
Intro to Kubernetes

logistics

- Class Hours:
- Start time is 9:30am
- End time is 4:30pm
- Class times may vary slightly for specific classes
- Breaks mid-morning and afternoon (10 minutes)

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

- Lunch:
- Lunch is 11:45am to 1pm
- Yes, 1 hour and 15 minutes
- Extra time for email, phone calls, or simply a walk.

- Miscellaneous
- Courseware
- Bathroom

Course Objectives

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

- State the function and purpose of Kubernetes
- Install and Configure Kubernetes
- Describe the Kubernetes architecture:
  - Cluster components
  - Pods, Services and Deployments
  - Expose applications with Services
- Deploy multi-tier containerized applications
- Interact with Kubernetes thorough the command-line and Dashboard

Agenda

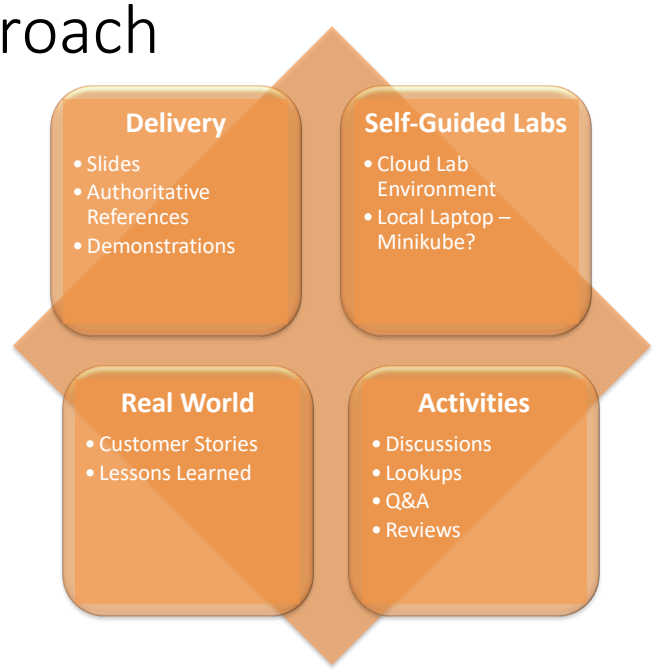
Day 1:

- Welcome and Introductions
- Introduction to Containers
- Introduction to Kubernetes
- Kubernetes Architecture
- Pods
- Networking
- Services
- Wrap-up Day 1

Day 2:

- Day 1 Review
- Deployments
- Data Persistence
- Role Based Access Controls
- Managing state of cluster
- ConfigMaps and Secrets
- Ingress load balancing
- Multi-node Kubernetes cluster
- Deploy multi-tier application
- Run Kubernetes locally with Minikube
- Course Wrap-up

Course Approach



End of Day Office Hours - 1:1 Assistance

The Training Dilemma

Meet the Instructor

Chris Hiestand

Software Engineer Consultant with a linux sysadmin and web/microservice programming background. Focused on the kubernetes ecosystem. Docker user since 2014, kubernetes since 2015.

www
https://kistek.consulting/

github
https://github.com/chrishiestand

mail
chris@kistek.consulting

Languages

- javascript/node.js
- python
- golang (in progress)
- php (please don't tell anyone)
- bash

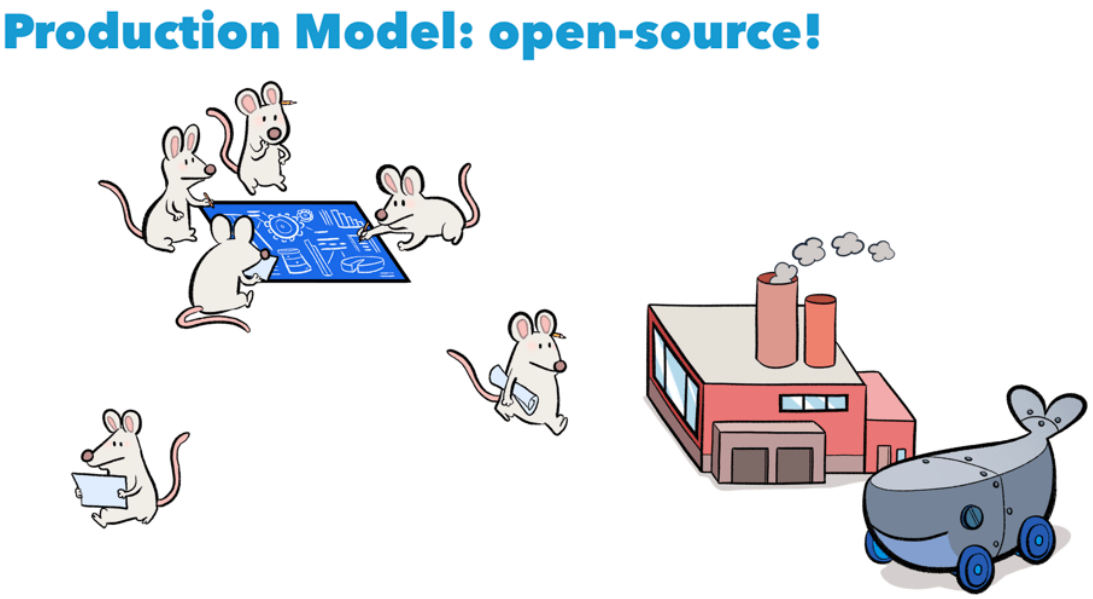
```
> docker run busybox echo hello world
hello world
```

Introductions

- Name
- Job Role
- Which statement best describes your Kubernetes experience?
  a. I am ***currently working*** with Kubernetes on a project/initiative
  b. I ***expect to work*** with Kubernetes on a project/initiative in the future
  c. I am ***here to learn*** about Kubernetes outside of any specific work related project/initiative
- Expectations for course (please be specific)

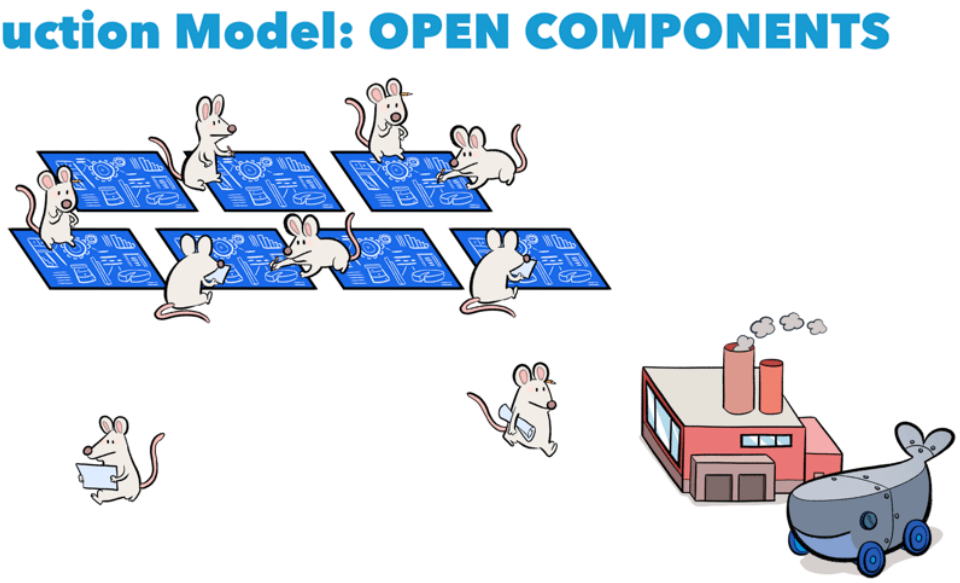
Docker Overview

What is Docker?



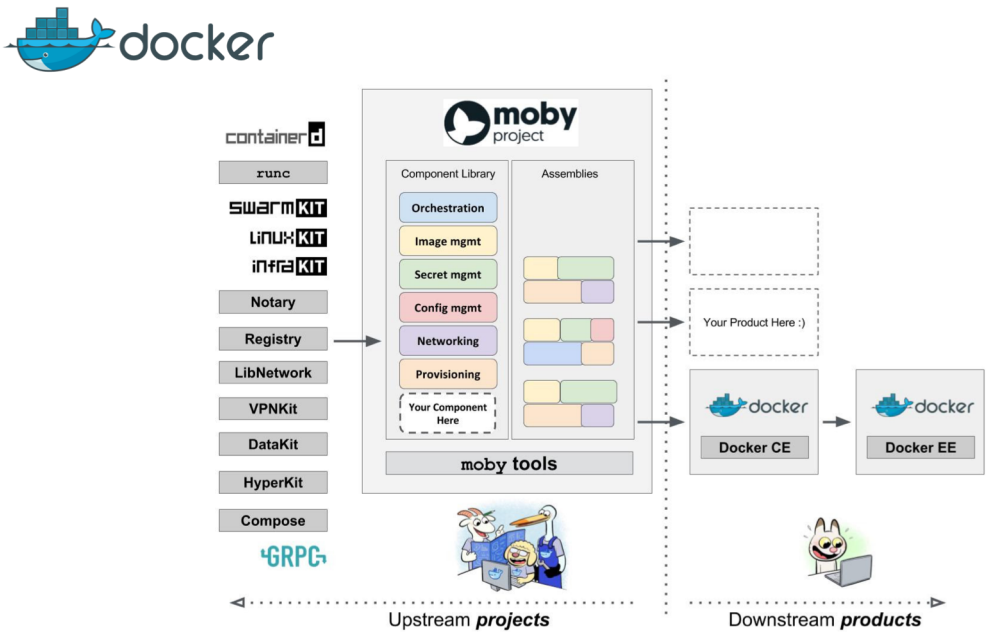
https://mobyproject.org

What is Docker?



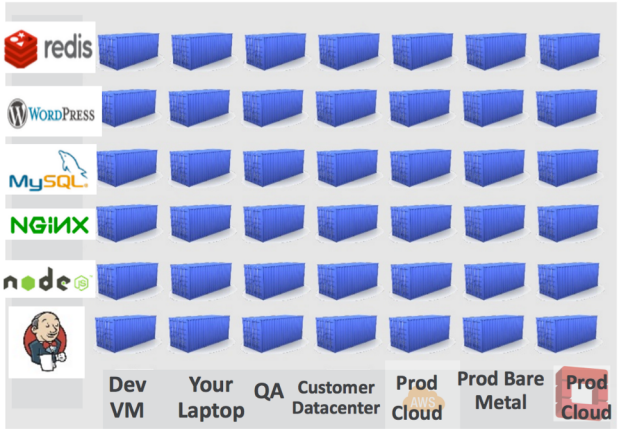
https://mobyproject.org

What is Docker?



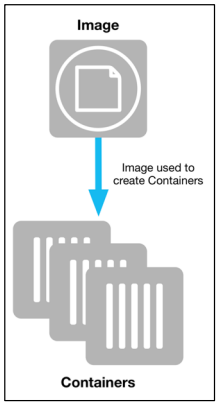
What is Docker?

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

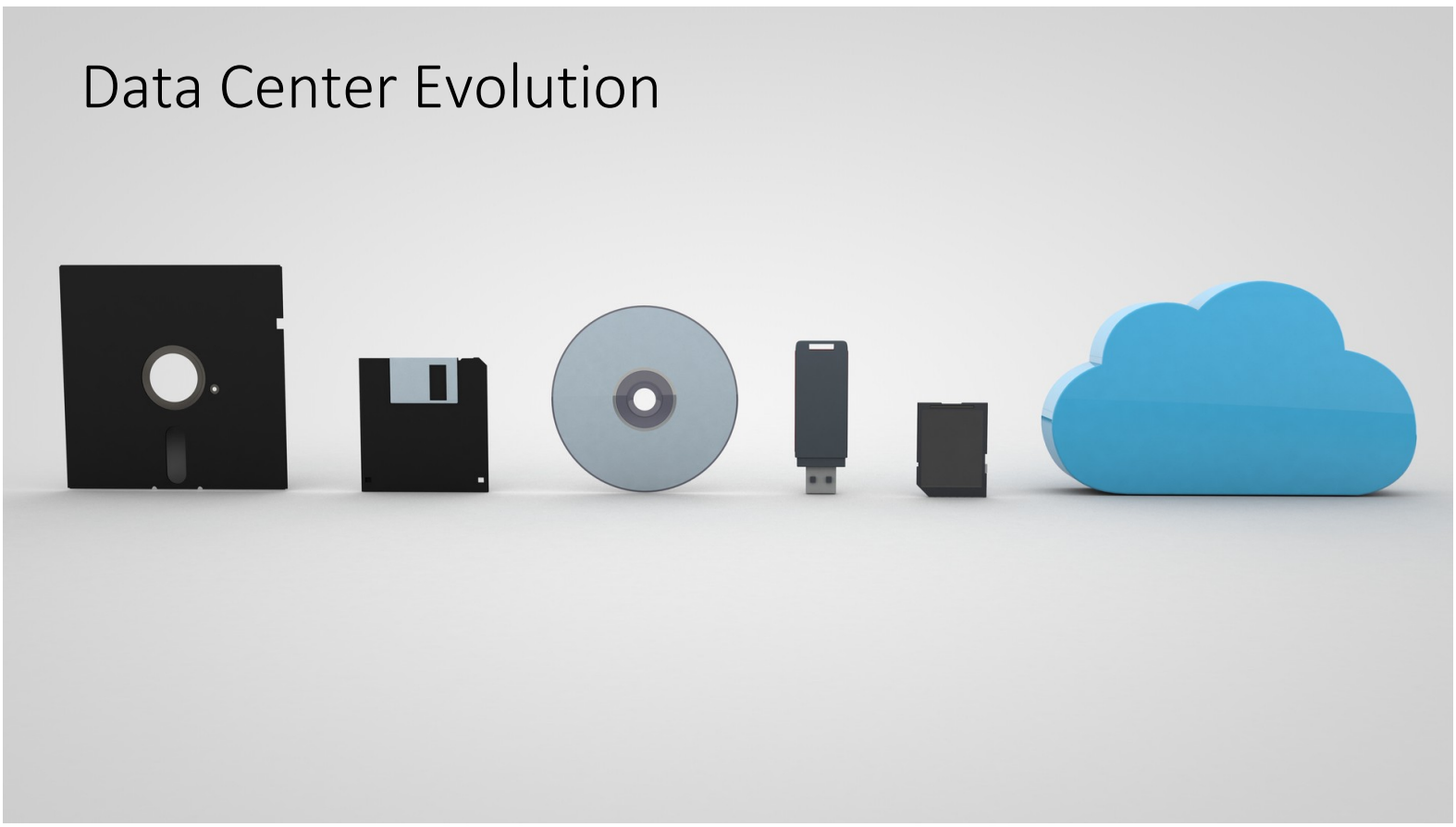
Image

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

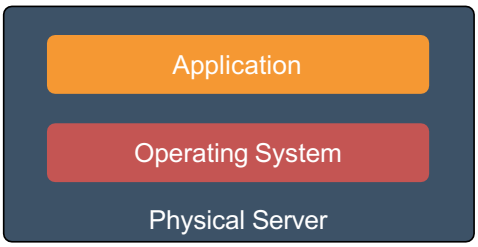
- Isolated application platform
- Contains everything needed to run your application
- Based on one or more images

Data Center Evolution



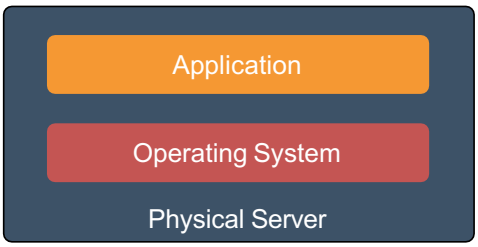
Monolithic

Monolithic Server Architecture



One physical server, one or more applications

Monolithic Server Architecture

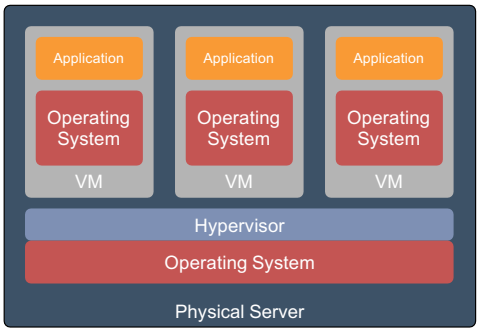


One physical server, one or more applications

Problems
- Slow deployment times
- Cost
- Wasted resources
- Difficult to scale
- Difficult to migrate
- Dependency Conflicts

Virtualized

Virtualized Infrastructure

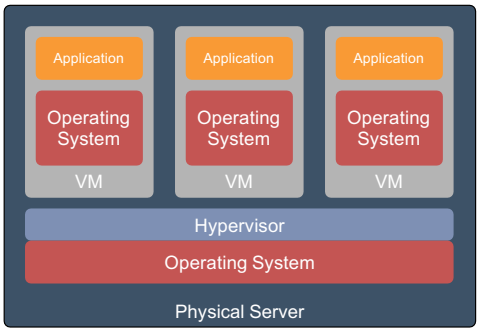


One physical server, multiple applications

Discussion

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

Virtualized Infrastructure - Advantages

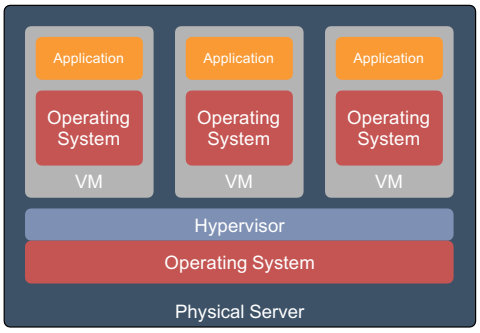


One physical server, multiple applications

Advantages

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

Virtualized Infrastructure - Limitations

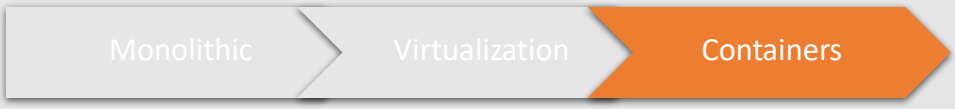


One physical server, multiple applications

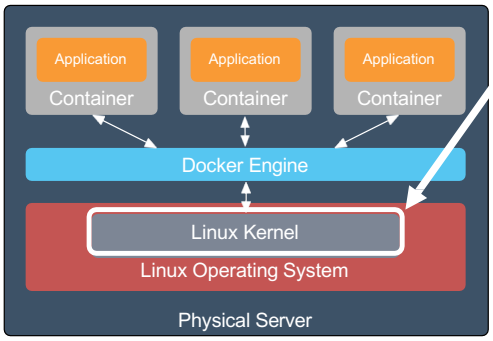
Limitations

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

Containers

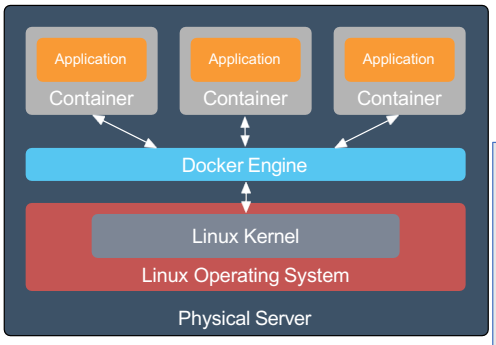


Containers



Shared kernel on the host to run multiple guest applications

Containers - Advantages

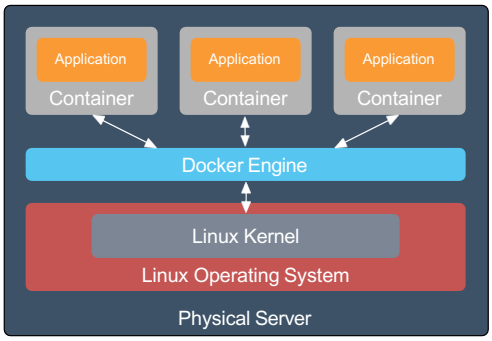


Shared kernel on the host to run multiple guest applications

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

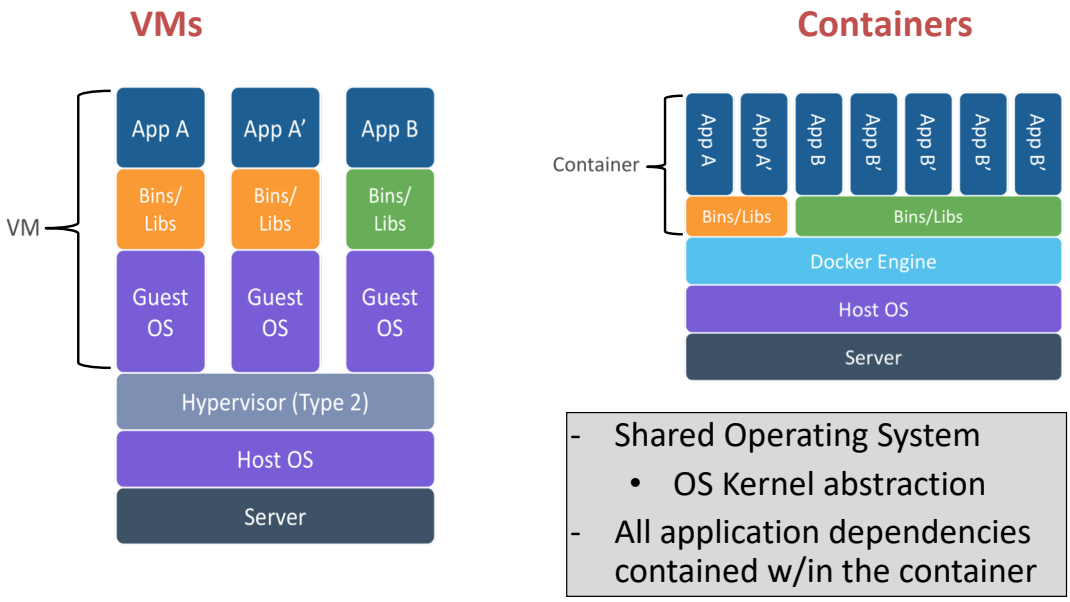


Shared kernel on the host to run multiple guest applications

Container Challenges

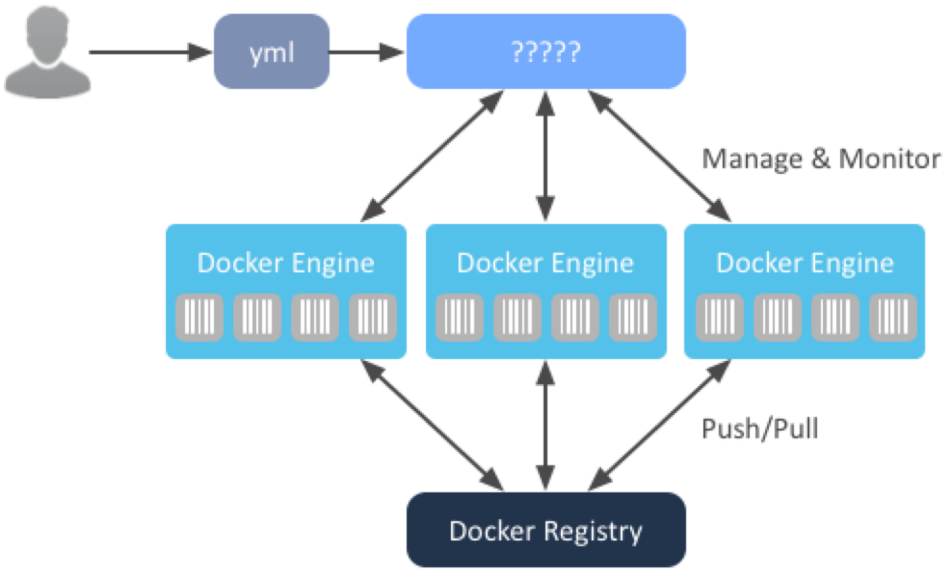
- Early Docker focused on single-node operations
- Up to user to cluster Docker hosts and manage deployment of containers on cluster
- User solves for automatic scale out of applications
- User solves for service discovery between application components (microservices)

