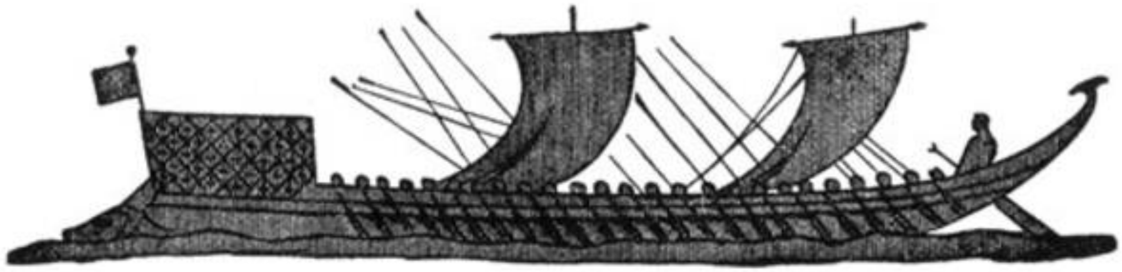
# Chapter 3
# Container Orchestration with Kubernetes

# Learning Topics

- Overview of Kubernetes
- Kubernetes Architecture
- Kubernetes Installation – Bare Metal
- Key Components
- Deploy Dockerized Apps
- Horizontal Pod AutoScaler

# Kubernetes Meaning

Greek for "pilot" or
"Helmsman of a ship"



# K8s History

- Project that was spun out of Google as an open source container orchestration platform.
- Built from the lessons learned in the experiences of developing and running Google’s Borg and Omega.
- Designed from the ground-up as a **loosely coupled** collection of components centered around deploying, maintaining and scaling workloads.

# K8s History

- Contributors include Google, CodeOS, Redhat, Mesosphere, Microsoft, HP, IBM, VMWare, Pivotal, SaltStack etc.

- Kubernetes is loosely coupled, meaning that all the components have little knowledge of each other and function independently.
    - This makes them easy to replace and integrate with a wide variety of systems

- Written in Go Language

# Who Manages Kubernetes

CLOUD NATIVE
COMPUTING FOUNDATION

- Sub-Foundation of Linux Foundation
- A Vendor Neutral Entity to Manage “Cloud Native” Projects
- Focused on
  - Containers
  - Dynamic Orchestration
  - Many More Services

# Container Issues

**Scheduling**: Where should my containers run?

**Lifecycle and health**: Keep my containers running despite failures

**Discovery**: Where are my containers now?

**Monitoring**: What's happening with my containers?

**Auth{n,z}**: Control who can do things to my containers

**Aggregates**: Compose sets of containers into jobs

**Scaling**: Making jobs bigger or smaller

# What Does Kubernetes Do?

- Groups containers that make up an application into logical units for easy management and discovery
- It acts as an engine for resolving state by converging actual and the **desired state** of the system
- It is declarative, you tell it what you want it to be, and it figures it out
  - e.g. ‘I want 3 instances of x’ and it just does it, if something dies, it brings it back to get to 3

# Most Popular Use Cases

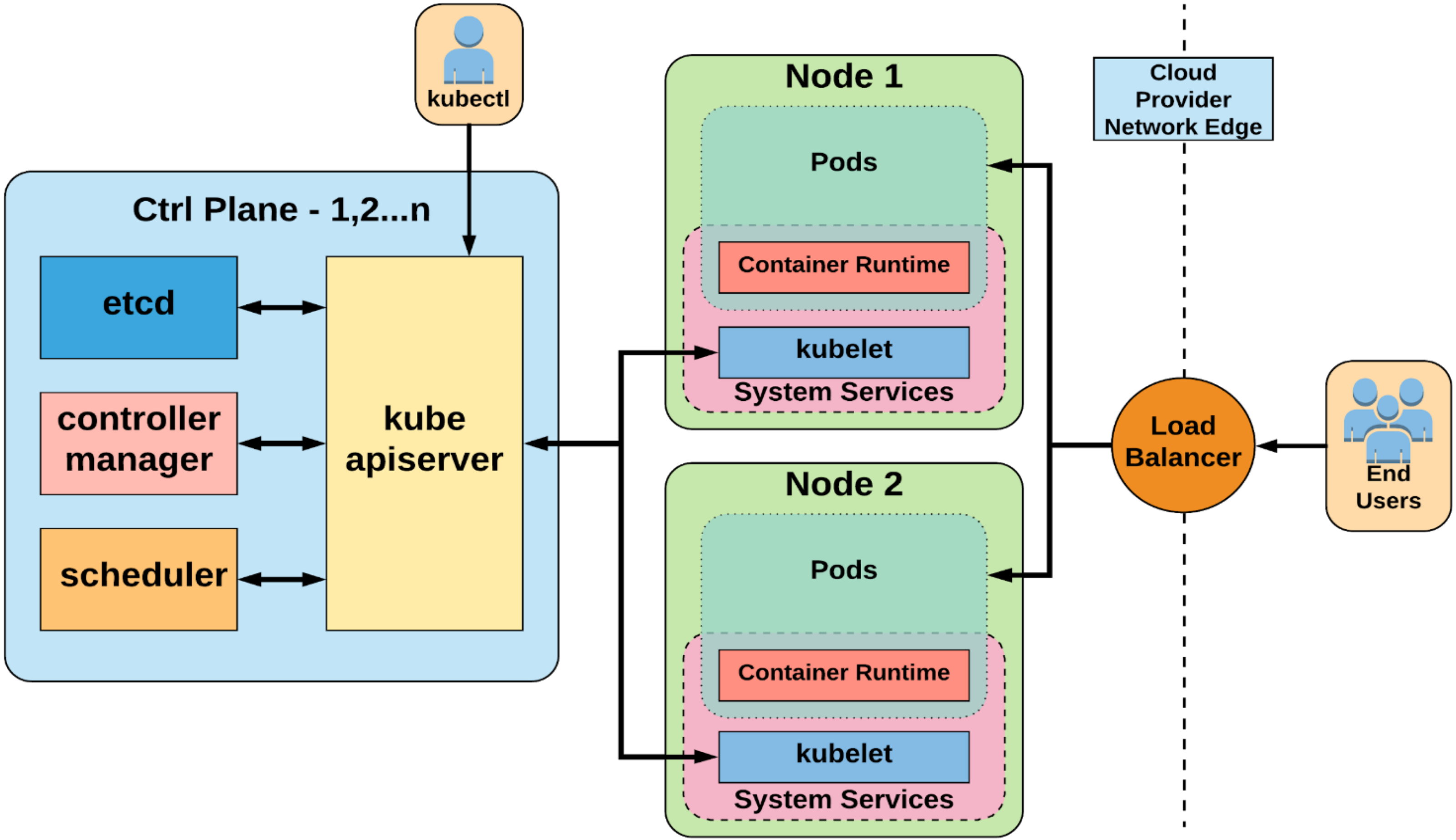
- Autoscale Workloads
- Blue/Green Deployments
- Scheduled Cronjobs
- Manage Stateless Applications
- Easily Integrate and Support 3rd Party Apps
- Provide Native Methods of Service Discovery

# Kubernetes Features

- Kubernetes is an open-source system for
  - Automating Deployment
  - Scaling
  - Management

  of containerized applications
- Kubernetes can scale without increasing your ops team

# Kubernetes Architecture



# Key Terminologies

- **Pod** - A group of Containers
- **Labels** - Labels for identifying pods
- **Kubelet** - Container Agent
- **Proxy** - A load balancer for Pods
- **etcd** - A metadata service
- **cAdvisor** - Container Advisor provides resource usage/performance statistics
- **Replication Controller** - Manages replication of pods
- **Scheduler** - Schedules pods in worker nodes
- **API Server** - Kubernetes API server

# Control Plane Components

- **Kube-apiserver**
  - Gate keeper for everything in kubernetes
  - EVERYTHING interacts with kubernetes through the apiserver
- **Etcd**
  - Distributed storage back end for kubernetes
  - The apiserver is the only thing that talks to it
- **Kube-controller-manager**
  - The home of the core controllers
- **Kube-scheduler**
  - Handles placement

# Kube-apiserver

- Provides a forward facing REST interface into the kubernetes control plane and datastore.
- All clients and other applications interact with kubernetes **strictly** through the API Server.
- Handles authn, authz, request validation, mutation and admission control and serves as a generic front end to the backing datastore

# etcd

- etcd acts as the cluster datastore.
- Purpose in relation to Kubernetes is to provide a strong, consistent, highly durable and highly available key-value store for persisting cluster state.
- Stores objects and config information.

