
  - image: tomcat
    name: test-tomcat
    ports:
    - containerPort: 8080
  - image: mysql
    name: test-mysql
    ports:
    - containerPort: 3306
```

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

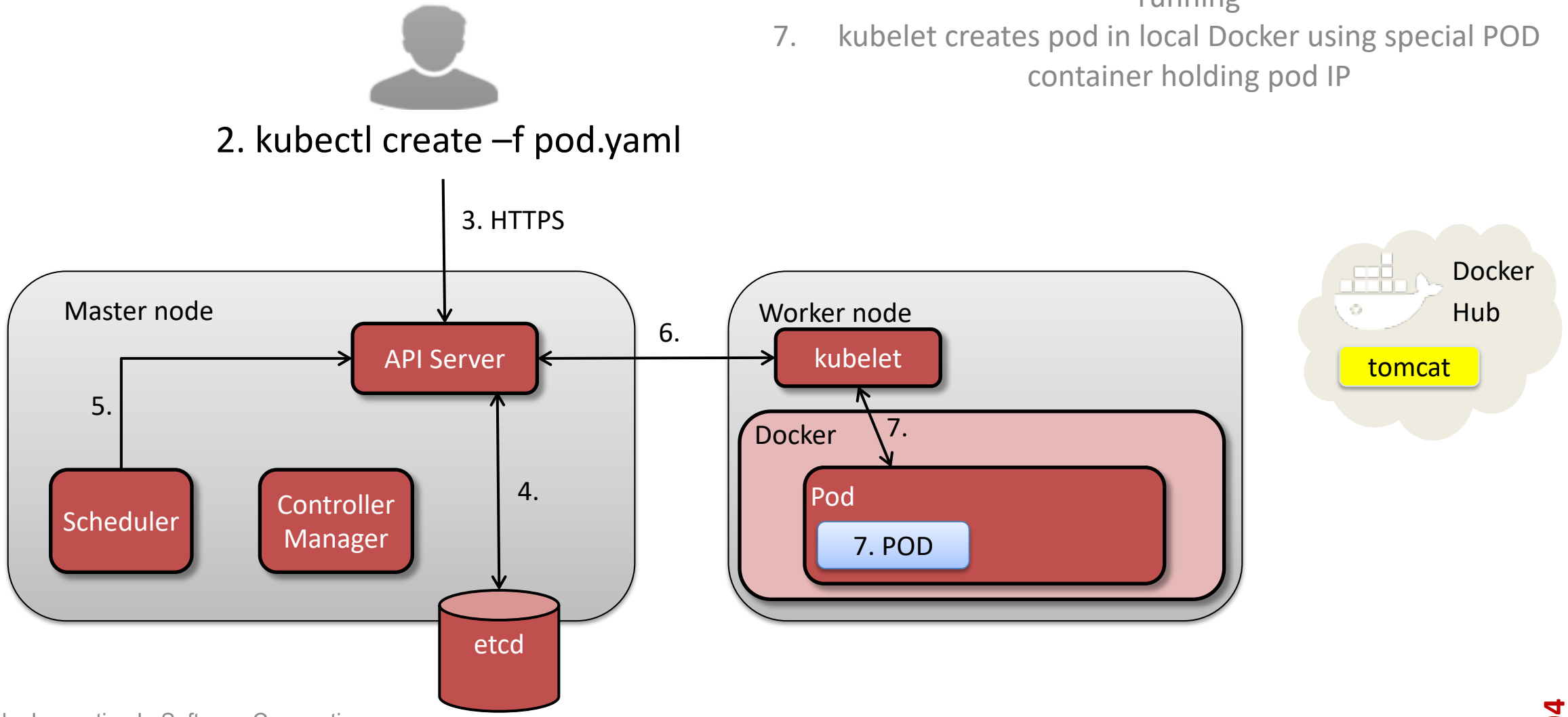
Pod Creation Process



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

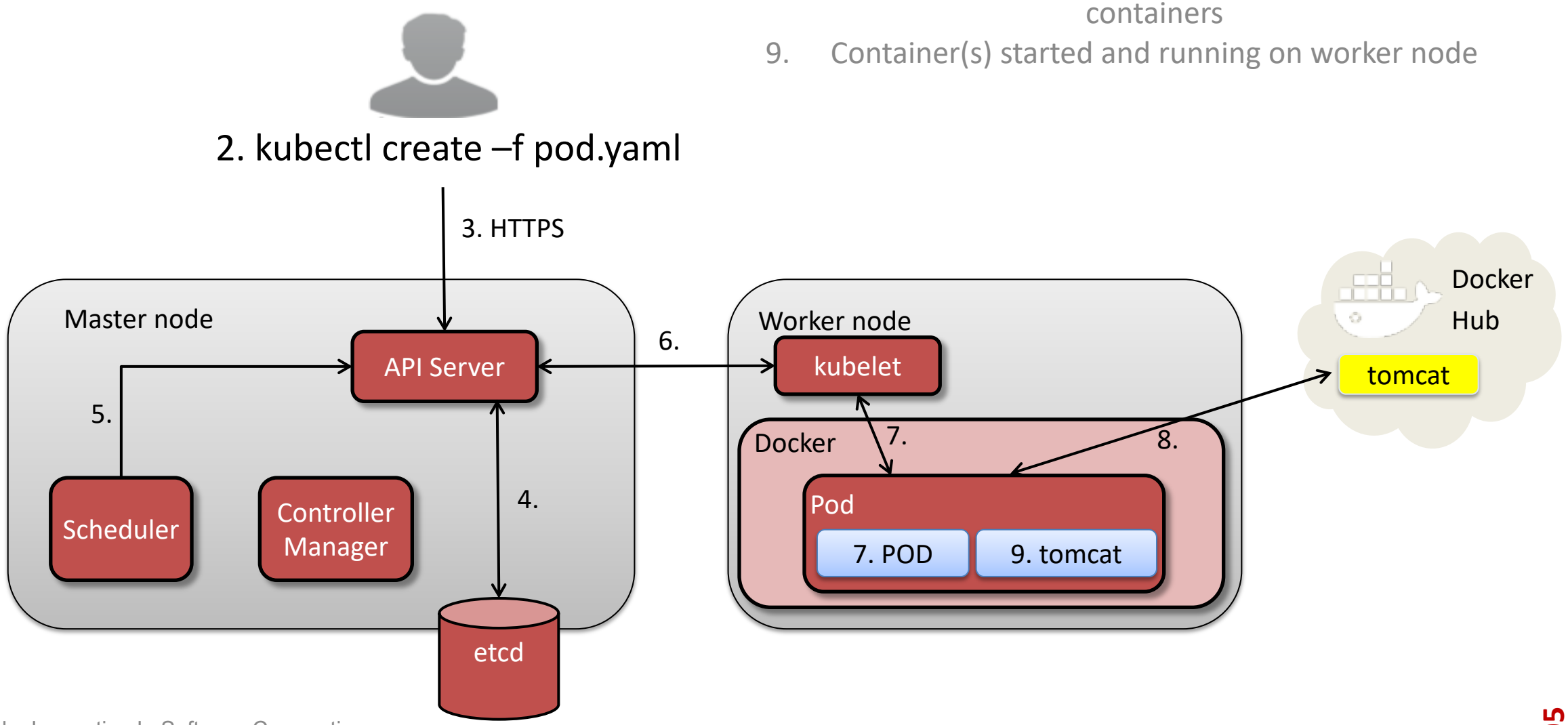
Pod Creation Process

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

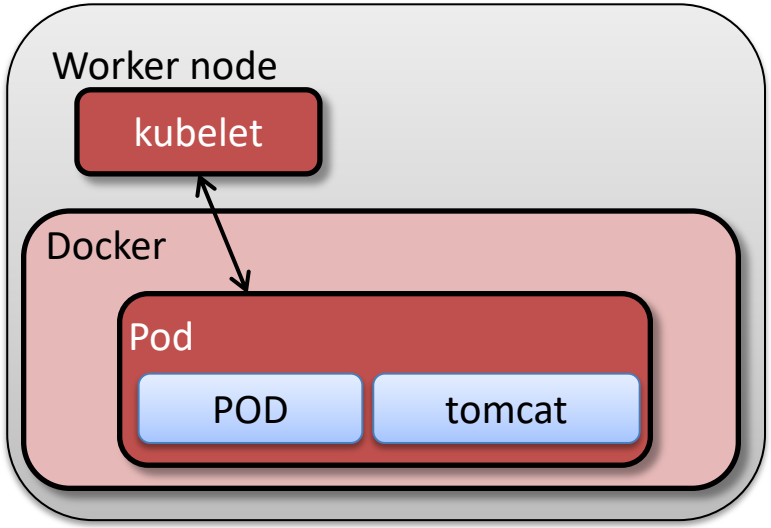
8. Kubelet pulls Docker image(s) for pod workload containers
9. Container(s) started and running on worker node



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
```

