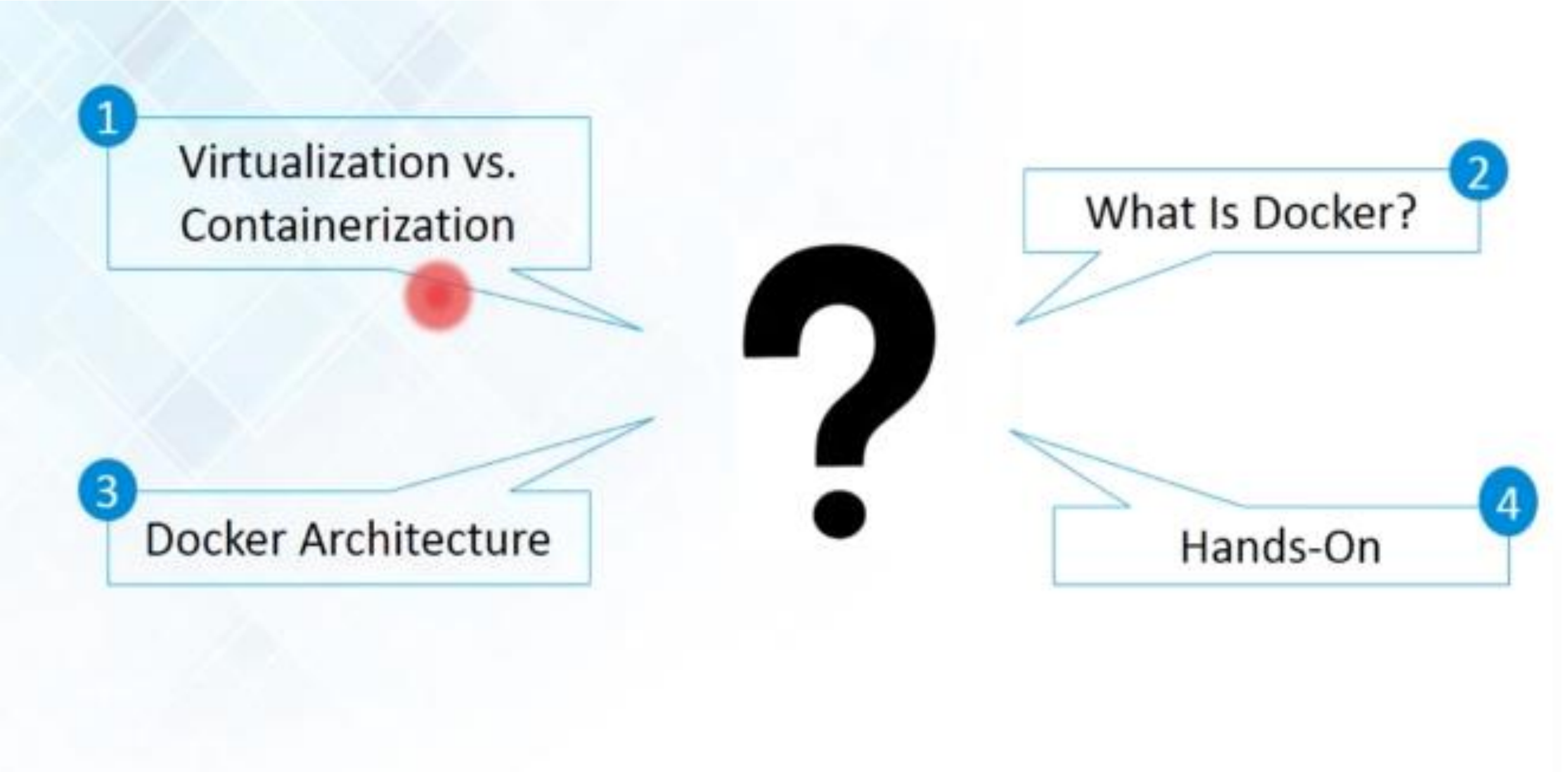
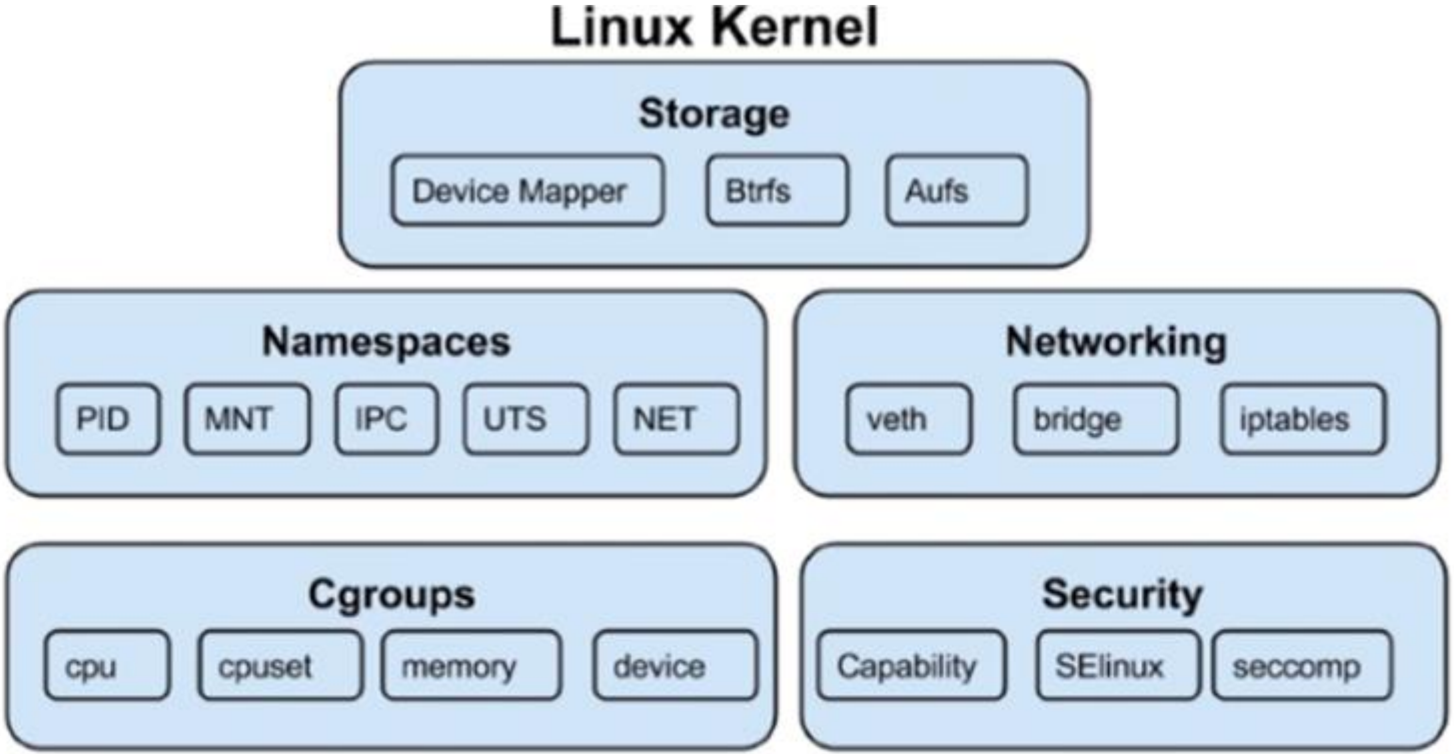
What is Docker



Docker Internals



Linux Kernel

Storage
- Device Mapper
- Btrfs
- Aufs

Namespaces
- PID
- MNT
- IPC
- UTS
- NET

Networking
- veth
- bridge
- iptables

Cgroups
- cpu
- cpuset
- memory
- device

Security
- Capability
- SElinux
- seccomp

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.
- 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.

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

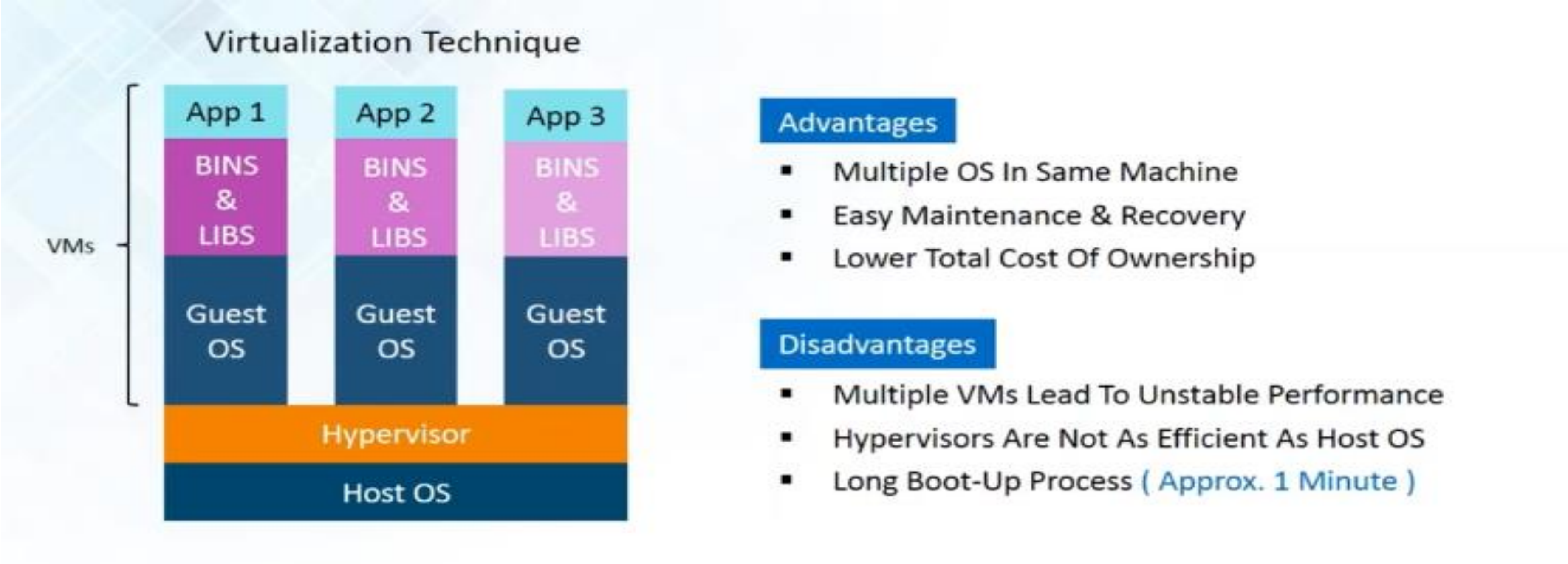
Docker Internals

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



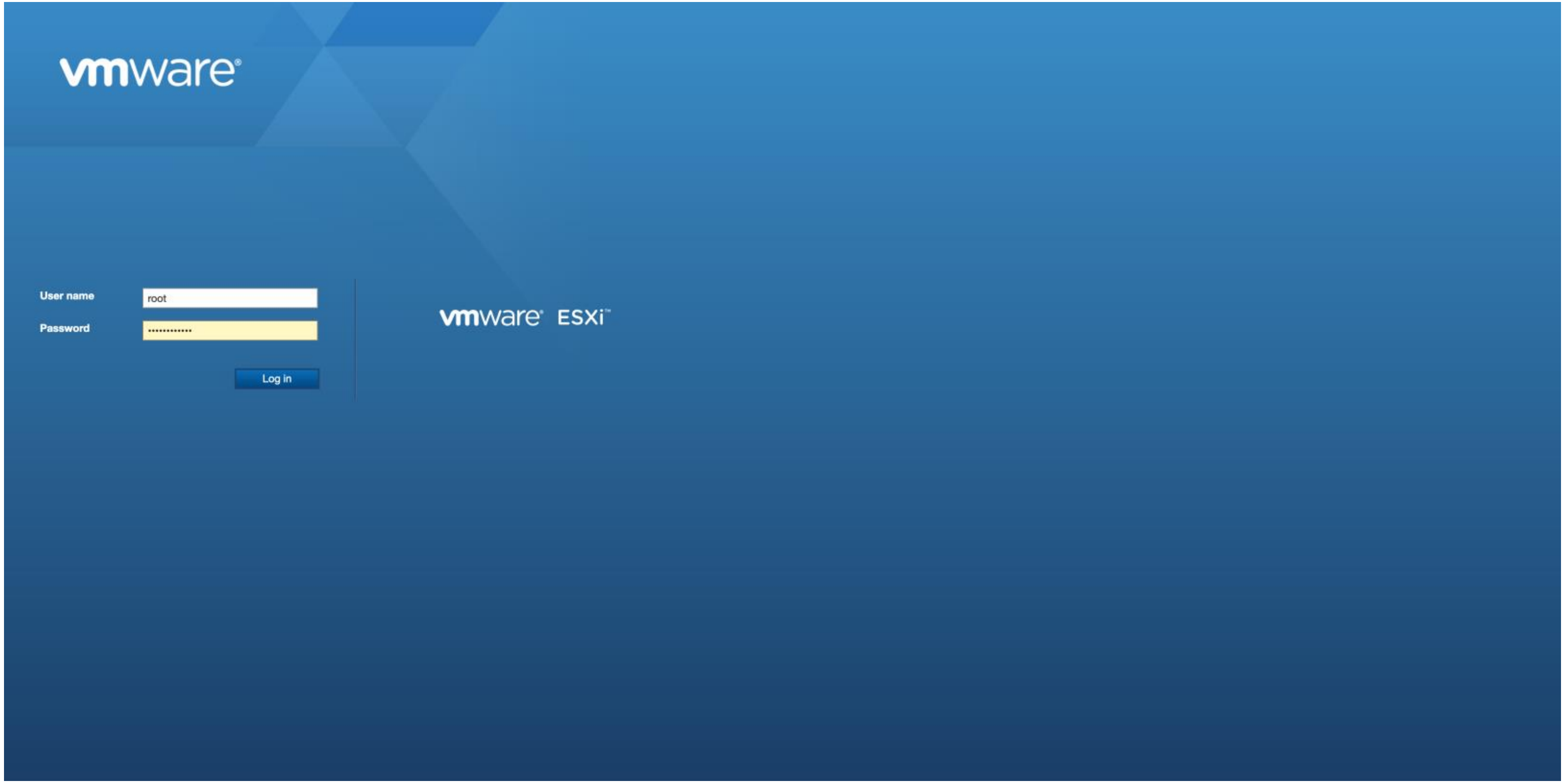
Advantages

- 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)