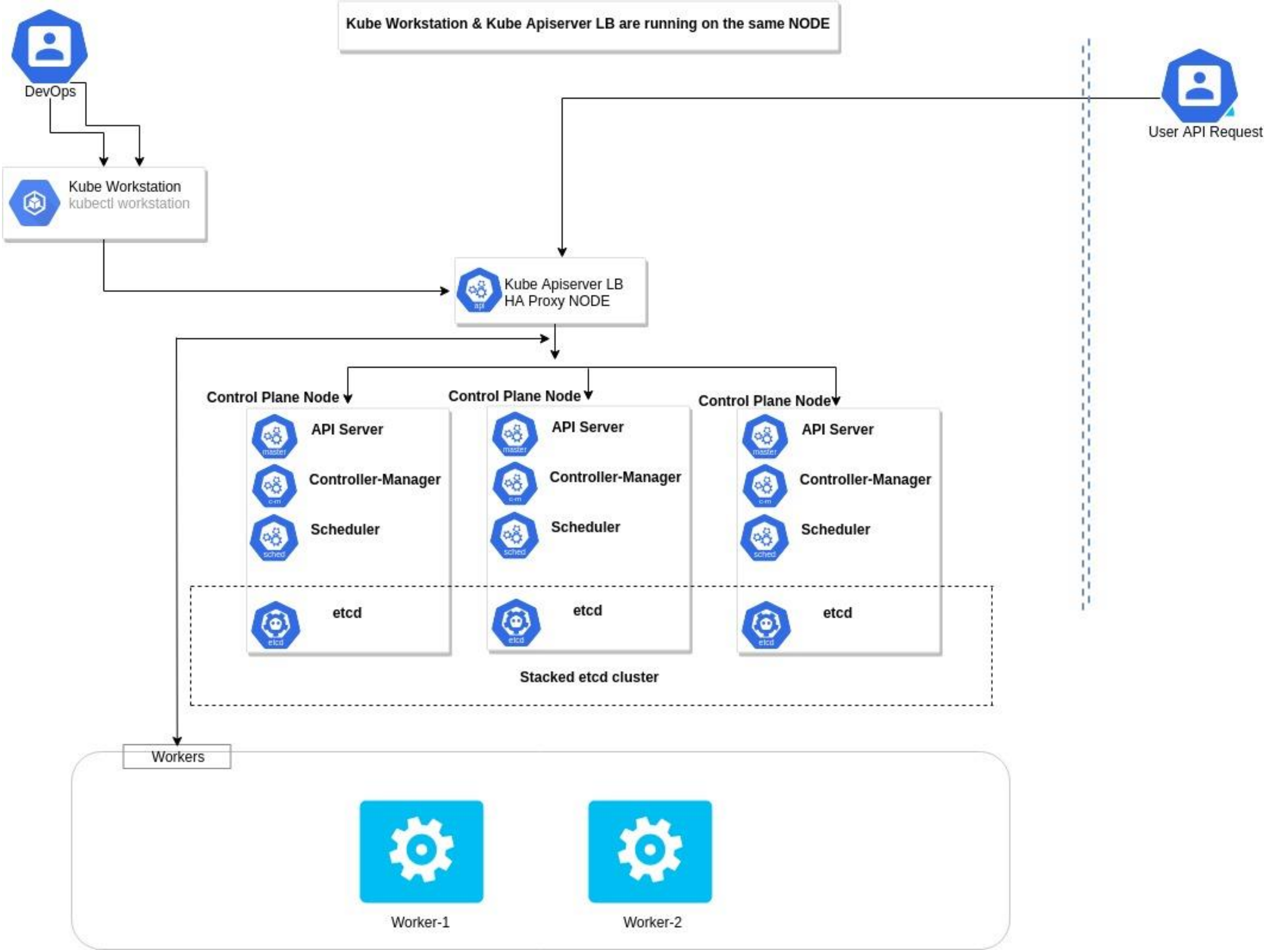
# Kube-controller-manager

- Its the director behind the scenes
- Serves as the primary daemon that manages all core component control loops.
- Monitors the cluster state via the apiserver and **steers the cluster towards the desired state**.
- Does NOT handle scheduling, just decides what the desired state of the cluster should look like
  - e.g. receives request for a deployment, produces replicaset, then produces pods

# Kube-scheduler

- Scheduler decides which nodes should run which pods
- Serves as the primary daemon that manages all core component control loops.
- Monitors the cluster state via the apiserver and **steers the cluster towards the desired state**.
- Does NOT handle scheduling, just decides what the desired state of the cluster should look like
  - e.g. receives request for a deployment, produces replicaset, then produces pods

# HA Cluster Architecture



# Node Components

- **Kubelet**
  - Agent running on every node, including the control plane
- **Kube-proxy**
  - The network ‘plumber’ for Kubernetes services
  - Enables in-cluster load-balancing and service discovery
- **Container Runtime Engine**
  - The containerizer itself - typically docker

# Pods

- Smallest Unit of Work
- Collection of One or More Containers
- Share Volumes, Network Namespace
- Part of Single Context, Managed Together
- Ephemeral in Nature

# Kubelet

- Acts as the node agent responsible for managing the lifecycle of every pod on its host.
- Kubelet understands YAML container manifests that it can read from several sources:
    - file path
    - HTTP Endpoint
    - etcd watch acting on any changes
    - HTTP Server mode accepting container manifests over a simple API.
- The single host daemon required for a being a part of a kubernetes cluster

# Kube-proxy

- Manages the network rules on each node.
- Performs connection forwarding or load balancing for Kubernetes cluster services.
- Creates the rules on the host to map and expose services
- Available Proxy Modes:
  - Userspace
  - iptables
  - ipvs (default if supported)

# Container Runtime Engine

- A container runtime is a CRI (Container Runtime Interface) compatible application that executes and manages containers.
  - Containerd (docker)

- Kubernetes functions with multiple different containerizers

- Interacts with them through the CRI - container runtime interface

- CRI creates a ‘shim’ to talk between kubelet and the container runtime

# Lab – K8 Installation

1. Use your own Azure account
2. Create a resource group, launch all VMs in this resource group
3. Create HA Proxy LB Server
4. Create first master node
5. Create second and third master node
6. Setup stacked control plane
7. Initiate cluster
8. Setup worker nodes
9. Setup HA Proxy as workstation

# HA Proxy LB Server

1. Open ports
   - Http, Https and SSH
   - Open 6443 after VM is provisioned (default port for Kubeapi)
2. Ssh to VM
   - Bind port 6643

# Create 3 Master Nodes

1. Open ports
2. Ssh to VM
   - Add Kubernetes certificates
   - Install kubectl, kubeadm, kubelet and CNI (Container Networking Interface)
   - Install Docker
3. Repeat for other 2 master nodes

# Setup Control Plane

1. Ssh to HAProxy VM
2. Update config file
   - Update IP addresses to private IP addresses of HAProxy and 3 Master VMs
   - Restart HAProxy service

# Initiate Cluster

1. Ssh to First Master VM
2. Note API Version is set as “Stable”
3. Create cluster
4. Save “Join” command that will be used later
5. Install CNI

