DEVOPS – CI/CD KUBERNETES

23-11-2020 to 27-11-2020

Venkatesh Reddy Madduri && ShankarPrasad

Agenda

- Kubernetes Introduction
- Features
- Architecture
- Installation
- Kubernetes components in depth

Streamline Deployments using Docker

- Compose the application into Docker Images
- Break the application components into individual containers
- Split the data that’s shared between services into volumes
- Separate responsibilities so that each containers runs only one component/executable
- Store the changeable data (configurations, logs) as Volumes so that they are mounted on various containers

Modern development’s primary focus is often based around three central concepts:

- efficiency
- reliability
- repeatability

Why do we want a Container Orchestration System?

Imagine that you had to run hundreds of containers. You will need from a management angle to make sure that the cluster is up and running and have necessary features like:

- Health Checks on the Containers
- Launching a fixed set of Containers for a particular Docker image
- Scaling the number of Containers up and down depending on the load
- Performing rolling update of software across containers

Introduction

- Kubernetes is a container management technology developed in Google lab to manage containerized applications in different kind of environments such as physical, virtual, and cloud infrastructure.
- This is also known as the enhanced version of Borg which was developed at Google to manage both long running processes and batch jobs, which was earlier handled by separate systems.
- Kubernetes is also known as ‘k8s’. This word comes from the Greek language, which means a pilot or helmsman.
- Kubernetes in an open source container management designed by Google in 2014 and donated to Cloud Native Computing Foundation (CNCF).
- Start, stop, update, and manage a cluster of machines running containers in a consistent and maintainable way.
- K8s provides the tooling to manage the when, where, and how many of the application stack and its components.
- K8s also allows finer control of resource usage, such as CPU, memory, and disk space across our infrastructure.

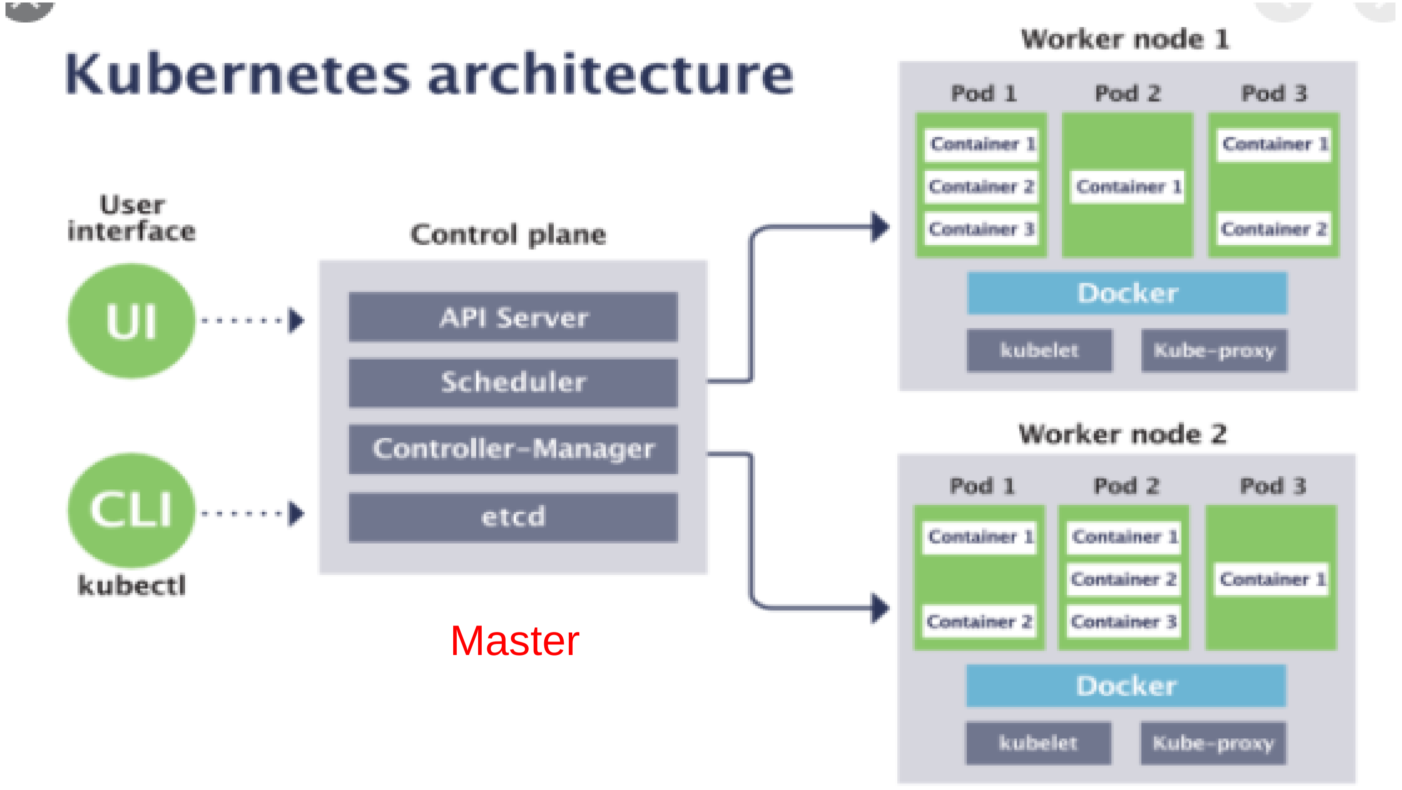
Features

Following are some of the important features of Kubernetes.

- Containerized infrastructure
- Application-centric management
- Auto-scalable infrastructure
- Environment consistency across development testing and production
- Loosely coupled infrastructure, where each component can act as a separate unit
- Higher density of resource utilization
- Predictable infrastructure which is going to be created

One of the key components of Kubernetes is, it can run application on clusters of physical and virtual machine infrastructure. It also has the capability to run applications on cloud. It helps in moving from **host-centric** infrastructure to **container-centric** infrastructure.

Architecture



Master

Architecture

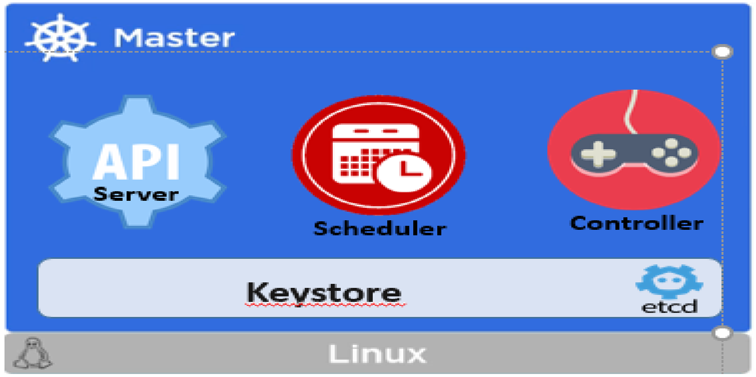
Kubernetes actually follows client-server architecture. It consists of the two main components:

1. Master Node (Control Plane)
2. Slave / Worker Node

Master Node or Kubernetes Control Plane

- ➢ The master coordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.
- ➢ The following are main components of master node

  - ❖ API Server
  - ❖ Scheduler
  - ❖ Controller Manager
  - ❖ ETCD



Architecture – Kubernetes Master

API Server

- The Kubernetes API lets you query and manipulate the **state of objects** and **all the operations** in Kubernetes.
- **Kubeconfig** is a package along with the server side tools that can be used for communication. It exposes Kubernetes API.
- The Kubernetes API server receives the REST (**Representational state transfer**) commands which are sent by the user. After receiving, it validates the REST requests, process, and then executes them.
- After the execution of REST commands, the resulting state of a cluster is saved in **'etcd'** as a distributed **key-value** store.

Scheduler

- This is one of the key components of Kubernetes master.
- It is a service in master responsible for distributing the **workload**.
- It is responsible for tracking **utilization of working** load on cluster nodes and then placing the workload on which resources are available and accept the workload.
- The scheduler is responsible for workload utilization and allocating **pod** to new node.

Architecture

Controller Manager

- It is a **daemon** that executes in the **non-terminating** control loops and is responsible for collecting and sending information to API server..
- The controllers in a master node perform a task and manage the state of the cluster.
- The controller manager executes the various types of controllers for handling the **nodes, endpoints, etc**. The key controllers are **replication controller, endpoint controller, namespace controller, and service account controller**.

ETCD

- Acts as the memory for the Brian(master)
- It stores the configuration information i.e **Cluster state**
- It is a high availability key value store that can be distributed among multiple nodes
- It is accessible only by Kubernetes API server

Architecture

Worker/Slave node

- The Worker node in a Kubernetes is also known as **minions**.
- A worker node is a physical machine that executes the applications using pods. It contains all the essential services which allow a user to assign the resources to the scheduled containers.

Following are the different components which are presents in the Worker or slave node:

Kubelet

- This component is an agent service that executes on each worker node in a cluster.
- It ensures that the pods and their containers are running smoothly.
- Every kubelet in each worker node communicates with the master node. It also starts, stops, and maintains the containers which are organized into pods directly by the master node.