Virtualization

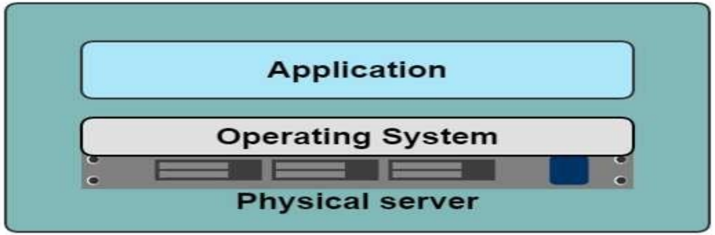


User name root

Password ············

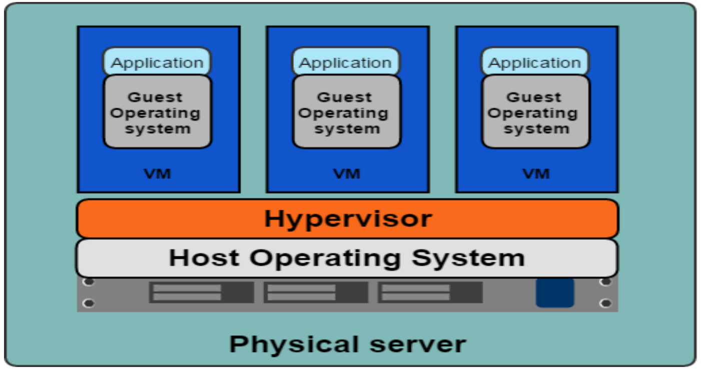
Virtualization

One application on one physical server



- Slow deployments
- Huge costs
- Wasted resources
- Difficult to scale, migrate
- Vendor lock in

Hypervisor Based Virtualization



- Better resource pooling

  One physical machine divided into multiple virtual machines

- Easier to scale
- VMs in the cloud

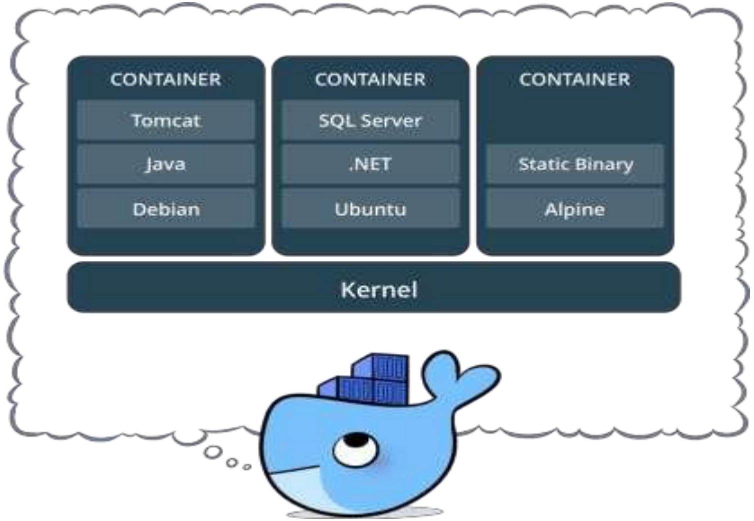
  Rapid elasticity

  Pay as you go model

Why Docker

- ➢ Each VM still requires

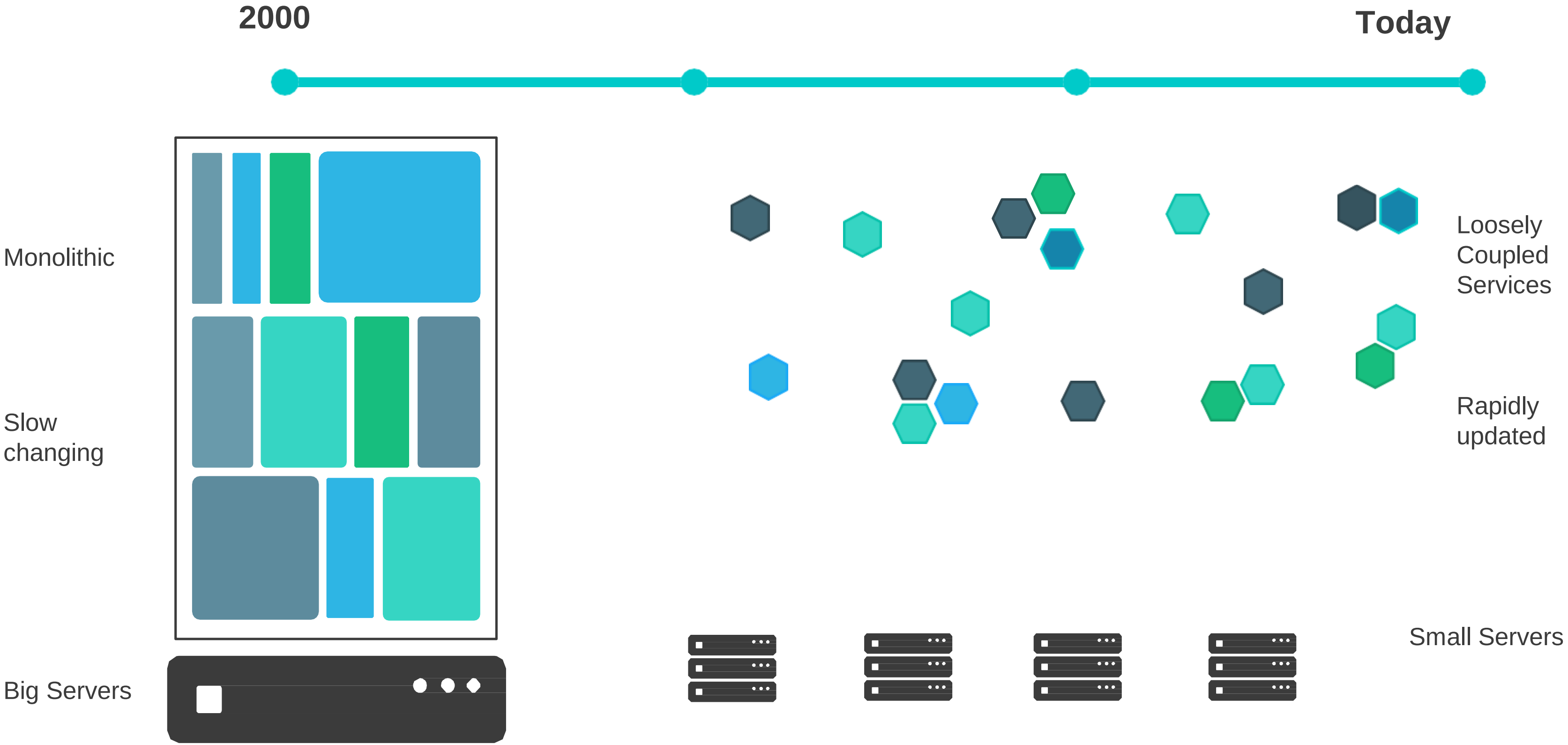
  CPU allocation, Storage, RAM, Guest OS
- ➢ The more VMs you run, the more resources you need
- ➢ Application portability not guaranteed



- ➢ Standardized packaging for software and dependencies
- ➢ Isolate apps from each other
- ➢ Share the same OS kernel
- ➢ Works with all major Linux and Windows Server

Transformation

Applications are transforming



2000

Today

Monolithic

Slow changing

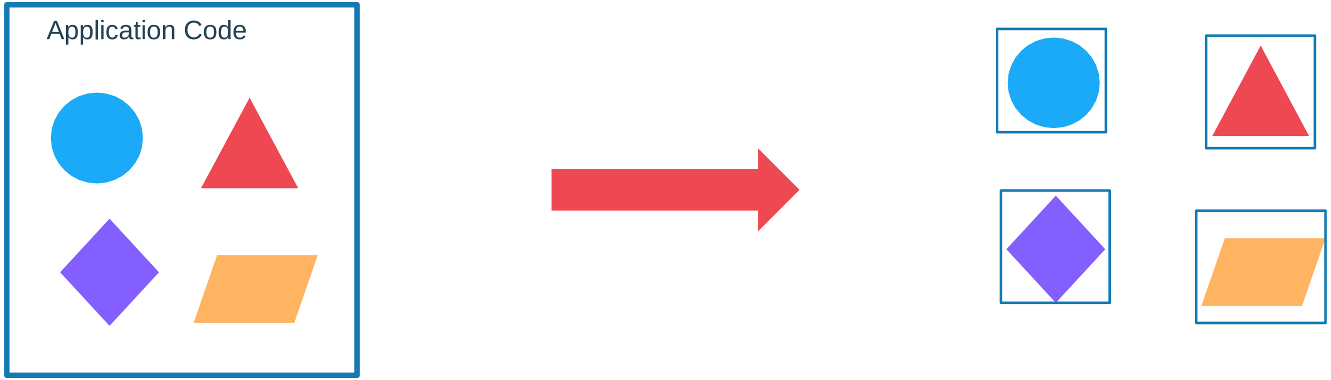
Big Servers

Loosely Coupled Services

Rapidly updated

Small Servers

Applications Modernisation



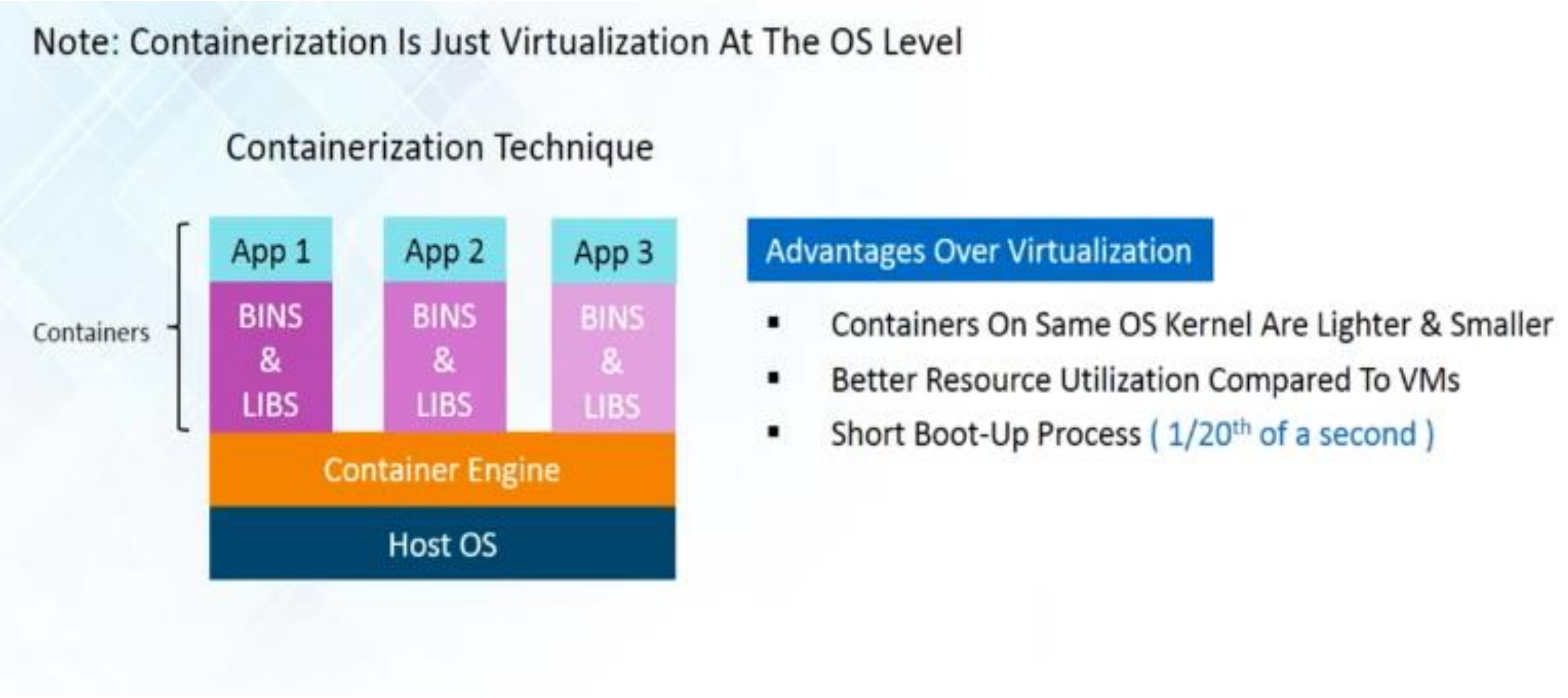
Developer Issues:

- Minor code changes require full re-compile and re-test
- Application becomes single point of failure
- Application is difficult to scale

Microservices: Break application into separate operations

Make the app independently scalable, stateless, highly available by design

Docker Info



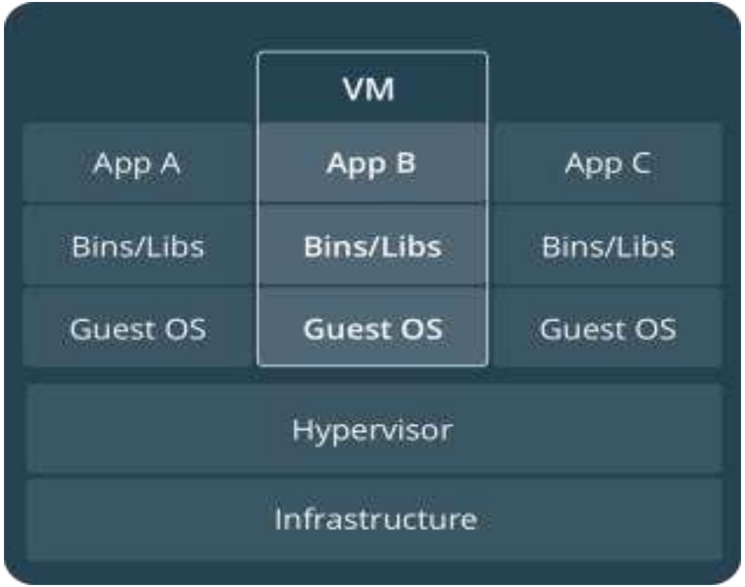
Note: Containerization Is Just Virtualization At The OS Level