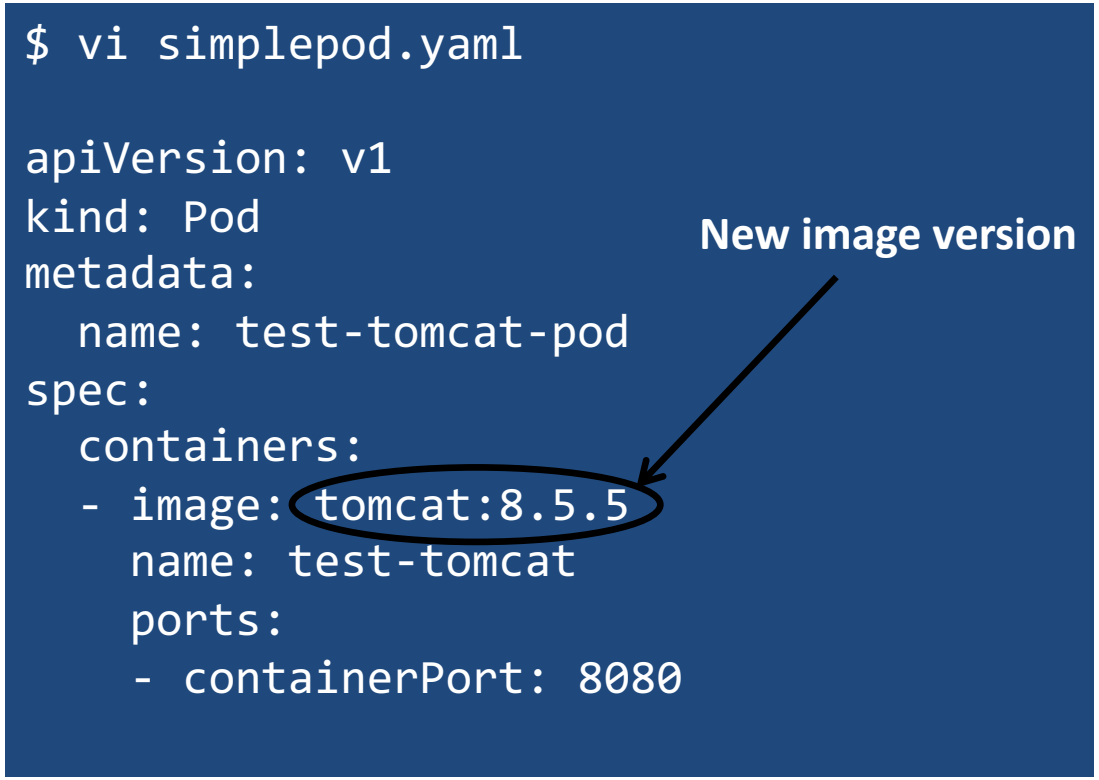


Modifying a Pod

Change the container version

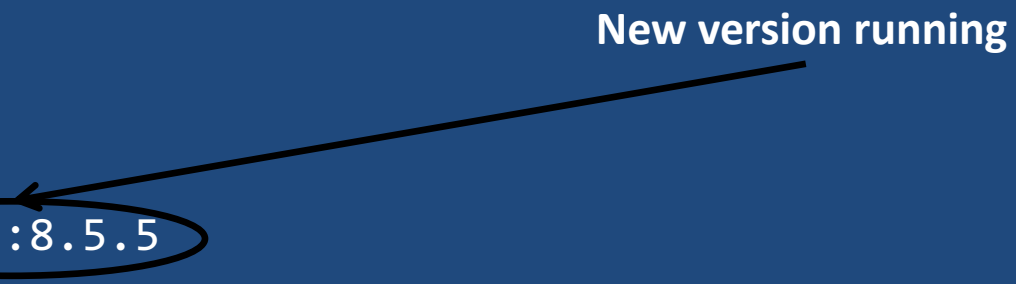
- You can make changes to the desired state of a pod via updating the manifest file
- Changes can then be applied to the pod via the command
  - `kubectl apply –f <manifest.yaml>`
- Changing a container image as shown will result in K8s automatically killing and recreating the pod's workload container

```
$ 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
```



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
...
```



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

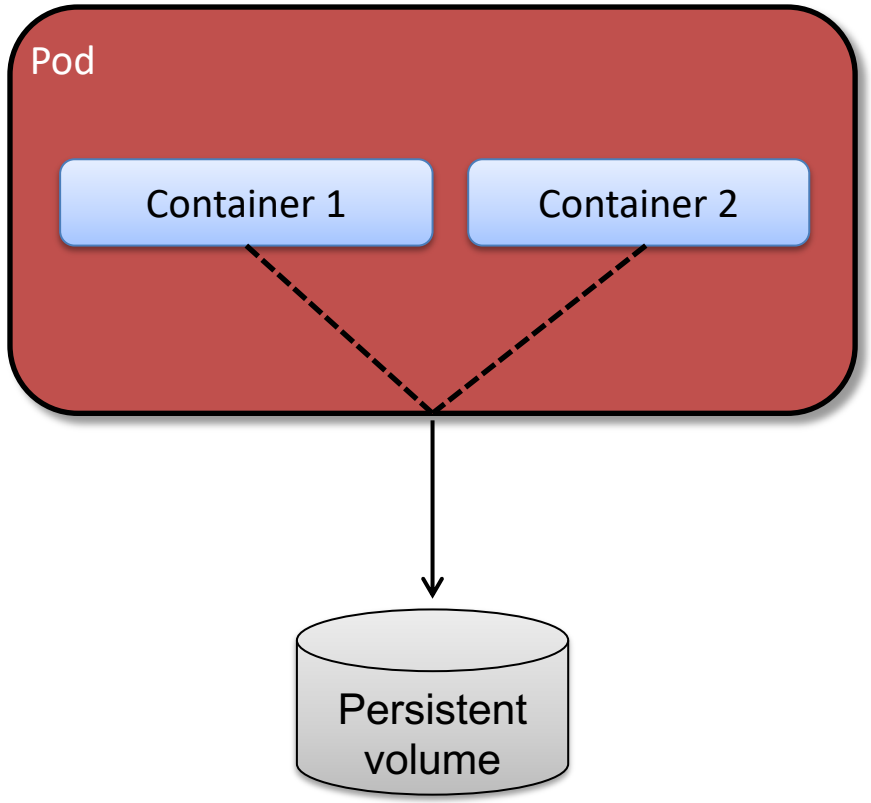
- You can display pod labels via a flag on the *kubectl* command

```
$ 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

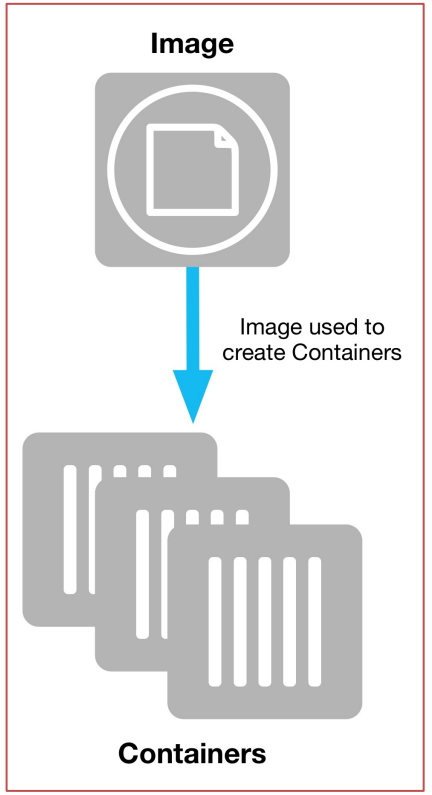
Docker allows you to package an application with its dependencies into a standardized unit for software development and deployment

