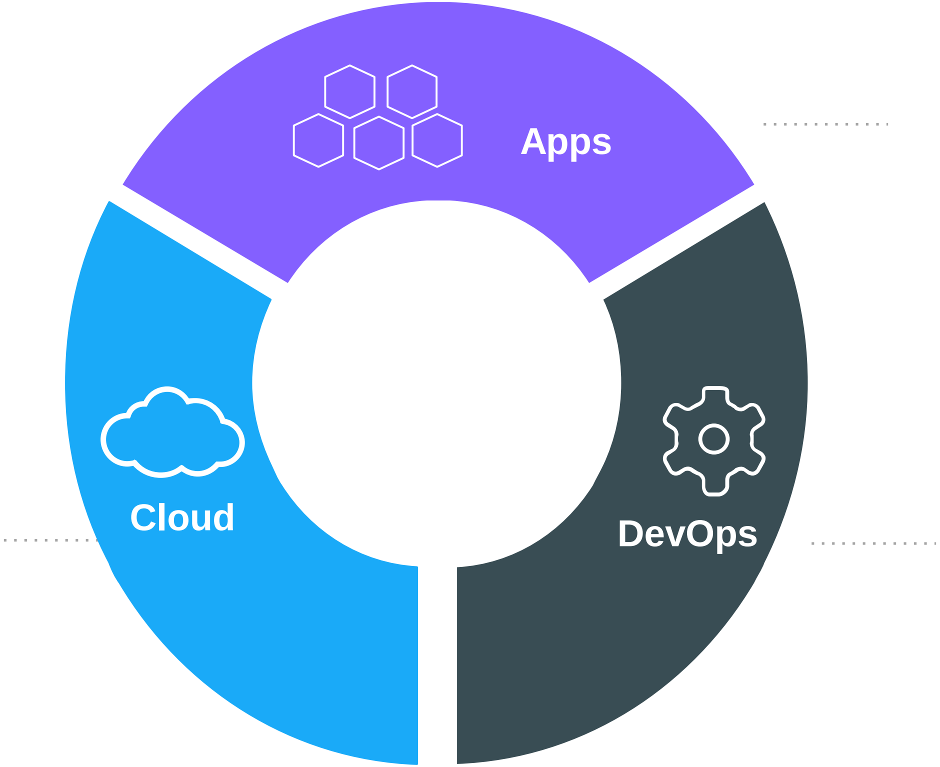
Introduction to Docker

The IT Landscape is Changing



Movement in the cloud

80%

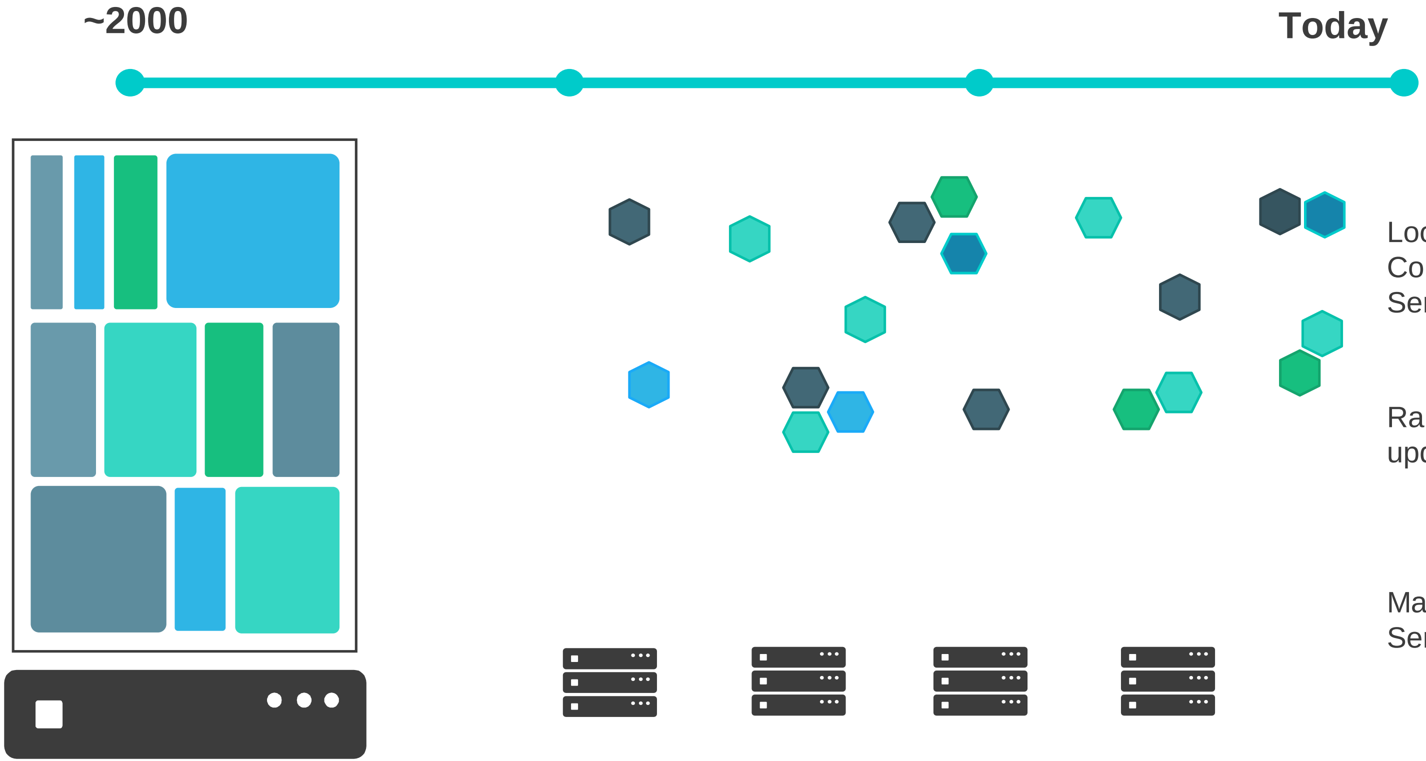
Migrate workloads to cloud

Portability across environments

