What is Kubernetes?

- Started by Google in 2014. First released in 2015
- Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.
- Donated to Cloud Native Computing Foundation
- 100% open source
- Written in Go language

Certified Kubernetes Administrator Program

- The Certified Kubernetes Administrator (CKA) program was created by the Cloud Native Computing Foundation (CNCF), in collaboration with The Linux Foundation, to help develop the Kubernetes ecosystem.
- The purpose of the CKA program is to provide assurance that CKAs have the skills, knowledge, and competency to perform the responsibilities of Kubernetes administrators
- Performance-based problems to be solved in a command line.
- Time 3 Hours
- Cost $300 with 1 free retake
- Questions 24
- 74% passing

CKA domains & weights

- Installation, Configuration & Validation 12%
- Application Lifecycle Management 8%
- Core Concepts 19%
- Networking 11%
- Scheduling 5%
- Security 12%
- Cluster Maintenance 11%
- Logging / Monitoring 5%
- Storage 7%
- Troubleshooting 10%

CKA Exam Curriculum

Installation, Configuration & Validation – 12%

- Design a Kubernetes Cluster
- Install Kubernetes Masters and Nodes
- Configure a highly-available Kubernetes cluster
- Know where to get the Kubernetes release binaries
- Provision underlying infrastructure to deploy a Kubernetes cluster
- Choose a network solution
- Choose your Kubernetes infrastructure configuration
- Analyze end-to-end test results
- Run Node end-to-end Tests
- Install and use kubeadm to install, configure, and manage clusters

CKA Exam Curriculum

Core Concepts – 19%

- Understand the Kubernetes API primitives
- Understand the Kubernetes cluster architecture
- Understand Services and other network primitives

Application Lifecycle Management – 8%

- Understand deployments and how to perform rolling update and rollbacks
- Know various ways to configure applications
- Know how to scale applications
- Understand the primitives necessary to create a self-healing application

CKA Exam Curriculum

Networking – 11%

- Understand the networking configuration on the cluster nodes
- Understand Pod networking concepts
- Understand Service Networking
- Deploy and configure network load balancer
- Know how to use Ingress rules
- Know how to configure and use the cluster DNS
- Understand CNI

CKA Exam Curriculum

Scheduling – 5%

- Use label selectors to schedule Pods
- Understand the role of DaemonSets
- Understand how resource limits can affect Pod scheduling
- Understand how to run multiple schedulers and how to configure Pods to use them
- Manually schedule a pod without a scheduler
- Display scheduler events

CKA Exam Curriculum

Security – 12%

- Know how to configure authentication and authorization
- Understand Kubernetes security primitives
- Know how to configure network policies
- Create and manage TLS certificates for cluster components
- Work with images securely
- Define security contexts
- Secure persistent key value store

CKA Exam Curriculum

Cluster Maintenance – 11%

- Understand Kubernetes cluster upgrade process
- Facilitate operating system upgrades
- Implement backup and restore methodologies

Logging / Monitoring – 5%

- Understand how to monitor all cluster components
- Understand how to monitor applications
- Manage cluster component logs
- Manage application logs

CKA Exam Curriculum

Storage – 7%

- Understand persistent volumes and know how to create them
- Understand access modes for volumes
- Understand persistent volume claims primitive
- Understand Kubernetes storage objects
- Know how to configure applications with persistent storage

Troubleshooting – 10%

- Troubleshoot application failure
- Troubleshoot control plane failure
- Troubleshoot worker node failure
- Troubleshoot networking

Certification Details

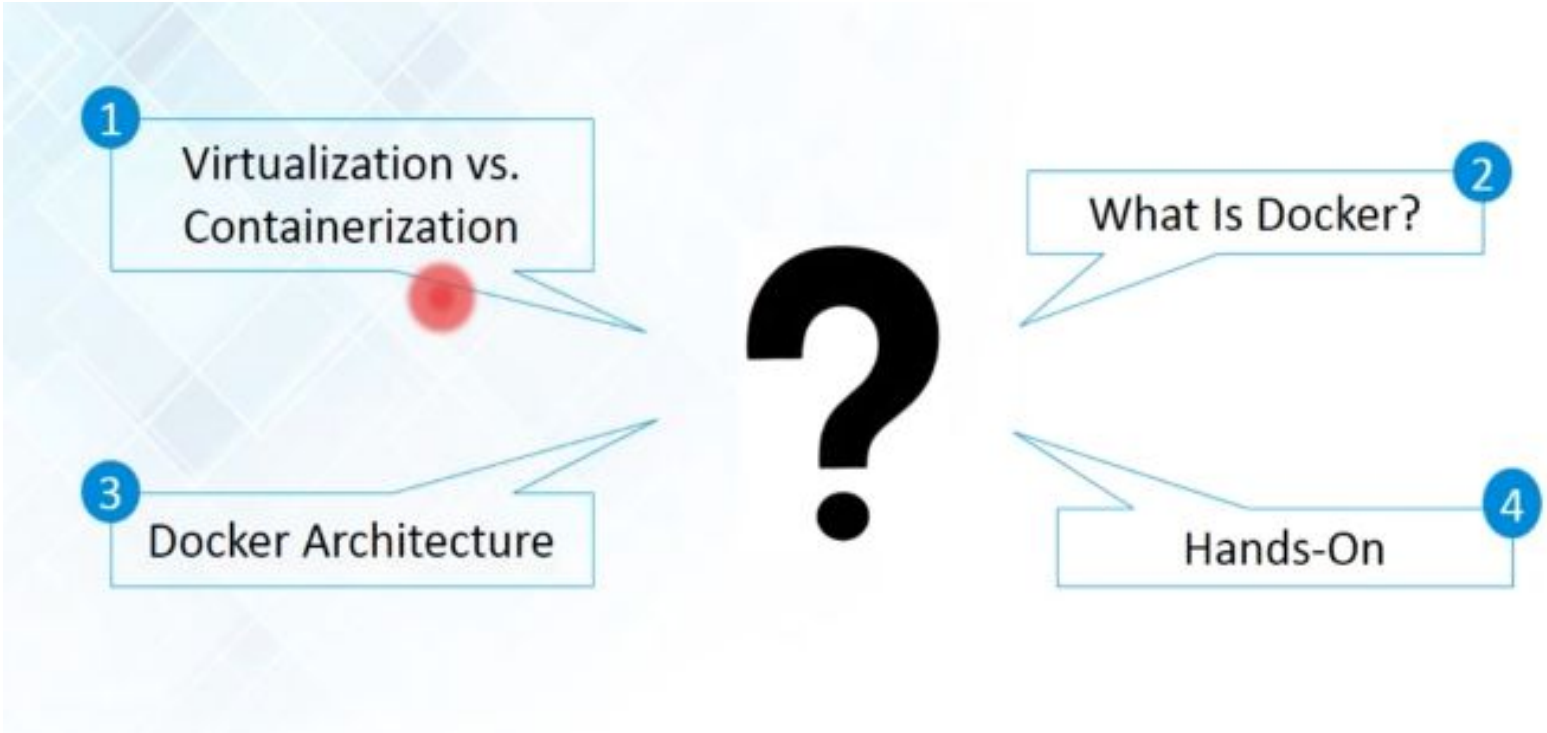
Certified Kubernetes Administrator: https://www.cncf.io/certification/cka/

Exam Curriculum (Topics): https://github.com/cncf/curriculum

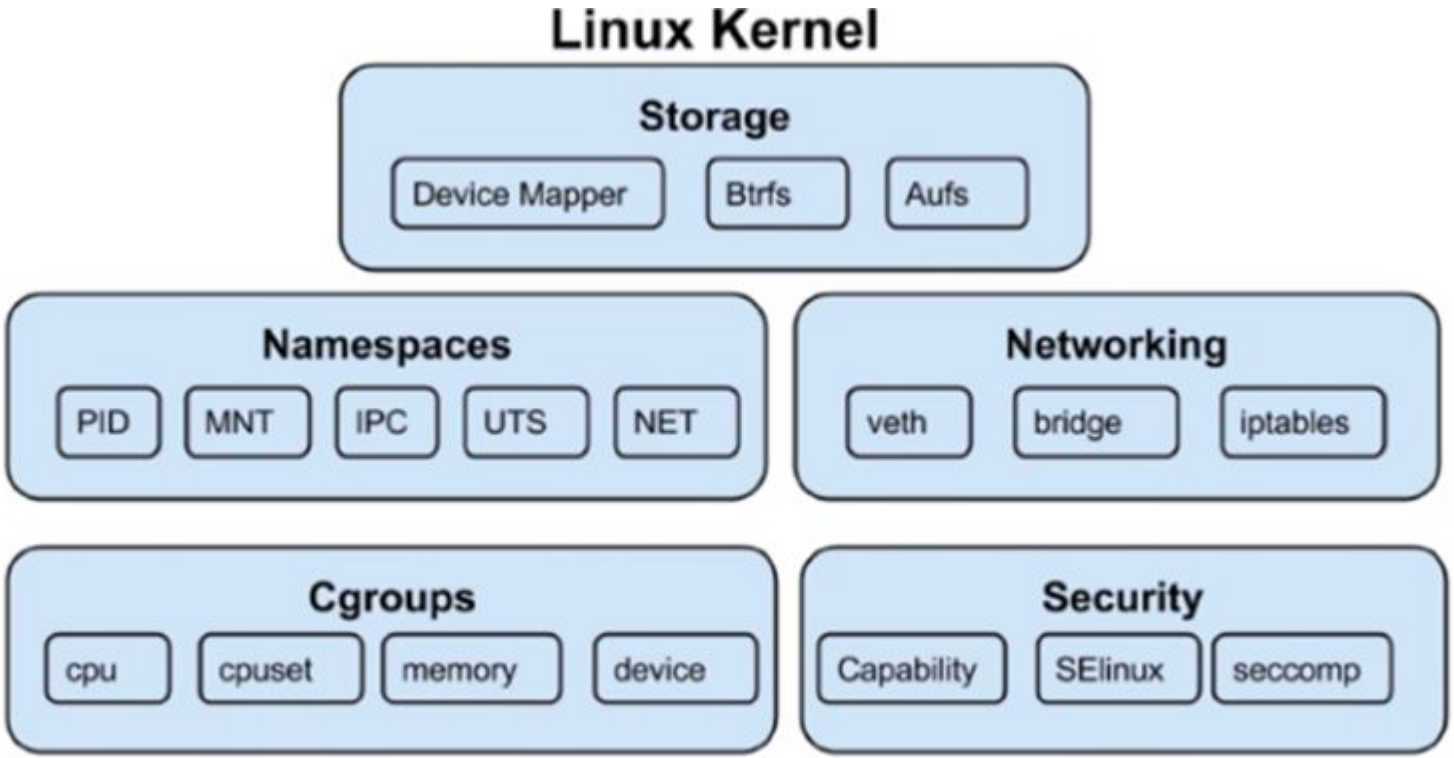
Candidate Handbook: https://www.cncf.io/certification/candidate-handbook

Exam Tips: http://training.linuxfoundation.org/go//Important-Tips-CKA-CKAD

Docker


1
Virtualization vs. Containerization
2
What Is Docker?
?
3
Docker Architecture
4
Hands-On

Docker Internals



Docker Internals

- Container are method of operating system virtualization that allow you to run application and its dependencies in resource isolated process.
- Containers allows you to easily package an application code, configurations and dependencies into easy to use building blocks and that application can be deployed quickly and consistently regardless of deployment environment

Docker Internals

Linux kernel features that create the walls between container and other processes running on the host.

