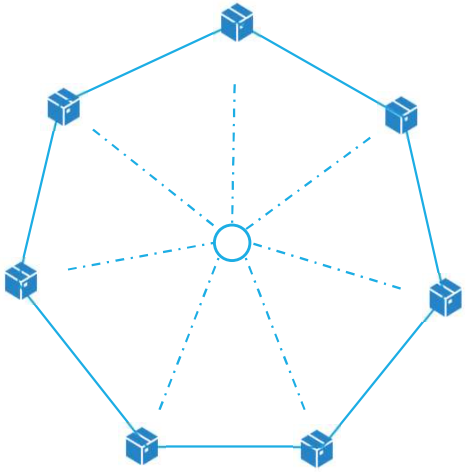


kubernetes

What is Container?

A History Lesson

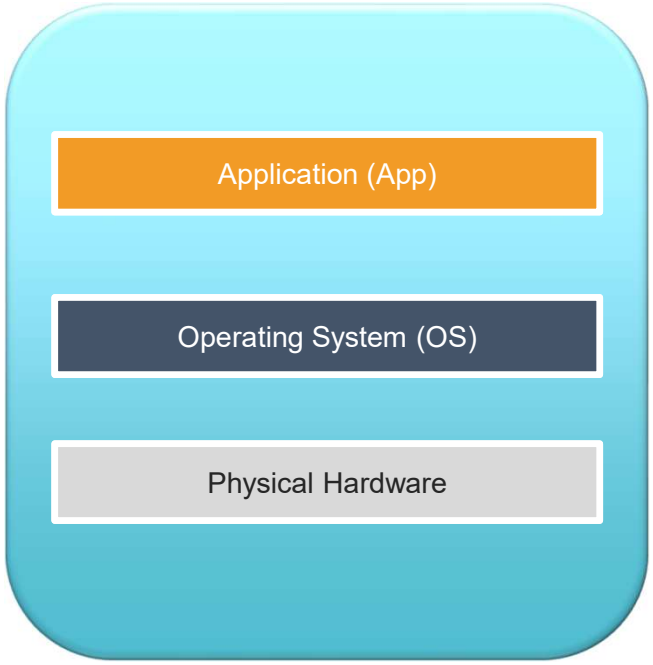
The problem got the solution by a technology called “Hypervisor-Based Virtualization”.

One physical server can contain multiple running applications.

Each application need a VM to run the application binaries.

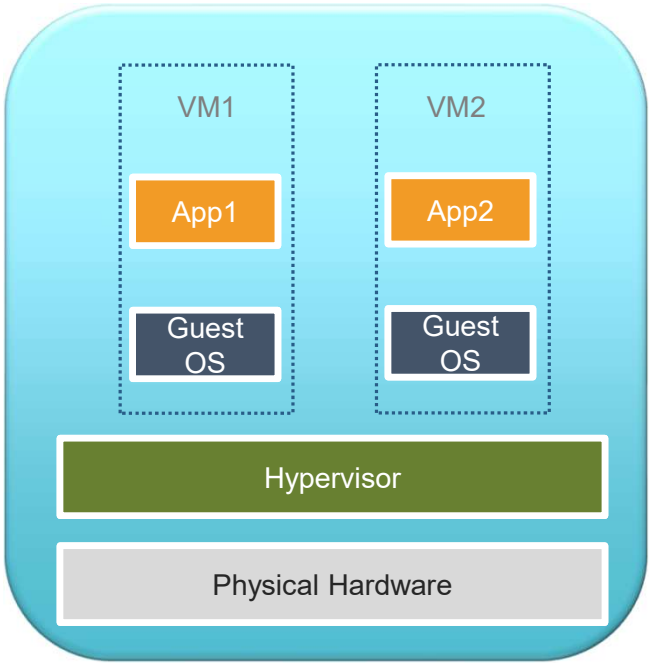
A History Lesson

In the traditional ages of development and deployment of application



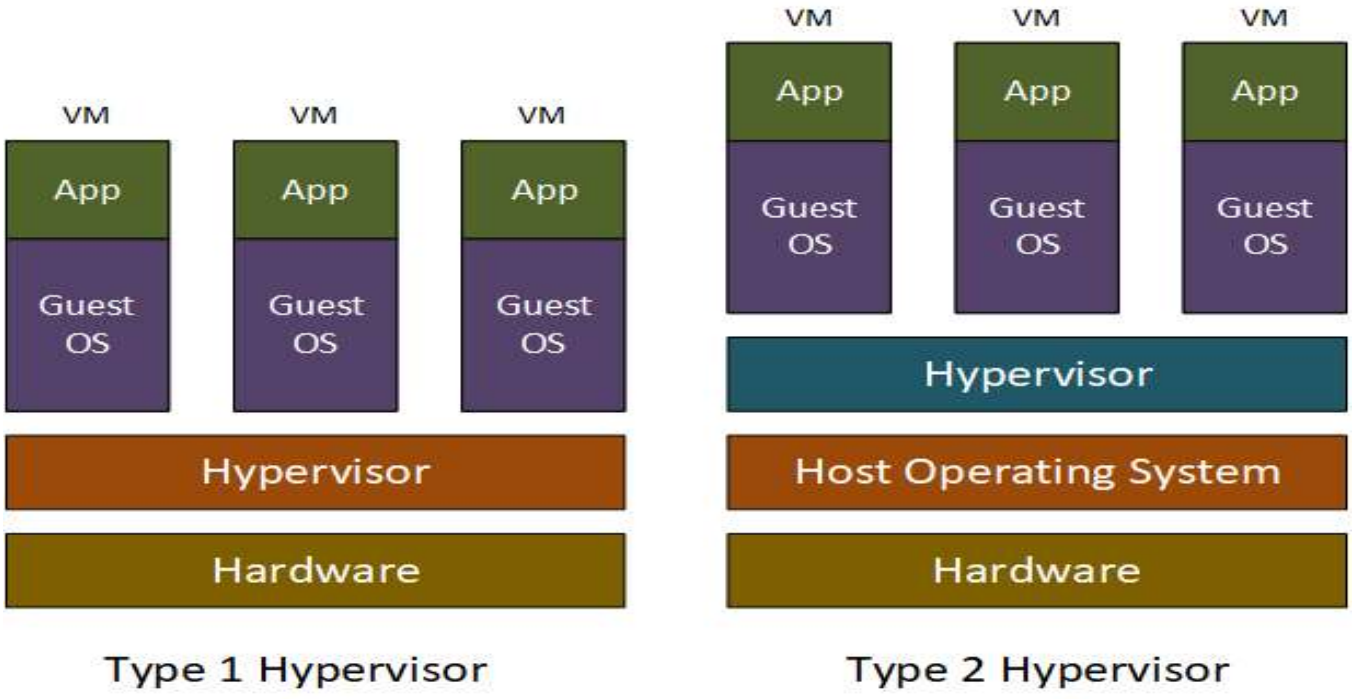
Traditional Approach

Virtualization

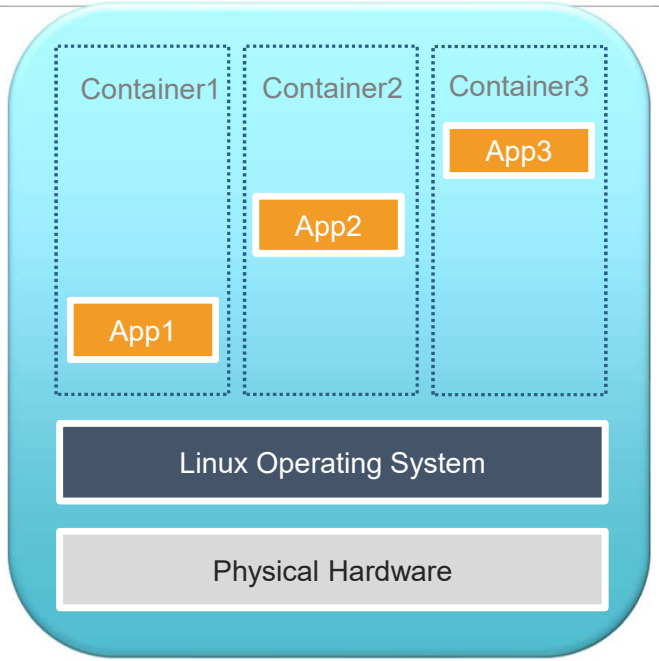


Server Virtualization

Virtualization



Containerization



Containerization

Introducing Containers

Container based virtualization uses the kernel on the host's operating system to run multiple guest instances

Each guest instance is called a "Container"

Each container has its own

- Root Filesystem
- Processes
- Memory
- Devices
- Network Ports

From outside its looks like a VM but its not a VM

Introducing Containers

Container based virtualization uses the kernel on the host's operating system to run multiple guest instances

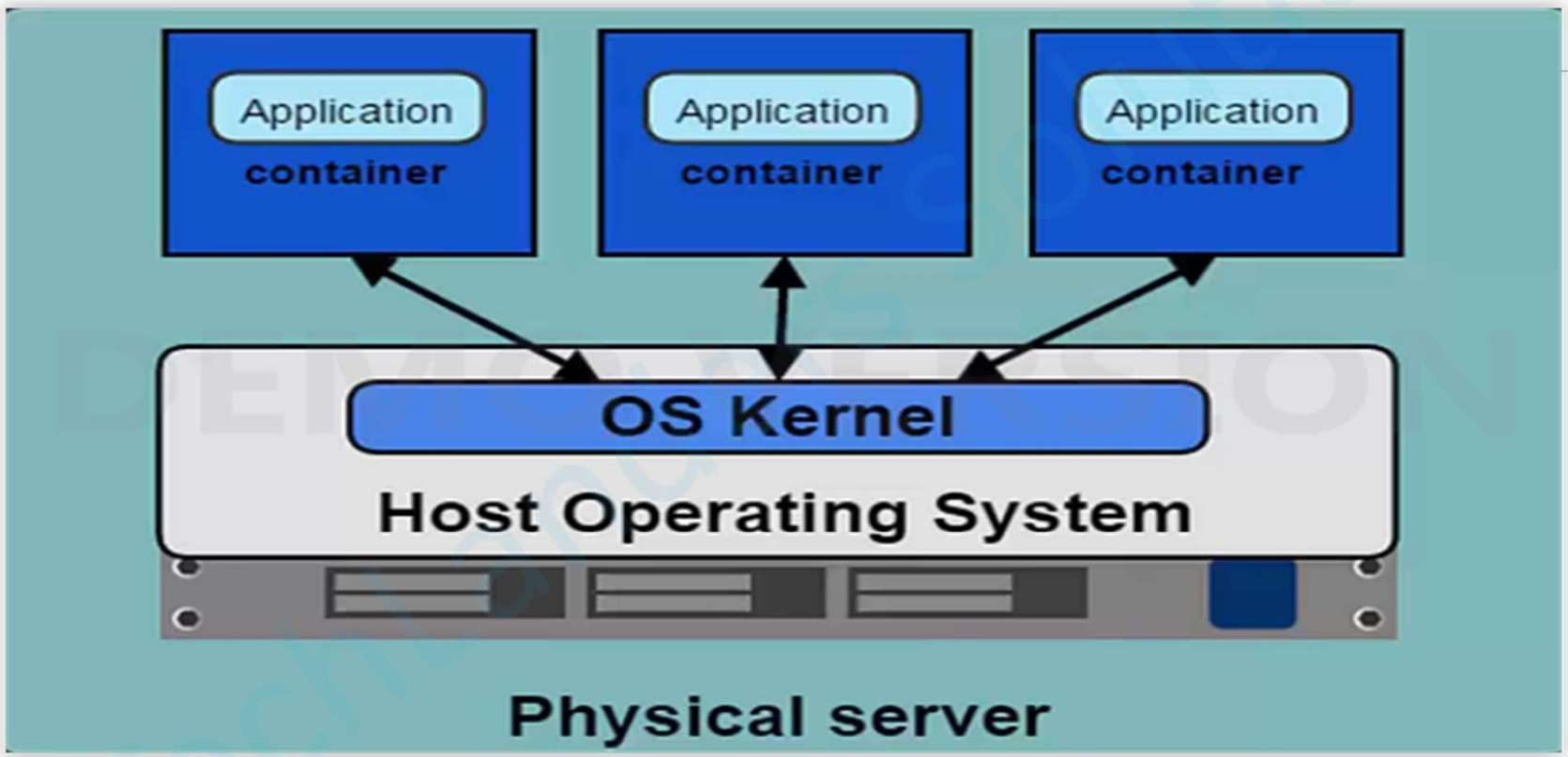
Each guest instance is called a "Container"

Each container has its own

- Root Filesystem
- Processes
- Memory
- Devices
- Network Ports

From outside its looks like a VM but its not a VM

Overview of Containers



Overview of Containers

A question comes in our mind why do we need different containers of our applications.

A question to audience, I want to install:

- Three application
- Pre-requisites all three require different Java Versions
- What will be the possible solution for this design

Containers VS VM's

Container are more light weight

No need to install dedicated guest OS, no virtualization like VM is required

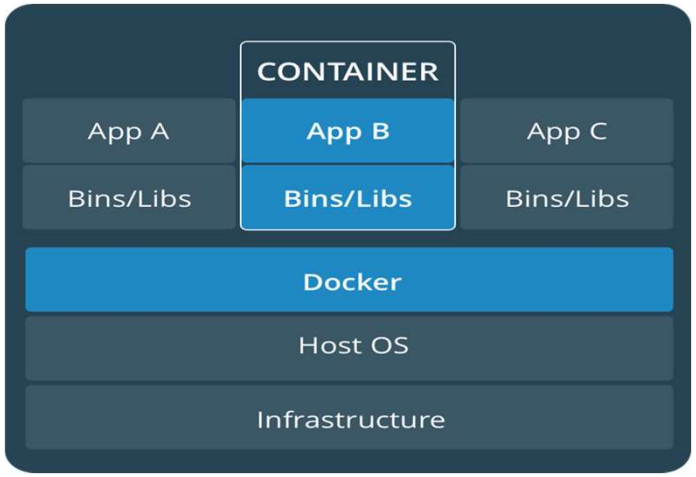
Stop/Start time is very fast

Less CPU, RAM, Storage Space required

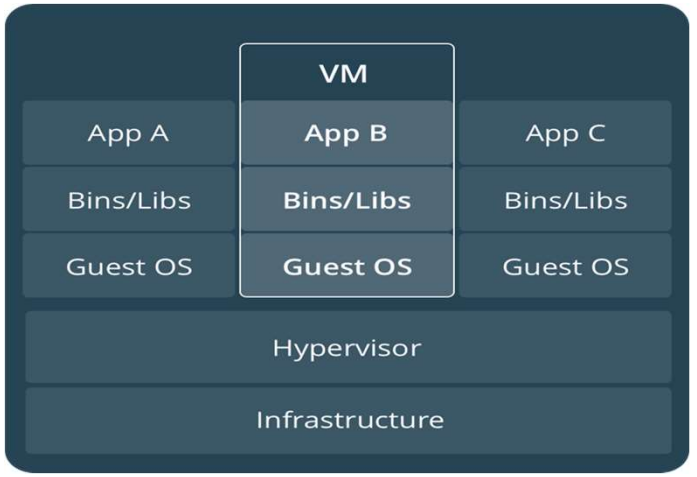
More containers per machine than VM's

Great Portability

Containers VS VM's

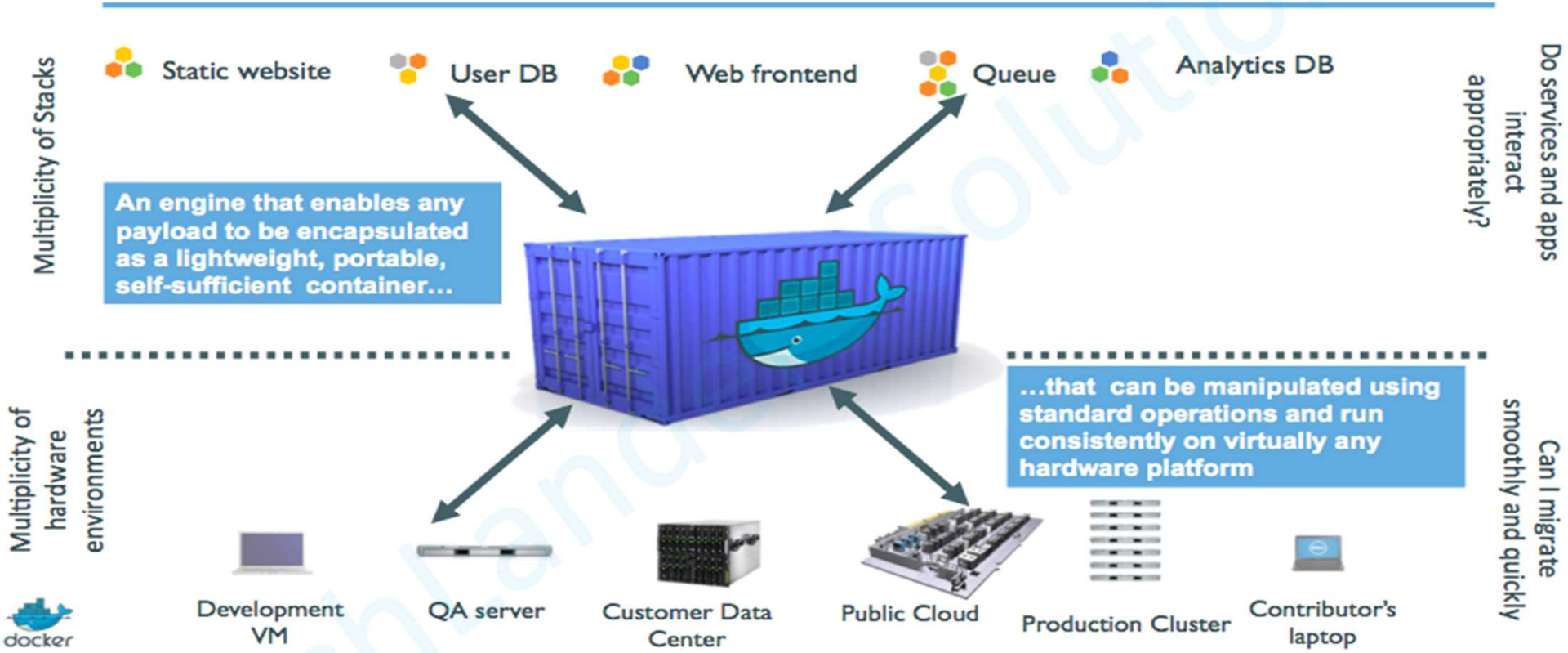


Containers are an app level construct



VMs are an infrastructure level construct to turn one machine into many servers

Shipping Container for Applications



Results

Speed

- No OS to boot = applications online in seconds

Portability

- Less dependencies between process layers = ability to move between infrastructure

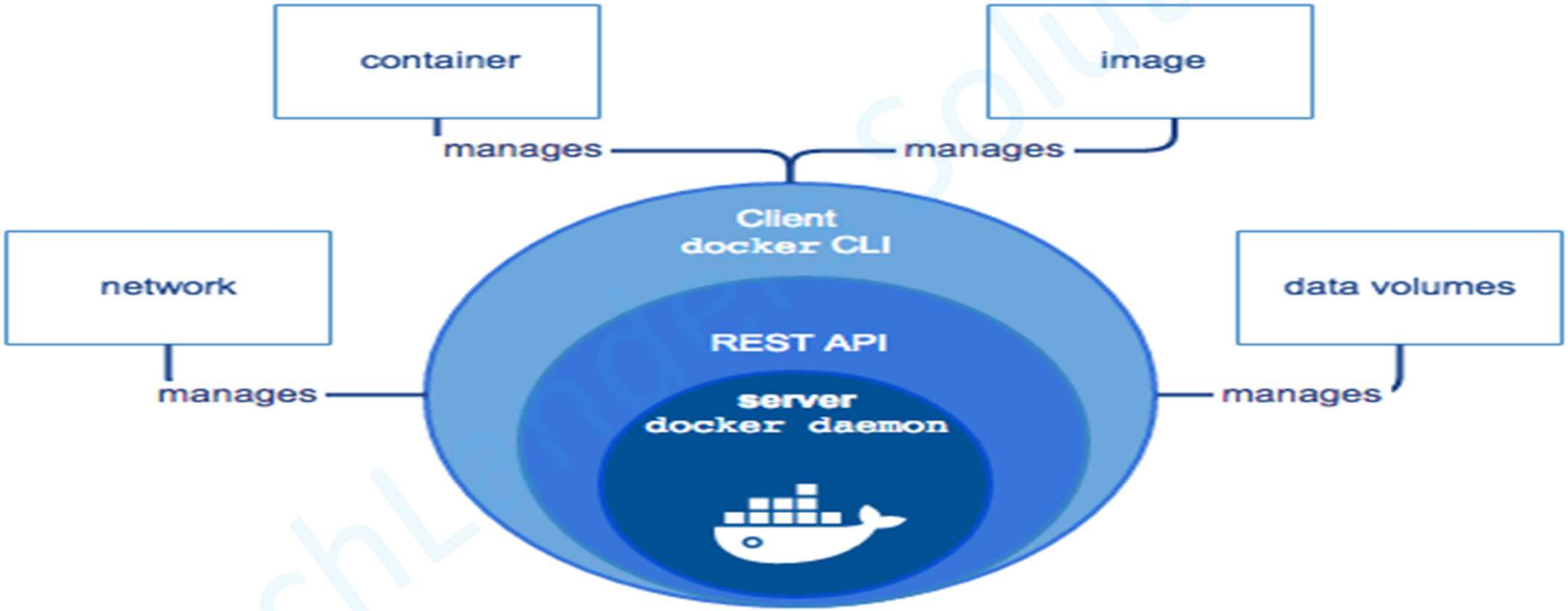
Efficiency

- Less OS overhead
- Improved VM density

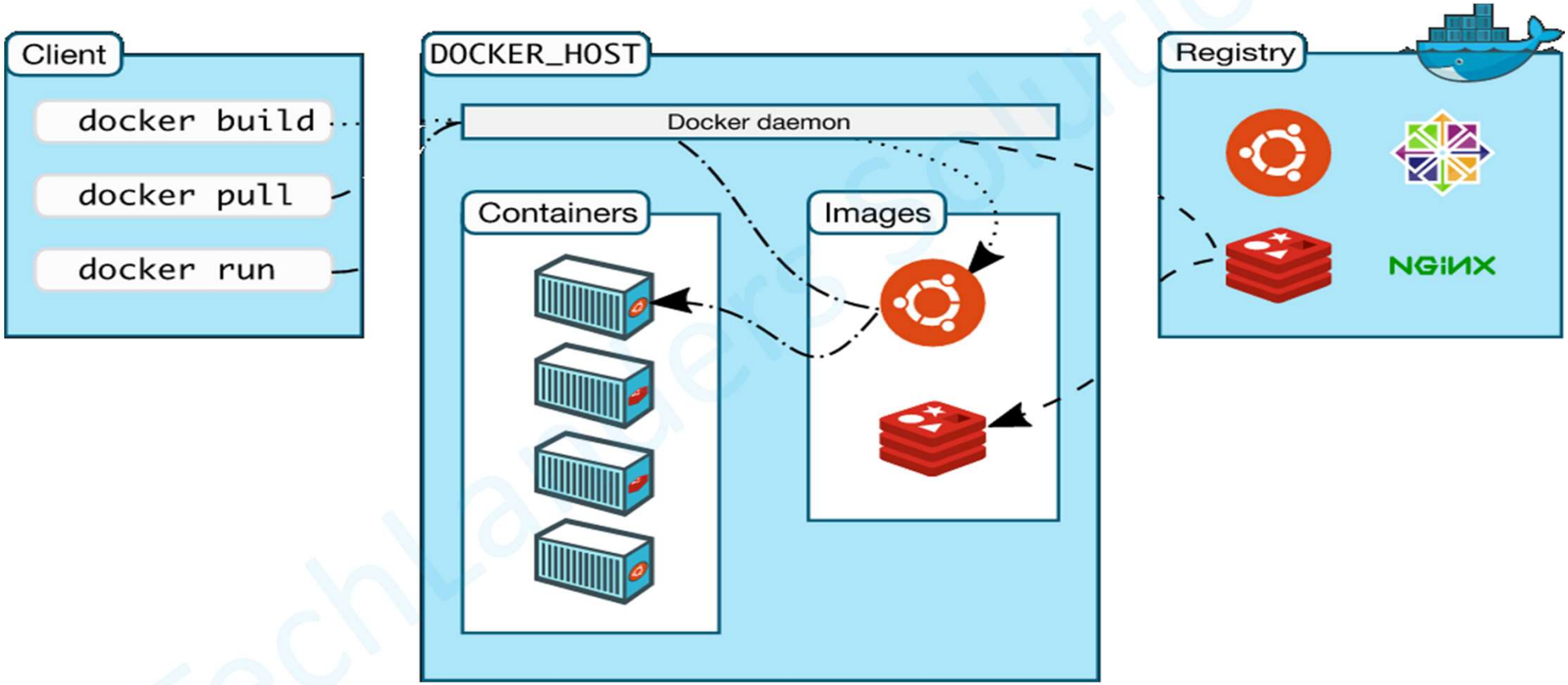
Session: 2

Docker Components

Docker Engine



Docker Architecture



Docker Images

- A Docker image is a read-only template with instructions for creating a Docker container.
- For example, an image might contain an Ubuntu operating system with Apache web server and your web application installed. You can build or update images from scratch or download and use images created by others.
- A docker image is described in text file called a Dockerfile, which has a simple, well-defined syntax.
- Docker images are the build component of Docker.

Docker Containers

- A Docker container is a running instance of a Docker image.
- You can run, start, stop, move, or delete a container using Docker API or CLI commands.
- When you run a container, you can provide configuration metadata such as networking information or environment variables.
- Each container is an isolated and secure application platform, but can be given access to resources running in a different host or container, as well as persistent storage or databases.
- Docker containers are the run component of Docker.

Docker Registries

- A docker registry is a library of images.
- A registry can be public or private, and can be on the same server as the Docker daemon or Docker client, or on a totally separate server.
- Docker registries are the distribution component of Docker.
- “Docker Hub” is known as global registry.

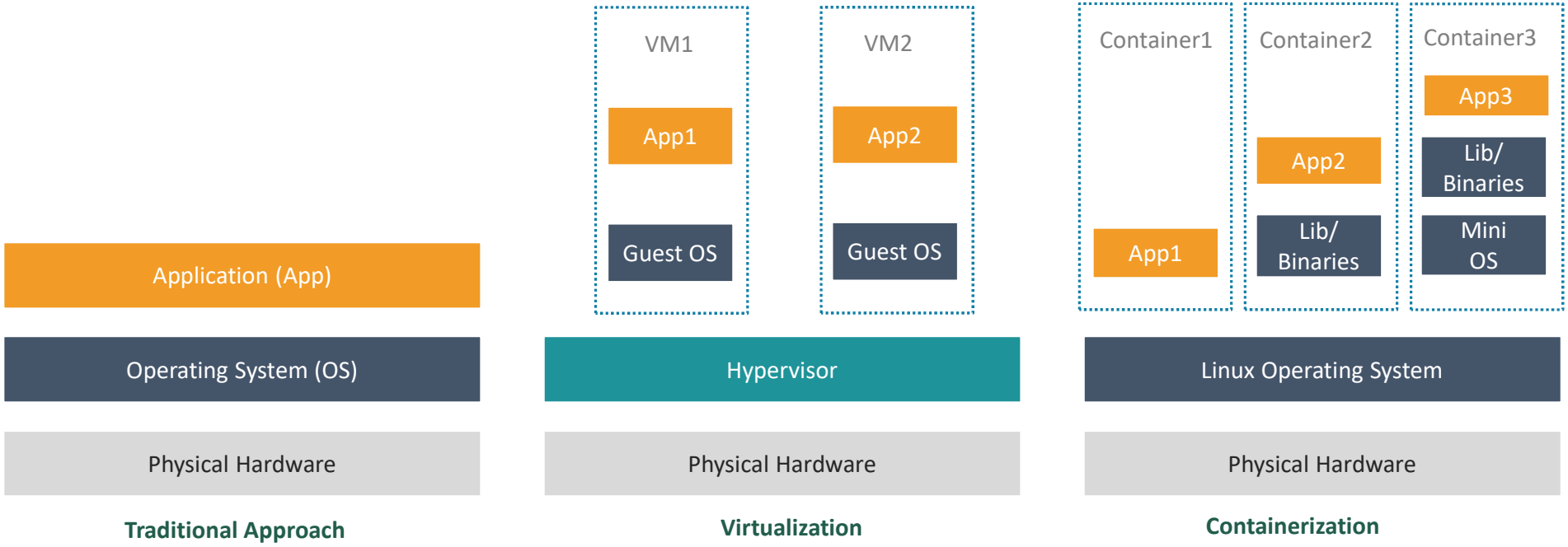
Session: 3

Classroom Environment

Docker Engine Install Demo

- Docker Engine/Client would be installed on Training Environment as demo LAB.
- https://docs.docker.com/engine/installation/linux/centos/

Containers


Application (App)
Operating System (OS)
Physical Hardware
Traditional Approach
VM1
App1
Guest OS
VM2
App2
Guest OS
Hypervisor
Physical Hardware
Virtualization
Container1
App1
Container2
App2
Lib/ Binaries
Container3
App3
Lib/ Binaries
Mini OS
Linux Operating System
Physical Hardware
Containerization

Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.

