# Chapter 4
# Containerization with Docker

# Learning Topics

- Overview of Docker
- Docker Architecture
- Container & Images
- Docker Hub
- Docker Best Practices

# All Roads Lead to Cloud

Movement in the cloud



80%

Migrate workloads to cloud

Portability across environments

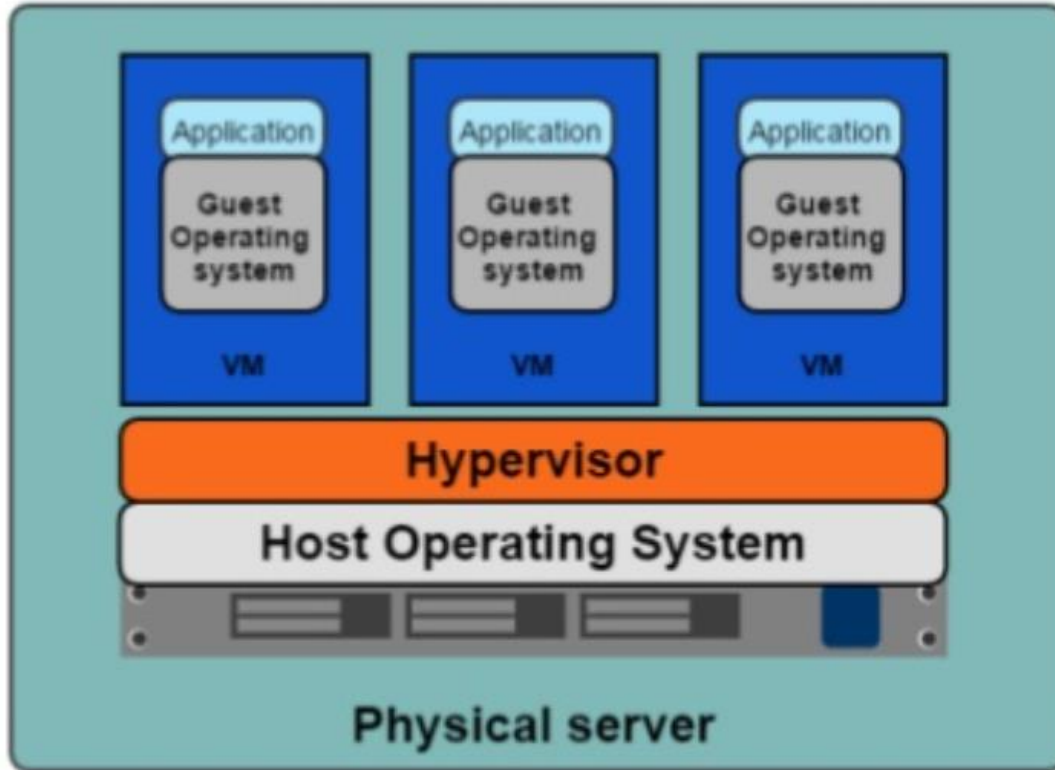Want to avoid cloud vendor lock-in