Virtual Machines vs. Containers



- Shared Operating System
  - OS Kernel abstraction
- All application dependencies contained w/in the container

End Goal: Manage Cluster as Single Pool of Resources



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

Container based virtualization

High Level – Lightweight VM

- Own:
  - Process Space
  - Network Interface
  - Filesystems

- Can:
  - Run cmds as root
  - Have its own /sbin/init (different from host)

Container based virtualization

High Level – Lightweight VM

- Own:
  - Process Space
  - Network Interface
- Can:
  - Run cmds as root
  - Have its own /sbin/init (different from host)

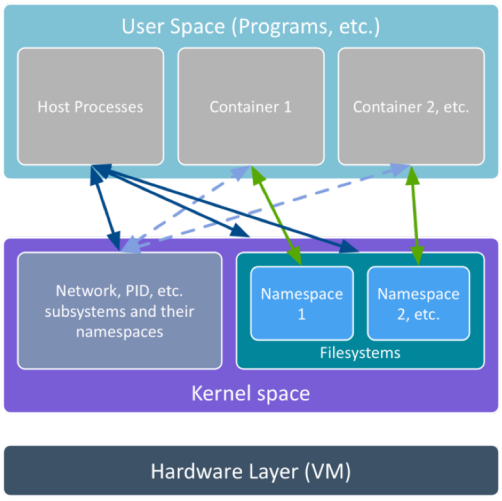
Low Level – chroot expanded

- Container = isolated processes
- Share kernel with host
- No device emulation

Isolation with Namespaces

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

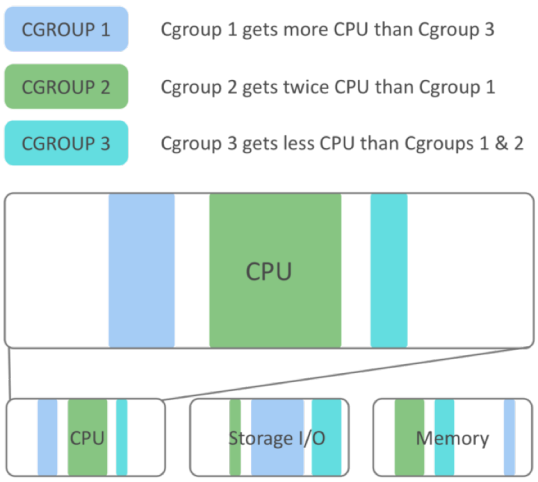
- Namespace wrap a global system resource in an abstraction layer
- Processes running in that namespace think they have their own, isolated resource
- Isolation includes:
  - Network stack
  - Process space
  - Filesystem mount points
  - etc.



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

Micro-Services

- 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

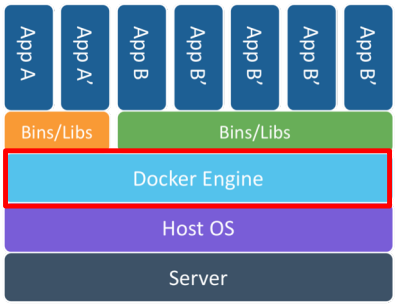
The Docker Platform Components

Docker Engine

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

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

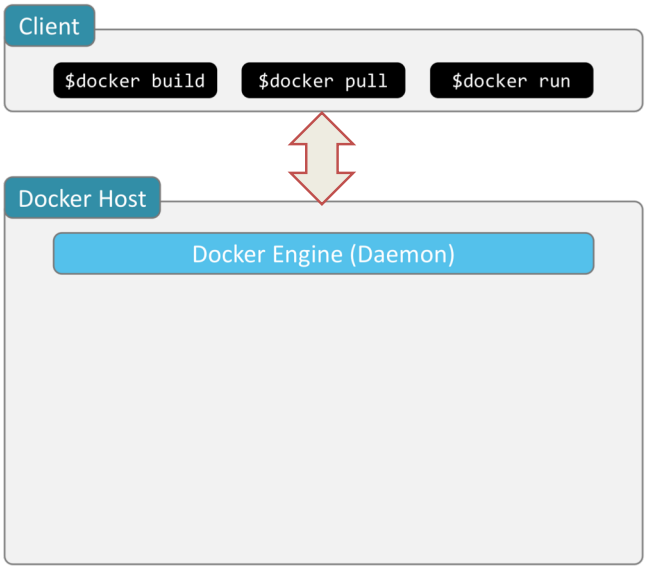
- Also known as **Docker Daemon**
- Uses Linux Kernel namespaces and control groups
  - **Linux Kernel (>= 3.10)**
- Namespaces provide an isolated workspace



Docker Engine

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

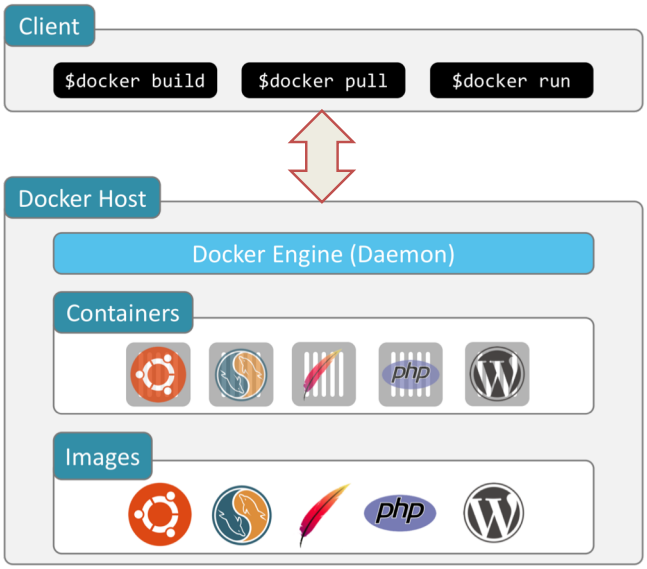
- The Docker Client is the `docker` binary
  - Primary interface to the Docker Host
- Accepts commands and communicates with the Docker Engine (Daemon)



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

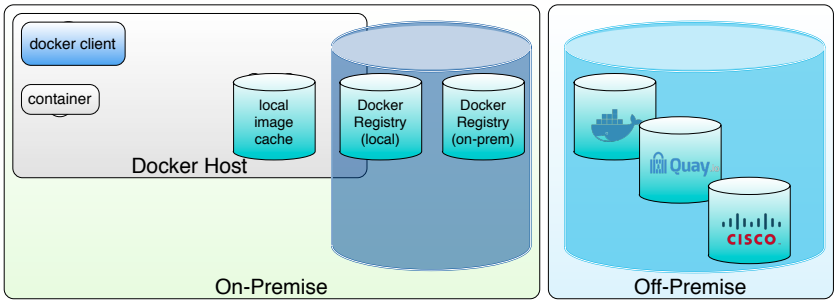
- Docker Engine **Pushes** Images to a Registry
- Version Control
- Docker Engine **Pulls** Images to Run

Docker Registry

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

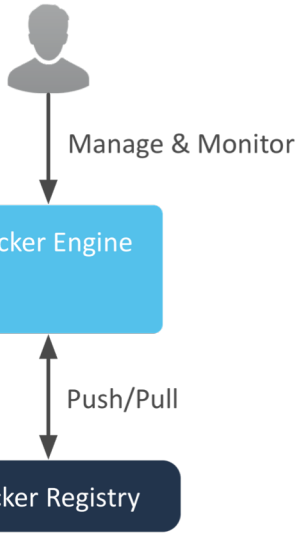
Types of Docker Registries

- 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



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

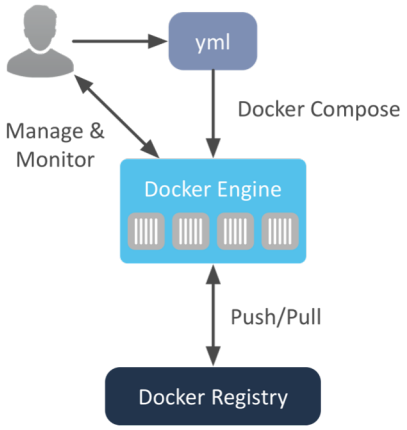
Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

Tool to create and manage multi-container applications

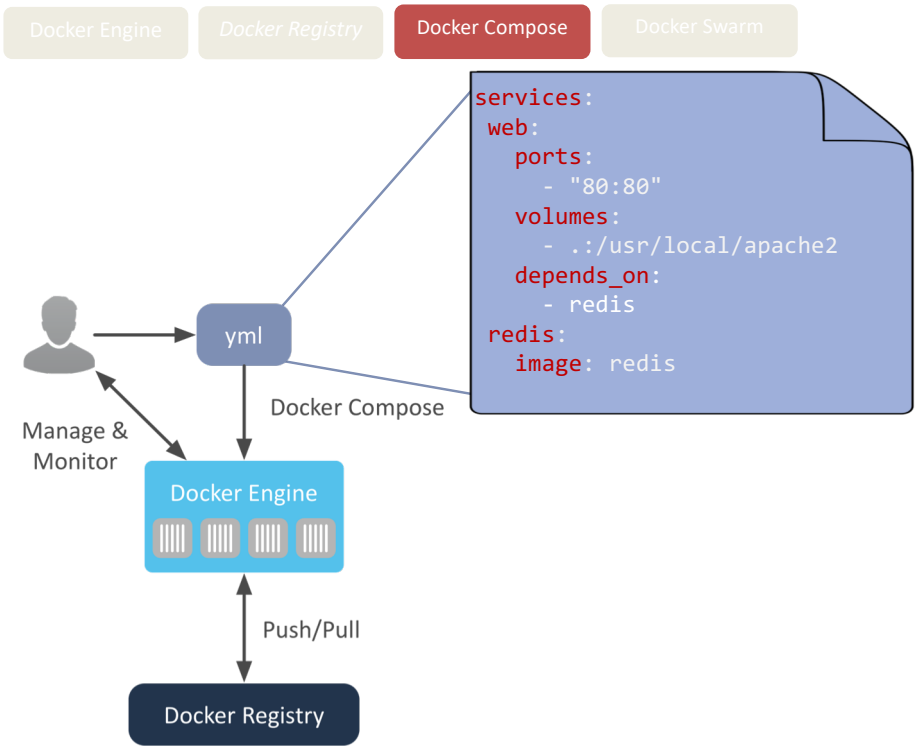
- Applications defined in a single file:

  docker-compose.yml

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



Docker Compose

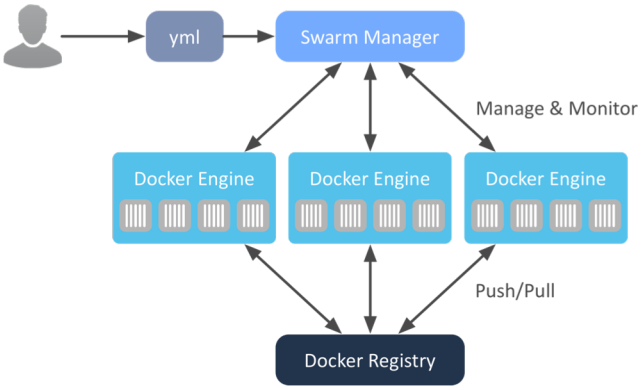


Docker Swarm

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

Clusters Docker hosts and schedules containers

- Native Clustering for Docker
  - Turn a pool of Docker hosts into a single, virtual host
  - Serves the standard Docker API



Create a Docker Hub Account

1. Navigate to: **http://hub.docker.com**

2. Select an ID & Password
   - This is YOUR PERSONAL account

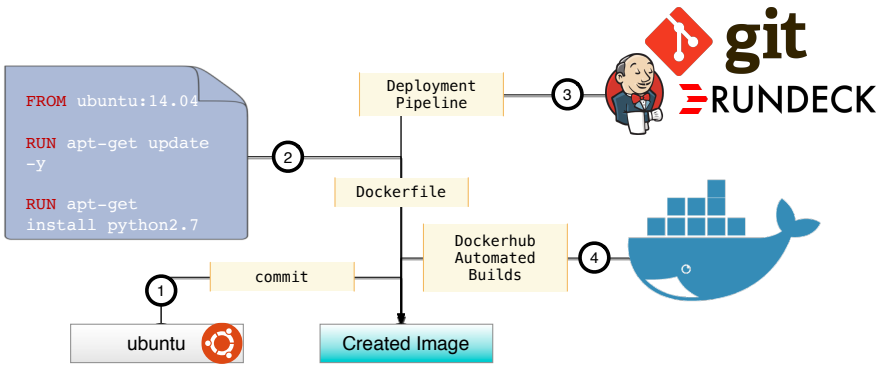
3. Confirm Your Email Address

Docker Images

Methods to Create an Image

Images can be created from:

1. The command-line
2. **An image from a Dockerfile**
3. An image from a deployment pipeline
4. An image from Docker automated builds



Anatomy of Docker Image

Docker Image Terminology

Image

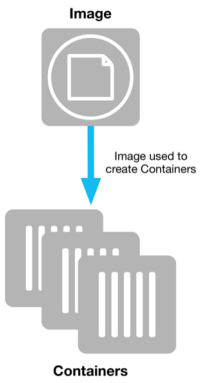
Union Filesystem

Dockerfile

Container

Hierarchy of files, with meta-data for how to create/run a container

- Read only template used to create containers
- Can be exported or modified to new images
- Created manually or through automated processes
- Stored in a Registry (Docker Hub, Docker Trusted Registry, etc.)



Docker Image Terminology

Image

Union Filesystem

OverlayFS/UnionFS – Used by Docker to layer images

- Not a distributed File System

Dockerfile

Container

Docker Image Terminology

Image

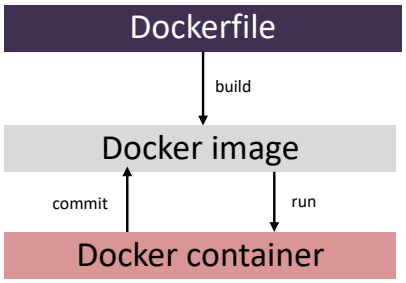
Union Filesystem

Dockerfile

Container

Configuration File (script) for creating images. Defines:

- Existing image to be the starting point
- Set of instructions to augment that image (each of which results in a new layer of the file system)
- Meta-data such as ports exposed
- The command to execute when the image is run



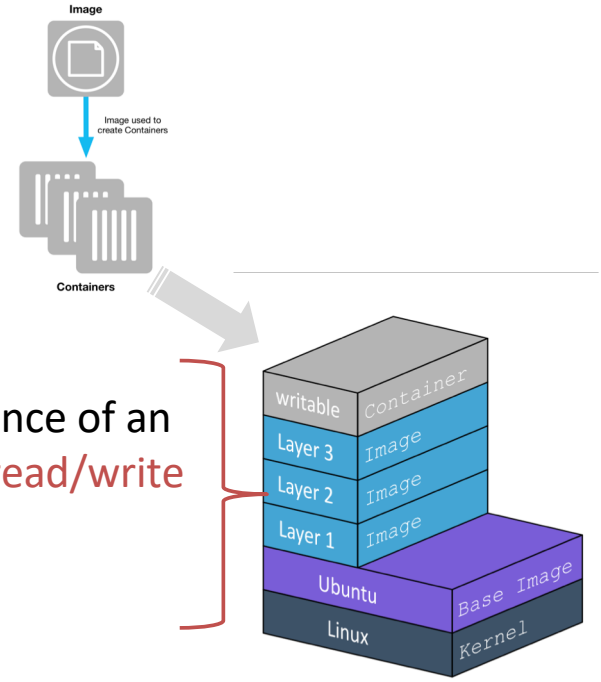
Docker Image Terminology

Image

Union Filesystem

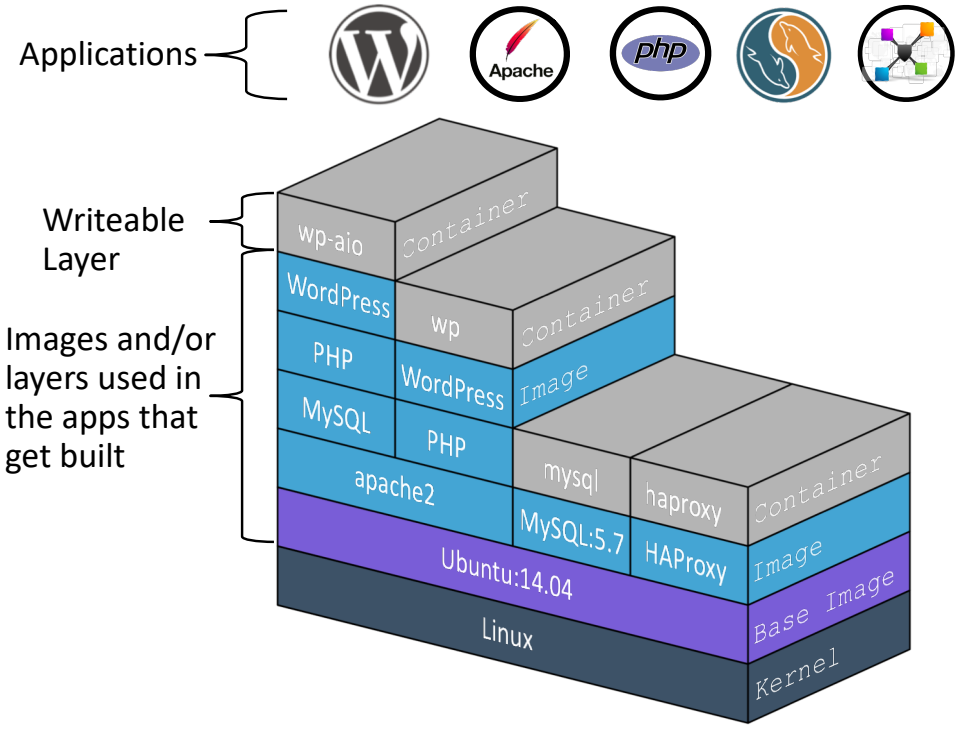
Dockerfile

Container



Runtime instance of an image plus a read/write layer

Applications, Containers, and Images



Union File System

Image/Container Interactions

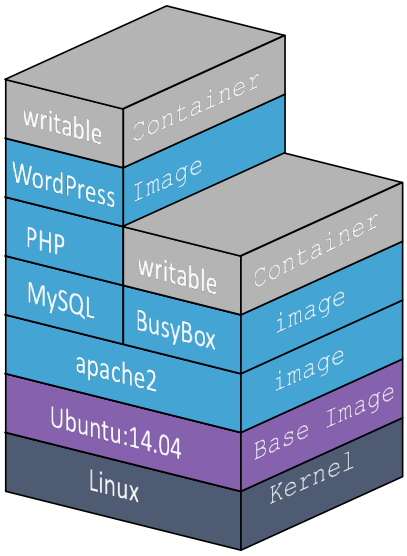
Pull an image from a Docker Registry

Run a container from an image

Add a file to a running container – Example, the image requires an additional file called index.html