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)

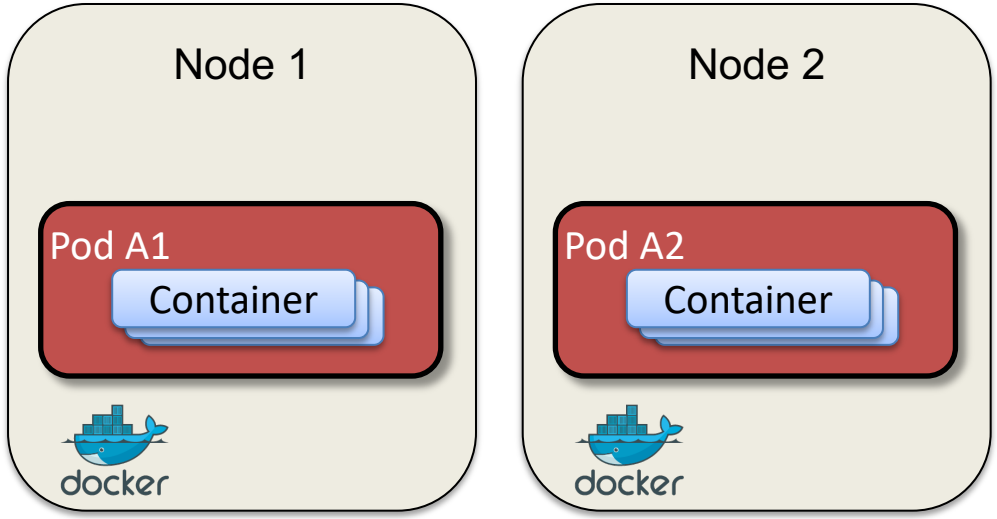
Container

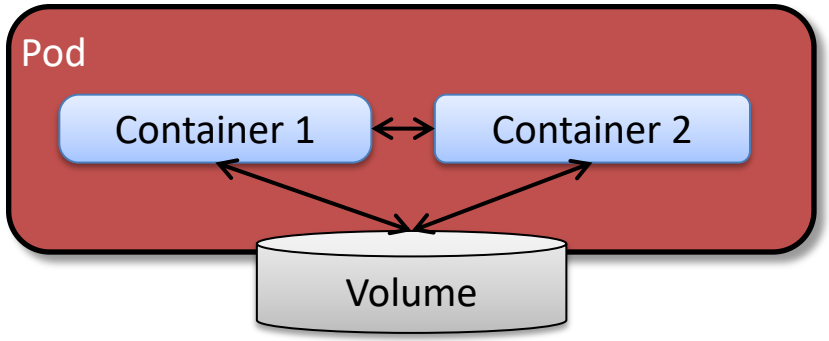
- Isolated application instance
- Created from a Docker image
- Based on Linux kernel primitives
  - Namespaces (resource visibility)
  - Control groups/cgroups (resource limits)



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

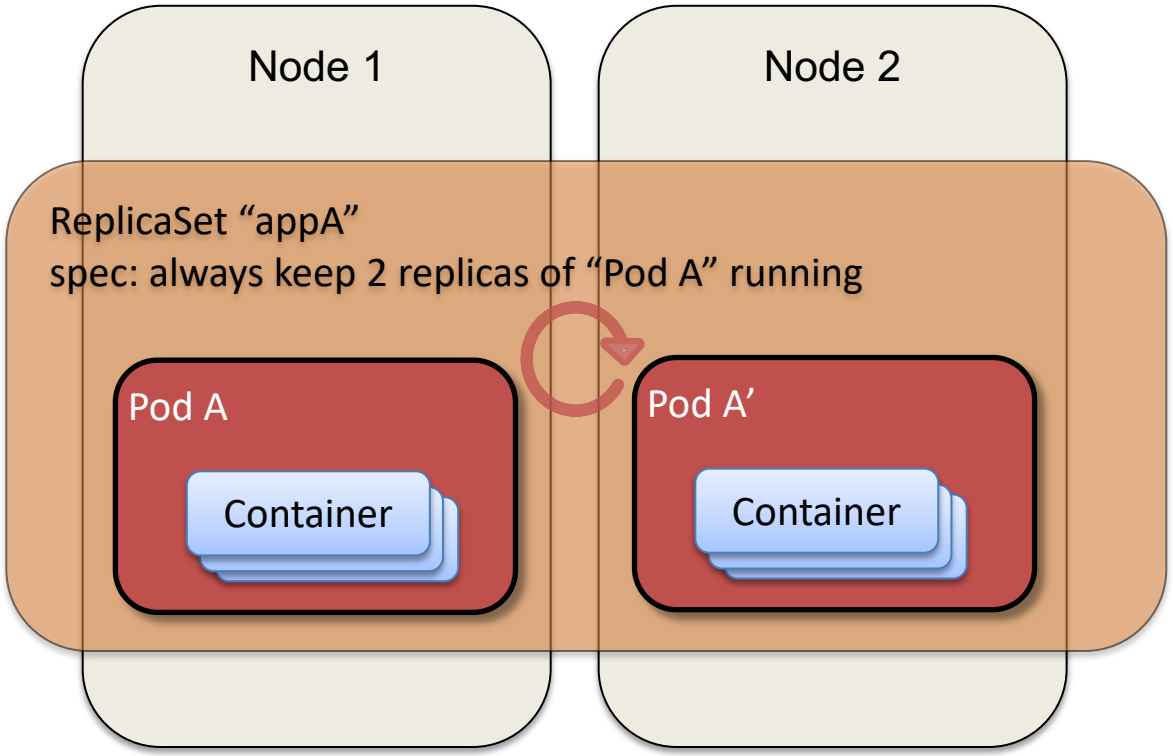




Controllers for Different Application Patterns

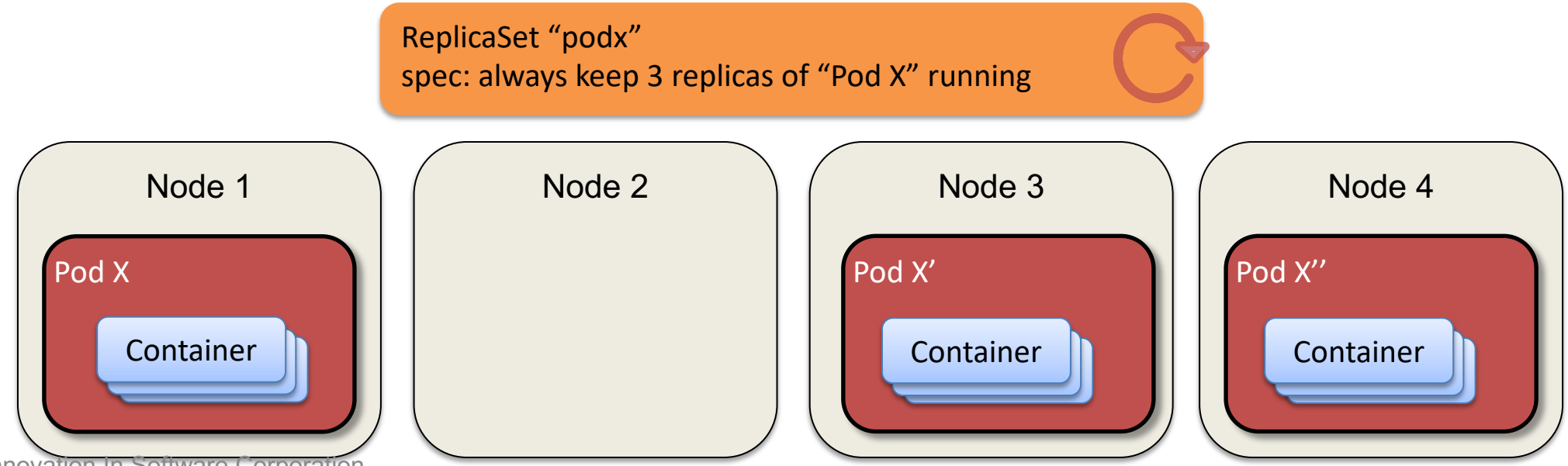
- K8s controller objects used to create and manage Pods according to different application patterns => control loops
  - **ReplicaSets** manage sets of replicas of stateless workloads to ensure availability
  - **StatefulSets** manage stateful workloads on stable storage to ensure consistency
  - **DaemonSets**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