Want to avoid cloud vendor lock-in

State of App development Survey: Q1 - 2016

Applications are transforming



Monolithic

Slow changing

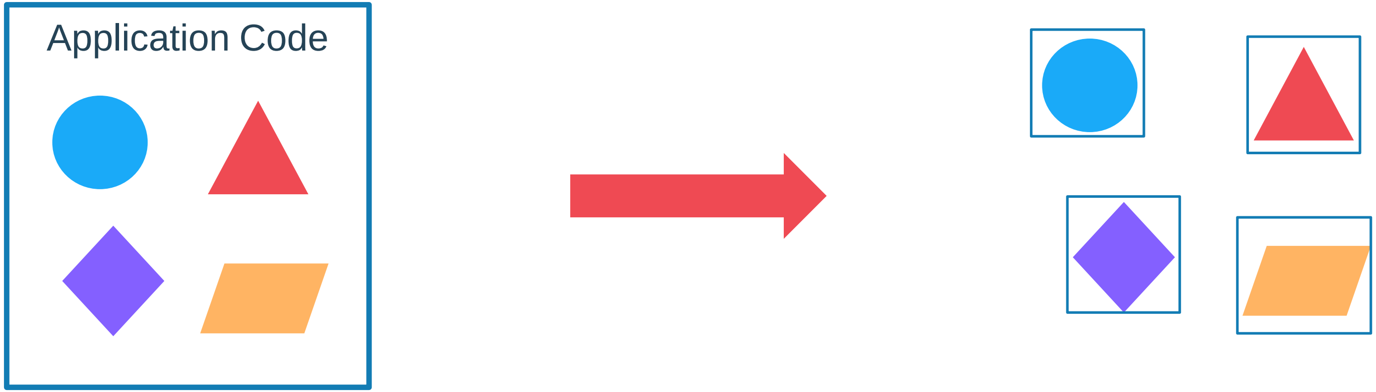
Big Servers

Loosely Coupled Services

Rapidly updated

Many Small Servers or devices