# Evolving Workloads

# Multiple Requirements to Fill



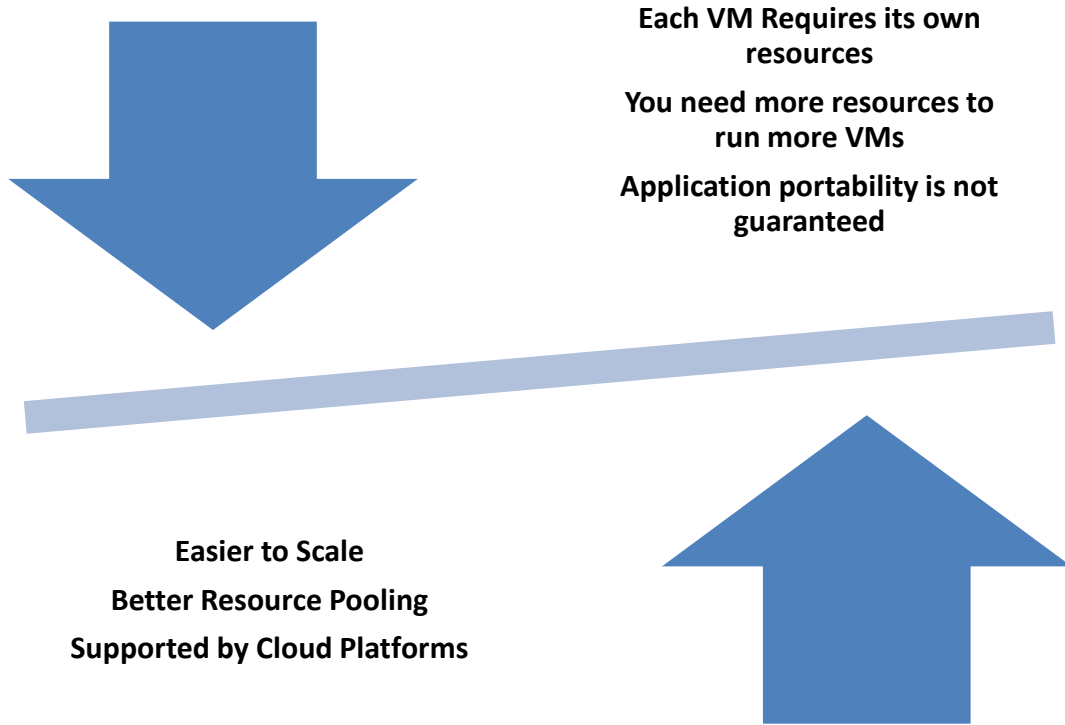| | |
|---|---|
| Bare Metal | Virtual |
| On Premises | Cloud |
| Linux | Windows |
| Traditional | Microservices |

# Hypervisor Based Deployments
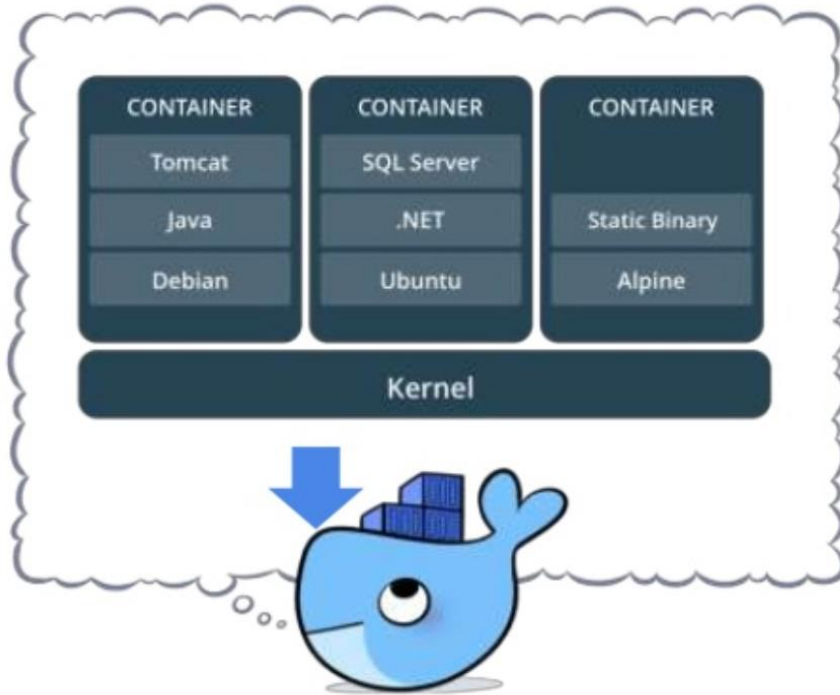


- One physical server can host multiple applications
- Each application runs in a virtual machine (VM)