To understand Containers We have to start with **Linux Cgroup and Namespace & Union File System.**

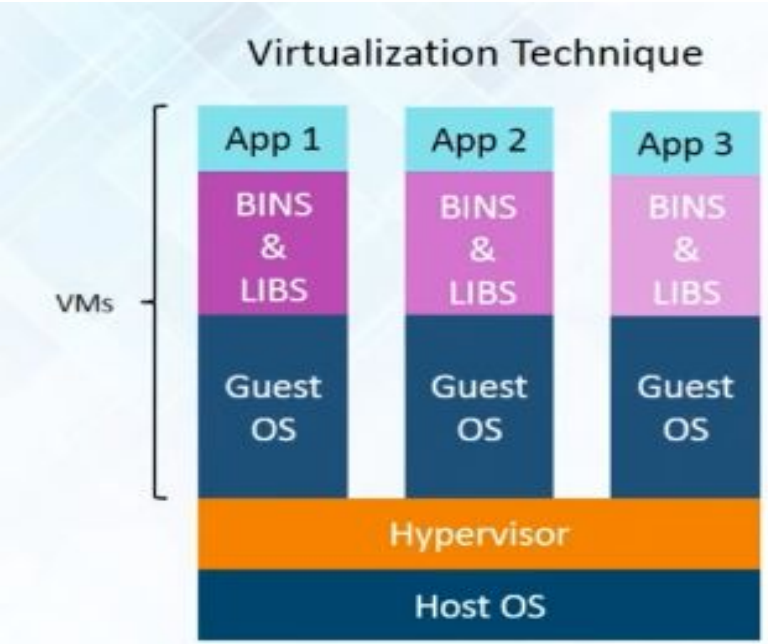
Linux Namespace - Wrap a set of system resources and present them to a process to make it look like they are dedicated to that process.

Docker Internals

Linux Cgroup - Governs the isolation and usage of system resources such as cpu and memory for group of process.E.g. - If you have a application that takes up lot of cpu cycles and memory such as scientific computing application you can put the application in a cgroup to limit a CPU and memory usage.

- **Namespaces** deal with resource isolation for single process
- **Cgroup** manages resources for group of processes

Virtualization



Advantages

- Multiple OS In Same Machine
- Easy Maintenance & Recovery
- Lower Total Cost Of Ownership

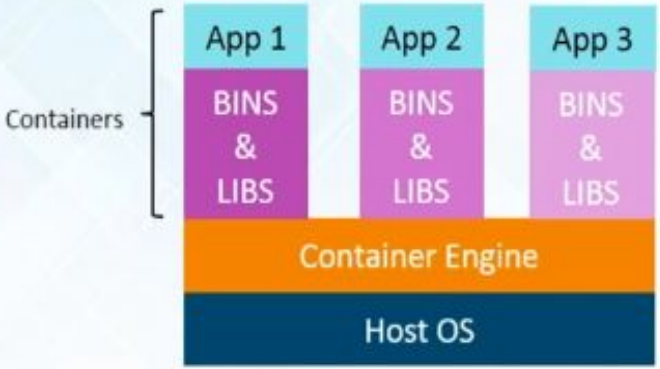
Disadvantages

- Multiple VMs Lead To Unstable Performance
- Hypervisors Are Not As Efficient As Host OS
- Long Boot-Up Process (Approx. 1 Minute)

Containerization

Note: Containerization Is Just Virtualization At The OS Level

Containerization Technique

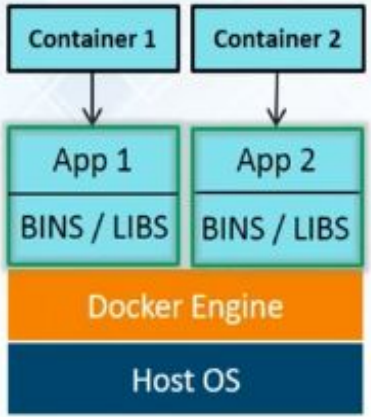


Advantages Over Virtualization

- Containers On Same OS Kernel Are Lighter & Smaller
- Better Resource Utilization Compared To VMs
- Short Boot-Up Process (1/20th of a second)

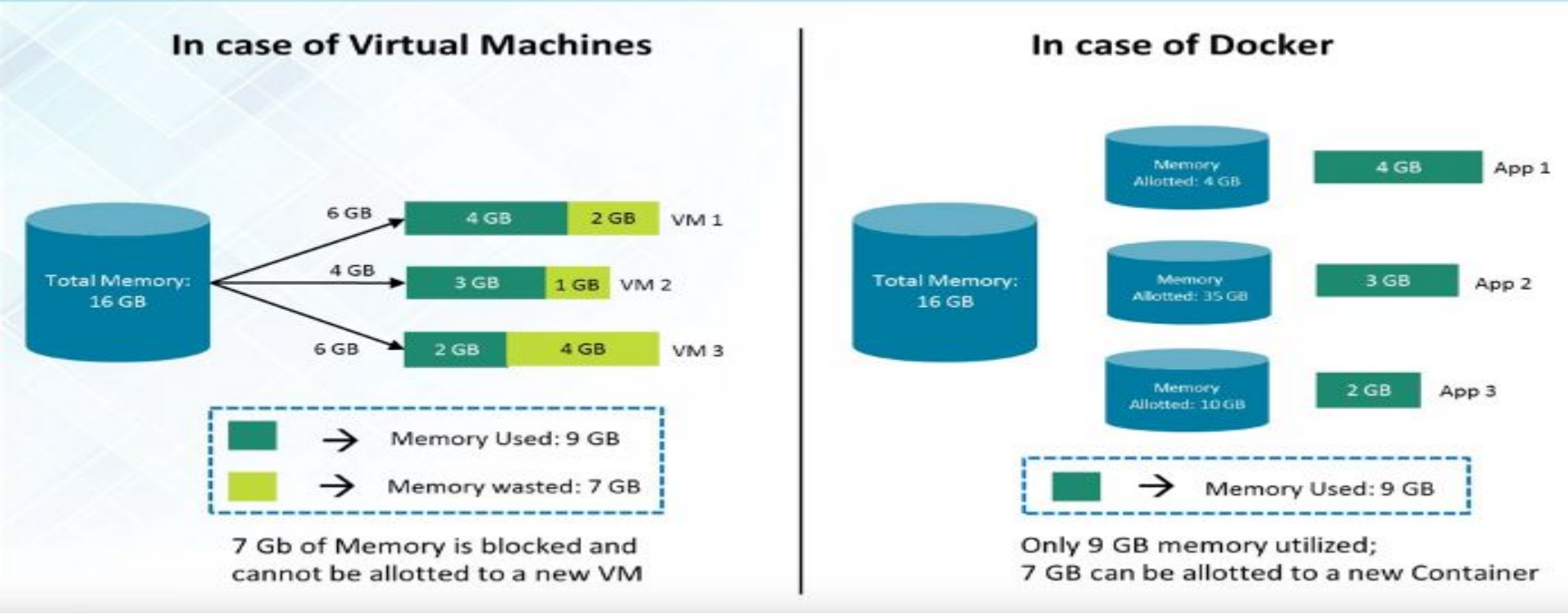
Docker Info

Docker is a Containerization platform which packages your application and all its dependencies together in the form of Containers so as to ensure that your application works seamlessly in any environment be it Development or Test or Production.