Application Modernization



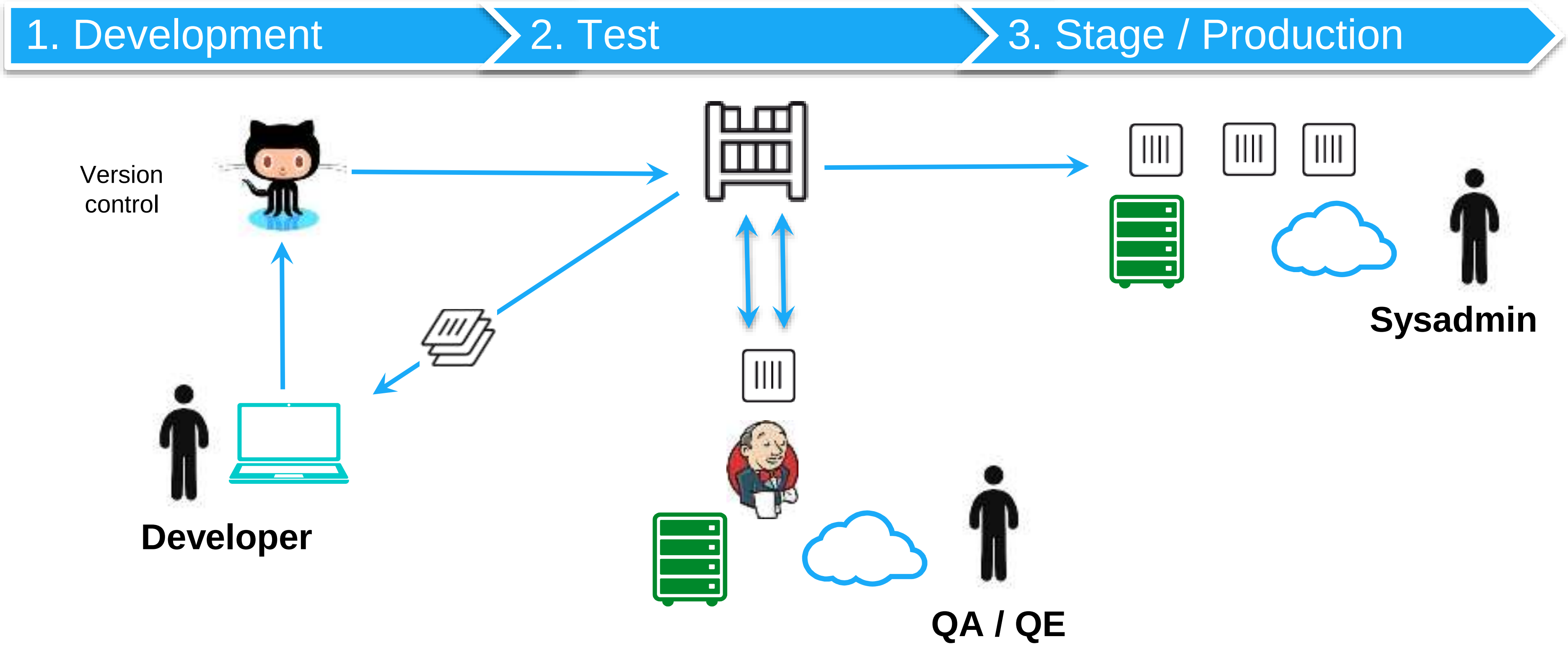
Developer Issues:

- 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

12-Factor Apps: Make the app independently scalable, stateless, highly available by design