# Virtual Machines – Pros & Cons



Each VM Requires its own resources

You need more resources to run more VMs

Application portability is not guaranteed

Easier to Scale

Better Resource Pooling

Supported by Cloud Platforms

# Containers

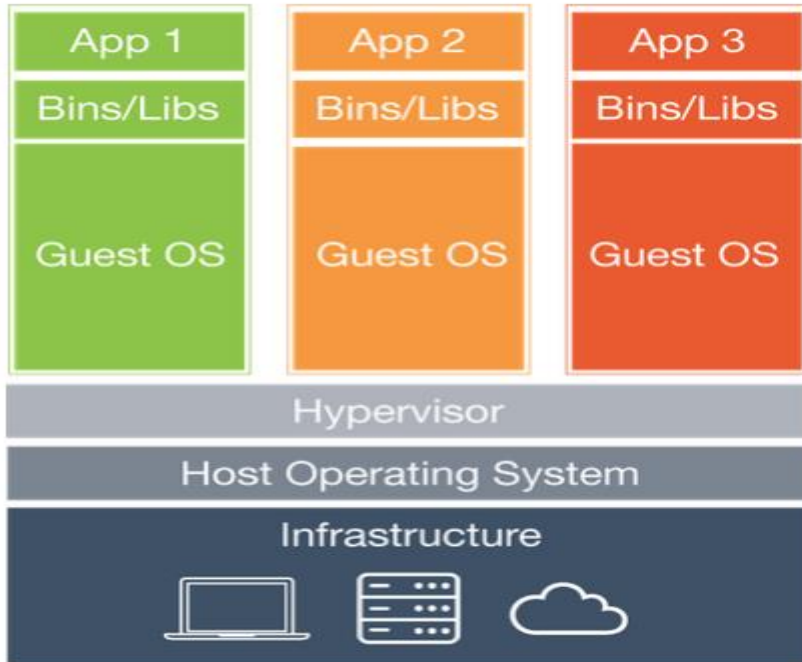- An object for holding or transporting something



- Standardized packaging for software and dependencies
- Share same OS Kernel
- Isolate apps from each other

# VMs Vs Containers

Infrastructure level construct

| App 1 | App 2 | App 3 |
|-------|-------|-------|
| Bins/Libs | Bins/Libs | Bins/Libs |
| Guest OS | Guest OS | Guest OS |

Hypervisor

Host Operating System

Infrastructure

App level construct

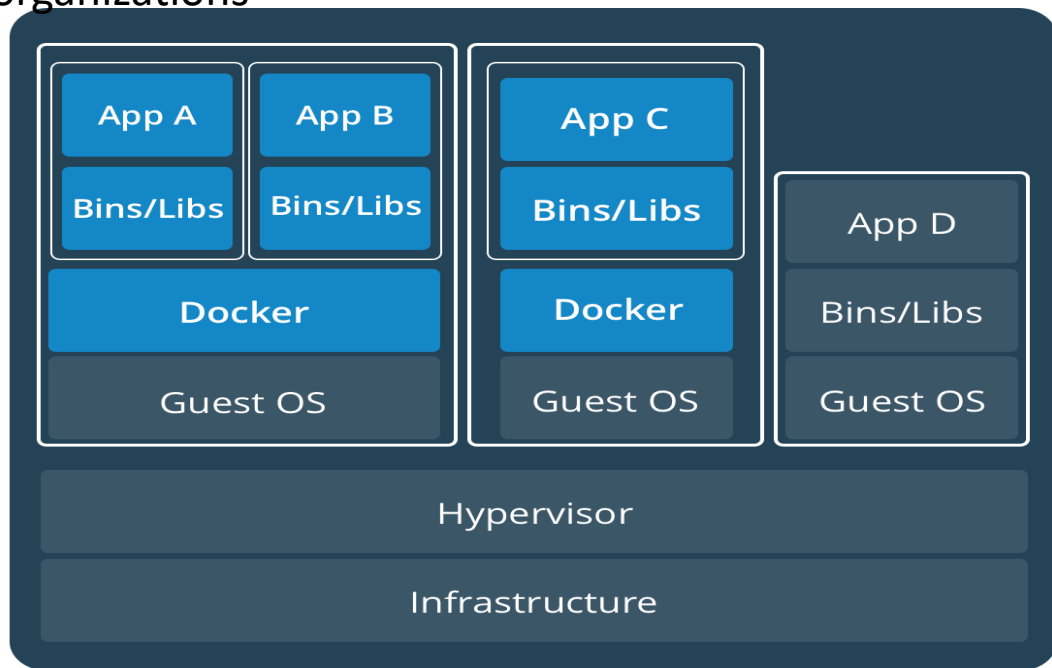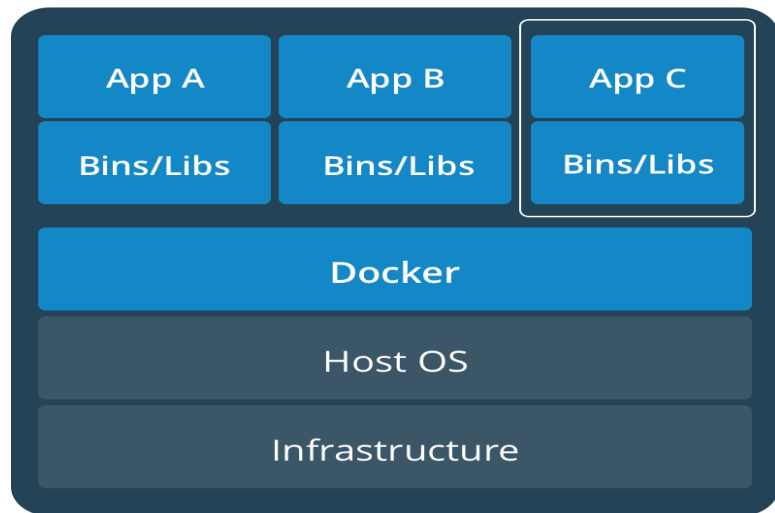| App 1 | App 2 | App 3 |
|-------|-------|-------|
| Bins/Libs | Bins/Libs | Bins/Libs |