Change to a running container – Example, the image requires an update of the index.php file

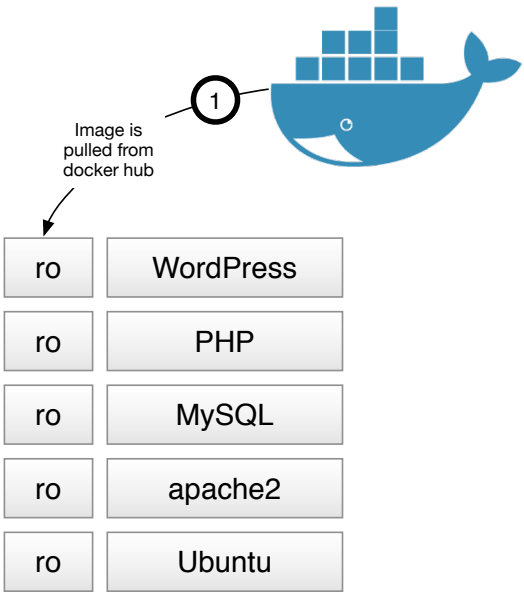
Delete files from a running container – All containers share the same host kernel



Pull the Docker image

The WordPress All-In-One image is pulled from Docker Hub

```
$docker inspect wordpress-aio
```



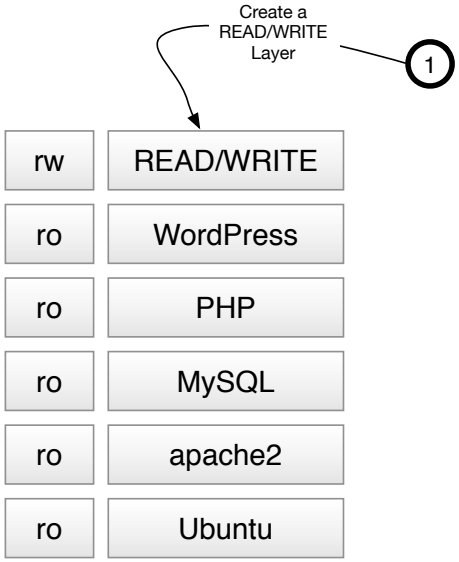
Docker images are:

- Pulled or created
- Stored in a local image repo
- READ-ONLY
- The basis of all containers

Run the Docker container

A container is made from the previously pulled WordPress AIO image

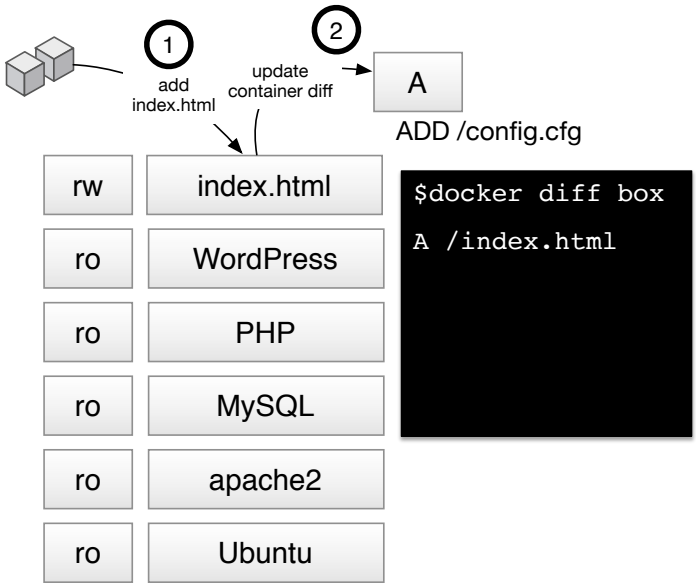
1. `$docker run wordpress-aio`
2. Container created
3. READ/WRITE layer created
4. All **ADD**, **CHANGE**, **DELETE** are committed to the READ/WRITE Layer



Add a file to Docker container

index.html is **created** on the WordPress container

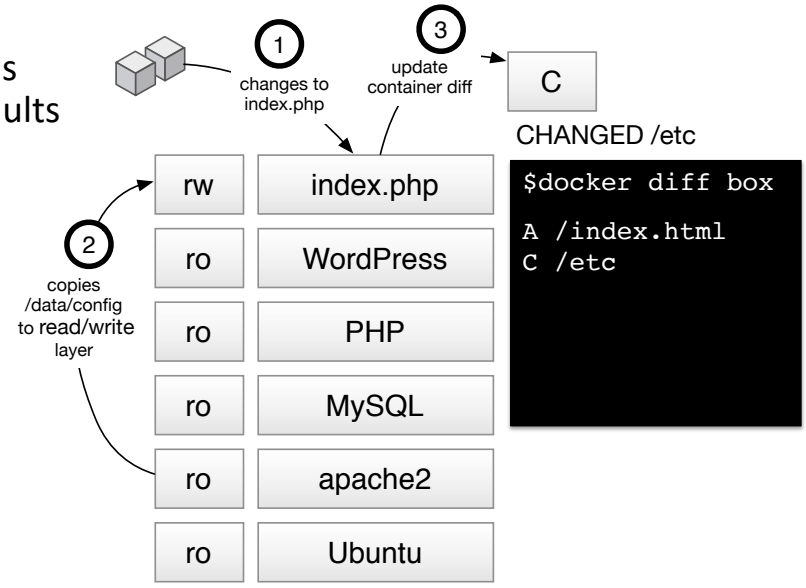
1. A file named index.html is created
2. The created file is written to the READ/WRITE layer



Change a file in Docker container

The index.php file is **updated** from defaults

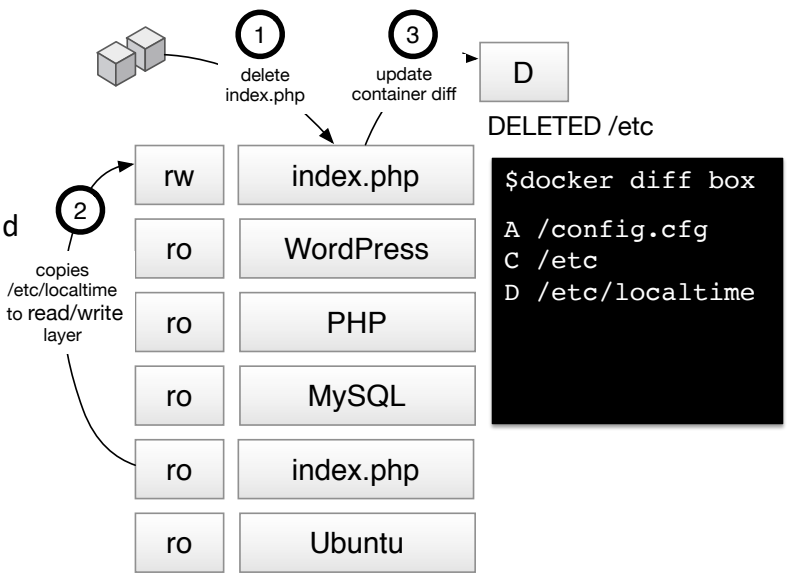
1. The file is edited
2. copies the file from READ-ONLY layer to READ/WRITE layer
3. update diff



Delete a file on Docker container

The localtime file is **deleted**

1. The file is deleted
2. copy data from READ-ONLY layer to READ/WRITE layer and marked as DELETED
3. update diff



UnionFS ADD, CHANGE, DELETE

A **ADD** – Add the data blocks to the READ/WRITE layer

C **CHANGE –** Copies the data blocks from the READ ONLY layer to the READ/WRITE layer and makes changes. Writes are committed to the READ/WRITE layer

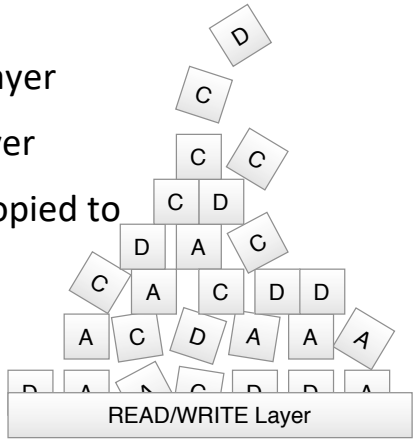
D **DELETE** – Copies the data blocks from the READ ONLY layers to the READ/WRITE layer and makes it as DELETED

Copy-On-Write

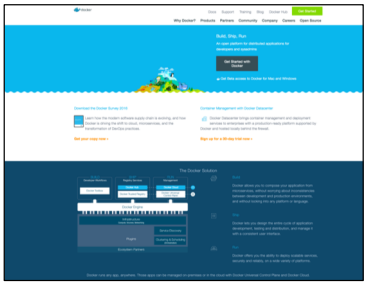
- Only when changes are made to the R/W UnionFS is data written
- Sometimes referred to as “Change Block Tracking” or “Copy-On-Change”

Key Takeaways

- **Images** are:
  - Highly portable and readily available
  - Reusable for many deployments
- **ADD** – ADD DATA to READ/WRITE Layer
- **CHANGE –** ADD DATA to READ/WRITE Layer
- **DELETE –** ADD DATA to READ/WRITE Layer
- **ALL** ADDs, CHANGEs, and DELETEs are copied to the READ/WRITE Layer



Docker Resources



Homepage

https://www.docker.com

Documentation

https://docs.docker.com

Freenode IRC

#docker

Stack Overflow

stackoverflow.com

Forums

forums.docker.com

Twitter

@docker

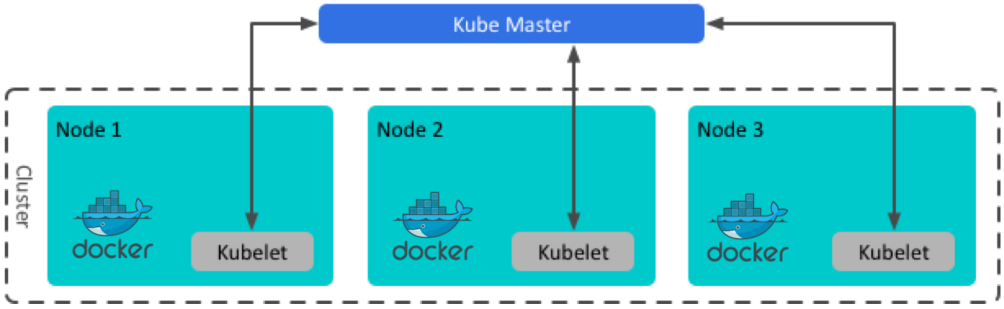

Questions

Lab: Docker Images

Kubernetes – Containers at scale

Kubernetes: Containers at Scale

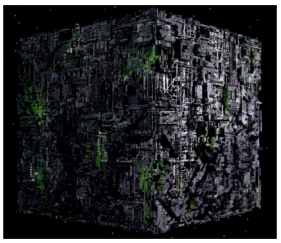
Kubernetes provides the infrastructure for container-centric deployment and operation of applications



- 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

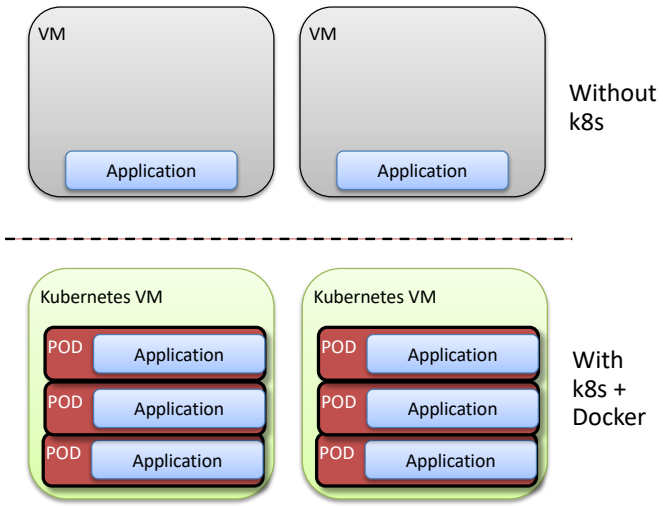
Original project name:

Seven of Nine

(the friendly Borg)

Kubernetes: Google Cloud Platform

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



Kubernetes today – CNCF Open Source Project

- Cloud Native Computing Foundation (CNCF) founded in 2015 as Linux Foundation Collaborative Project
- Kubernetes contributed by Google to CNCF at same time, still the flagship project
  - One of the most active projects on GitHub
- CNCF membership includes growing range of major industry vendors, including Cisco
- CNCF also governs the major container engine projects, thanks to recent donations
  - containerd (core runtime of Docker Engine)
  - rkt

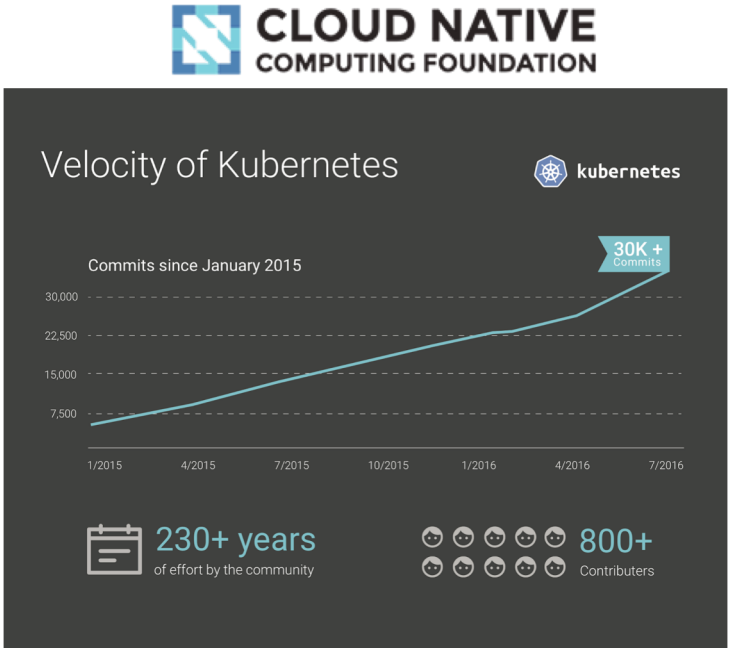


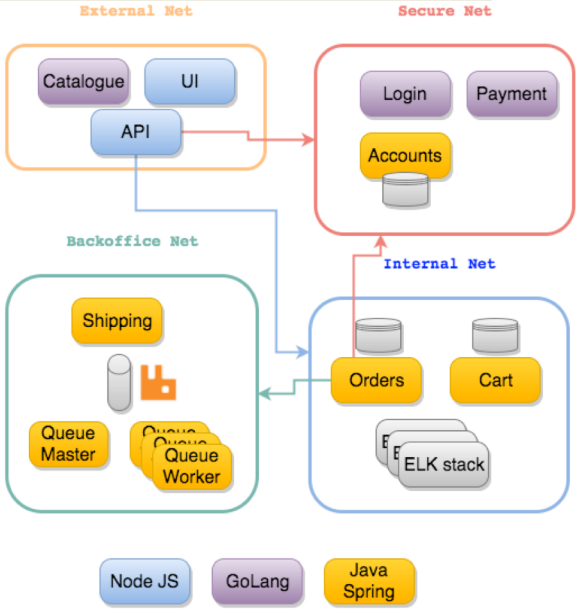
Image Source: https://cloudplatform.googleblog.com/2016/07/from-Google-to-the-world-the-Kubernetes-origin-story.html

Kubernetes Use Cases

Kubernetes:Cloud-Native Application Deployment

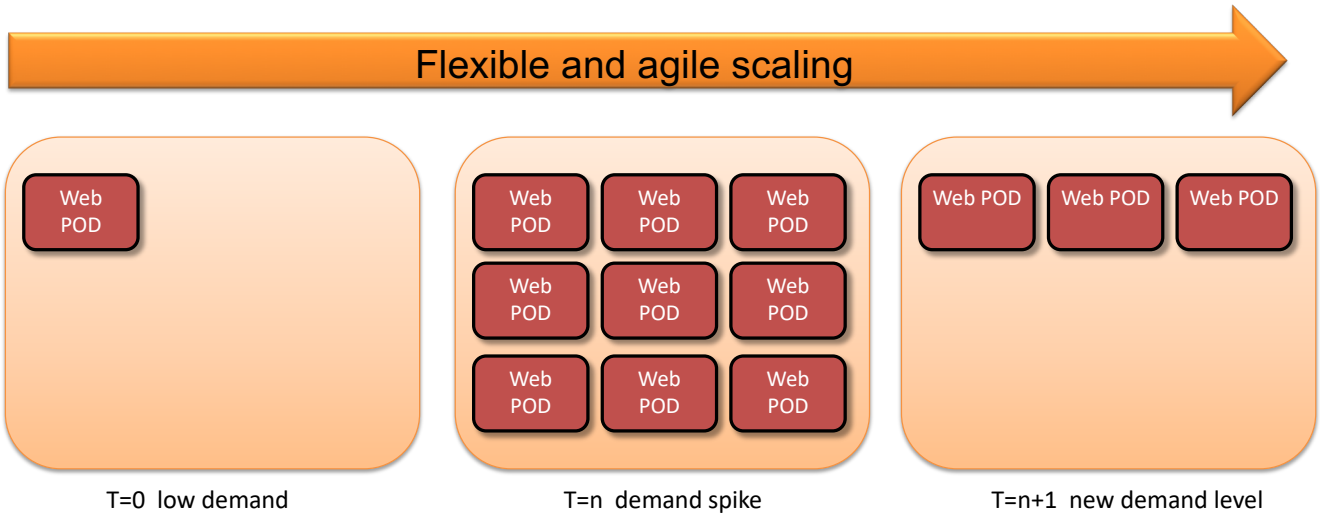
Cloud-native applications, aka microservice-based or 12-factor apps

- 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

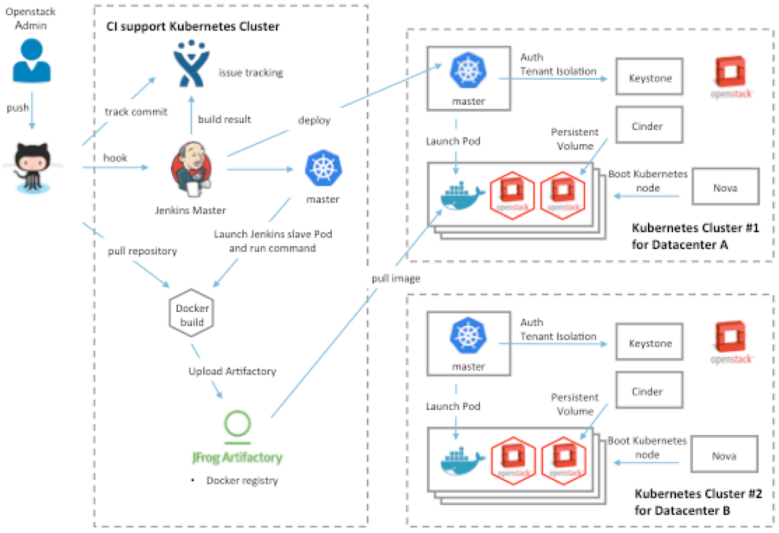
Managing execution environments in a CI/CD pipeline

- 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



Kubernetes: Platform as a Service (PaaS)

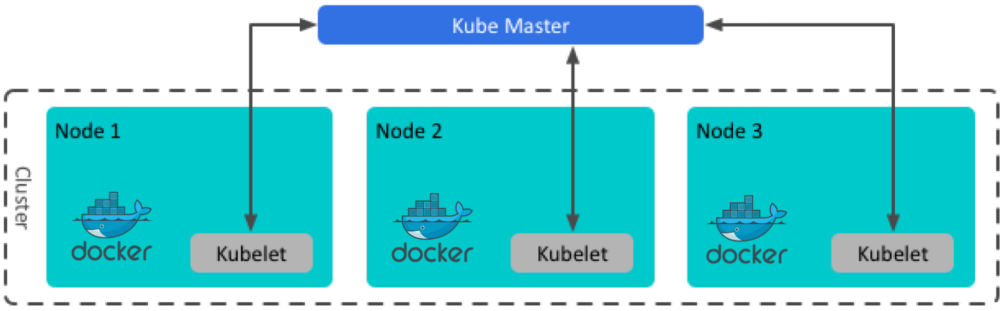
Many PaaS implementations run on Kubernetes