Continuous Integration and Delivery



Tug of War Between Developers and Ops

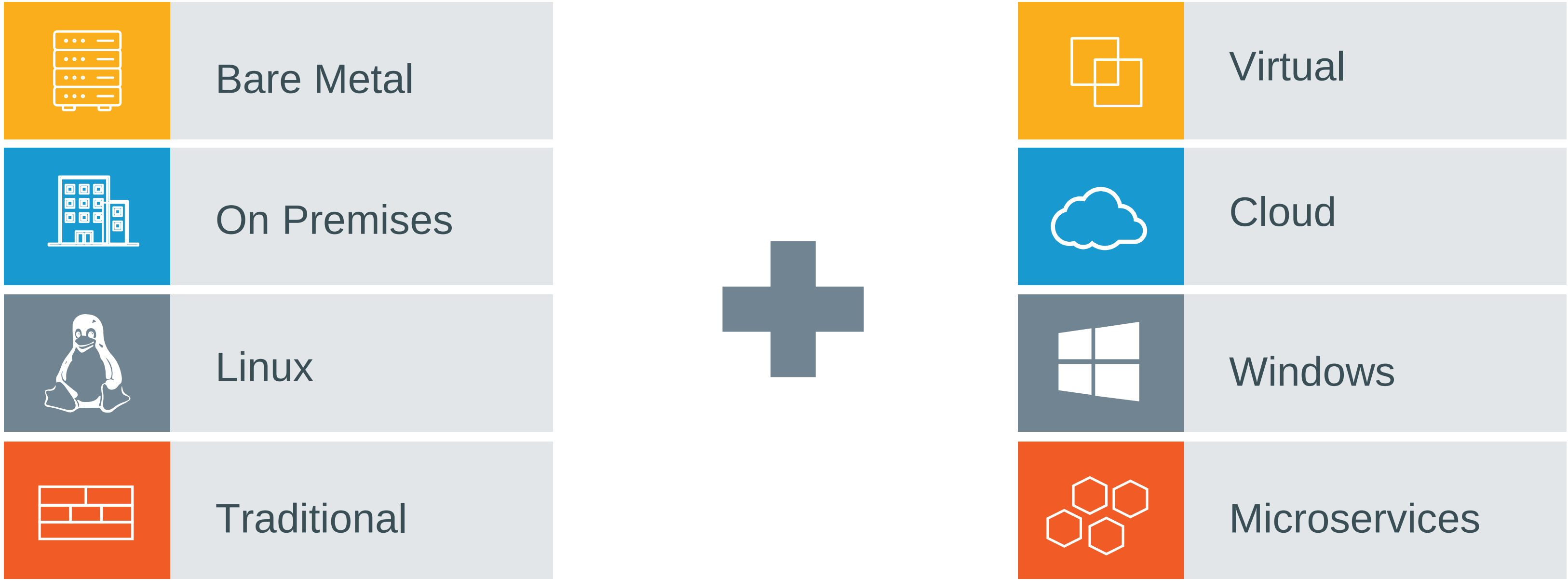
Developers

- 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

Organizations Must Deal with Diverse Technology



Bare Metal

On Premises

Linux

Traditional

Virtual

Cloud

Windows

Microservices

…and Diverse Organizations

Developers

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

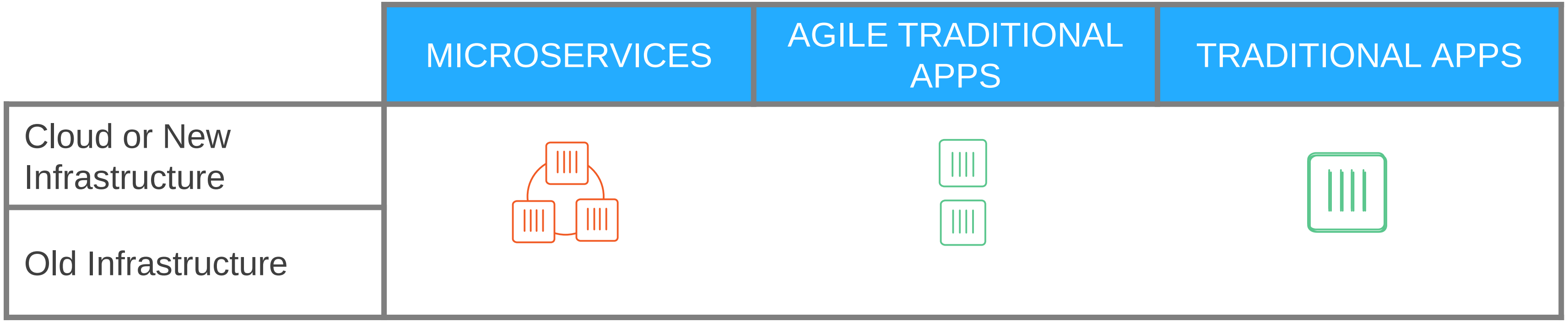
IT Operations