Kube-proxy

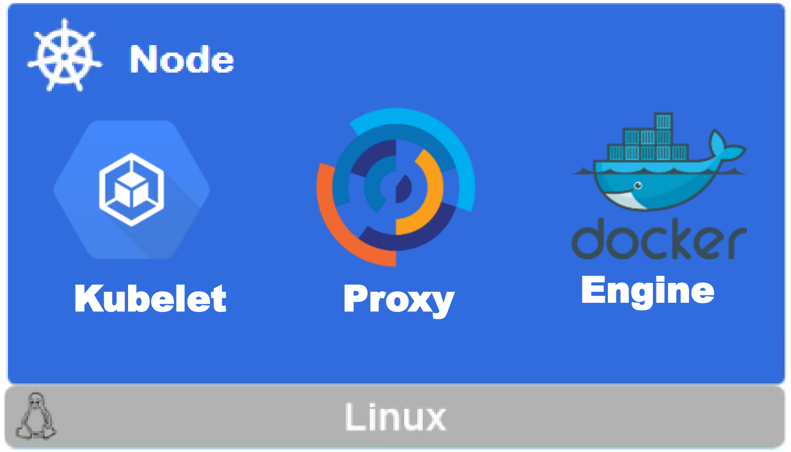
- It is a proxy service of Kubernetes, which is executed simply on each worker node in the cluster.
- The main aim of this component is request forwarding.
- Each node interacts with the Kubernetes services through **Kube-proxy**

Pods

- A Prod represents set of running containers on node

Kubernetes Nodes

Kubernetes Nodes(Minions)



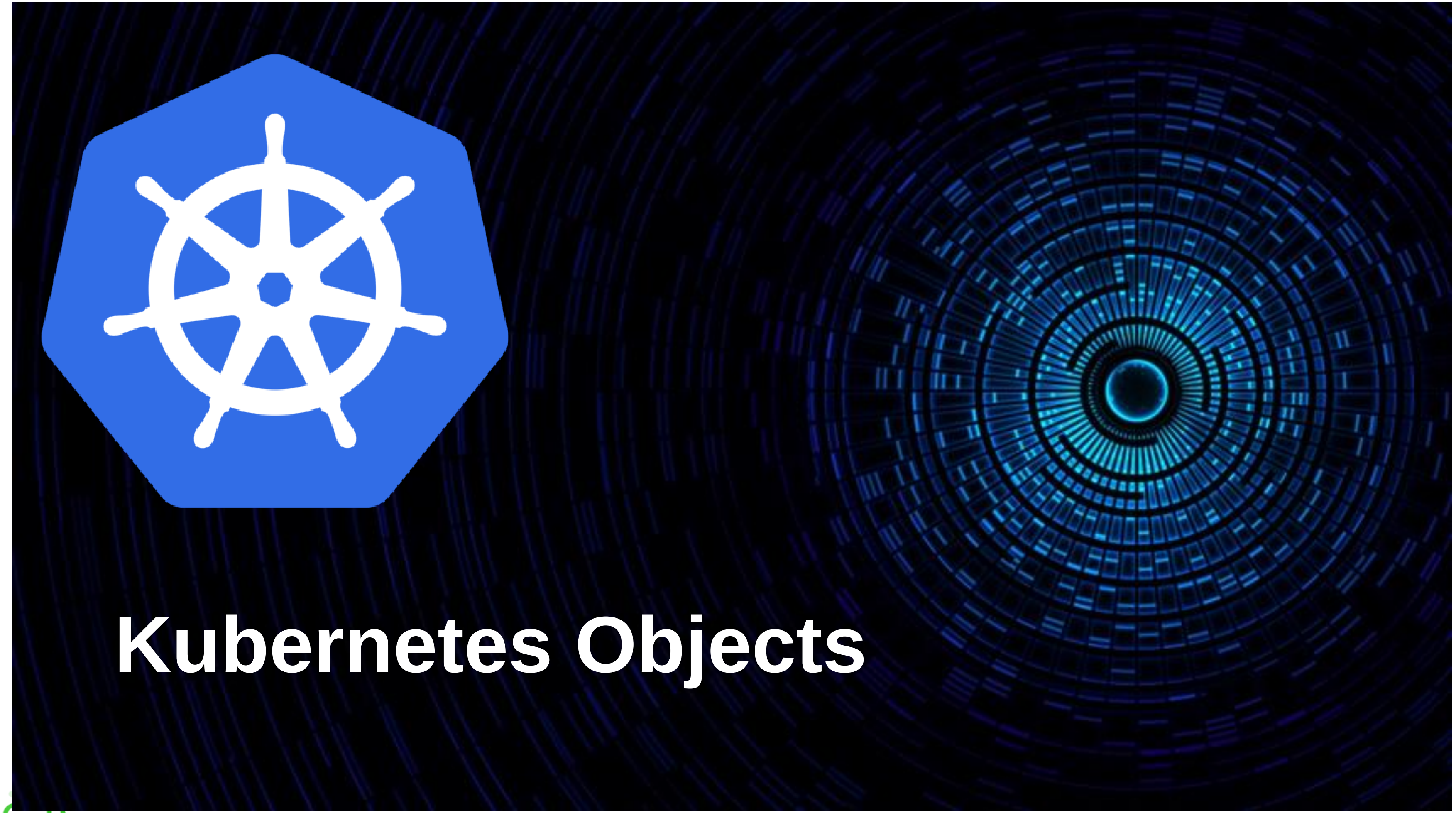
- Nodes are the machines in which containers run
- Node consists of these components:
    - kubelet
    - kube-proxy
    - Container engine
- These components, receive workloads to execute, and update the cluster on their status
- To run and manage a container's lifecycle, we need a container runtime on the worker node. Some examples of container runtimes are:
    - containerd
    - rkt
    - lxd

kubelet

- The main kubernetes agent
- Interacts with the API server on the Master
- Registers the Node with the cluster
- Make sure that containers are running in a Pod
- Updates the workloads that have been invoked by the scheduler:
  - Mount Volumes
  - Run containers
  - Report the status of the node
  - Run container liveness probe

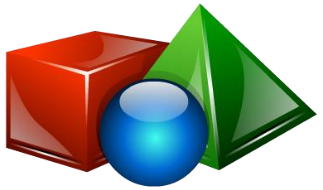
Kube-proxy

- kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.
- It maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.
- It used to forward the traffic and any filtering if configured.



Kubernetes Objects

Kubernetes Objects



- Kubernetes uses Objects to represent the state of your 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
- Once you create the object, the Kubernetes system will constantly work to ensure that object exists and maintain cluster's **desired state**
- Every Kubernetes object includes two nested fields that govern the object's configuration: the **object *spec*** and the **object *status***
- The ***spec***, which we provide, describes your *desired state* for the object–the characteristics that you want the object to have.
- The ***status*** describes the *actual state* of the object, and is supplied and updated by the Kubernetes system.
- All objects in the are identified by a Unique Name and a UID.

Kubernetes Objects

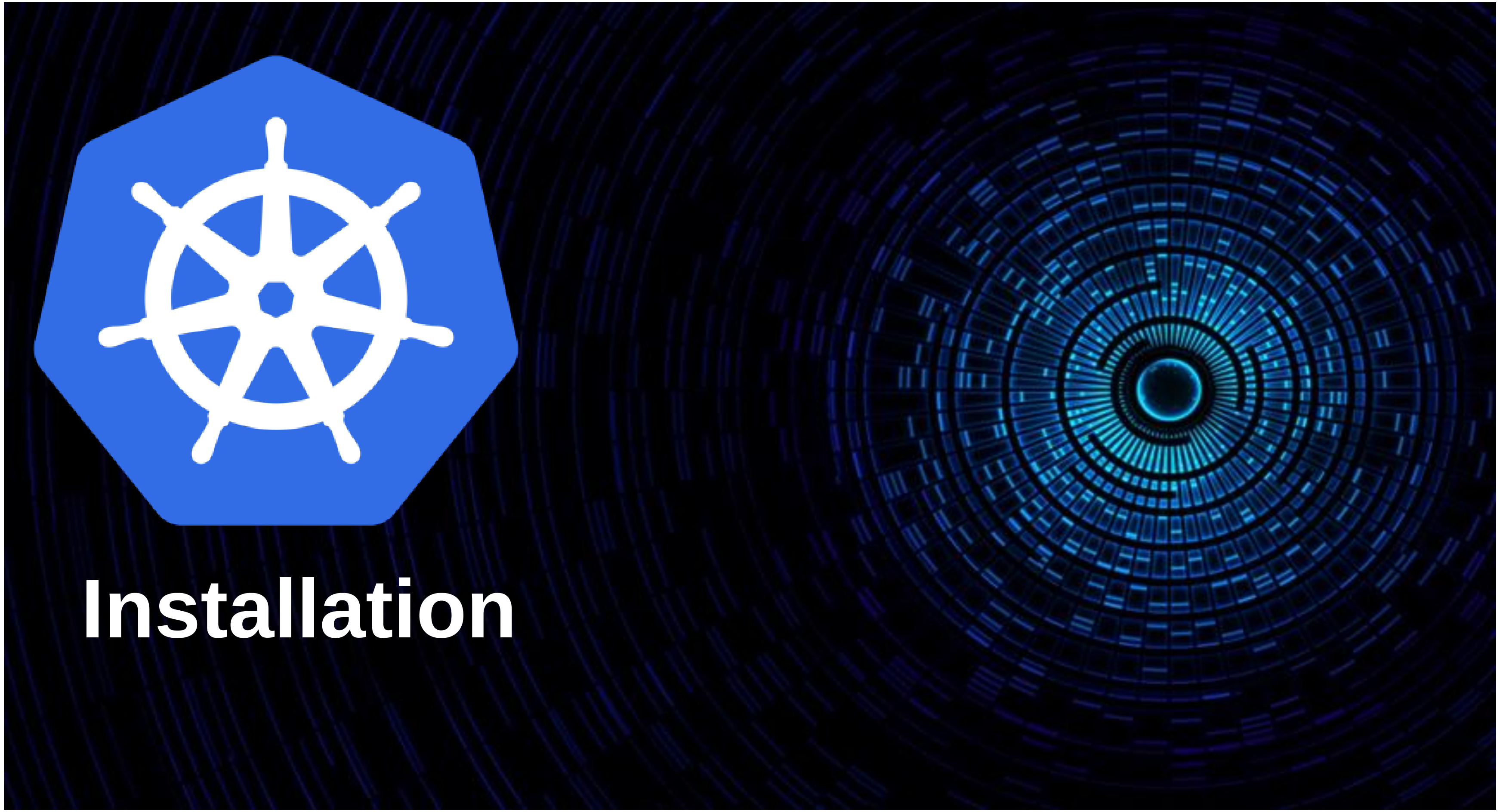
- The basic Kubernetes objects include:
  - Pod
  - Service
  - Volume
  - Namespace
- In addition, Kubernetes contains a number of higher-level abstractions called Controllers. Controllers build upon the basic objects, and provide additional functionality:
  - ReplicaSet
  - Deployment
  - StatefulSet
  - DaemonSet
  - Job