# Copy Certificate

1. Applicable on Master 2 and Master 3
2. Copy certificate files from Master 1 to Master 2 and Master 3

# Join Master VMs

1. Applicable on Master 2 and Master 3
2. Run kubeadm join command on these two VMs using token generated on Master 1

# Setup Worker Nodes

1. Create two VMs
2. Open ports for Kubelet API and for NodePort services
3. Install Docker, kubeadm, kubelet and CNI
4. Run Join Token command from Master 1 VM

# Setup HAProxy VM as Workstation

1. Create two VMs
2. Open ports for Kubelet API and for NodePort services
3. Install Docker, kubeadm, kubelet and CNI
4. Run Join Token command from Master 1 VM

# Lab – K8 Dashboard

- Use your own Azure account

# API Server

# API Overview

- The REST API (API-Server) is the keystone of Kubernetes
- Every component communicates with API-Server
- Everything within Kubernetes is an API Object
- Object Oriented Application Architecture

# API Groups

- Every object in kubernetes belongs to an API Group
- The API Group is a REST compatible path

**Format:**

**/apis/*<group>*/*<version>*/*<resource>***

**Examples:**

**/apis/apps/v1/deployments**

**/apis/batch/v1beta1/cronjobs**

# API Versioning

- 3 tiers of API Maturity Level
    1. Alpha – May be buggy, disabled by default
    2. Beta – Code is well tested, enabled by default
    3. Stable – Released, stable and API schema will not change.

https://kubernetes.io/docs/concepts/overview/kubernetes-api/#api-versioning

# Kubernetes Objects

# Kubernetes Objects

- Persistent entities in the kubernetes system
- Kubernetes uses these entities to represent the state of cluster
  – What containerized applications are running (and on which nodes)
  – The resources available to those applications
  – The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

# Kubernetes Objects Cont.

- Record of Intent
  - Once object is created, kubernetes will constantly work to ensure that object exists
  - Tells about desired state of cluster
  - Need to use kubernetes API to create/modify/delete objects
  - Includes 2 nested object fields
    - Spec – You must provide, describes your desired state for the object
    - Status – Describes actual state of the object, supplied and updated by kubernetes

# Describe Kubernetes Objects

- Often, information is provided in a .yaml file to kubectl
- Kubectl converts the information to JSON when making the API request.

**Example Object**


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
```


**Example Status Snippet**


```yaml
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: 2018-02-14T14:15:52Z
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: 2018-02-14T14:15:49Z
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: 2018-02-14T14:15:49Z
    status: "True"
    type: PodScheduled
```

# Replica Sets

# Replica Sets

- Primary method of managing pod replicas and their lifecycle
- Includes their scheduling, scaling, and deletion.
- Their job is simple: **Always ensure the desired number of pods are running.**
- You define the number of pods you want running with the replicas field.
- Pods are created with name based off replicaset name + random 5 character string

# Replica Sets Cont.

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  template:
    metadata:
      labels:
        app: nginx
        env: prod
    spec:
      containers:
      - name: nginx
        image: nginx:stable-alpine
        ports:
        - containerPort: 80
```

```
$ kubectl get pods
NAME              READY   STATUS    RESTARTS   AGE
rs-example-9l4dt  1/1     Running   0          1h
rs-example-b7bcg  1/1     Running   0          1h
rs-example-mkll2  1/1     Running   0          1h
```

```
$ kubectl describe rs rs-example
Name:         rs-example
Namespace:    default
Selector:     app=nginx,env=prod
Labels:       app=nginx
              env=prod
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=nginx
           env=prod
  Containers:
   nginx:
    Image:        nginx:stable-alpine
    Port:         80/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
  Normal  SuccessfulCreate  16s   replicaset-controller  Created pod: rs-example-mkll2
  Normal  SuccessfulCreate  16s   replicaset-controller  Created pod: rs-example-b7bcg
  Normal  SuccessfulCreate  16s   replicaset-controller  Created pod: rs-example-9l4dt
```

# Deployments

# Deployments

- Declarative method of managing Pods via **ReplicaSets**
- Provides rollback functionality and update control
- Updates are managed through the **pod-template-hash** label
- Each iteration creates a unique label that is assigned to both the **ReplicaSet** and subsequent Pods

# Lab 3 – Deploy Java App on K8 Cluster

- Use your Java app image from Docker lab

# Lab 4 – Deploy .NET App on K8 Cluster

- Use your .NET app image from Docker lab

# Lab 5 – Deploy Python App on K8 Cluster

- Use your Python app image from Docker lab

# Deployments Cont.

- **revisionHistoryLimit**: The number of previous iterations of the Deployment to retain.
- **strategy**: Describes the method of updating the Pods based on the type. Valid options are **Recreate** or **RollingUpdate**.
  - **Recreate**: All existing Pods are killed before the new ones are created.
  - **RollingUpdate**: Cycles through updating the Pods according to the parameters: maxSurge and maxUnavailable.
- **maxSurge** == how many ADDITIONAL replicas we want to spin up while updating

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-example
spec:
  replicas: 3
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    <pod template>
```

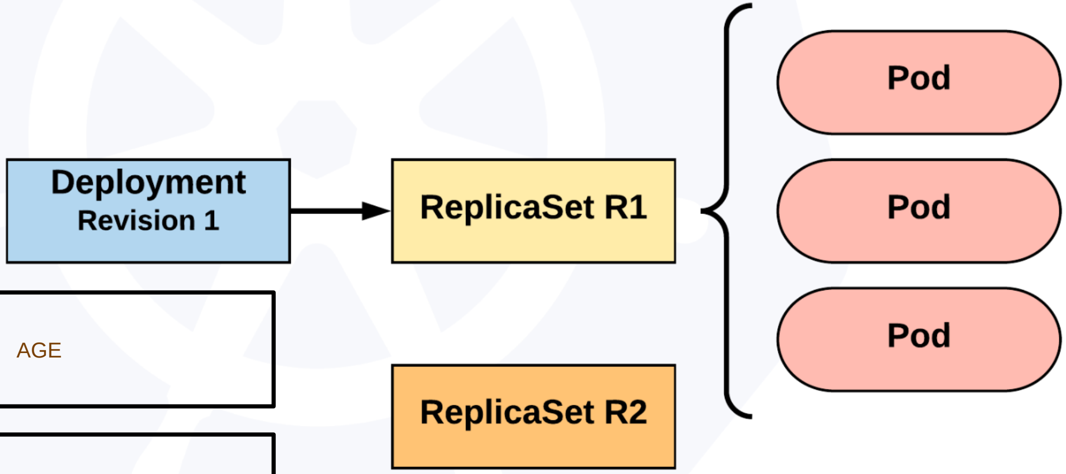
# RollingUpdate Deployment

Updating pod template generates a new **ReplicaSet** revision.

**R1 pod-template-hash:**
676677fff
**R2 pod-template-hash:**
54f7ff7d6d