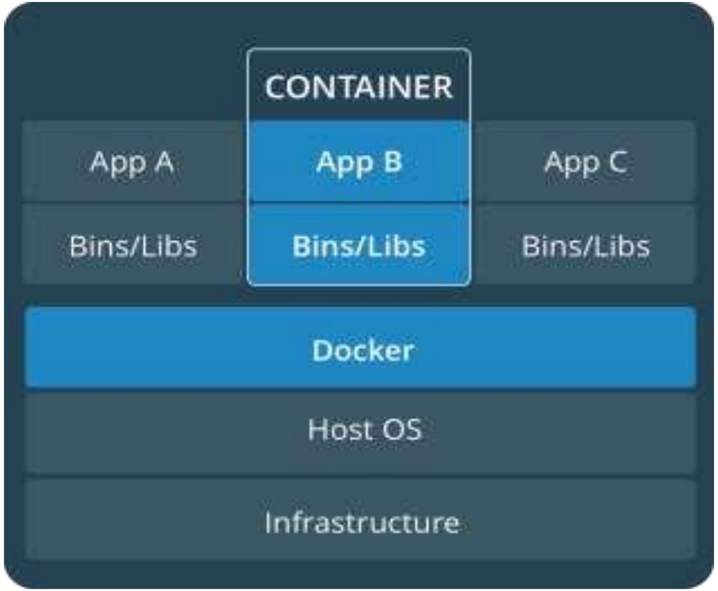
Containerization Technique

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)

Containers vs VMs

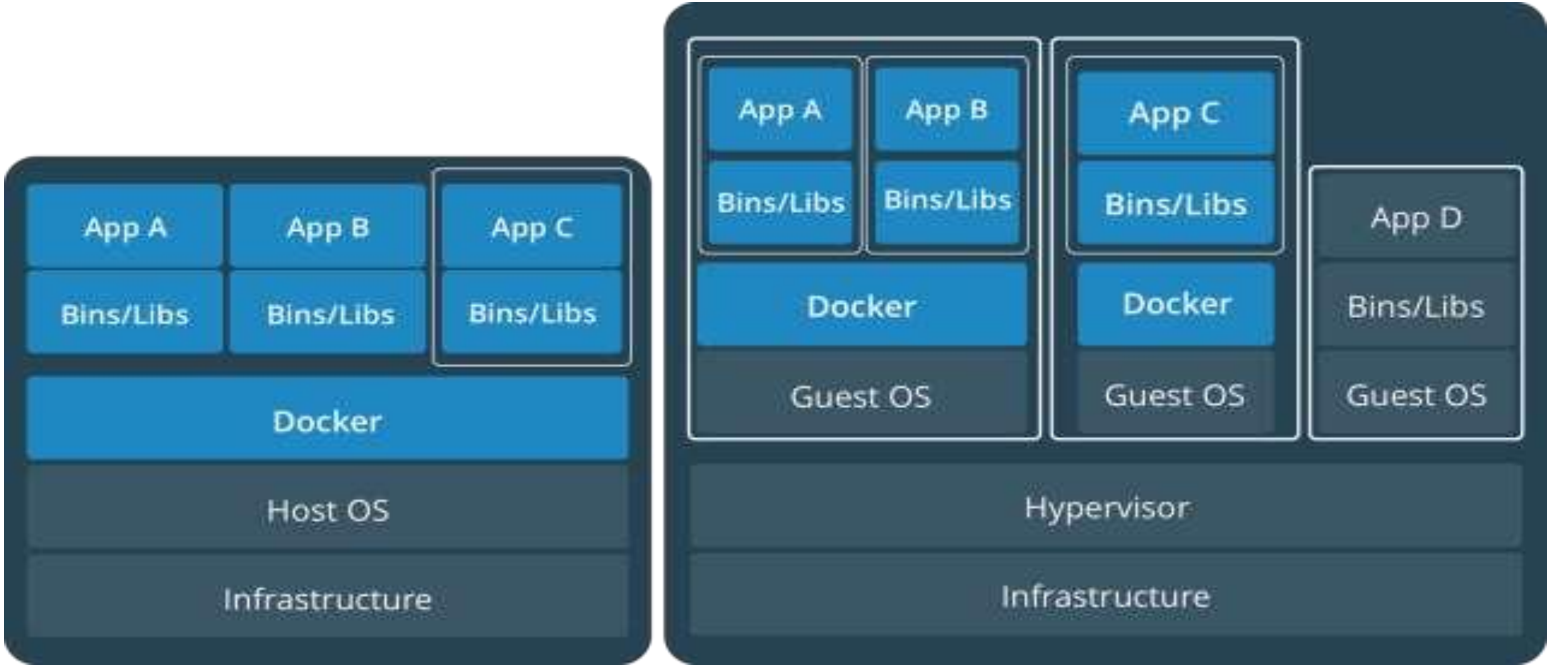




VMs are an infrastructure level construct to turn one machine into many servers

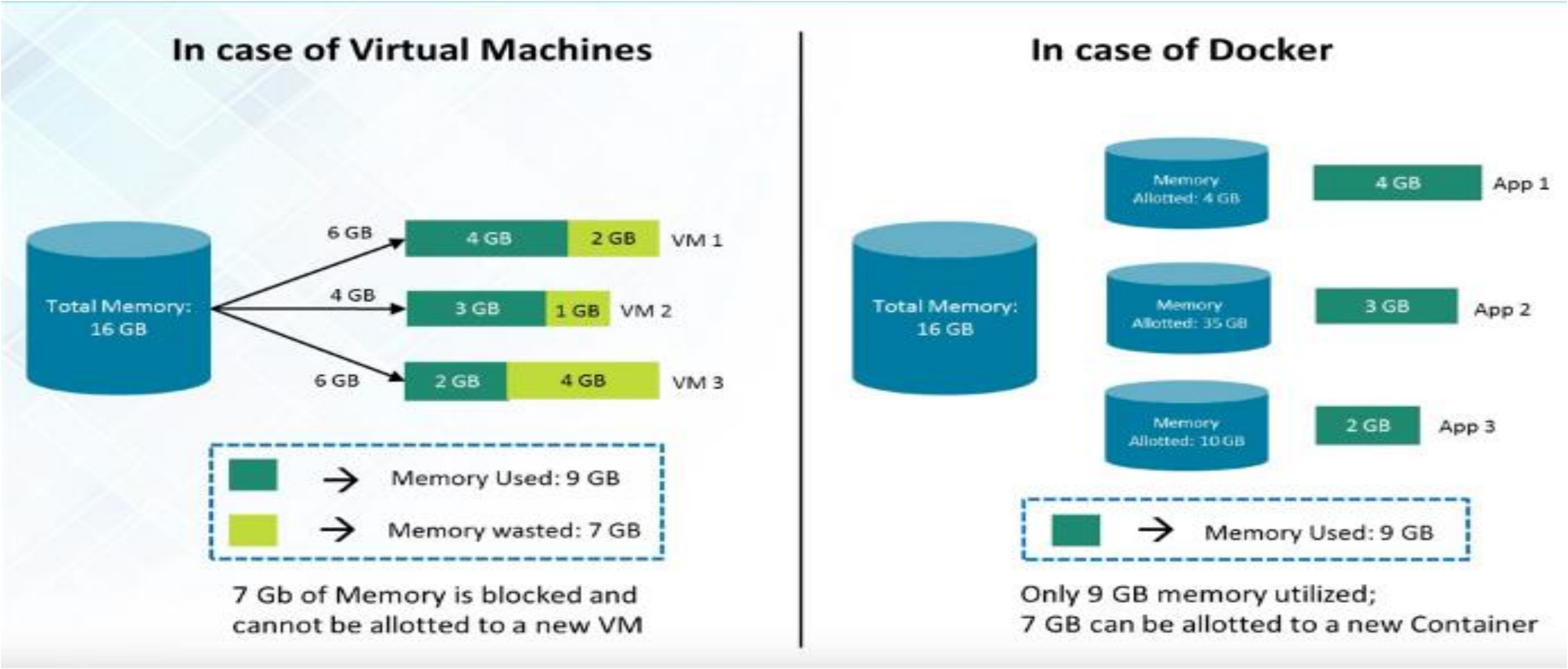
Containers are an app level construct

Containers vs VMs



Containers and VMs together provide a tremendous amount of flexibility for IT to optimally deploy and manage apps.