Kubernetes Objects Management

- The **kubectl** command-line tool supports several different ways to create and manage Kubernetes objects

Management technique	**Operates on**	**Recommended environment**
Imperative commands	Live objects	Development projects
Declarative object configuration	Individual files (yaml/json)	Production

- **Declarative** is about describing what you're trying to achieve, without instructing how to do it
- **Imperative** explicitly tells "how" to accomplish it



Installation

Kubernetes Configuration

All-in-One Single-Node Installation

- With all-in-one, all the master and worker components are installed on a single node. This is very useful for learning, development, and testing. This type should not be used in production. Minikube is one such example, and we are going to explore it in future chapters.

Single-Node etcd, Single-Master, and Multi-Worker Installation

- In this setup, we have a single master node, which also runs a single-node etcd instance. Multiple worker nodes are connected to the master node.

Single-Node etcd, Multi-Master, and Multi-Worker Installation

- In this setup, we have multiple master nodes, which work in an HA mode, but we have a single-node etcd instance. Multiple worker nodes are connected to the master nodes.

Infrastructure for Kubernetes Installation

- **Localhost Installation -** Minikube
- **On-Premise Installation** - Vagrant, Vmware, KVM
- **Cloud Installation**
    - ✓ Google Kubernetes Engine (GKE)
    - ✓ Azure Container Service (AKS)
    - ✓ Amazon Elastic Container Service for Kubernetes (EKS)
    - ✓ OpenShift
    - ✓ Platform9
    - ✓ IBM Cloud Container Service

Kubernetes Installation Tools/Resources:

- ❑ Kubeadm
- ❑ Kubespray
- ❑ kops

Setting Up a Single-Node Kubernetes Cluster with Minikube (EC2 Ubuntu)

- **Install Docker**

```
$  sudo yum update && yum install docker.io
```

- **Install kubectl**

```
$  curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl &&

chmod +x ./kubectl && sudo mv ./kubectl /usr/local/bin/kubectl
```

- **Install Minikube**

```
$  curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/
```

- **Start Minikube**

```
$  minikube start --vm-driver=none

$  minikube status
```

Master Node Inbound

PROTOCOL	PORT RANGE	SOURCE	PURPOSE
TCP	443	Worker Nodes, API Requests, and End-Users	Kubernetes API server.

Worker Node Inbound

PROTOCOL	PORT RANGE	SOURCE	PURPOSE
TCP	10250	Master Nodes	Worker node Kubelet healthcheck port.
TCP	30000-32767	External Application Consumers	Default port range for external service ports. Typically, these ports would need to be exposed to external load-balancers, or other external consumers of the application itself.
TCP	ALL	Master & Worker Nodes	Intra-cluster communication (unnecessary if `vxlan` is used for networking)
UDP	8285	Worker Nodes	flannel overlay network - *udp backend*. This is the default network configuration (only required if using flannel)
UDP	8472	Worker Nodes	flannel overlay network - *vxlan backend* (only required if using flannel)
TCP	179	Worker Nodes	Calico BGP network (only required if the BGP backend is used)

etcd Node Inbound

PROTOCOL	PORT RANGE	SOURCE	PURPOSE
TCP	2379-2380	Master Nodes	etcd server client API
TCP	2379-2380	Worker Nodes	etcd server client API (only required if using flannel or Calico).

Kubectl Usage

$ kubectl [COMMAND] [TYPE] [NAME] [flags]

COMMAND: Specifies the operation that you want to perform on one or more resources, for example create, apply, get, describe, delete, exec …

TYPE: Specifies the resource type i.e nodes, pods, services, rc, rs …

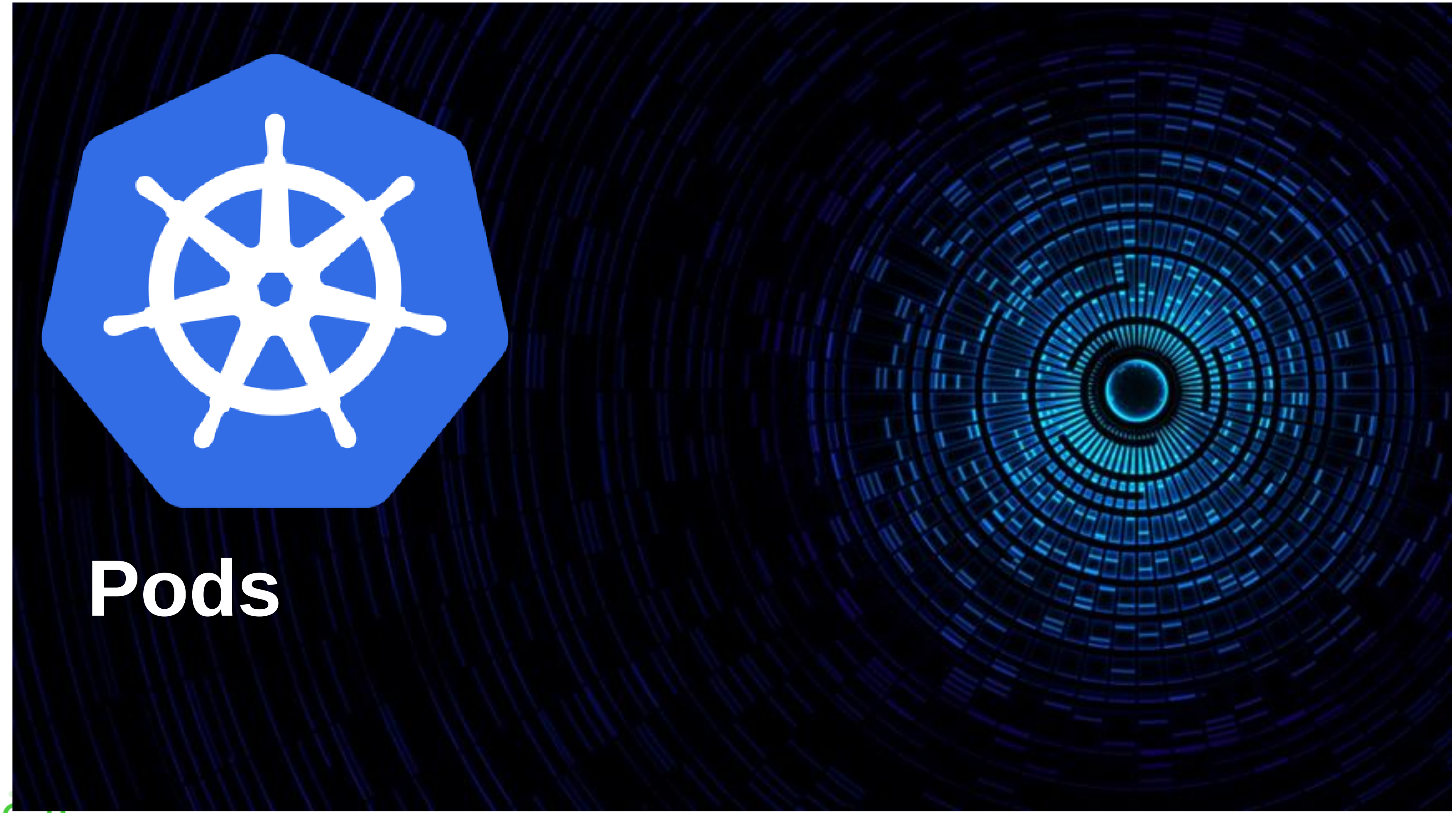
NAME: Specifies the name of the resource

flags: Optional arguments

```
$ kubectl version
```

```
$ kubectl get node
```

```
$ kubectl describe nodes <nodename>
```