With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code, you can significantly reduce the delay between writing code and running it in production.

The Docker Platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets you run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host. You can share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

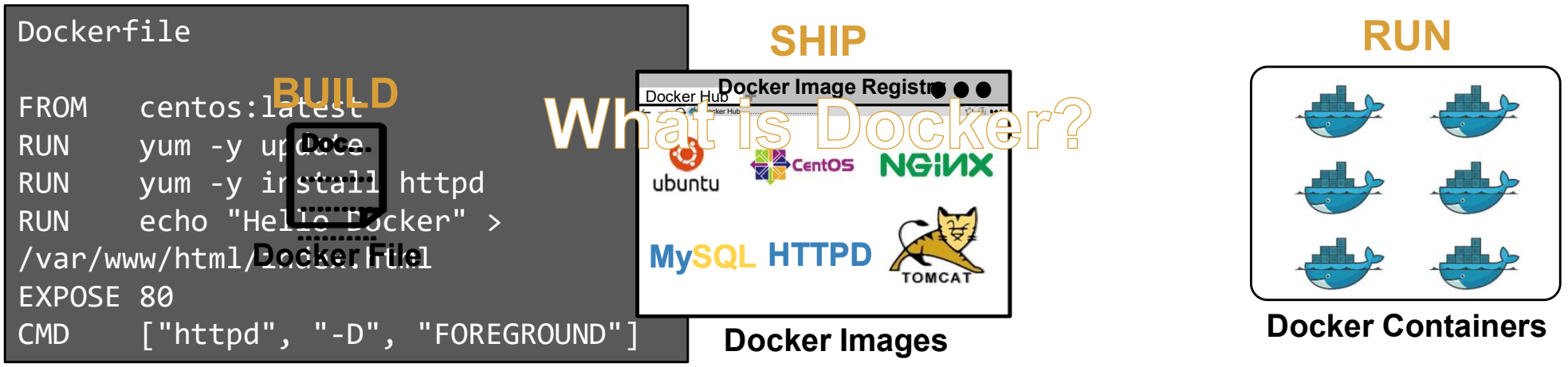
Docker provides tooling and a platform to manage the lifecycle of your containers:

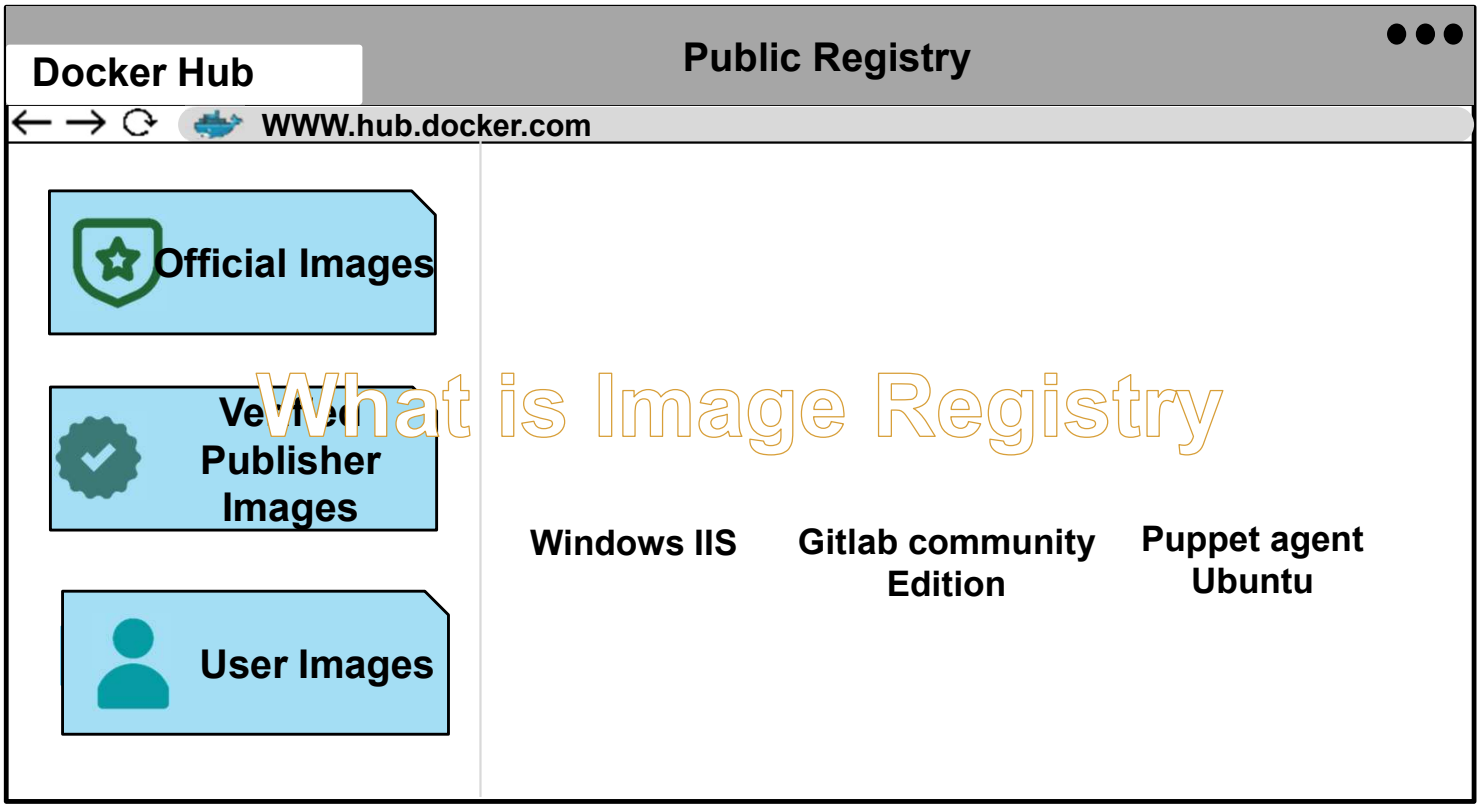
✓Develop your application and its supporting components using containers.

✓The container becomes the unit for distributing and testing your application.

✓When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Docker is a platform to Build, Ship and Run containerized applications




Docker Hub
Public Registry
WWW.hub.docker.com
Official Images
Verified Publisher Images
User Images
What is Image Registry
Windows IIS
Gitlab community Edition
Puppet agent Ubuntu

Session: 2

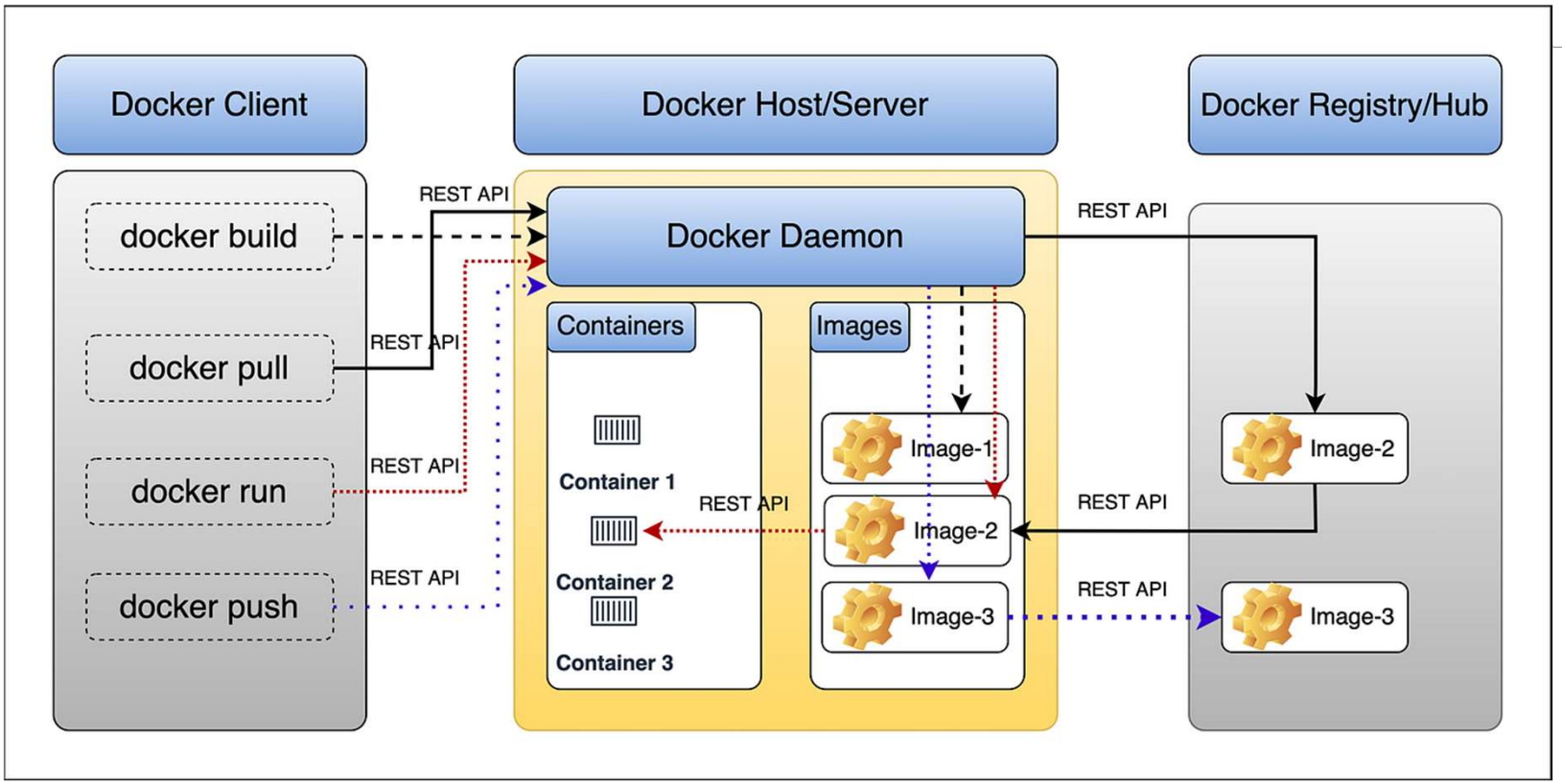
Docker Architecture

Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

The key components of Docker include the Docker Engine, Docker CLI (Command-Line Interface), Docker Compose, and Docker Registry.

Docker Architecture

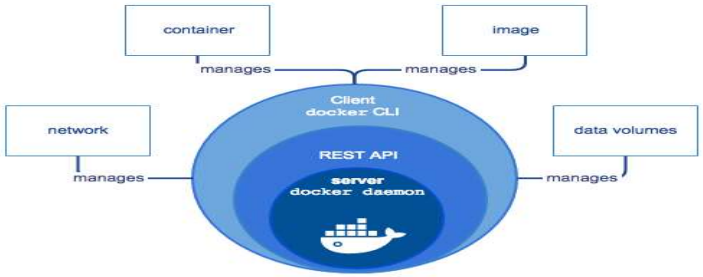


Docker Engine

The Docker Engine is the core component responsible for creating and managing containers. It includes several subcomponents, such as the Docker daemon, REST API, and the Docker command-line interface.

Docker Daemon: This is a background process that manages Docker containers on a host system. It listens for Docker API requests and manages container lifecycles.

Docker API: The API allows interaction with the Docker daemon, enabling users and applications to control Docker containers programmatically..



Docker CLI (Command-Line Interface)

The Docker CLI is a command-line tool that allows users to interact with the Docker Engine. It provides a set of commands for building, managing, and inspecting containers.

Common Commands:

✓*docker build: Builds a Docker image from a Dockerfile.*

✓*docker run: Creates and starts a container based on a specific image.*

✓*docker ps: Lists running containers.*

✓*docker images: Lists available Docker images.*

✓*docker exec: Runs a command inside a running container.*

Docker Registry

A Docker Registry is a centralized repository for storing and distributing Docker images. It allows users to share and access container images, facilitating collaboration and deployment.

Public Registries: Examples include Docker Hub, where users can find and share public Docker images.

Private Registries: Organizations can set up their own private registries for secure storage and distribution of custom Docker images.

Docker Images

Description: Docker images are lightweight, standalone, and executable packages that contain everything needed to run an application, including the code, runtime, libraries, and system tools.

Layered File System: Docker images are composed of layers, each representing a specific set of changes. This layered file system allows for efficient image sharing and distribution.

Docker Container

A Docker container is a running instance of a Docker image or Docker containers are the run component of Docker.

You can run, start, stop, move, or delete a container using Docker API or CLI commands.

When you run a container, you can provide configuration metadata such as networking information or environment variables.

Each container is an isolated and secure application platform, but can be given access to resources running in a different host or container, as well as persistent storage or databases.

Containers share the host OS kernel but have their own user space, providing process-level isolation.

Container are more light weight

No need to install dedicated guest OS,

Less CPU, RAM, Storage Space required

More containers per machine than VM’s

Great Portability

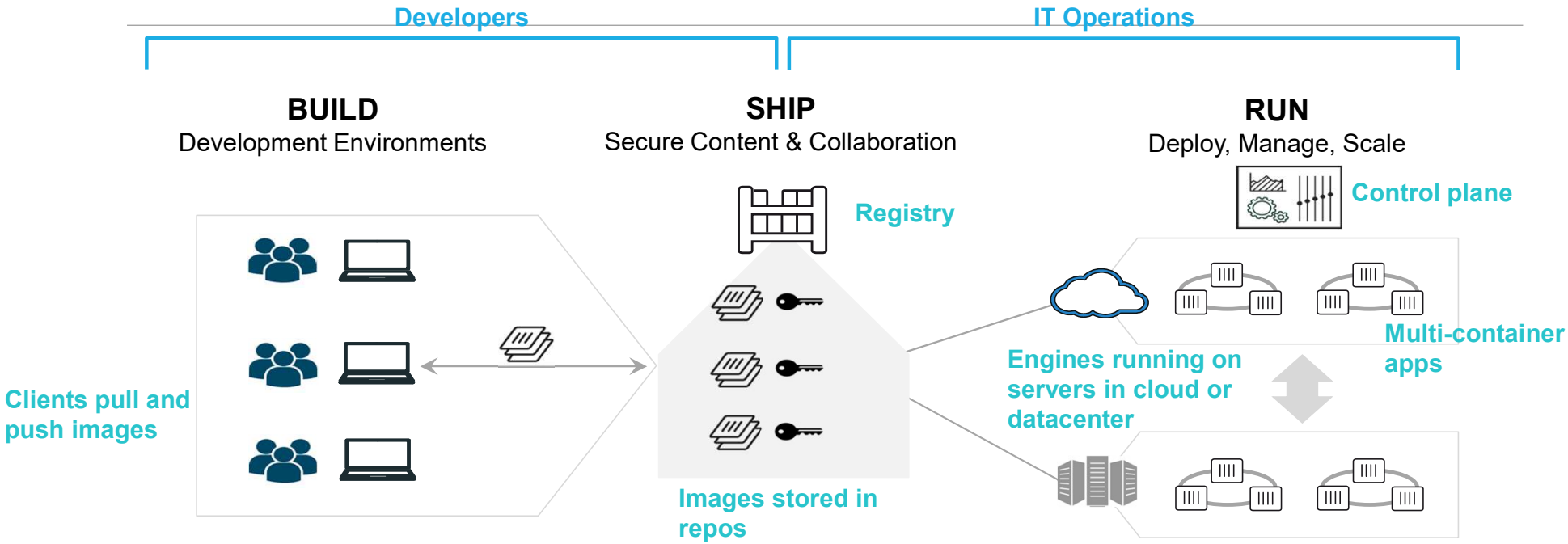
Docker Features

Lightweight : Containers running on a single machine all share the same operating system kernel so they start instantly and make more efficient use of RAM. Images are constructed from layered filesystems so they can share common files, making disk usage and image downloads much more efficient.

Open source and work with all OS : Docker containers are based on open standards allowing containers to run on all major Linux distributions and Microsoft operating systems with support for every infrastructure.

Secure : Containers isolate applications from each other and the underlying infrastructure while providing an added layer of protection for the application.

Container as a Service



Session: 3

Classroom Environment

Docker Engine Install Demo

- Docker Engine/Client would be installed on Training Environment as demo LAB.
- https://docs.docker.com/engine/installation/linux/centos/

Docker Installation Key Points

- docker.service Systemd File:

[root@TechLanders lib]# more /usr/lib/systemd/system/docker.service

- Docker Socket file:

[root@TechLanders lib]# file /var/run/docker.sock
/var/run/docker.sock: socket

- Docker PID file and Docker Container PID file (This is Docker Daemon which will have pid1)

root@TechLanders run]# cat /var/run/docker.pid
3715
[root@TechLanders libcontainerd]# cat /var/run/docker/containerd/docker-containerd.pid
4251

- Docker Detailed Information:

[root@TechLanders libnetwork]# docker info

Manage Docker as a non-root user

- The docker daemon binds to a Unix socket instead of a TCP port.
- By default that Unix socket is owned by the user ”root” and other users can only access it using sudo.
- The docker daemon always runs as the root user.
- If you don’t want to use sudo when you use the docker command, add users to Unix group called ”docker” example: usermod -aG docker <user-name>
- When the docker daemon starts, it makes the ownership of the Unix socket read/writable by the docker group.

Session: 4

Containers

How Container works

- A container uses the host machine's Linux kernel, and consists of any extra files you add when the image is created, along with metadata associated with the container at creation or when the container is started.

- Each container is built from an image.

- The image defines the container's contents, which process to run when the container is launched, and a variety of other configuration details.

- The Docker image is read-only. When Docker runs a container from an image, it adds a read-write layer on top of the image (using a UnionFS) in which your application runs.

How Container works

- When you use the ”docker run” CLI command, the Docker Engine client instructs the Docker daemon to run a container.

---

docker run ubuntu ps ax

- This example tells the Docker daemon to run a container using the centos Docker image, to remain in the foreground in interactive mode (-i), provide a tty terminal (-t) and to run the /bin/bash command.

docker run -i -t centos /bin/bash

docker run -i -t centos (bash will be the default shell)

Because if you exit the current running process /bin/bash (pid 1), container will stop/exit. Use “Ctrl pq” to safe exit without stopping the container.

How Container works

```
[root@TechLanders libcontainerd]# docker run -i -t centos /bin/bash
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
d9aaf4d82f24: Pull complete
Digest: sha256:eba772bac22c86d7d6e72421b4700c3f894ab6e35475a34014ff8de74c10872e
Status: Downloaded newer image for centos:latest
[root@9a06b1a61fc5 /]#
```

```
[root@TechLanders overlay]# ls -lrt /var/lib/docker/image/overlay2/repositories.json
-rw-------. 1 root root 545 Sep 18 03:17 /var/lib/docker/image/overlay2/repositories.json
```

How Container works

[root@TechLanders overlay]# cat repositories.json

{"Repositories":{"centos":{"centos:latest":"**sha256:196e0ce0c9fbb31da595b893dd39bc9fd4aa78a474bbdc21459a3ebe855b7768**","centos@sha256:**eba772bac22c86d7d6e72421b4700c3f894ab6e35475a34014ff8de74c10872e**":"sha256:196e0ce0c9fbb31da595b893dd39bc9fd4aa78a474bbdc21459a3ebe855b7768"},"hello-world":{"hello-world:latest":"sha256:05a3bd381fc2470695a35f230afefd7bf978b566253199c4ae5cc96fafa29b37","hello-world@sha256:1f19634d26995c320618d94e6f29c09c6589d5df3c063287a00e6de8458f8242":"sha256:05a3bd381fc2470695a35f230afefd7bf978b566253199c4ae5cc96fafa29b37"}}}

[root@TechLanders overlay]# docker image list

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
centos	latest	**196e0ce0c9fb**	3 days ago	197MB
hello-world	latest	05a3bd381fc2	5 days ago	1.84kB

[root@TechLanders overlay]# docker image inspect centos

More Useful - Container

- Do something in our container:

Lets suppose we try to use “talk” for communication.

Let’s check how many packages are installed:

rpm –qa | wc –l

Run in background - Container

- Containers can be started in the background, with the -d flag (daemon mode):

```
docker run -d jpetazzo/clock
```

We don't see the output of the container.

But don't worry: Docker collects that output and logs it!

docker ps -a

docker logs <conatiner-id>

Docker gives us the ID of the container.

List Running Containers

- With docker ps, just like the UNIX ps command, lists running processes.

```
docker ps
docker ps –l
docker ps –a
docker ps -q
```

The (truncated) ID of our container.

The image used to start the container.

That our container has been running (Up) for a couple of minutes.

Now, start multiple containers and use “docker ps” to list them.

Stop our Container

There are two ways we can terminate our detached container.

- Killing it using the docker “kill” command.
- Stopping it using the docker “stop” command.

The first one stops the container immediately, by using the KILL signal.

The second one is more graceful. It sends a TERM signal, and after 10 seconds, if the container has not stopped, it sends KILL.

Removing Container

Let’s remove our container:

```
docker rm <yourContainerID>
```

Restarting a Container

When a container has exited, it is in stopped state.

It can then be restarted with the “start” command.

```
docker start <containerID>
```

The container must be running.

You can also use restart command

```
docker restart <container-id>
```

LAB1

Create one Ubuntu Container in interactive and terminal mode

Exit out of the container

Check the status of container

Restart the container

Get attached to the container

Get out of container without exiting ,, stop the container

Remove the container

Create the container now in detached mode and follow the same steps ..

LAB2

Create a new nginx container using (-d) option.

Docker run –d nginx

Now check the difference using docker ps –a command

Try to access this container

Docker attach <container-ID>

cat /etc/os-release (THIS WILL NOT WORK) (BECAUSE WE ARE NOT USING INETRACTIVE IN THE CREATION)

Use docker exec -it <CONATINER-ID> /bin/bash

Remove everything

Docker stop

Docker rm

Docker run –dt nginx (YOU NEED TO PROVIDE A TERMINAL HERE) → THIS IS TRUE WITH BASE IMAGES

Docker attach <container-ID>

LAB3

On your host machine install httpd package

#For centos

- Yum install httpd –y

#For ubuntu :

-- apt-get install apache2

Verify installation:

rpm –qa | grep –i httpd

#For ubuntu

find / -name apache2 or dpkg –list | grep –i apache2

Create a container :

#For ubuntu :

docker run –it –name c1 ubuntu

docker run –it centos

Learning - Containers have isolated independent root filesystem

LAB3 cont..

Check the package inside it:

dpkg --list | grep -i apache

*Install the package if not there :

apt-get update –y

apt-get install apache2

dpkg --list | grep apache2

#For centos :

rpm –qa | grep –i httpd

•If facing an error while installing httpd in centos container run below :

sudo sed -i 's/mirrorlist/#mirrorlist/g' /etc/yum.repos.d/CentOS-*

sudo sed -i 's|#baseurl=http://mirror.centos.org|baseurl=http://vault.centos.org|g' /etc/yum.repos.d/CentOS-*

sudo yum update –y

Docker Networking

- Docker Networking allows you to create a Network of Docker Containers managed by a master node called the manager.
- Containers inside the Docker Network can talk to each other by sharing packets of information.

Docker Networking

Network Drivers

1. **Bridge:** If you build a container without specifying the kind of driver, the container will only be created in the bridge network, which is the default network.
2. **Host:** Containers will not have any IP address they will be directly created in the system network which will remove isolation between the docker host and containers.
3. **None:** IP addresses won't be assigned to containers. These containments are not accessible to us from the outside or from any other container.

Docker Network Command

1. **Docker network :** The Docker Network command is the main command that would allow you to create, manage, and configure your Docker Network.
2. **docker network ls :** To list all the Docker Networks, you can use the **list** command.
3. **Docker Network Create command :** With the help of the "Create" command, we can create our own docker network and can deploy our containers in it.

   docker network create --driver <driver-name> <bridge-name>
4. **Using the Docker Network Connect command :** Using the "Connect" command, you can connect a running Docker Container to an existing Network.

   docker network connect <network-name> <container-name or id>
5. **Using the Docker Network Disconnect command :** The disconnect command can be used to remove a Container from the Network.

   docker network disconnect <network-name> <container-name or id>

Docker Volume

Docker Storage

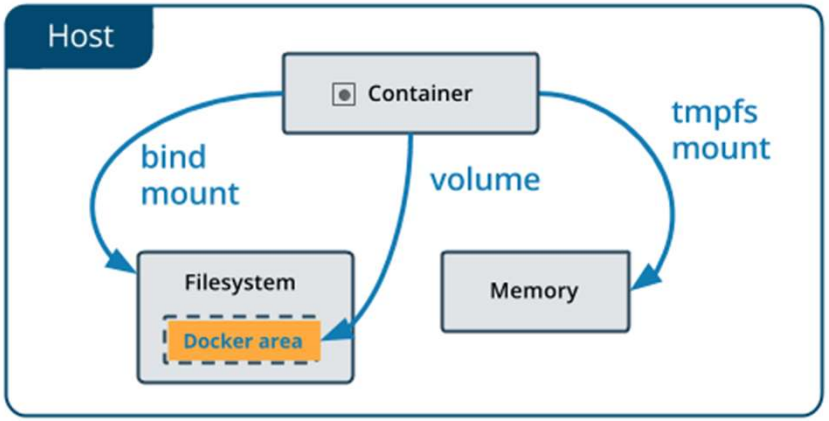
By default all files created inside a container are stored on a writable container layer. This means that:

- The data doesn't persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.
- A container's writable layer is tightly coupled to the host machine where the container is running. You can't easily move the data somewhere else.
- Writing into a container's writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel.
- This extra abstraction reduces performance as compared to using data volumes, which write directly to the host filesystem.

Docker Storage

- Docker has two options for containers to store files on the host machine, so that the files are persisted even after the container stops

    1. Volumes
    2. Bind mounts.



Docker Storage

1. **Volumes** are stored in a part of the host filesystem which is managed by Docker (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
2. **Bind mounts** may be stored anywhere on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time

Docker Storage

Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container.
- Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

Docker Volume Commands

1. Create a volume : A Docker volume can be created and managed outside of a container by using the following command.

$ docker volume create [volume_name]

A directory for the volume is automatically created under the /var/lib/docker/volume/path folder by Docker on the host.

2. List the volumes : To list the volumes, run the following command.

$ docker volume list

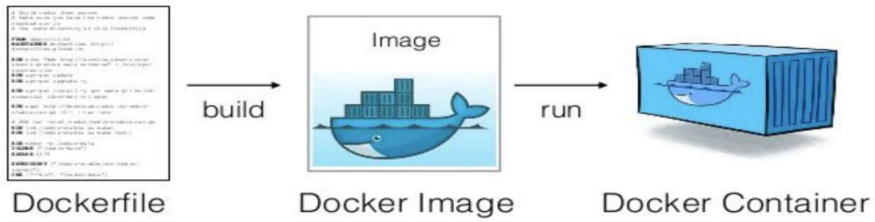
3. **Mounting Docker Volumes :** After creating the Volume, the next step is to mount the Volume to Docker Containers. We will create a Docker Container with the Ubuntu base Image and mount the data Volume to that Container using the -v flag.

Option 1 : docker run -it -v volumename:/shared-volume --name my-container-01 ubuntu

Option 2: docker run --mount source=[volume_name],destination=[path_in_container] [docker_image]

Docker Image

- Docker Image is an executable package of software that includes everything needed to run an application.
- This image informs how a container should instantiate, determining which software components will run and how.
- An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.
- A container is a runnable instance of an image.



Create Docker Image

You can create a docker image in a two ways.