- Kubernetes control plane built on open API – components like controllers and schedulers are pluggable and extensible
- Vendors such as Red Hat and Deis* build more advanced features and services on top of Kubernetes infrastructure

Overview of Kubernetes Clusters

Kubernetes Clusters

- 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: Master

Main Components:

Master Node:

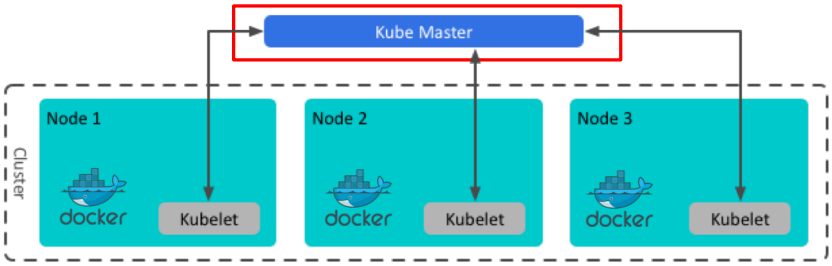
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

Worker Nodes:

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

Clusters:

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



Kubernetes Components: Nodes

Main Components:

Master Node:

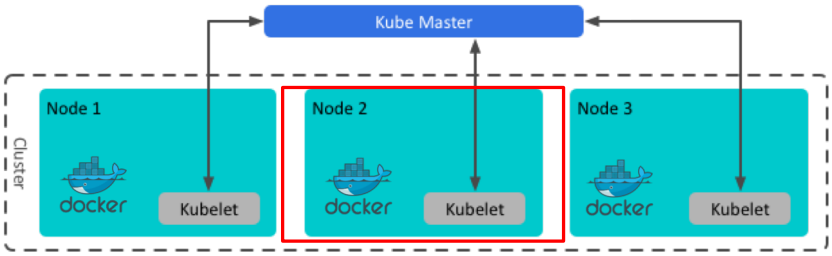
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

Worker Nodes:

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

Clusters:

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



Kubernetes Cluster Components

Main Components:

Master Node:

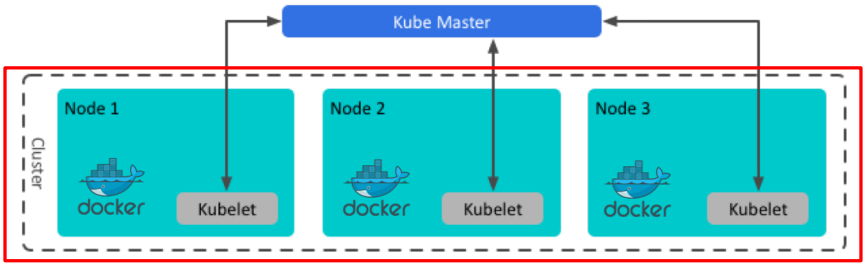
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

Worker Nodes:

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

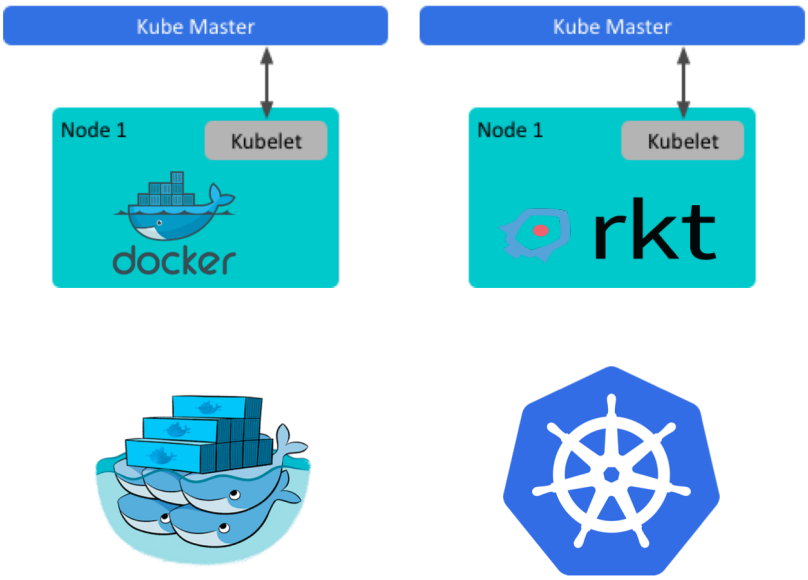
Clusters:

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



Kubernetes and Docker

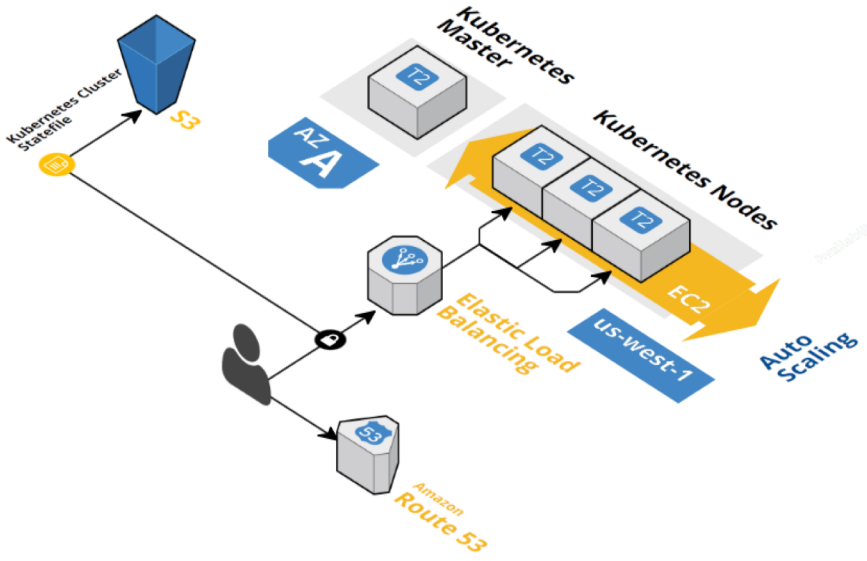
- Kubernetes does not include its own Linux container runtime
- Most deployments use Docker as the container engine, currently
- Since version 1.3, Kubernetes also supports rkt, the CoreOS engine for containers
- Co-opetition between K8s and Docker
  - Docker Swarm is competing technology for clustering Docker hosts
  - Docker, Inc. folded Swarm into Docker Engine v. 1.12



Kubernetes AWS Lab Environment

Kubernetes Clusters for Today's Lab Explorations

- Lab clusters will be installed on AWS
- Each lab cluster will consist of
  - Kubernetes Master
  - 2x Kubernetes Nodes
- Lab environments will be deleted after the course concludes

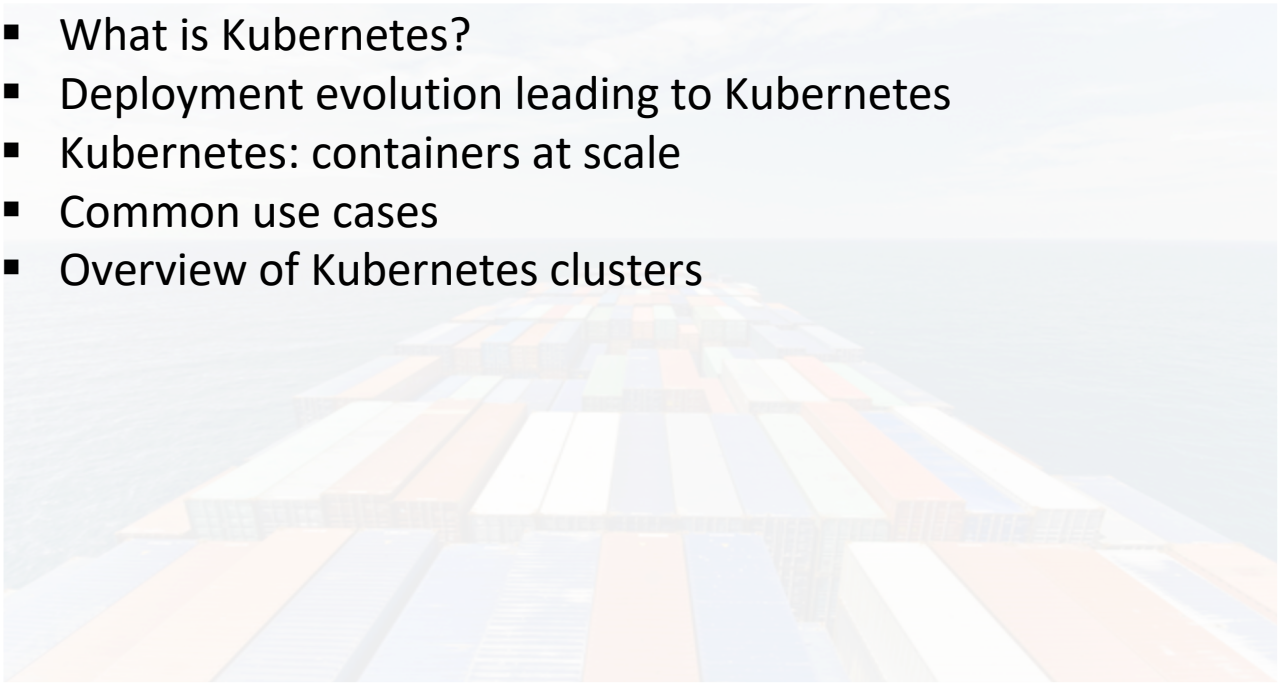


Lab: Install Kubernetes

Summary, Review, Q&A

Summary

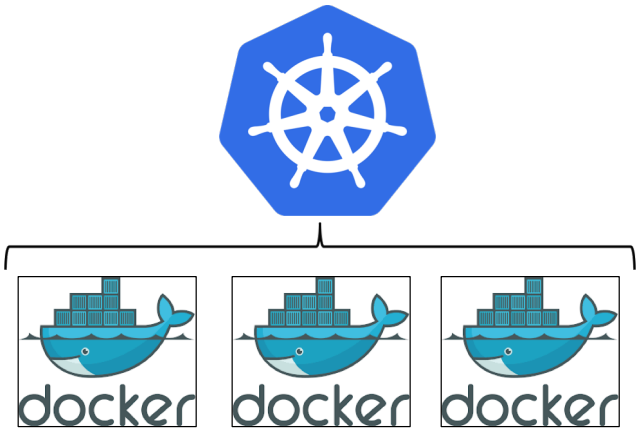
- What is Kubernetes?
- Deployment evolution leading to Kubernetes
- Kubernetes: containers at scale
- Common use cases
- Overview of Kubernetes clusters



Review: Introduction to Kubernetes

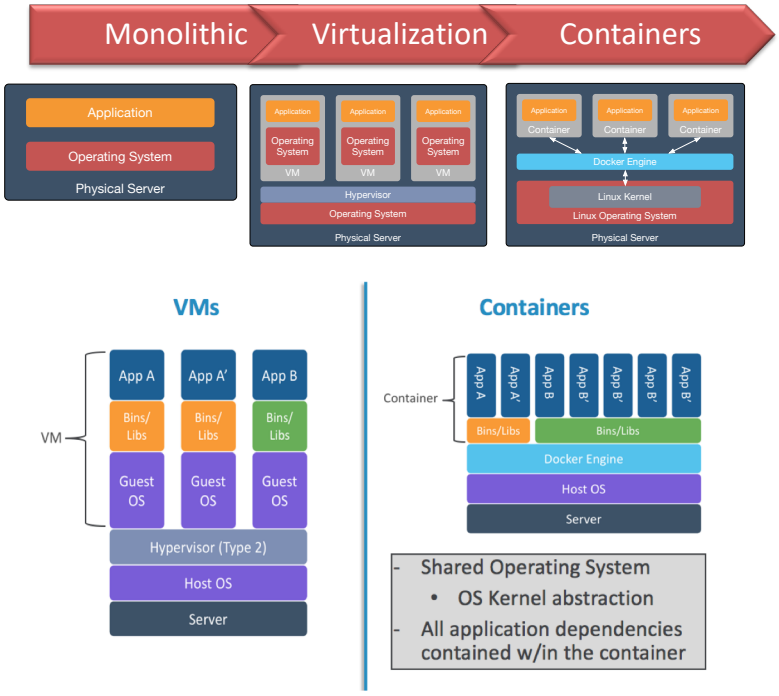
- What is Kubernetes?
- Deployment evolution leading to Kubernetes
- Kubernetes: containers at scale
- Common use cases
- Overview of Kubernetes clusters
- Minikube: Local Kubernetes cluster

- **Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.***



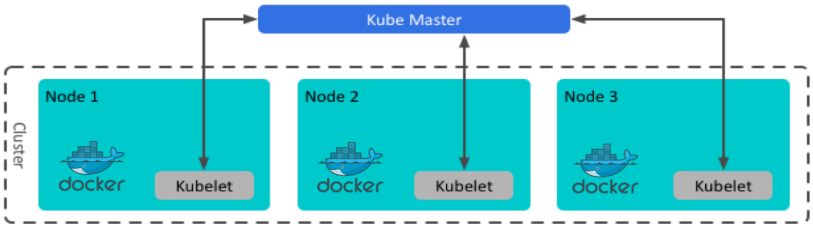
Review: What is Kubernetes?

- What is Kubernetes?
- Deployment evolution leading to Kubernetes
- Kubernetes: containers at scale
- Common use cases
- Overview of Kubernetes clusters
- Minikube: Local Kubernetes cluster



Review: Introduction to Kubernetes

- What is Kubernetes?
- Deployment evolution leading to Kubernetes
- Kubernetes: containers at scale
- Common use cases
- Overview of Kubernetes clusters
- Minikube: Local Kubernetes cluster

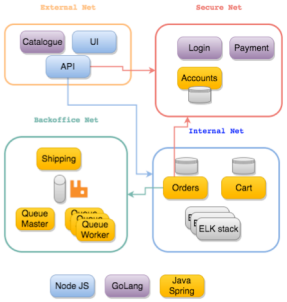


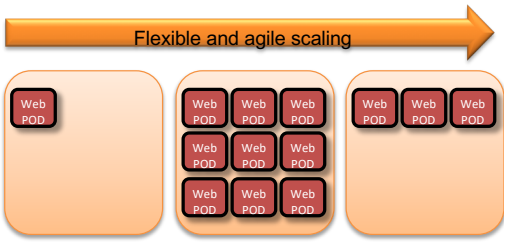
- Builds on 10+ years of engineering experience at Google
- Control plane manages efficient scheduling of containers across hosts
- Key application management features, including
  - App elasticity and self-healing
  - Naming and discovery
  - Request load balancing
  - Log access and resource monitoring

Review: Introduction to Kubernetes

- What is Kubernetes?
- Deployment evolution leading to Kubernetes
- Kubernetes: containers at scale
- Common use cases
- Overview of Kubernetes clusters
- Minikube: Local Kubernetes cluster

Cloud Native Application Deployment





Elastic Service Scaling

CI/CD Pipelines

RED HAT OPENSHIFT
DEIS
Eldarion Cloud
HASURA

PaaS Infrastructure

Review: Introduction to Kubernetes

- What is Kubernetes?
- Deployment evolution leading to Kubernetes
- Kubernetes: containers at scale
- Common use cases
- Overview of Kubernetes clusters
- Minikube: Local Kubernetes cluster

Cluster Components:

Master Node:

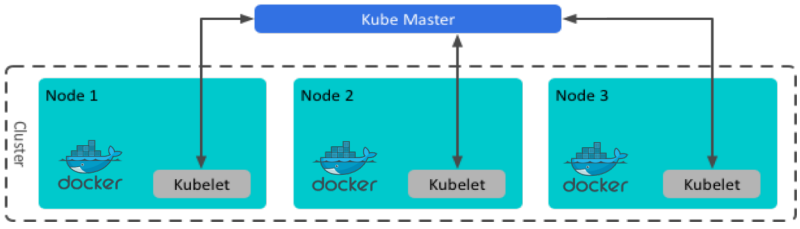
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

Worker Nodes:

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

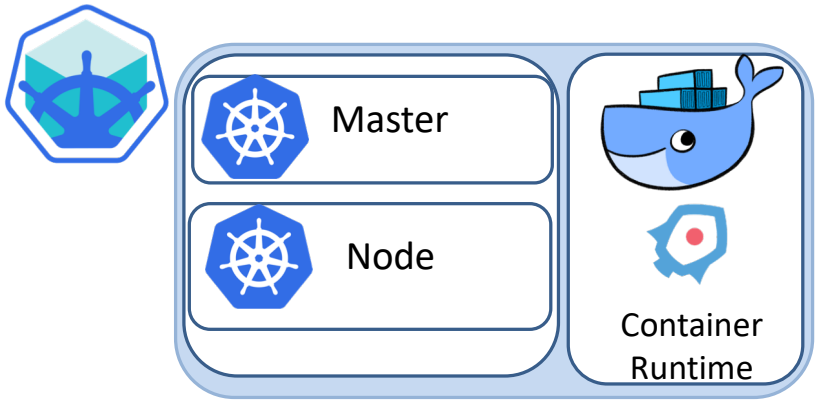
Clusters:

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



Review: Introduction to Kubernetes

- What is Kubernetes?
- Deployment evolution leading to Kubernetes
- Kubernetes: containers at scale
- Common use cases
- Overview of Kubernetes clusters
- Minikube: Local Kubernetes cluster