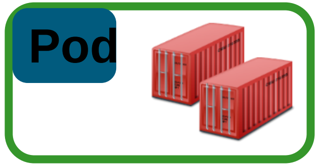


Pods

POD

- A Pod is the smallest and simplest unit in the Kubernetes object model that you create or deploy.
- A Pod represents a running process on your cluster.
- A Pod encapsulates an application container, storage resources, a unique network IP, and options that govern how the container(s) should run
- Kubernetes manages the Pods rather than the containers directly.
- Pods in a Kubernetes cluster can be used in two main ways:
  - Pods that run a single container - common use-case
  - Pods that run multiple containers that need to work together
- Each Pod is assigned a **unique IP** address.
- Every container in a Pod shares the IP address and network ports.
- Containers inside a Pod can communicate with one another using localhost.
- All containers in the Pod can access the shared volumes, allowing those containers to share data.
- Kubernetes scales pods and not containers

Understanding Pods



- When a Pod gets created, it is scheduled to run on a Node in your cluster.
- The Pod remains on that Node until the process is terminated, the pod object is deleted, the pod is evicted for lack of resources, or the Node fails.
- If a Pod is scheduled to a Node that fails, or if the scheduling operation itself fails, the Pod is deleted
- If a node dies, the pods scheduled to that node are scheduled for deletion, after a timeout period.
- A given pod (UID) is not “rescheduled” to a new node; instead, it will be replaced by an identical pod, with even the same name if desired, but with a new UID
- Volumes in a pod will exists as long as that pod (with that UID) exists. If that pod is deleted for any reason, volume is also destroyed and created as new on new pod
- Kubernetes uses a Controller, that handles the work of managing the Pod instances.
- A Controller can create and manage multiple Pods, handling replication, rollout and providing self-healing capabilities

Kubernetes YAML Basics

API: version

Kind: object we are trying to achieve

metadata: name and label

spec: set of data which defines the desired state for the resource

-- properties you can set for a Container:

name

image

command

args

workingDir

ports

env

resources

volumeMounts

livenessProbe

readinessProbe

livecycle

terminationMessagePath

imagePullPolicy - Always

securityContext

stdin

stdinOnce

tty

Pod Single Container Pod – Example: pod1.yml

```
kind: Pod                              # Object Type
apiVersion: v1                         # API version
metadata:                              # Set of data which describes the Object
  name: testpod                        # Name of the Object
spec:                                  # Data which describes the state of the Object
  containers:                          # Data which describes the Container details
    - name: c00                        # Name of the Container
      image: ubuntu                    # Base Image which is used to create Container
      command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
  restartPolicy: Never                 # Defaults to Always
```

- **Always** means that the container will be restarted even if it exited with a zero exit code
- **OnFailure** means that the container will only be restarted if it exited with a non-zero exit code
- **Never** means that the container will not be restarted regardless of why it exited.

- Create or update an Object

$ **kubectl create -f pod1.yml** (or)

$ **kubectl apply -f pod1.yml**

- Get the list of the pod objects available

$ **kubectl get pods**

$ **kubectl get pods -o wide**

$ **kubectl get pods --all-namespaces**

- Get the details of the pod object

$ **kubectl describe pod <podname>**

- Get running logs from the container inside the pod object

$ **kubectl logs -f <podname>**

$ **kubectl logs -f <podname> -c <containername>**

Pod Annotate - Example: podannotate.yml

Annotations allow you to add non-identifying metadata to Kubernetes objects. Examples include phone numbers of persons responsible for the object or tool information for debugging purposes. In short, annotations can hold any kind of information that is useful and can provide context to DevOps teams.

```
kind: Pod
apiVersion: v1
metadata:
  name: testpod2
  annotations:                                # comments to describe the Object usage
    description: our first testpod
    contact : Venkat
    cell: 414.341.5643
spec:
  containers:
    - name: c00
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

$ **kubectl apply -f podannotate.yml**

- Run OS commands in an existing pod (1 container pod)

$ **kubectl exec <podname> -- <OScmd>**

$ **kubectl exec testpod2 -- hostname -i**

- Run OS commands in an existing container(multi container pod)

$ **kubectl exec <podname> -c <containername> -- <OScmd>**

- Attach to the running container interactively

$ **kubectl exec <podname> -i -t -- /bin/bash (or)**

$ **kubectl attach <podname> -i**

- Delete a Pod

$ **kubectl delete pods <podname> (or)**

$ **kubectl delete -f <YAML>**

Pod Multi-container Pod - Example: podmulcont.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: testpod3
spec:
  containers:
    - name: c00
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
    - name: c01
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo Hello-Students; sleep 5 ; done"]
```

```
$ kubectl apply -f podmulcont.xml
```

Labels

- **Labels** are the mechanism you use to organize Kubernetes objects
- A label is a key-value pair without any predefined meaning that can be attached to the Objects
- Labels are similar to Tags in AWS or Git where you use a name to quick reference
- So you’re free to choose labels as you need it to refer an environment which is used for Dev or Testing or Production, refer an product group like DepartmentX, DepartmentY
- Multiple labels can be added to a single object

Pod Labels – Example: podlabels.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: labelspod
  labels:                                    # Specifies the Label details under it
    <Keyname1>: <value>
    <Keyname2>: <value>
spec:
    containers:
       - name: c00
         image: ubuntu
         command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

$ **kubectl apply -f podlabels.yml**

- To see list of pods available with details of Labels if any attached to it

$ **kubectl get pods --show-labels**

- Add a label to an existing pod

$ **kubectl label pods <podname> <labelkey>=<value>**

- List pods matching a label

$ **kubectl get pods -l <label>=<value>**

- We can also delete pods based on label selection

$ **kubectl delete pods -l <label>=<value>**

Scaling and Replication

Scaling & Replication

- ➢ Kubernetes was designed to orchestrate multiple containers and replication.
- ➢ Need for multiple containers/replication helps us with these:

Reliability: By having multiple versions of an application, you prevent problems if one or more fails.

Load balancing: Having multiple versions of a container enables you to easily send traffic to different instances to prevent overloading of a single instance or node

Scaling: When load does become too much for the number of existing instances, Kubernetes enables you to easily scale up your application, adding additional instances as needed

Rolling updates: updates to a service by replacing pods one-by-one

Replication Controller

- A Replication Controller is a object that enables you to easily create multiple pods, then make sure that that number of pods always exists.
- If a pod created using RC will be automatically replaced if they does crash, fail, deleted or terminated
- Using RC is recommended if you just want to make sure 1 pod is always running, even after system reboots
- You can run the RC with 1 replica & the RC will make sure the pod is always running

RC – Example: rc.yml

```
kind: ReplicationController                # this defines to create the object of replication type
apiVersion: v1
metadata:
  name: replicationcontroller
spec:
  replicas: 2                              # this element defines the desired number of pods
  selector:                                # tells the controller which pods to watch/belong to this Replication Controller
    myname: adam                           # these must match the labels
  template:                                # template element defines a template to launch a new pod
    metadata:
      name: testpod6
      labels:                              # selector values need to match the labels values specified in the pod template
        myname: adam
    spec:
     containers:
       - name: c00
         image: ubuntu
         command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

```
$ kubectl apply -f rc.yml
$ kubectl get pods --show-labels
$ kubectl get pods -l myname=adam
$ kubectl get rc
```

- Delete the pod & RC will recreate it

```
$ kubectl delete pods <podname>
$ kubectl describe rc <replicationcontrollername>
```

- Scale Replicas:

```
$ kubectl scale --replicas=<num>  rc/<replicationcontrollername>    (OR)
$ kubectl scale --replicas=<num> <resourcetype> -l <key>=<value>
$ kubectl scale --replicas=2 rc -l myname=adam
```

- Delete Replicationcontroller:

```
$ kubectl delete rc <replicationcontrollername>
```

Replication Set

- ReplicaSet is the next generation Replication Controller
- The replication controller only supports equality-based selector whereas the replica set supports set-based selector i.e filtering according to set of values
- ReplicaSet rather than the Replication Controller is used by other objects like Deployment

RS – Example: rs.yml

```
kind: ReplicaSet                          # Defines the object to be ReplicaSet
apiVersion: apps/v1                       # Replicaset is not available on v1
metadata:
  name: myreplicaset
spec:
  replicas: 2                             # this element defines the desired number of pods
  selector:                               # tells the controller which pods to watch/belong to this Replication Set
    matchExpressions:                     # these must match the labels
      - {key: myname, operator: In, values: [adam, adamm, aadam]}
      - {key: env, operator: NotIn, values: [production]}
  template:                               # template element defines a template to launch a new pod
    metadata:
      name: testpod7
      labels:                             # selector values need to match the labels values specified in the pod template
        myname: adam
    spec:
     containers:
       - name: c00
         image: ubuntu
         command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