```
$ kubectl get replicaset
NAME                DESIRED   CURRENT   READY    AGE
mydep-6766777fff        3         3         3        5h
```

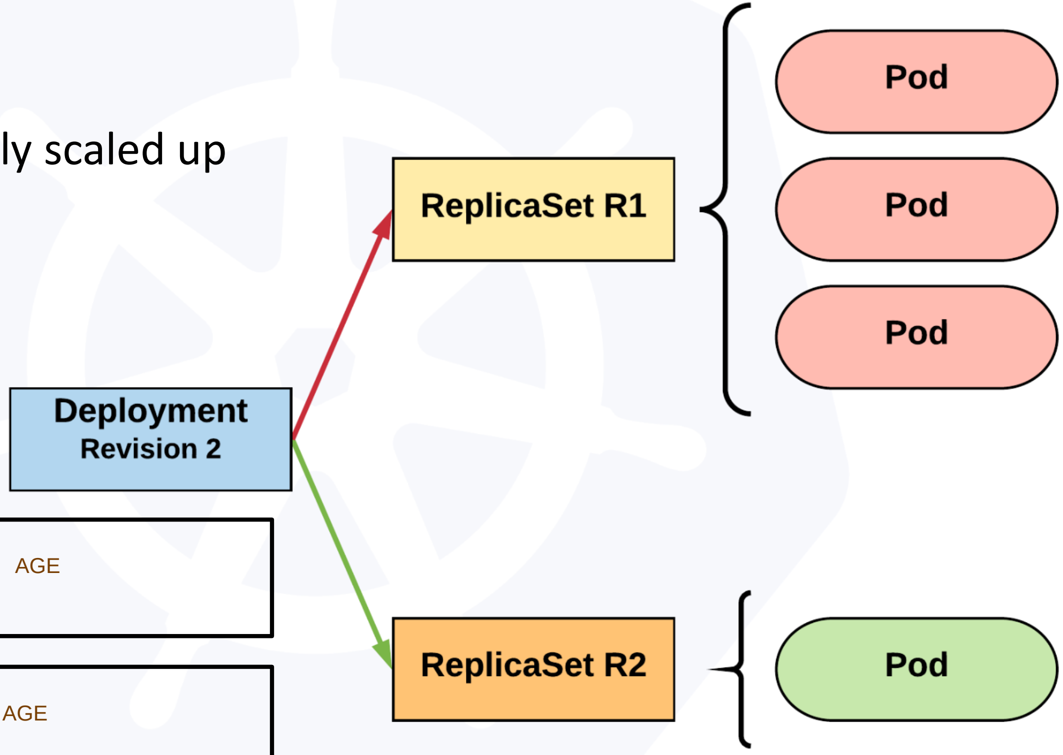
```
$ kubectl get pods
NAME                      READY    STATUS    RESTARTS   AGE
mydep-6766777fff-9r2zn   1/1      Running   0          5h
mydep-6766777fff-hsfz9   1/1      Running   0          5h
mydep-6766777fff-sjxhf   1/1      Running   0          5h
```



# RollingUpdate Deployment

New **ReplicaSet** is initially scaled up based on maxSurge.

**R1 pod-template-hash:**
676677fff
**R2 pod-template-hash:**
**54f7ff7d6d**




```
$ kubectl get replicaset
NAME               DESIRED   CURRENT   READY    AGE
mydep-54f7ff7d6d       1         1         1      5s
mydep-6766777fff       2         3         3      5h
```

```
$ kubectl get pods
NAME                      READY    STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll    1/1      Running   0          2s
mydep-6766777fff-9r2zn    1/1      Running   0          5h
mydep-6766777fff-hsfz9    1/1      Running   0          5h
mydep-6766777fff-sjxhf    1/1      Running   0          5h
```

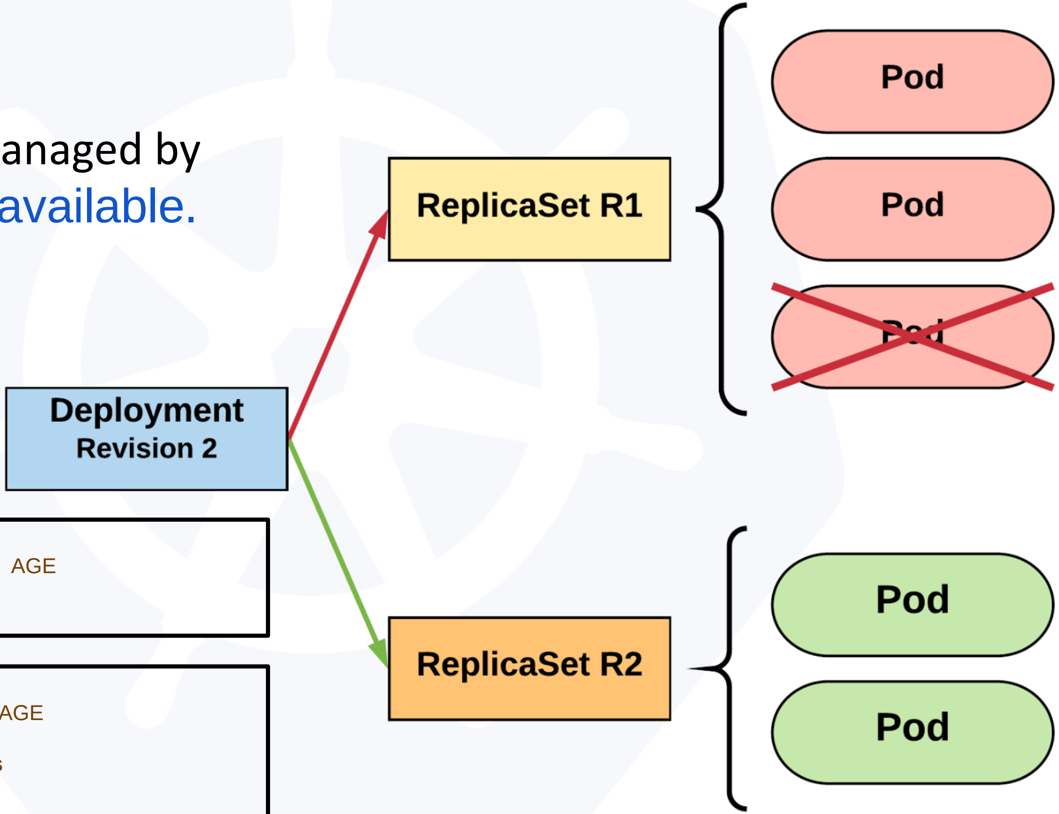
# RollingUpdate Deployment

Phase out of old Pods managed by maxSurge and maxUnavailable.

**R1 pod-template-hash:**
676677fff
**R2 pod-template-hash:**
**54f7ff7d6d**

```
$ kubectl get replicaset
NAME                DESIRED   CURRENT   READY    AGE
mydep-54f7ff7d6d       2         2         2       8s
mydep-6766777fff       2         2         2       5h
```

```
$ kubectl get pods
NAME                     READY    STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll   1/1      Running   0          5s
mydep-54f7ff7d6d-cqvlq   1/1      Running   0          2s
mydep-6766777fff-9r2zn   1/1      Running   0          5h
mydep-6766777fff-hsfz9   1/1      Running   0          5h
```



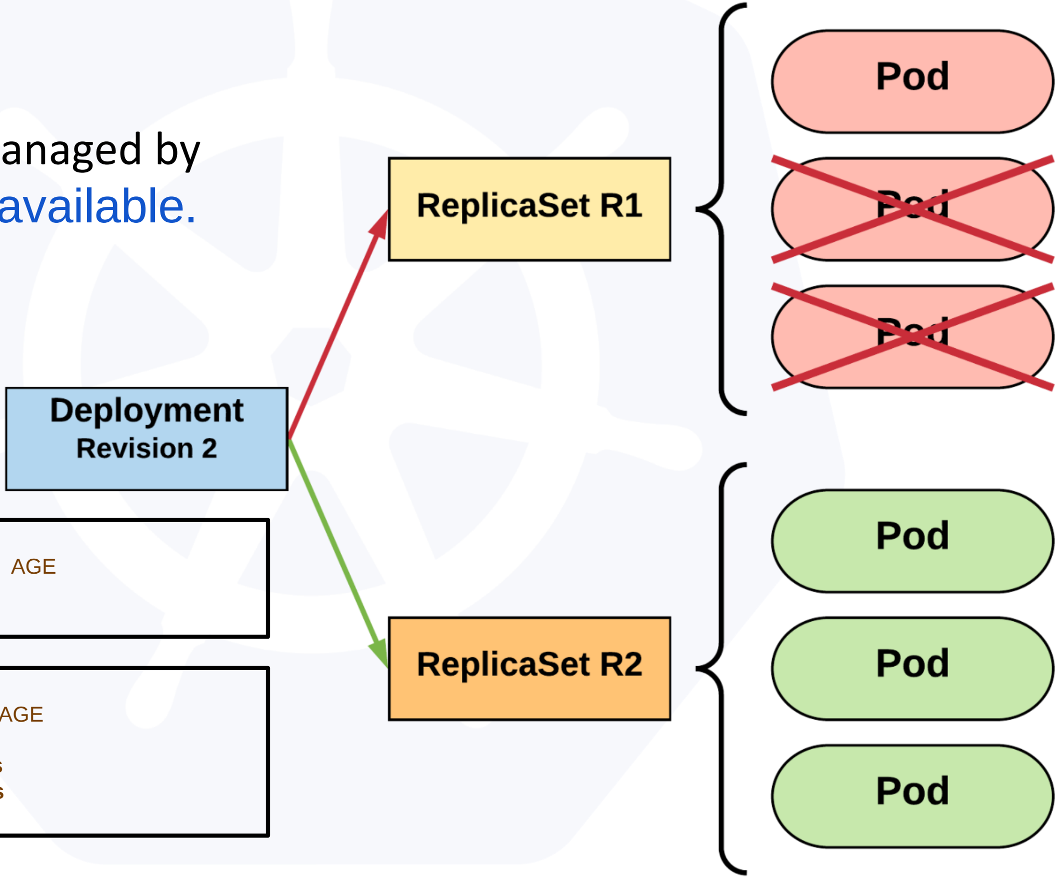
# RollingUpdate Deployment

Phase out of old Pods managed by maxSurge and maxUnavailable.

**R1 pod-template-hash:**
676677fff
**R2 pod-template-hash:**
**54f7ff7d6d**