Docker Info



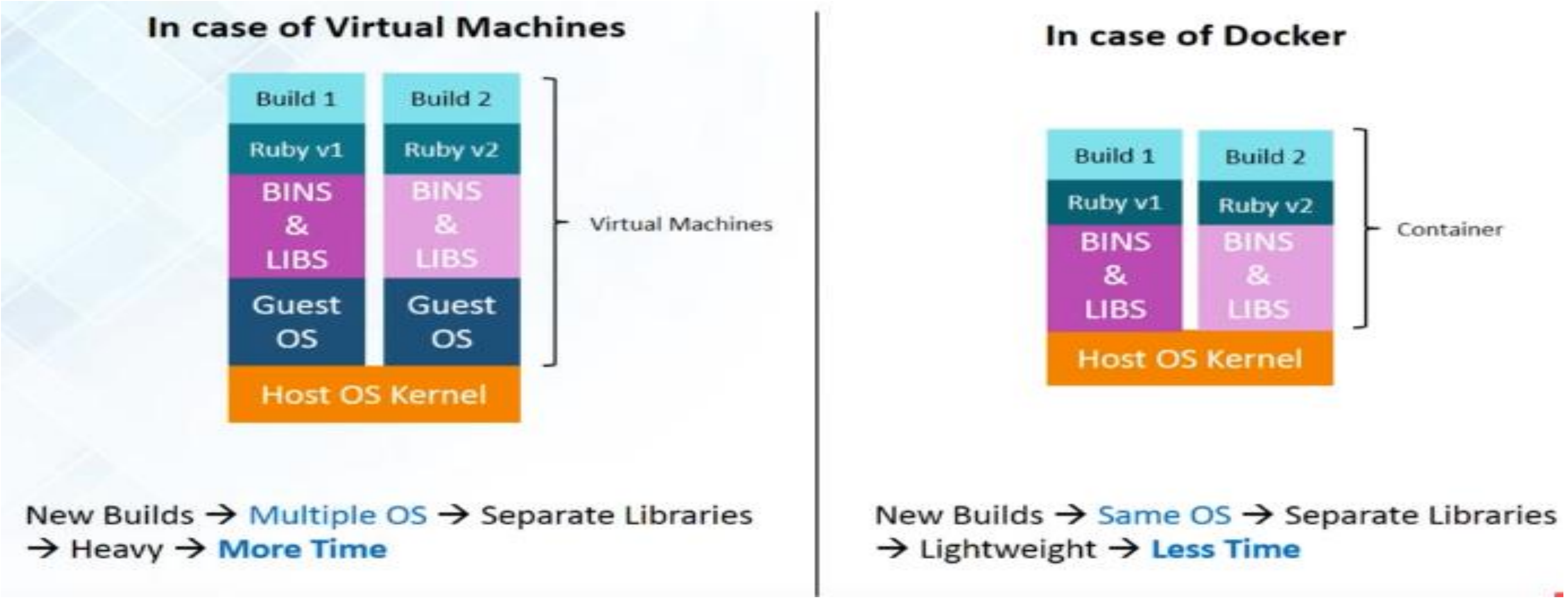
In case of Virtual Machines

7 Gb of Memory is blocked and cannot be allotted to a new VM

In case of Docker

Only 9 GB memory utilized;
7 GB can be allotted to a new Container

Docker Info



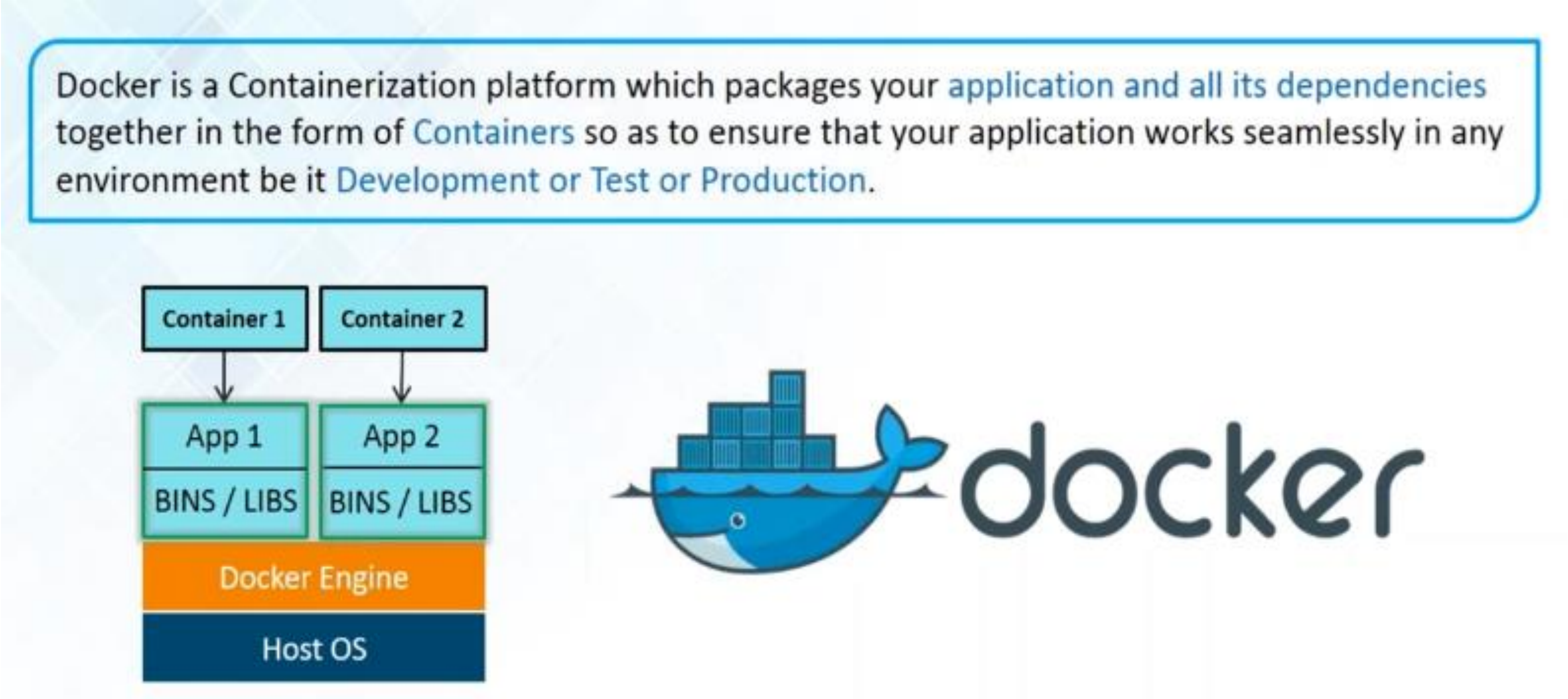
In case of Virtual Machines

New Builds → Multiple OS → Separate Libraries → Heavy → **More Time**

In case of Docker

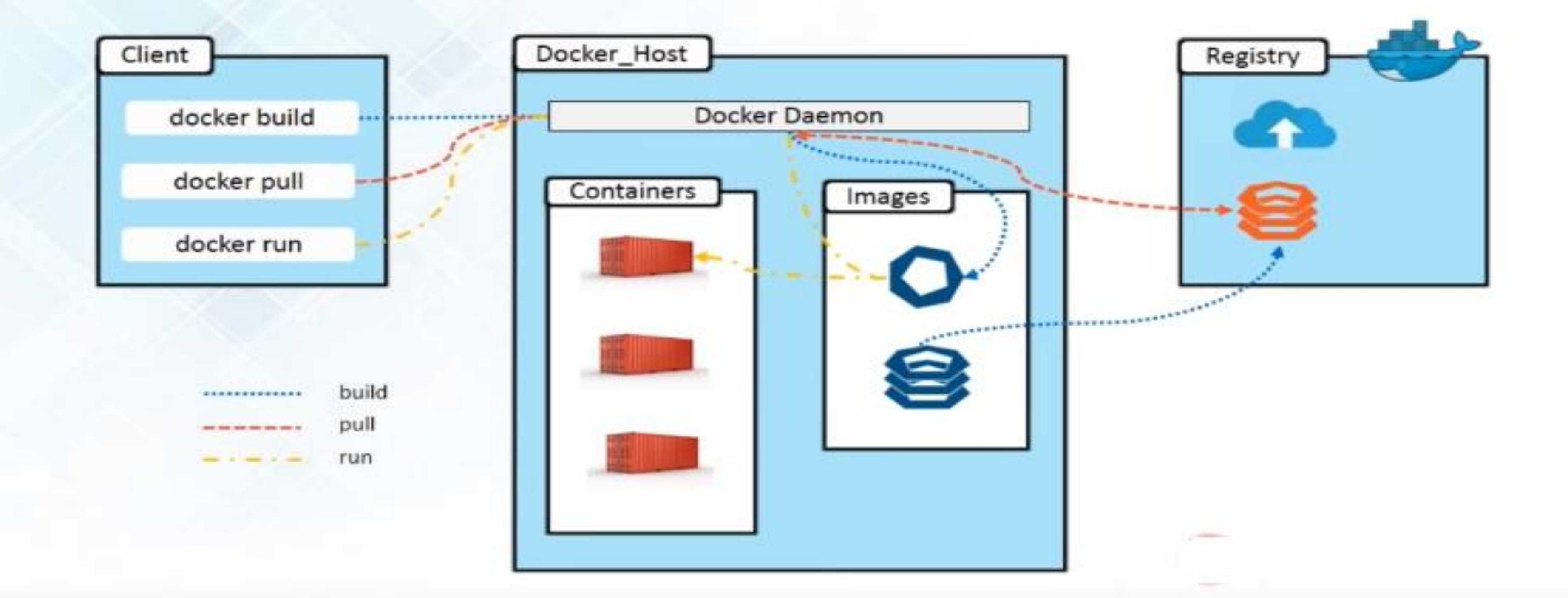
New Builds → Same OS → Separate Libraries → Lightweight → **Less Time**

Docker Information

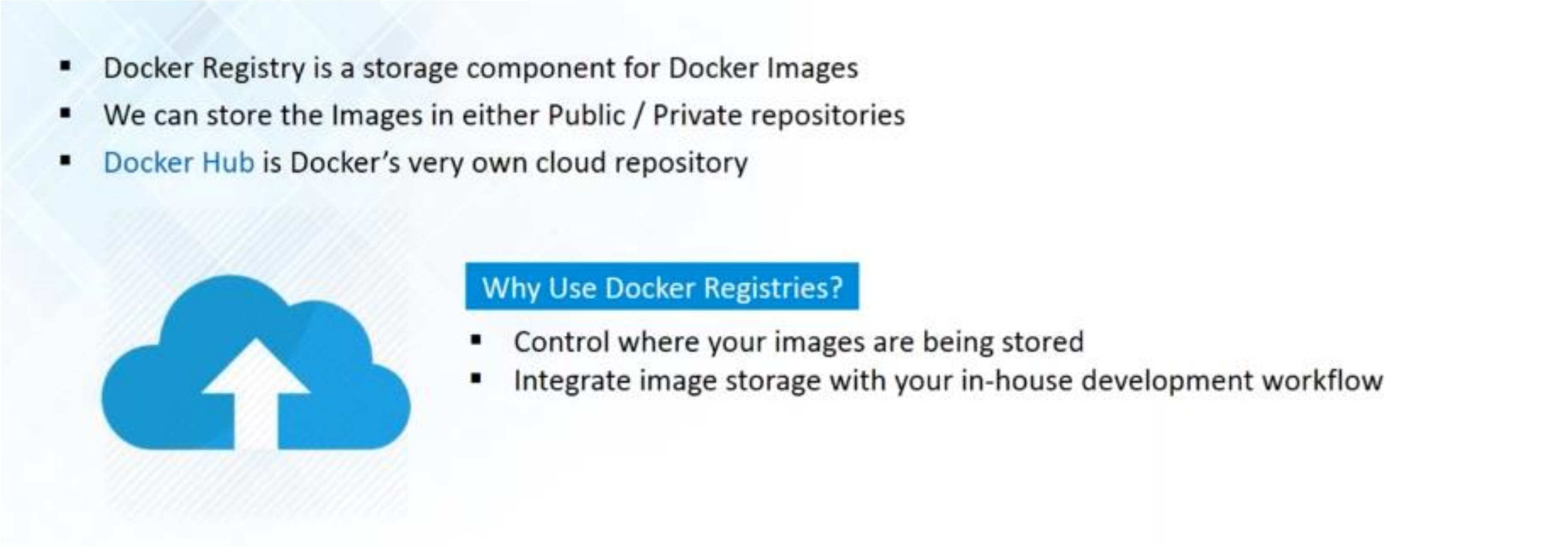


Docker is a Containerization platform which packages your application and all its dependencies together in the form of Containers so as to ensure that your application works seamlessly in any environment be it Development or Test or Production.

Docker Architecture



Docker Registry

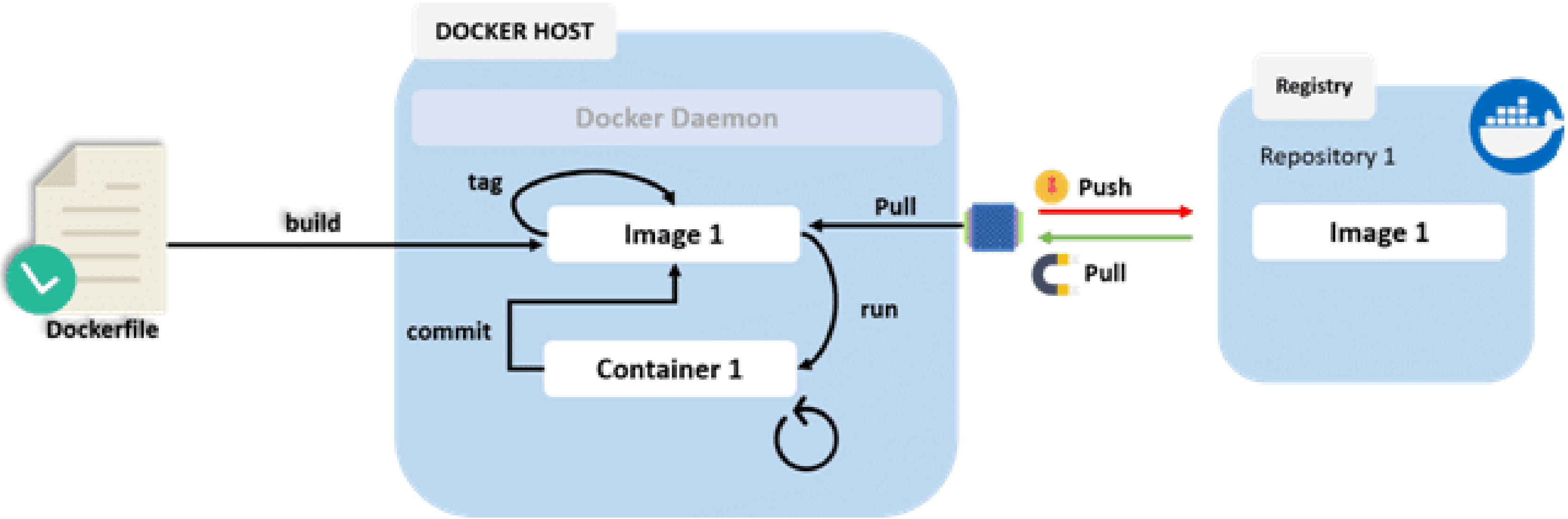


- Docker Registry is a storage component for Docker Images
- We can store the Images in either Public / Private repositories
- Docker Hub is Docker's very own cloud repository

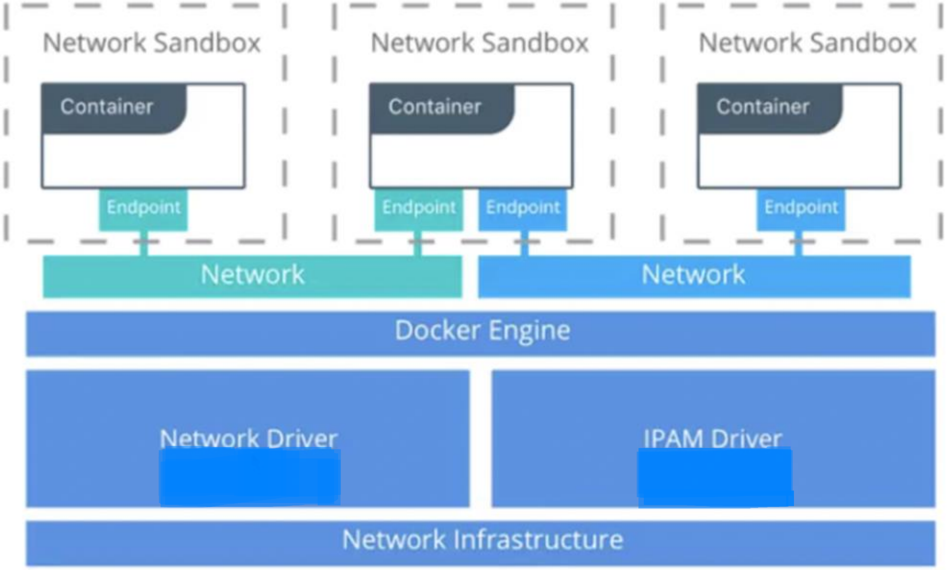
Why Use Docker Registries?