Docker Engine

Operating System

Infrastructure
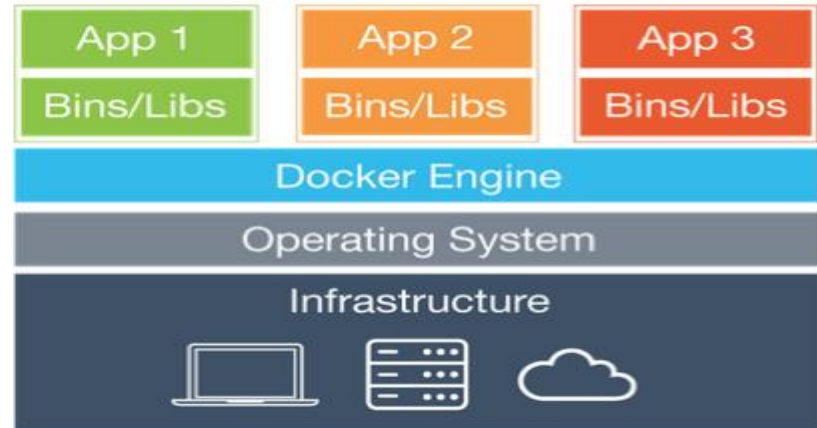
# Containers AND VMs

- Containers can run on standalone physical server and virtual machines
- Provides tremendous flexibility for organizations
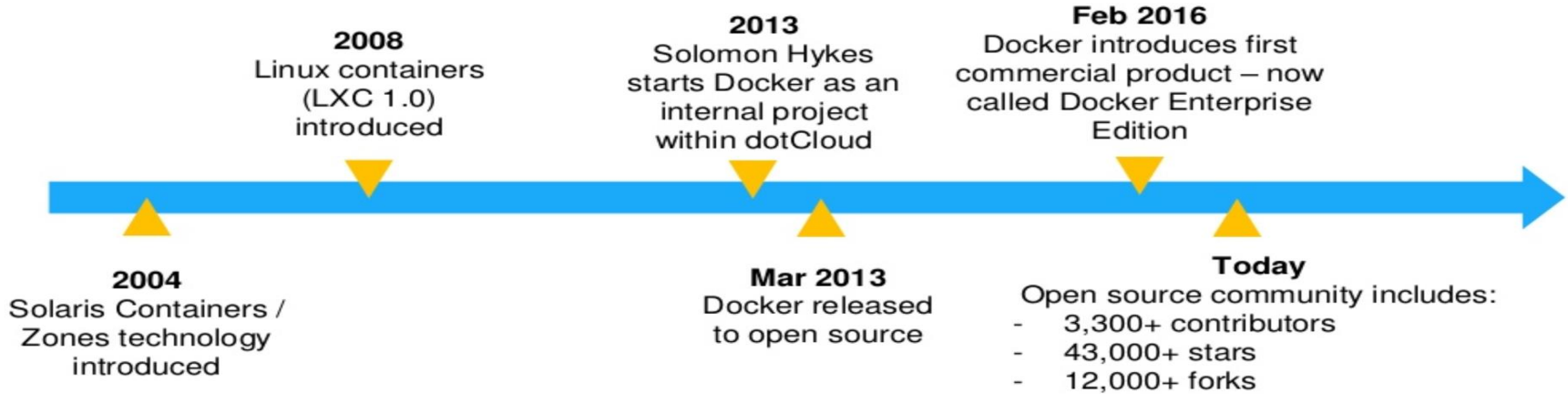- Deploy and manage apps