Controller example: ReplicaSet



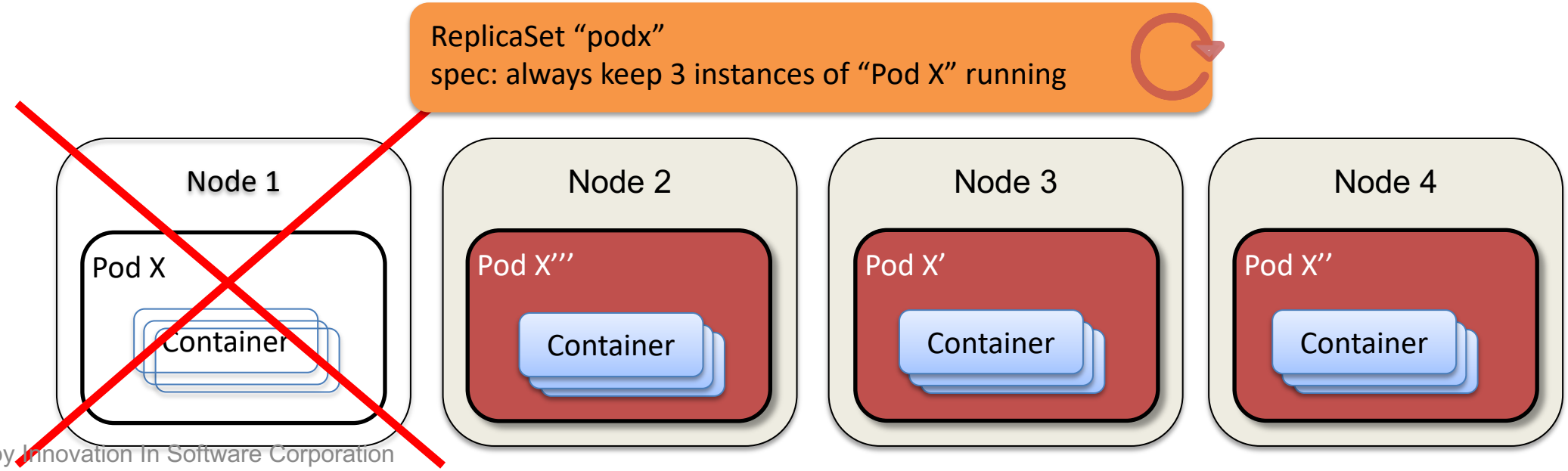
Controller Example: ReplicaSets

- 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

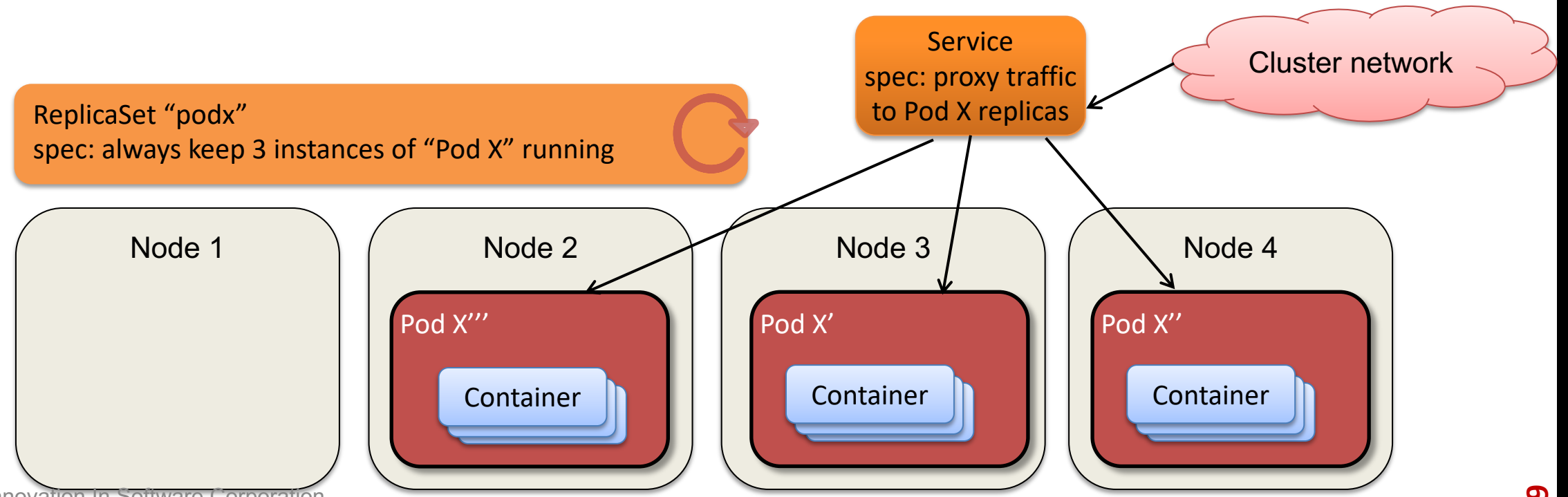
- When a Node fails, its Pods are lost
- K8s system manages the state of the ReplicaSet back to the declared configuration
- Changing the configuration will result in management to new state, e.g. scale out



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

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

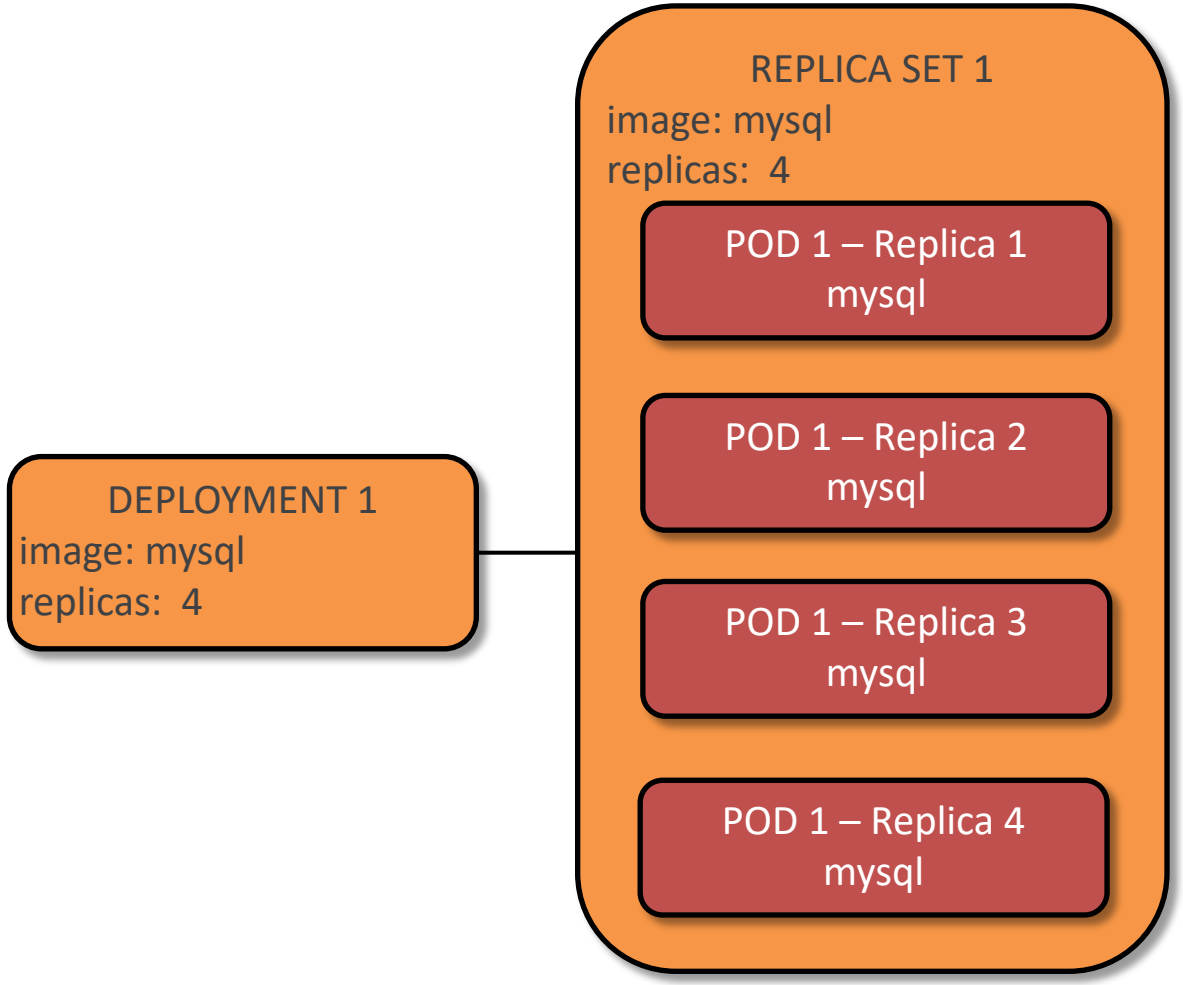
What if my Application isn't Stateless?