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

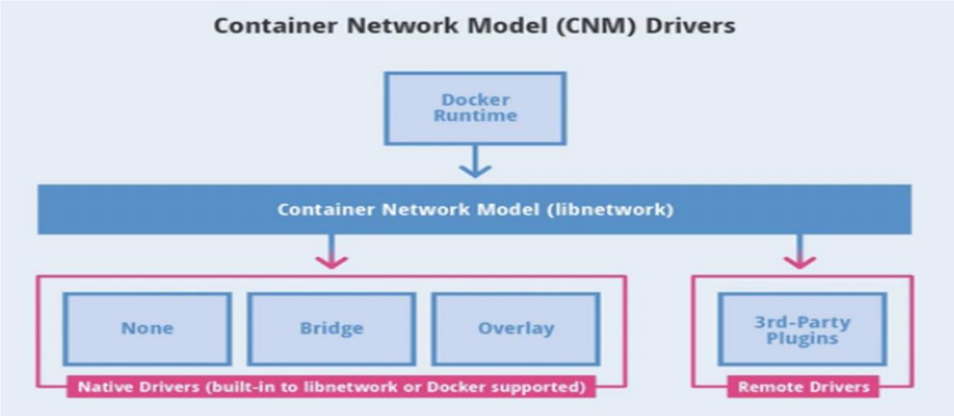
Typical Workflow



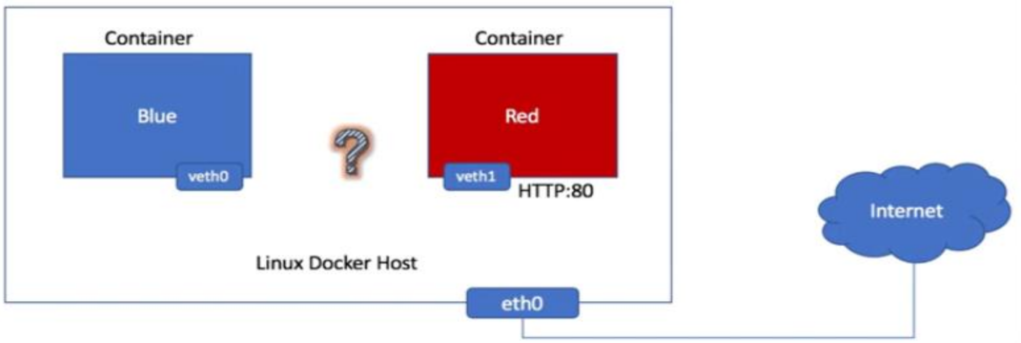
The Container Network Model



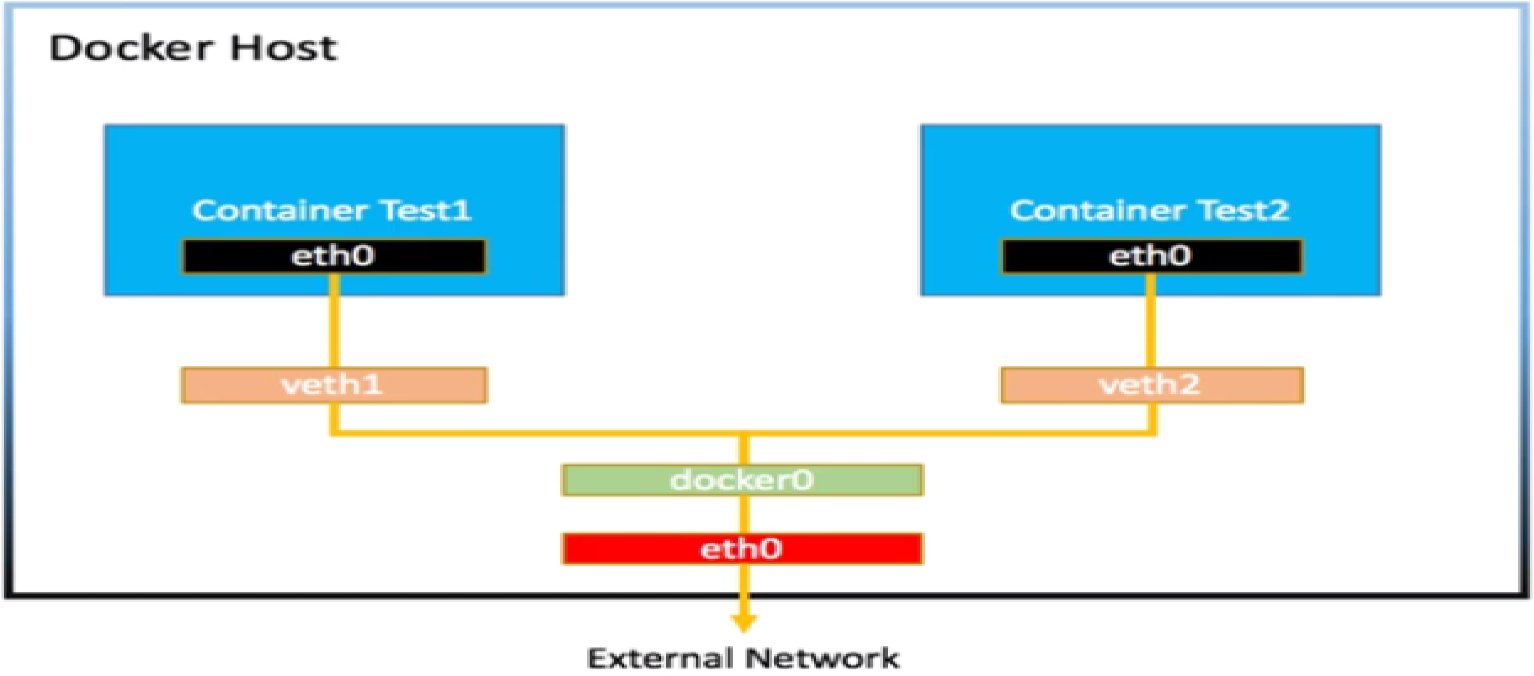
Docker Info


Container Network Model (CNM) Drivers
Docker Runtime
Container Network Model (libnetwork)
None
Bridge
Overlay
Native Drivers (built-in to libnetwork or Docker supported)
3rd-Party Plugins
Remote Drivers

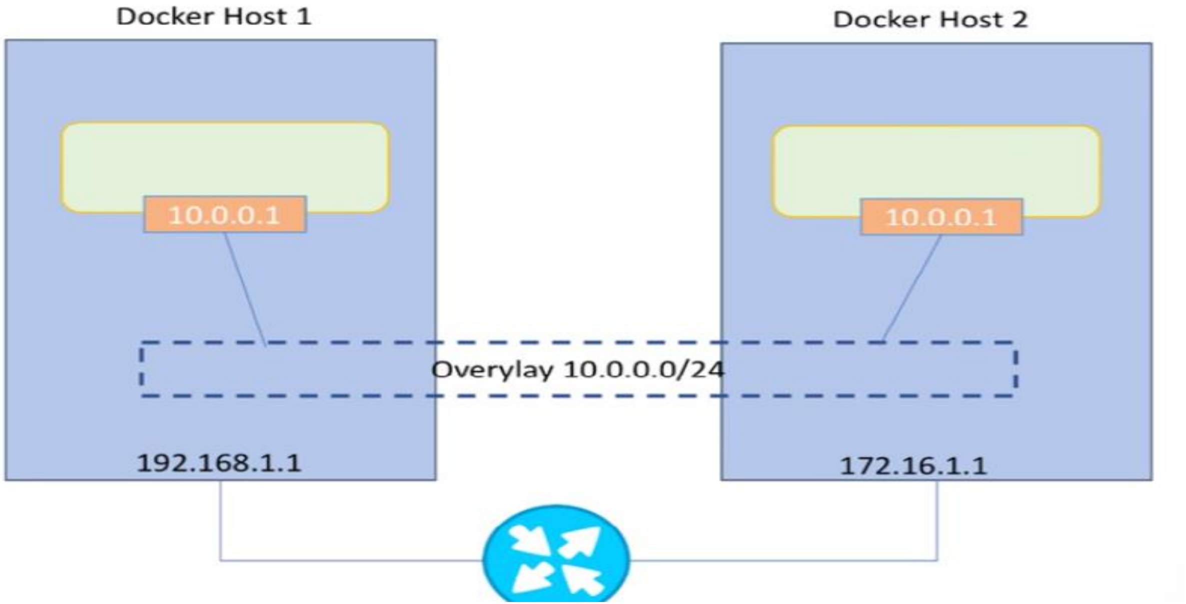
- → How two different containers in the same host communicate with each other?
- → How the container communicate with the outside of Linux host (Internet)?
- → How access container from the local docker host and outside of the docker host?



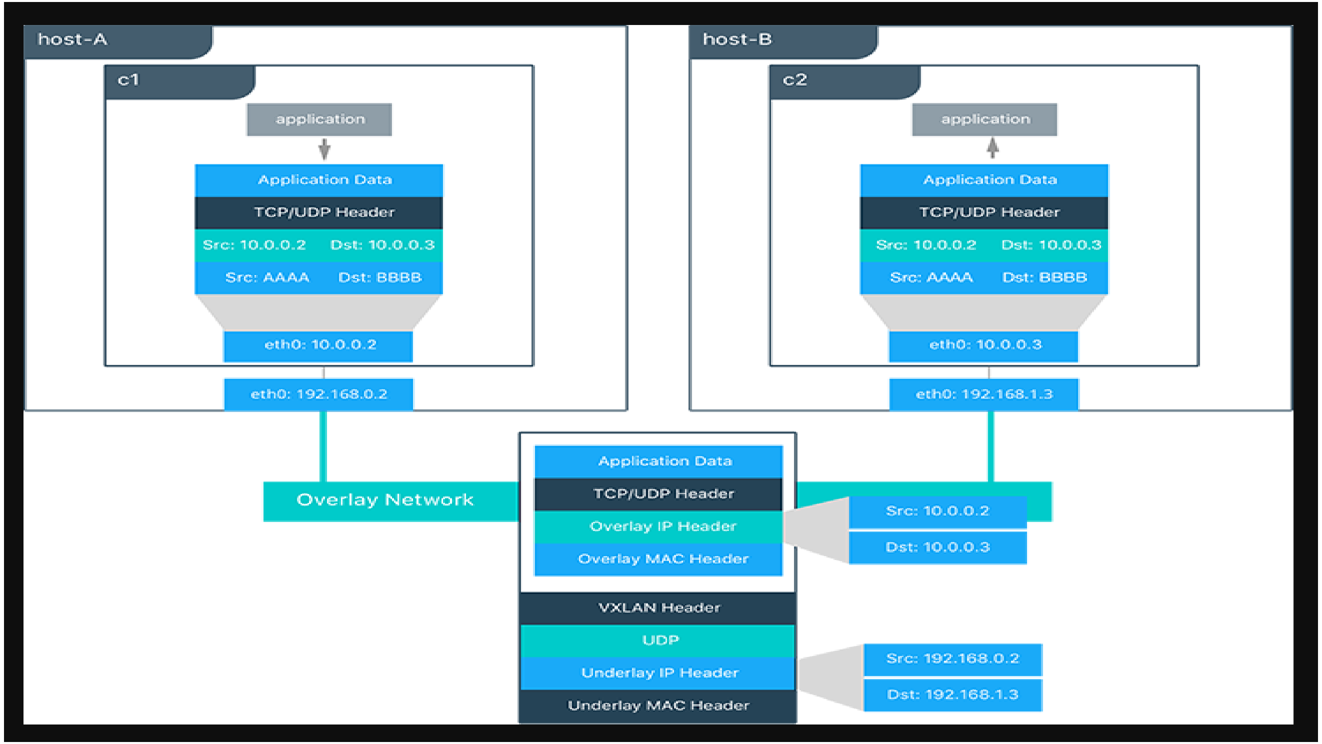
Docker Info


Docker Host
Container Test1
eth0
Container Test2
eth0
veth1
veth2
docker0
eth0
External Network

Multi host Networking

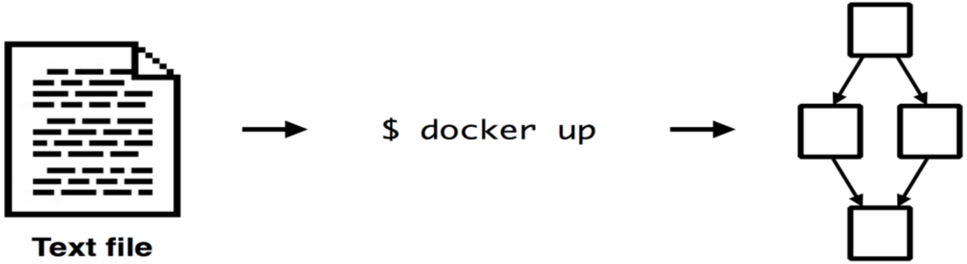


Multi host Networking



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



- 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

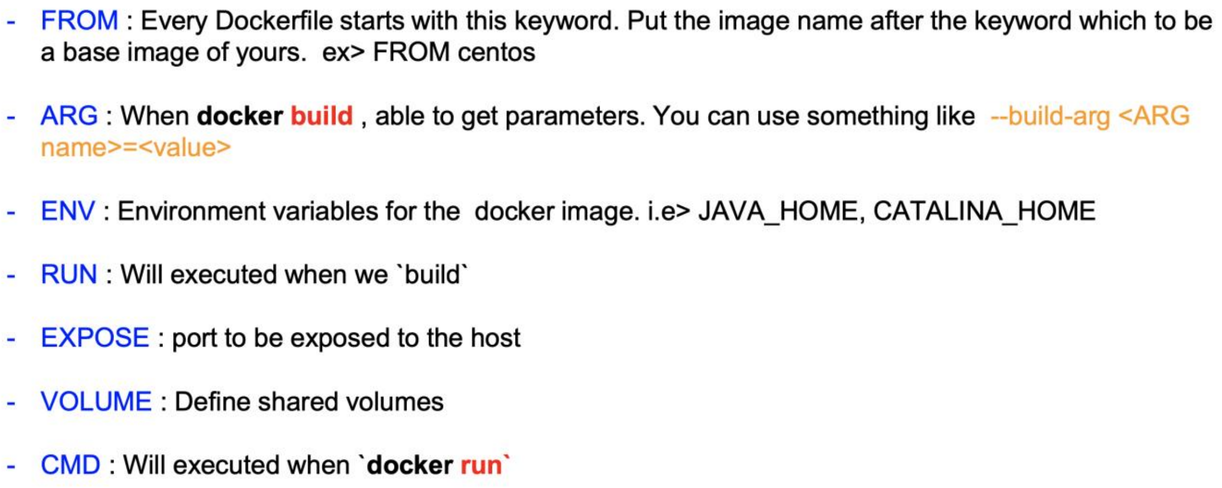
```
from flask import Flask
from redis import Redis
import os
import socket

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)
```

Dockerfile

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

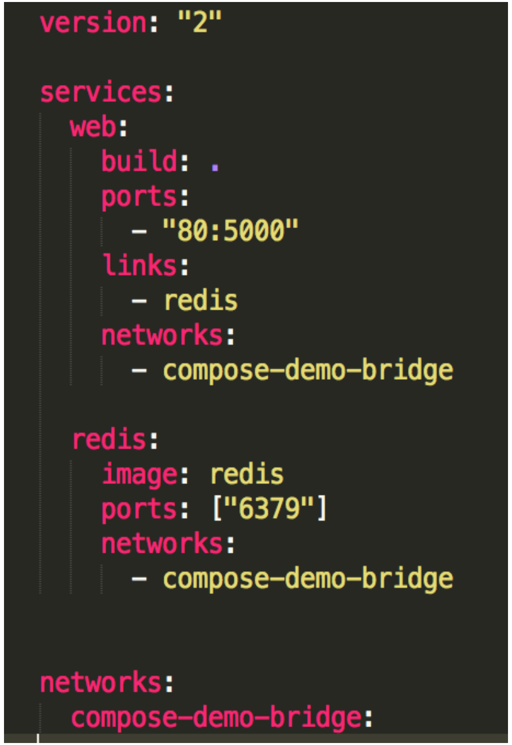
requirements.txt

```
flask
redis
```

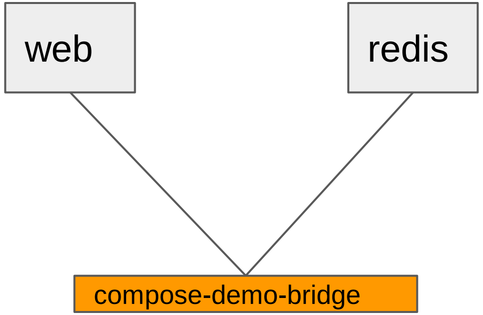


- FROM : Every Dockerfile starts with this keyword. Put the image name after the keyword which to be a base image of yours. ex> FROM centos
- ARG : When **docker build** , able to get parameters. You can use something like --build-arg <ARG name>=<value>
- ENV : Environment variables for the docker image. i.e> JAVA_HOME, CATALINA_HOME
- RUN : Will executed when we `build`
- EXPOSE : port to be exposed to the host
- VOLUME : Define shared volumes
- CMD : Will executed when `**docker run**`