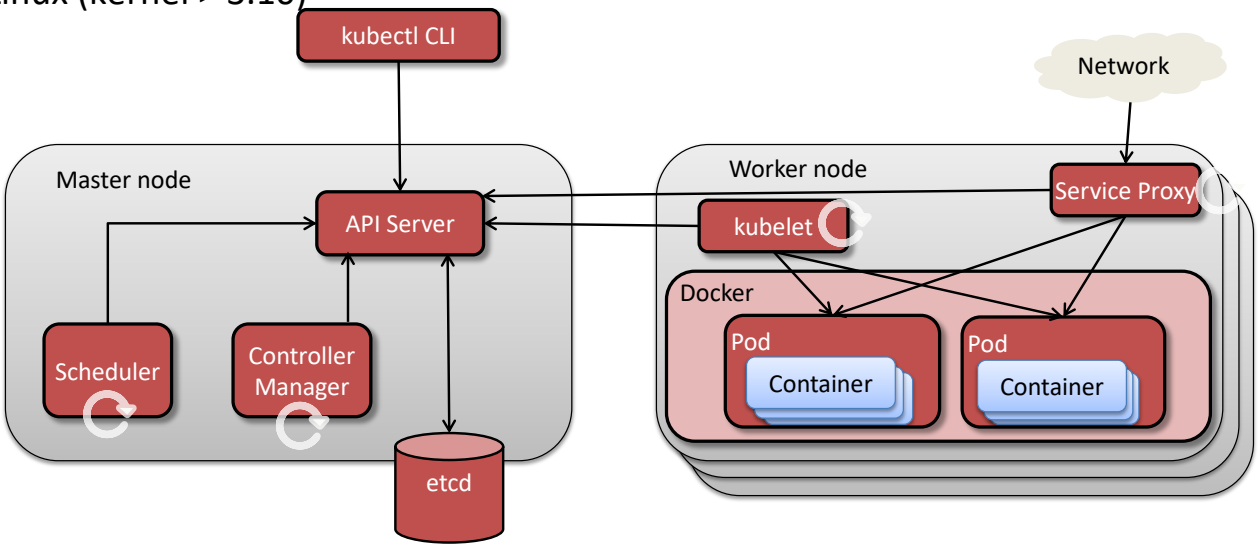



Questions

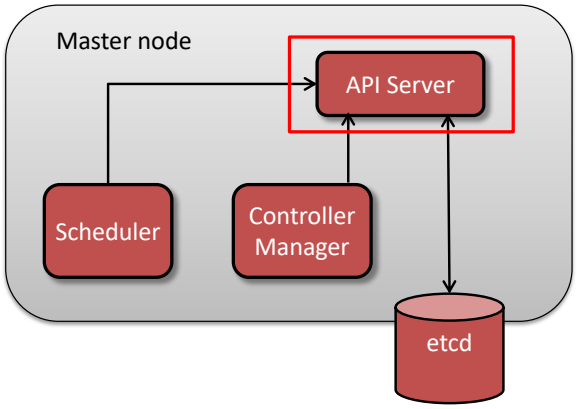
Kubernetes Architecture

Kubernetes Cluster Architecture

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



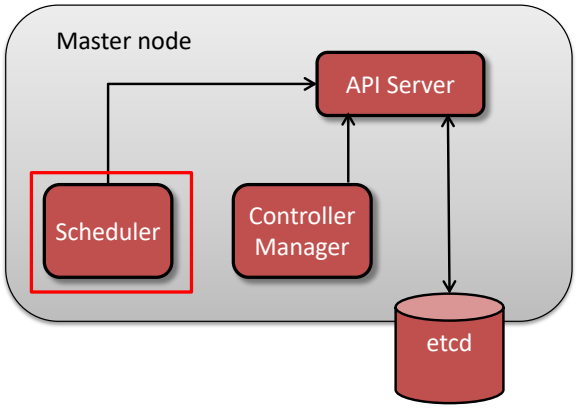
Kubernetes Master Node Components



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

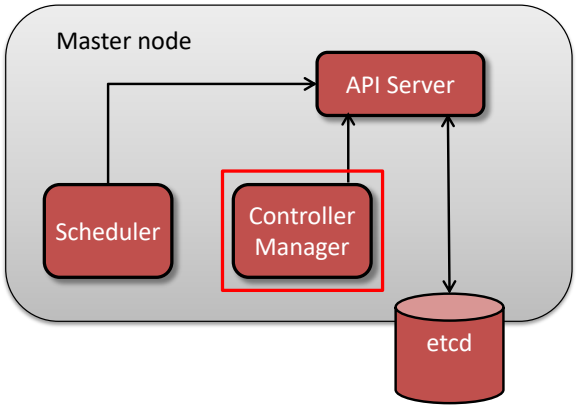
Kubernetes Master Node Components



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

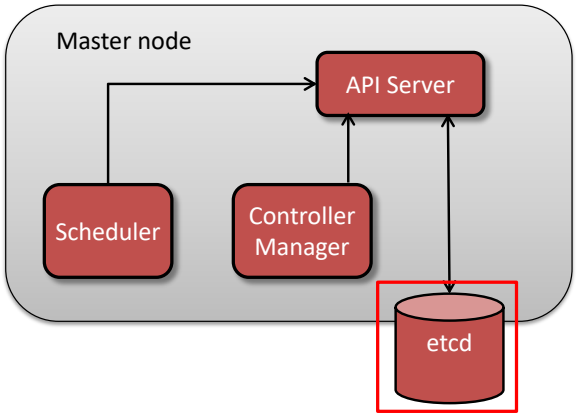
Kubernetes Master Node Components



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

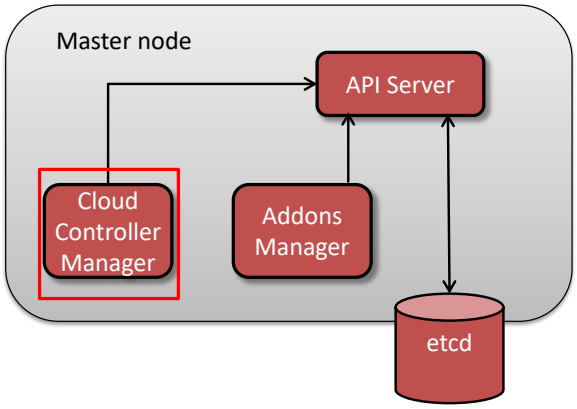
Kubernetes Master Node Components



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

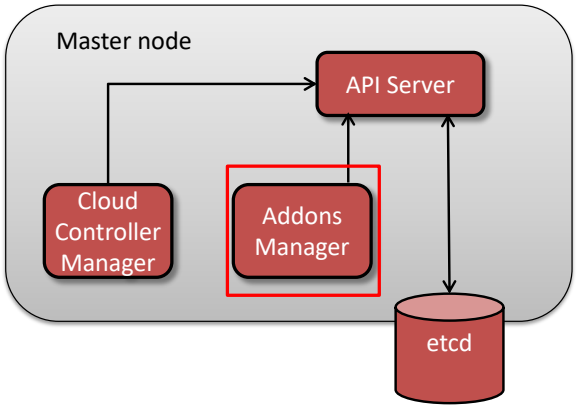
Master Node Additional Components



K8s components written in Go (golang.org)

- **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

Master Node Additional Components

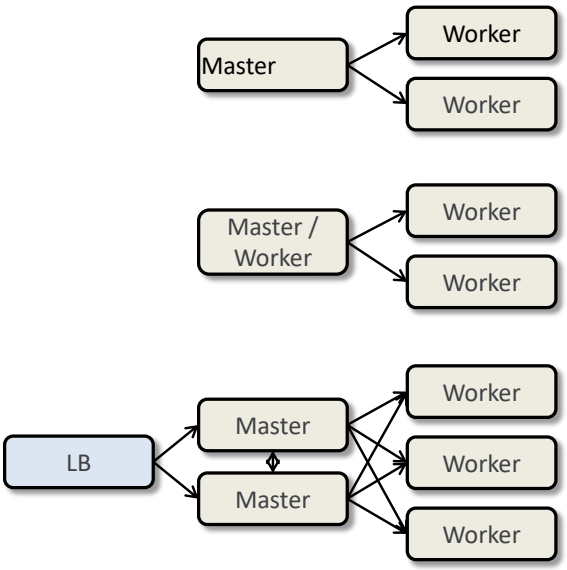


K8s components written in Go (golang.org)

- **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

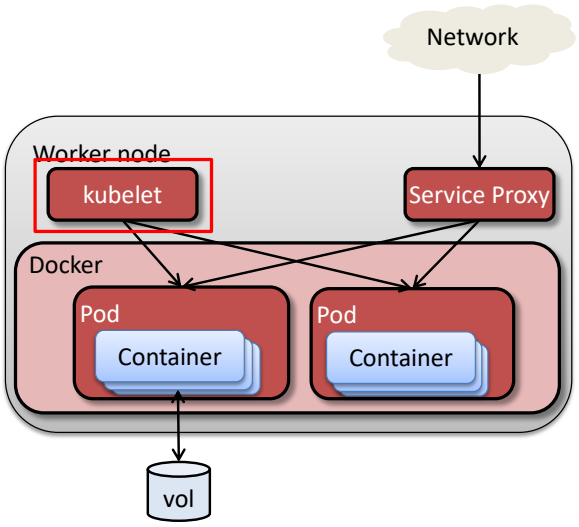
Kubernetes Master Node Deployment Options

- Simple cluster has a single master node, with single API/scheduler/controller
- At small scale, master may also be a worker node
- Can create a resilient control plane for K8s using a 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



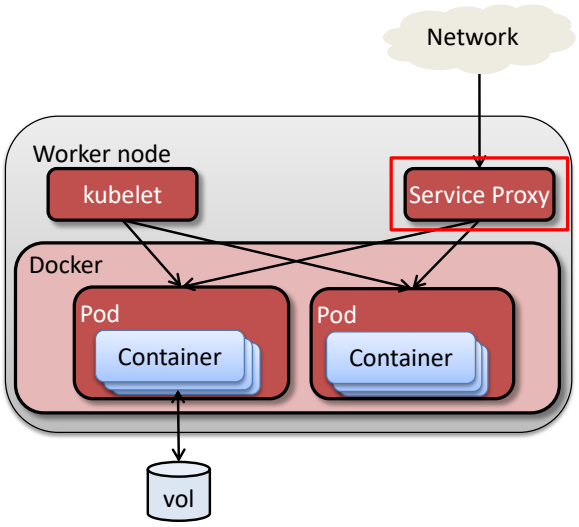
Kubernetes Worker Node Components

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



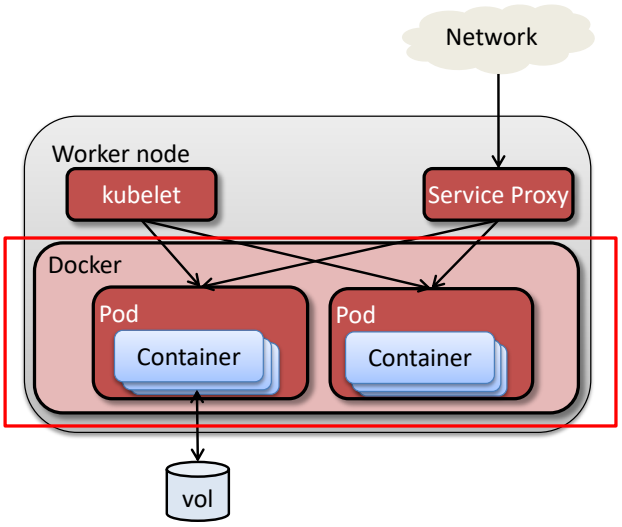
Kubernetes Worker Node Components

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



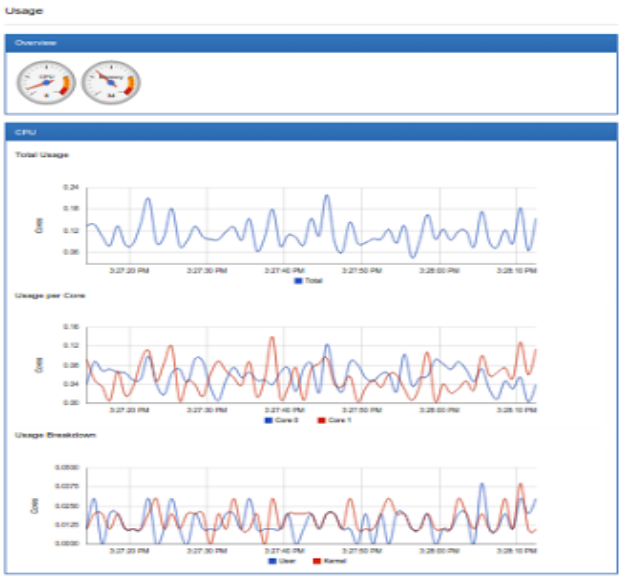
Kubernetes Worker Node Components

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



Kubernetes Nodes are Docker Nodes

- It's possible to connect to Docker on a Worker node and check statuses, e.g.
  - See cAdvisor performance graphs for the host, if cAdvisor is deployed
- The kubelet will only manager Docker containers created by K8s
- If you create containers directly in Docker, K8s will ignore them
  - Generally, all K8s system components themselves run as containers
  - May be two Docker daemons running under the covers

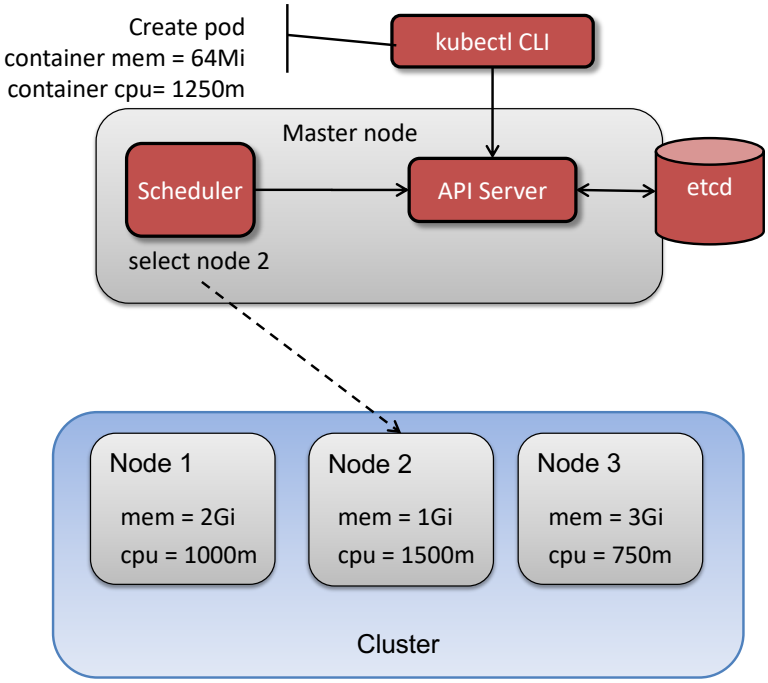


cAdvisor UI

Kubernetes Pod Scheduling

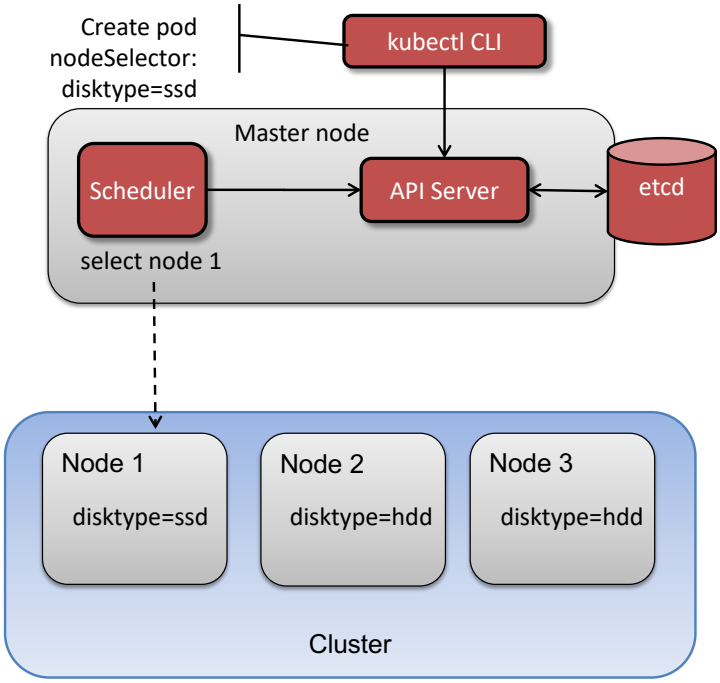
Pod Scheduling Overview

- kube-scheduler on the master node assigns new pod requests to appropriate worker nodes
- Default scheduler takes account of
  - Available node CPU/RAM
  - Resource requests from new pod – sum of resource requests of pod containers
- Default scheduler will automatically
  - Schedule pod on node with sufficient free resources
  - Spread pods across nodes in the cluster
- Can specify custom schedulers for pods



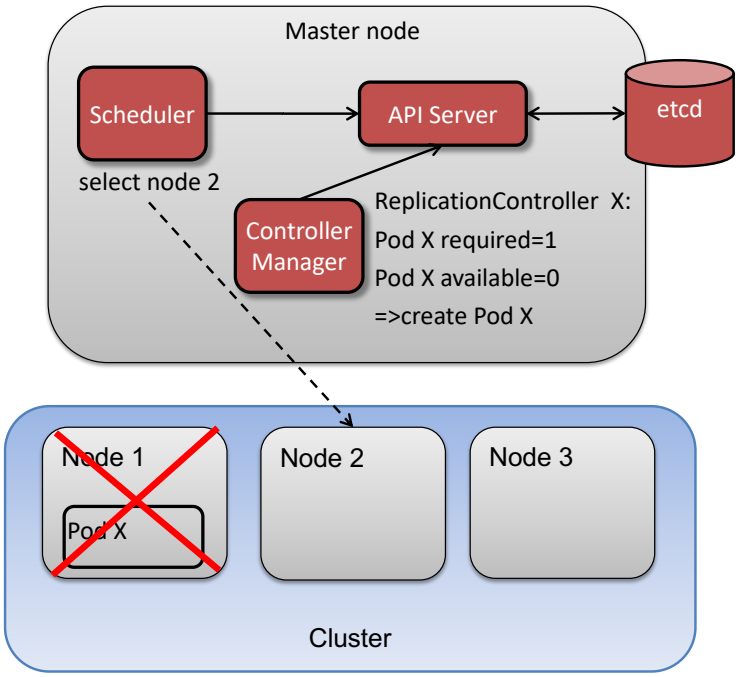
Pod Scheduling with Default Scheduler

- Nodes carry labels indicating topology and other resource notes
- Users can require pods to be scheduled on nodes with specific label(s) via a nodeSelector in container spec
- K8s 1.6 adds more advanced scheduling options, including
  - Node affinity/anti-affinity generalizing the nodeSelector feature
  - Node taints that prevent scheduling of pods to nodes unless pod 'tolerates' the taint
  - Pod affinity/anti-affinity to control relative placement of pods



Pod Re-creation Driven by Control Loops

- kube-scheduler performs same node selection operation when new pod created due to e.g. node loss
- kube-controller-manager runs controllers like ReplicationController managing number of pod instances available
- kube-controller-manager will initiate request for new pods as needed, which will be scheduled by kube-scheduler per pod/container spec



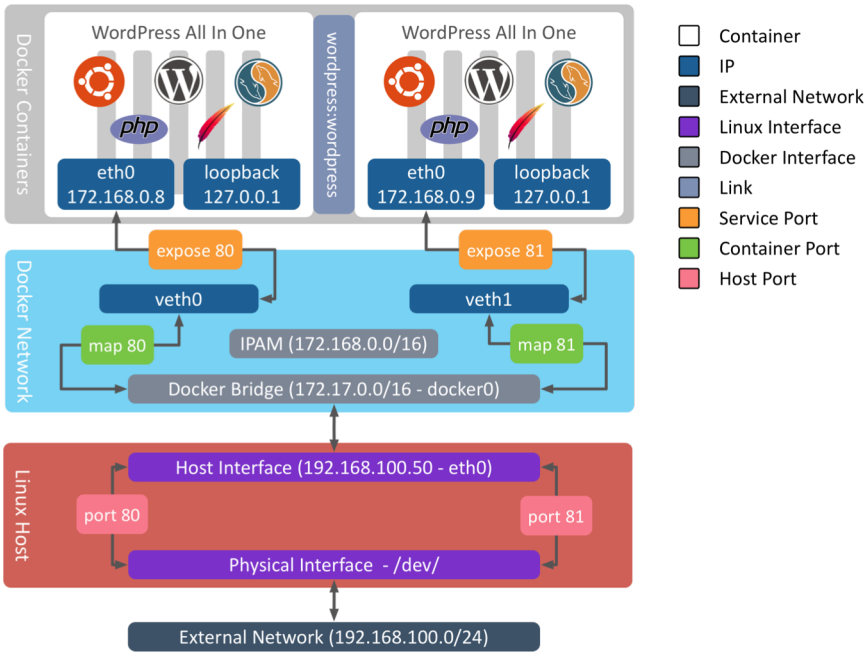

Questions

Lab: Commands & Namespaces

Kubernetes Network Models

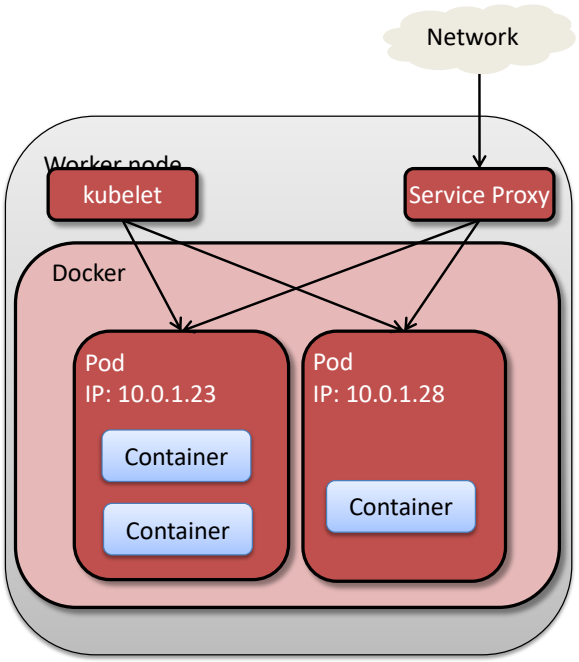
Review: Typical Container Networking in Docker

- Kubernetes approaches networking differently than Docker does by default
- Typical Docker uses host-private networking
  - Containers allocated veth interface on local Linux bridge
  - Container veth assigned IP from private subnet on bridge
  - Containers on same host can communicate directly via bridge
  - Communications between containers on different hosts requires host port assignment and forwarding



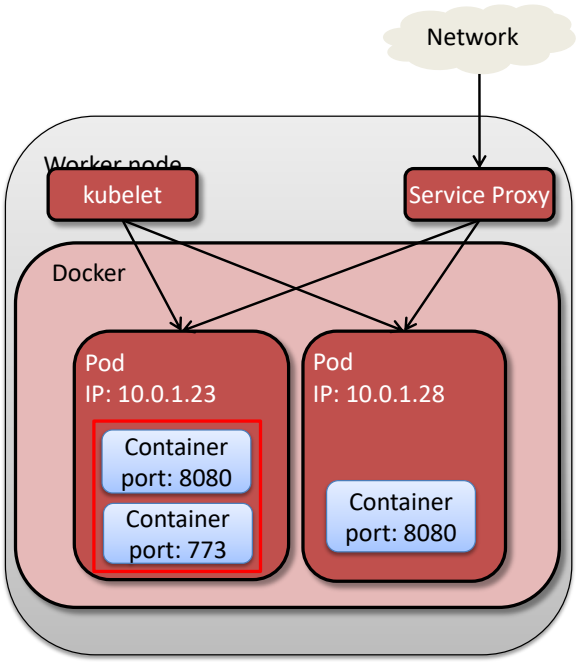
Networking in Kubernetes – One IP per Pod

- Kubernetes design goal is to simplify configuration – tracking lots of port mappings is hard!
- K8s networking principles:
  - All containers can communicate with other containers without NAT
  - All nodes can communicate with containers without NAT
  - Container sees its own IP exactly as other containers and nodes do
- K8s assigns IP to every **pod** in the cluster as part of flat shared network