1. Using the running container
2. Using the Docker file

The Process of Creating Images From Containers

At a high level, the process of creating a Docker image from a container involves three steps:

1. **Create a container from an existing image:** The first step is to choose a base image you want to customize and run it as a container.
2. **Make changes to the container:** Once you have the container up and running, you make changes to it. You could modify files, install additional software or do whatever you need to meet your requirements.
3. **Commit the changes to create a new image:** After you've made the desired changes to the container, the next step is to commit those changes to create a brand-new Docker image. Now, you can use the new image to spin up new containers with your customizations.

docker container commit -a "sandeep" -m "Changed default c1 welcome message" c1 image1

Note that it's considered best practice to use the -a flag to sign the image with an author and include a commit message using the -m flag.

Docker File

- Docker can build images automatically by reading the instructions from a Dockerfile.
- A Docker file is a text document that contains all the commands a user could call on the command line to assemble an image.
- A Dockerfile is a plain text file that contains instructions for building a Docker container image.
- Docker containers are lightweight, portable, and consistent environments that encapsulate an application and its dependencies.
- The default filename to use for a Dockerfile is Dockerfile, without a file extension.
- Using the default name allows you to run the docker build command without having to specify additional command flags

```
FROM ubuntu:22.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Docker File

Here are some important rules to keep in mind when working with Dockerfiles:

1. **Use a Minimal Base Image**
2. **Single Responsibility Principle**
3. **Combine Commands**
4. **Cache Efficiency**
5. **Use Specific Tags**
6. **Avoid Installing Unnecessary Packages**
7. **Clear Temporary Files**
8. **Minimize Image Size**
9. **Use COPY Instead of ADD**
10. **Use ENV for Configuration**
11. **Protect Sensitive Information**
12. **Optimize Layer Ordering**
13. **LABEL for Metadat**
14. **User Privileges**
15. **Document Your Dockerfile**

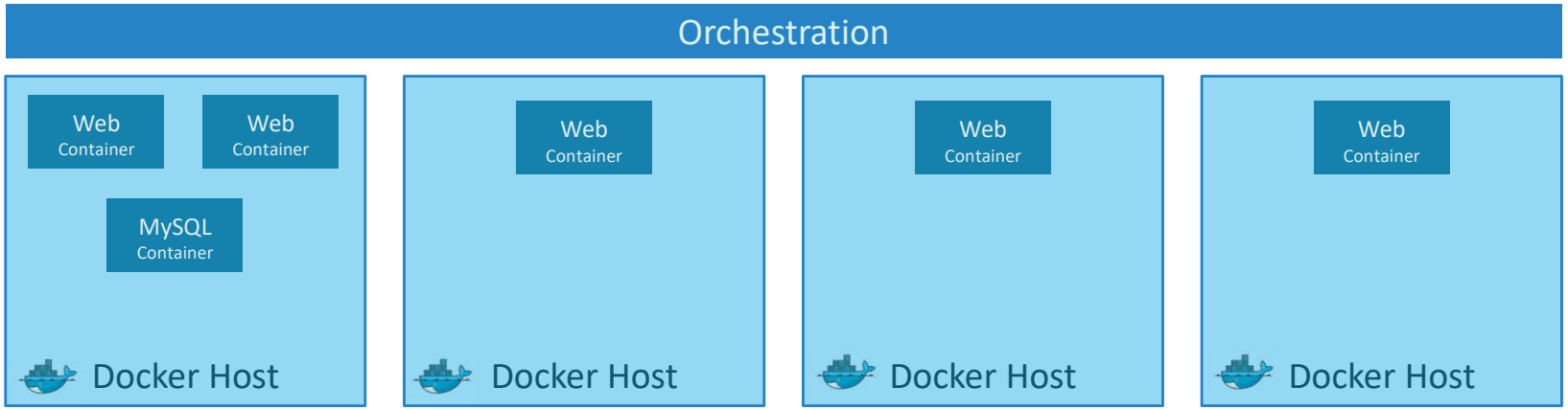
Container Orchestration

Sandeep Sharma

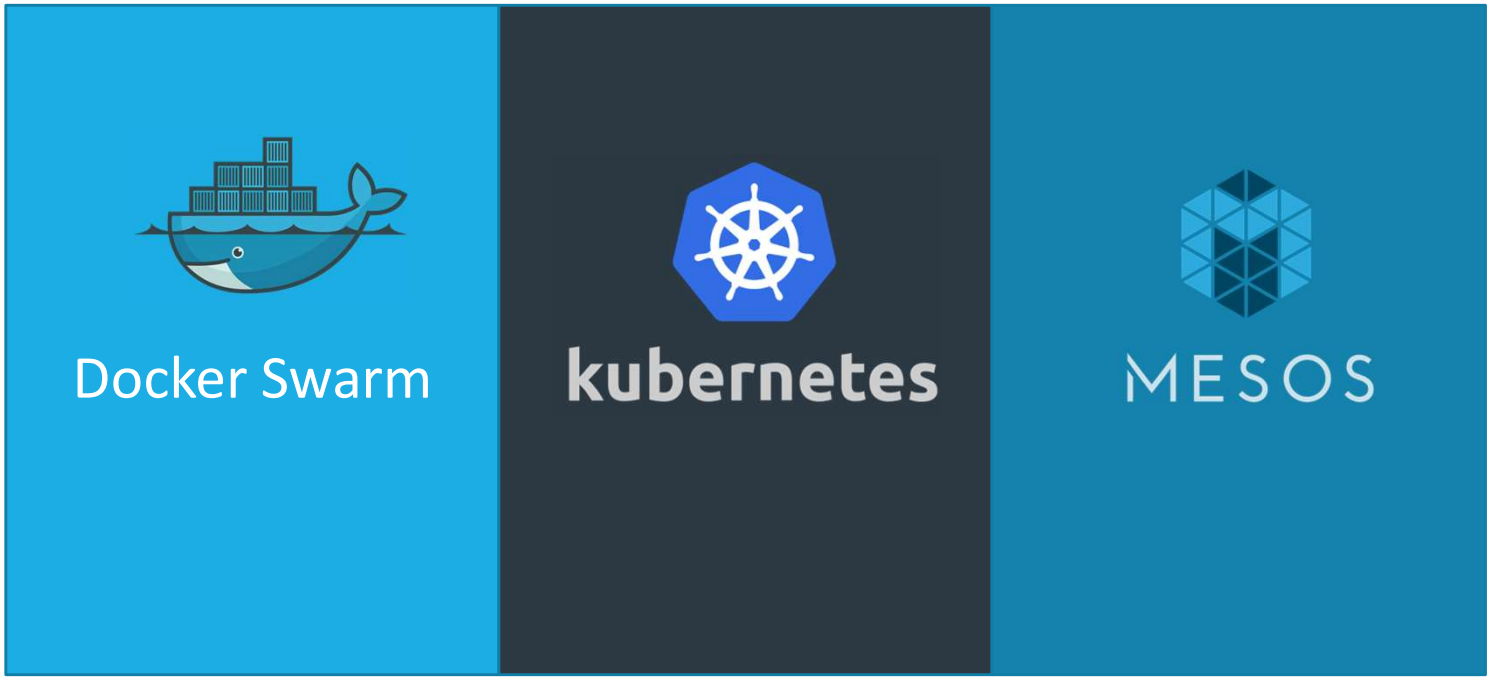
Containers Limitation?

High Availability?
Overlay Network?
Application Centric or Infra Centric?
Versioning of Application – Rollout, Rollback?
Scaling?
Autoscaling?
Monitoring?
Dependency between containers?

Container orchestration



Orchestration Technologies



What is Kubernetes?

The Kubernetes project was started by Google in 2014.

Kubernetes builds upon a decade and a half of experience that Google has with running production workloads at scale.

Kubernetes can run on a range of platforms, from your laptop, to VMs on a cloud provider, to rack of bare metal servers.

Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure.

portable: with all public, private, hybrid, community cloud

self-healing: auto-placement, auto-restart, auto-replication, auto-scaling

Why Kubernetes

Kubernetes can schedule and run application containers on clusters of physical or virtual machines.

host-centric infrastructure to a **container-centric** infrastructure.

Orchestrator

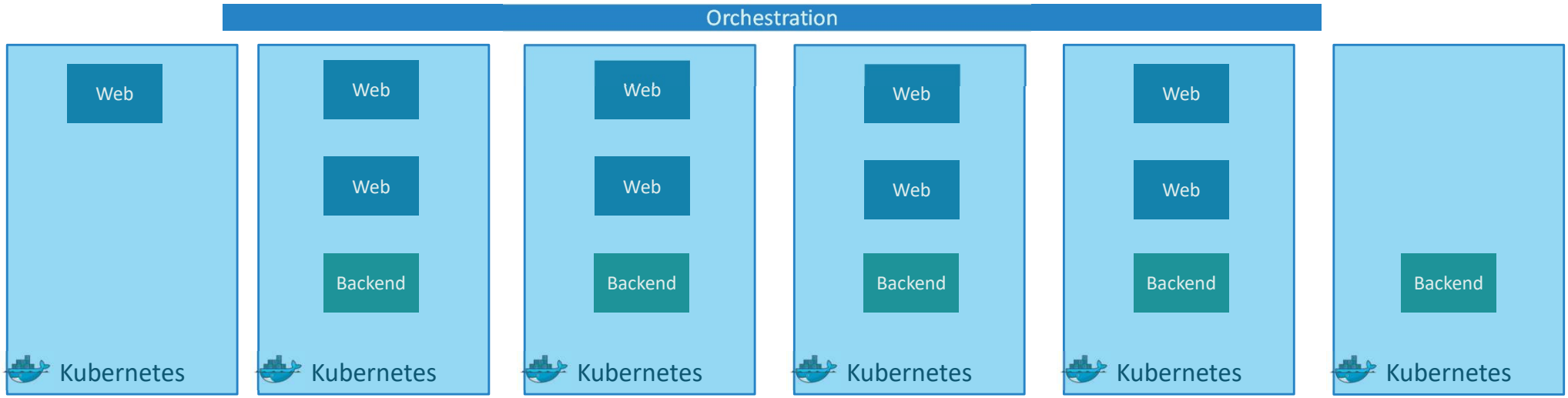
Load balancing

Auto Scaling

Application Health checks

Rolling updates

Kubernetes Advantage



And that is kubernetes..

Setup

Minikube

Kubeadm

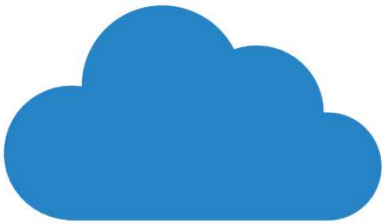
Google Cloud Platform

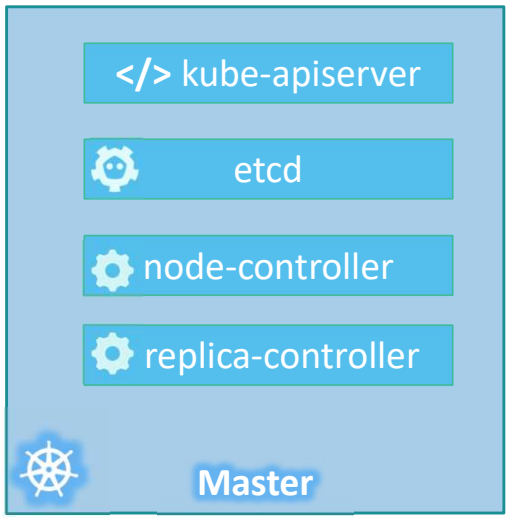
Amazon Web Services

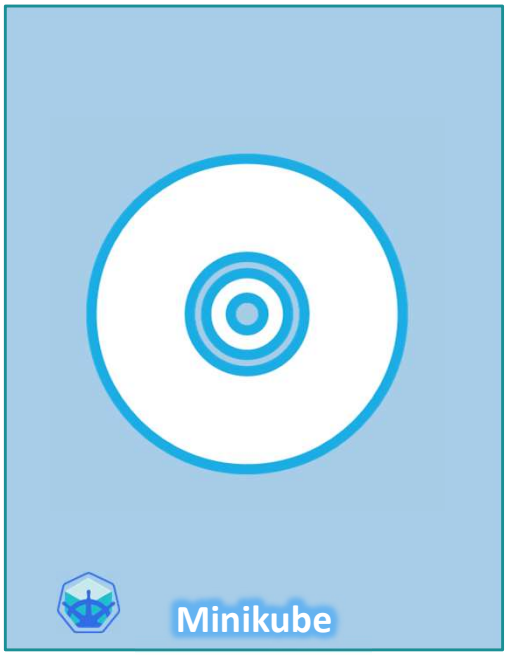
play-with-k8s.com

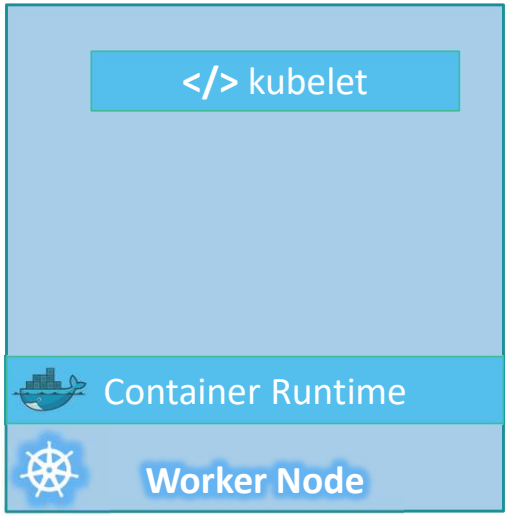
Setup Kubernetes

Minikube

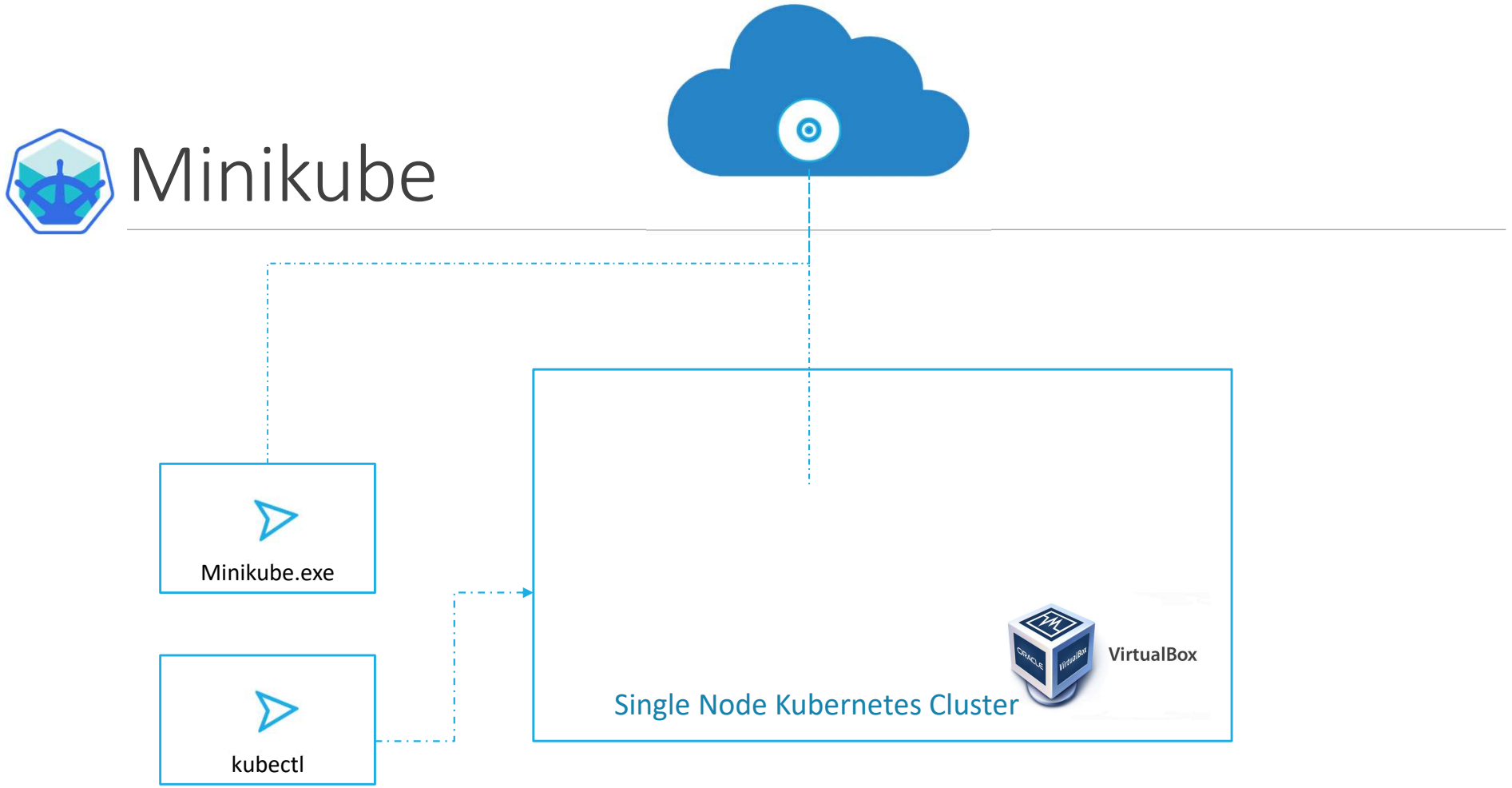



</> kube-apiserver
etcd
node-controller
replica-controller
Master


Minikube


</> kubelet
Container Runtime
Worker Node

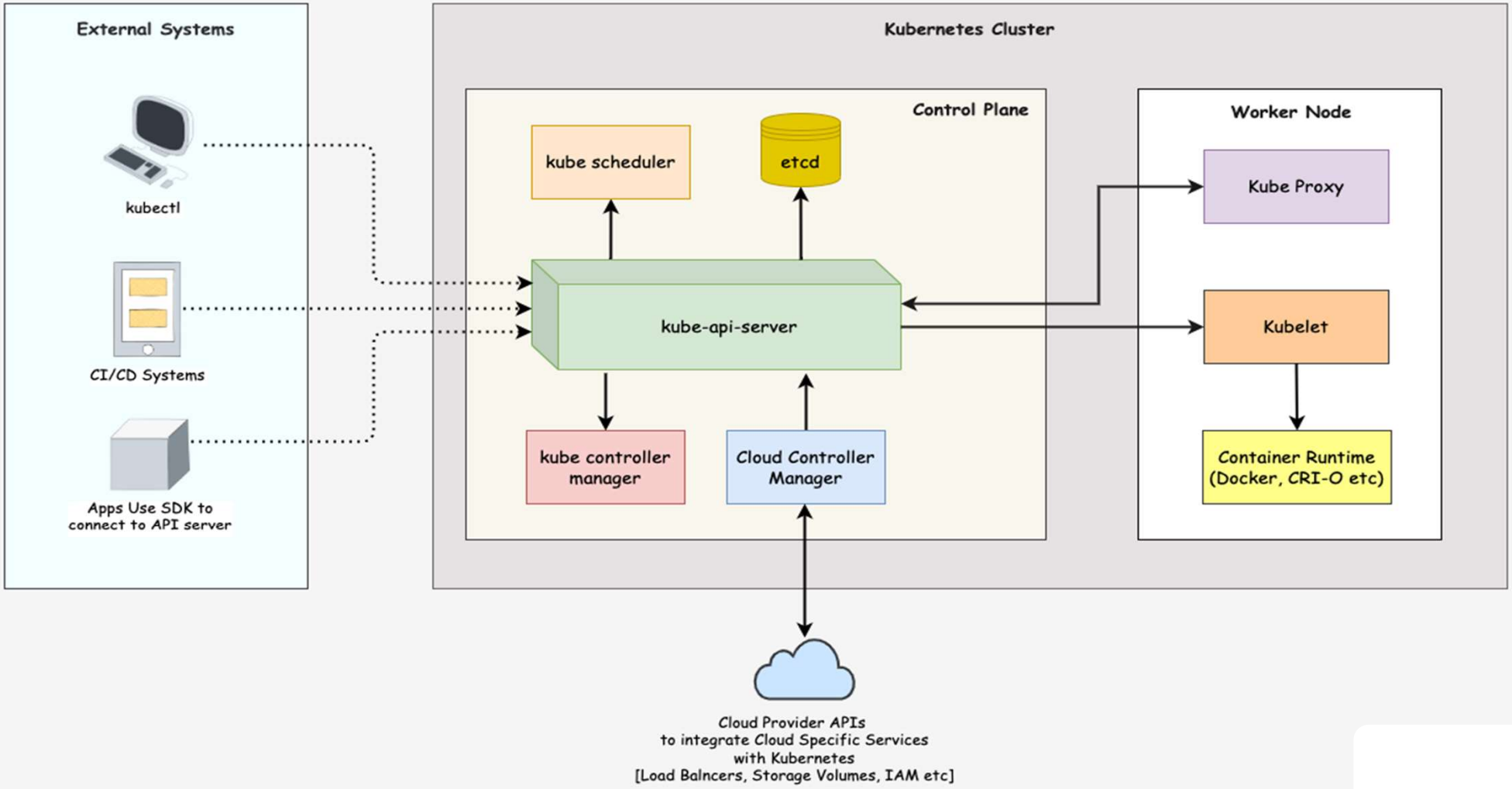
Minikube


Minikube.exe
kubectl
Single Node Kubernetes Cluster
VirtualBox

Demo

minikube

Setup



Kubernetes Cluster

A Kubernetes cluster consists of two types of resources:

Master: Which coordinates with the cluster

The Master is responsible for managing the cluster. The master coordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.

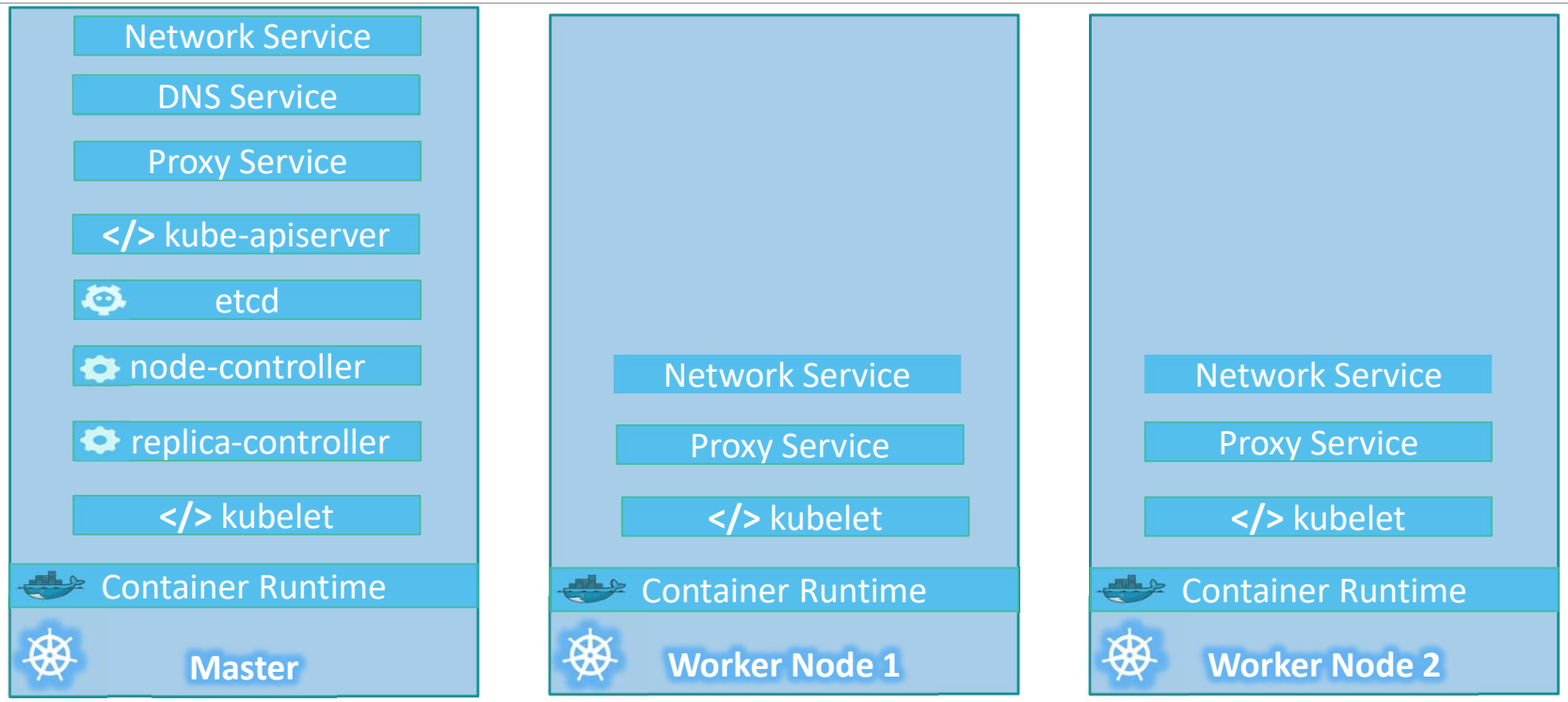
Nodes: Are the workers that run application

A node is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster.

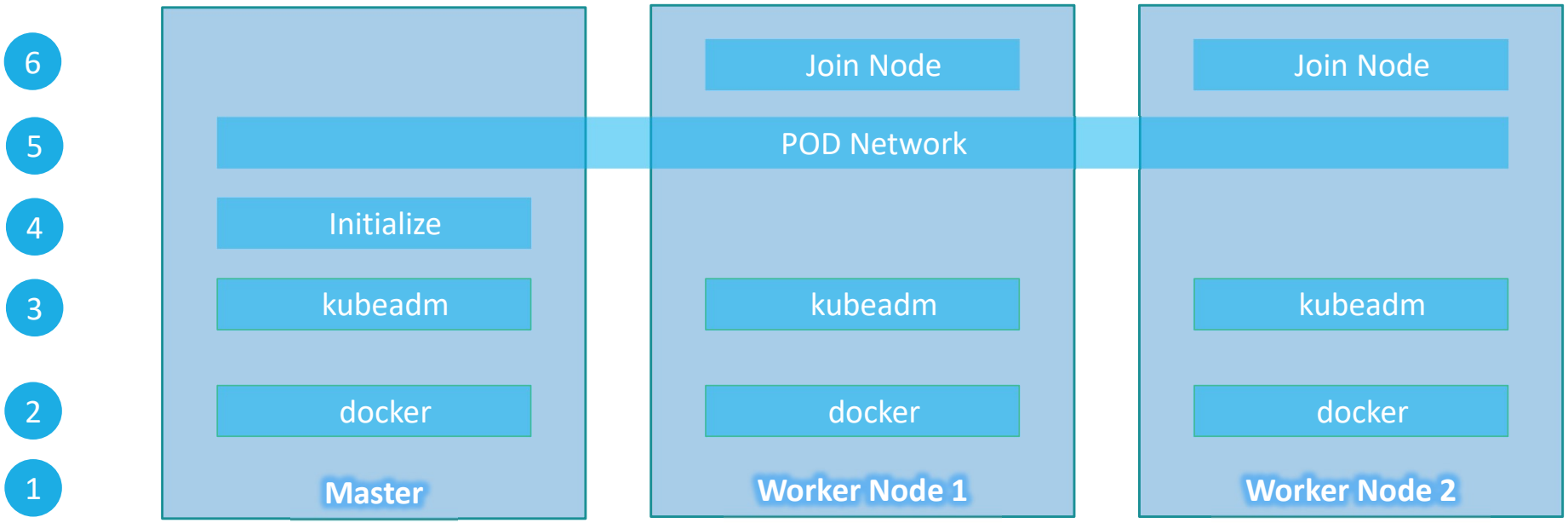
Masters manage the cluster and the nodes are used to host the running applications.

The nodes communicate with the master using the Kubernetes API, which the master exposes.

Kubernetes Components



Steps


6
5
4
3
2
1
Join Node
Join Node
POD Network
Initialize
kubeadm
kubeadm
kubeadm
docker
docker
docker
Master
Worker Node 1
Worker Node 2

Demo

kubeadm

Note: Pod Network Providers haven’t yet updated codes for V1.11 version of K8’s

Use: --feature-gates=CoreDNS=false along with POD network to use kube-dns services

Setup - kubeadm

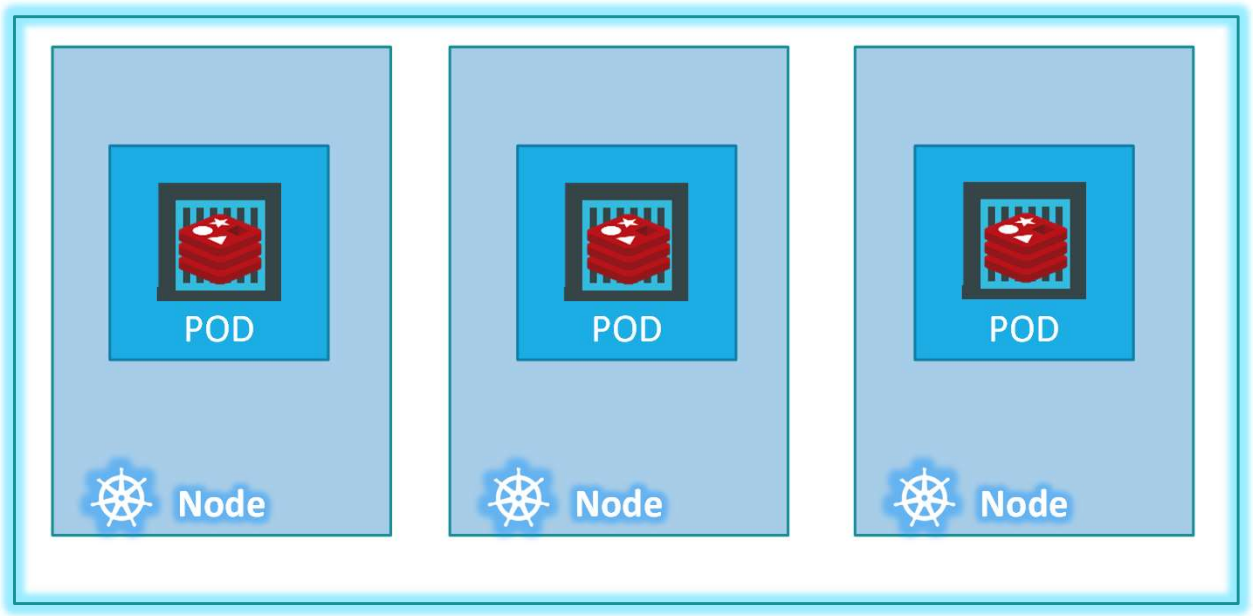
POD

Assumptions

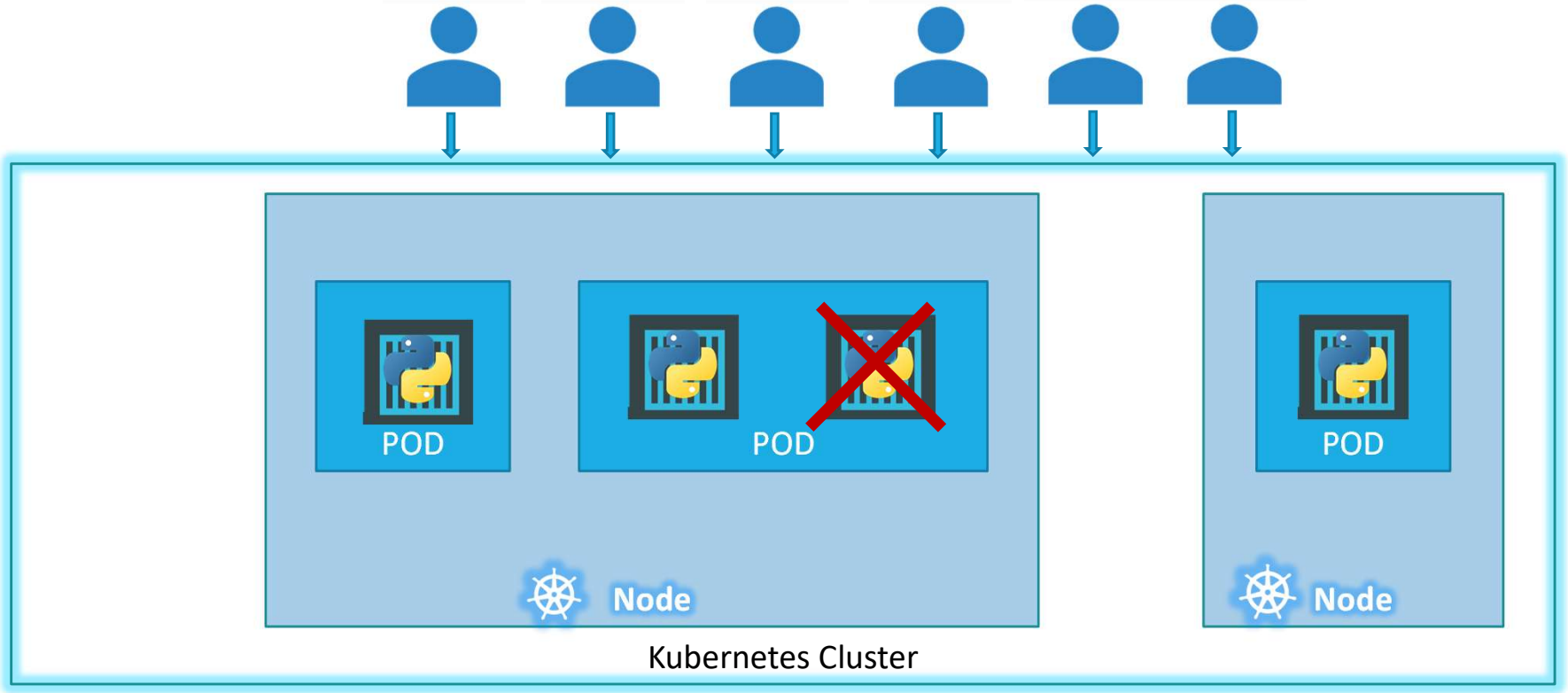
Docker Image

Kubernetes Cluster

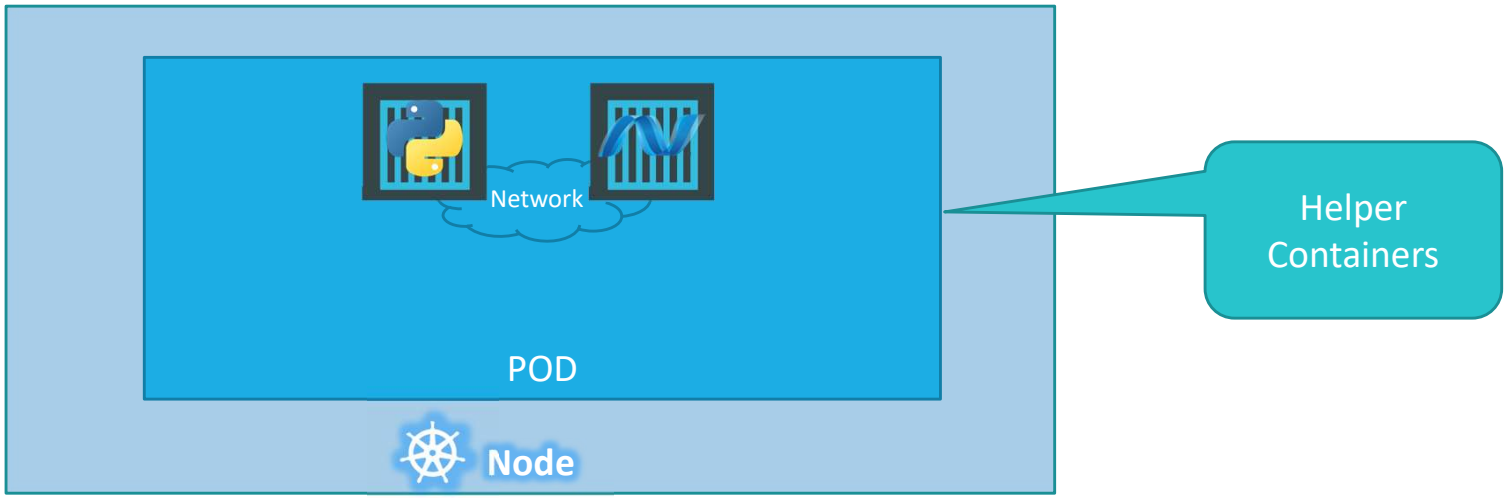
POD



POD



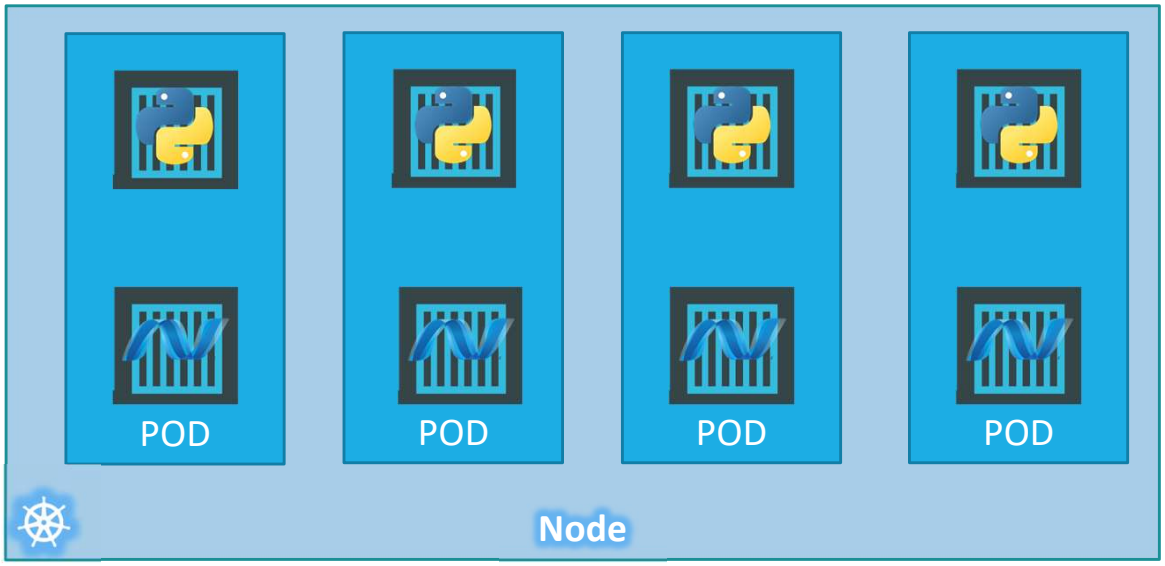
Multi-Container PODs



PODs Again!