Docker Info



```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

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?

Solutions

Kubernetes

Docker Swarm

Amazon EKS

AKS

GKE

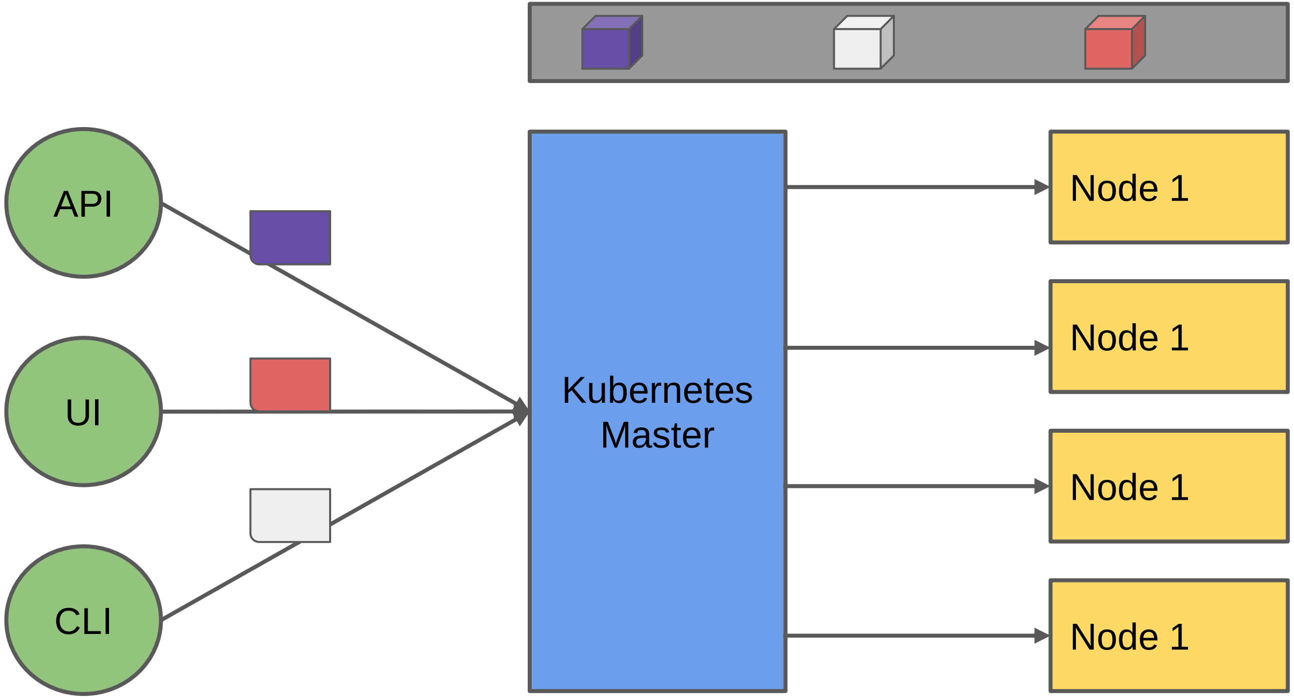
Kubernetes Overview

- 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

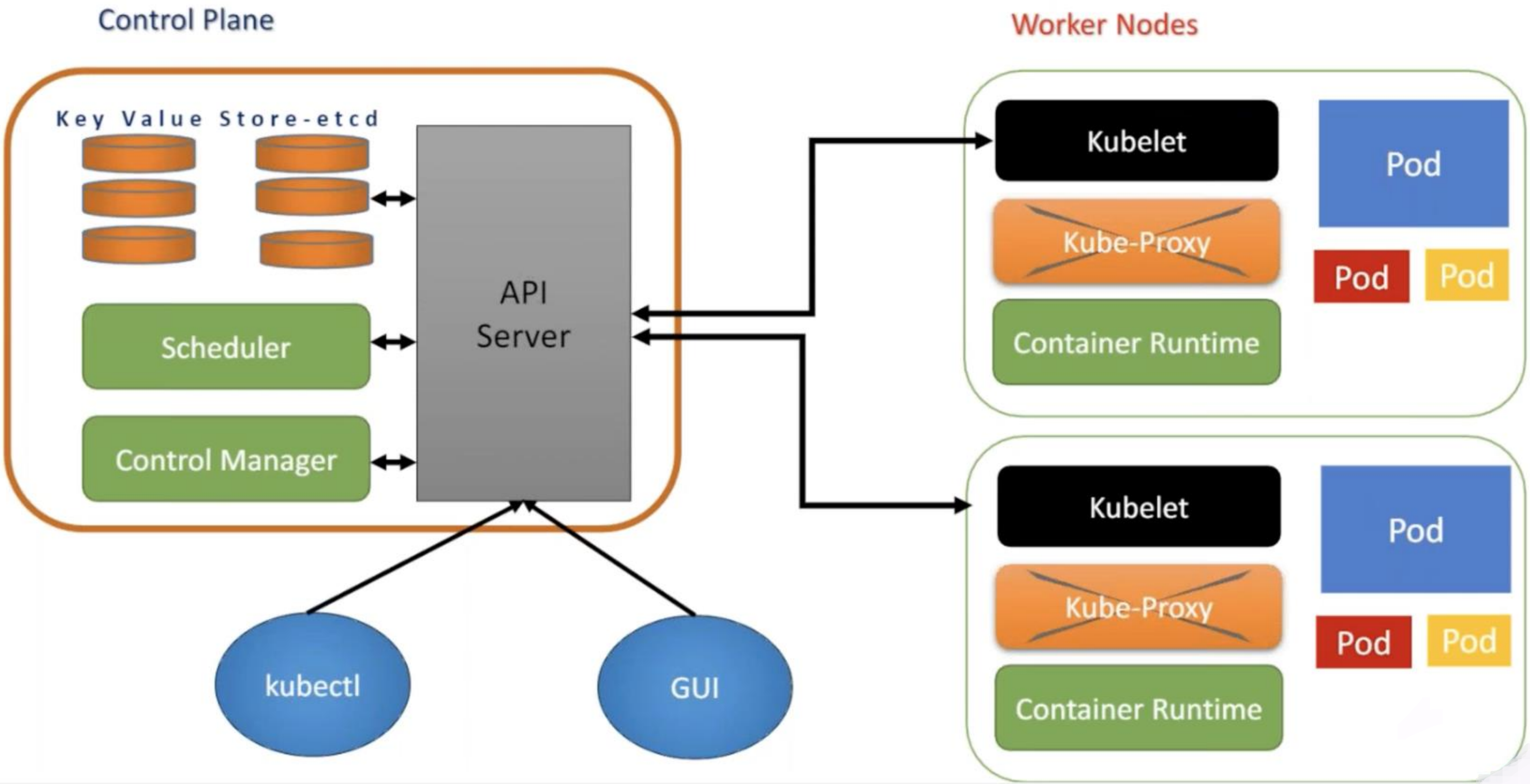
Kubernetes – Basic Terms

- Kubernetes -- the whole orchestration system. K8s in short
- Kubectl -- CLI to configure Kubernetes and manage apps. Kube control
- Node -- Single server in the kubernetes cluster
- Kubelet -- Kubernetes agent running on nodes
- Control plane -- Set of containers that manage the cluster
    - Includes API server, scheduler, controller manager, etcd, etc
    - Also called master

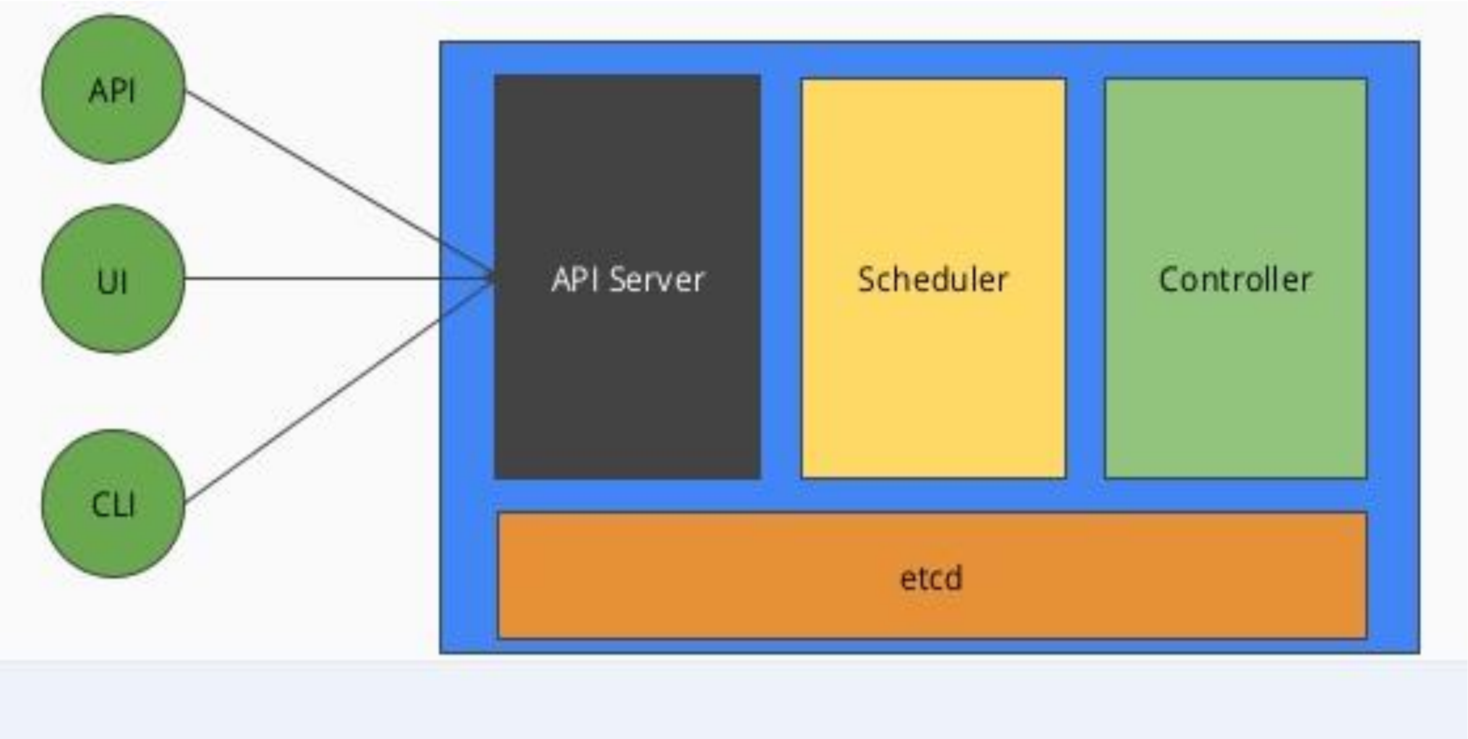
Kubernetes Architecture



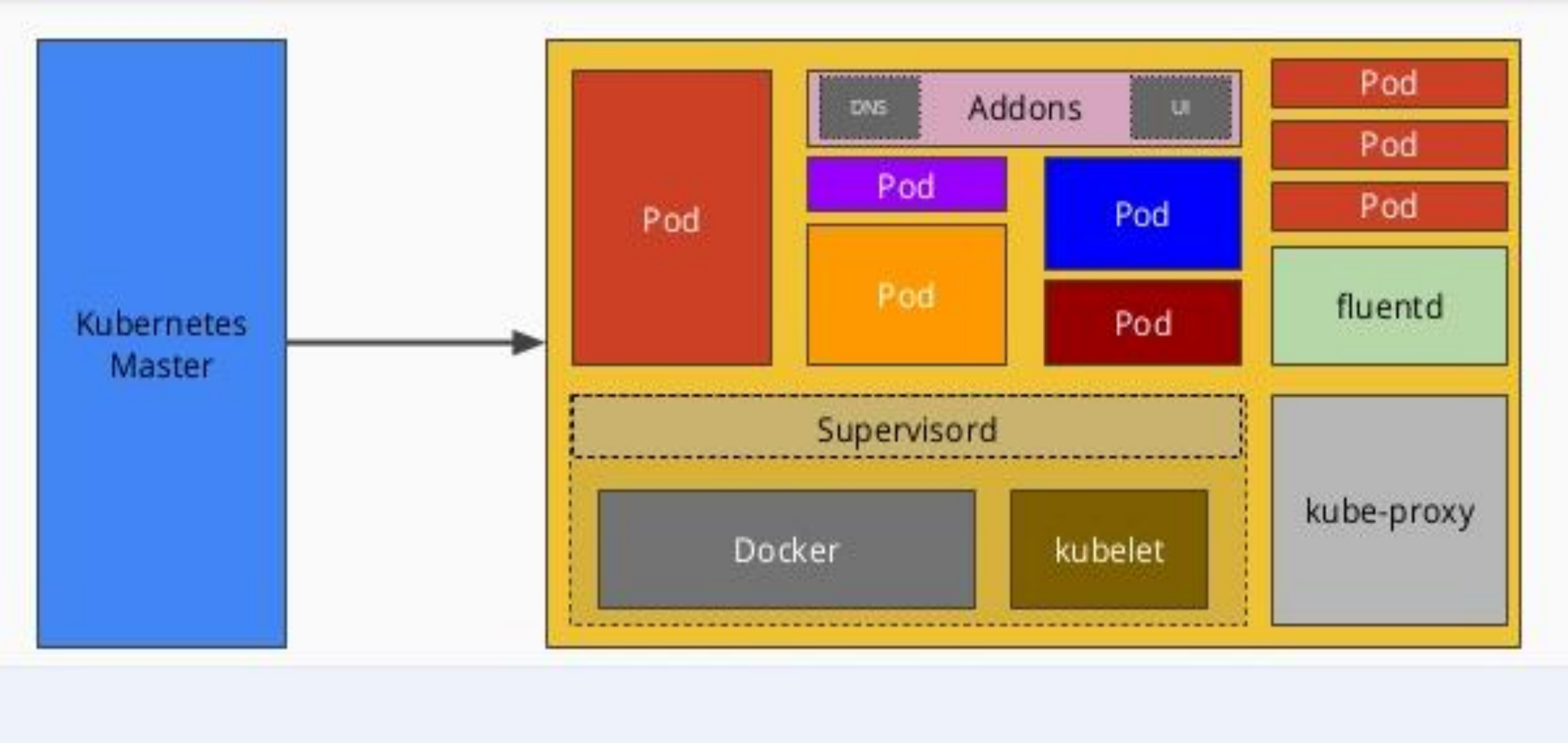
Kubernetes Architecture



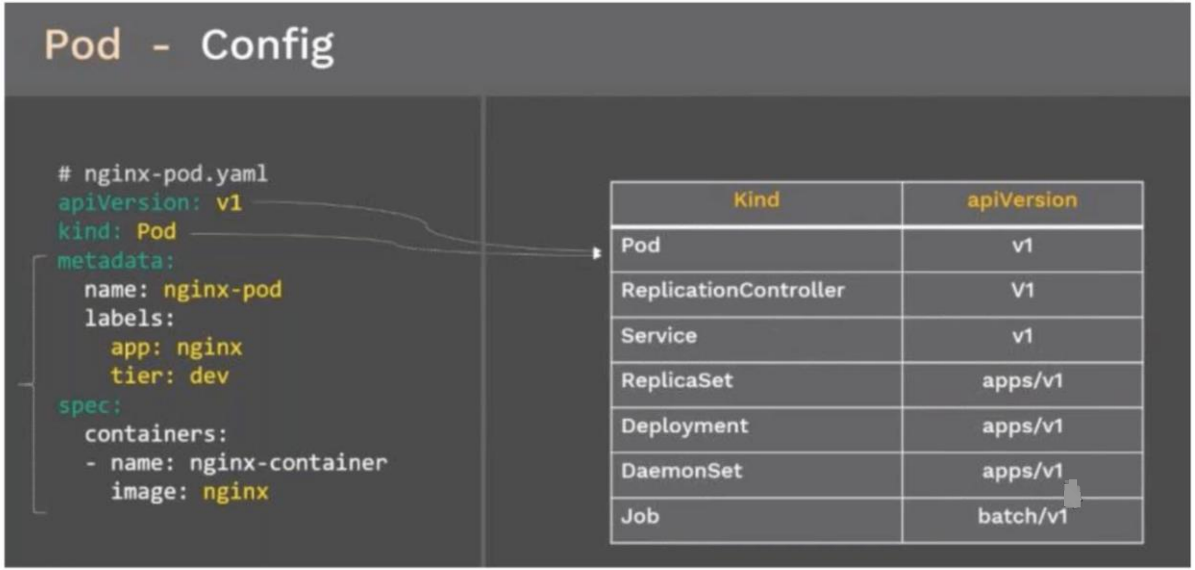
Kubernetes Architecture – Master



Kubernetes Architecture – Node



POD



Pod - Config