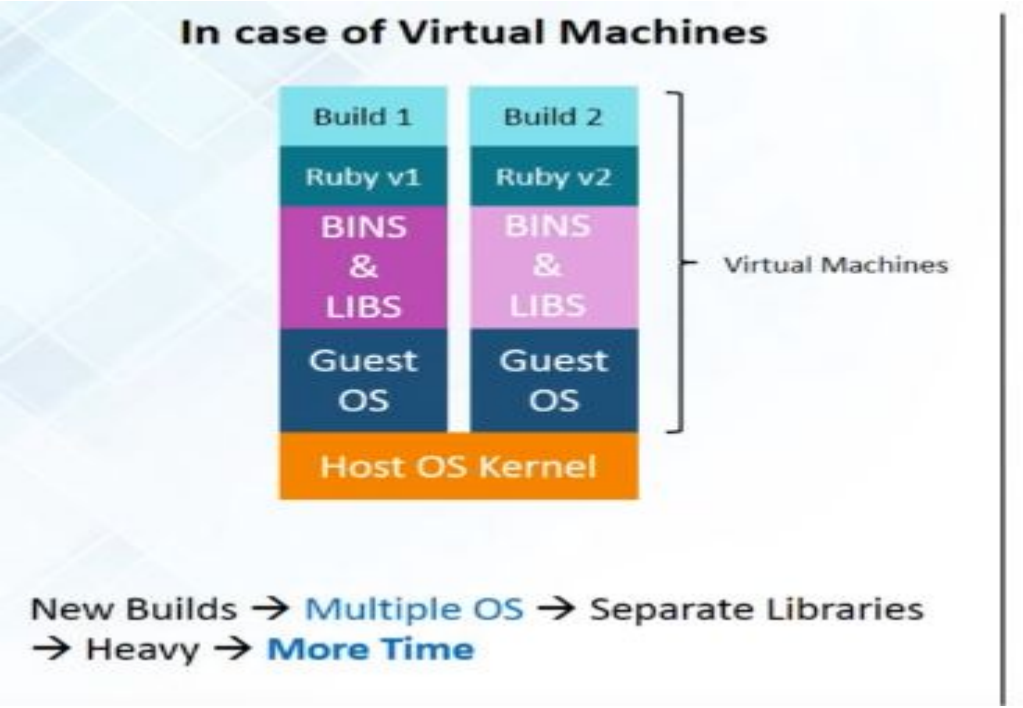


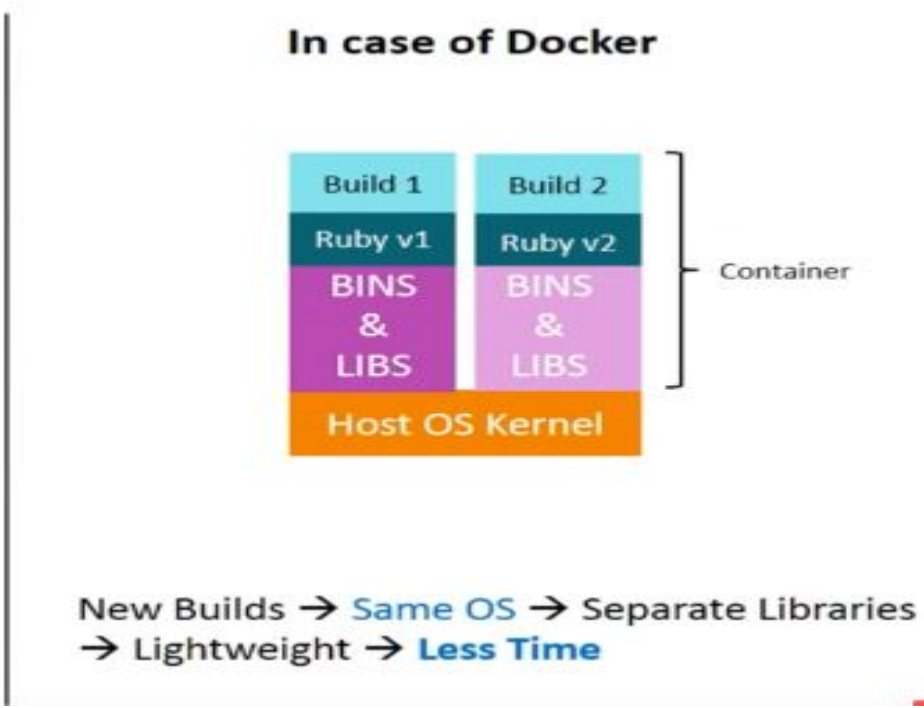
Docker Info

Docker Info

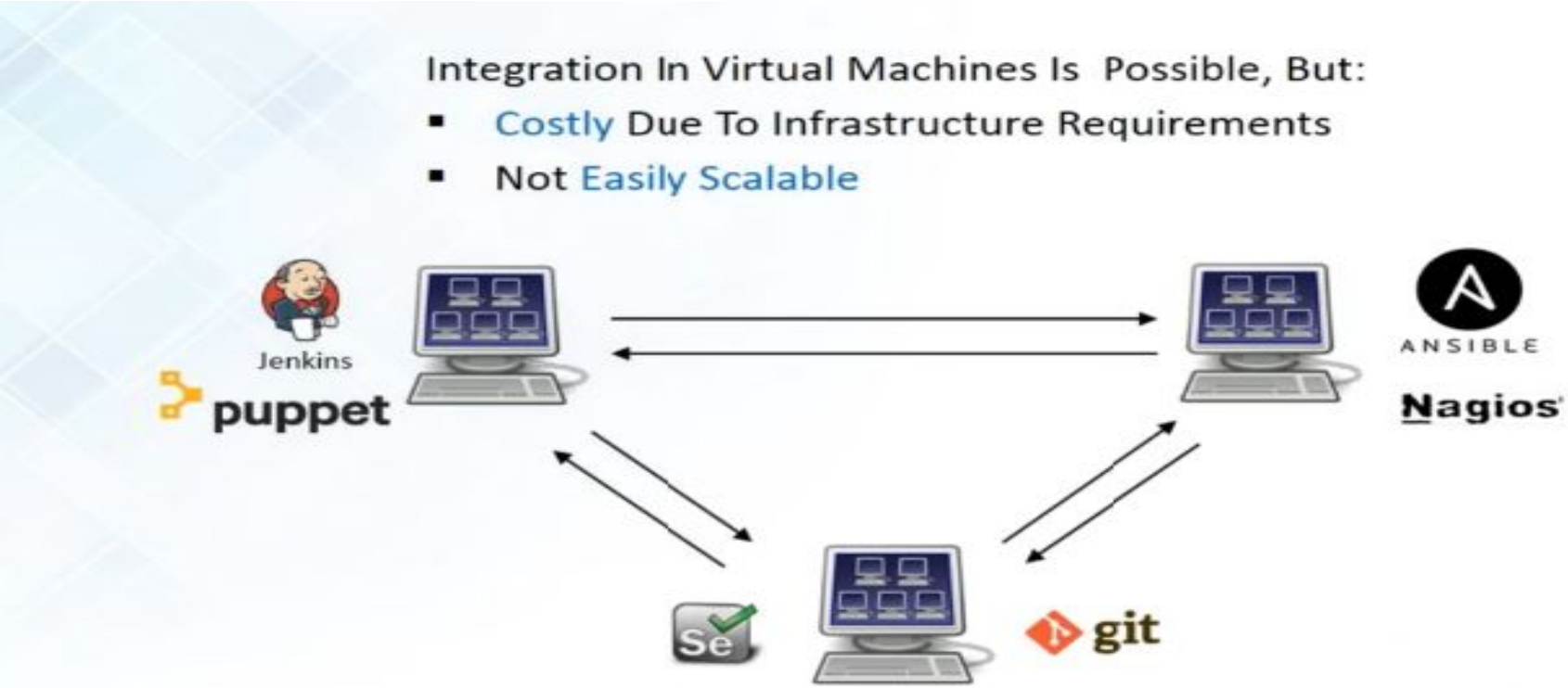

In case of Virtual Machines
Total Memory: 16 GB
6 GB
4 GB
6 GB
4 GB
2 GB
VM 1
3 GB
1 GB
VM 2
2 GB
4 GB
VM 3
Memory Used: 9 GB
Memory wasted: 7 GB
7 Gb of Memory is blocked and cannot be allotted to a new VM
In case of Docker
Total Memory: 16 GB
Memory Allotted: 4 GB
4 GB
App 1
Memory Allotted: 35 GB
3 GB
App 2
Memory Allotted: 10 GB
2 GB
App 3
Memory Used: 9 GB
Only 9 GB memory utilized;
7 GB can be allotted to a new Container

Docker Info


In case of Virtual Machines
Build 1
Build 2
Ruby v1
Ruby v2
BINS & LIBS
BINS & LIBS
Guest OS
Guest OS
Host OS Kernel
Virtual Machines
New Builds → Multiple OS → Separate Libraries → Heavy → More Time


In case of Docker
Build 1
Build 2
Ruby v1
Ruby v2
BINS & LIBS
BINS & LIBS
Host OS Kernel
Container
New Builds → Same OS → Separate Libraries → Lightweight → Less Time

Integration in VMs



Who Can Use Docker

Docker is designed to benefit both Developers and System Administrators, making it a part of many DevOps toolchains.

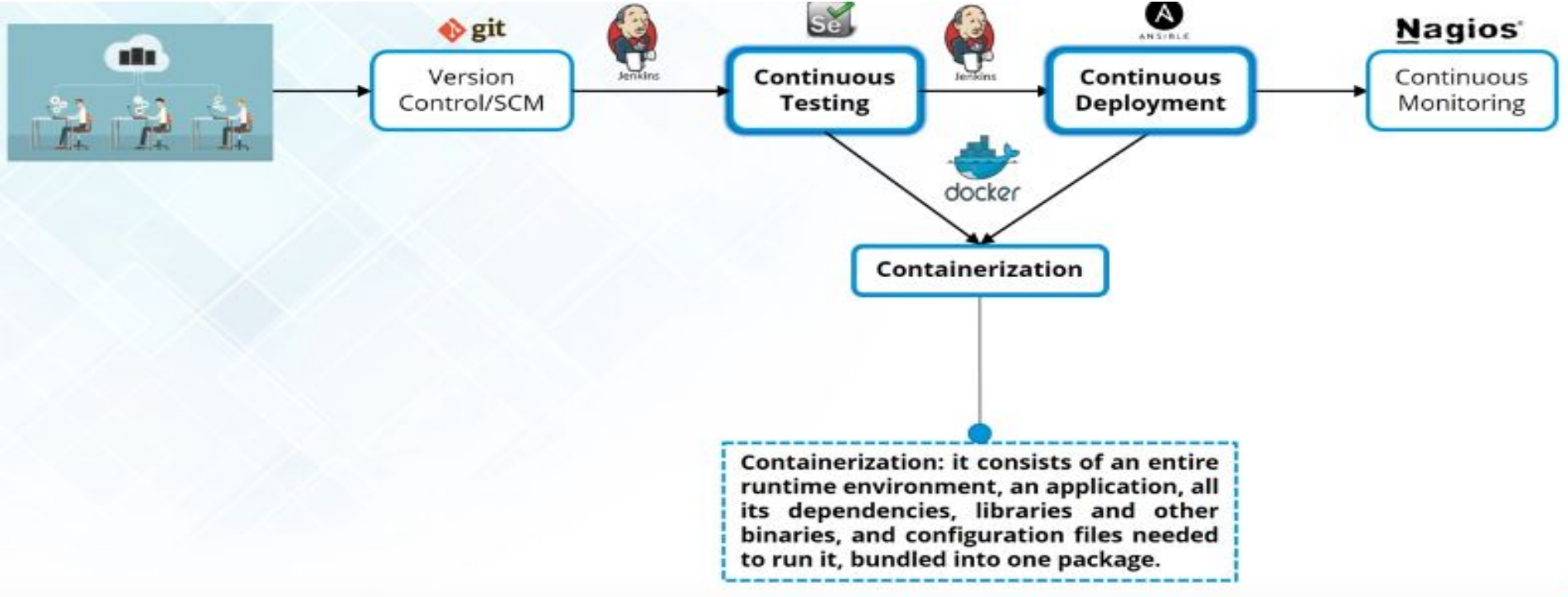
Developers can write code without worrying about the testing / production environment

Dev

OPS

SysAdmins need not worry about Infrastructure as Docker can easily scale up / scale down the no. of systems

How is Docker Used In Devops



Docker Images & Containers

run

Docker Images

Docker Containers

- Read Only Template Used To Create Containers
- Built By Docker Users
- Stored In Docker Hub Or Your Local Registry

- Isolated Application Platform
- Contains Everything Needed To Run The Application
- Built From One Or More Images

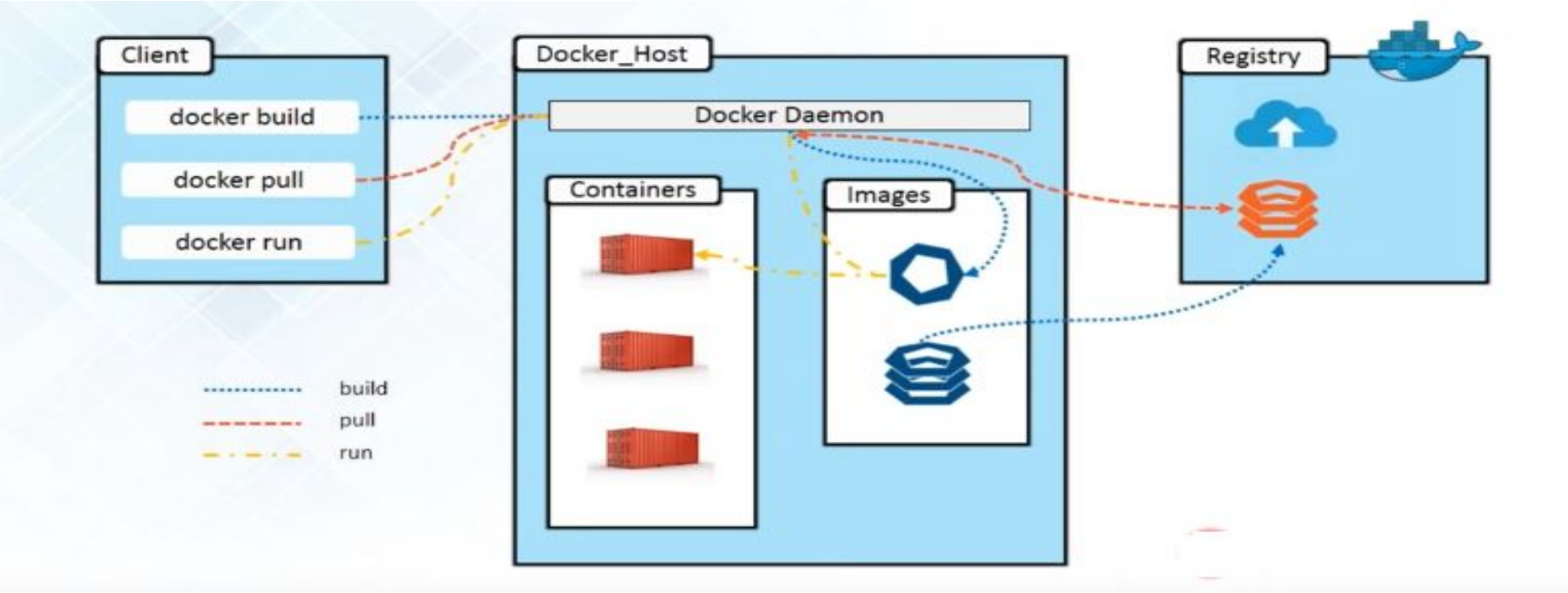
Docker Registry

- Docker Registry is a storage component for Docker Images
- We can store the Images in either Public / Private repositories
- Docker Hub is Docker's very own cloud repository

Why Use Docker Registries?