```
docker run python-app
docker run python-app
docker run python-app
docker run python-app

docker run helper –link app1

docker run helper –link app2

docker run helper –link app3

docker run helper –link app4
```

App	Helper	Volume
Python1	App1	Vol1
Python2	App2	Vol2



Note: I am avoiding networking and load balancing details to keep explanation simple.

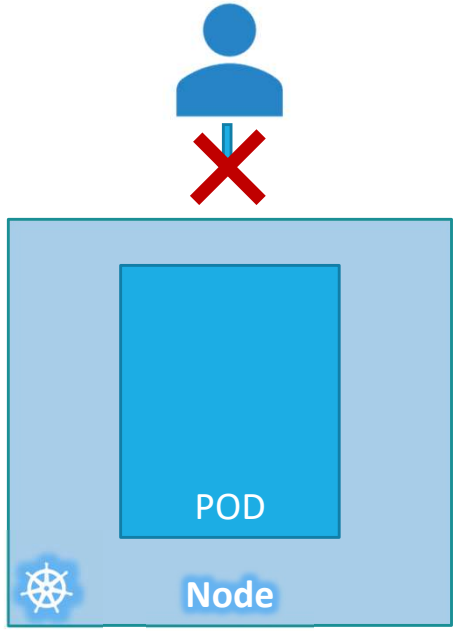
kubectl

```
kubectl run nginx --image nginx
```

```
kubectl get pods
```

```
C:\Kubernetes>kubectl get pods
NAME                    READY     STATUS              RESTARTS   AGE
nginx-8586cf59-whssr    0/1       ContainerCreating   0          3s
```

```
C:\Kubernetes>kubectl get pods
NAME                    READY     STATUS    RESTARTS   AGE
nginx-8586cf59-whssr    1/1       Running   0          8s
```



Demo

POD

YAML
Introduction

Sandeep Sharma

POD

With YAML

Commands

```
> kubectl get pods
NAME            READY     STATUS    RESTARTS   AGE
myapp-pod       1/1       Running   0          20s
```

```
> kubectl describe pod myapp-pod
Name:         myapp-pod
Namespace:    default
Node:         minikube/192.168.99.100
Start Time:   Sat, 03 Mar 2018 14:26:14 +0800
Labels:       app=myapp
              name=myapp-pod
Annotations:  <none>
Status:       Running
IP:           10.244.0.24
Containers:
  nginx:
    Container ID:   docker://830bb56c8c42a86b4bb70e9c1488fae1bc38663e4918b6c2f5a783e7688b8c9d
    Image:          nginx
    Image ID:       docker-pullable://nginx@sha256:4771d09578c7c6a65299e110b3ee1c0a2592f5ea2618d23e4ffe7a4cab1ce5de
    Port:           <none>
    State:          Running
      Started:      Sat, 03 Mar 2018 14:26:21 +0800
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-x95w7 (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
Events:
  Type    Reason                 Age   From               Message
  ----    ------                 ----  ----               -------
  Normal  Scheduled              34s   default-scheduler  Successfully assigned myapp-pod to minikube
  Normal  SuccessfulMountVolume  33s   kubelet, minikube  MountVolume.SetUp succeeded for volume "default-token-x95w7"
  Normal  Pulling                33s   kubelet, minikube  pulling image "nginx"
  Normal  Pulled                 27s   kubelet, minikube  Successfully pulled image "nginx"
  Normal  Created                27s   kubelet, minikube  Created container
  Normal  Started                27s   kubelet, minikube  Started container
```

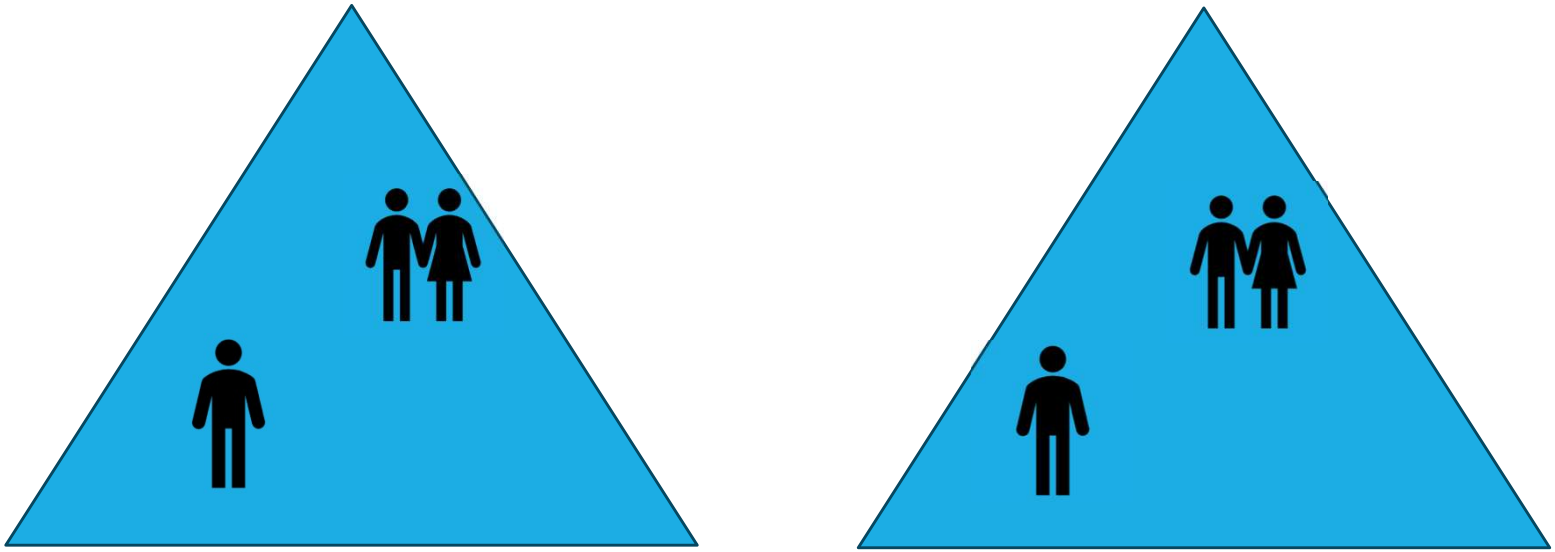
Demo

POD Using YAML

Namespace

Namespace

A Kubernetes namespace helps separate a cluster into logical units. It helps granularly organize, allocate, manage, and secure cluster resources.



Kubernetes Namespaces Concepts

There are two types of Kubernetes namespaces: Kubernetes system namespaces and custom namespaces.

Default Kubernetes namespaces

Here are the four default namespaces Kubernetes creates automatically:

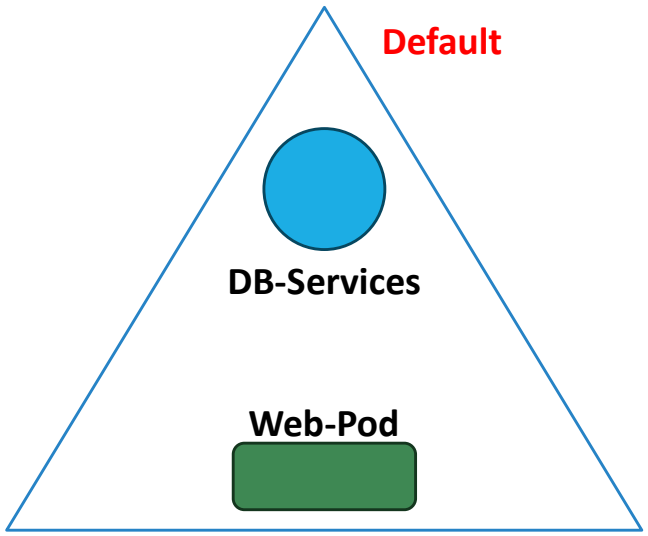
•**default**—a default space for objects that do not have a specified namespace.

•**kube-system**—a default space for Kubernetes system objects, such as kube-dns and kube-proxy, and add-ons providing cluster-level features, such as web UI dashboards, ingresses, and cluster-level logging.

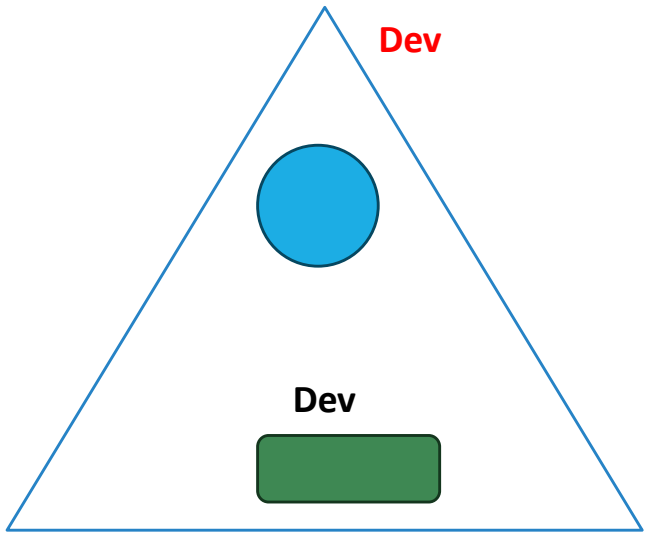
•**kube-public**—a default space for resources available to all users without authentication.

•**kube-node-lease**—a default space for objects related to cluster scaling.

Namespace



MySQL.Connect(“DB-Services”)

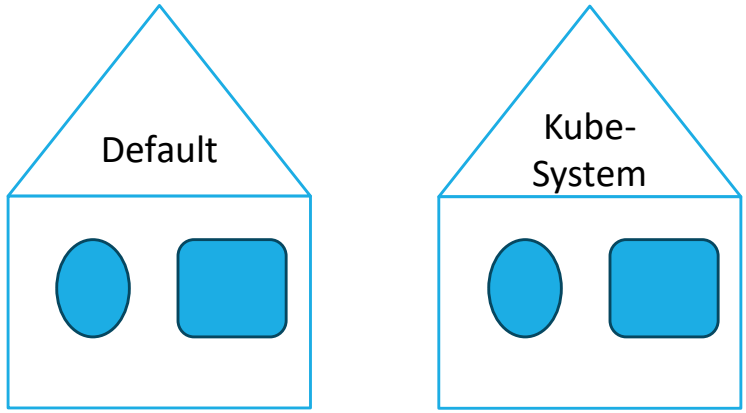


MySQL.Connect(“DB-Services.Dev.svc.cluster.local”)

Get Pods in Namespace

```
root@master:~# kubectl get pods
No resources found in default namespace.
```

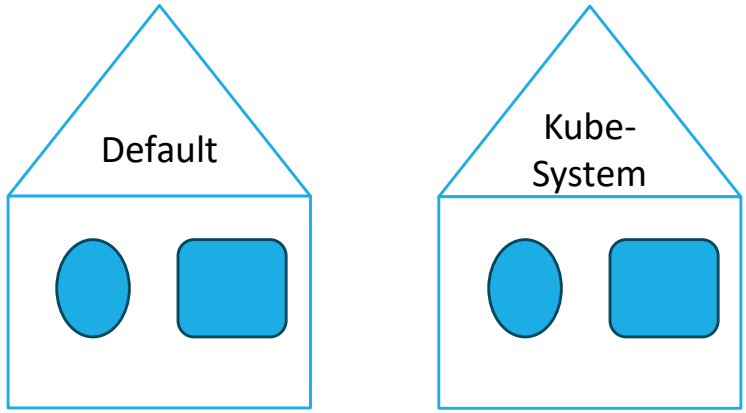
```
root@master:~# kubectl get pods -n kube-system
NAME                                   READY   STATUS    RESTARTS   AGE
calico-kube-controllers-68d86f8988-666zt   1/1     Running   1          3d
calico-node-84lnf                      1/1     Running   1          3d
calico-node-gbstd                      1/1     Running   1          3d
calico-node-khmfs                      1/1     Running   1          3d
coredns-558bd4d5db-f2s8v               1/1     Running   1          3d
coredns-558bd4d5db-q4p8b               1/1     Running   1          3d
etcd-master                            1/1     Running   1          3d
kube-apiserver-master                  1/1     Running   1          3d
kube-controller-manager-master         1/1     Running   1          3d
kube-proxy-bxv7c                       1/1     Running   1          3d
kube-proxy-dkwlq                       1/1     Running   1          3d
kube-proxy-kp2kv                       1/1     Running   1          3d
kube-scheduler-master                  1/1     Running   1          3d
```



Get Pods in Namespace

```
root@master:~# kubectl get pods
No resources found in default namespace.
```

```
root@master:~# kubectl get pods -n kube-system
NAME                                   READY   STATUS    RESTARTS   AGE
calico-kube-controllers-68d86f8988-666zt   1/1     Running   1          3d
calico-node-84lnf                      1/1     Running   1          3d
calico-node-gbstd                      1/1     Running   1          3d
calico-node-khmfs                      1/1     Running   1          3d
coredns-558bd4d5db-f2s8v               1/1     Running   1          3d
coredns-558bd4d5db-q4p8b               1/1     Running   1          3d
etcd-master                            1/1     Running   1          3d
kube-apiserver-master                  1/1     Running   1          3d
kube-controller-manager-master         1/1     Running   1          3d
kube-proxy-bxv7c                       1/1     Running   1          3d
kube-proxy-dkwlq                       1/1     Running   1          3d
kube-proxy-kp2kv                       1/1     Running   1          3d
kube-scheduler-master                  1/1     Running   1          3d
```



Add Pod in specific Namespace

```
root@master:~# vi poddefination.yaml
root@master:~# kubectl create -f poddefination.yaml
pod/pod created
root@master:~# kubectl get pod
NAME   READY   STATUS    RESTARTS   AGE
pod    1/1     Running   0          15s
root@master:~#
```

```
root@master:~# kubectl create -f poddefination.yaml --namespace=dev
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  namespace: your-namespace-name
spec:
  containers:
    - name: container-1
      image: nginx:latest
    - name: container-2
      image: busybox:latest
```

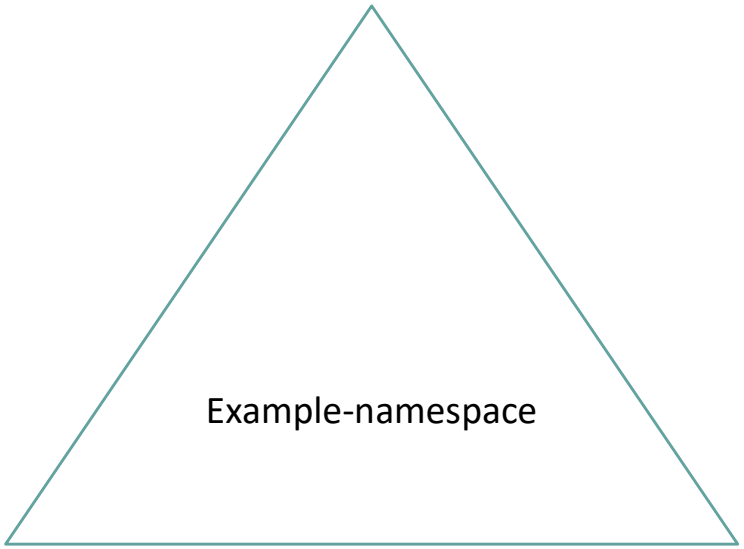
Add Pod in specific Namespace

```
root@master:~# vi poddefination.yaml
root@master:~# kubectl create -f poddefination.yaml
pod/pod created
root@master:~# kubectl get pod
NAME   READY   STATUS    RESTARTS   AGE
pod    1/1     Running   0          15s
root@master:~#
```

```
root@master:~# kubectl create -f poddefination.yaml --namespace=dev
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  namespace: your-namespace-name
spec:
  containers:
    - name: container-1
      image: nginx:latest
    - name: container-2
      image: busybox:latest
```

Create Namespace

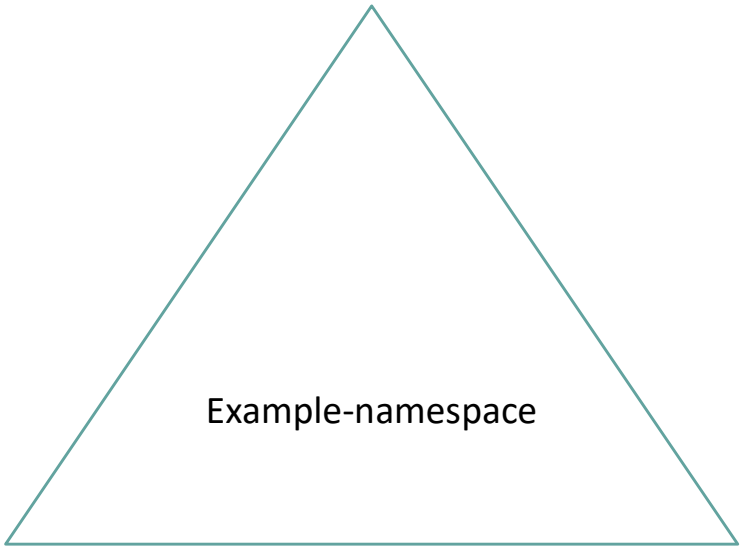