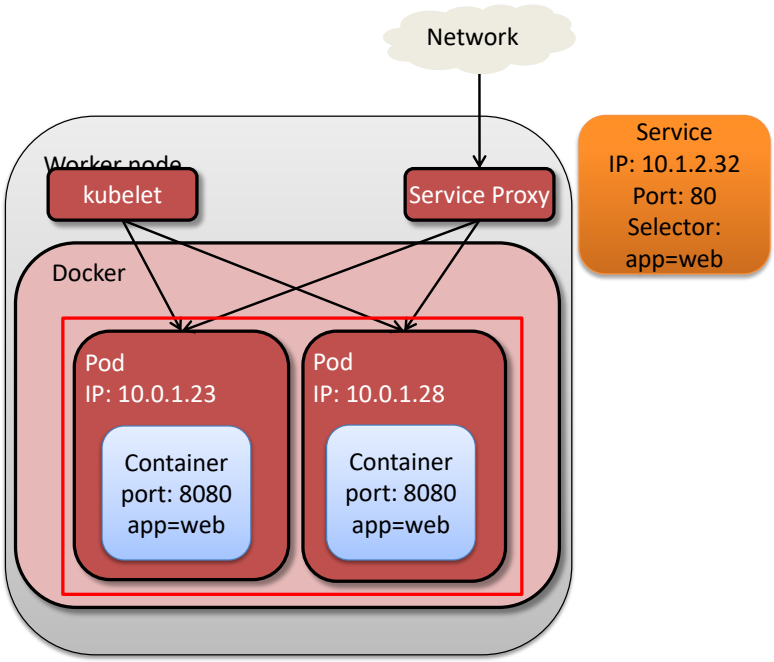
Container Communications – Same Pod

- Containers in the same pod share the same network namespaces and IP address
- Containers in same pod can reach each other's port on 'localhost'
- Need to manage ports used by co-located containers to avoid conflicts, but done in same pod spec
- Other containers in the cluster can connect to the pod's containers via their ports on the pod's IP

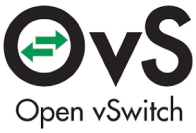


Abstraction Layer for Applications

- Pod IP's are simple but tricky to track across replicas, restarted pods, etc.
- Service objects permit allocation of a stable IP that forwards to multiple pods matching the service's selector
- kube-proxy maintains iptables rules for service objects on each worker node
  - NATs requests to a backend pod
  - Maps each service port to backend pod port



Network Implementation Options for Kubernetes

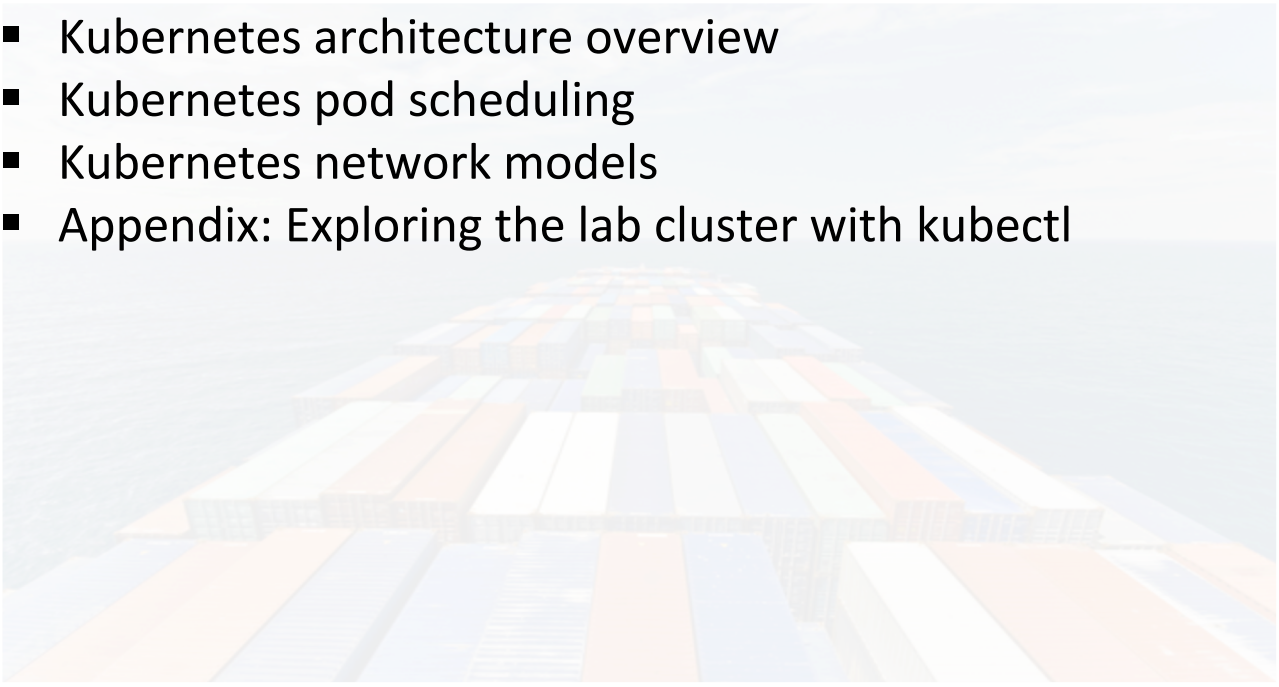


- Multiple options for implementation of networking to Kubernetes model
- Open source options include
  - Contiv (Cisco)
  - OpenContrail (Juniper)
  - Flannel (CoreOS)
  - L2 / Linux Bridge
  - Open vSwitch & OVN
  - Project Calico (Tigera)
  - Weave Net (Weaveworks)
- GKE (Google) has their own implementation

Summary, Review, and Q&A

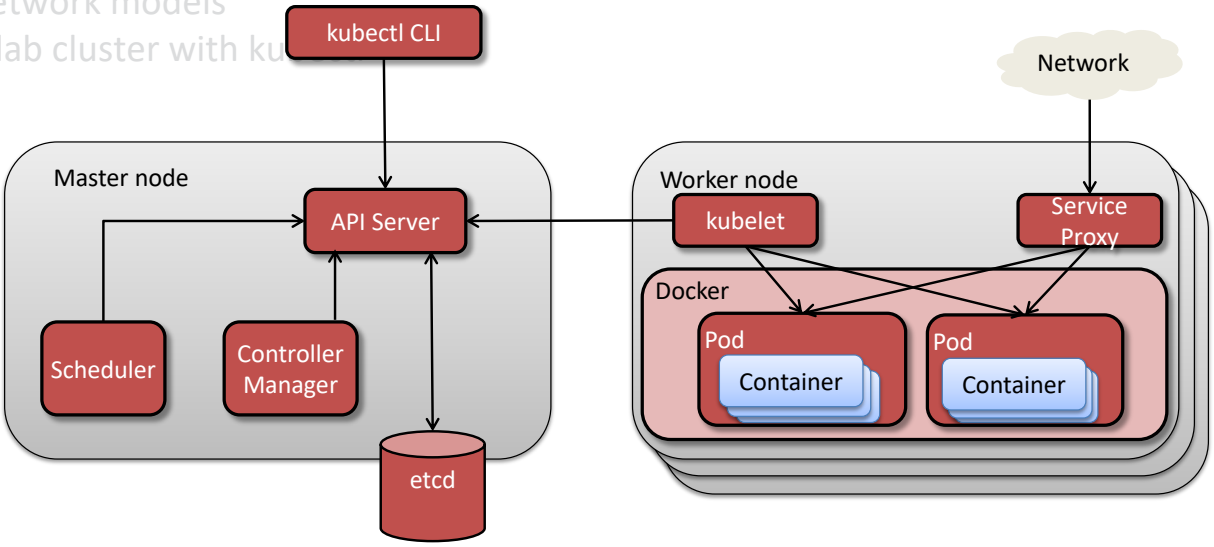
Module Summary

- Kubernetes architecture overview
- Kubernetes pod scheduling
- Kubernetes network models
- Appendix: Exploring the lab cluster with kubectl



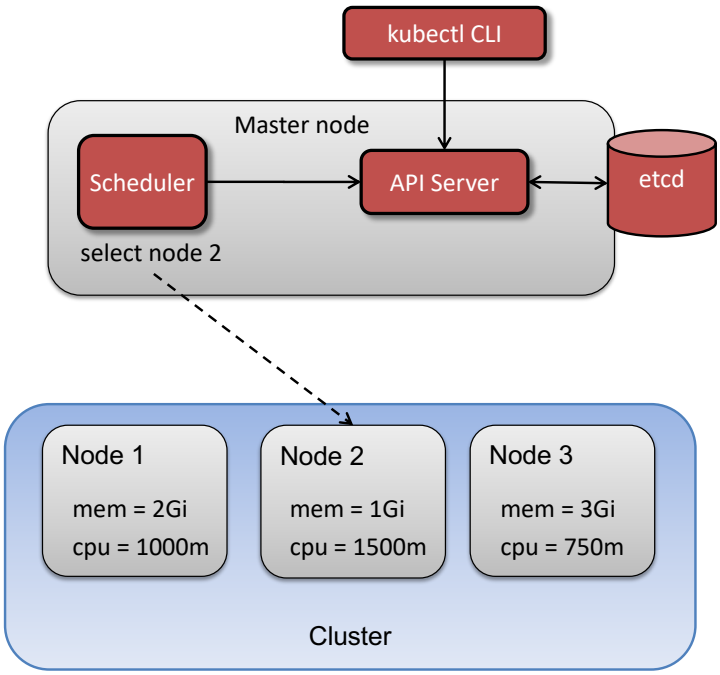
Review: Kubernetes Architecture

- Kubernetes architecture overview
- Kubernetes pod scheduling
- Kubernetes network models
- Exploring the lab cluster with ku



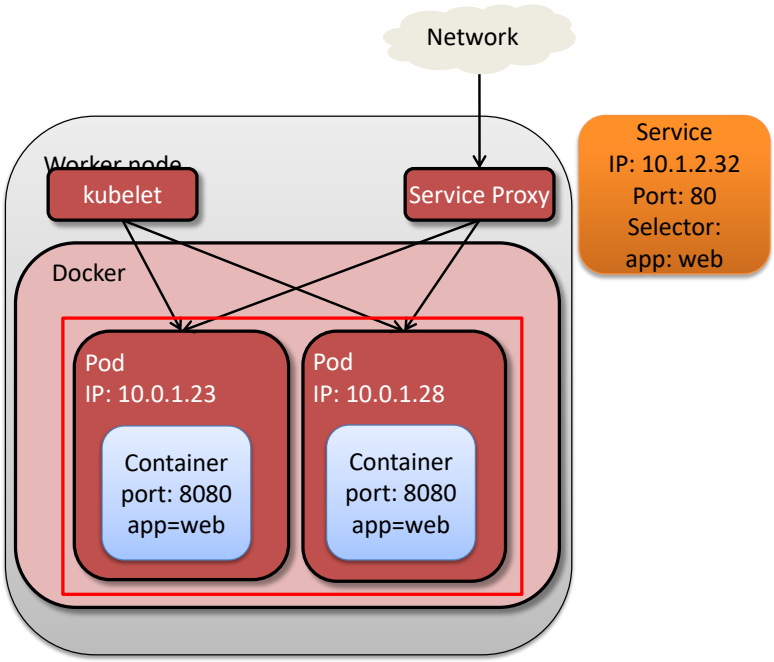
Review: Kubernetes Architecture

- Kubernetes architecture overview
- Kubernetes pod scheduling
- Kubernetes network models
- Exploring the lab cluster with kubectl



Review: Kubernetes Architecture

- Kubernetes architecture overview
- Kubernetes pod scheduling
- Kubernetes network models
- Exploring the lab cluster with kubectl



Review: Kubernetes Architecture

- Kubernetes architecture overview
- Kubernetes pod scheduling
- Kubernetes network models
- Exploring the lab cluster with kubectl

```
$ kubectl get nodes
```

```
$ kubectl describe node <NODENAME>
```

```
$ kubectl cluster-info

Kubernetes master is running at https://192.168.99.100:8443

KubeDNS is running at
https://192.168.99.100:8443/api/v1/proxy/namespaces/kube-
system/services/kube-dns

kubernetes-dashboard is running at
https://192.168.99.100:8443/api/v1/proxy/namespaces/kube-
system/services/kubernetes-dashboard
```


Questions

How Kubernetes Runs Workloads

All Workloads are Containerized

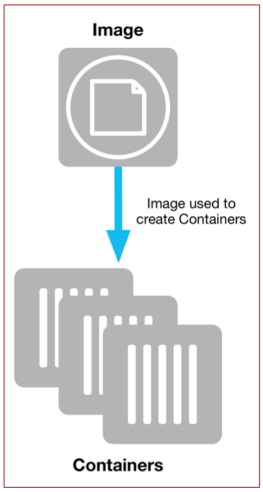
Docker allows you to package an application with its dependencies into a standardized unit for software development and deployment

Image

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

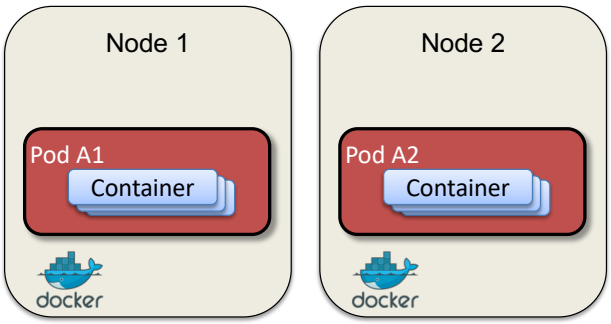
Container

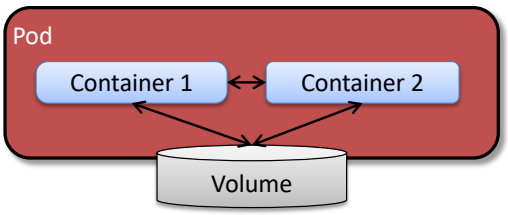
- Isolated application instance
- Created from a Docker image
- Based on Linux kernel primitives
  - Namespaces (resource visibility)
  - Control groups/cgroups (resource limits)



Kubernetes Pods

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

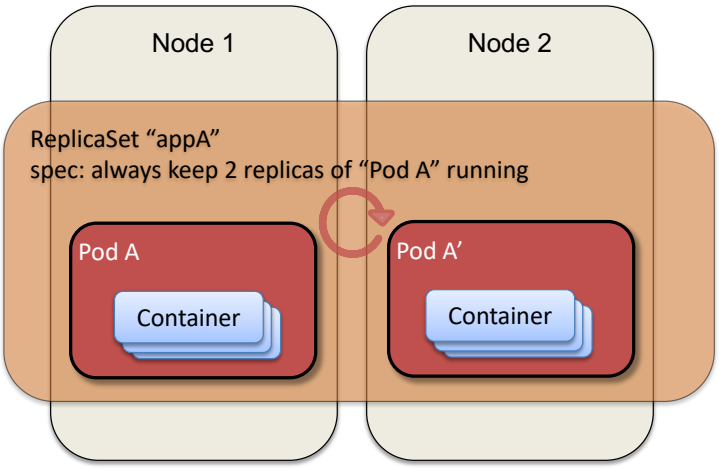




Controllers for Different Application Patterns

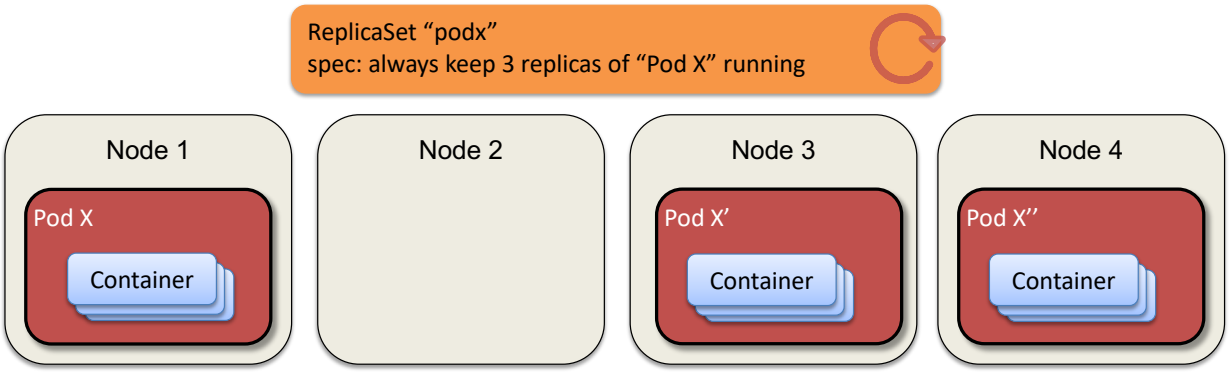
- K8s controller objects used to create and manage Pods according to different application patterns => control loops
- **ReplicaSets** manage sets of replicas of stateless workloads to ensure availability
- **StatefulSets** manage stateful workloads on stable storage to ensure consistency
- **DaemonSets**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

Controller example: ReplicaSet



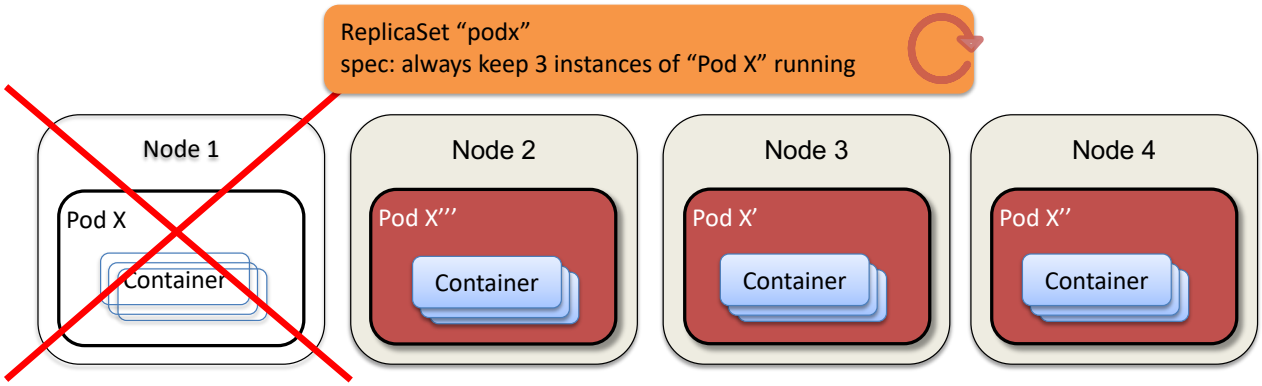
Controller Example: ReplicaSets

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



Replication Ensures Application Availability

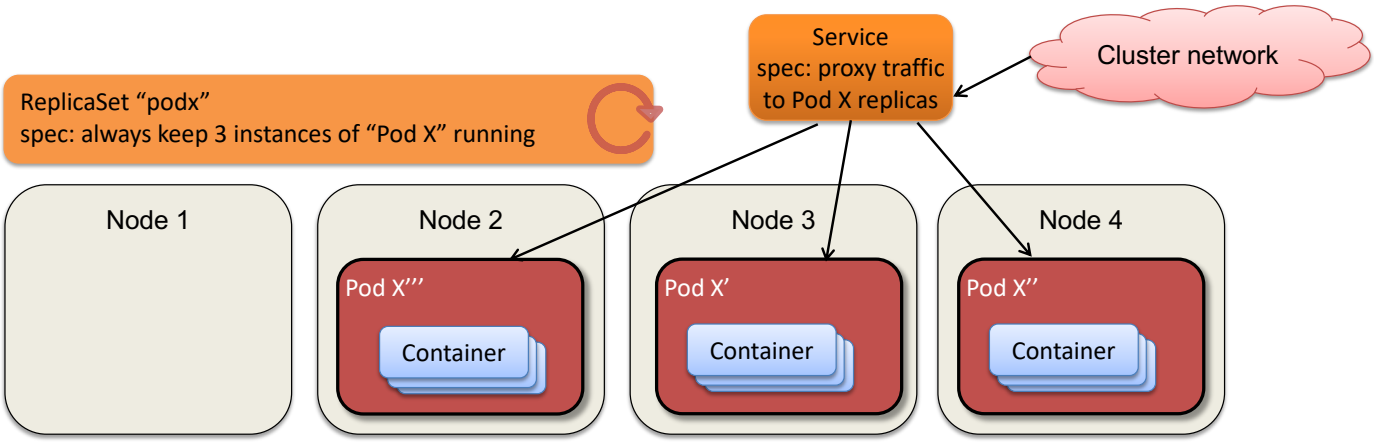
- When a Node fails, its Pods are lost
- K8s system manages the state of the ReplicaSet back to the declared configuration
- Changing the configuration will result in management to new state, e.g. scale out