- Control where your images are being stored
- Integrate image storage with your in-house development workflow

Docker Architecture



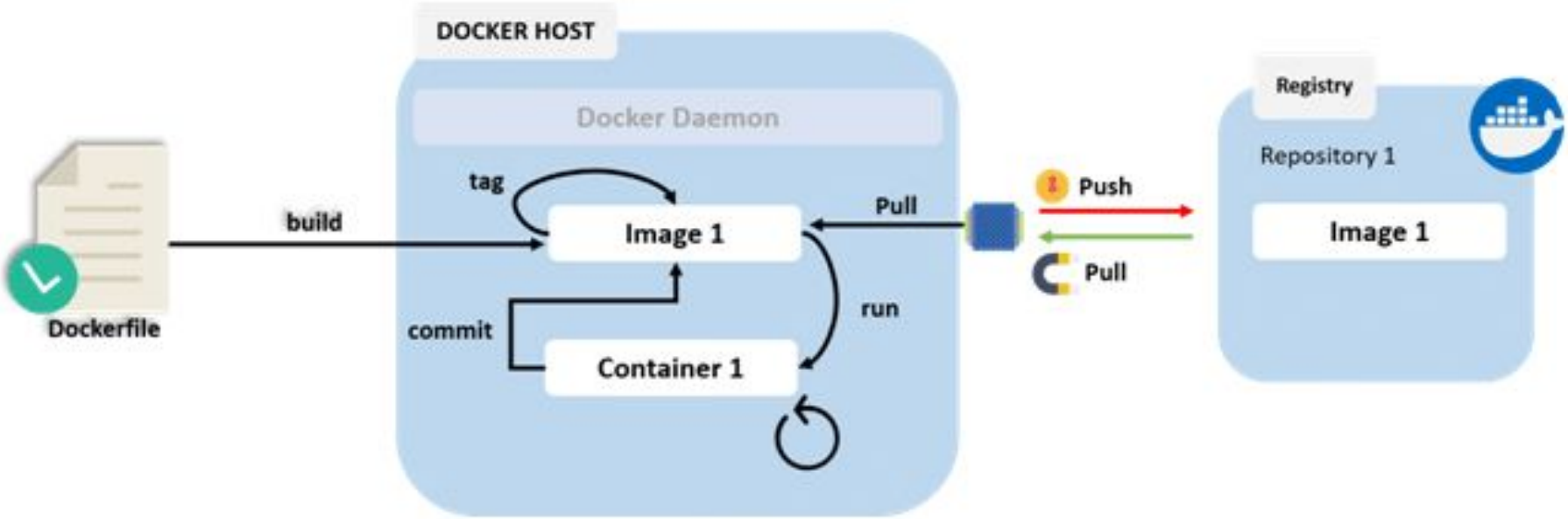
Docker Installation

```
sudo apt-get install apt-transport-https ca-certificates curl
software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

sudo apt-get update

sudo apt-get install docker-ce

docker --version

sudo docker run hello-world
```

Docker Compose Installation

```
sudo apt-get -y install python-pip

sudo pip install docker-compose

COMPOSE FILE:   docker-compose.yml

My-test:

      image: hello-world



docker-compose up
```

Typical Workflow



Basic Docker Commands

To Pull a Docker image from the Docker hub, we can use the command:
$ docker pull <image-name:tag>

To Run that image, we can use the command:
$ docker run <image-name:tag> or **$ docker run <image-id>**

To list down all the images in our system, we can give the command:
$ docker images

To list down all the running containers, we can use the command:
$ docker ps

To list down all the containers (even if they are not running), we can use the command:
$ docker ps -a

Building Images

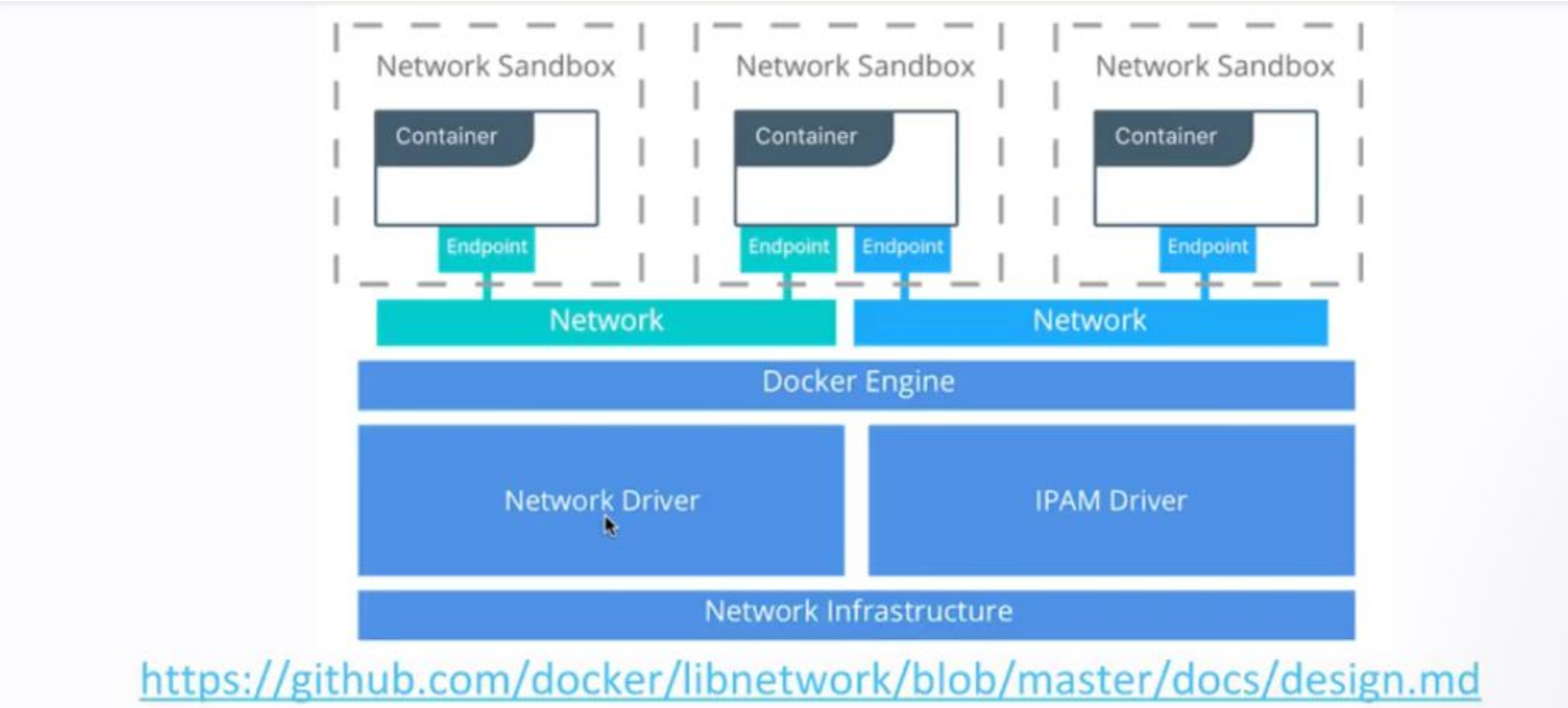
- Images are comprised of multiple layers
- Each layer in an Image is an Image of its own
- They comprise of a Base Image layer which is read-only
- Any changes made to an Image are saved as layers on top of the Base Image layer
- Containers are generated by running the Image layers which are stacked one above the other

Docker File Instructions

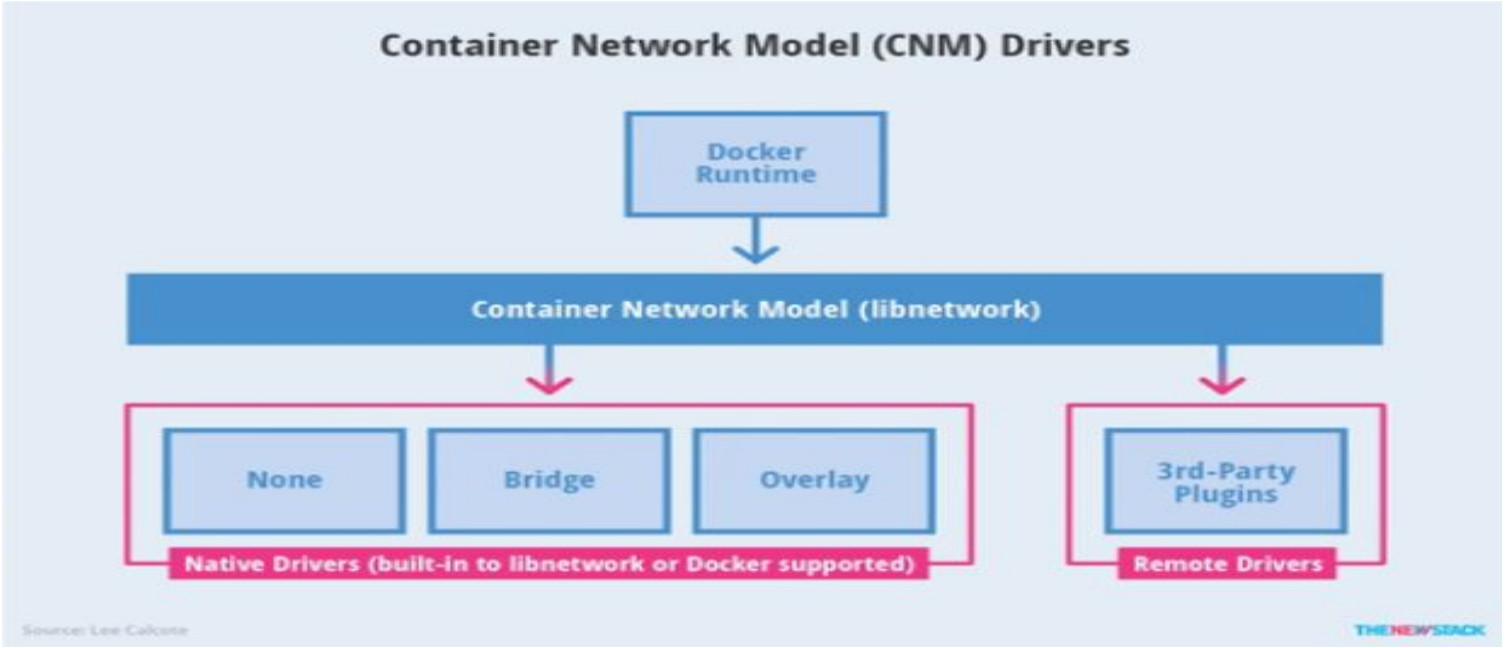
https://docs.docker.com/engine/reference/builder/#environment-replacement

Instruction	Comments
FROM <image>:<tag>	Tells Docker what will be the base for the image being created
ADD	Copies the files from the source on the host into the Docker image's own filesystem at the desired destination. Command is not only about copying files from the local filesystem--you can use it to get the file from the network.
COPY	It will copy new files or directories from <source path> and adds them to the filesystem of the container at the path <destination path>.
CMD / ENTRYPOINT	CMD ["executable","parameter1","parameter2"] his is the so-called exec form CMD command parameter1 parameter2 This a shell form of the instruction
RUN	Instruction will execute any commands in a new layer on top of the current image and then commit the results
EXPOSE	Instruction informs Docker that the container listens on the specified network ports at runtime.