```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: example-namespace
```

```
kubectl apply -f namespace.yaml
```

```
kubectl create namespace my-namespace
```

Create Namespace




```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: example-namespace
```

```
kubectl apply -f namespace.yaml
```

```
kubectl create namespace my-namespace
```

Tips & Tricks

Working YAML Files

Coding Exercises

Resources

Link to Versions and Groups - https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.9/#replicaset-v1-apps

https://plugins.jetbrains.com/plugin/9354-kubernetes-and-openshift-resource-support

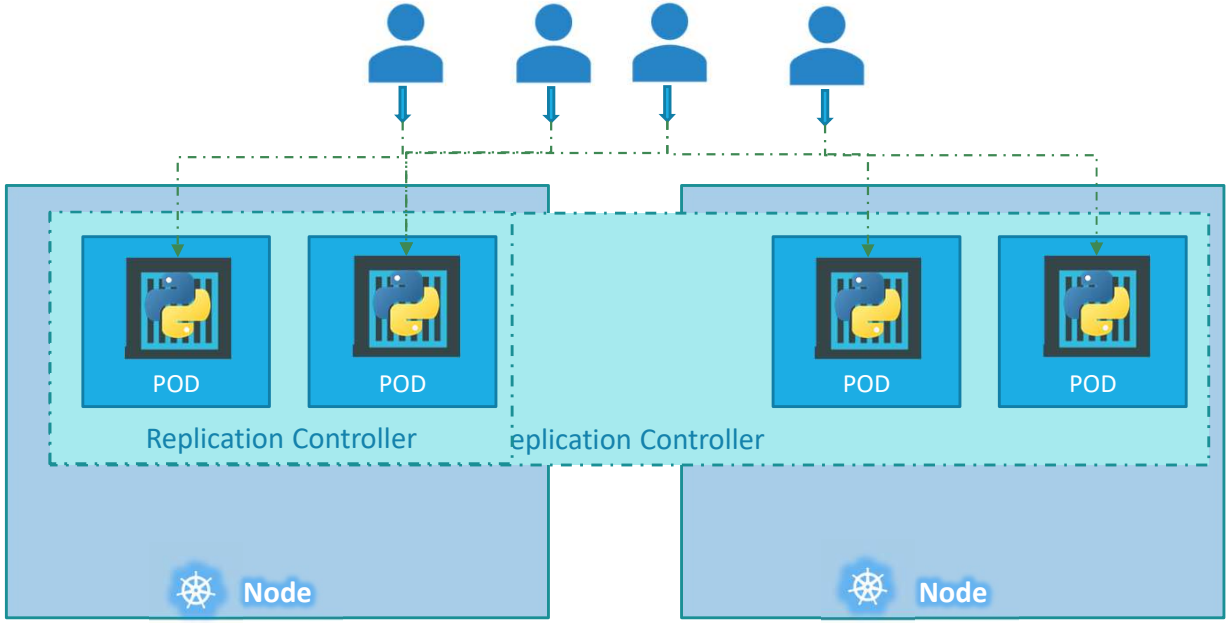
For Pods: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.11/#pod-v1-core

Replication Controller

Sandeep Sharma

High Availability

Load Balancing & Scaling



Replication Controller

Replica Set

Replication Controller

A *ReplicationController* ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

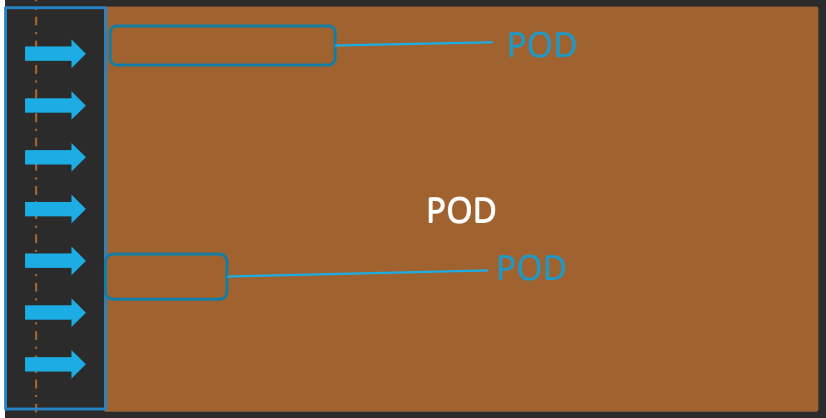
If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods. Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated

rc-definition.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
      app: myapp
      type: front-end
spec:
  template:
    replicas: 3
```

Replication Controller

Replication Controller



pod-definition.yml

```
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

```
> kubectl create -f rc-definition.yml
replicationcontroller "myapp-rc" created
```

```
> kubectl get replicationcontroller
NAME       DESIRED   CURRENT   READY     AGE
myapp-rc   3         3         3         19s
```

```
> kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
myapp-rc-4lvk9   1/1       Running   0          20s
myapp-rc-mc2mf   1/1       Running   0          20s
myapp-rc-px9pz   1/1       Running   0          20s
```

Replica Set

- A ReplicaSet (RS) is a Kubernetes object used to maintain a stable set of replicated pods running within a cluster at any given time.
- A ReplicaSet is a Kubernetes resource used to maintain a specified number of identical pod replicas within a cluster. It is responsible for monitoring the health of the modules it manages and ensuring that the required number of replicas are always running, thus providing self-healing capabilities.
- ReplicaSet automatically detect and recover from module failures by creating new replicas to replace the failed ones, ensuring that the desired state of the application is maintained. With their self-healing feature, ReplicaSet contribute to the overall resilience and high availability of applications in Kubernetes clusters.

replicaset-definition.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-repl
  labels:
      app: myapp
      type: front-end
spec:
  template:
      POD
  replicas: 3
  selector:
     matchLabels:
        type: front-end
```

```
error: unable to recognize "replicaset-
definition.yml": no matches for /, Kind=ReplicaSet
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod
  labels:
     app: myapp
     type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

```
> kubectl create -f replicaset-definition.yml
replicaset "myapp-replicaset" created
> kubectl get replicaset
NAME            DESIRED   CURRENT   READY     AGE
myapp-replicaset   3         3         3         19s
> kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
myapp-replicaset-9ddl9   1/1       Running   0          45s
myapp-replicaset-9jtpx   1/1       Running   0          45s
myapp-replicaset-hq84m   1/1       Running   0          45s
```

Replication Controller v/s Replica Set

Replication Controller	Replica Set
The Replication Controller is the original form of replication in Kubernetes	ReplicaSets are a higher-level API that gives the ability to easily run multiple instances of a given pod
The Replication Controller uses **equality-based selectors** to manage the pods.	ReplicaSets Controller uses **set-based selectors** to manage the pods.
The rolling-update command works with Replication Controllers	The rolling-update command won't work with ReplicaSets.
Replica Controller is deprecated and replaced by ReplicaSets.	Deployments are recommended over ReplicaSets.

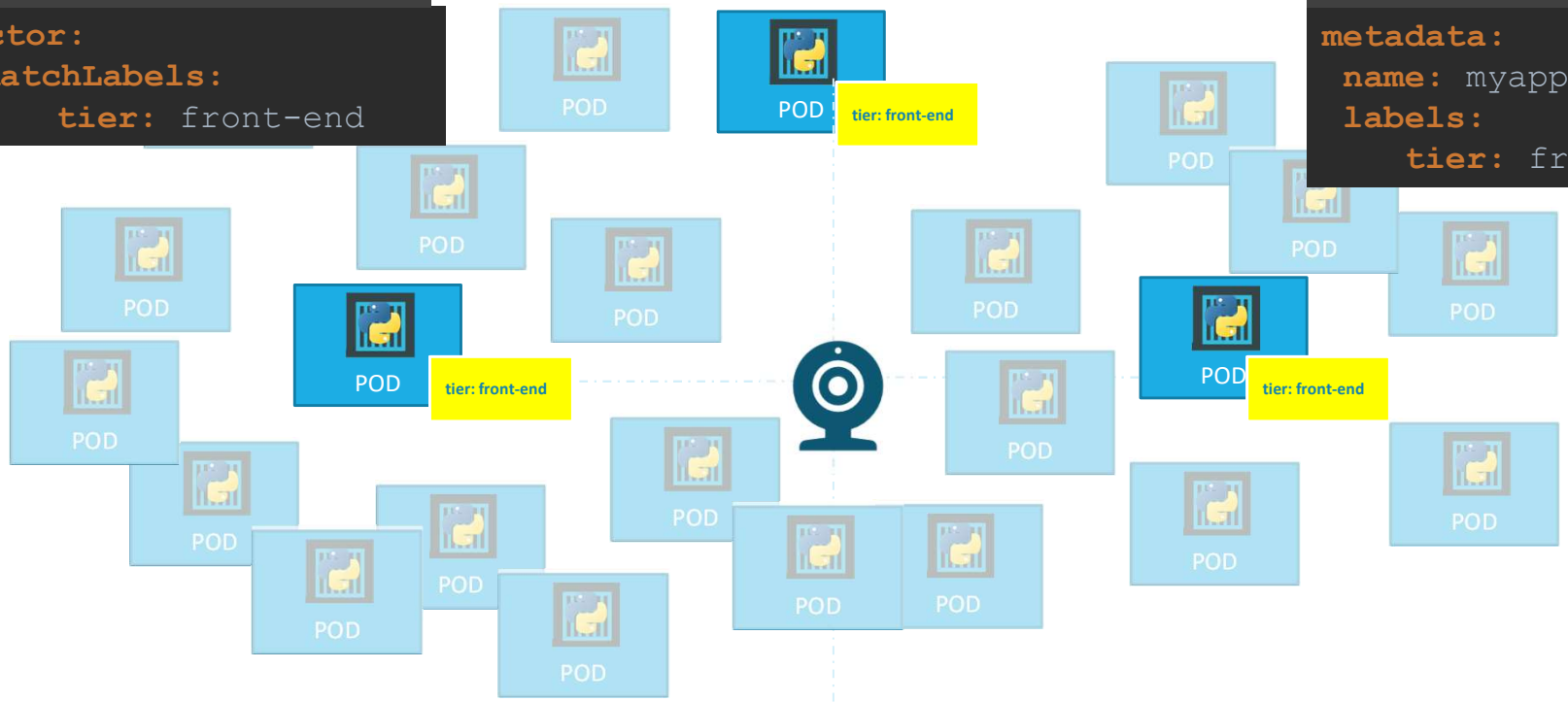
Labels and Selectors

replicaset-definition.yml

```
selector:
    matchLabels:
        tier: front-end
```

pod-definition.yml

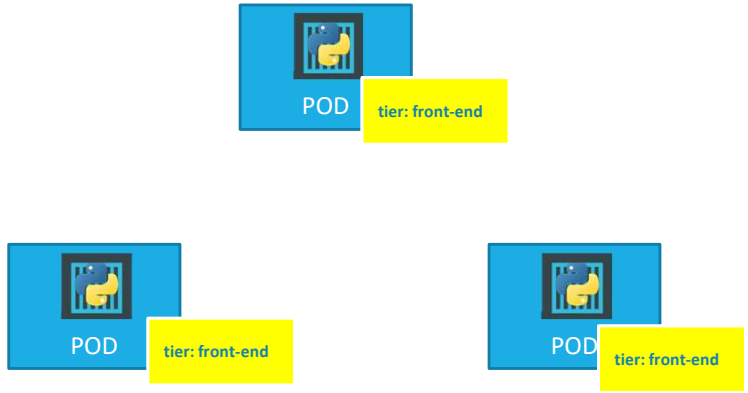
```
metadata:
  name: myapp-pod
  labels:
     tier: front-end
```



replicaset-definition.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 3
  selector:
    matchLabels:
       type: front-end
```

Template



Scale

```
> kubectl replace -f replicaset-definition.yml
```

```
> kubectl scale --replicas=6 -f replicaset-definition.yml
```

```
> kubectl scale --replicas=6 replicaset myapp-replicaset
```

TYPE NAME

replicaset-definition.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 6
  selector:
    matchLabels:
      type: front-end
```

commands

```
> kubectl create –f replicaset-definition.yml
```

```
> kubectl get replicaset
```

```
> kubectl delete replicaset myapp-replicaset
```

*Also deletes all underlying PODs

```
> kubectl replace -f replicaset-definition.yml
```

```
> kubectl scale –replicas=6 -f replicaset-definition.yml
```

Demo

ReplicaSet

ReplicaSet as an Horizontal Pod Autoscaler Target

https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#replicaset-as-an-horizontal-pod-autoscaler-target

Deployment

Sandeep Sharma

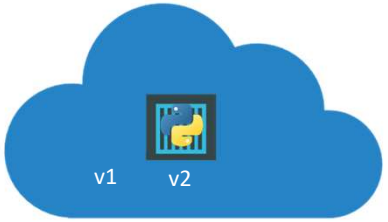
Deployment

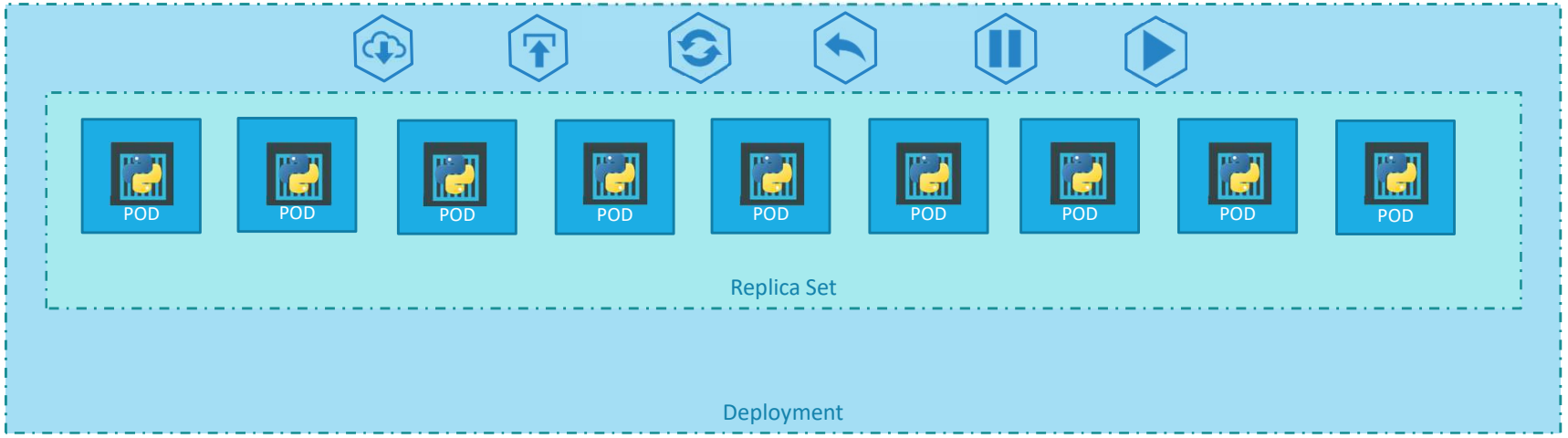
- Deployment refers to a resource that manages the deployment and scaling of containerized applications. It automates the process of updating your application instances, ensuring the desired number of pod replicas are running and can handle rolling updates, rollbacks, and scaling.
- In simpler terms, a Deployment is a way to automate the process of creating and managing multiple replicas of your application, making it easier to manage, update, and scale your application without worrying about the underlying infrastructure.

Deployment Features

- **Scaling**: A deployment allows you to scale the number of replicas of your application either up or down, manually or automatically, based on resource utilization.
- **Rolling updates**: Deployments support rolling updates, where new versions of the application are gradually rolled out, without downtime. Kubernetes replaces the old version with the new one, pod by pod.
- **Rollback**: If something goes wrong during an update, Kubernetes can roll back to a previous, stable version of the deployment.
- **Self-healing**: If a pod crashes or becomes unresponsive, Kubernetes ensures the correct number of replicas are maintained by automatically restarting or replacing failed pods.

Deployment


v1
v2


POD
POD
POD
POD
POD
POD
POD
POD
POD
Replica Set
Deployment

Definition

```
> kubectl create -f deployment-definition.yml
deployment "myapp-deployment" created
```

```
> kubectl get deployments
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
myapp-deployment  3         3         3            3           21s
```

```
> kubectl get replicaset
NAME                          DESIRED   CURRENT   READY     AGE
myapp-deployment-6795844b58   3         3         3         2m
```

```
> kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
myapp-deployment-6795844b58-5rbjl   1/1       Running   0          2m
myapp-deployment-6795844b58-h4w55   1/1       Running   0          2m
myapp-deployment-6795844b58-lfjhv   1/1       Running   0          2m
```

deployment-definition.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
         app: myapp
         type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 3
  selector:
    matchLabels:
       type: front-end
```

commands

```
> kubectl get all
NAME                     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/myapp-deployment   3         3         3            3           9h

NAME                               DESIRED   CURRENT   READY     AGE
rs/myapp-deployment-6795844b58   3         3         3         9h

NAME                                   READY     STATUS    RESTARTS   AGE
po/myapp-deployment-6795844b58-5rbjl   1/1       Running   0          9h
po/myapp-deployment-6795844b58-h4w55   1/1       Running   0          9h
po/myapp-deployment-6795844b58-lfjhv   1/1       Running   0          9h
```

Demo

Deployment

Demo

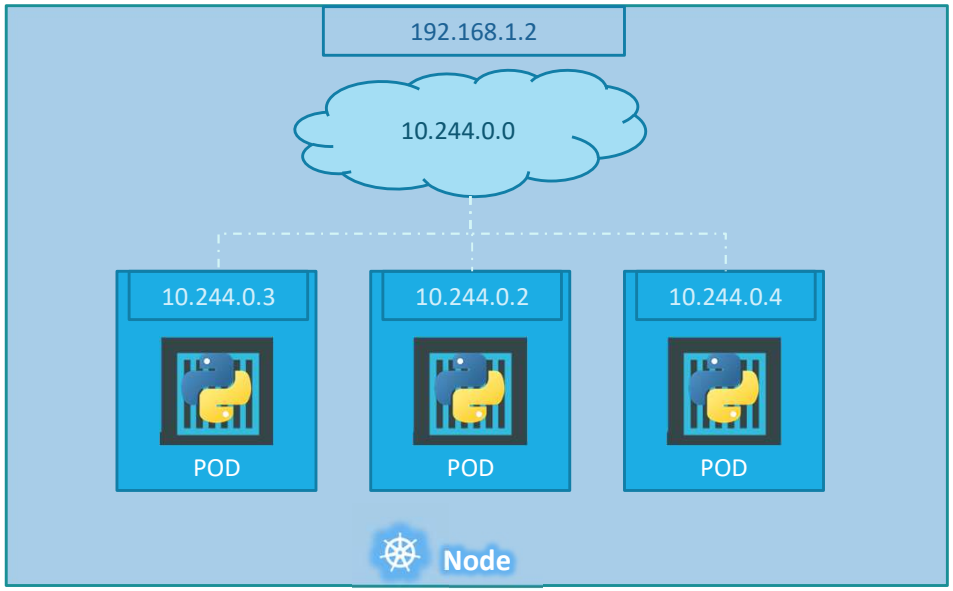
Deployment

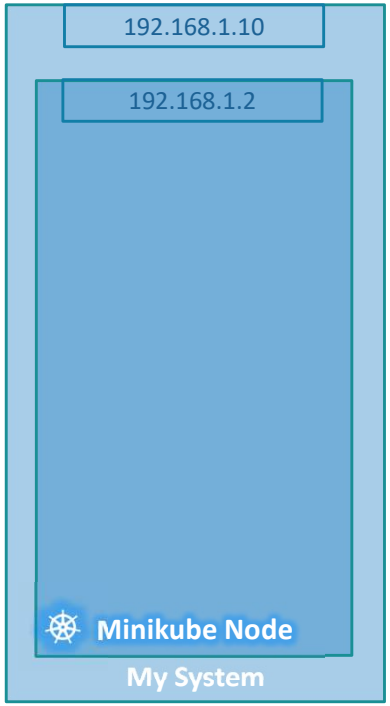
Networking 101

Sandeep Sharma

Kubernetes Networking - 101

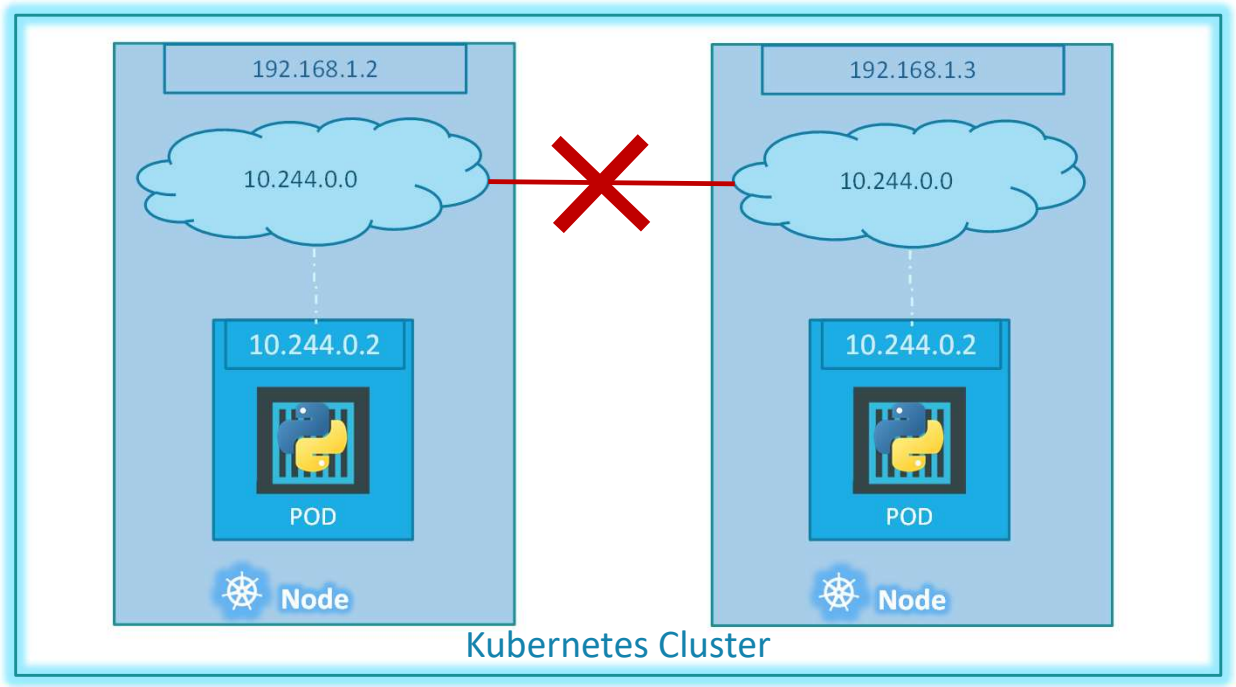
- IP Address is assigned to a POD

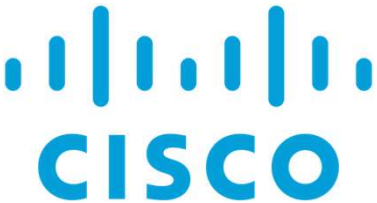


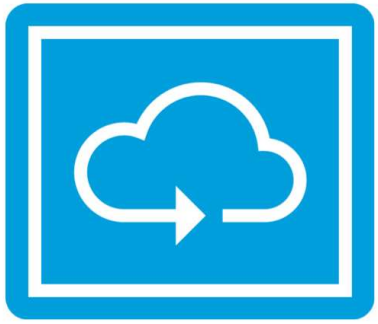


Cluster Networking

- All containers/PODs can communicate to one another without NAT
- All nodes can communicate with all containers and vice-versa without NAT



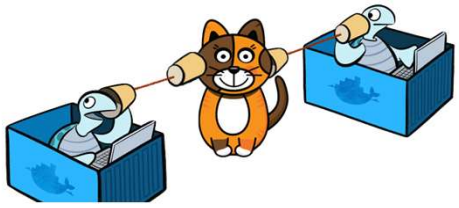

CISCO




cilium


flannel


vmware
NSX



Cluster Networking Setup

(3/4) Installing a pod network

You **MUST** install a pod network add-on so that your pods can communicate with each other.

The network must be deployed before any applications. Also, kube-dns, an internal helper service, will not start up before a network is installed. kubeadm only supports Container Network Interface (CNI) based networks (and does not support kubenet).

Several projects provide Kubernetes pod networks using CNI, some of which also support Network Policy. See the add-ons page for a complete list of available network add-ons. IPv6 support was added in CNI v0.6.0. CNI bridge and local-ipam are the only supported IPv6 network plugins in 1.9.

Note: kubeadm sets up a more secure cluster by default and enforces use of RBAC. Please make sure that the network manifest of choice supports RBAC.

You can install a pod network add-on with the following command:

```
kubectl apply -f <add-on.yaml>
```

NOTE: You can install **only one** pod network per cluster.

Choose one... | Calico | Canal | Flannel | Kube-router | Romana | Weave Net

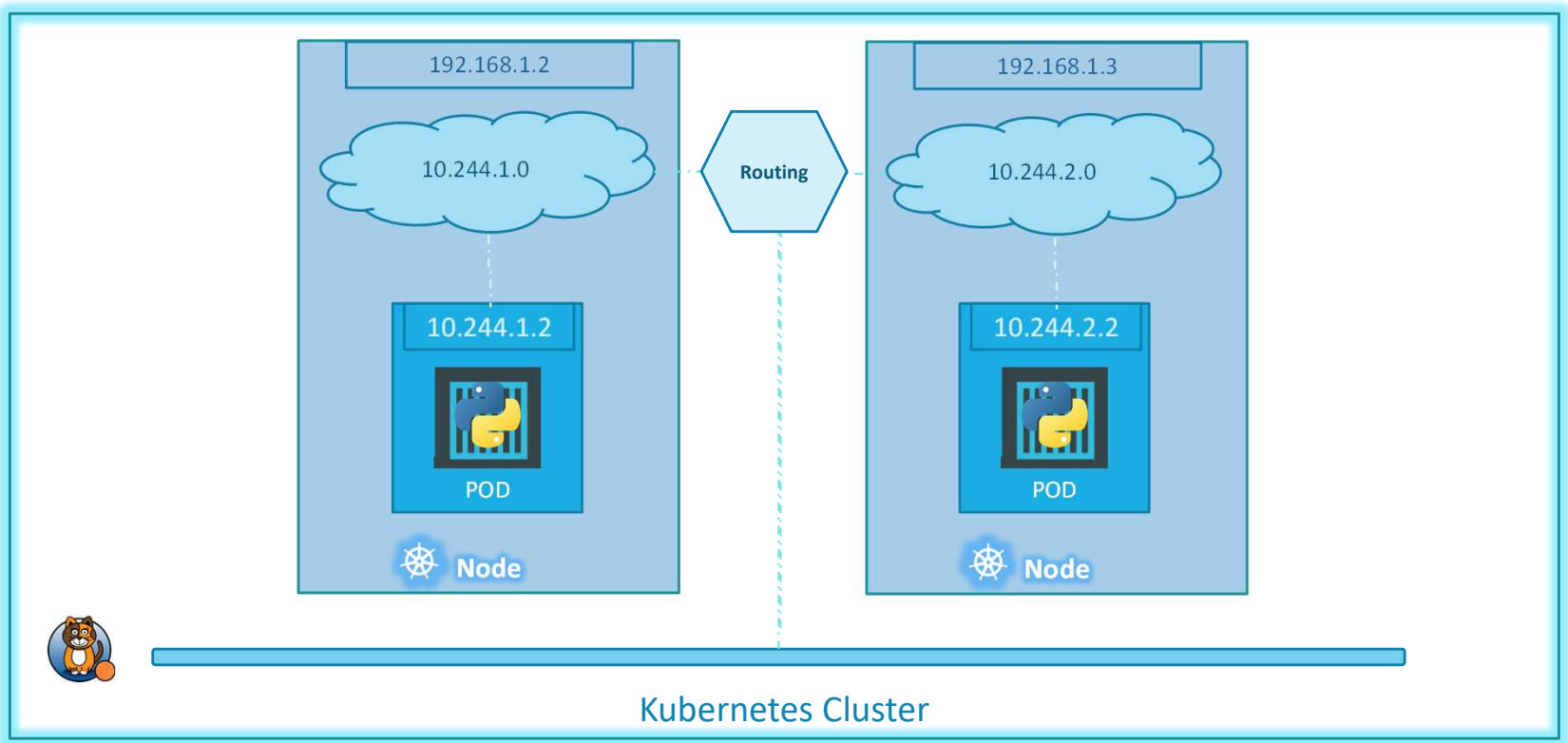
Refer to the Calico documentation for a kubeadm quickstart, a kubeadm installation guide, and other resources.

Note:

- In order for Network Policy to work correctly, you need to pass `--pod-network-cidr=192.168.0.0/16` to `kubeadm init`.
- Calico works on `amd64` only.

```
kubectl apply -f https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/kubeadm/1.7/calico.yaml
```

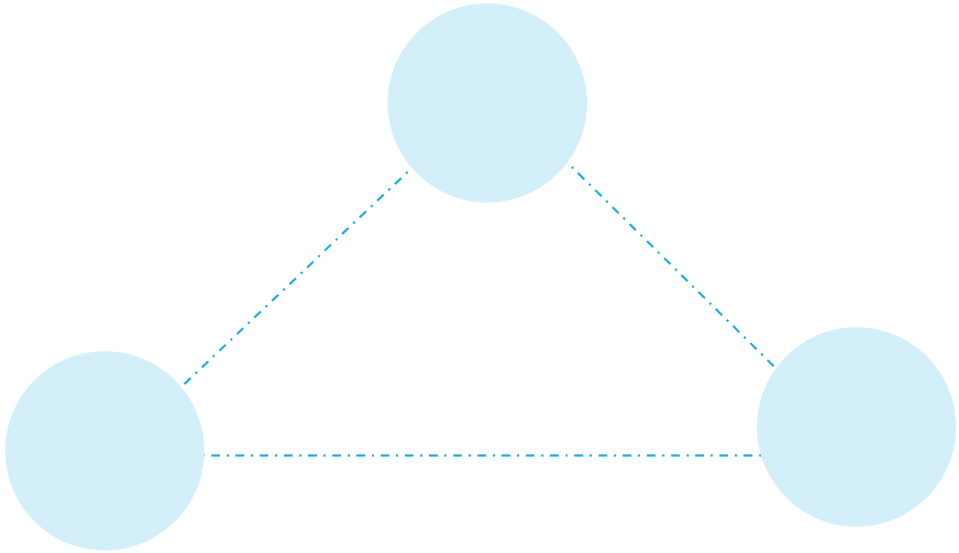
Cluster Networking



Demo

Networking

Services



Service

Kubernetes Services are resources that map network traffic to the Pods in your cluster. You need to create a Service each time you expose a set of Pods over the network, whether within your cluster or externally.