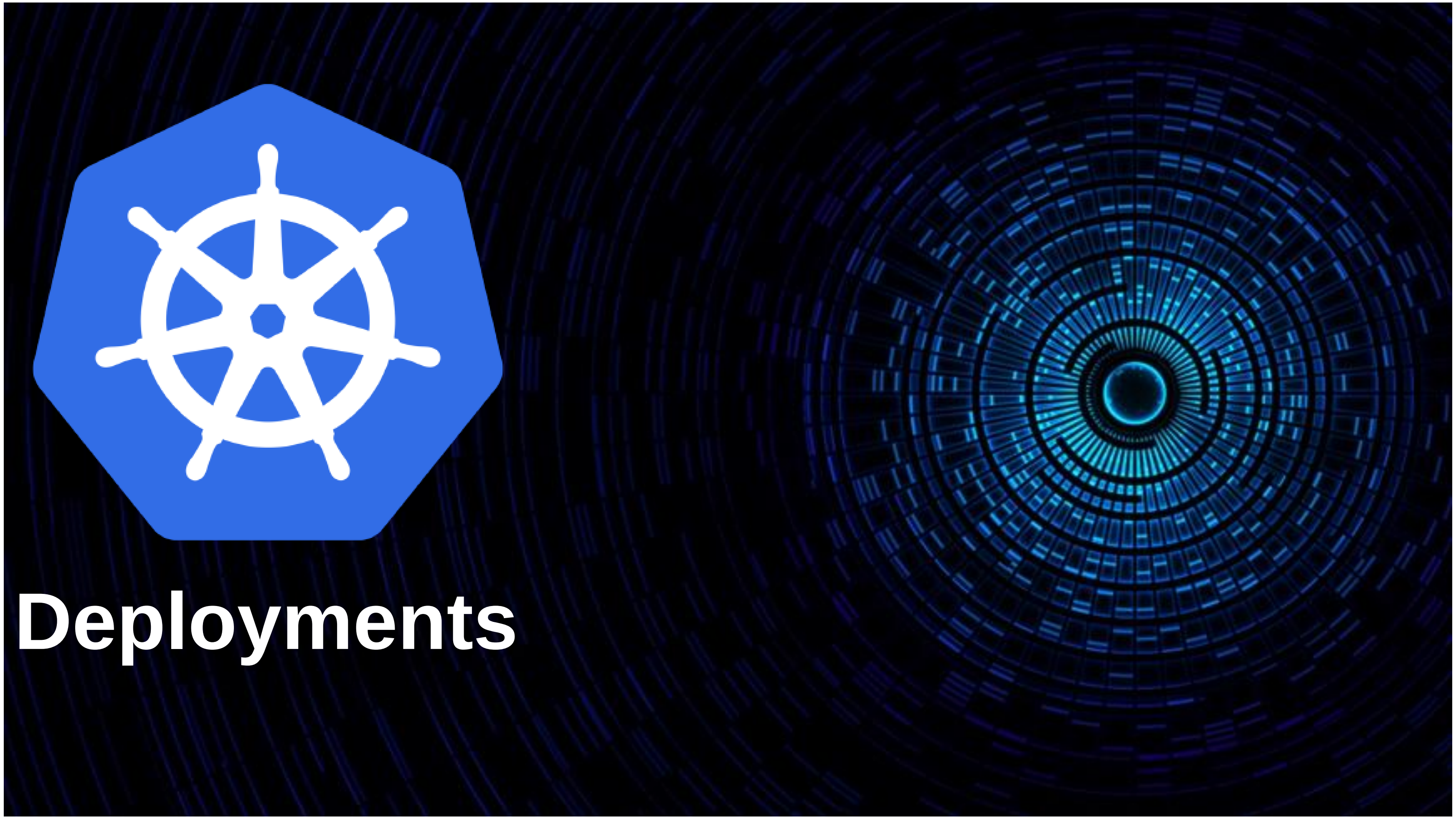
```
$ kubectl apply -f rs.yml
$ kubectl get rs
$ kubectl describe rs <replicasetname>
```

- Scale Replicas:

```
$ kubectl scale --replicas=1  rs/myreplicaset    (OR)
$ kubectl scale --replicas=2 rs -l myname=adam
```

- Delete ReplicaSet:

```
$ kubectl delete rs <replicasetname>
```



Deployments

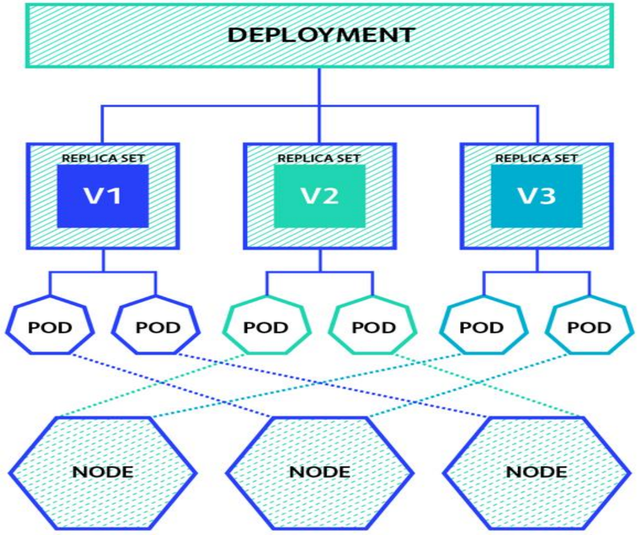
Deployments

- Just using RC & RS might be cumbersome to deploy apps, update or rollback apps in the cluster
- A **Deployment** object acts as a supervisor for pods, giving you fine-grained control over how and when a new pod is rolled out, updated or rolled back to a previous state
- When using Deployment object, we first define the **state** of the App, then k8s master schedules mentioned app instance onto specific individual Nodes
- K8s then monitors, if the Node hosting an instance goes

down or pod is deleted, the Deployment controller replaces it.

- This provides a self-healing mechanism to address

machine failure or maintenance.



The following are typical use cases for Deployments:

- **Create a Deployment to rollout a ReplicaSet**. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
- **Declare the new state of the Pods** by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the **revision** of the Deployment.
- **Rollback to an earlier Deployment revision** if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
- **Scale up the Deployment** to facilitate more load.
- **Pause the Deployment** to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
- **Use the status of the Deployment** as an indicator that a rollout has stuck.
- **Clean up older ReplicaSets** that you don’t need anymore

Services

Services

- When using RC, pods are terminated and created during scaling or replication operations
- When using Deployments, while updating the image version the pods are terminated & new pods take the place of older pods.
- Pods are very dynamic i.e they come & go on the k8s cluster and on any of the available nodes & it would be difficult to access the pods as the pods IP changes once its recreated
- Service Object is an logical bridge between pods & endusers, which provides Virtual IP (VIP) address.
- Service allows clients to reliably connect to the containers running in the pod using the VIP.
- The VIP is not an actual IP connected to a network interface, but its purpose is purely to forward traffic to one or more pods.
- kube-proxy is the one which Keeps the mapping between the VIP and the pods up-to-date, which queries the API server to learn about new services in the cluster.

- Although each Pod has a unique IP address, those IPs are not exposed outside the cluster.
- Services helps to expose the VIP mapped to the pods & allows applications to receive traffic
- Labels are used to select which are the Pods to be put under a Service.
- Creating a Service will create an endpoint to access the pods/Application in it.
- Services can be exposed in different ways by specifying a type in the Service Spec:
  - **ClusterIP** (default) - Exposes VIP only reachable from within the cluster.
  - **NodePort** - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using <**NodeIP**>**:**<**NodePort**>.
  - **LoadBalancer** - Created by cloud providers that will route external traffic to every node on the NodePort (ex: ELB on AWS).
- By default service can run only between ports 30000-32767

Health Checks

Health Checks

- A Pod is considered ready when all of its Containers are ready.
- In order to verify if a container in a pod is **healthy** and **ready** to serve traffic, Kubernetes provides for a range of health checking mechanisms
- **Health checks**, or **probes** are carried out by the kubelet to determine when to restart a container (for livenessProbe) and used by services and deployments to determine if a pod should receive traffic (for readinessProbe).
- For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.
- One use of readiness probes is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.
- For running healthchecks we would use cmds specific to the application.
- If the command succeeds, it **returns 0**, and the kubelet considers the Container to be **alive** and **healthy**. If the command returns a **non-zero value**, the kubelet kills the Container and **restarts** it.