- 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

Enabling a Journey

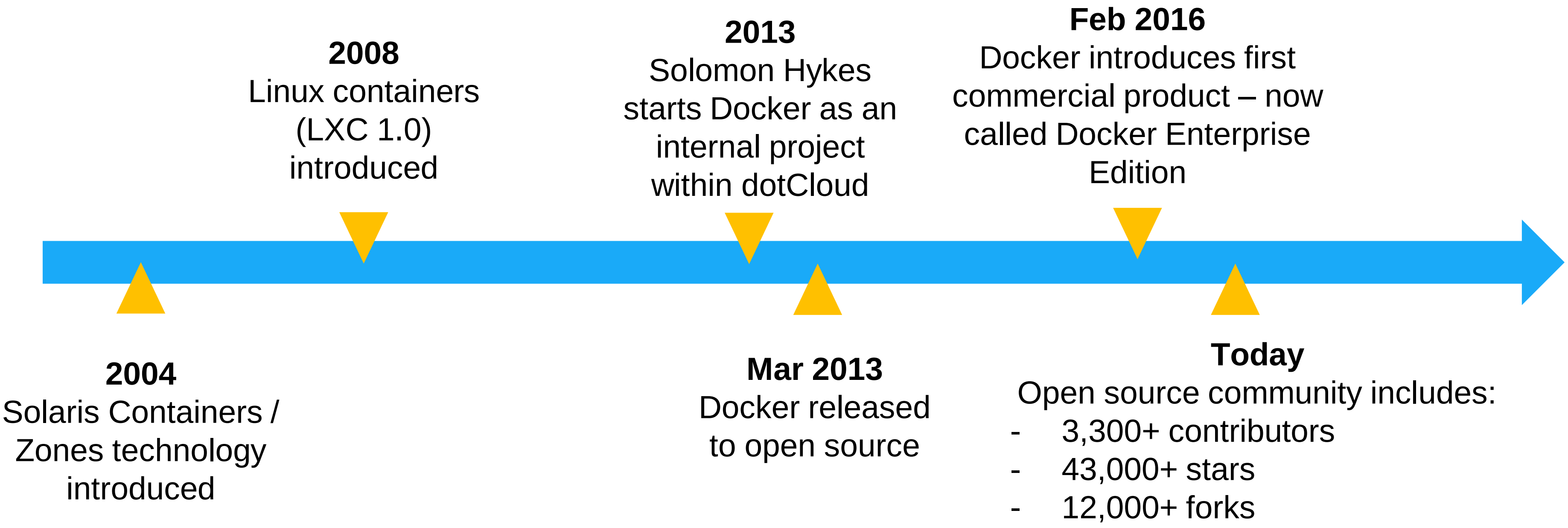


	MICROSERVICES	AGILE TRADITIONAL APPS	TRADITIONAL APPS
Cloud or New Infrastructure			
Old Infrastructure			

…that is past AND future proof

Docker and Container Overview

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

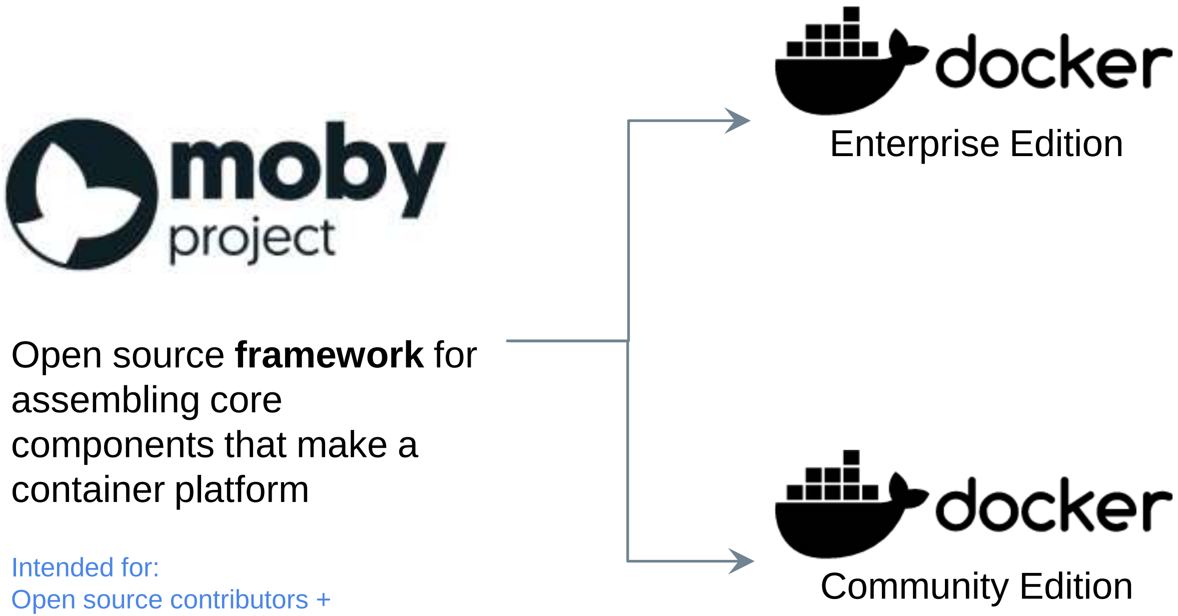
Mar 2013
Docker released to open source

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

Today
Open source community includes:

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

The Docker Family Tree



Open source **framework** for assembling core components that make a container platform

Intended for:
Open source contributors +
ecosystem developers

Enterprise Edition

Subscription-based, commercially supported **products** for delivering a secure software supply chain

Intended for:
Production deployments +
Enterprise customers

Community Edition

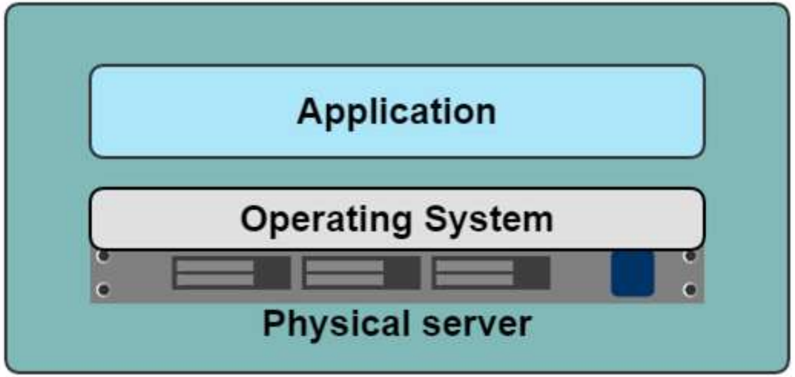
Free, community-supported **product** for delivering a container solution

Intended for:
Software dev & test

A History Lesson

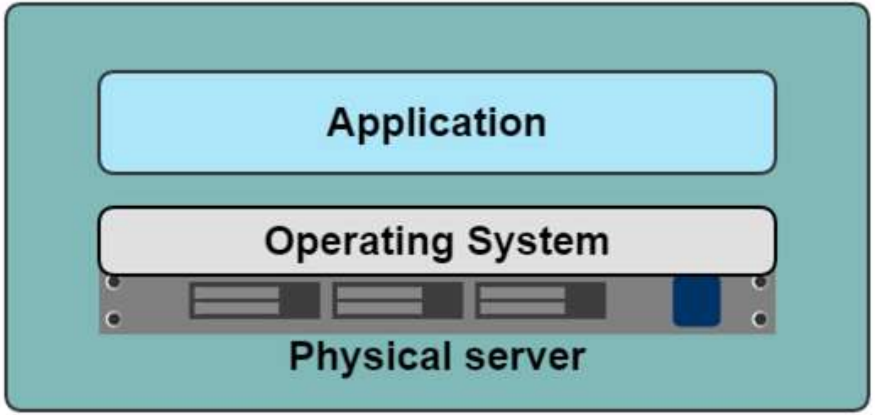
In the Dark Ages

One application on one physical server



Historical limitations of application deployment

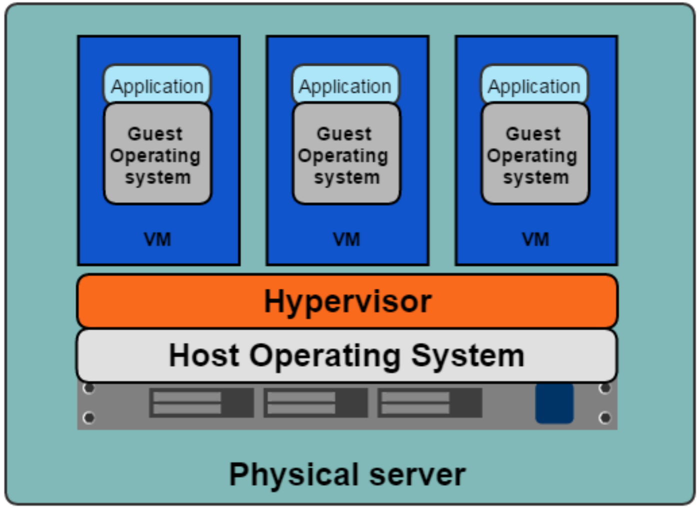
- Slow deployment times
- Huge costs
- Wasted resources
- Difficult to scale
- Difficult to migrate
- Vendor lock in