# What is Docker?

**Docker** is an open platform for developers and sysadmins to build, ship, and run distributed applications, whether on laptops, data centre VMs, or the cloud.

Docker is a technology to package an application and all its dependencies into a single, easily transportable container

Fixes the traditional "But it works on my machine" problem

# Docker History

**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 includes:
- 3,300+ contributors
- 43,000+ stars
- 12,000+ forks

# Docker Success Story



| 14M Docker Hosts | 900K Docker apps | 77K% Growth in Docker job listings | 12B Image pulls Over 390K% Growth | 3300 Project Contributors |
|---|---|---|---|---|

# Advantages of Docker

- Consistency
  - Write once, deploy anywhere
- A complete platform
  - Manage entire lifecycle
  - Base engine for containers
  - Registry for image management
  - Compose for orchestration
  - Swarm for clustering
  - Machine for provisioning

# Use Cases of Docker

- Distributed Applications
- Microservices
- Continuous Integration
- Continuous Deployment
- Setting up Development Environment
- Build, Ship and Run Any App, Anywhere

# Docker – Lab 1

- Run "Hello World" Docker Container

# Docker Basics

**Image**
The basis of a Docker container. The content at rest.

**Container**
The image when it is 'running.' The standard unit for app service

**Engine**
The software that executes commands for containers. Networking and volumes are part of Engine. Can be clustered together.

**Registry**
Stores, distributes and manages Docker images

**Control Plane**
Management plane for container and cluster orchestration

# 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.
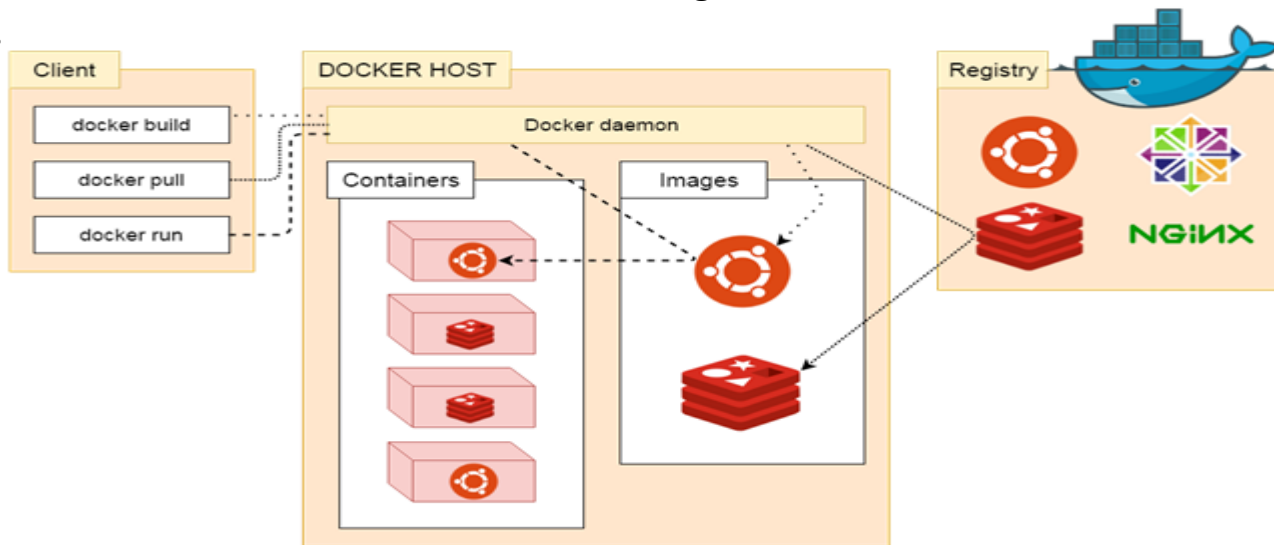
# Docker Daemon

- The Docker daemon listens for Docker API requests

- It manages Docker objects such as images, containers, networks, and volumes

- A daemon can also communicate with other daemons to manage Docker services.