LivenessProbe With CMD – Example: livenessprobecmd.yml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: mylivenessprobe
spec:
  containers:
  - name: liveness
    image: ubuntu
    args:
    - /bin/sh
    - -c
```

```
 - touch /tmp/healthy; sleep 1000
    livenessProbe:
# define the health check
      exec:
        command:
# command to run periodically
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5                          # Wait for the specified time before it runs the first probe
      periodSeconds: 5
# Run the above command every 5 sec
      timeoutSeconds: 30                              # Seconds to timeout if the cmd is not responding
```

$ kubectl apply –f livenessprobecmd.yml

$ kubectl describe pod mylivenessprobe

- Delete the file & kubelet would recreate the container

$ kubectl exec mylivenessprobe -- rm /tmp/healthy

$ kubectl describe pod mylivenessprobe

Readiness check

- While both liveness and readiness check and describe the state of the service, they serve quite a different purpose.
- Liveness describes if the pod has started and everything inside it is ready to take the load. If not, then it gets restarted. This does not include any external dependencies like for example databases, which are clearly something the service doesn't have control over. We don't get much benefit if we restart the service when the database is down, because it will not help.
- This is where Readiness checks come in. This is basically a check if the application can handle the incoming requests.
- It should do a sanity check against both internal and external elements that make the service go, so the connection to the database should be checked there.
- If the check fails, the service is not being restarted, but the Kubernetes will not direct any traffic to it. This is particularly useful if we perform a rolling update and the new version has some issue connecting to the DB. In this case, it stays not ready, but those instances are not being restarted.

When to use Readiness and Liveness Probes

- Despite how great readiness and liveness probes can be, they’re not always necessary. When updating deployments, Kubernetes will already wait for a replacement pod to start running before removing the old pod. Additionally, if a pod stops running, it will automatically try to restart it. Where these probes prove their worth is the **time between when a pod starts running and when your service actually starts functioning.** Kubernetes already knows if your container is running, probes let it know if your container is functioning.

- If one of our services take time between when a pod starts running and the service is actually responding to requests can be significant (5–10 seconds). Without a readiness probe, we’d end up with at least that much downtime every time we updated that deployment.

- Additionally, we have a number of services that can fail in ways that don’t always result in the container crashing. These aren’t particularly common, but when it happens, it’s nice to have a livenessprobe around to catch the issue and restart the container.

Volumes

- Containers are ephemeral (short lived in nature).
- All data stored inside a container is deleted if the container crashes. However, the kubelet will restart it with a clean state, which means that it will not have any of the old data
- To overcome this problem, Kubernetes uses Volumes. A Volume is essentially a directory backed by a storage medium. The storage medium and its content are determined by the Volume Type.
- In Kubernetes, a Volume is attached to a Pod and shared among the containers of that Pod.
- The Volume has the same life span as the Pod, and it outlives the containers of the Pod - this allows data to be preserved across container restarts.

Volume Types

A Volume Type decides the properties of the directory, like size, content, etc. Some examples of Volume Types are:

- node-local types such as **emptyDir(sharing internal volumes, ex-if you have containers running in single pod)** and **hostPath**
- file-sharing types such as **nfs**
- cloud provider-specific types like **awsElasticBlockStore**, **azureDisk**, or **gcePersistentDisk**
- distributed file system types, for example **glusterfs** or **cephfs**
- special-purpose types like **secret**, gitRepo
- **persistentVolumeClaim**

emptyDir

- Use this when we want to share contents between multiple containers on the same pod & not to the host machine
- An **emptyDir** volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node.
- As the name says, it is initially empty.
- Containers in the Pod can all read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each Container.
- When a Pod is removed from a node for any reason, the data in the emptyDir is deleted forever.
- A Container crashing does **not** remove a Pod from a node, so the data in an emptyDir volume is safe across Container crashes.

hostPath

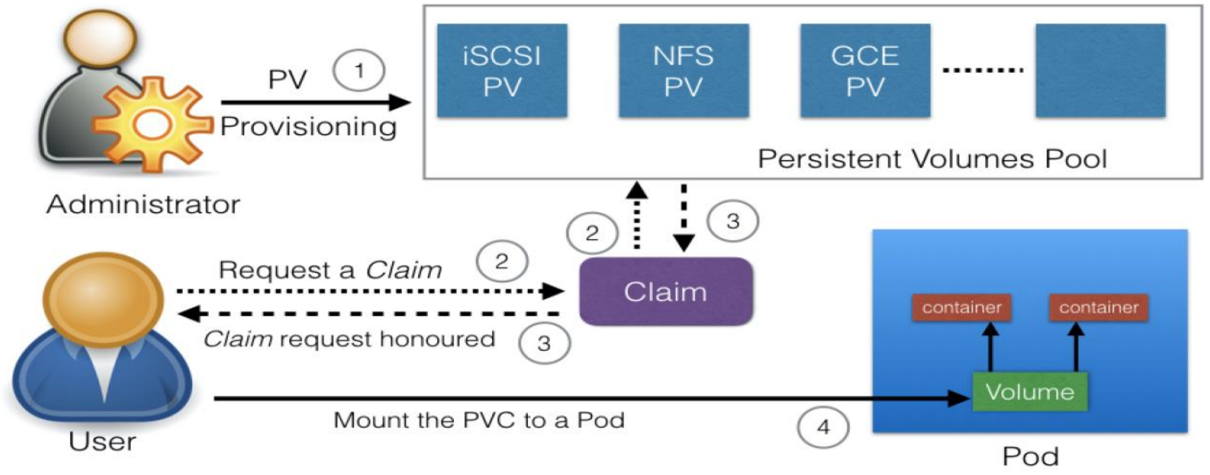
- Use this when we want to access the content of a pod/container from hostmachine
- A hostPath volume mounts a file or directory from the host node’s filesystem into your Pod.
- For example, some uses for a hostPath are:
  - running a Container that needs access to Docker internals; use a hostPath of /var/lib/docker
  - allowing a Pod to specify whether a given hostPath should exist prior to the Pod running, whether it should be created, and what it should exist as

PersistentVolumes

- In a typical IT environment, storage is managed by the storage/system administrators. The end user will just get instructions to use the storage, but does not have to worry about the underlying storage management.
- In the containerized world, we would like to follow similar rules, but it becomes challenging, given the many Volume Types we have seen earlier. Kubernetes resolves this problem with the **PersistentVolume (PV)** subsystem.
- A persistent volume (PV) is a cluster-wide resource that you can use to store data in a way that it persists beyond the lifetime of a pod.
- The PV is not backed by locally-attached storage on a worker node but by networked storage system such as EBS or NFS or a distributed filesystem like Ceph.
- K8s provides APIs for users and administrators to manage and consume storage. To manage the Volume, it uses the PersistentVolume API resource type, and to consume it, uses the PersistentVolumeClaim API resource type.

PersistentVolumeClaims

- In order to use a PV you need to claim it first, using a persistent volume claim (PVC).
- The PVC requests a PV with your desired specification (size, access modes, speed, etc.) from Kubernetes and once a suitable PersistentVolume is found, it is bound to a PersistentVolumeClaim.
- After a successful bound to a pod, you can mount it as a volume.
- Once a user finishes its work, the attached PersistentVolumes can be released. The underlying PersistentVolumes can then be reclaimed and recycled for future usage.



awsElasticBlockStore

- An awsElasticBlockStore volume mounts an Amazon Web Services (AWS) EBS Volume into your Pod. Unlike emptyDir, which is erased when a Pod is removed, the contents of an EBS volume are preserved and the volume is merely unmounted.
- This means that an EBS volume can be pre-populated with data, and that data can be "handed off" between Pods.
- There are some restrictions when using an awsElasticBlockStore volume:
    - the nodes on which Pods are running must be AWS EC2 instances
    - those instances need to be in the same region and availability-zone as the EBS volume
    - EBS only supports a single EC2 instance mounting a volume

Namespace

- You can name your object, but if many are using the cluster then it would be difficult for managing.
- A namespace is a group of related elements that each have a unique name or identifier. Namespace is used to uniquely identify one or more names from other similar names of different objects, groups or the namespace in general.
- Kubernetes namespaces help different projects, teams, or customers to share a Kubernetes cluster & provides:
  - A scope for every Names.
  - A mechanism to attach authorization and policy to a subsection of the cluster.
- By default, a Kubernetes cluster will instantiate a default namespace when provisioning the cluster to hold the default set of Pods, Services, and Deployments used by the cluster
- We can use resource quota on specifying how many resources each namespace can use.
- Most Kubernetes resources (e.g. pods, services, replication controllers, and others) are in some namespaces. And low-level resources, such as nodes and persistentVolumes, are not in any namespace.
- Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all.

Create new namespaces

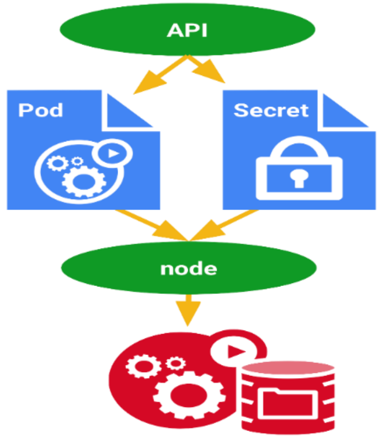
- Lets assume we have shared Kubernetes cluster for dev & QA use cases.
- The dev team would like to maintain a space in the cluster where they can get a view on the list of Pods, Services, and Deployments they use to build and run their application. In this no restrictions are put on who can or cannot modify resources to enable agile development.
- For QA team we can enforce strict procedures on who can or cannot manipulate the set of Pods, Services, and Deployments.
- Partition the Kubernetes cluster into two namespaces: dev & QA

```
$ kubectl get namespaces
```

Secrets & ConfigMap

Secrets

- You don’t want sensitive information such as a database password or an API key kept around in clear text
- Secrets provide you with a mechanism to use such information in a safe and reliable way with the following properties:
  - Secrets are namespaced objects, that is, exist in the context of a namespace
  - You can access them via a volume or an environment variable from a container running in a pod
  - The secret data on nodes is stored in tmpfs volumes (Tmpfs is a file system which keeps all files in virtual memory. Everything in tmpfs is temporary in the sense that no files will be created on your hard drive.)
  - A per-secret size limit of 1MB exists
  - The API server stores secrets as plaintext in etcd



- Secrets can be created :
  - from a text file
  - from a Yaml file
- Secrets can be accessed in following ways:
  - use secrets as environment variables
  - use secrets as volumes in the pod

Create a secret from a text file or dir (generic type)