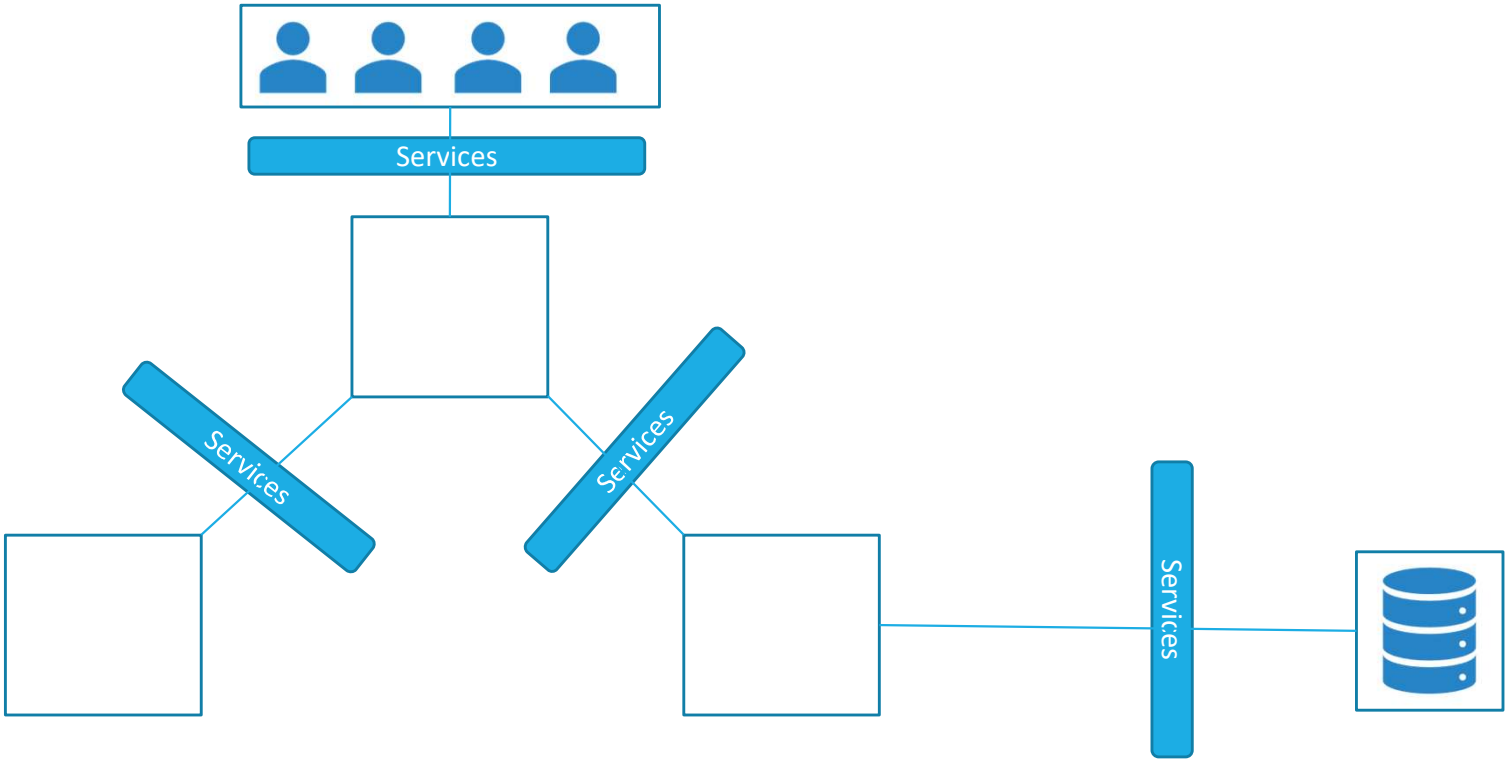
Kubernetes Services are API objects that enable network exposure for one or more cluster Pods. Services are integral to the Kubernetes networking model and provide important abstractions of lower-level components, which could behave differently between different clouds.

Service

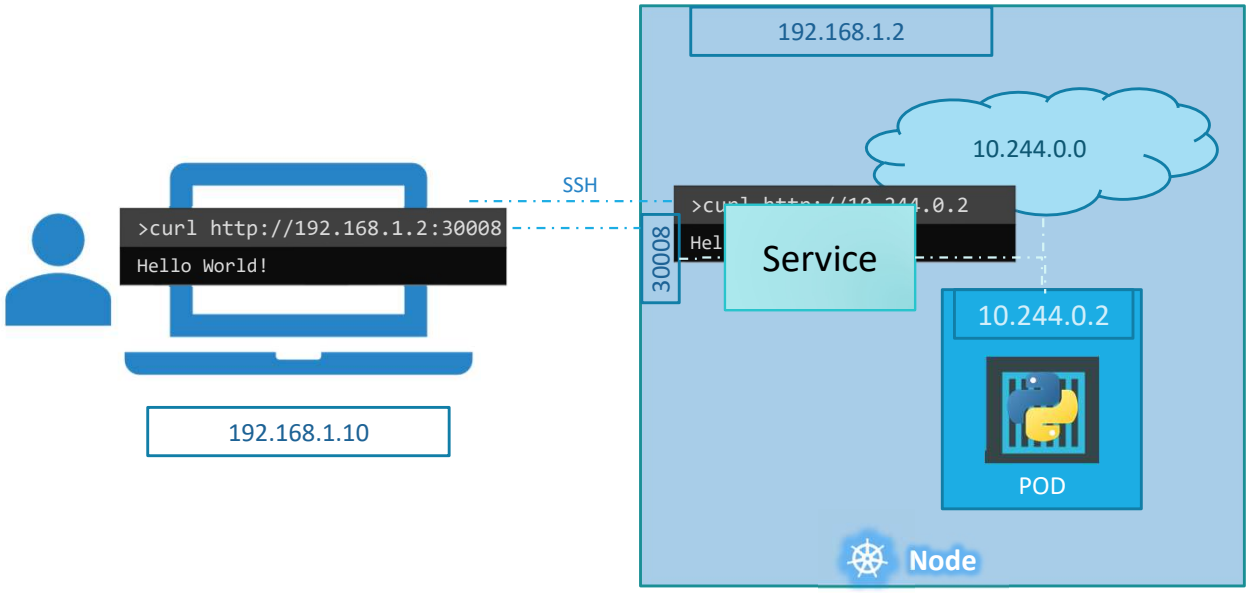
Services are necessary because of the distributed architecture of Kubernetes clusters. Apps are routinely deployed as Pods that could have thousands of replicas, spanning hundreds of physical compute Nodes. When a user interacts with your app, their request needs to be routed to any one of the available replicas, regardless of where it's placed.

Services sit in front of your Pods to achieve this behavior. All network traffic flows into the Service before being redirected to one of the available Pods. Your other apps can then communicate with the service's IP address or DNS name to reliably access the Pods you've exposed.

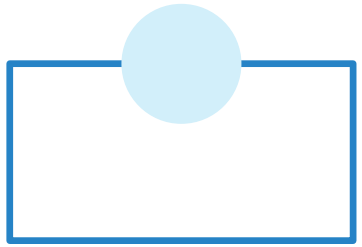
Services



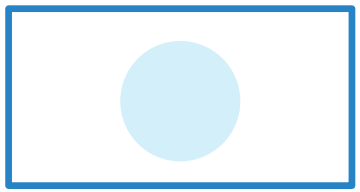
Service


>curl http://192.168.1.2:30008
Hello World!
192.168.1.10
SSH
30008
192.168.1.2
10.244.0.0
Service
10.244.0.2
POD
Node

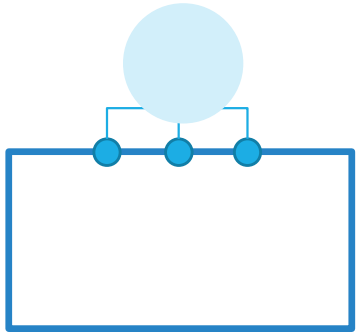
Services Types



NodePort



ClusterIP



LoadBalancer

Service Type

1. ClusterIP Services

ClusterIP Services assign an IP address that can be used to reach the Service from within your cluster. This type doesn't expose the Service externally.

ClusterIP is the default service type used when you don't specify an alternative option. It's the most common kind of service you'll use as it enables simple internal networking for your workloads.

Service Type

2. NodePort Services

NodePort Services are exposed externally through a specified static port binding on each of your Nodes. Hence, you can access the Service by connecting to the port on any of your cluster's Nodes. NodePort Services are also assigned a cluster IP address that can be used to reach them within the cluster, just like ClusterIP Services.

Service Type

3. LoadBalancer Services

LoadBalancer Services are exposed outside your cluster using an external load balancer resource. This requires a connection to a load balancer provider, typically achieved by integrating your cluster with your cloud environment. Creating a LoadBalancer service will then automatically provision a new load balancer infrastructure component in your cloud account. This functionality is automatically configured when you use a managed Kubernetes service such as Amazon EKS or Google GKE.

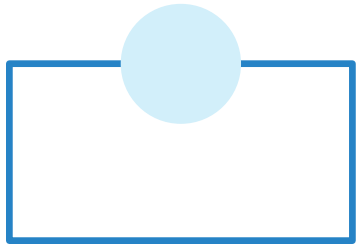
Service Type

4. ExternalName Services

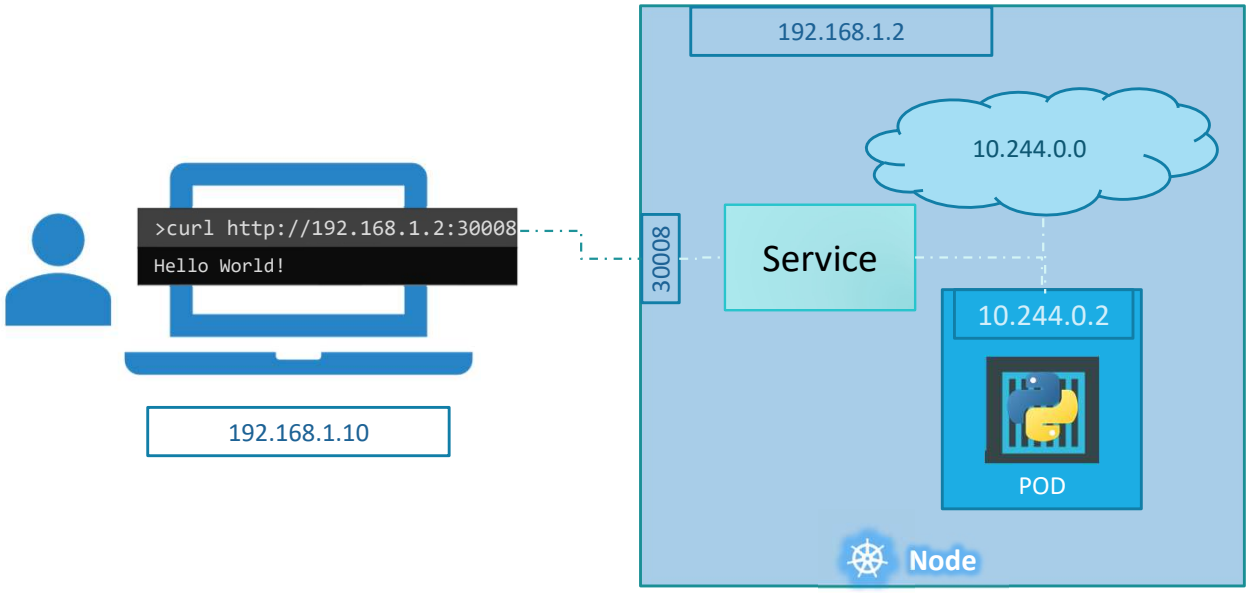
ExternalName Services allow you to conveniently access external resources from within your Kubernetes cluster. Unlike the other Service types, they don't proxy traffic to your Pods.

Characteristic	ClusterIP	NodePort	LoadBalancer	ExternalName	Headless
Accessibility	Internal	External	External	Internal	Internal
Use case	Expose Pods to other Pods in your cluster	Expose Pods on a specific Port of each Node	Expose Pods using a cloud load balancer resource	Configure a cluster DNS CNAME record that resolves to a specified address	Interface with external service discovery systems
Suitable for	Internal communications between workloads	Accessing workloads outside the cluster, for one-off or development use	Serving publicly accessible web apps and APIs in production	Decoupling your workloads from direct dependencies on external service URLs	Advanced custom networking that avoids automatic Kubernetes proxying
Client connection type	Stable cluster-internal IP address or DNS name	Port on Node IP address	IP address of external load balancer	Stable cluster-internal IP address or DNS name	Stable-cluster internal IP address or DNS name that also enables DNS resolution of the Pod IPs behind the Service
External dependencies	None	Free port on each Node	A Load Balancer component (typically billable by your cloud provider)	None	None

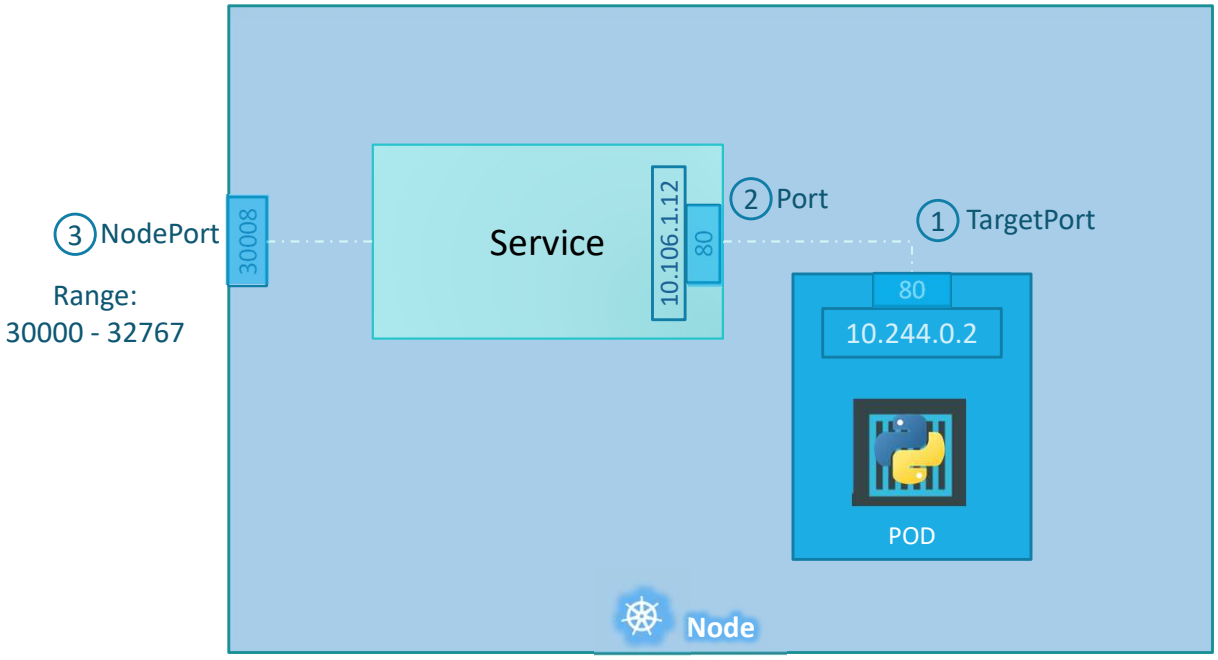
NodePort



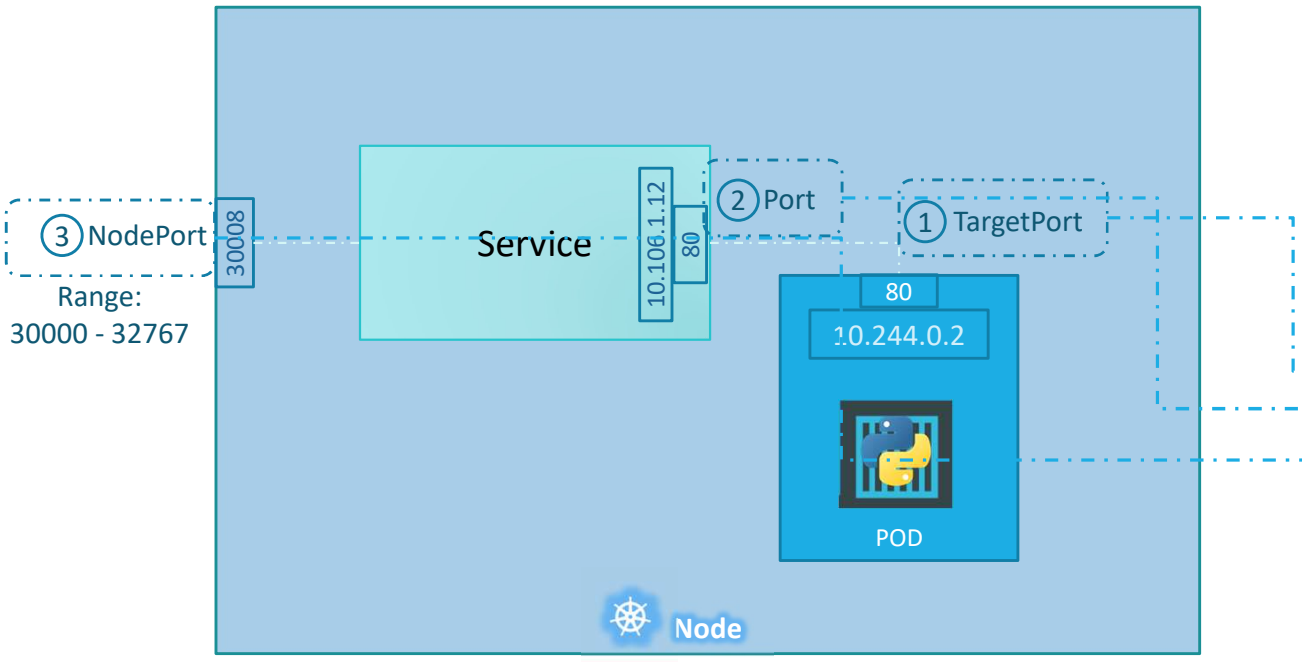
Service - NodePort



Service - NodePort



Service - NodePort



service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
      targetPort: 80
      port: 80
      nodePort: 30008
```

Service - NodePort

service-definition.yml

```
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
    selector:
```

pod-definition.yml

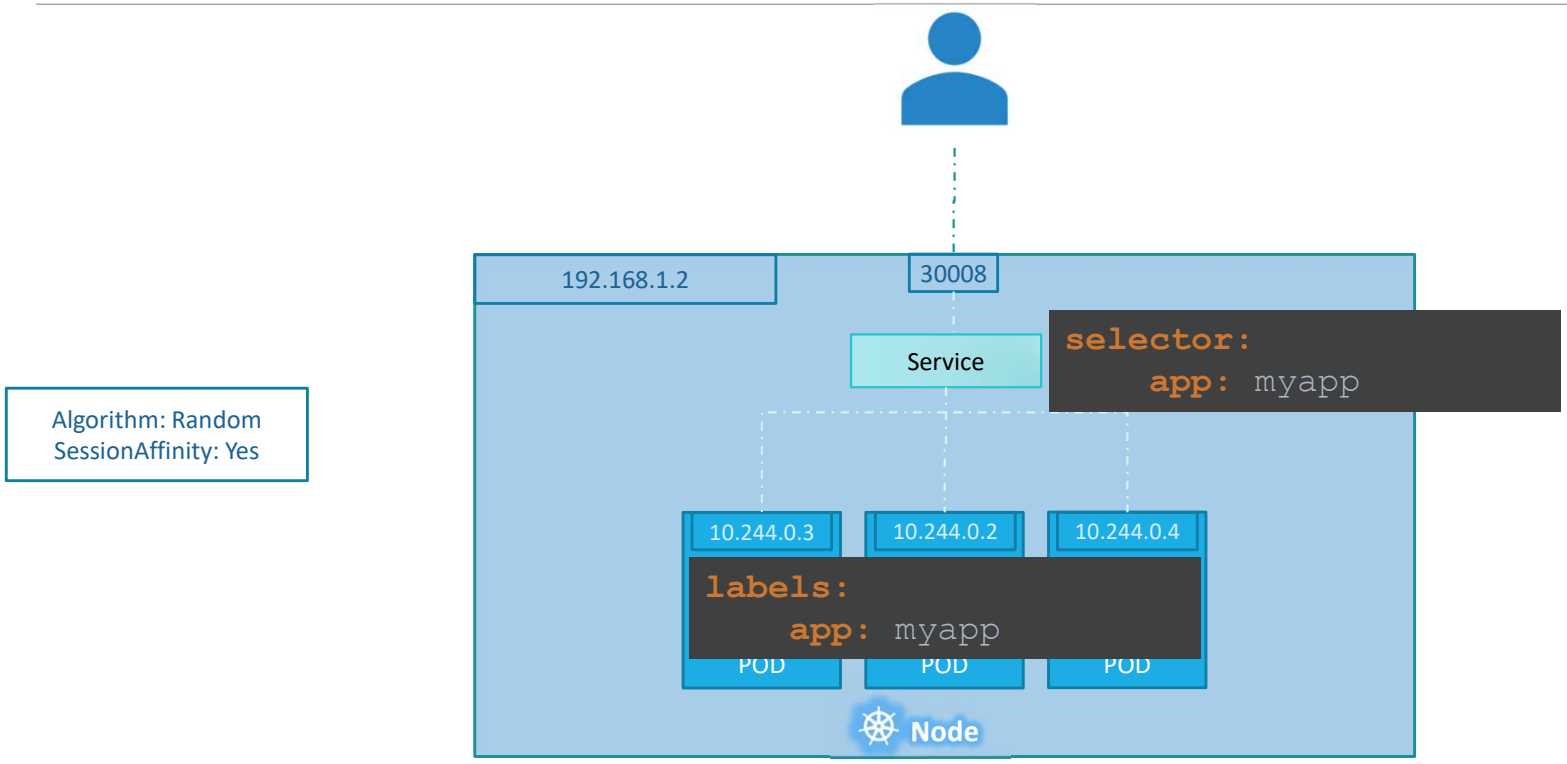
```
        app: myapp
```

```
> kubectl create -f service-definition.yml
service "myapp-service" created
```

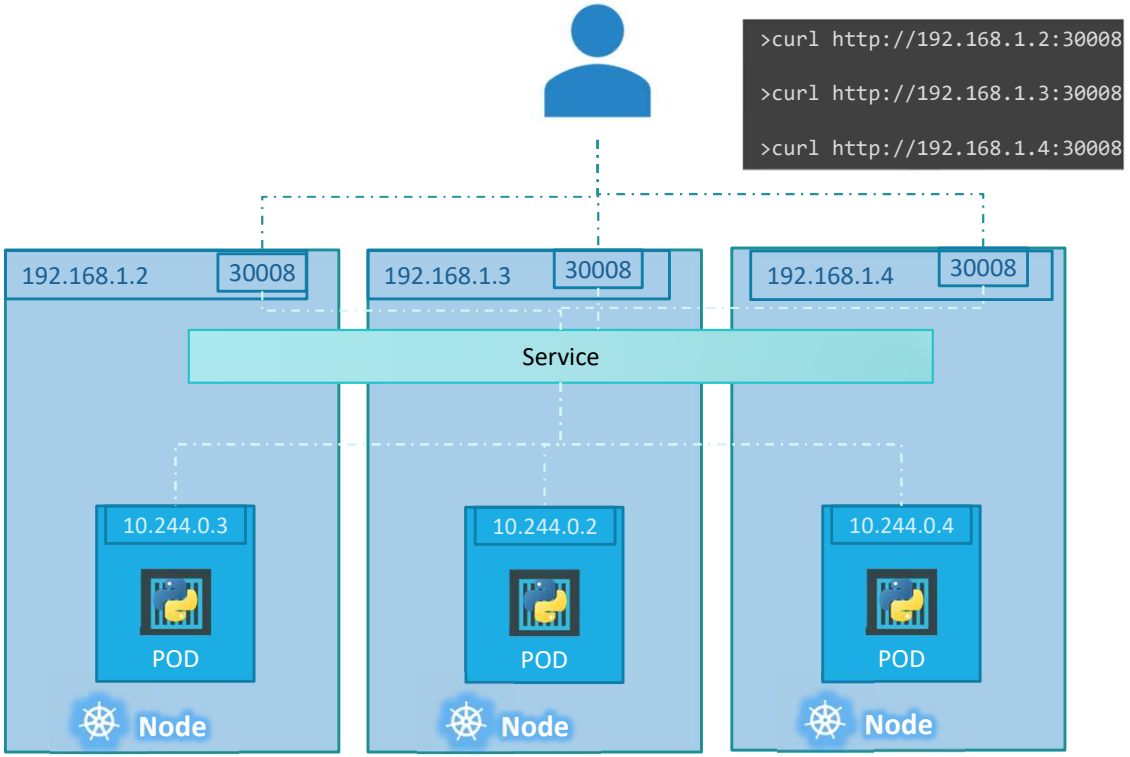
```
> kubectl get services
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP          16d
myapp-service   NodePort    10.106.127.123   <none>        80:30008/TCP     5m
```