# Docker Client

- The Docker client is the primary way that many Docker users interact with Docker

- When we use commands such as docker run, the client sends these commands to docker daemon, which carries them out

- The docker command uses the Docker API

- The Docker client can communicate with more than one daemon

# Docker Registries

- A Docker *registry* stores Docker images.

- Docker Hub and Docker Cloud are public registries that anyone can use

- Docker is configured to look for images on Docker Hub by default

- You can configure your own private registry

- When you use the **docker pull** or **docker run** commands, the required images are pulled from your configured registry

- When you use the **docker push** command, your image is pushed to your configured registry

# Images

- In Docker, everything is based on Images

- An *image* is a read-only template with instructions for creating a Docker container

- Usually 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.

# Lab 2 – Run Container From Image

- Create couple of images
  - Ubuntu
  - Apache Server
- Run all commands for images in your environment

# Containers

- The basic purpose of Docker is to run containers
- A container is a runnable instance of an image
- You can create, start, stop, move, or delete a container using the Docker API or CLI
- When a container is removed, any changes to its state that are not stored in persistent storage disappear
- A container is defined by its image as well as any configuration options you provide to it when you create or start it

# Containers Cont.

- Container can expose ports and volumes to interact with other containers or/and outer world
- Containers are always created from images
- Containers gives you instant application portability
- All containers must use the same operating system
- A container is defined by its image as well as any configuration options provided to it when it is created or started

# Container Commands - Lifecycle

- docker create creates a container but does not start it.

- docker rename allows the container to be renamed.

- docker run creates and starts a container in one operation.

- docker rm deletes a container.

# Container Commands – Start and Stop

- docker start starts a container so it is running.
- docker stop stops a running container.
- docker restart stops and starts a container.
- docker wait blocks until running container stops.
- docker attach will connect to a running container.
- docker exec -it containername /bin/bash to enter a running instance

# Container Commands – Information

- [docker ps](#) shows running containers.
- docker ps -a shows running and stopped containers.
- [docker inspect](#) looks at all the info on a container (including IP address).
- [docker port](#) shows public facing port of container.
- [docker top](#) shows running processes in container.
- [docker stats](#) shows containers' resource usage statistics.
- docker stats --all shows a running list of containers.
- [docker cp](#) copies files or folders between a container and the local filesystem.