```
$ echo "root" > username.txt; echo "password" > password.txt
$ kubectl create secret <type> <secretname> --from-file=<filetoread>
$ kubectl create secret generic mysecret --from-file=username.txt --from-file=password.txt
$ kubectl describe secrets/<secretname>
```

Create secret from a YAML file – Example: yamlsecrets.yml

```
apiVersion: v1
kind: Secret           # this defines to create a secret object
metadata:
  name: myyamlsecret
type: Opaque
data:
  <keyname>: <value>   # value must be in  'base64 data‘
#  myname: YWRhbQo=
$ echo adam | base64
$ kubectl apply -f yamlsecrets.yml
$ kubectl describe secrets/myyamlsecret
```

Accessing the secret using environment variable– Example: yamlenvsecrets.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: yamlenvsecret
spec:
  containers:
  - name: c1
    image: centos
    command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
    env:
    - name: MYENVUSER          # env name in which value of the key is stored
      valueFrom:
        secretKeyRef:
          name: myyamlsecret     # name of the secret created
          key: myname              # name of the key
```

```
$ kubectl apply -f yamlenvsecrets.yml

$ kubectl exec yamlenvsecret -it -- /bin/bash

  - env |grep MYENVUSER
```

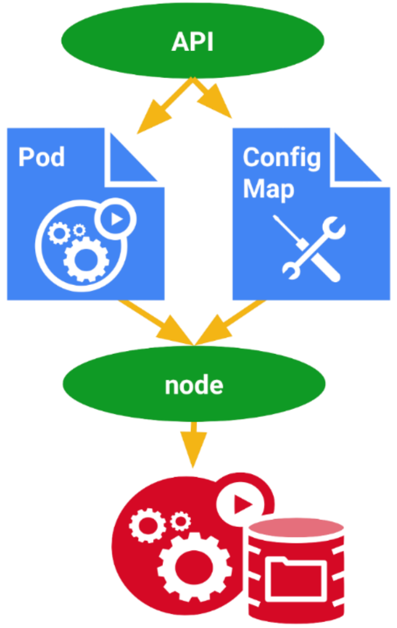
ConfigMap

- While performing application deployments on kubernetes cluster, sometimes we need to change the application configuration file depending on environments like dev, qa, stage or prod.
- Changing this application configuration file means we need to change source code, commit the change, creating a new image and then go through the complete deployment process.
- Hence these configurations should be decoupled from image content in order to keep containerized applications portable.
- This is where Kubernetes's ConfigMap comes handy. It allows us to handle configuration files much more efficiently.
- ConfigMaps are useful for storing and sharing non-sensitive, unencrypted configuration information. Use secrets otherwise.
- ConfigMap can be used to store fine-grained information like individual properties or entire config files.
- ConfigMaps are not intended to act as a replacement for a properties file.

Create a ConfigMap:

- We can create configmaps:
  - From command
  - From files

- ConfigMap can be accessed in following ways:
  - As environment variables
  - As volumes in the pod



Create ConfigMap from a YAML file – Example: yamlconfig.yml

```
apiVersion: v1
kind: ConfigMap              # Defines the object to be Configmap
metadata:
  name: myyamlmap
#  namespace: default        # Can specify which namespace to relate
data:
  <keyname> : <value>        # data to be stored
 # SKILL: devops
 # CLASS: kubernetes
```

```
$ kubectl apply -f yamlconfig.yml
$ kubectl describe configmaps/<mapname>
$ kubectl get configmap <mapname> -o yaml
```

Accessing Config using environment variable - Example: yamlenvconfig.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: yamlenvconfig
spec:
  containers:
  - name: c1
    image: centos
    command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
    env:
    - name: MYENV         # env name in which value of the key is stored
      valueFrom:
        configMapKeyRef:
          name: myyamlmap
          key: SKILL              # name of the key
```

```
$ kubectl apply -f yamlenvconfig.yml

$ kubectl exec yamlenvsecret -it -- /bin/bash

  - env |grep MYENV
```

Managing Compute Resources for Containers

- A pod in Kubernetes will run with no limits on CPU and memory
- You can optionally specify how much CPU and memory (RAM) each Container needs.
- Scheduler decides about which nodes to place Pods, only if the Node has enough CPU resources available to satisfy the Pod CPU request
- CPU is specified in units of **cores**, and memory is specified in units of **bytes**.
- Two types of constraints can be set for each resource type: **requests** and **limits**
  - A **request** is the amount of that resources that the system will guarantee for the container, and Kubernetes will use this value to **decide on which node to place the pod**
  - A **limit** is the maximum amount of resources that Kubernetes will allow the container to use. In the case that request is not set for a container, it defaults to limit. If limit is not set, then if defaults to 0 (unbounded).
- CPU values are specified in “millicpu” and memory in MiB

Pod Resources – Example: podresources.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: resources
spec:
  containers:
  - name: resource
    image: centos
    command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
    resources:                                    # Describes the type of resources to be used
      requests:
        memory: "<value in mebibytes>" # A mebibyte is 1,048,576 bytes, ex: 64Mi
        cpu: "<value in millicores >"          # CPU core split into 1000 units (milli = 1000), ex: 100m
      limits:
        memory: "<value in mebibytes>" # ex: 128Mi
        cpu: "<value in millicores >"  # ex: 200m
```

```
$ kubectl apply -f podresources.yml
$ kubectl describe pods resources
```

- To see the pods that uses the most cpu and memory

```
$ kubectl top pod --all-namespaces
```

- To see the total cpu, memory usage of all pods

```
$ kubectl get nodes --no-headers | awk '{print $1}' | xargs -I {} sh -c 'echo {};
kubectl describe node {} | grep Allocated -A 5 | grep -ve Event -ve Allocated -
ve percent -ve -- ; echo’
```

Resource Quotas

- A Kubernetes cluster can be divided into namespaces. If a Container is created in a namespace that has a default CPU limit, and the Container does not specify its own CPU limit, then the Container is assigned the default CPU limit.
- Namespaces can be assigned ResourceQuota objects, this will limit the amount of usage allowed to the objects in that namespace. You can limit:
  - Compute
  - Storage
  - Memory
- Here are two of the restrictions that a resource quota imposes on a namespace:
  - Every Container that runs in the namespace must have its own CPU limit.
  - The total amount of CPU used by all Containers in the namespace must not exceed a specified limit.

ResourceQuota - Example: quotadeploy.yml

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: deployments
spec:
  replicas: 3
  template:
    metadata:
      name: testpod8
      labels:
        objtype: deployment
    spec:
     containers:
       - name: c00
         image: ubuntu
         command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
         resources:
            requests:
              cpu: "200m"
```

```
$ kubectl apply -f quotadeploy.yml --namespace=testing
$ kubectl get pods --namespace=testing
$ kubectl describe deployments deployments --namespace=testing
$ kubectl delete -f cpulimit.yml --namespace=testing
```

Configure Default CPU Requests and Limits for a Namespace

- **Create a LimitRange and a Pod**: **cpudefault.yml**

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
spec:
  limits:
  - default:
      cpu: 1
    defaultRequest:
      cpu: 0.5
    type: Container
```

```
$ kubectl apply -f cpudefault.yml --namespace=testing
```

- **Create a pod - cpupod.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo
spec:
  containers:
  - name: default-cpu-demo-ctr
    image: nginx
```

```
$ kubectl apply -f cpupod.yml --namespace=testing
$ kubectl get pod default-cpu-demo --output=yaml --namespace=testing
```

- The output shows that the Pod’s Container has a CPU request of 500 millicpus and a CPU limit of 1 cpu. These are the default values specified by the LimitRange

What if you specify a Container’s limit, but not its request?

- **Create a Pod in which the Container specifies a CPU limit, but not a request: cpupod2.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-2
spec:
  containers:
  - name: default-cpu-demo-2-ctr
    image: nginx
    resources:
      limits:
        cpu: "1"
```

```
$ kubectl apply -f cpupod2.yml --namespace=testing
$ kubectl get pod default-cpu-demo-2 --output=yaml --namespace=testing
```

- The output shows that the Container’s CPU request is set to match its CPU limit.
- Notice that the Container was not assigned the default CPU request value of 0.5 cpu.

What if you specify a Container’s request, but not its limit?