- 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

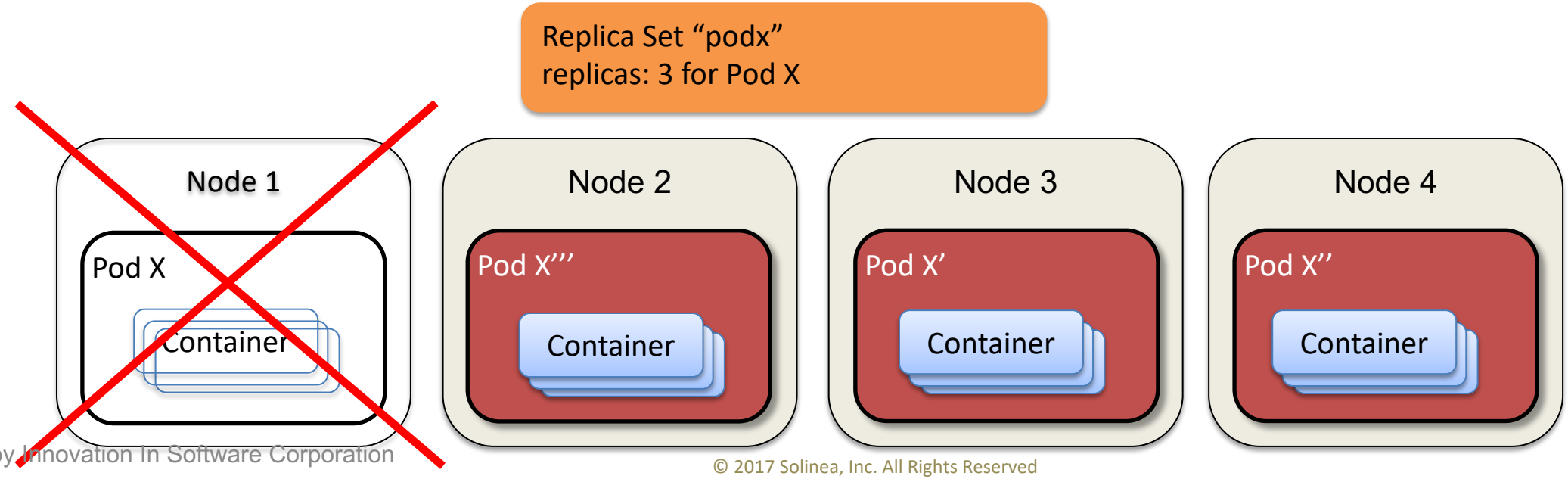
- Deployment creates and manages a Replica Set that manages a set of pods
- Replica count can be adjusted as needed to scale the Replica Set out and back
- Replica Set successor to the ReplicationController object



What is a Replica Set?