```
$ kubectl get replicaset
NAME                DESIRED   CURRENT   READY    AGE
mydep-54f7ff7d6d         3         3       3      10s
mydep-6766777fff        0         1        1       5h
```

```
$ kubectl get pods
NAME                     READY    STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll   1/1      Running   0          7s
mydep-54f7ff7d6d-cqvlq   1/1      Running   0          5s
mydep-54f7ff7d6d-gccr6   1/1      Running   0          2s
mydep-6766777fff-9r2zn   1/1      Running   0          5h
```



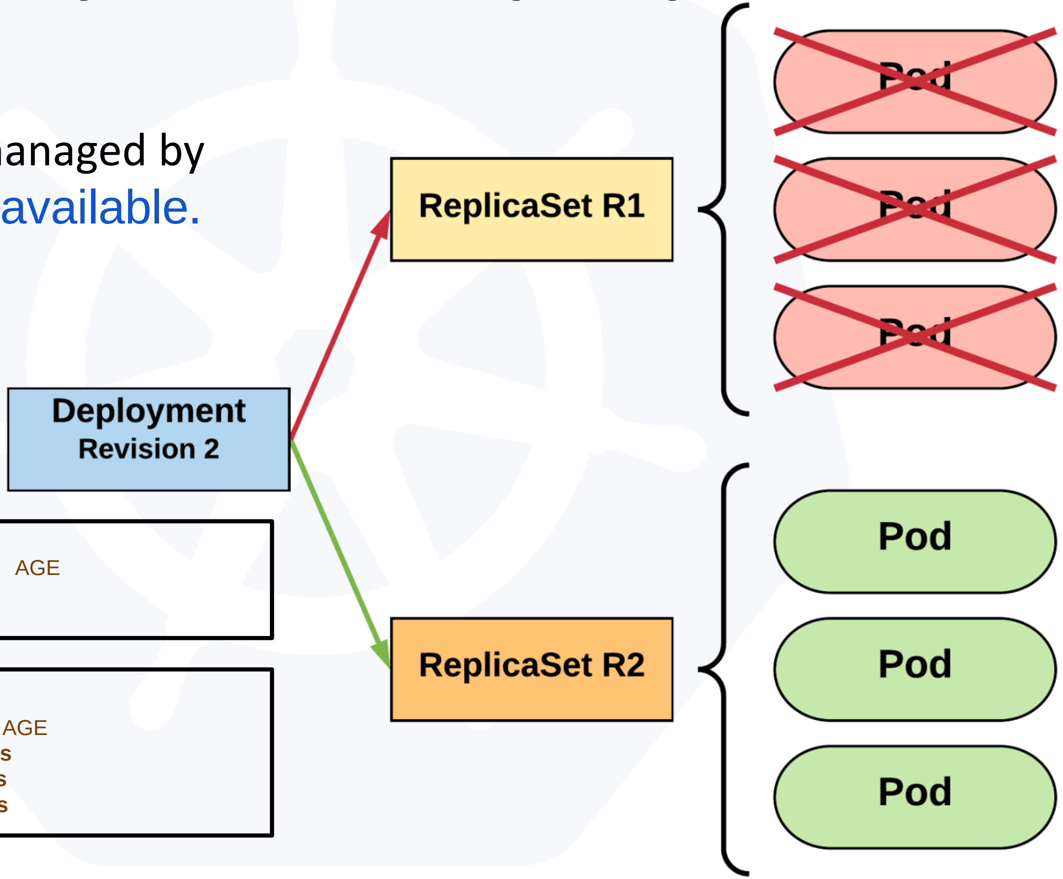
# RollingUpdate Deployment

Phase out of old Pods managed by
maxSurge and maxUnavailable.

**R1 pod-template-hash:**
676677fff
**R2 pod-template-hash:**
**54f7ff7d6d**

```
$ kubectl get replicaset
NAME                DESIRED   CURRENT   READY    AGE
mydep-54f7ff7d6d       3         3        3       13s
mydep-6766777fff       0         0        0       5h
```

```
$ kubectl get pods
NAME                        READY    STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll      1/1      Running   0          10s
mydep-54f7ff7d6d-cqvlq      1/1      Running   0          8s
mydep-54f7ff7d6d-gccr6      1/1      Running   0          5s
```



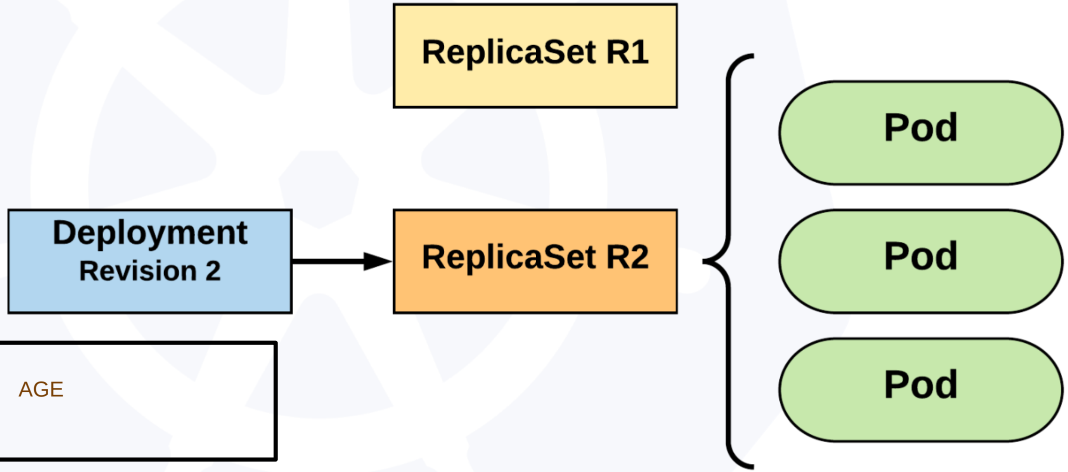
# RollingUpdate Deployment

Updated to new deployment revision completed.

**R1 pod-template-hash:**
676677fff
**R2 pod-template-hash:**
**54f7ff7d6d**