# Lab 3 – Run Basic Commands

- Run all commands for images in your environment

# Containers as a Service
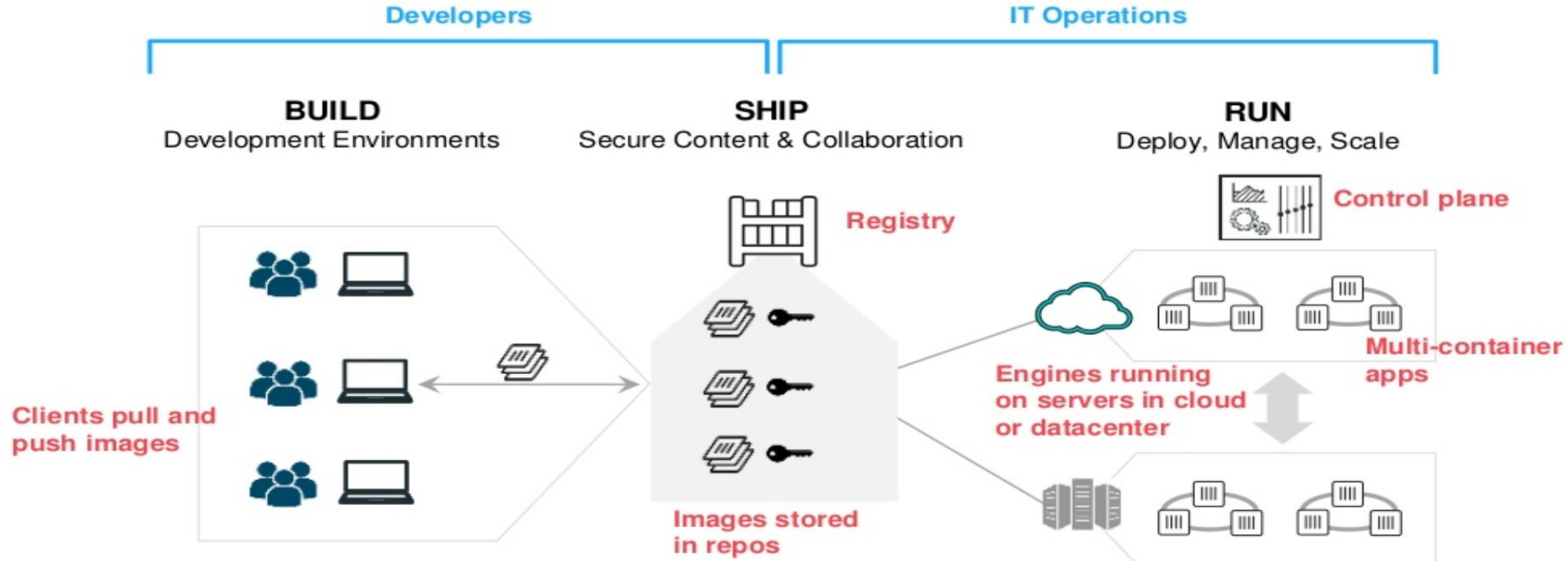
# Lab 4 – Modify Running Container

# Dockerfile

- A text document that contains all the commands, in order, a user could call on the command line to assemble an image
- Build instructions to build the image
- Usually dockerfile is called **Dockerfile**
- Located in root of context
  - docker build -f /path/to/a/Dockerfile .

# Dockerfile Instructions

- A Dockerfile must start with a `**FROM**` instruction

- The **FROM** instruction specifies the base image from which you are building

- Docker treats lines that *begin* with **#** as a comment

# Lab 5 – Dockerfile

# Docker Hub

- Docker Hub is a cloud-based registry service
- Docker users and partners create, test, store and distribute container images
- Centralized resource for container image discovery, distribution and change management
- User and team collaboration and workflow automation throughout the development pipeline.

# Docker Hub - Features

- **Image Repositories** - Lets you share images with co-workers, customers, or the Docker community at large

- **Organizations** - Create work groups to manage access to image repositories

- **GitHub and Bitbucket Integration** - Add the Hub and your Docker Images to your current workflows

# Lab 6 - Docker Hub – Repositories

- Create new repository in Docker Hub
- Create an image
  - docker commit <existing-container> <hub-user>/<repo-name>[:<tag>]
- Push an image in your repository
  - docker push <hub-user>/<repo-name>:<tag>
- Pull image from your neighbour's repository and run a container using this image

# Docker Compose

- Tool for defining and running multi-container Docker applications
- Uses YAML file for configuration
- 3 Step Process
  - Create Dockerfile with Environment Information
  - Define Services for Your App in YAML file
  - Run Docker Compose

# Lab 7 - Docker Compose

- Create a Stack Using Docker Compose

# Container Best Practices

- Containers should be immutable
- Containers should be ephemeral
- Containers should be lightweight
- One container, One responsibility, One process
- Store share data in volumes, not in containers
- Don't store credentials in the image

# Docker Development Best Practices

- Keep your images small

- Use tags to reference specific versions of your images

- Store data using volumes

# Docker Security Best Practices

- Run Docker inside a virtual machine

- Docker image ids are sensitive information. Should be treated as passwords, not exposed to outside world.

- Set the container to be read-only

- Set volumes to be read-only

- Define and run user in your Dockerfile so you don't run as root inside the container

- Don't use an image unless it's official

This concludes Chapter 4 – Containerization with Docker

Let us move to Chapter 5 – Configuration Management with Ansible