Container Networking Model (CNM)

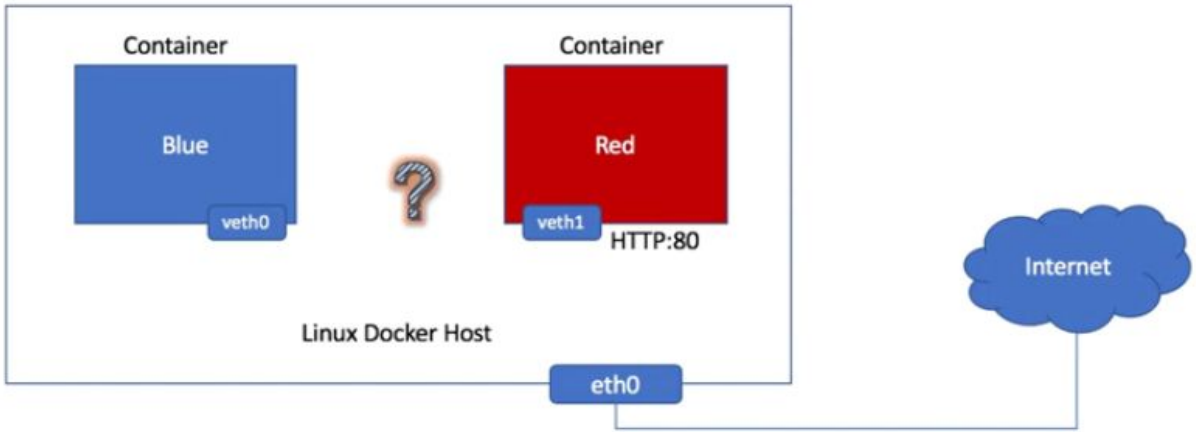


Container Networking Model

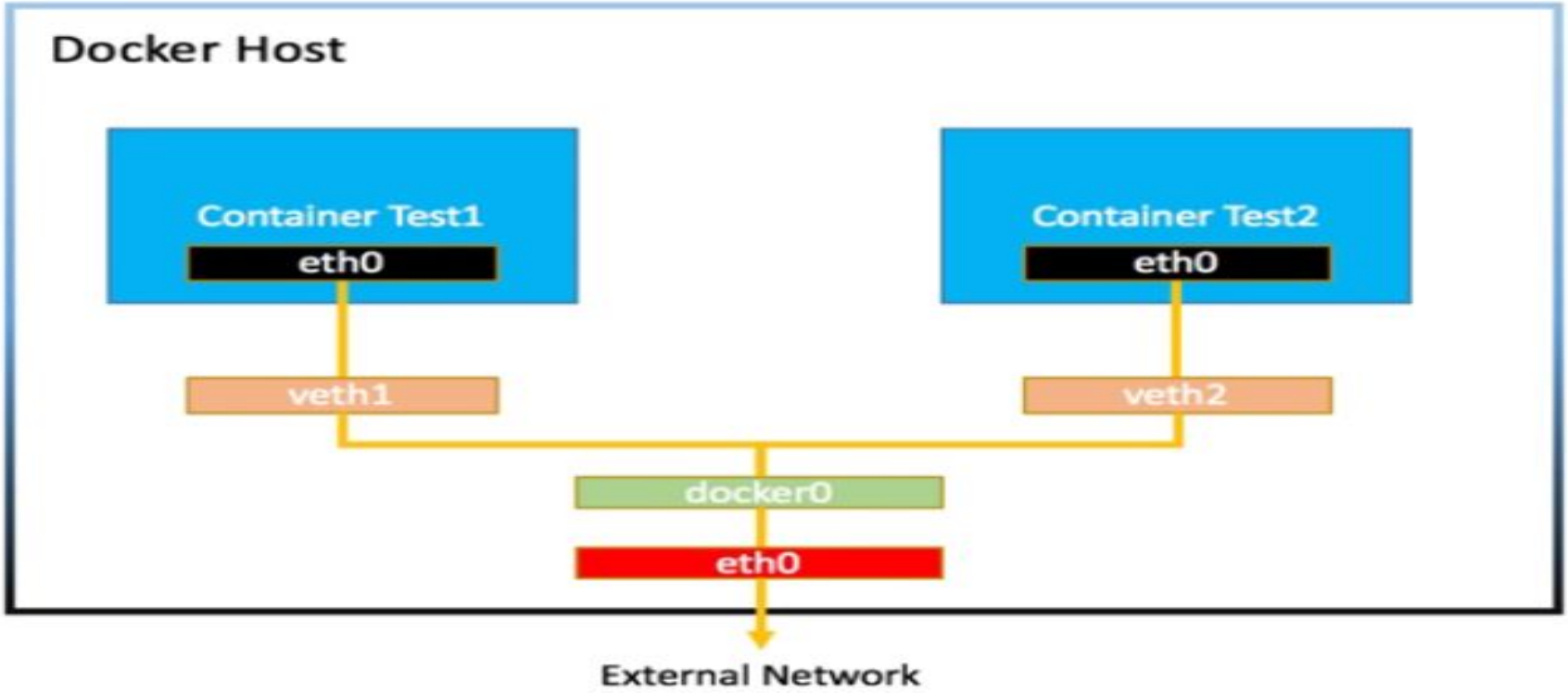


Single Host Container Networking

- How two different containers in the same host communicate with each other?
- How the container communicate with the outside of Linux host (Internet)?
- How access container from the local docker host and outside of the docker host?

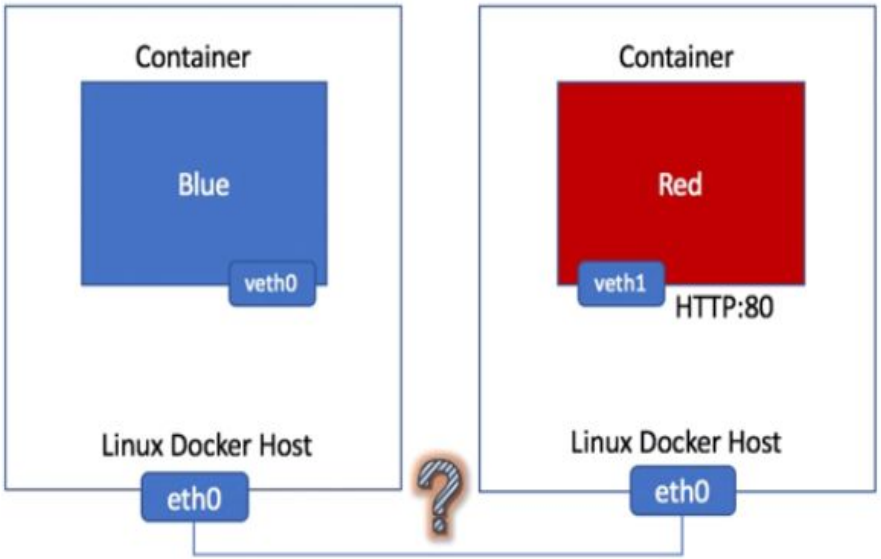


Solution: Linux Bridge and Iptables



Multi Host Container Networking

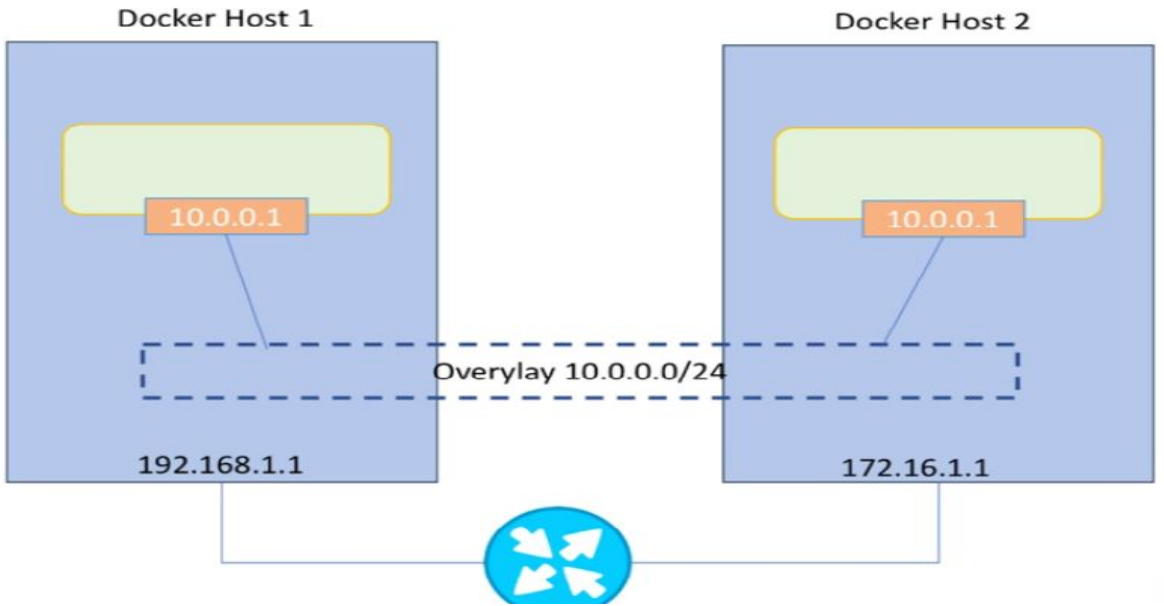
➔ How two containers located on different docker host communicate with each other?



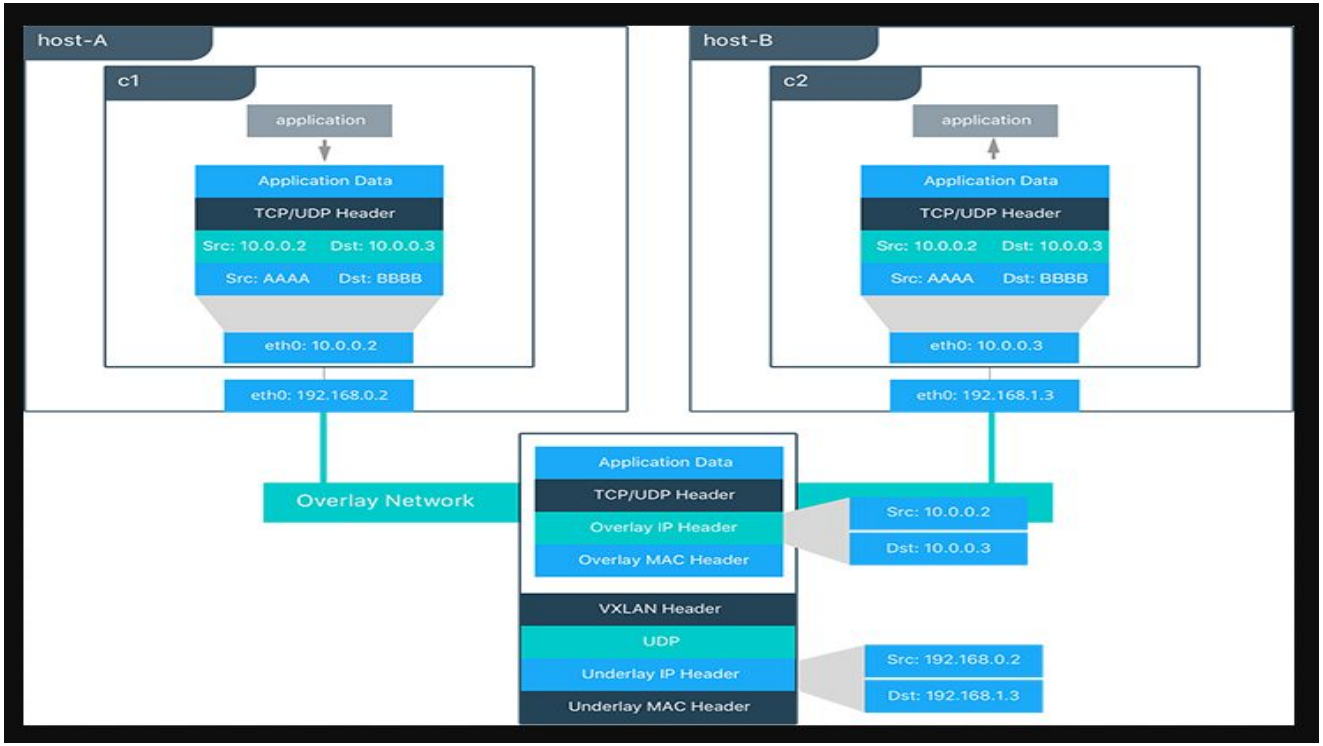
Solution: Tunnel and Routing

- Tunnel
  - Docker build-in overlay network: VXLAN
  - OVS: VXLAN or GRE
  - Flannel: VXLAN or UDP
  - Weave: VXLAN or UDP
- Routing
  - Calico: Layer 3 routing based on BGP
  - Contiv: Layer 3 routing based on BGP