```yaml
# nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
    tier: dev
spec:
  containers:
  - name: nginx-container
    image: nginx
```

Kind	apiVersion
Pod	v1
ReplicationController	V1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1
DaemonSet	apps/v1
Job	batch/v1

DEMO:-Pod Creation

Kubernetes Basics

- **Nodes**: Hosts that run Kubernetes applications
- **Cluster**: Pool of compute, storage and network resources
- **Containers**: Units of packaging
- **Labels/Selector**: Key-Value pairs for identification
- **Pods**: Units of deployment
- **Deployment**: Auto Pod management
- **Replication Controller**: Ensures availability and scalability
- **ReplicaSet**: Advanced Replication controller with help of regular expression
- **Services**: Collection of pods exposed as an endpoint
- **Volumes**: To store data permanently
- **Secrets/ConfigMap**: To pass secrets and config files
- **DaemonSet**: To run similar task pod on each Node

Kubernetes Services

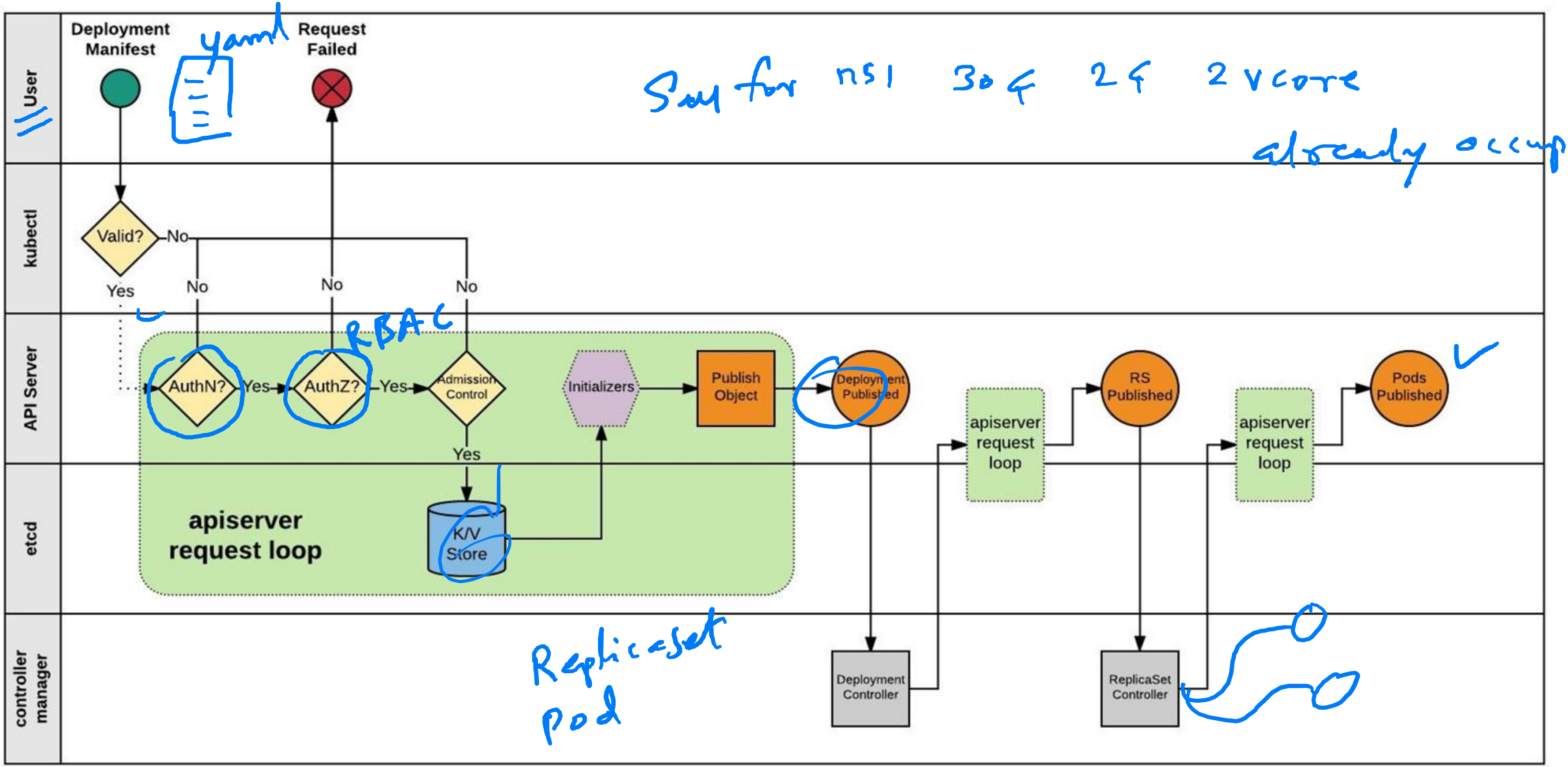
- Exposing PODs to the outside world. Services are required for discovery.
- Labels and selectors are used to route to the appropriate POD
- Service Types
    - ClusterIP - Services makes internal pod accessible
    - NodePort - Service does default load balancing
    - LoadBalancer - Service does load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

Ingress - Path based load balancing with service type LoadBalancer and NodePort

Kubernetes – Helm Charts

- Makes deployment easy. POD management, configuration etc, separately managed than manifest
- Package manager for kubernetes
- Single command for installing, upgrading and deleting releases.

Workflow



yaml

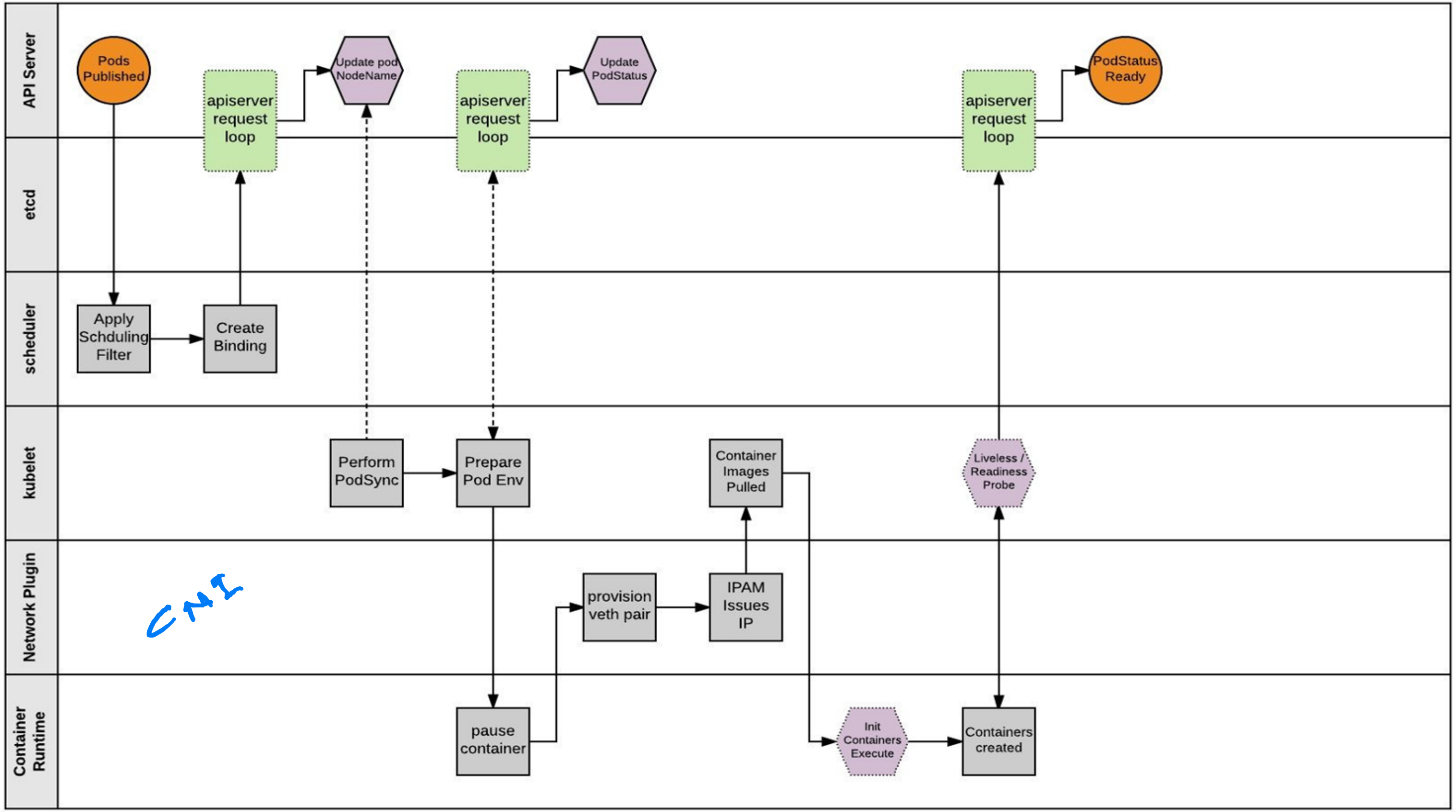
RBAC

Say for ns1 30G 2G 2 vcore

already occupi

Replicaset
pod

h


API Server
etcd
scheduler
kubelet
Network Plugin
Container Runtime
Pods Published
apiserver request loop
Update pod NodeName
apiserver request loop
Update PodStatus
apiserver request loop
PodStatus Ready
Apply Schduling Filter
Create Binding
Perform PodSync
Prepare Pod Env
Container Images Pulled
Liveless / Readiness Probe
CNI
provision veth pair
IPAM Issues IP
pause container
Init Containers Execute
Containers created

Multinode cluster setup

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



apt-get update

apt-get install -y docker.io

apt-get install -y kubelet kubeadm kubectl kubernetes-cni
```

Multinode 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
```

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.

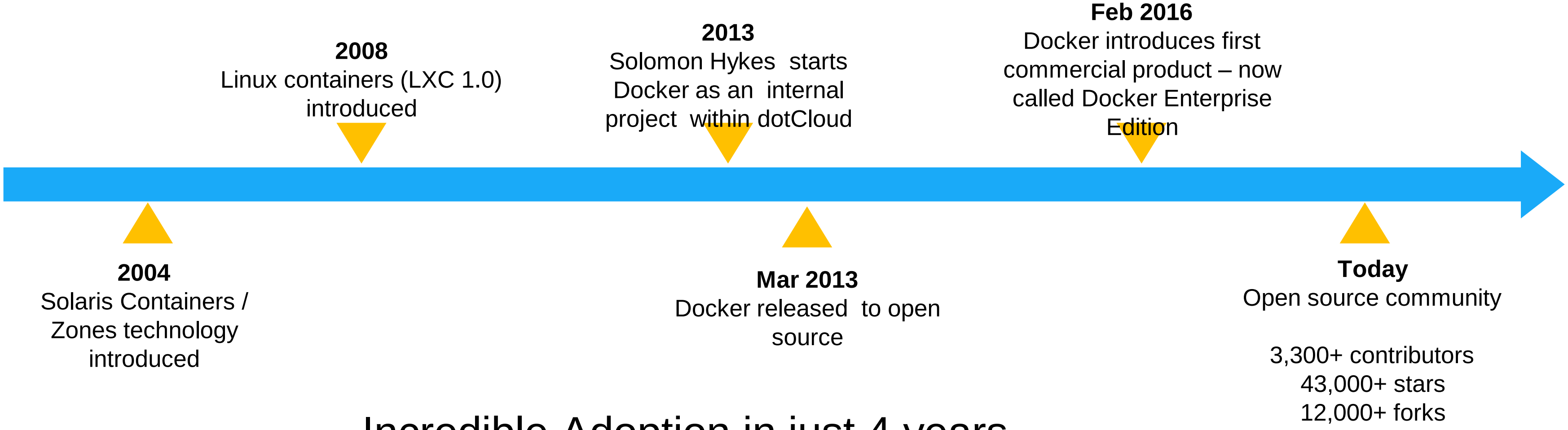
Kubernetes INside Docker