```
> curl http://192.168.1.2:30008
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
```

Service - NodePort



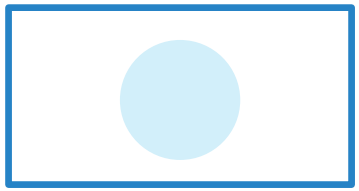
Service - NodePort



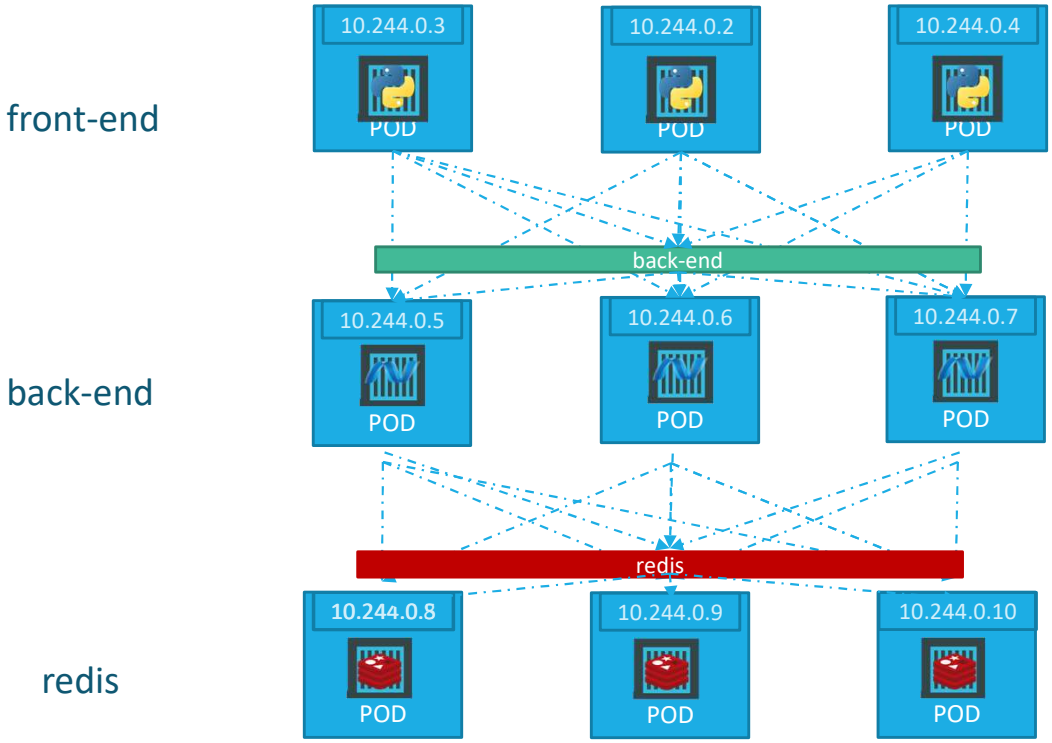
Demo

Service - NodePort

ClusterIP



ClusterIP


front-end
back-end
redis
10.244.0.3
POD
10.244.0.2
POD
10.244.0.4
POD
back-end
10.244.0.5
POD
10.244.0.6
POD
10.244.0.7
POD
redis
10.244.0.8
POD
10.244.0.9
POD
10.244.0.10
POD

service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: back-end

spec:
    type: ClusterIP
    ports:
     - targetPort: 80
       port: 80

    selector:
```

pod-definition.yml

```
> kubectl create -f service-definition.yml
service “back-end" created

> kubectl get services
NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes    ClusterIP   10.96.0.1        <none>        443/TCP    16d
back-end      ClusterIP   10.106.127.123   <none>        80/TCP     2m
```

```yaml
      app: myapp
      type: back-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

Demo

Service - NodePort

References

https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

Rollout and Versioning

Revision 1


nginx:1.7.0


nginx:1.7.0


nginx:1.7.0


nginx:1.7.0


nginx:1.7.0


nginx:1.7.0


nginx:1.7.0


nginx:1.7.0


nginx:1.7.0

Revision 2


nginx:1.7.1


nginx:1.7.1


nginx:1.7.1


nginx:1.7.1


nginx:1.7.1


nginx:1.7.1


nginx:1.7.1


nginx:1.7.1

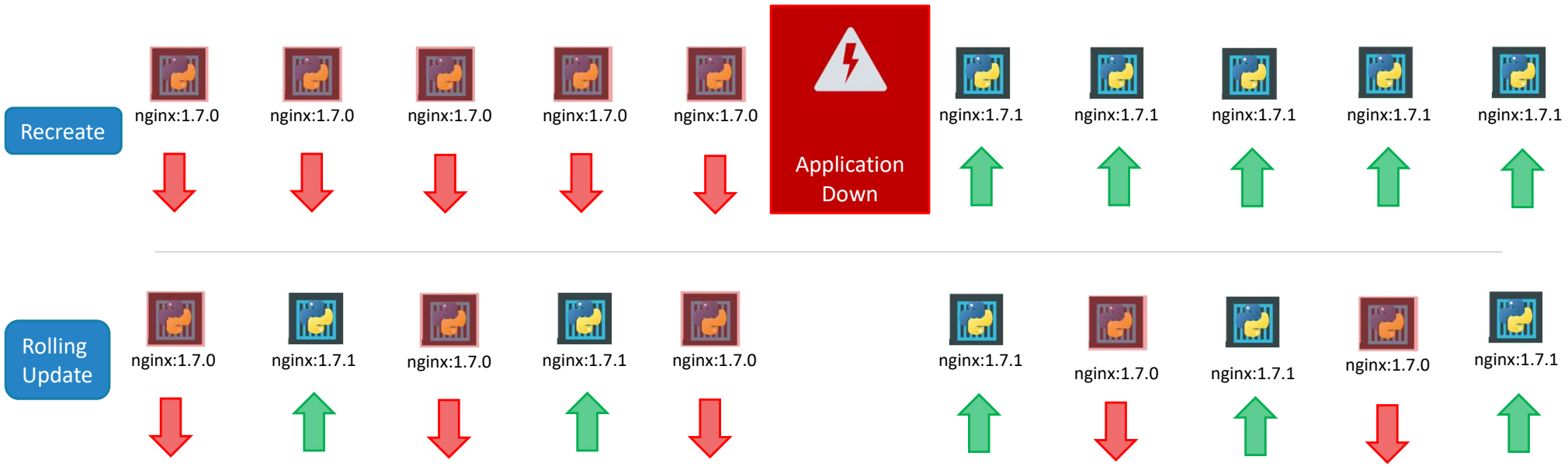

nginx:1.7.1

Rollout Command

```
> kubectl rollout status deployment/myapp-deployment
Waiting for rollout to finish: 0 of 10 updated replicas are available...
Waiting for rollout to finish: 1 of 10 updated replicas are available...
Waiting for rollout to finish: 2 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 4 of 10 updated replicas are available...
Waiting for rollout to finish: 5 of 10 updated replicas are available...
Waiting for rollout to finish: 6 of 10 updated replicas are available...
Waiting for rollout to finish: 7 of 10 updated replicas are available...
Waiting for rollout to finish: 8 of 10 updated replicas are available...
Waiting for rollout to finish: 9 of 10 updated replicas are available...
deployment "myapp-deployment" successfully rolled out
```

```
> kubectl rollout history deployment/myapp-deployment
deployments "myapp-deployment"
REVISION  CHANGE-CAUSE
1         <none>
2         kubectl apply --filename=deployment-definition.yml --record=true
```

Deployment Strategy



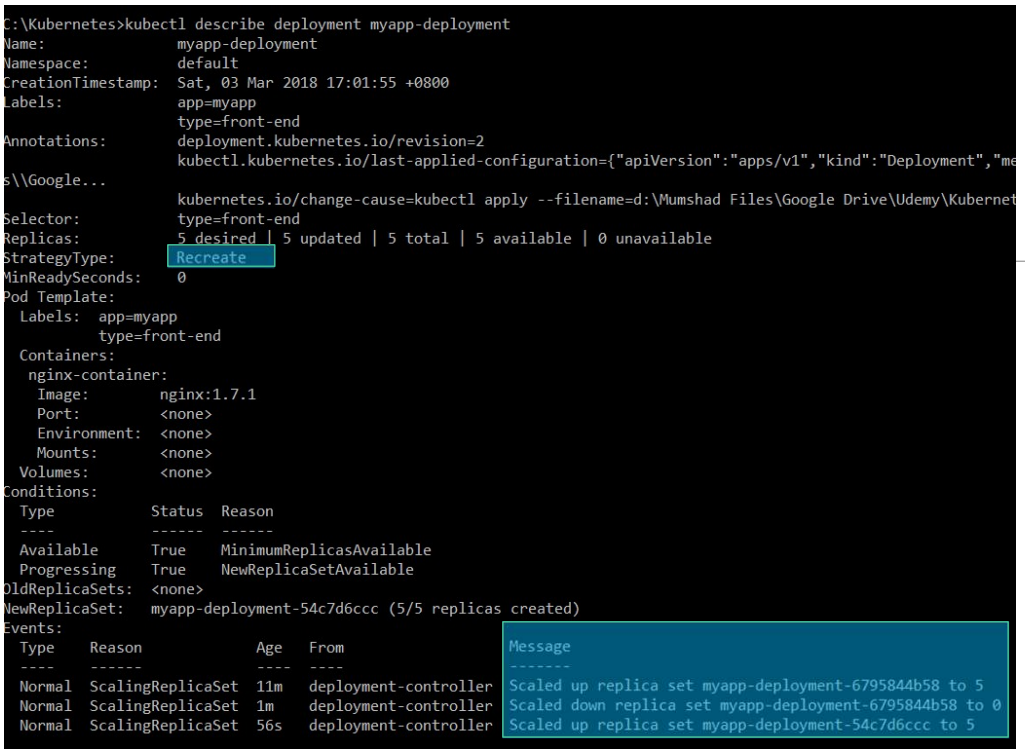
Kubectl apply

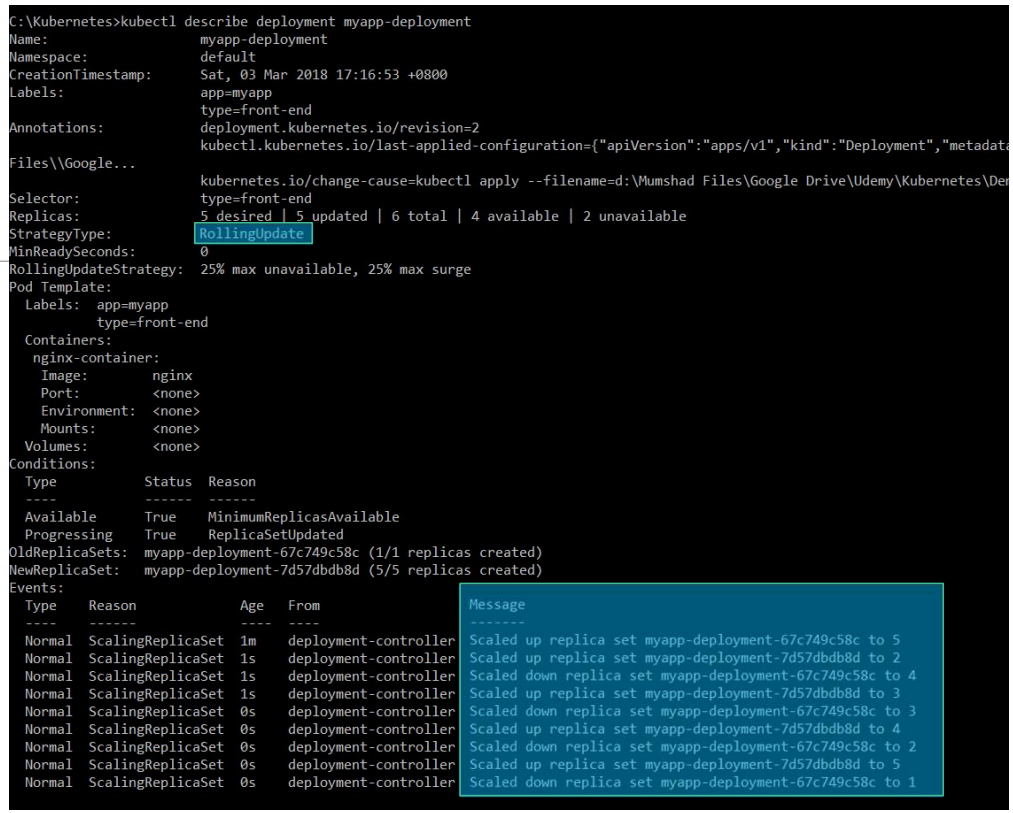
```
> kubectl apply -f deployment-definition.yml
deployment "myapp-deployment" configured
```

```
> kubectl set image deployment/myapp-deployment \
                  nginx=nginx:1.9.1
deployment "myapp-deployment" image is updated
```

deployment-definition.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
         app: myapp
         type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.1
  replicas: 3
  selector:
    matchLabels:
       type: front-end
```

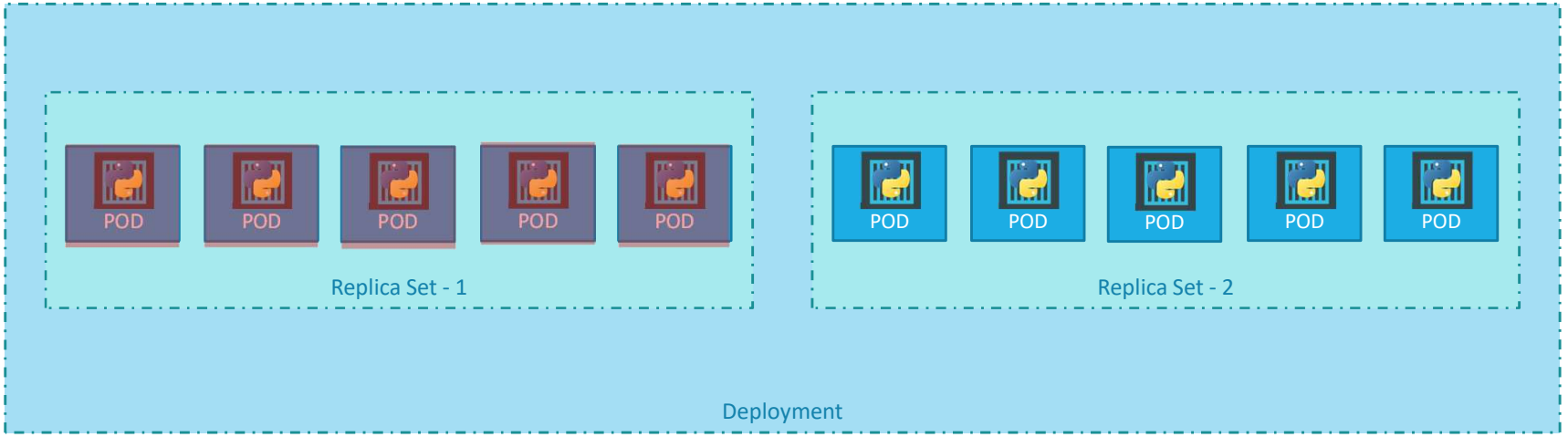




Recreate

RollingUpdate

Upgrades

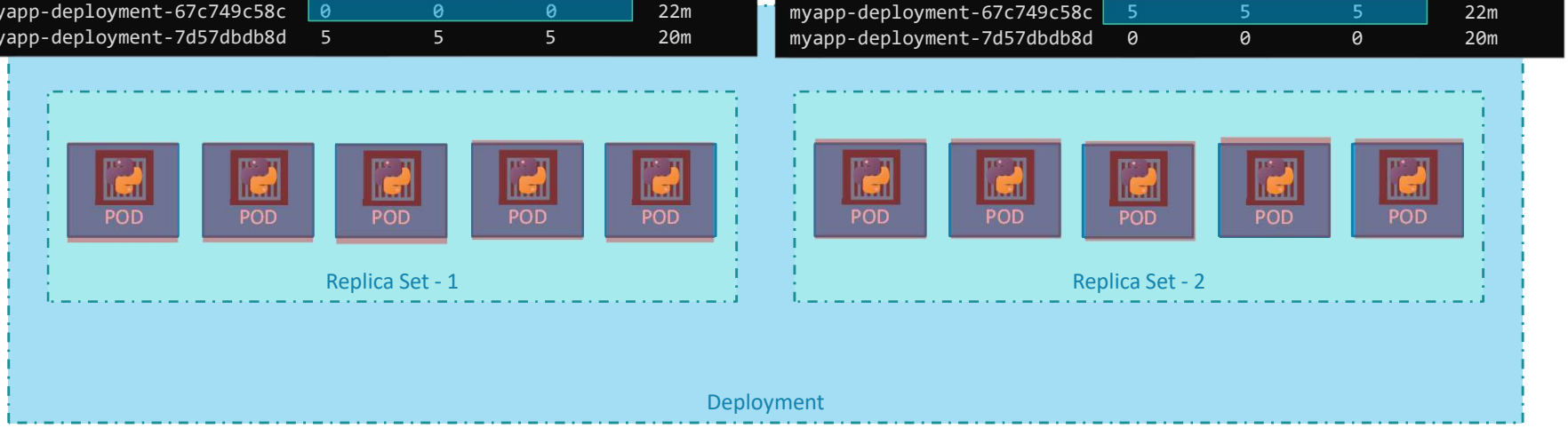



```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY     AGE
myapp-deployment-67c749c58c   0         0         0         22m
myapp-deployment-7d57dbdb8d   5         5         5         20m
```

Rollback

```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c   0         0         0       22m
myapp-deployment-7d57dbdb8d   5         5         5       20m
```

```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c   5         5         5       22m
myapp-deployment-7d57dbdb8d   0         0         0       20m
```




```
> kubectl rollout undo deployment/myapp-deployment
deployment “myapp-deployment” rolled back
```

kubectl run

```
> kubectl run nginx --image=nginx
deployment "nginx" created
```

Summarize Commands

Create

```
> kubectl create –f deployment-definition.yml --record=true
```

Get

```
> kubectl get deployments
```

Update

```
> kubectl apply –f deployment-definition.yml
```

```
> kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1
```

Status

```
> kubectl rollout status deployment/myapp-deployment
```

```
> kubectl rollout history deployment/myapp-deployment
```

Rollback

```
> kubectl rollout undo deployment/myapp-deployment
```

HEALTHCHECKS

HEALTH CHECKS

- If your application malfunctions, the pod and container can still be running, but the application might not work anymore.
- To detect and resolve problems with your application, you can run healthchecks:
- Two different types of HC:
- - Running a command in the container periodically
- - Periodic checks on a URL (HTTP)
- the typical production application behind a load-balancer should always have heathchecks implemented in some way to ensure availability and resiliency of the app

HEALTH CHECKS

kubectl get nodes

kubectl get nodes --show-labels

kubectl create -f <yamlfile>

kubectl describe pod (you can edit the values)

kubectl edit pod <POD-NAME>

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-podmonitor
  labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: httpd-container
    image: httpd
    ports:
    - name: httpd-port
      containerPort: 8080

    livenessProbe:
      httpGet:
        path: /
        port: 8080
      initialDelaySeconds: 15
      timeoutSeconds: 30
  nodeSelector:
    hardware: high-spec
```

Static Pod

Individual pods on Workers

Static Pods

Static Pods are managed directly by the kubelet daemon on a specific node, without the API server observing them.

Unlike Pods that are managed by the control plane (for example, a Deployment); instead, the kubelet watches each static Pod (and restarts it if it fails).

Static Pods are always bound to one Kubelet on a specific node.

The kubelet automatically tries to create a mirror Pod on the Kubernetes API server for each static Pod. This means that the Pods running on a node are visible on the API server, but cannot be controlled from there.

The Pod names will be suffixed with the node hostname with a leading hyphen.

Static Pods

K8S version >1.21.x:

Put pod.yaml file under below path on node to get it started automatically:
/etc/kubernetes/manifests/

For previous versions:
1) Create a directory i.e. /etc/kubelet.d/
2) Put yaml files into it
3) Run below command:
KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-domain=kube.local --pod-manifest-path=/etc/kubelet.d/“
4) Restart kubelet via
systemctl restart kubelet

Init Container

Start before others

Init Containers

What if you need one container in pod to run successfully, before other containers get started in it to setup the base (like Filesystem, Binaries availability/Dependency etc)?

A Pod can have multiple containers running apps within it, but it can also have one or more init containers, which are run before the app containers are started.

Init containers are exactly like regular containers, except:

Init containers always run to completion.
Each init container must complete successfully before the next one starts.

If a Pod's init container fails, the kubelet repeatedly restarts that init container until it succeeds.

However, if the Pod has a restartPolicy of Never, and an init container fails during startup of that Pod, Kubernetes treats the overall Pod as failed.

Init Containers

https://kubernetes.io/docs/concepts/workloads/pods/init-containers/

DaemonSet

Run on all Machines

Sandeep Kumar Sharma

DaemonSets

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a cluster storage daemon on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

SECRETS

Sandeep Kumar Sharma

SECRETS

- Secrets provides a way in K8's to distribute credentials, keys, passwords or secret data to the pods.
- K8 itself used this mechanism to provide the credentials to access the internal API's
- Secrets is one way to provide secrets, native to K8, other way is by using external vault services
- Secrets can be used as: **environment variables**
- or **as a file in a pod** (this need volume to be mounted and volumes have the files having secrets)
- First generate a Secrets using files:
- echo -n "root" > ./username.txt
- echo -n "password" > ./password.txt
- kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./password.txt
- **NOTE: You can also use SSH keys**

SECRETS

```
echo -n "root" | base64
echo -n "password" | base64
```

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secrets
type: Opaque
data:
  username: cm9vdA==
  password: cGFzc3dvcmQ=
```

```
kubectl create -f <yaml>
kubectl get secrets
kubectl describe secrets
<Secret-Name>
```

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: httpd-container
    image: httpd
    volumeMounts:
    - name: cred-volume
      mountPath: /etc/creds
      readOnly: true
  volumes:
  - name: cred-volume
    secret:
      secretName: <Secret-
Name>
```

```
kubectl create -f <YAML>
kubectl describe pod
<POD-Name>
```

Login into the container and see the secrets will be in:

/etc/creds/username
/etc/creds/password

Execute mount and you will see default secrets shared by Kubernetes: tmpfs on /run/secrets/kubernetes.io/serviceaccount type tmpfs (ro,relatime,seclabel)

SECRETS

Secrets with Environment Variables

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: httpd-container
    image: httpd
  env:
  - name: SECRET_USERNAME
    valueFrom:
      secretKeyRef:
        name: db-secret
        key: username
  -name: SECRET_PASSWD
.....
```

ConfigMaps

ConfigMaps

- Configuration parameters that are not secret, can be put in a configmap

- The input is again key-value pair and can be read from any of the below methods:

- - environments variables
- - volumes
- - Container command line parameters

- A configmap can be a complete configuration file, This file can be then mounted using volumes where applications are expecting the file

- This way we can change the container without changing the container directly (without changing the container image)

- **CREATE A CONFIG FILE**
- **kubectl create configmap <CONFIG_MAP_NAME> --from-file=<file-name>**
- **Then create a pod with volumes**

ConfigMaps

```
/usr/local/apache2/htdocs/index.html

<html><body><h1>It works again with
Configmaps!</h1></body></html>

kubectl create configmap nginx-config
--from-file=index.html

kubectl get configmap

kubectl get configmap -o yaml

(here you will see a key (in our case
index.html) and values)
```

```
apiVersion: v1
kind: Pod
metadata:
  name: helloworld-nginx
  labels:
    app: helloworld-nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: config-volume
      mountPath:
/usr/share/nginx/html
  volumes:
    - name: config-volume
      configMap:
        name: nginx-config
        items:
        - key: index.html
          path: index.html
```

```
apiVersion: v1
kind: Service
metadata:
  name: helloworld-
nginx-service
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: helloworld-
nginx
  type: NodePort
```

PV and PVC

Persistent Storage in Kubernetes

Sandeep Kumar Sharma

Persistent Storage

A Persistent Volume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes but have a lifecycle independent of any individual Pod that uses the PV.

A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany

Access Types and Methods

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWSElasticBlockStore	✓	-	-
AzureFile	✓	✓	✓
AzureDisk	✓	-	-
CephFS	✓	✓	✓
Cinder	✓	-	-
CSI	depends on the driver	depends on the driver	depends on the driver
FC	✓	✓	-
FlexVolume	✓	✓	depends on the driver
Flocker	✓	-	-
GCEPersistentDisk	✓	✓	-

Persistent Volume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Persistent Volume Claim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

DashBoards

Sandeep Kumar Sharma

DashBoards

- K8's comes with a WebUI you can use instead of the kubectl commands.
- You will get an overview of running applications on your cluster.
- Creating and Modifying individual kubernetes resources and workloads can also be done like KUBECTL create and delete command.
- In general you can access the kubernetes webUI at https://<MASTER>/ui
- Note: Now the process has been secured and changed since 1.8+ version

DashBoards

- https://github.com/kubernetes/dashboard

- cd dashboard-master/ src/deploy/recommended/ kubernetes-dashboard.yaml

- Expose the NodePort for AWS/Public IP's, and your service should look like:

- spec:
- **type: NodePort**
- ports:
- - port: 443
- targetPort: 8443
- selector:
- k8s-app: kubernetes-dashboard

DashBoards

- kubectl create -f kubernetes-dashboard.yaml
- kubectl get pods -n namespace
- kubectl get pods -n kube-system
- kubectl get service -n kube-system

- Excess the dashboard using WEB with mentioned port

- Now use the below link to create a user and view the dashboard:

- https://github.com/kubernetes/dashboard/wiki/Creating-sample-user

DashBoards

- https://github.com/kubernetes/dashboard/wiki/Creating-sample-user

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kube-system
```

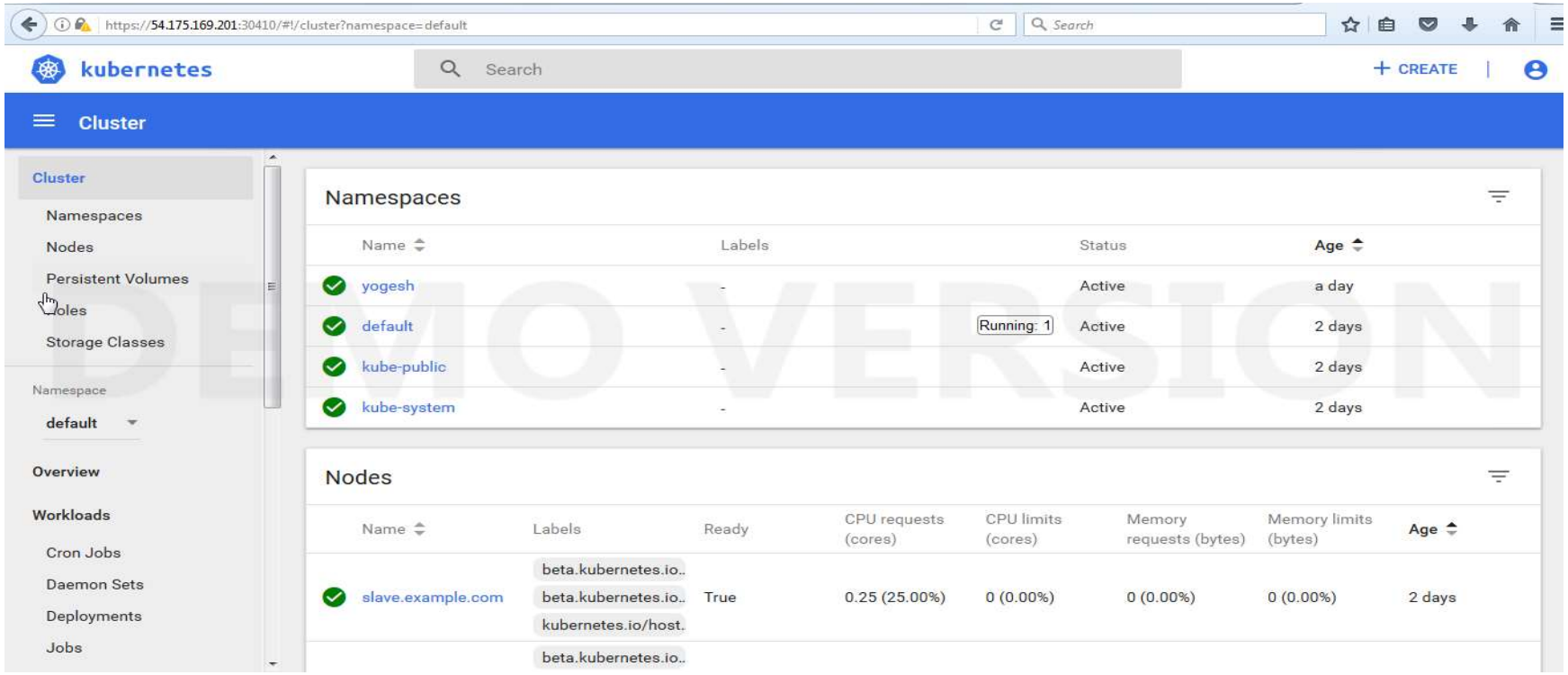
```
apiVersion:
rbac.authorization.k8s.io/v1b
eta1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup:
rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kube-system
```

```
$ kubectl create -f
<yaml.file>

$ kubectl -n kube-system
describe secret $(kubectl -n
kube-system get secret | grep
admin-user | awk '{print
$1}')

$ kubectl get services -n
kube-system
```

Service Accounts can be managed with CLI

```
kubectl get serviceaccounts --all-namespaces

kubectl get roles --all-namespaces

kubectl get rolebinding

kubectl create serviceaccount gagan

kubectl create role pod-reader --verb=get --verb=list --verb=watch --resource=pods

kubectl create rolebinding gagan --clusterrole=pod-reader --user=gagan
```

DashBoards


https://54.175.169.201:30410/#!/cluster?namespace=default
Search
kubernetes
Search
+ CREATE
Cluster
Cluster
Namespaces
Nodes
Persistent Volumes
Roles
Storage Classes
Namespace
default
Overview
Workloads
Cron Jobs
Daemon Sets
Deployments
Jobs
Namespaces
Name
Labels
Status
Age
yogesh
-
Active
a day
default
-
Running: 1
Active
2 days
kube-public
-
Active
2 days
kube-system
-
Active
2 days
Nodes
Name
Labels
Ready
CPU requests (cores)
CPU limits (cores)
Memory requests (bytes)
Memory limits (bytes)
Age
slave.example.com
beta.kubernetes.io..
beta.kubernetes.io..
kubernetes.io/host..
beta.kubernetes.io..
True
0.25 (25.00%)
0 (0.00%)
0 (0.00%)
0 (0.00%)
2 days
DEMO VERSION

Addition/Deletion of a node to the Cluster

Node Taint & Toleration

Node affinity is a property of Pods that *attracts* them to a set of nodes (either as a preference or a hard requirement).

Taints are the opposite -- they allow a node to repel a set of pods.

Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/

Node cordon and draining

```
kubectl cordon my-node          # Mark my-node as unschedulable
kubectl drain my-node           # Drain my-node in preparation for maintenance
kubectl uncordon my-node        # Mark my-node as schedulable
```

Remove node

To remove a Kubernetes worker node from the cluster, perform the following operations.

Migrate pods from the node

```
$ kubectl drain  <node-name> --delete-local-data --ignore-daemonsets
```

Prevent a node from scheduling new pods use – Mark node as unschedulable

```
$ kubectl cordon <node-name>
```

Revert changes made to the node by ‘kubeadm join‘ – Run on worker node to be removed