Multi Host Networking



Multi Host Networking



Docker Compose

Multi-container apps are a hassle

- ➔ Build images from Dockerfiles
- ➔ Pull images from the Hub or a private registry Configure and create containers
- ➔ Start and stop containers

Docker Compose

Overview

•Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application’s services. Then, using a single command, you create and start all the services from your configuration.

app.py

```
from flask import Flask
from redis import Redis
import os
import socket

app = Flask(__name__)
redis = Redis(host=os.environ.get('REDIS_HOST', 'redis'), port=6379)

@app.route('/')
def hello():
    redis.incr('hits')
    return 'Hello Container World! I have been seen %s times and my hostname is %s.\n' % (redis.get('hits'),socket.gethostname())

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

requirements.txt

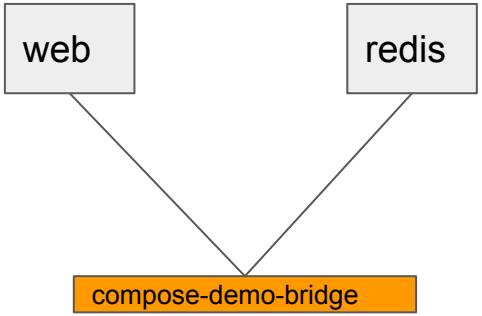
```
flask

redis
```

Dockerfile

```
FROM python:2.7
MAINTAINER R K "rajendrait99@gmail.com"
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
EXPOSE 5000
CMD [ "python", "app.py" ]
```

docker-compose.yml

```yaml
version: "2"

services:
  web:
    build: .
    ports:
      - "80:5000"
    links:
      - redis
    networks:
      - compose-demo-bridge

  redis:
    image: redis
    ports: ["6379"]
    networks:
      - compose-demo-bridge


networks:
  compose-demo-bridge:
```



Why we need Container Orchestration?

- Do you need scaling beyond one host?
- Do you need high availability?
- Are your containers truly stateless?

…

Production requirement

- **Availability** – it just has to be working all the time, with as little downtimes as possible
- **Performance** – our server needs to handle traffic, so performance is important
- **Easy deployment & rollback**
- **Gathering logs & metrics**

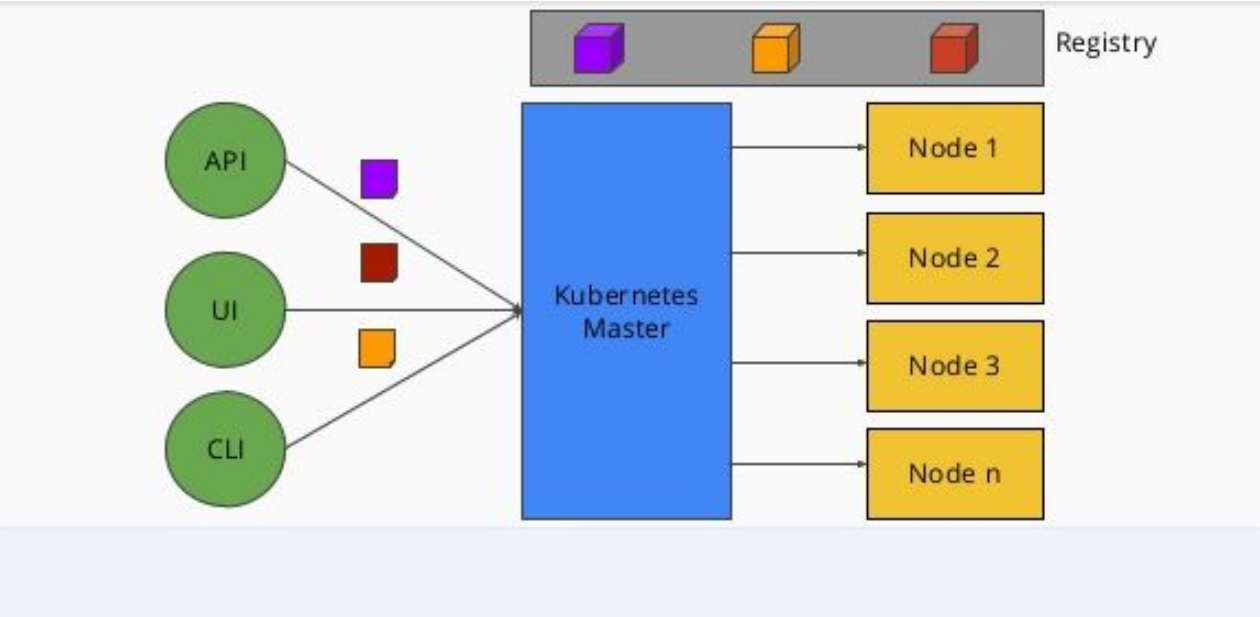
Solutions

- **Kubernetes**
- **Docker Swarm**
- **Amazon EKS**
- **AKS**
- **GKE**

What is Kubernetes?

- Started by Google in 2014. First released in 2015
- Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.
- Donated to Cloud Native Computing Foundation
- 100% open source
- Written in Go language

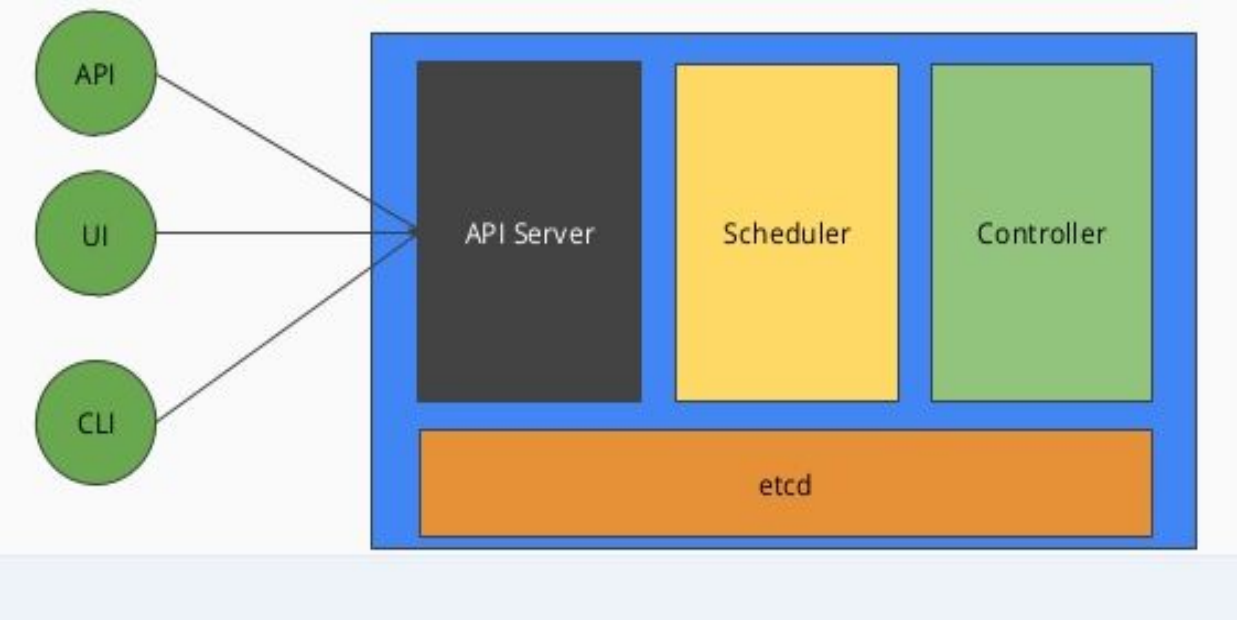
Kubernetes - Architecture



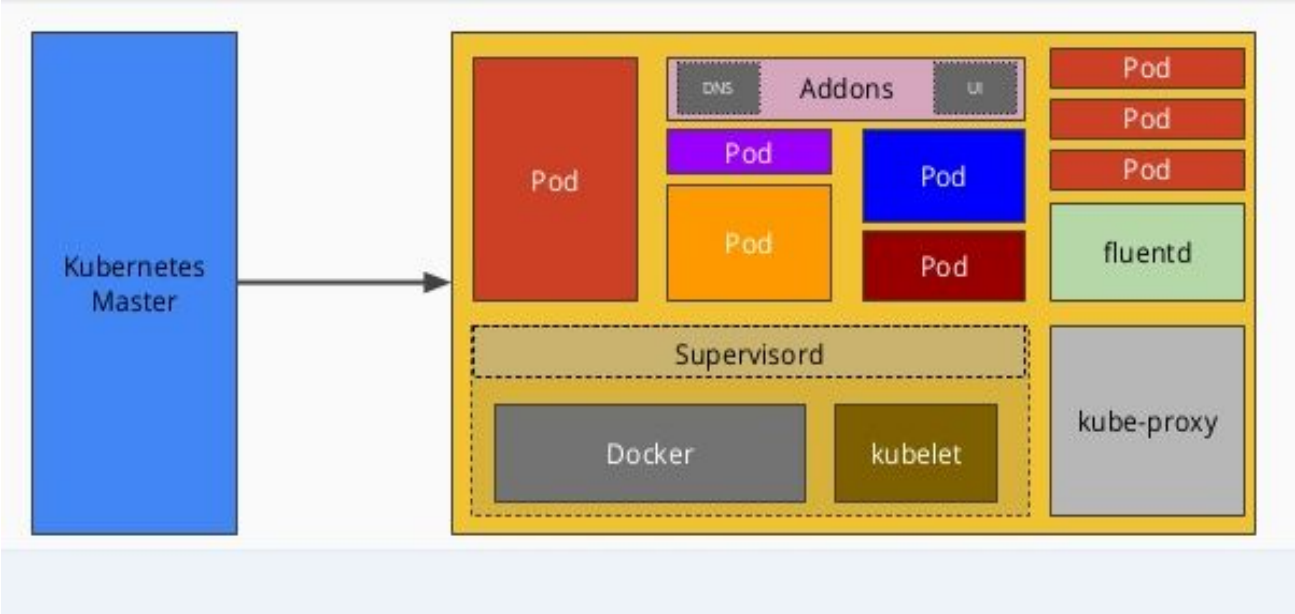
Production requirement

- **Scheduling:** Where do these containers run?
- **Management:** Who manages their life cycle?
- **Service Discovery:** How do they find each other?
- **Load Balancing:** How to route requests?

Kubernetes - Architecture - Master



Kubernetes - Architecture - Node



Multi Node Cluster Setup

1. Run below steps on master and slave machine
   - apt-get update && apt-get install -y apt-transport-https
   - curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
   - cat <<EOF > /etc/apt/sources.list.d/kubernetes.list
     deb http://apt.kubernetes.io/ kubernetes-xenial main
     EOF
2. apt-get update
3. apt-get install -y docker.io
4. apt-get install -y kubelet kubeadm kubectl kubernetes-cni

Multi Node Cluster Setup

Init Master node and Slave Nodes

- Run below command on master node.

kubeadm init --apiserver-advertise-address=<Master IP> --pod-network-cidr=10.244.0.0/16 --skip-preflight-checks

- Copy the join command that you will get after executing above init command.
  Example: kubeadm join --token 8ccfe0.7cd3dd5f91ef9796 <Master IP>:6443
- Execute join command on all the slave nodes
- Start using cluster from Master node

Note : We can do this from any node where kubectl installed

- Run below command to copy configuration to default directory

sudo cp /etc/kubernetes/admin.conf $HOME/

sudo chown $(id -u):$(id -g) $HOME/admin.conf

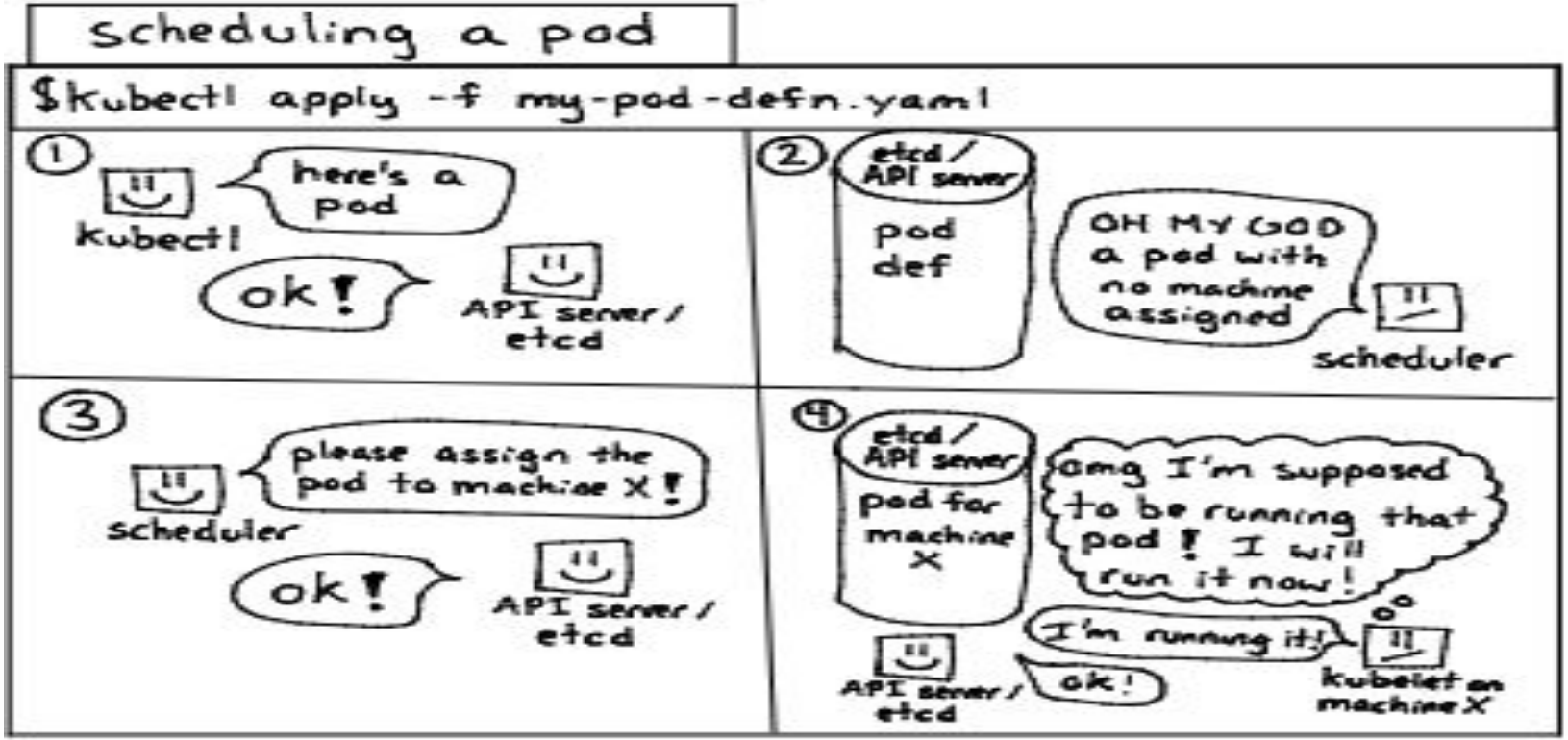
export KUBECONFIG=$HOME/admin.conf

- Setup Kubernetes Network.Run below command on master node

kubectl apply -f

http://docs.projectcalico.org/v2.2/getting-started/kubernetes/installation/hosted/kubeadm/1.6/calico.yaml

Scheduling a Pod



KIND

- **KIND** is a tool for running local Kubernetes clusters using Docker container “nodes”.

- It was primarily designed for testing Kubernetes itself, but may be used for local development or CI.

- **K**ubernetes **IN**side **D**ocker

Kind Installation

curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.8.1/kind-linux-amd64

chmod +x ./kind