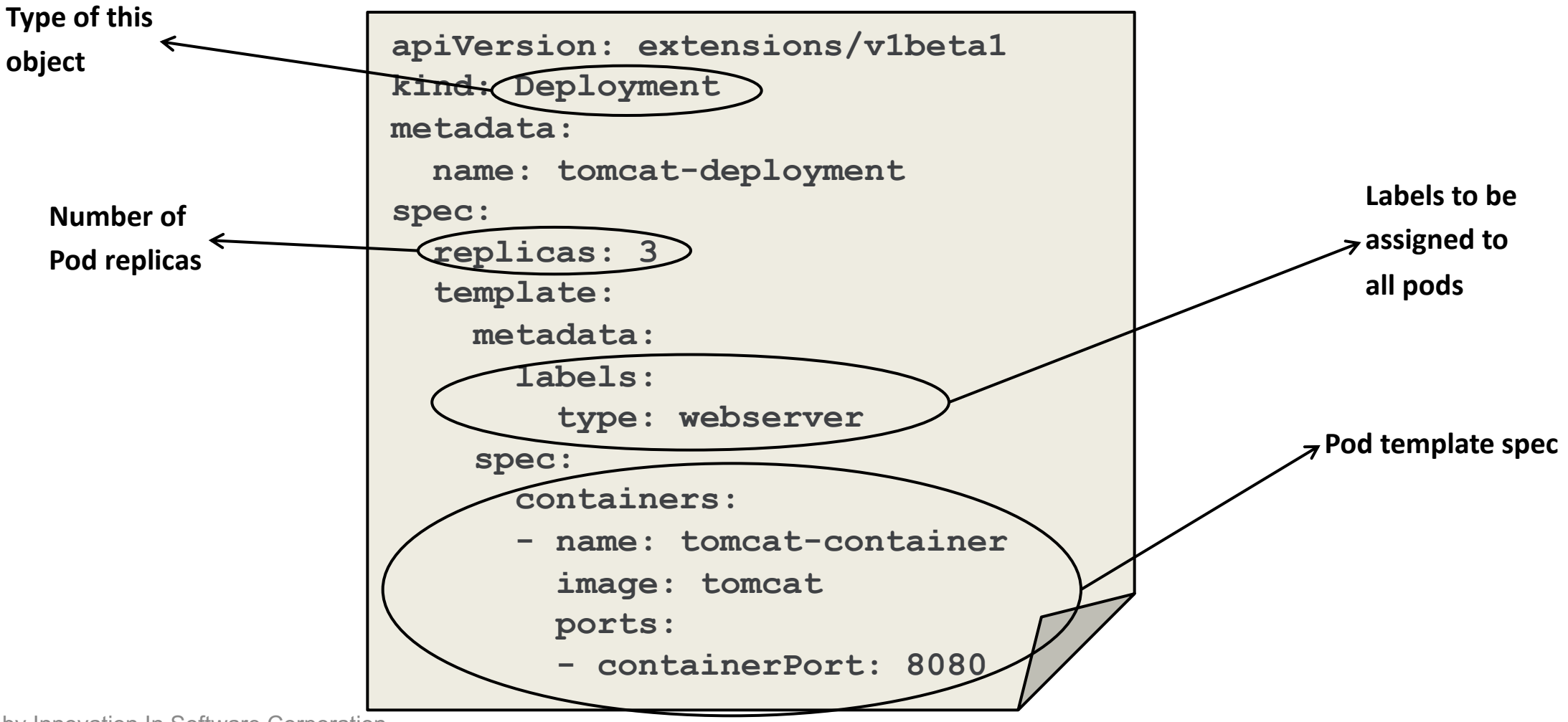
Provides scaling and high availability

- 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

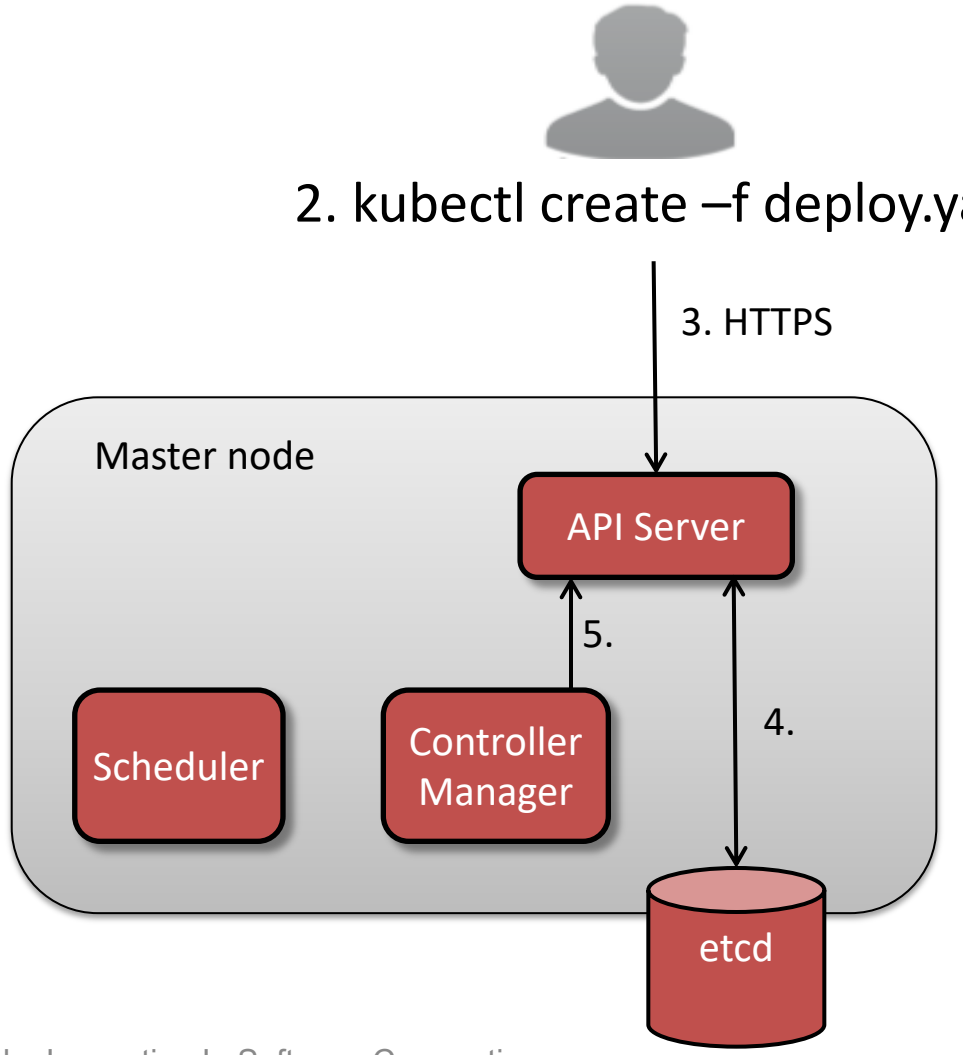


Examining a Deployment Manifest File

simpledeployment.yaml



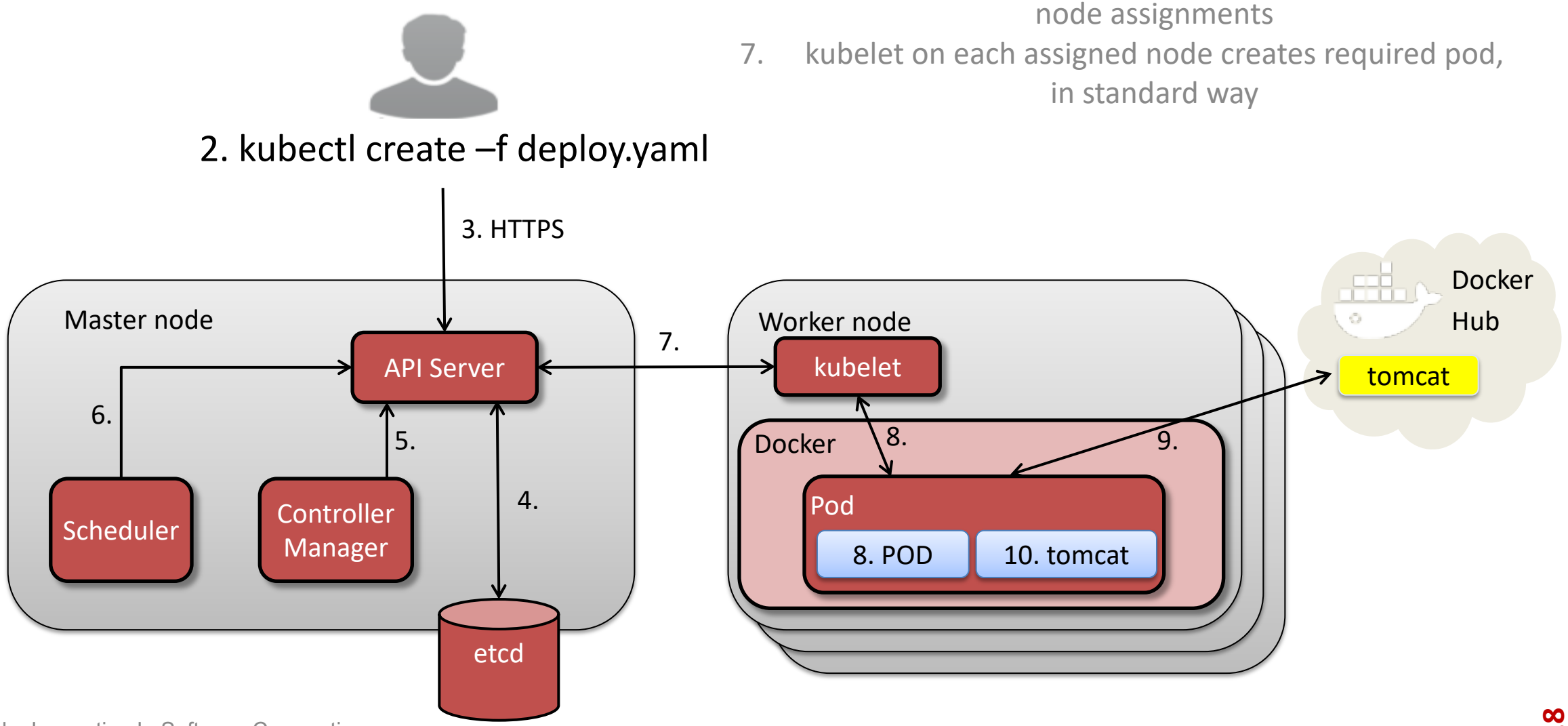
Deployment Creation Process



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



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