- **Create a Pod in which the Container specifies a CPU request, but not a limit: cpupod3.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-3
spec:
  containers:
  - name: default-cpu-demo-3-ctr
    image: nginx
    resources:
      requests:
        cpu: "0.75"
```

```
$ kubectl create -f cpupod3.yml --namespace=testing
$ kubectl get pod default-cpu-demo-3 --output=yaml --namespace=testing
```

- The output shows that the Container’s CPU request is set to the value specified in the Container’s configuration file.
- The Container’s CPU limit is set to 1 cpu, which is the default CPU limit for the namespace

Configure Min & Max Memory Constraints for a Namespace

- **Create a LimitRange and a Pod**: **memdefault.yml**

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
spec:
  limits:
  - max:
      memory: 1Gi
    min:
      memory: 500Mi
    type: Container
```

```
$ kubectl apply -f memdefault.yml --namespace=testing

$ kubectl get limitrange mem-min-max-demo-lr --namespace=testing
```

- **Create a pod - mempod.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo
spec:
  containers:
  - name: constraints-mem-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "800Mi"
      requests:
        memory: "600Mi"
```

```
$ kubectl apply -f mempod.yml --namespace=testing

$ kubectl get pod constraints-mem-demo --output=yaml --namespace=testing
```

- The output shows that the Container has a memory request of 600 MiB and a memory limit of 800 MiB. These satisfy the constraints imposed by the LimitRange.

Attempt to create a Pod that exceeds the maximum memory constraint

- **Create a Pod in which the Container specifies a memory request of 800 MiB and a memory limit of 1.5 GiB: mempod2.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-2
spec:
  containers:
  - name: constraints-mem-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "1.5Gi"
      requests:
        memory: "800Mi"
```

```
$ kubectl apply -f mempod2.yml --namespace=testing

$ kubectl get pod constraints-mem-demo-2  --output=yaml --namespace=testing
```

- **The output shows that the Pod does not get created, because the Container specifies a memory limit that is too large**

Attempt to create a Pod that does not meet the minimum memory request

- **Create a Pod in which the Container specifies a memory request of 100 MiB and a memory limit of 800 MiB: mempod3.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-3
spec:
  containers:
  - name: constraints-mem-demo-3-ctr
    image: nginx
    resources:
      limits:
        memory: "800Mi"
      requests:
        memory: "100Mi"
```

```
$ kubectl apply -f mempod3.yml --namespace=testing
$ kubectl get pod constraints-mem-demo-3  --output=yaml --namespace=testing
```

- The Pod does not get created, because the Container specifies a memory request that is too small

Create a Pod that does not specify any memory request or limit

- **Create a Pod in which the Container does not specify a memory request, and it does not specify a memory limit: mempod4.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-4
spec:
  containers:
  - name: constraints-mem-demo-4-ctr
    image: nginx
```

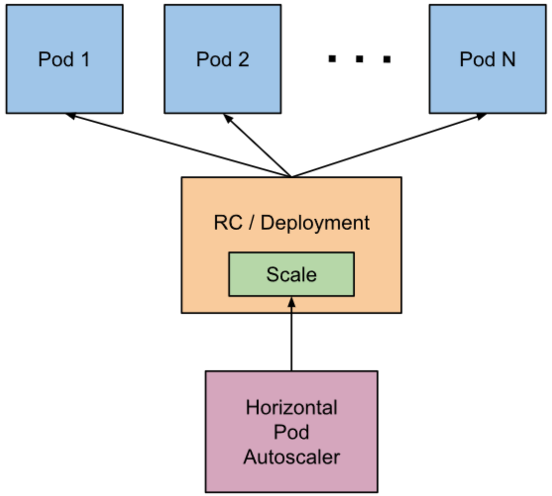
```
$ kubectl create -f mempod4.yml --namespace=testing

$ kubectl get pod constraints-mem-demo-4  --output=yaml --namespace=testing
```

- The output shows that the Pod’s Container has a memory request of 1 GiB and a memory limit of 1 GiB

Autoscalling (Horizontal Pod Autoscaler)

- K8s has the possibility to automatically scale pods based on observed CPU utilization (or with custom metrics), which is Horizontal Pod Autoscaling
- Scaling can be done only for scalable objects like controller, deployment or replica set.
- HPA is implemented as a Kubernetes API resource and a controller.
- The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user.



How does the Horizontal Pod Autoscaler work?

- The HPA is implemented as a control loop, with a period controlled by the controller manager’s --horizontal-pod-autoscaler-sync-period flag (with a default value of 15 seconds).
- During each period, the controller manager queries the resource utilization against the metrics specified in each HorizontalPodAutoscaler definition.
- For per-pod resource metrics (like CPU), the controller fetches the metrics from the resource metrics API for each pod targeted by the HorizontalPodAutoscaler.
- Then, if a target utilization value is set, the controller calculates the utilization value as a percentage of the equivalent resource request on the containers in each pod.
- If a target raw value is set, the raw metric values are used directly. The controller then takes the mean of the utilization or the raw value (depending on the type of target specified) across all targeted pods, and produces a ratio used to scale the number of desired replicas

Logging

- Logging is one option to understand what is going on inside your applications and the cluster at large

```
apiVersion: v1
kind: Pod
metadata:
  name: logme
spec:
  containers:
  - name: loggen
    image: centos
    command:
      - "bin/bash"
      - "-c"
      - "while true; do echo $(date) | tee /dev/stderr; sleep 1; done"
```

```
$ kubectl apply -f log.yml
$ kubectl logs logme
$ kubectl logs -f logme
```

- To view the five most recent log lines

```
$ kubectl logs --tail=5 logme
```

- To view the five most recent log lines live

```
$ kubectl logs --tail=5 -f logme
```

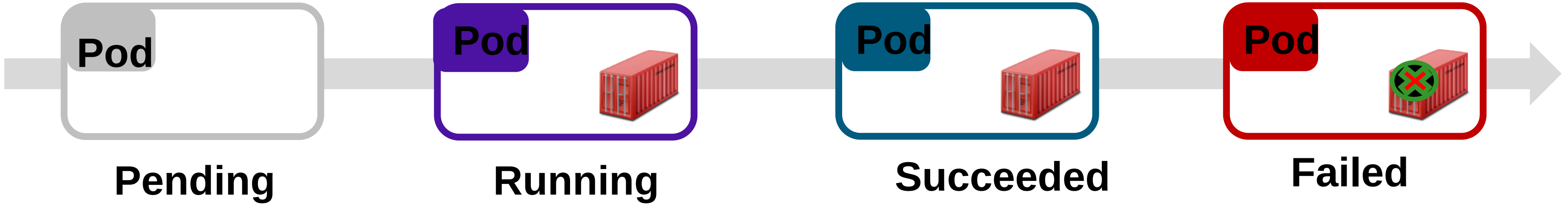
- To view the log from the past 10 sec

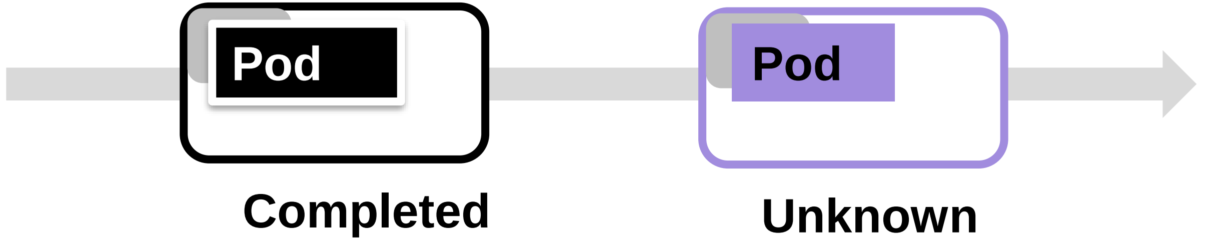
```
$ kubectl logs --since=10s logme
```

Pod LifeCycle

Pod Phases

- The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle.
- Here are the possible values for phase:





Pending:

- The Pod has been accepted by the Kubernetes system, but its not **running**
- one or more of the Container images is still **downloading**
- If the pod cannot be scheduled because of resource **constraints**

Running:

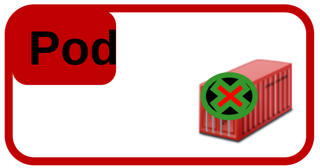
- The Pod has been bound to a node
- All of the Containers have been created
- At least one Container is still running, or is in the process of starting or restarting

Succeeded:

- All Containers in the Pod have terminated in success, and will not be restarted

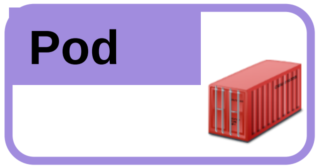
Failed:

- ➢ All Containers in the Pod have terminated, and at least one Container has terminated in failure.
- ➢ The Container either exited with non-zero status or was terminated by the system



Unknown:

- State of the Pod could not be **obtained**
- Typically due to an **Error** in **Network** or **communicating** with the host of the Pod



Completed:

- The pod has run to completion as there's nothing to keep it running eg. Completed Jobs

Pod Conditions

- A Pod has a PodStatus, which has an array of PodConditions through which the Pod has or has not passed.
- Using **'kubectl describe pod PODNAME'** you can get the condition of a Pod
- These are the possible types:
  - **PodScheduled**: the Pod has been scheduled to a node
  - **Ready**: the Pod is able to serve requests and will be added to the load balancing pools of all matching Services
  - **Initialized**: all init containers have started successfully
  - **Unschedulable**: the scheduler cannot schedule the Pod right now, for example due to lacking of resources or other constraints
  - **ContainersReady**: all containers in the Pod are ready

Container States

- Once Pod is assigned to a node by scheduler, kubelet starts creating containers using container runtime
- Using **‘kubectl get pod PODNAME –o yaml’** state is displayed for each container within that Pod
- There are three possible states of containers:
  - **Waiting**: A container in Waiting state still indicates that its pulling images, applying Secrets, etc
  - **Running**: Indicates that the container is executing without issues. Once a container enters into Running
  - **Terminated**: Indicates that the container completed its execution and has stopped running.A container enters into this when it has successfully completed execution or when it has failed for some reason

Thank you