```
$ kubectl get replicaset
NAME                 DESIRED   CURRENT   READY    AGE
mydep-54f7ff7d6d         3         3         3       15s
mydep-6766777fff        0         0         0       5h
```

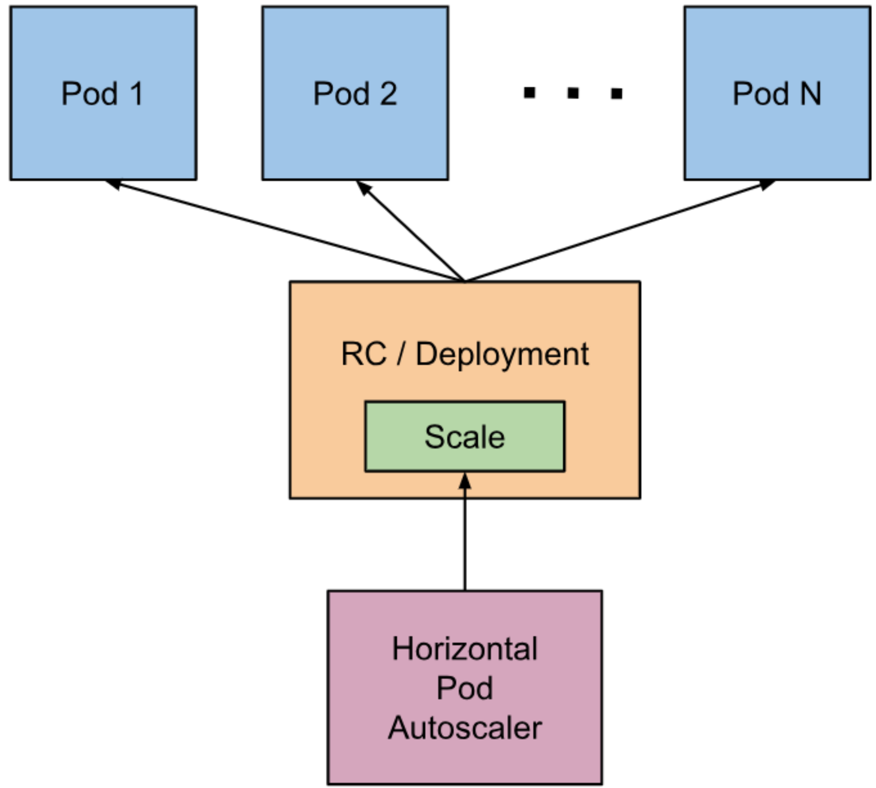
```
$ kubectl get pods
NAME                          READY    STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll    1/1       Running   0          12s
mydep-54f7ff7d6d-cqvlq    1/1       Running   0          10s
mydep-54f7ff7d6d-gccr6    1/1       Running   0          7s
```



# Horizontal Pod Autoscaling (HPA)

# Horizontal Pod Autoscaling (HPA)

- The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU utilization or other metrics
- The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller




https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

# Lab 4 – Auto Scale Pods

1. Deploy your app
2. Add load on your app
3. Number of Pods will go up, automatically, based on load
4. Reduce load on your app
5. Number of Pods will go down, automatically, based on load

# Monitoring Kubernetes

# Monitoring Challenges for Kubernetes

1. Many, smaller pieces to monitor
2. Keeping track of pods and containers
3. Health of deployed applications and containers
4. Availability of resources in a deployment/cluster

# Monitoring Kubernetes – Data Sources

1. **K8 Hosts** Running Kubelet
2. **K8 Process** i.e. Kubelet Metrics : give you details on a Kubernetes node and the jobs it's running
   1. Metrics for apiserver
   2. kube-scheduler
   3. kube-controller-manager
3. Kubelet's Built-in **cAdvisor** : collects, aggregates, processes, and exports metrics for your running containers
4. **Kube-state Metrics** : gives you information at the cluster level

# Metrics Server

- Cluster-wide aggregator of resource usage data
- Metric server collects metrics from the Summary API, exposed by kubelet on each node
  - 5000 nodes clusters with 30 pods per node, supported by kubernetes 1.6
  - To collect 10 metrics from each pod per node
    - 10 x 5000 x 30 / 60 = 25000 metrics per second by average
  - This required a new server instead of API service, hence metrics server was created

# Lab 5 – Monitor K8

1. Uses Prometheus & Grafana
2. Monitor Nodes
3. Monitor Pods
4. Monitor Deployments
5. And Much More

# Storage Options

# Storage

- Many workloads require exchanging data between containers, or persisting some form of data
- 4 types of storage
  1. Volumes
  2. Persistent Volumes
  3. Persistent Volume Claims
  4. Storage Classes

# Persistent Volume

- A **PersistentVolume** (PV) represents a storage resource
- PVs are a **cluster wide resource** linked to a backing storage provider: NFS, GCEPersistentDisk, RBD etc.
- Generally provisioned by an administrator
- Their lifecycle is handled independently from a pod
- **CANNOT** be attached to a Pod directly. Relies on a **PersistentVolumeClaim**

# Persistent Volume Cont.

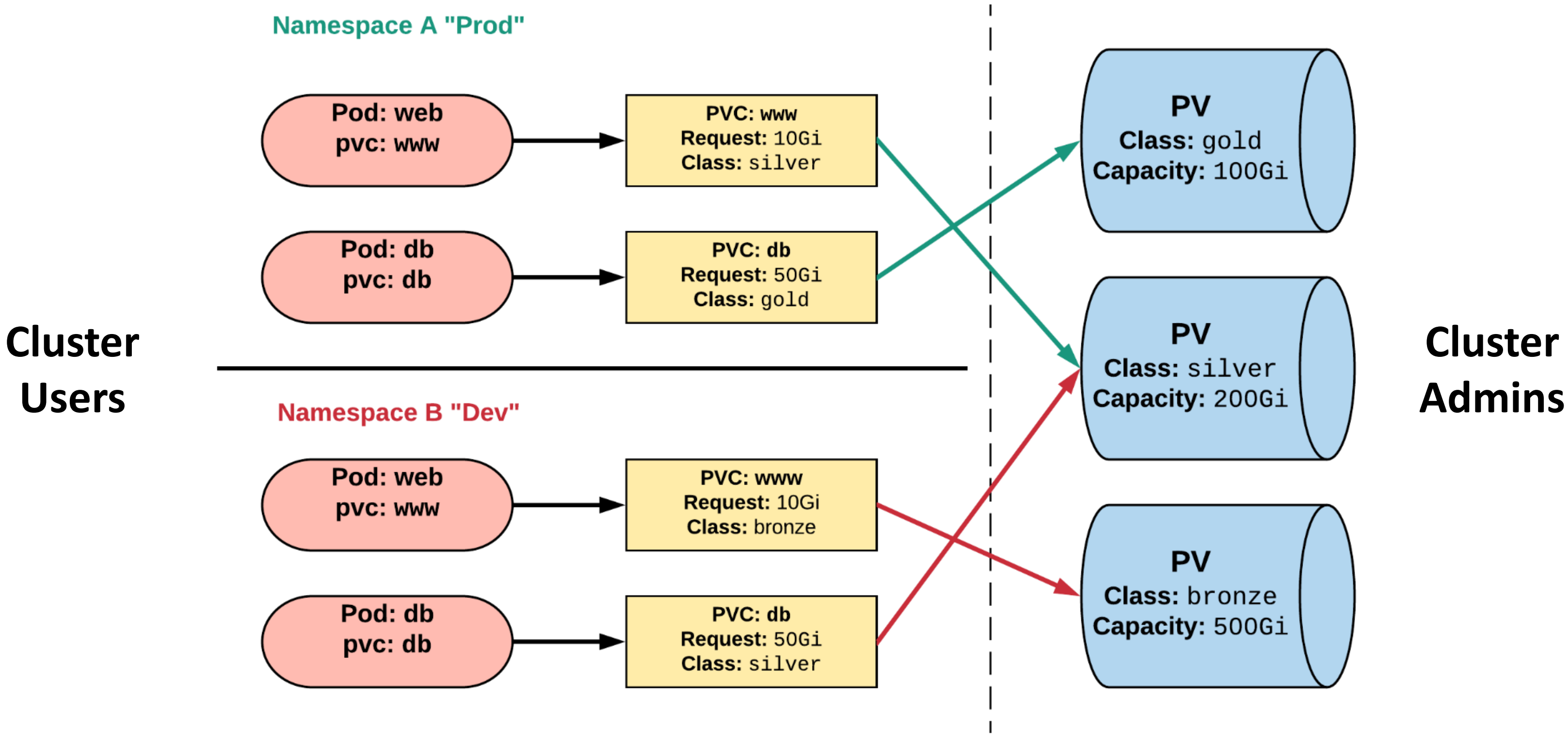
- You define the capacity, whether you want a filesystem or a block device, and the **access mode**
  - RWO - only a single pod will be able to (through a PVC) mount this.
  - ROM - many pods can mount this, but none can write.
  - RWM - many pods can mount and write.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfsserver
spec:
  capacity:
    storage: 50Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Delete
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /exports
    server: 172.22.0.42
```