```
sudo mv ./kind /usr/local/bin/kind
```

Kubectl Installation

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

```
chmod +x ./kubectl
```

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

Kind config

```
# three node (two workers) cluster config
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
```

Writing YAML

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: rss-site
  labels:
    app: web
spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
    - name: rss-reader
      image: nickchase/rss-php-nginx:v1
      ports:
        - containerPort: 88
```

Kubernetes Namespace

- Namespace as a virtual cluster inside your Kubernetes cluster
- Default namespace
- Creating Namespaces
- Viewing Namespaces
- Creating Resources in the Namespace
- Viewing resources in the Namespace
- Managing test, staging, and production environments within the same cluster

Kubernetes - Services

- Exposing PODs to the outside world. Services are required for discovery.
- Labels and selectors are used to route to the appropriate POD
- Service Types
    - ClusterIP - Services makes internal pod accessible
    - NodePort - Allow external user traffic and load balancing
    - LoadBalancer - Service does load balancing with external load balancer

Kubernetes - Services

```yaml
kind: Service
apiVersion: v1

metadata:
  name: hostname-service

spec:
  type: NodePort
  selector:
    app: echo-hostname
  ports:
    - nodePort: 30163
      port: 8080
      targetPort: 80
```

Make the service available to network requests from external clients

Forward requests to pods with label of this value

`nodePort`
access service via this external port number

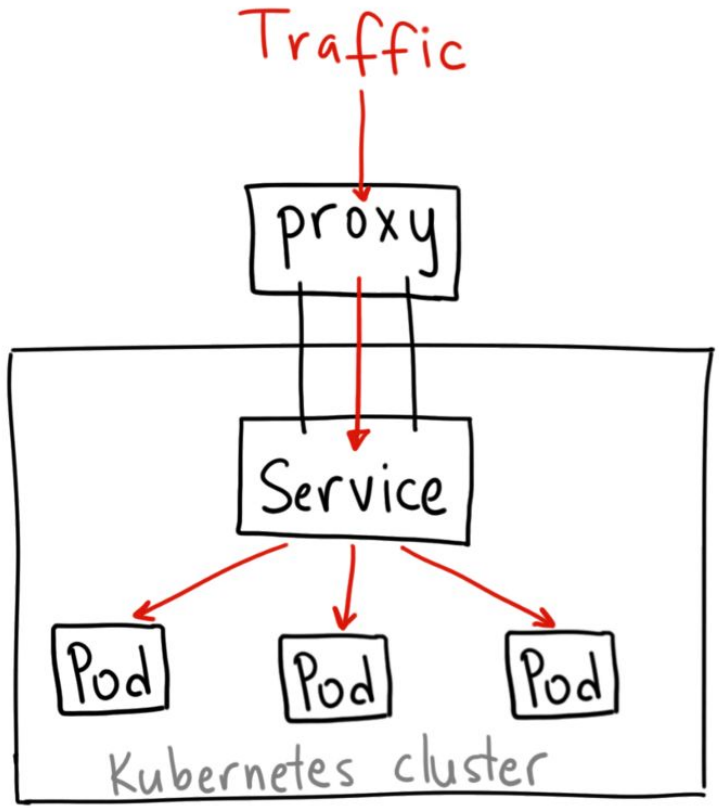
`port`
port number exposed internally in cluster

`targetPort`
port that containers are listening on

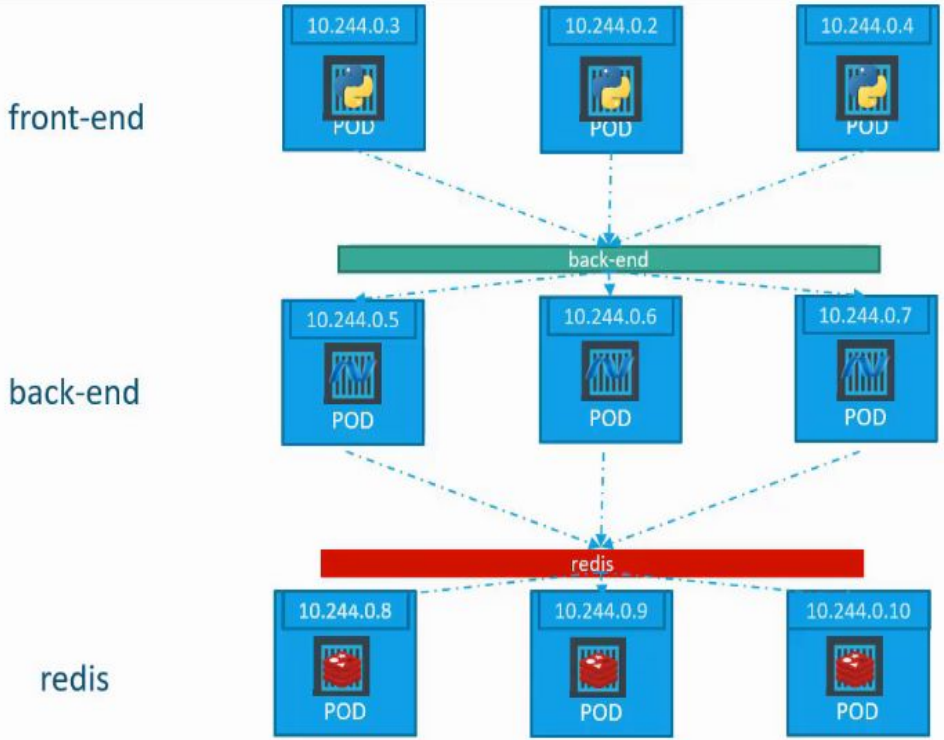
Service - ClusterIP

- A ClusterIP service is the default Kubernetes service. It gives you a service inside your cluster that other apps inside your cluster can access. There is no external access.