Kubernetes Services Expose Applications

Services are named load balancers for application endpoints

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



Kubernetes 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

Example: Pod Object Description

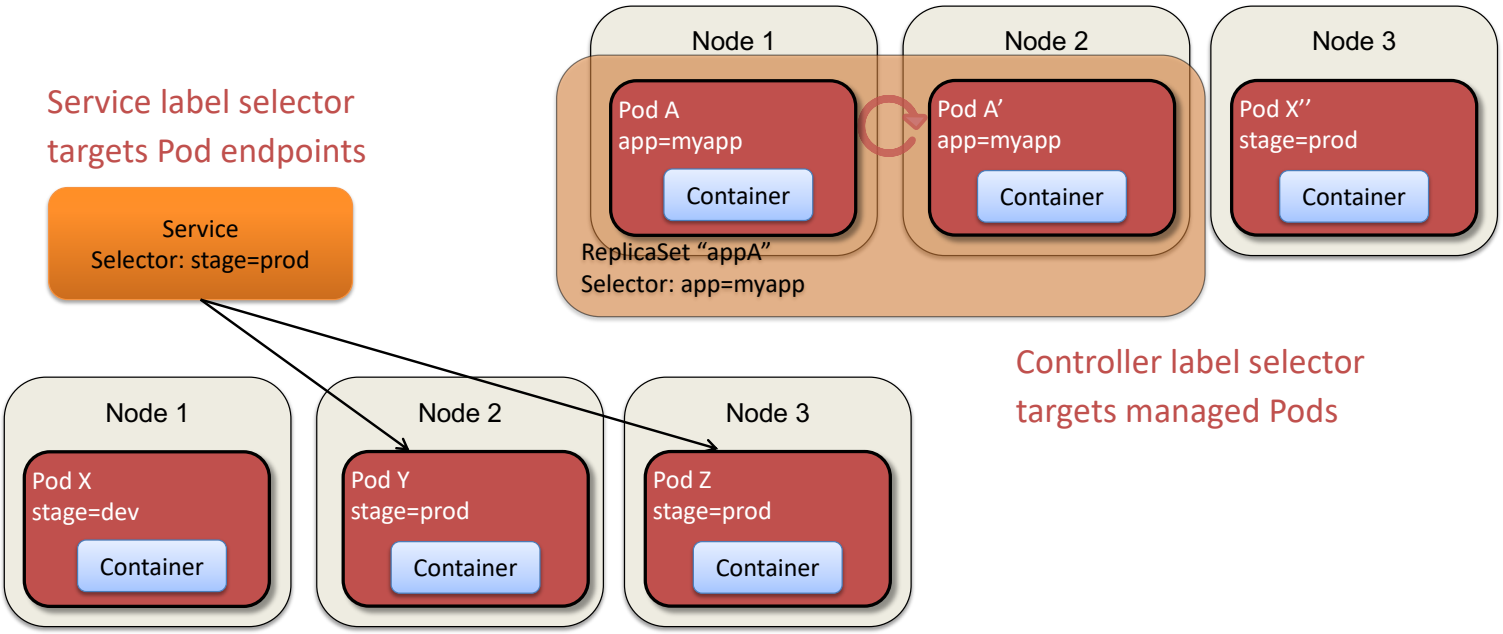
- Kubernetes API allows retrieval of object runtime state in choice of format
  - Full **metadata**, with auto-generated values
  - Full **spec**, with defaults
  - Current **status**
- Users can update object configuration and state via K8s API operations
- Object state also subject to change by K8s system, or other objects

```
$ kubectl get pods/nginx –o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/limit-ranger: 'LimitRanger plugin set: cpu request for
container nginx'
  creationTimestamp: 2017-07-12T17:13:54Z
  labels:
    app: website
    tier: frontend
  name: nginx
  namespace: default
  resourceVersion: "115751"
  selfLink: /api/v1/namespaces/default/pods/nginx
  uid: 7b6ac5ac-6725-11e7-804c-06ac88706266
spec:  ….
```

Key Object Metadata Attributes

- Every object created has a **name** and a **UID**
  - **Name** must be unique in the object's namespace
  - **UID** created by K8s and unique in the lifetime of the cluster
- **Namespaces** can be used to create multiple virtual clusters in the same K8s system
  - **default** is the default namespace
  - **kube-system** is the namespace for objects created by the K8s system
- **Labels** are key/value pairs that users can use to assign attributes to objects
  - Principal method for identifying and selecting sets of objects in K8s
  - K8s also provides **annotations** for automated user systems to decorate objects
- K8s creates and maintains a **resourceVersion** for every object to track updates as part of optimistic concurrency control

Object Grouping and Selection via Labels

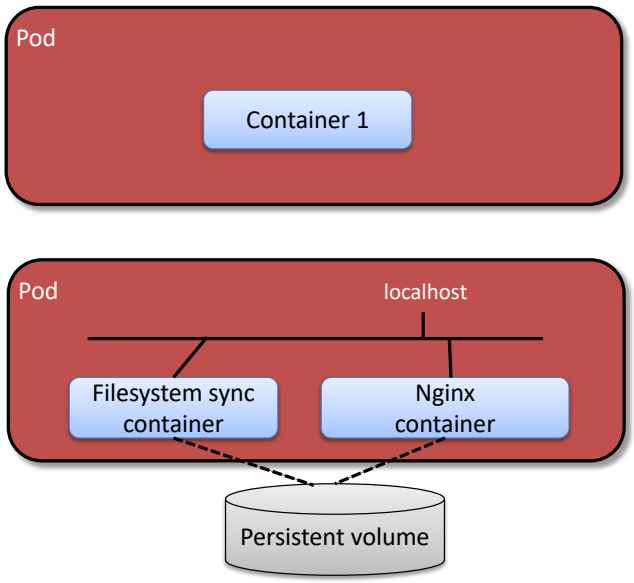


Kubernetes Pods

What is a Kubernetes Pod?

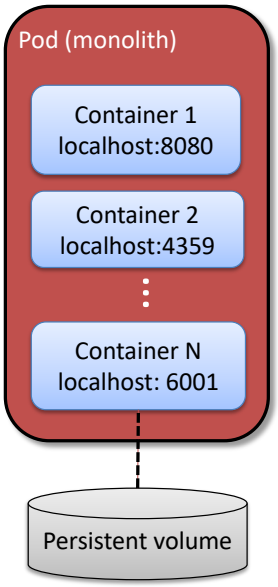
Kubernetes design intention: Pod == application instance

- 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



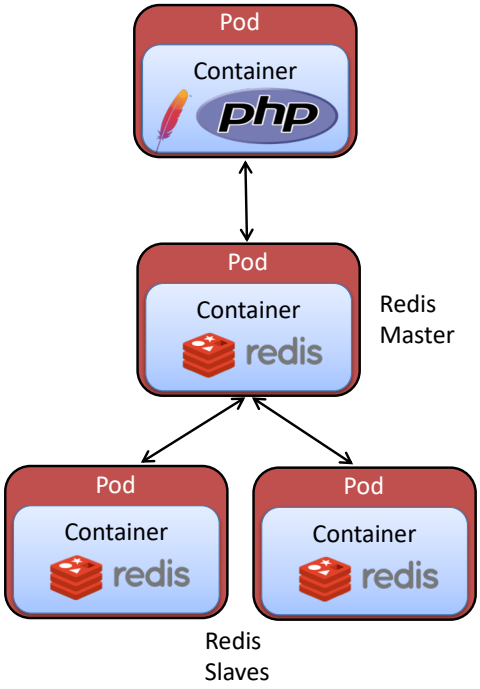
Pods Enable Deployment Flexibility

- 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



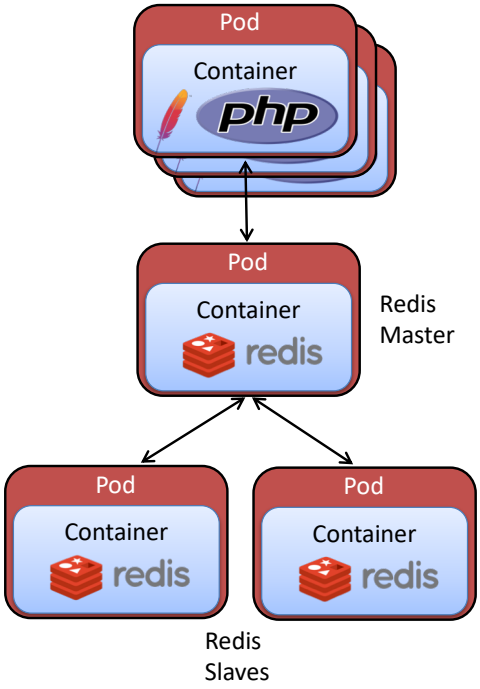
Pods Enable Deployment Flexibility

- 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

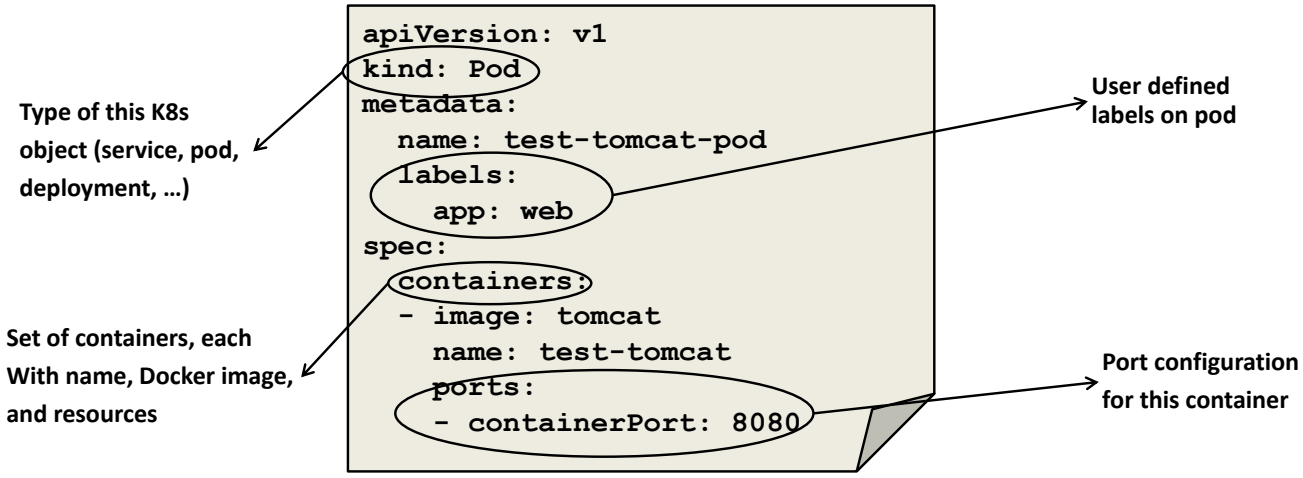
Like other K8s objects, pods can be defined in YAML or JSON files

- 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



- Configuration options similar to creating Docker container directly

Defining a Pod with Multiple Containers

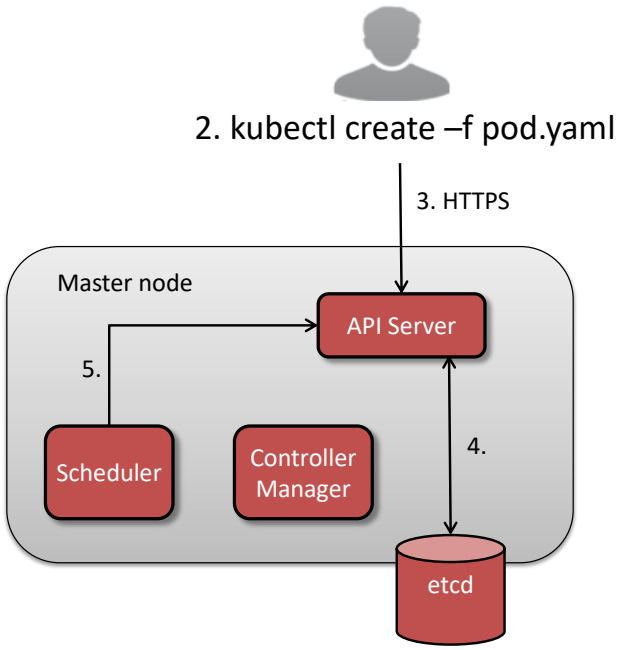
```
apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
spec:
  containers:
  - image: tomcat
    name: test-tomcat
    ports:
    - containerPort: 8080
  - image: mysql
    name: test-mysql
    ports:
    - containerPort: 3306
```

→ 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

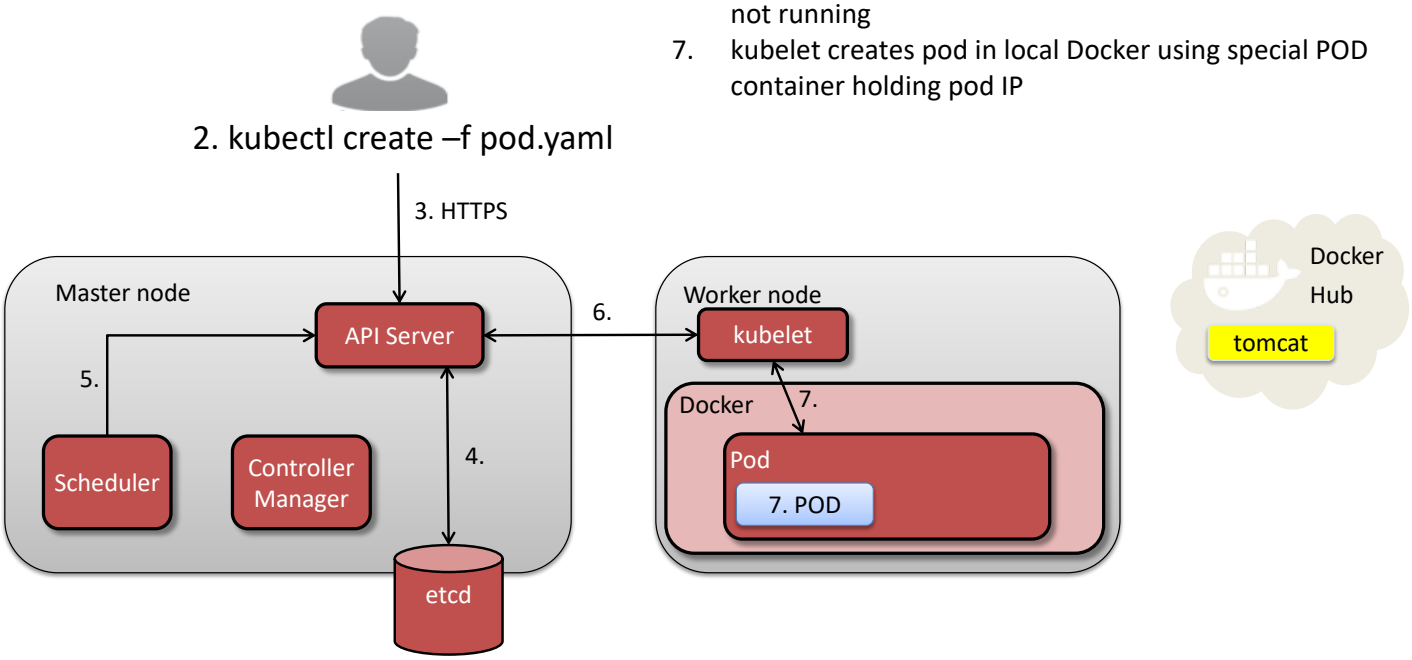
Pod Creation Process



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

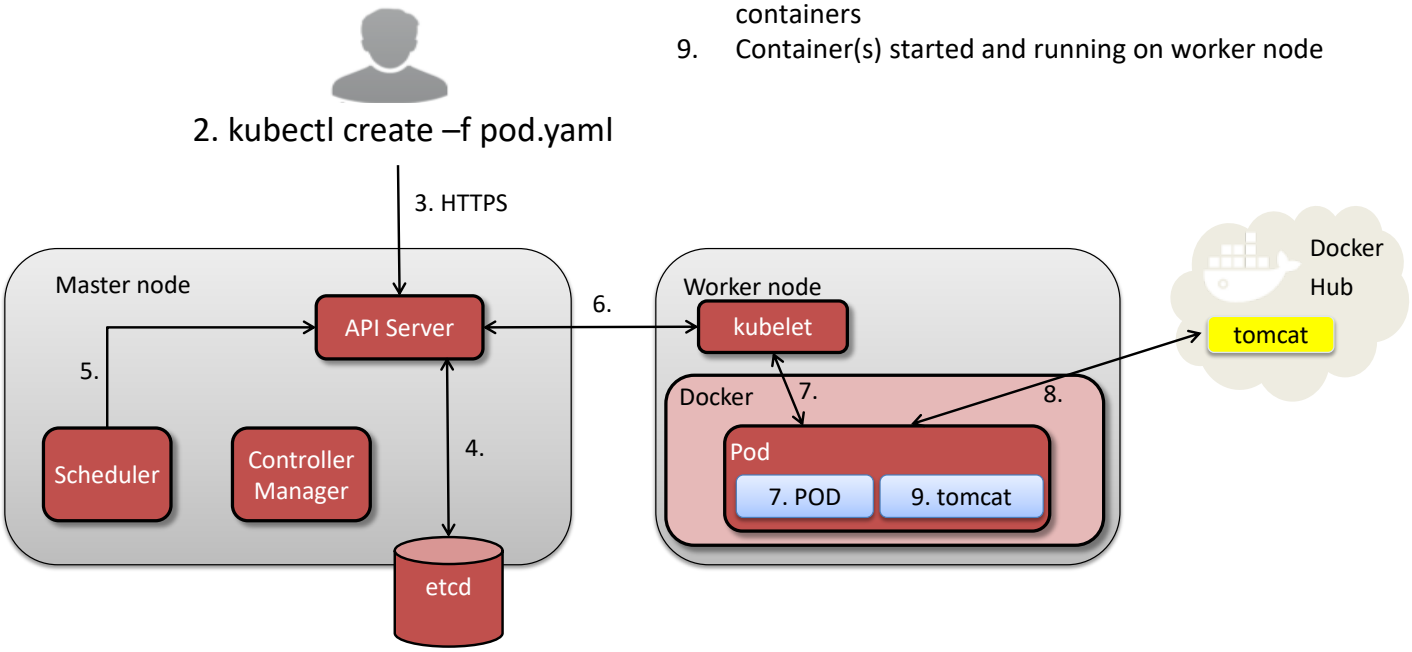
Pod Creation Process

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

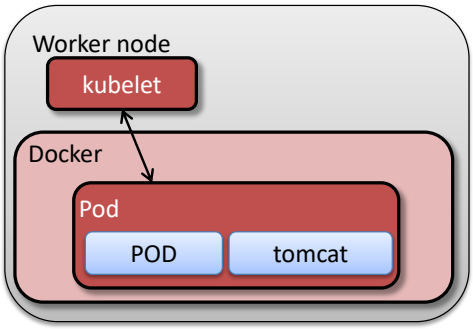
8. Kubelet pulls Docker image(s) for pod workload containers
9. Container(s) started and running on worker node



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
```



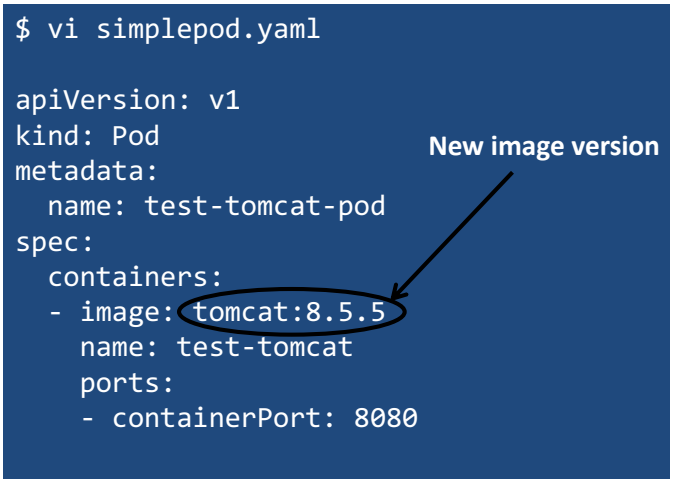
Modifying a Pod

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
...
```