```
$ sudo kubeadm reset # run it from worker node
```

Remove the node from cluster

```
$ kubectl delete node <node-name>
```

Node Addition

- Create a New instance and perform all of the pre-requisties.
- Check the token is valid or not by running "kubeadm token list"
- Now generate the token using "kubeadm token create --print-join-command"
- Run "kubeadm reset" on node if you ran the old join command
- Finally run the newly generated token command
- Go to master and run "kubectl get nodes"

NodeSelectors

Node Selector

- You can also use labels to tag your nodes
- Once nodes are tagged, you can use label selectors to let pods only run specific nodes
- there are two steps required to run a pod on a specific set of nodes

- first you tag the node

- then you add a nodeSelector to your pod configuration

Node Selector

kubectl get nodes
kubectl get nodes --show-labels
kubectl create -f <yamlfile>

Run the shown yaml file and check the pod status you will find the error stating "node(s) didn't match node selector."

kubectl get pods
kubectl describe pod <pod-name>
kubectl label nodes <nodename> hardware=high-spec
kubectl label nodes <nodename> hardware=low-spec

As soon as you label the node you will get the POD in running condition (auto-healing)

CASE2: Change the nodeSelector to other node and re-run

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: httpd-container
    image: httpd
  nodeSelector:
    hardware: high-spec
```

Setting Role Labels

Adding a Role to a Worker Node:

```
[root@master ~]# kubectl label node worker1 node-role.kubernetes.io/linux-worker=worker1
node/worker1 labeled
[root@master ~]# kubectl get nodes
NAME     STATUS   ROLES        AGE   VERSION
master   Ready    master       12d   v1.19.2
worker1  Ready    linux-worker  12d   v1.19.2
```

Removing a Role Label

```
[root@master ~]# kubectl label node worker1 node-role.kubernetes.io/linux-worker-
node/worker1 labeled
[root@master ~]# kubectl get nodes
NAME     STATUS   ROLES    AGE   VERSION
master   Ready    master   12d   v1.19.2
worker1  Ready    <none>   12d   v1.19.2
[root@master ~]#
```

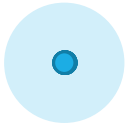
Kubernetes as a Service (PaaS Platform)

Sandeep Kumar Sharma

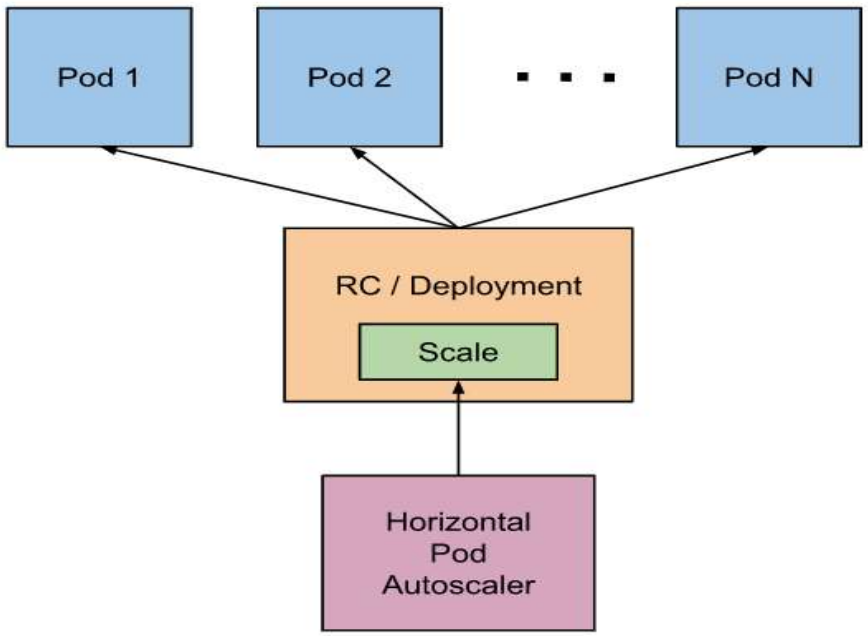
Kubernetes as a Service

- Many Cloud Service providers provides Kubernetes as a Service.
- They manages Kubernetes master service (Somewhere charged, somewhere no charges)
- Worker nodes are charged and assigned to us.
- We can deploy pods via GUI as well as command line.
- Examples – EKS, GKE, AKS

Autoscaling



Autoscaling



Autoscaling

kubectl autoscale deployment myapp-deployment --cpu-percent=50 --min=1 --max=10

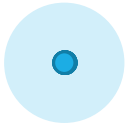
kubectl get hpa

#but you need to set a metric deployment first otherwise it'wont be able to collect the metric

kubectl delete hpa myapp-deployment

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

Helm Charts



Sandeep Kumar Sharma

Helm

Kubernetes Package Manager

In June 2018, Helm was adopted as an official CNCF project.

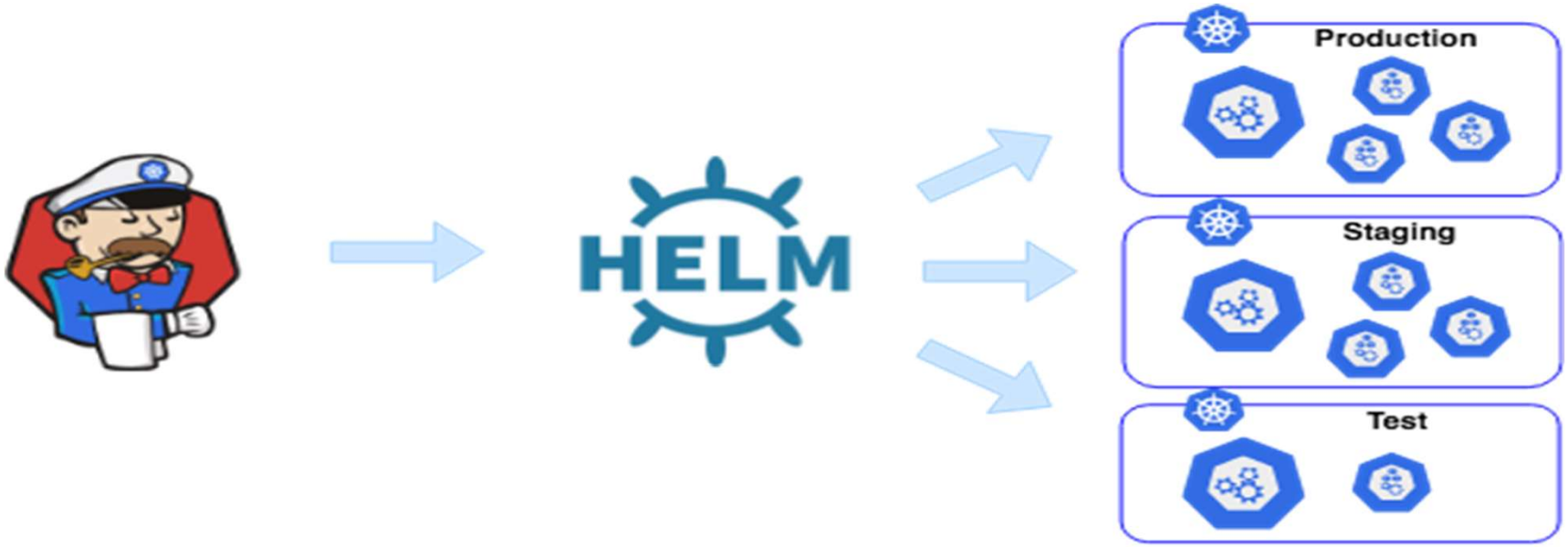
A Helm chart is simply a collection of YAML template files organized into a specific directory structure.

It play a large role in optimizing an organization's CI/CD integration with Kubernetes.

Manage multiple environments together

With you you'll have ability to provide application configuration during deployment. Not only you can specify the Kubernetes resources (deployments, services, etc.) that make up your application, but also environment-specific configuration for those resources.

Helm Charts



Working with Helm

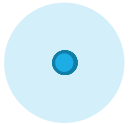
Do the Installation by running below or reach out to https://helm.sh/docs/intro/install:

https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash

Create chart with:
helm create [chartname]

Modify the values.yaml and put entries as per your requirement
helm package [chartname] // to create package
helm install [chartname] // to Install a chart
Helm ls // to list charts
Helm upgrade [generated-dep-name] chart-name // to upgrade
Helm rollback [generated-dep-name] version // to rollback
Helm delete [generated-dep-name] // to delete the deployed packages

Service Mesh



Sandeep Kumar Sharma

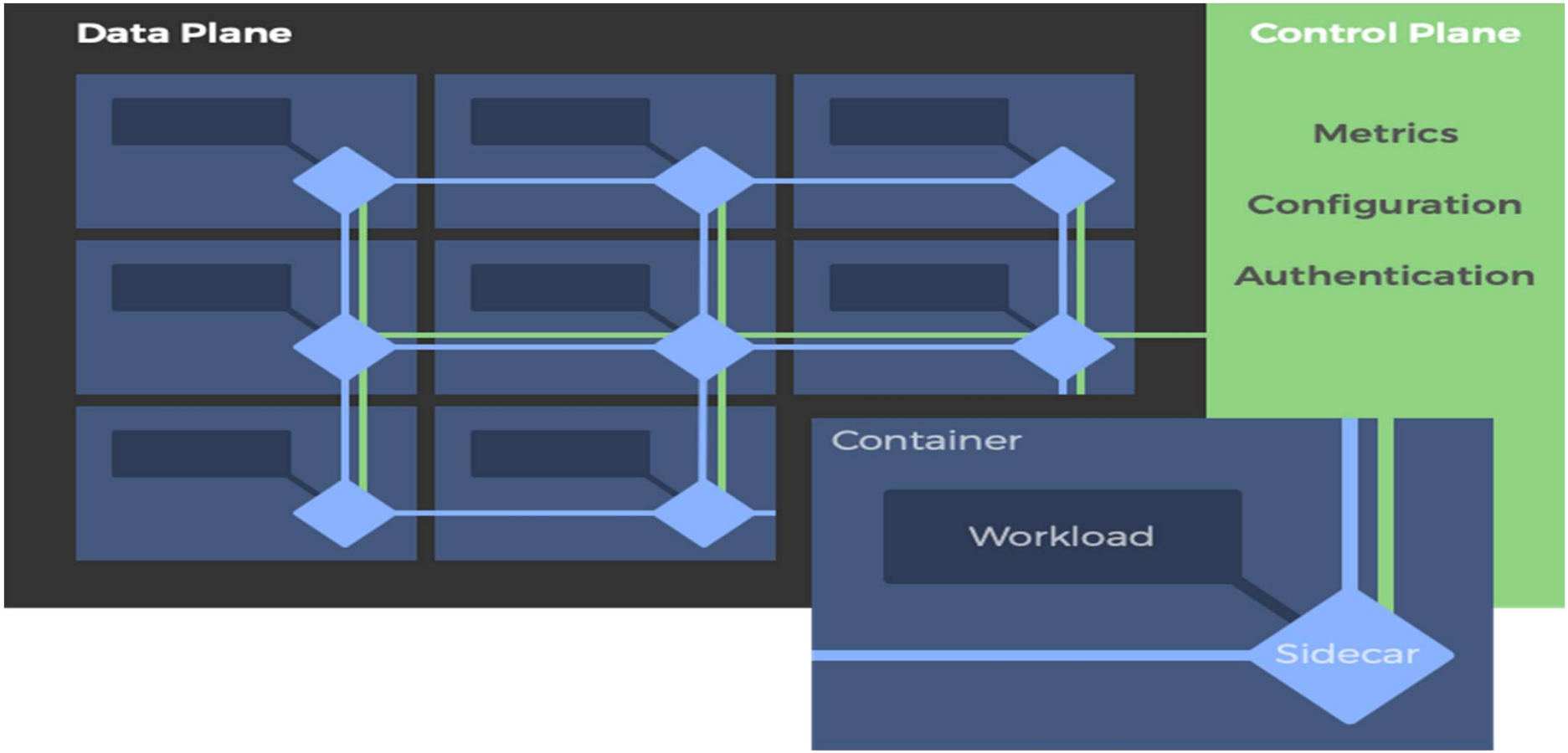
Service Mesh

Service Mesh is a “dedicated infrastructure layer for making service-to-service communication safe, fast, and reliable”

Benefits:

- Service discovery: Registry and discovery of services
- Routing: Intelligent load balancing and network routing, better health checks, automatic deployment patterns such as blue-green or canary deployments
- Resilience: retries, timeouts and circuit breakers
- Security: TLS-based encryption including key management
- Telemetry: collection of metrics and tracing identifiers

Service Mesh Architecture



Service Mesh- Key Players

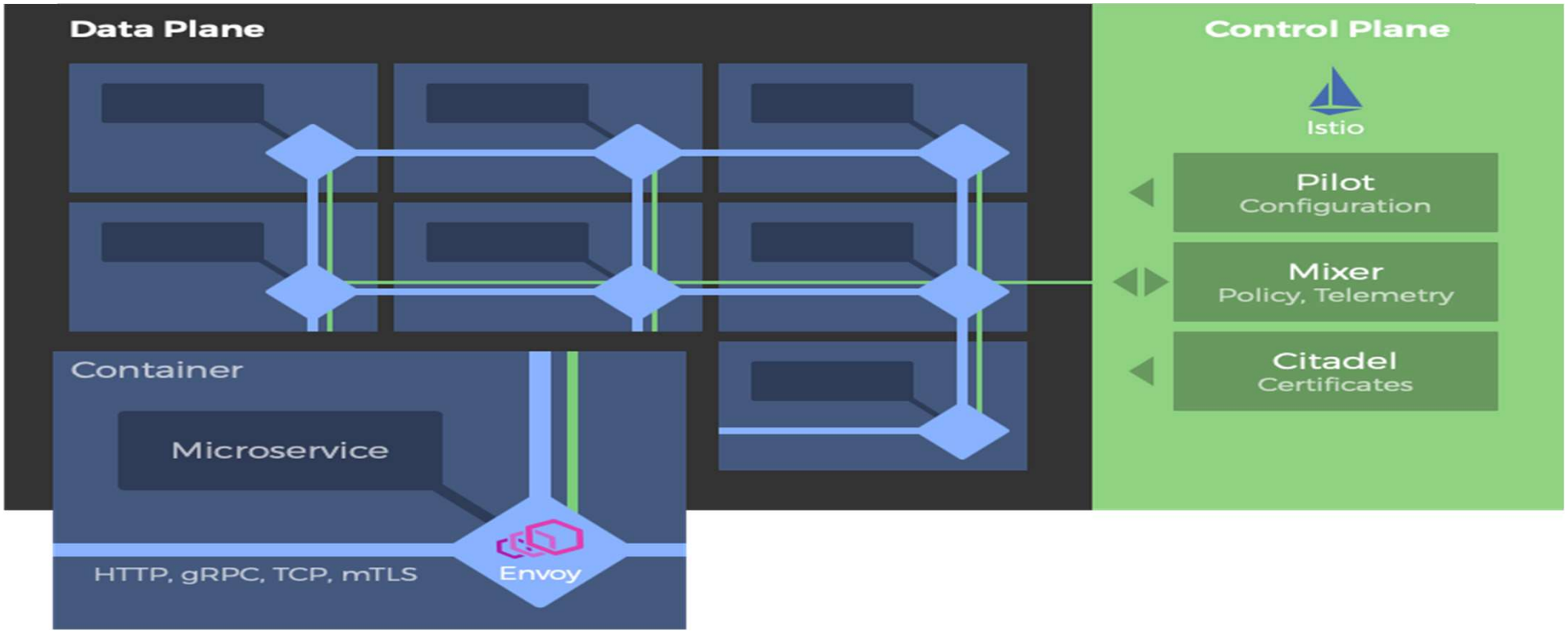
Envoy: an open-source proxy server developed at Lyft. Mainly Dataplane with convenient configuration API.

Linkerd: An open-source project sponsored by Buoyant and the original service mesh.

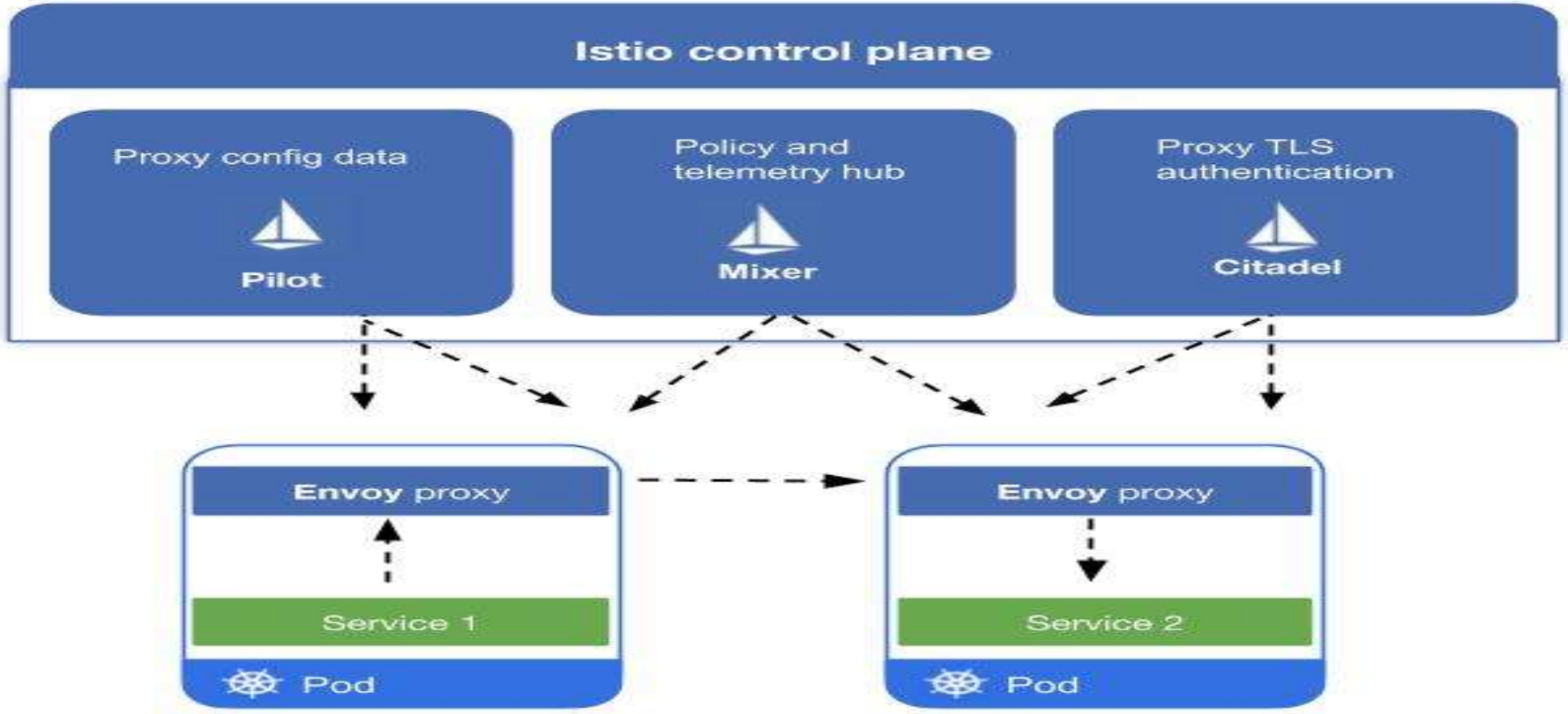
Istio: The most popular service mesh today. It was launched jointly by Google, IBM and Lyft and is expected to ultimately join the CNCF. Istio is a control plane that needs to be paired with a data plane to form a service mesh.

Consul: Sponsored by hashicorp. Specializes in service discovery. Consul works with several data planes and can be used with or without other control planes such as Istio.

Service Mesh- ISTIO



Service Mesh- ISTIO



Service Mesh- ISTIO

Pilot: Connectivity and Communication

- Traffic management
- Fault injection
- Layer 7 Load balancing

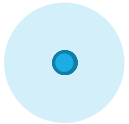
Mixer: Monitoring and Observability

- Backend abstraction with policy control and telemetry collections
- Latency
- Reliability- Additional caching for Sidecars

Citadel(Galley): Encryption and Authentication

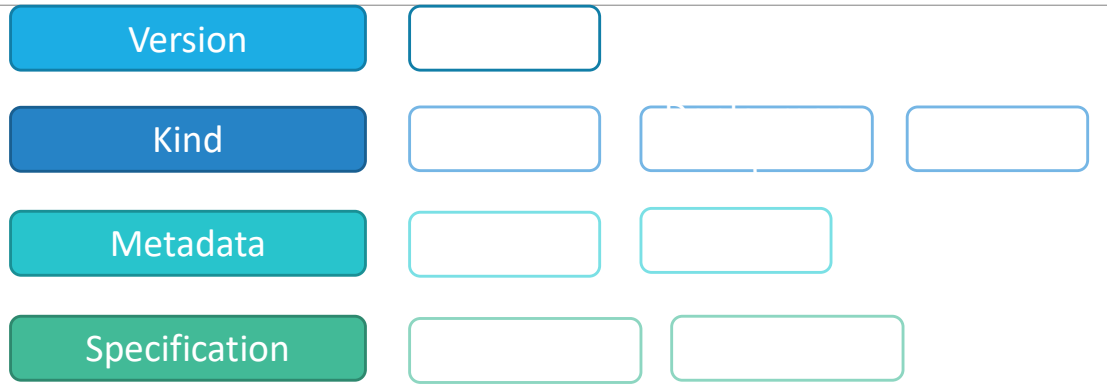
- Service authentication
- Authentication policy
- RBAC
- TLS authentication & Key management

Microservices



Sandeep Kumar Sharma

Kubernetes definition File



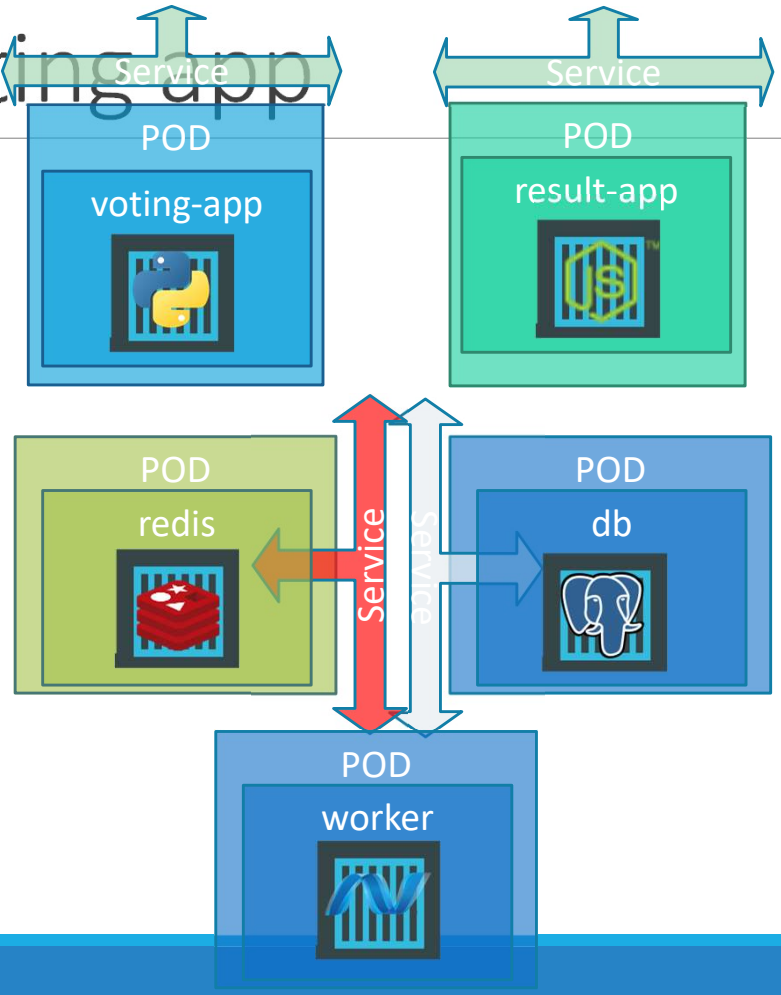
pod-definition.yml

```
apiVersion: v1


kind: pod


metadata:
    name: my-nginx

spec:

    containers:
      - name: nginx
        image: nginx
        ports:
          - containerPort: 80


kubectl create –f pod-definition.yml
```

Example voting app

Goals:

1. Deploy Containers
2. Enable Connectivity
3. External Access



Steps:

1. Deploy PODs
2. Create Services (ClusterIP)
3. Create Services (LoadBalancer)


Q&A

Deployment

Updates and Rollback

Sandeep Kumar Sharma

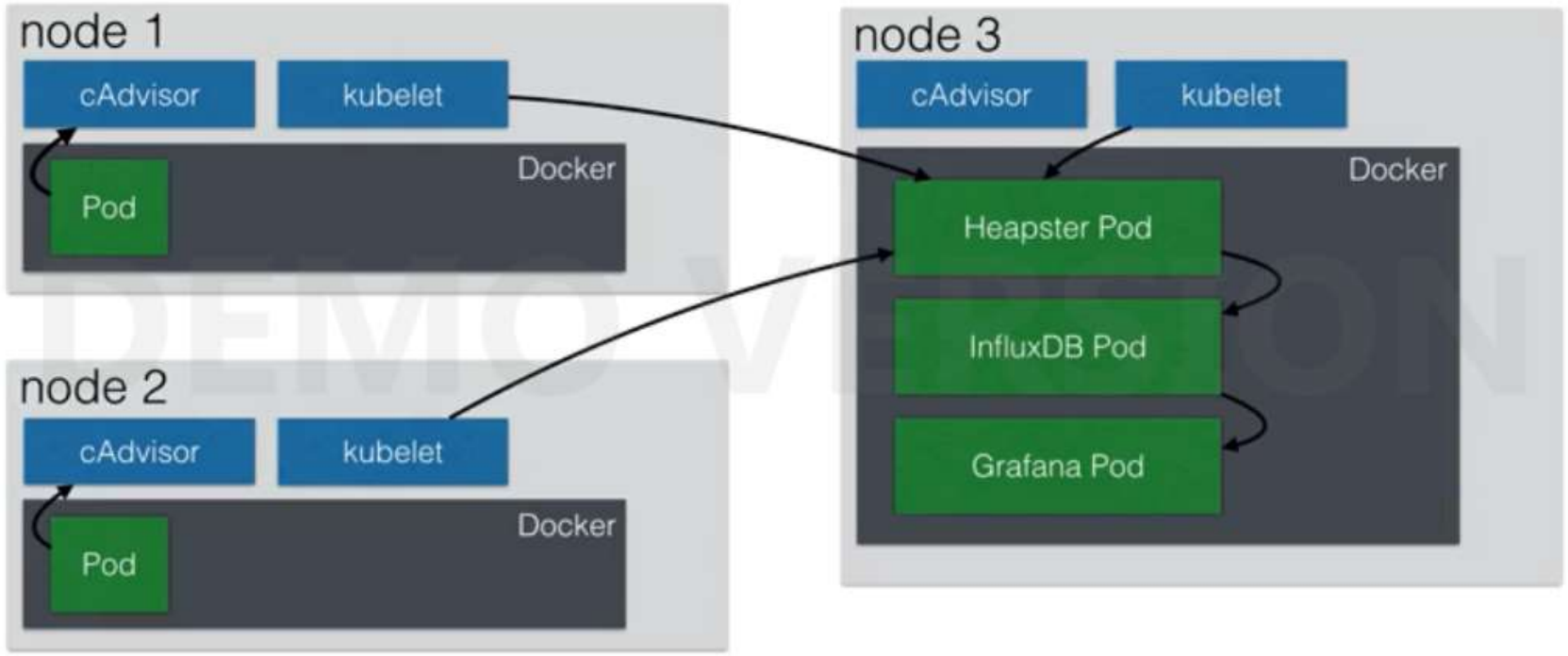
ResourceUsage Monitoring

Sandeep Kumar Sharma

ResourceMonitoring

- Heapster enables Container Cluster Monitoring and Performance Analysis
- It provides a monitoring platform for Kubernetes
- Heapster exports cluster metrics via REST endpoints
- We will use Heapster, InfluxDB and Grafana Dashboards for Visualizations
- All these 3 are started in PODS
- https://github.com/kubernetes/heapster
- We need to change some of the configurations, we will see in DEMO

ResourceMonitoring



Stay connected

Sandeep Kumar Sharma.singh@techlanders.com

contactus@techlanders.com

https://www.linkedin.com/in/Sandeep Kumar Sharma-gawri

+91 9654656667