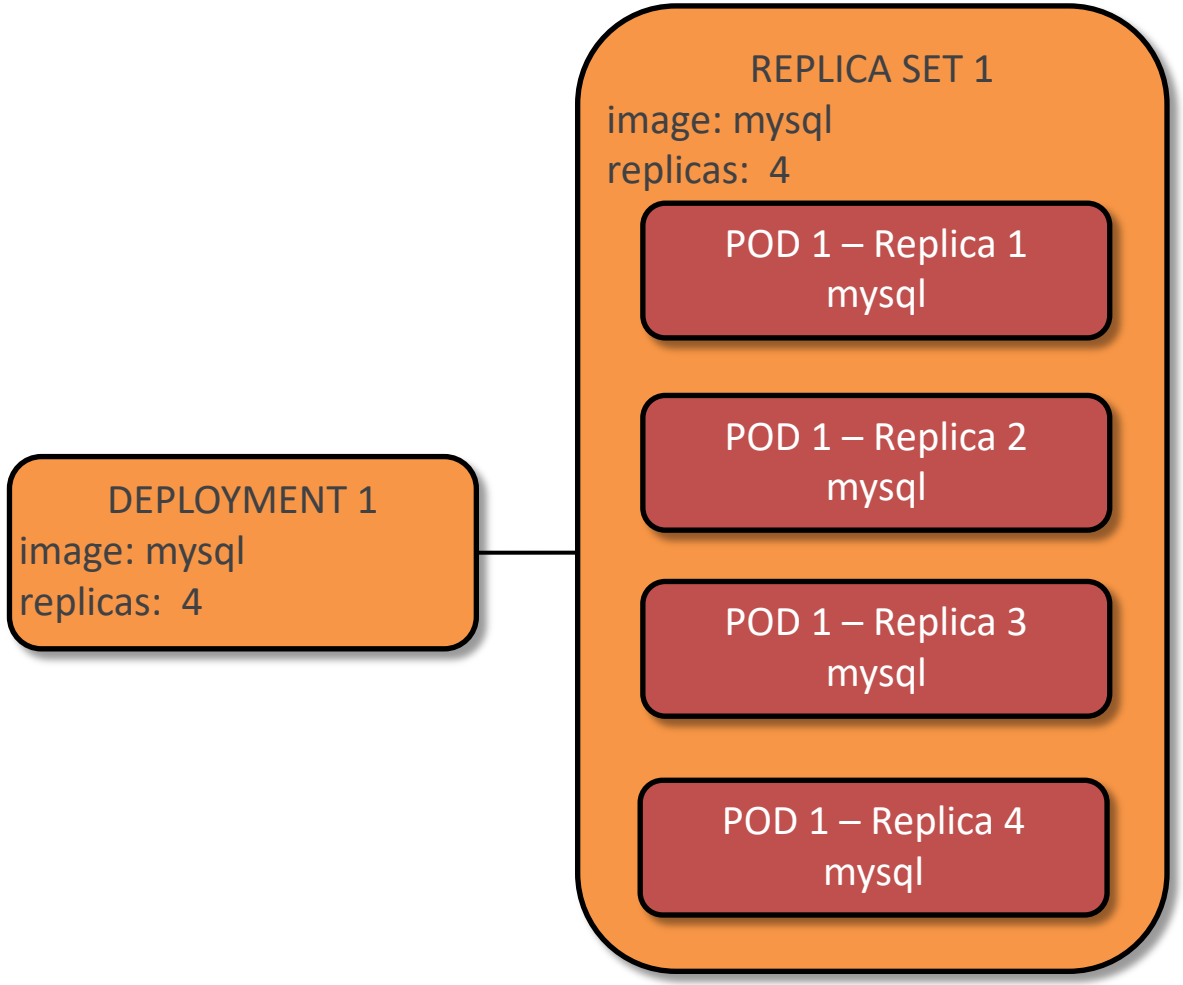
What if my Application isn't Stateless?

- 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

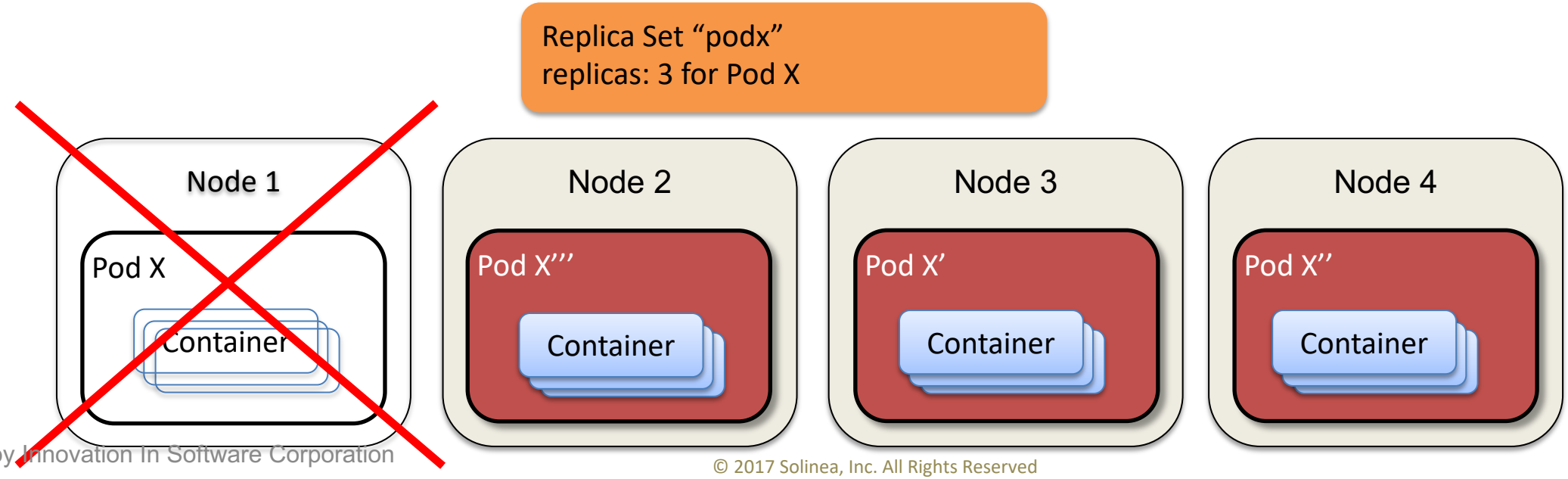
- Deployment creates and manages a Replica Set that manages a set of pods
- Replica count can be adjusted as needed to scale the Replica Set out and back
- Replica Set successor to the ReplicationController object



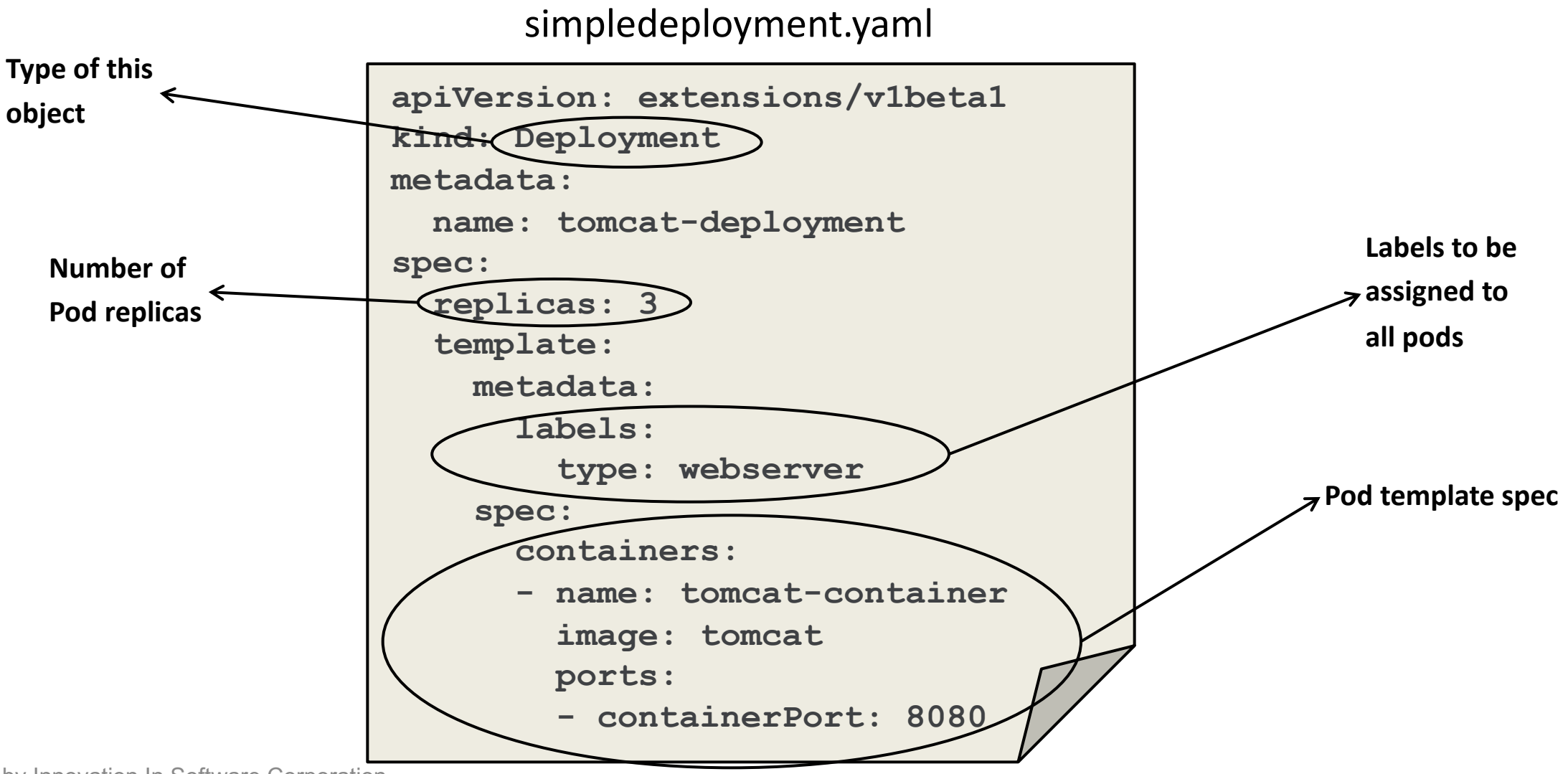
What is a Replica Set?