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

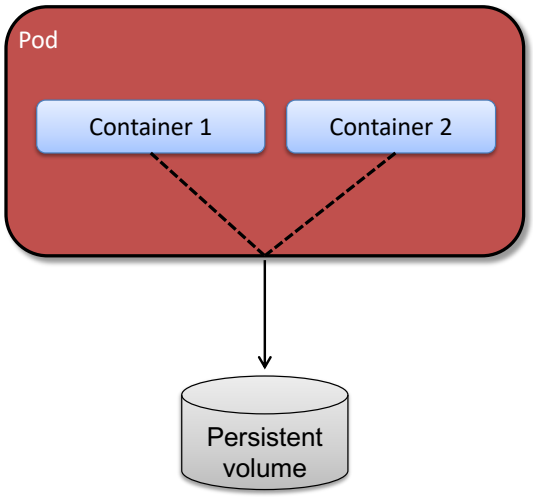
```
$ kubectl get pods --show-labels
NAME           READY   STATUS   RESTARTS AGE  LABELS
test-tomcat-pod 1/1     Running  1         1d  tier=frontend

$ 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

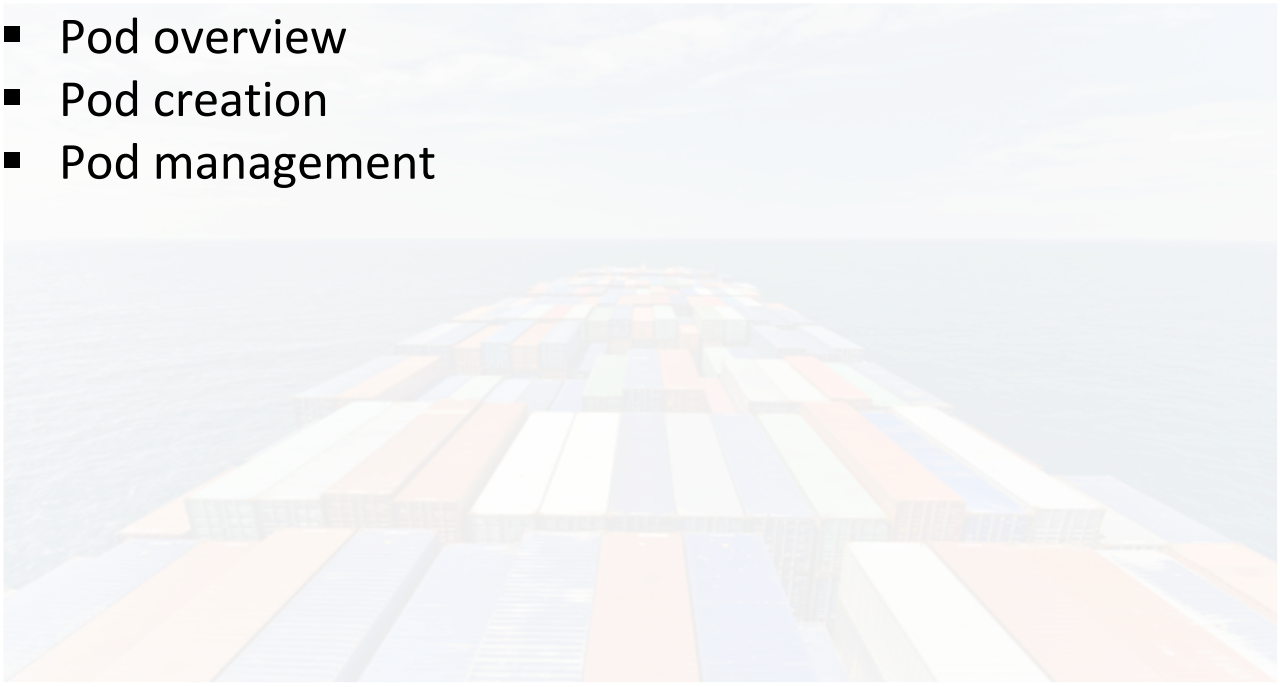
- When deleting a Pod, its containers will be removed and pod IP relinquished
- If an application needs to persist data, its pods must be configured to use persistent volumes for storage
- If a node dies, its local pods are also gone
- Best practice: use controller resources instead of managing pods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



Summary, Review, and Q&A

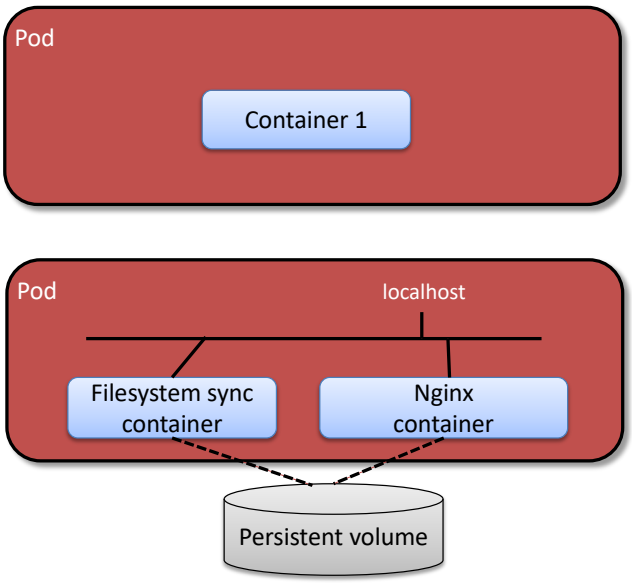
Review

- Pod overview
- Pod creation
- Pod management



Review: Kubernetes Pods and Services

- Pod overview
- Pod creation and management
  - Useful pod commands
- Services overview
- Services management
  - Useful services commands



- pod == application instance
- pod assigned cluster private IP
- pod has one or more containers sharing network and other resources

Review: Kubernetes Pods and Services

- Pod overview
- Pod creation and management
  - Useful pod commands
- Services overview
- Services management
  - Useful services commands

- Use manifest files to describe pods
- kubectl provides CRUD commands for objects
  - `kubectl get pod / kubectl describe pod`
  - `kubectl create pod / kubectl delete pod`
  - `kubectl apply –f <podmanifest.yaml>`

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

Kubernetes Services Overview

Kubernetes Service Objects:Microservices

Services provide abstraction between layers of an application

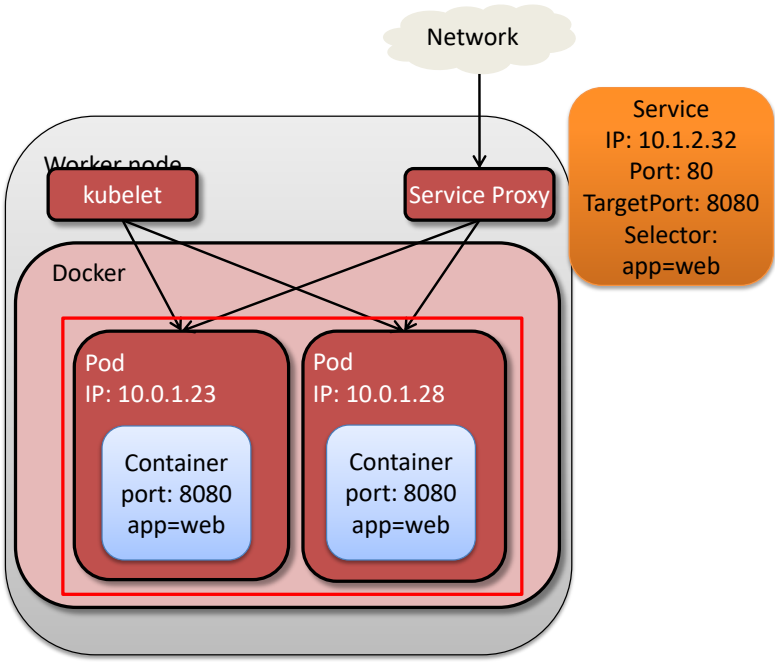
- **Service** object provides a stable IP for a collection of Pods
- Services use a label **selector** to target a specific set of pods as endpoints to receive proxied traffic
- Clients can reliably connect via the service IP and port(s), even as individual endpoint pods are dynamically created & destroyed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

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

sampleservice.yaml

Services Provide Abstraction Layer for Applications

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



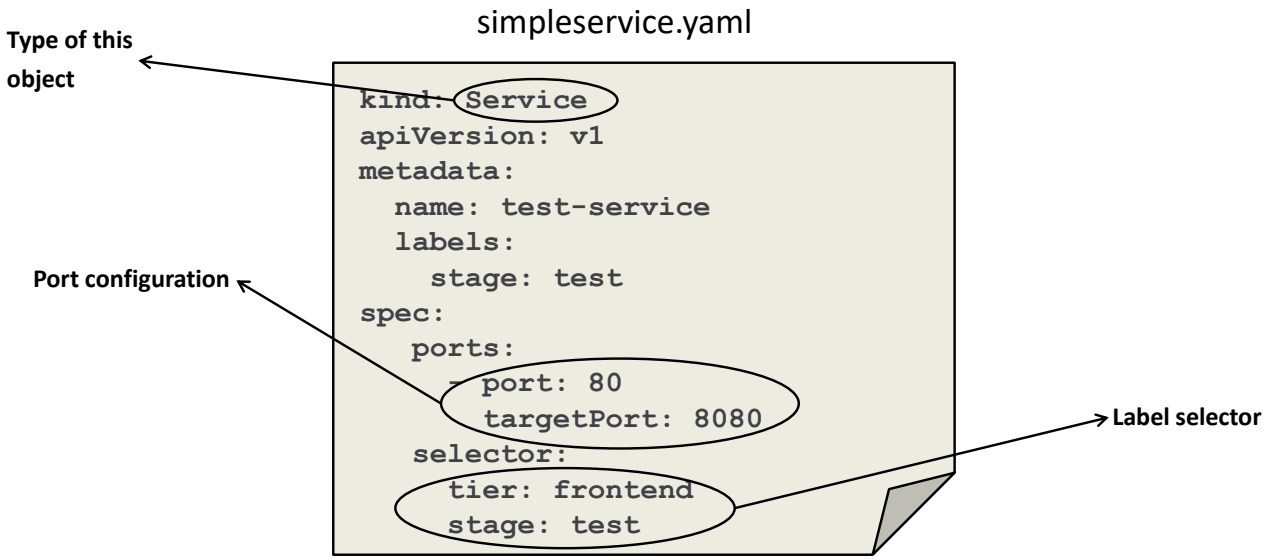
Defining a Service Using a Manifest File

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

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

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

Reviewing a Service Manifest File



ServiceTypes and Exposing Applications

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

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

Exposing Services at L7 through Ingresses

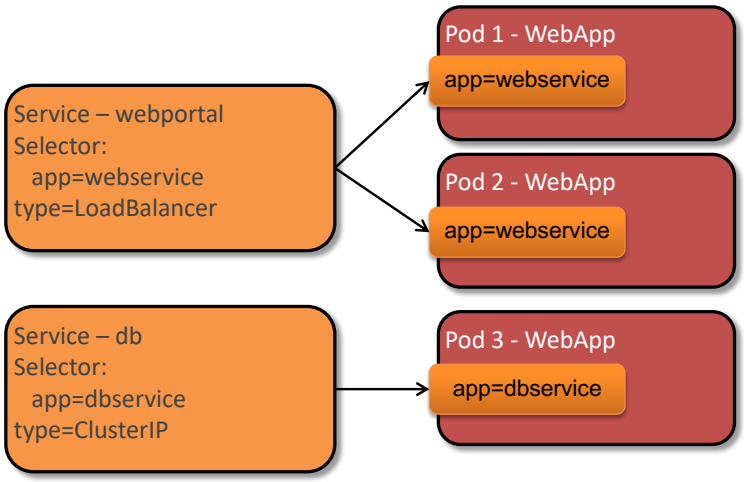
- Kubernetes also provides the facility to define **Ingress** resource to configure an external loadbalancer at L7
- Spec of Ingress resource is a set of rules matching HTTP host/url paths to specific Service backends
- Ingresses require the cluster to be running an appropriately configured Ingress controller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

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

Selecting Pods as Service Endpoints

Service's pod selector based on labels

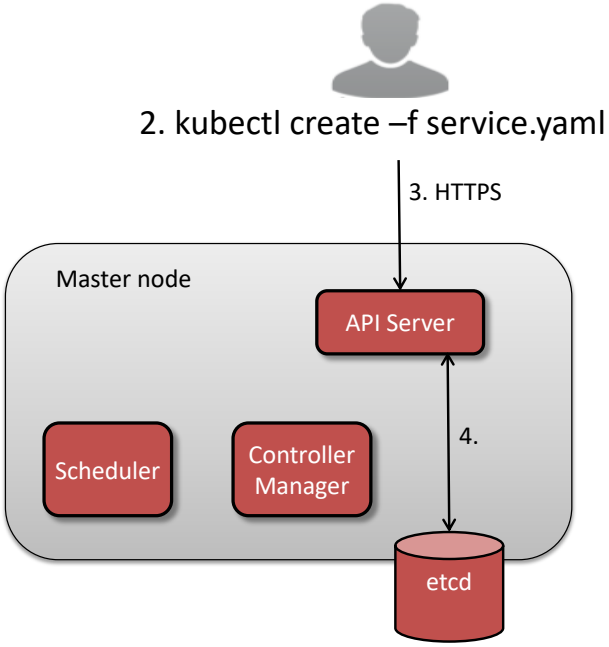
- Multiple pods can have the same label, unlike pod names which are unique in the namespace
- K8s system re-evaluates Service's selector continuously
- K8s maintains **endpoints** object of same name with list of pod IP:port's matching Service's selector




Questions

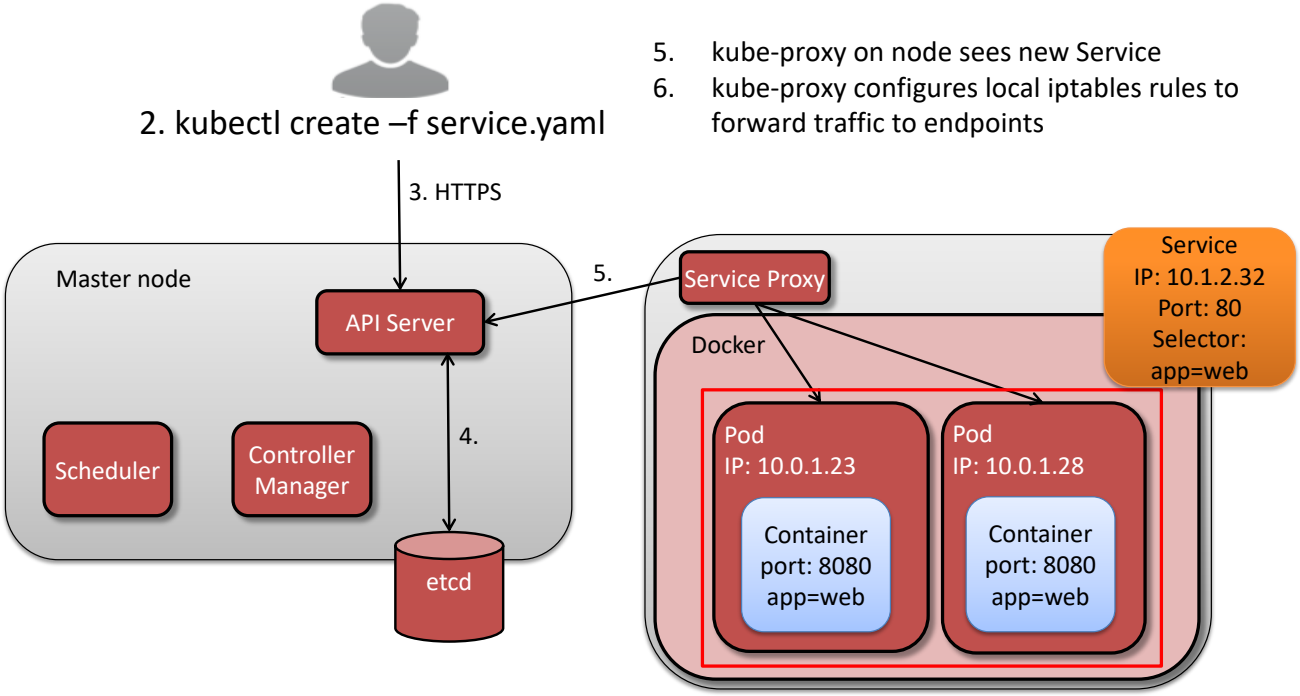
Services Management

Service Creation Process



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

Service Creation Process



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

Accessing Services Externally

Checking the service external IP

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

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

Deleting Services

Services can be deleted anytime

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

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

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

Service Discovery via DNS

Kubernetes advertises services via cluster DNS

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

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

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

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

Service Discovery via Environment Variables

Kubernetes advertises services via local environment

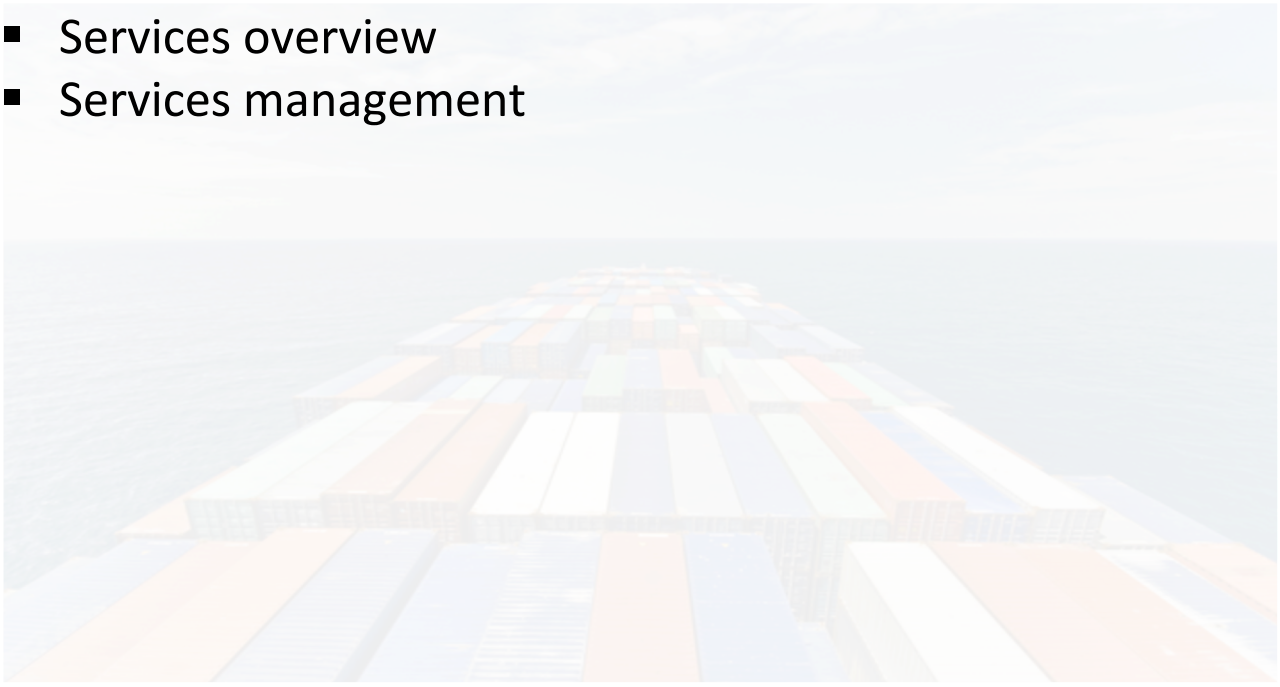
- When a Pod is started on a node, kubelet will create environment variables for Services in the Pod's namespace and system namespace

```
$ kubectl exec -ti busybox -- printenv | grep SERVICE
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
TEST_SERVICE_SERVICE_HOST=10.0.0.102
TEST_SERVICE_PORT_8080_TCP_PROTO=tcp
TEST_SERVICE_PORT_8080_TCP_PORT=8080
TEST_SERVICE_PORT=tcp://10.0.0.102:8080
TEST_SERVICE_PORT_8080_TCP_ADDR=10.0.0.102
TEST_SERVICE_SERVICE_PORT=8080
TEST_SERVICE_PORT_8080_TCP=tcp://10.0.0.102:8080
```

Summary, Review, and Q&A

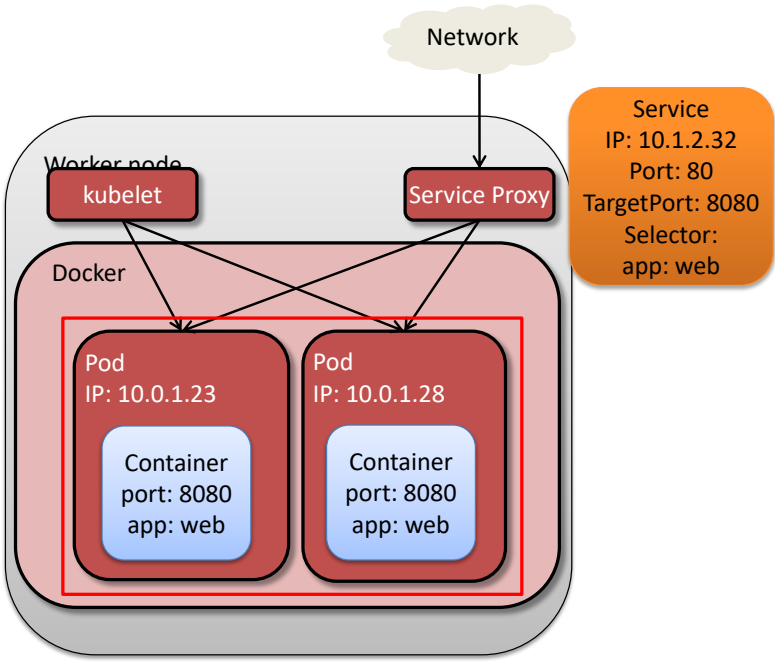
Module Summary

- Services overview
- Services management



Review: Kubernetes Pods and Services

- Services overview
- Services management
  - Useful services commands



- Service objects are abstractions that associate a stable IP with a set of pods selected by label
- Services used for
  - Communications between application tiers
  - expose applications outside the K8s cluster

Review: Kubernetes Pods and Services

- Services overview
- Services management
  - Useful services commands

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

- Use manifest files to describe services
- kubectl provides CRUD commands for objects
  - > kubectl get service / kubectl describe service
  - > kubectl create service / kubectl delete service
  - > kubectl apply –f <servicemanifest.yaml>
  - > kubectl edit <service_name>

Lab: Pods


Questions