# Persistent Volume Claims

- A **PersistentVolumeClaim** (PVC) is a **namespace** request for storage.
- Satisfies a set of requirements instead of mapping to a storage resource directly.
- Ensures that an application's '*claim*' for storage is portable across numerous backends or providers.
- PVCs can be named the same to make things consistent but point to different storage classes

# Persistent Volume & Claims



# Persistent Volume Phases

| Available | Bound | Released | Failed |
|---|---|---|---|
| PV is ready and available to be consumed. | The PV has been bound to a claim. | The binding PVC has been deleted, and the PV is pending reclamation. | An error has been encountered attempting to reclaim the PV. |

# Lab 6 – Running Stateful Apps on K8

# Kubernetes Networking

# Kubernetes Networking

- **Pod Network**
  - Cluster-wide network used for pod-to-pod communication managed by a CNI (***Container Network Interface***) plugin.
- **Service Network**
  - Cluster-wide range of **Virtual IPs** managed by **kube-proxy** for service discovery.

# Kubernetes Networking

- Unlike Docker, every pod gets its own cluster wide unique IP, and makes use of the CNI plugin

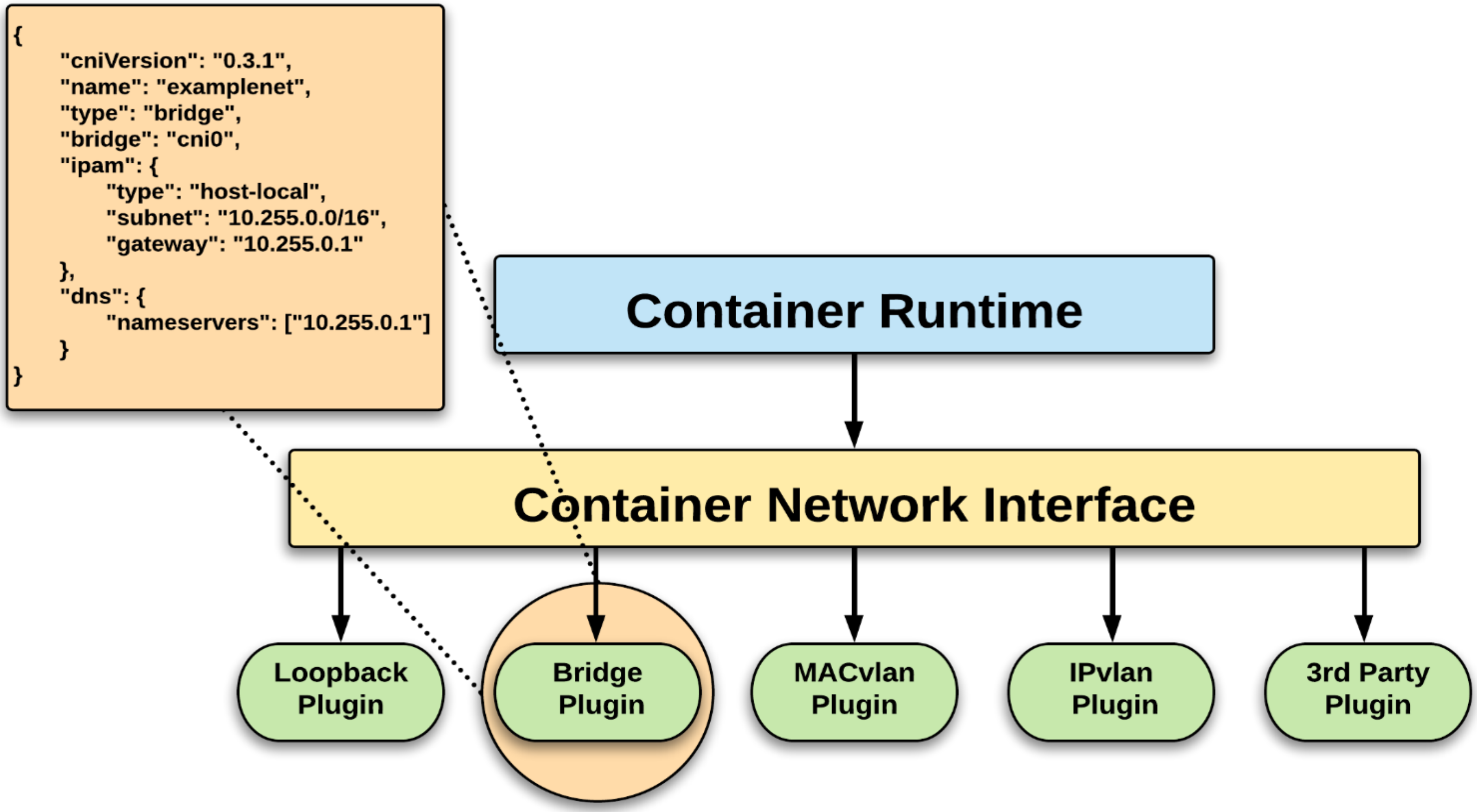
- Services are a separate range of static non-routable virtual IPs that are used like an internal LB or static IP

- Service IPs are special and can’t be treated like a normal IP, they are a ‘mapping’ stored and managed by kube-proxy
    - Pod IPs are pingable
    - Service IPs are not

# Container Network Interface (CNI)

- Pod networking within Kubernetes is plumbed via the Container Network Interface (CNI).
- Functions as an interface between the container runtime and a **network implementation plugin**.
- CNCF Project
- Uses a simple JSON Schema.
- CNI runtime focuses solely on container lifecycle connectivity

# CNI Overview



# CNI Plugins

- Amazon ECS
- Calico
- Cillium
- Contiv
- Contrail
- Flannel
- GCE
- kube-router
- Multus
- OpenVSwitch
- Romana
- Weave

# Fundamental Networking Rules

- All containers within a pod can communicate with each other unimpeded.
- All Pods can communicate with all other Pods without NAT.
- All nodes can communicate with all Pods (and vice-versa) without NAT.
- The IP that a Pod sees itself as is the same IP that others see it as.
- Pods are given a cluster unique IP for the duration of its lifecycle.

# Fundamental Networking Rules - II

- **Container-to-Container**
  - Containers within a pod exist within the **same network namespace** and share an IP.
  - Enables intrapod communication over *localhost*.
- **Pod-to-Pod**
  - Allocated **cluster unique IP** for the duration of its life cycle.
  - Pods themselves are fundamentally ephemeral.

# Fundamental Networking Rules - III

- **Pod-to-Service**
  - managed by **kube-proxy** and given a **persistent cluster unique IP**
  - exists beyond a Pod's lifecycle
  - kubernetes creates a cluster-wide IP that can map to n-number of pods
- **External-to-Service**
  - Handled by **kube-proxy.**
  - Works in cooperation with a cloud provider or other external entity (load balancer).