Provides scaling and high availability

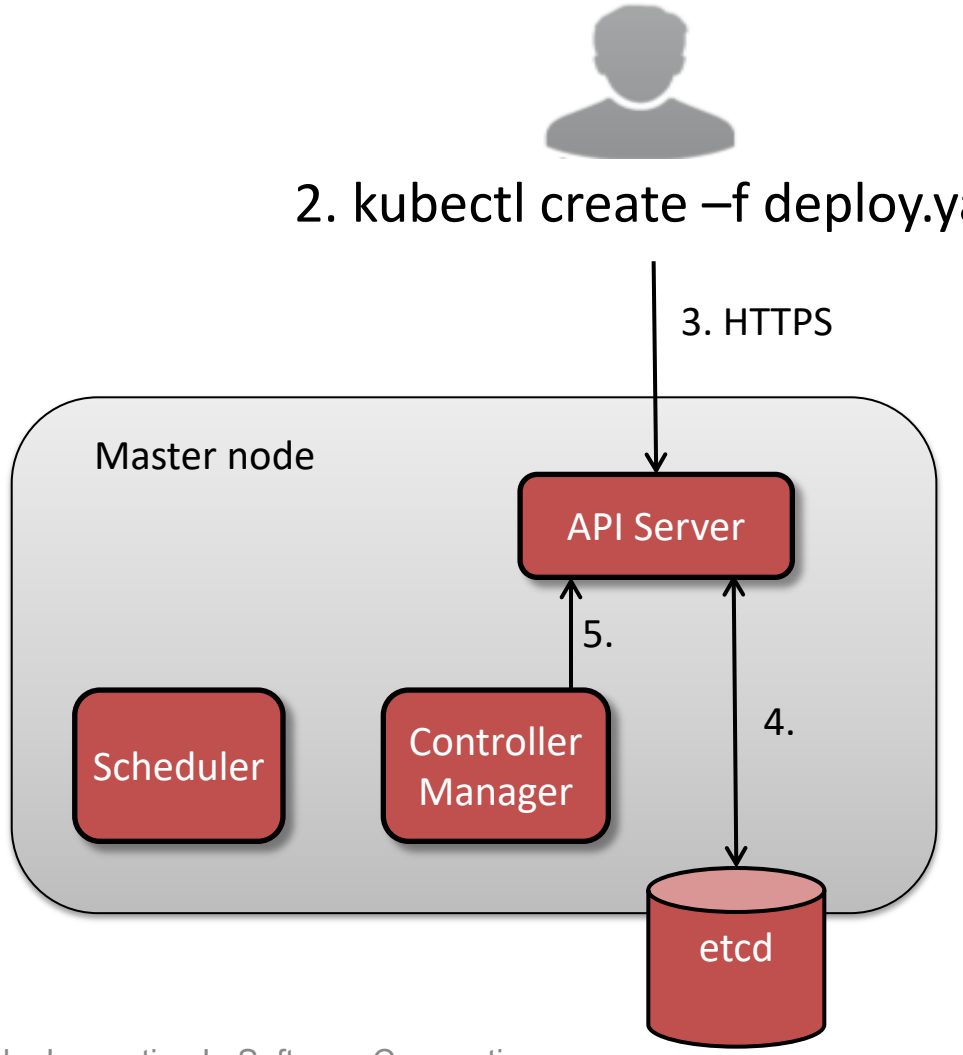
- Replica count can be changed to provide scaling on demand as needed
- If the node hosting a pod fails, the Kubernetes cluster will recreate the pod elsewhere to achieve the target number of replicas



Examining a Deployment Manifest File



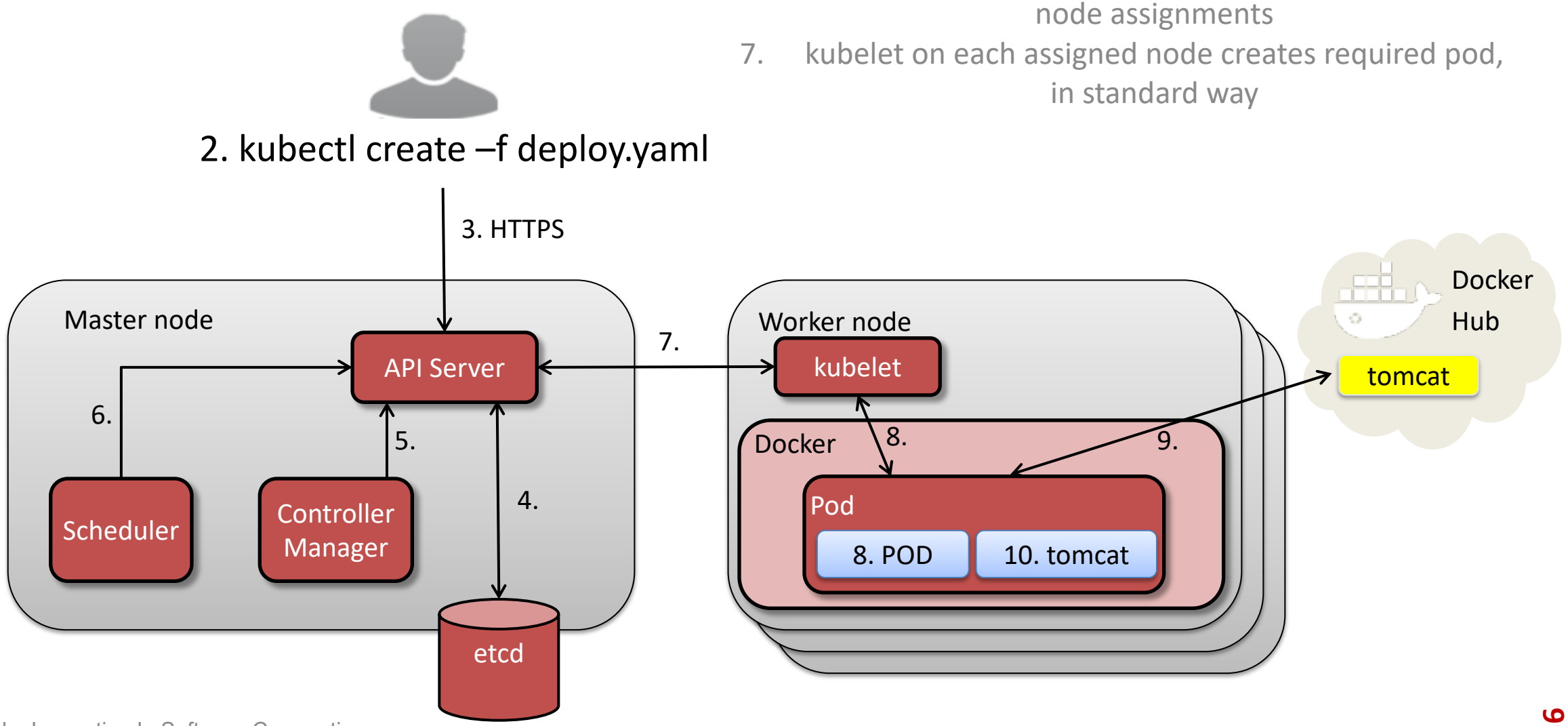
Deployment Creation Process



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



Lab: Deployments