```
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
```

History of Docker



2004
Solaris Containers / Zones technology introduced

2008
Linux containers (LXC 1.0) introduced

2013
Solomon Hykes starts Docker as an internal project within dotCloud

Mar 2013
Docker released to open source

Feb 2016
Docker introduces first commercial product – now called Docker Enterprise Edition

Today
Open source community

3,300+ contributors
43,000+ stars
12,000+ forks

Incredible Adoption in just 4 years

14M
Docker Hosts

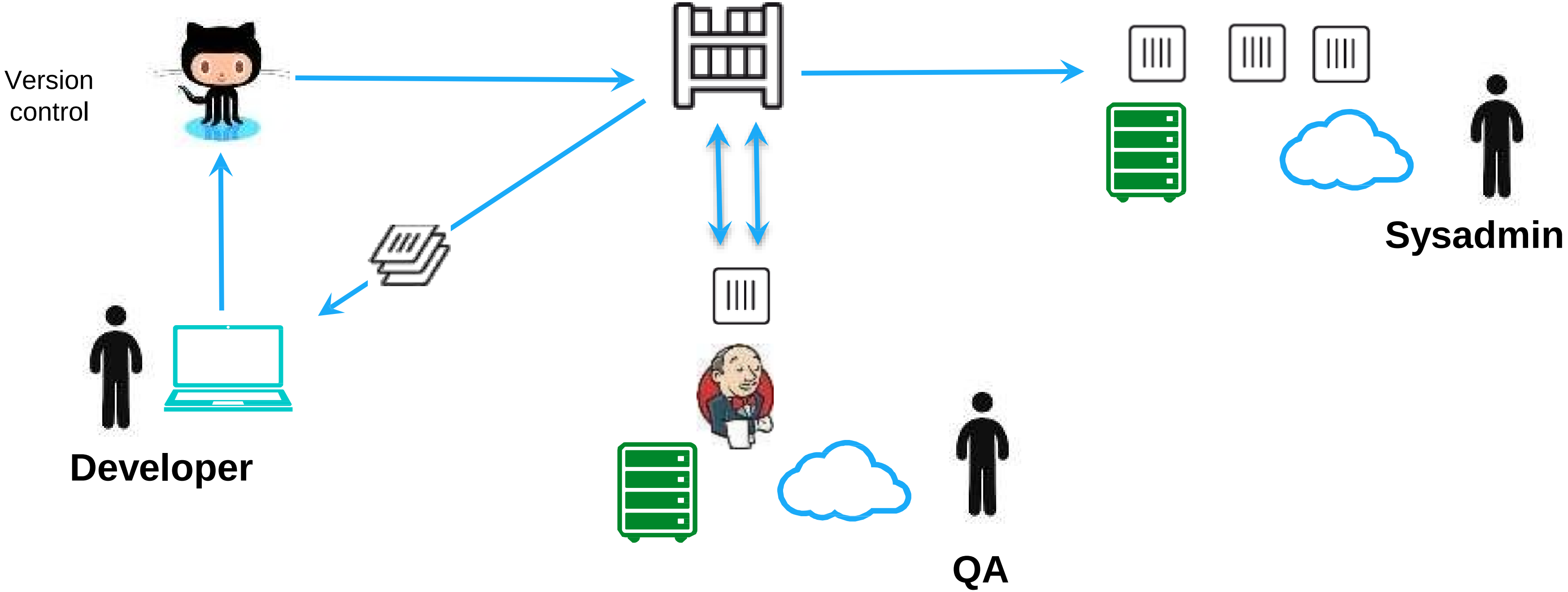
900K
Docker apps

77K%
Growth in Docker job listings

12B
Image pulls Over 390K% Growth

3300
Project Contributors

Continuous Integration and Delivery



Tug-of-war

Developers

- Freedom to create and deploy apps fast
- Define and package application needs

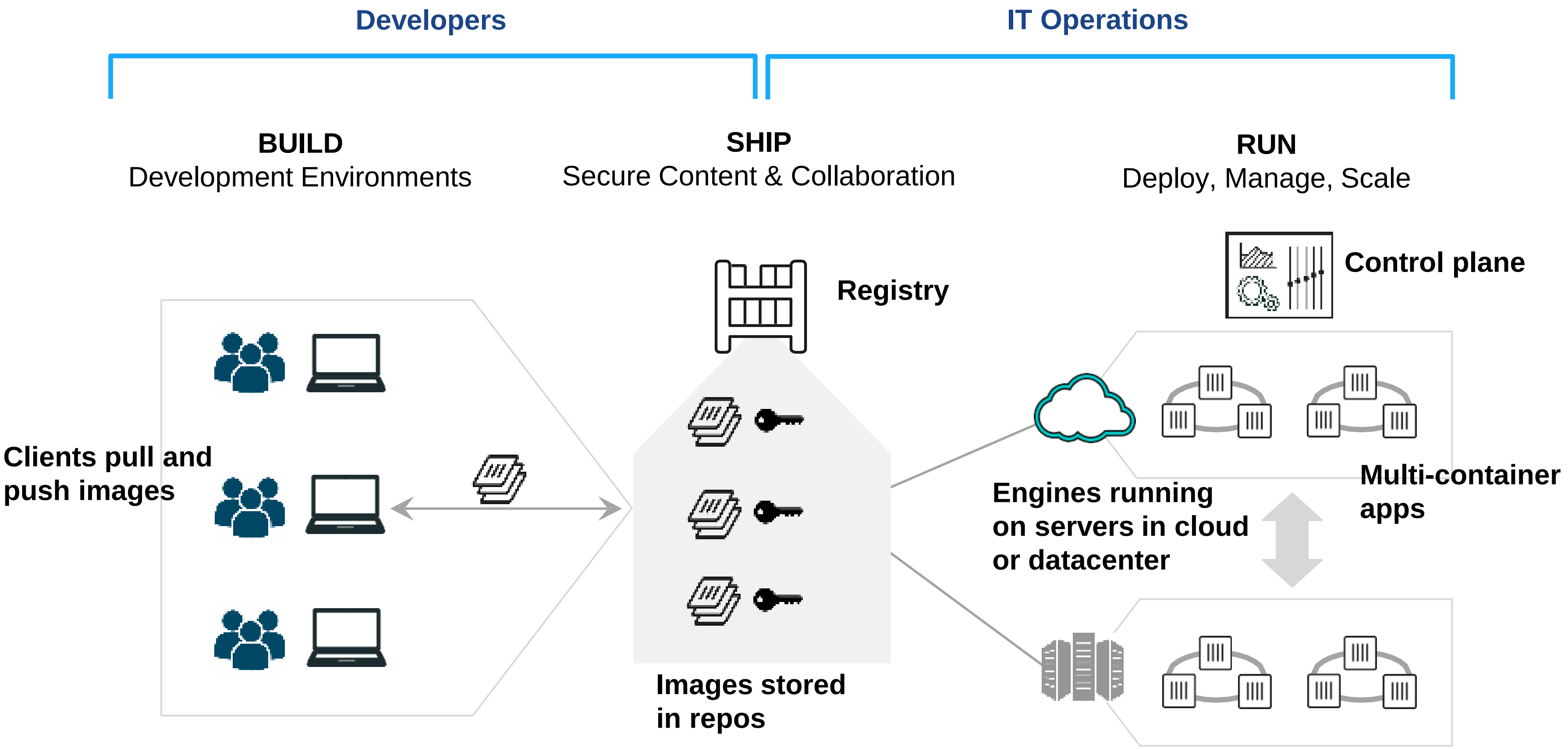
IT Operations

- Quickly and flexibly respond to changing needs
- Standardize, secure, and manage

The myth of bi-modal IT

	MICROSERVICES	TRADITIONAL APPS
Cloud or New Infrastructure	You are either here..	▨
Old Infrastructure	▨	…or here

Container as a service



Developers

IT Operations

BUILD
Development Environments

SHIP
Secure Content & Collaboration

RUN
Deploy, Manage, Scale

Registry

Control plane

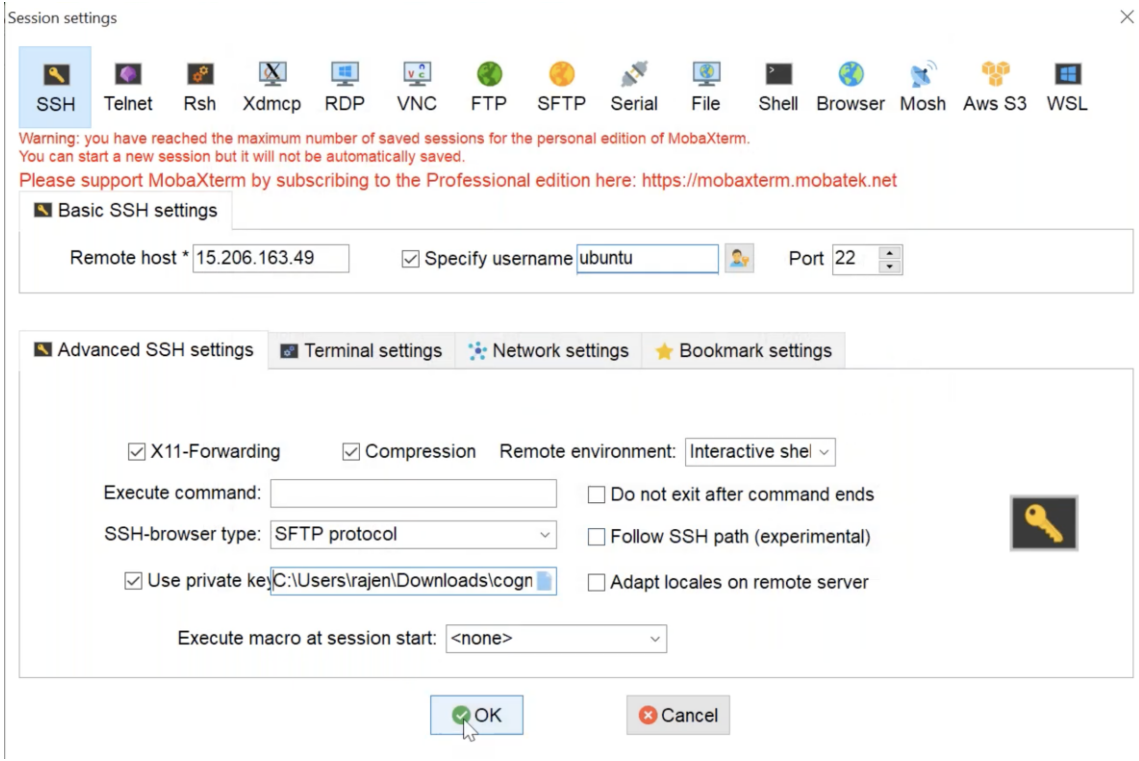
Clients pull and push images

Images stored in repos

Engines running on servers in cloud or datacenter

Multi-container apps

MobaXterm



Session settings

Warning: you have reached the maximum number of saved sessions for the personal edition of MobaXterm.
You can start a new session but it will not be automatically saved.

Please support MobaXterm by subscribing to the Professional edition here: https://mobaxterm.mobatek.net

Basic SSH settings

Remote host * 15.206.163.49 ☑ Specify username ubuntu Port 22

☑ X11-Forwarding ☑ Compression Remote environment: Interactive shel

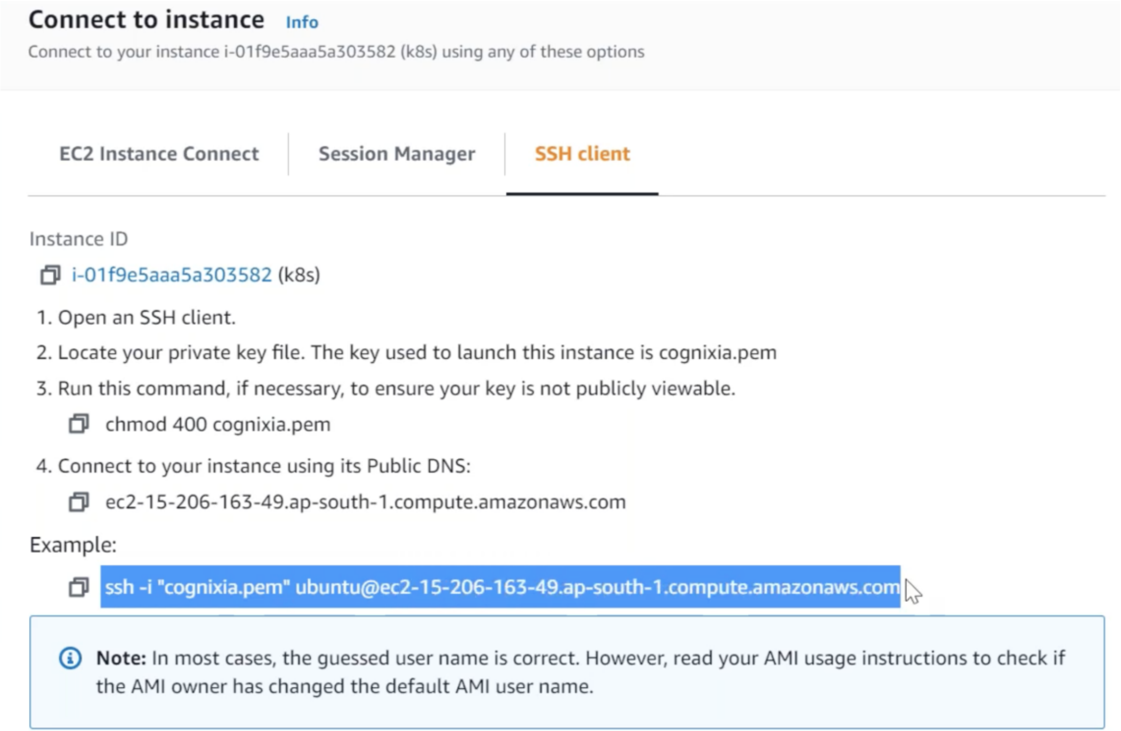
Execute command: □ Do not exit after command ends

SSH-browser type: SFTP protocol □ Follow SSH path (experimental)

☑ Use private keyC:\Users\rajen\Downloads\cogn □ Adapt locales on remote server

Execute macro at session start: <none>

h



Connect to instance

Connect to your instance i-01f9e5aaa5a303582 (k8s) using any of these options

Instance ID

i-01f9e5aaa5a303582 (k8s)

1. Open an SSH client.
2. Locate your private key file. The key used to launch this instance is cognixia.pem
3. Run this command, if necessary, to ensure your key is not publicly viewable.

```
chmod 400 cognixia.pem
```

4. Connect to your instance using its Public DNS:

```
ec2-15-206-163-49.ap-south-1.compute.amazonaws.com
```

Example:

```
ssh -i "cognixia.pem" ubuntu@ec2-15-206-163-49.ap-south-1.compute.amazonaws.com
```

Note: In most cases, the guessed user name is correct. However, read your AMI usage instructions to check if the AMI owner has changed the default AMI user name.