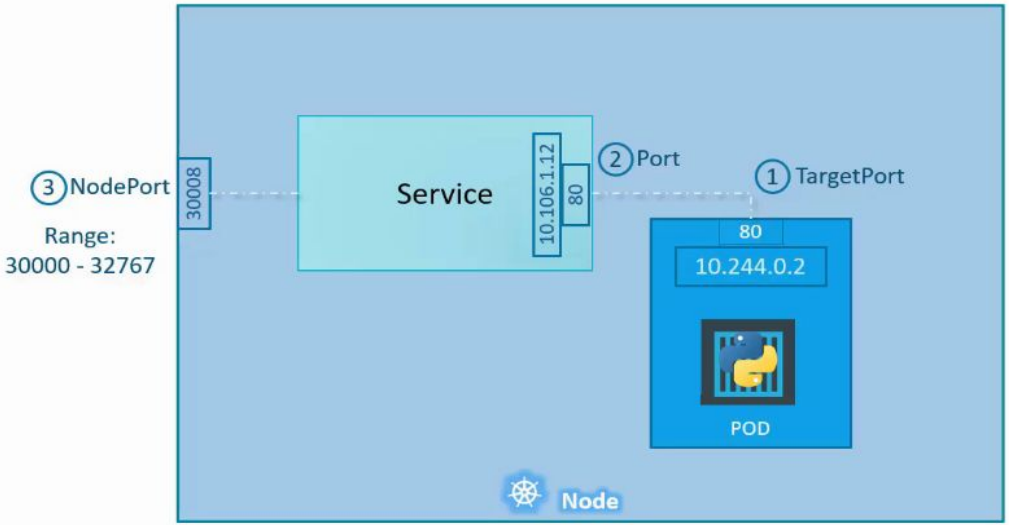
```
apiVersion: v1
kind: Service
metadata:
 name: my-internal-service
spec:
 selector:
   app: my-app
 type: ClusterIP
 ports:
 - name: http
   port: 80
   targetPort: 80
```



ClusterIP


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

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

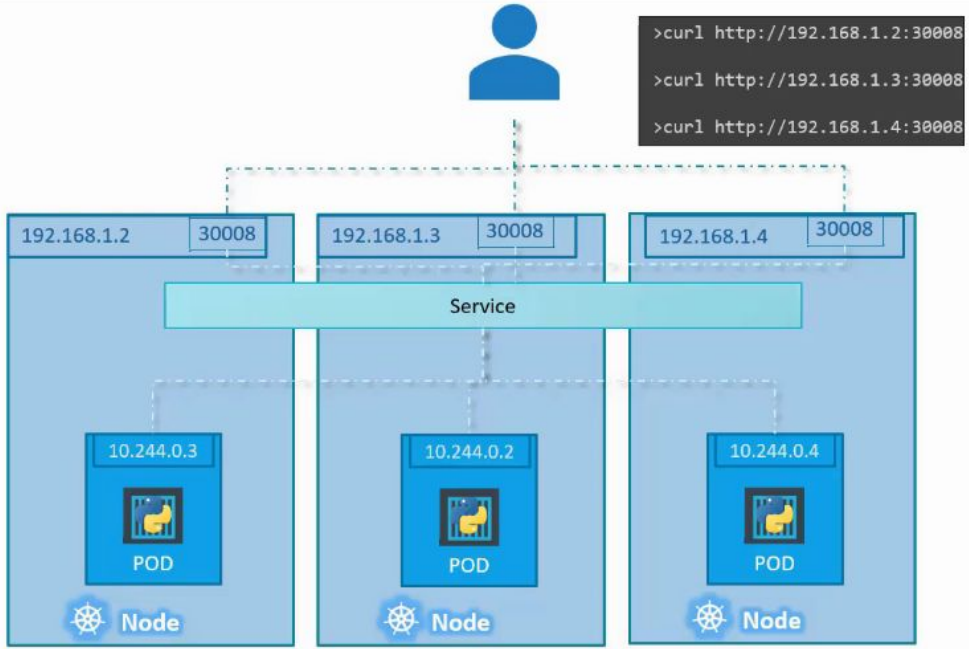
service-definition.yml

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

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

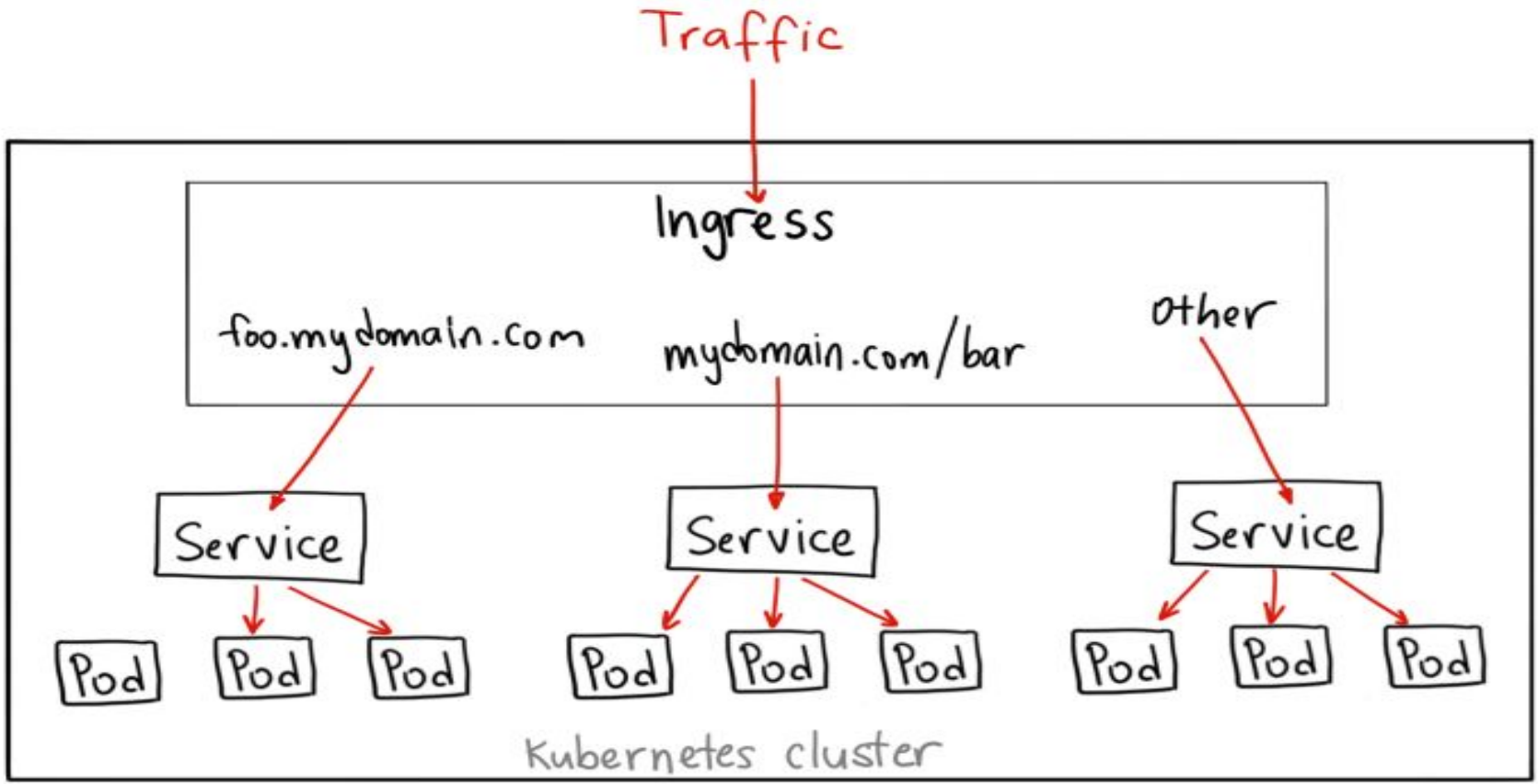
pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

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

Service - NodePort



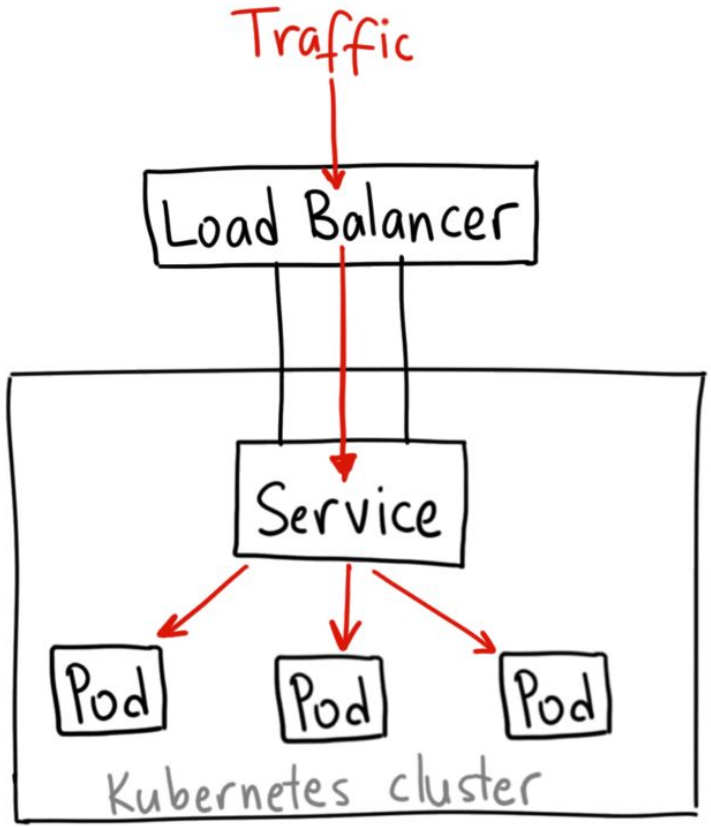
Service - Ingress



Service - Load Balancer

A LoadBalancer service is the standard way to expose a service to the internet

The big downside is that each service you expose with a LoadBalancer will get its own IP address, and you have to pay for a LoadBalancer per exposed service, which can get expensive!



Kubernetes - Deployment

- Deployments represent a set of multiple, identical Pods with no unique identities.

- A Deployment runs multiple replicas of your application and automatically replaces any instances that fail or become unresponsive

  The Kubernetes deployment object lets you:

  - Deploy a replica set or pod
  - Update pods and replica sets
  - Rollback to previous deployment versions
  - Scale a deployment
  - Pause or continue a deployment

Service - Ingress Yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: my-ingress
spec:
 backend:
   serviceName: other
   servicePort: 8080
 rules:
 - host: foo.mydomain.com
   http:
     paths:
     - backend:
         serviceName: foo
         servicePort: 8080
 - host: mydomain.com
   http:
     paths:
     - path: /bar/*
       backend:
         serviceName: bar
         servicePort: 8080
```

Kubernetes - Setup with kubeadm

- 3 Ubuntu 16.04 servers with 4GM RAM and private networking enabled
- On each of the three Ubuntu 16.04 servers run the following commands as root:

```
apt-get update && apt-get install -y apt-transport-https

curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -

cat <<EOF >/etc/apt/sources.list.d/kubernetes.list

deb http://apt.kubernetes.io/ kubernetes-xenial main

EOF

apt-get update

apt-get install -y kubelet=1.18.0-00 kubeadm=1.18.0-00 kubectl=1.18.0-00 docker.io
```

Kubernetes - Setup with kubeadm

Setup the Kubernetes Master

- kubeadm init
- mkdir -p $HOME/.kube
- sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
- sudo chown $(id -u):$(id -g) $HOME/.kube/config

Join your nodes to your Kubernetes cluster

- kubeadm join --token 702ff6.bc7aacff7aacab17 174.138.15.158:6443 --discovery-token-ca-cert-hash sha256:68bc22d2c631800fd358a6d7e3998e598deb2980ee613b3c2f1da8978960c8ab

Installing the Calico Add-On

- `curl https://docs.projectcalico.org/manifests/calico.yaml -O`
- kubectl apply -f calico.yaml