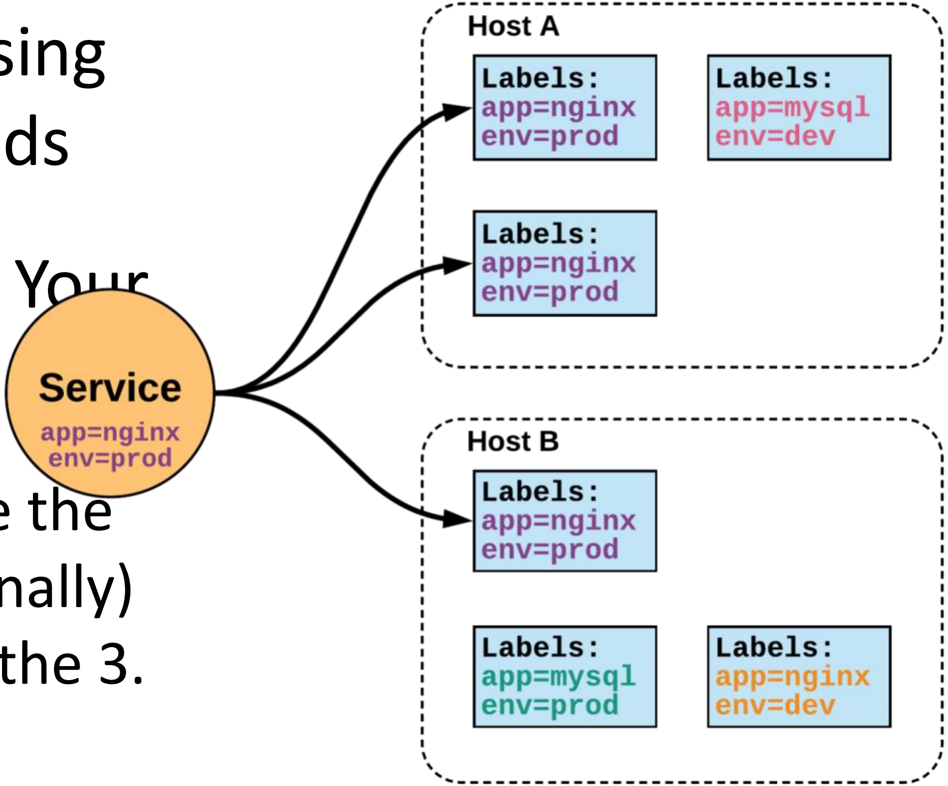
# Kubernetes Services

# Services

- A kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them
- Unified method of accessing the exposed workloads of Pods
- Durable resource
- Uses kube-proxy to provide simple load balancing

# Services (Proxy)

- Unified Method of Accessing Exposed Workloads of Pods
- Internal Load Balancer to Your Pod(s)
  - Create a service, reference the pods, for e.g. 3, and (internally) it will load balance across the 3.



# Service Types

- 4 Major Publishing Service Types
    1. ClusterIP (default)
    2. NodePort
    3. LoadBalancer
    4. Ingress

# ClusterIP Service

- ClusterIP is an internal LB for your application
- Exposes a service on a strictly cluster internal virtual IP

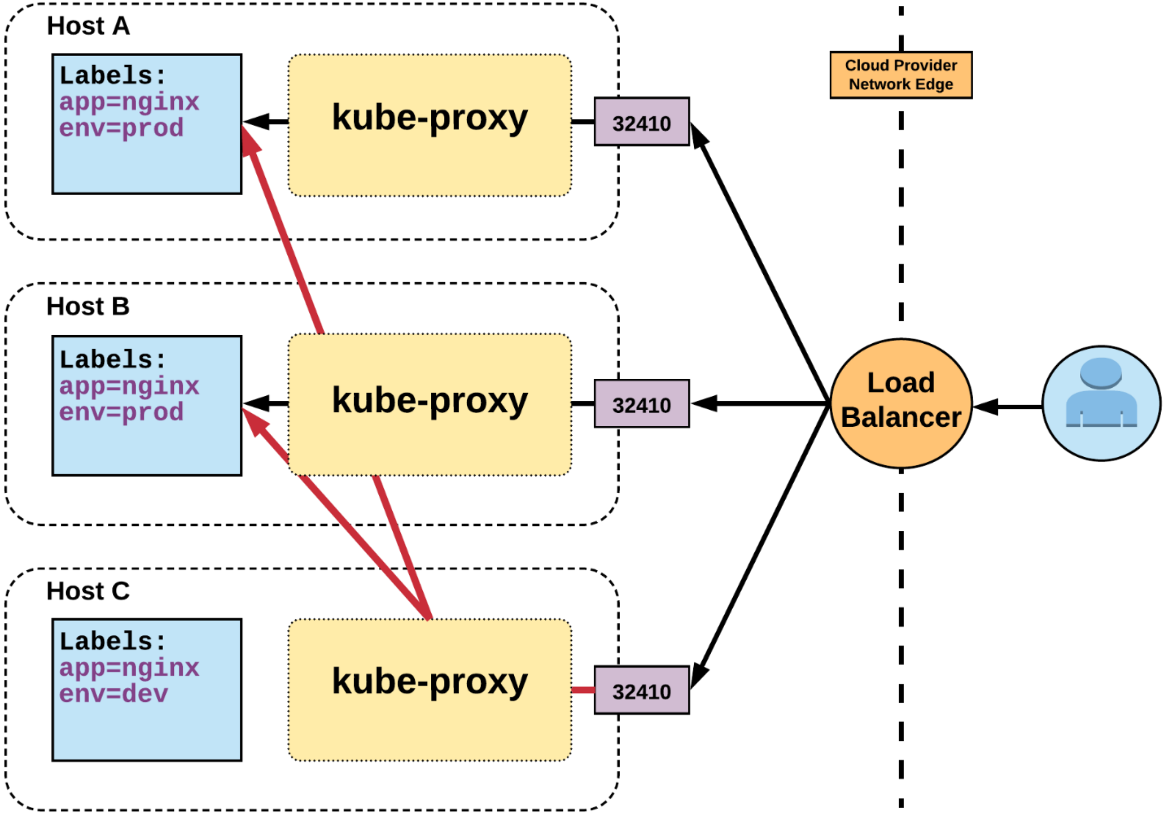
# NodePort Service

- Exposes the service on each Node’s IP at a static port (the NodePort)
- NodePort behaves just like ClusterIP, except it also exposes the service on a (random or specified) port on every Node in your cluster.
- Port can either be statically defined, or dynamically taken from a range between 30000-32767.
- You’ll be able to contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>

# LoadBalancer Service

- Exposes the service externally using a cloud provider’s load balancer
- NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

# LoadBalancer Service Cont.




Name: example-prod
Selector: app=nginx,env=prod
Type: LoadBalancer
IP: 10.96.28.176
LoadBalancer
Ingress: 172.17.18.43
Port: <unset> 80/TCP
TargetPort: 80/TCP
NodePort: <unset> 32410/TCP
Endpoints: 10.255.16.3:80,
10.255.16.4:80

# ExternalName Service

- Maps the service to the contents of the externalName field (e.g. app1.prod.example.com)
- **ExternalName** is used to reference endpoints **OUTSIDE** the cluster.
- Creates an internal **CNAME** DNS entry that aliases another.

# Ingress

- An API object that manages external access to the services in a cluster, typically HTTP
- Traffic routing is controlled by rules defined on the ingress resource
- An ingress does not expose arbitrary ports or protocols
- In order for the ingress resource to work, the cluster must have an ingress controller running

# Kubernetes Best Practices

- Configure Health Checks
- Use Multiple Clusters
- Begin with One Container in One Pod
- Keep Your Images Small

# Path Forward

- Start with a managed service
- Docker Documentation (Docker.com)
- K8 Documentation (Kuberntes.IO)
- Automate Image Creation, Storage and Deployment
- Create and Use Official Images Repository
- Use Docker Swarm for Small Workloads
- Start with Stateless Applications

This concludes Chapter 3

**Container Orchestration with Kubernetes**