A History Lesson

Hypervisor-based Virtualization

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



Benefits of VMs

- 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

Microsoft Azure

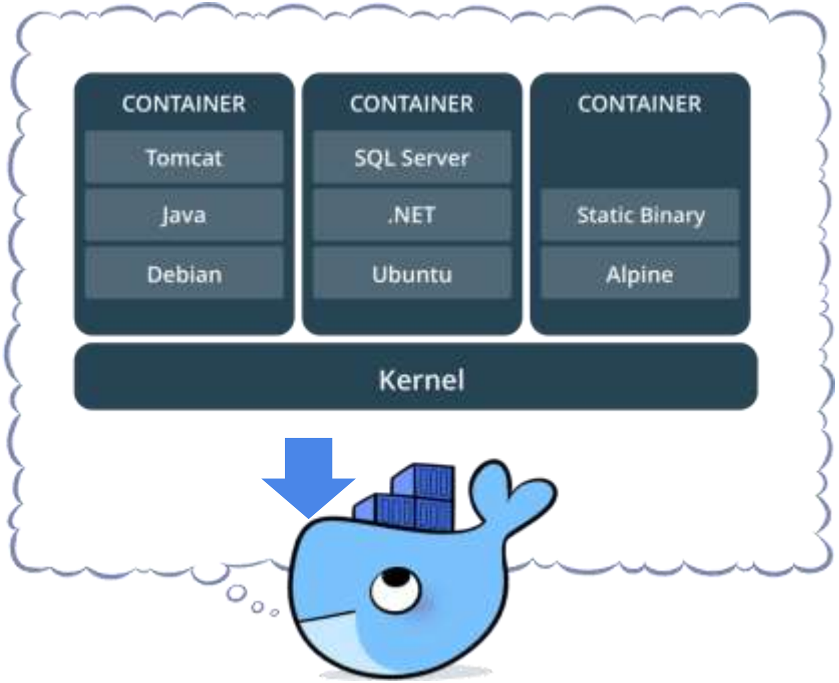
amazon web services™

vmware®

Limitations of VMs

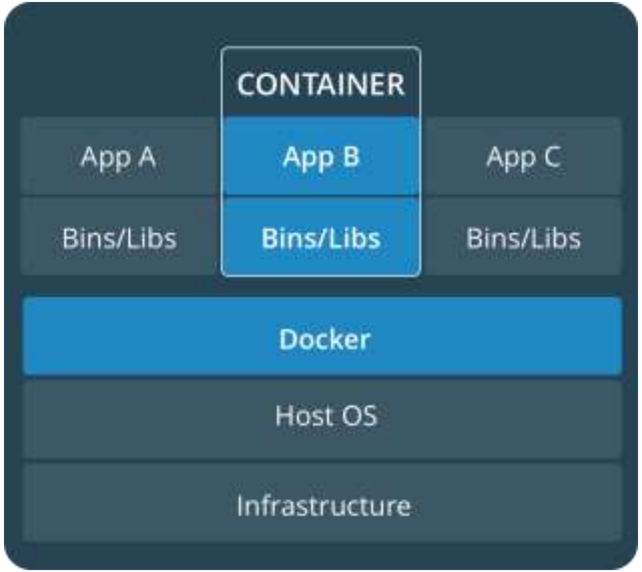
- Each VM stills requires
  - CPU allocation
  - Storage
  - RAM
  - An entire guest operating system
- The more VMs you run, the more resources you need
- Guest OS means wasted resources
- Application portability not guaranteed

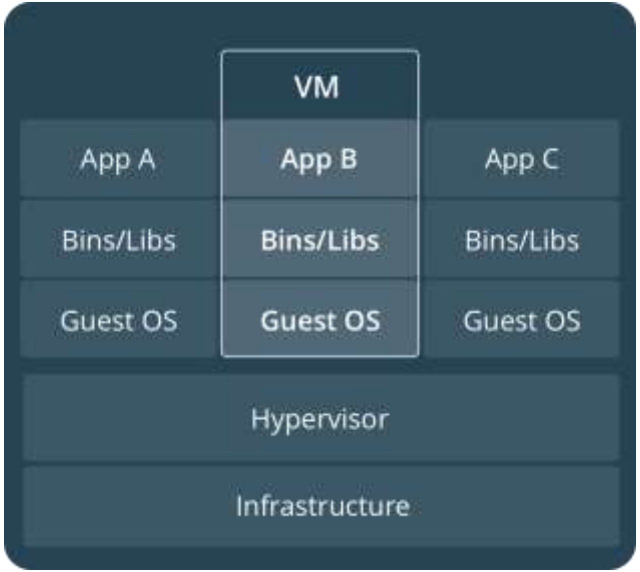
What is a container?



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

Comparing Containers and VMs

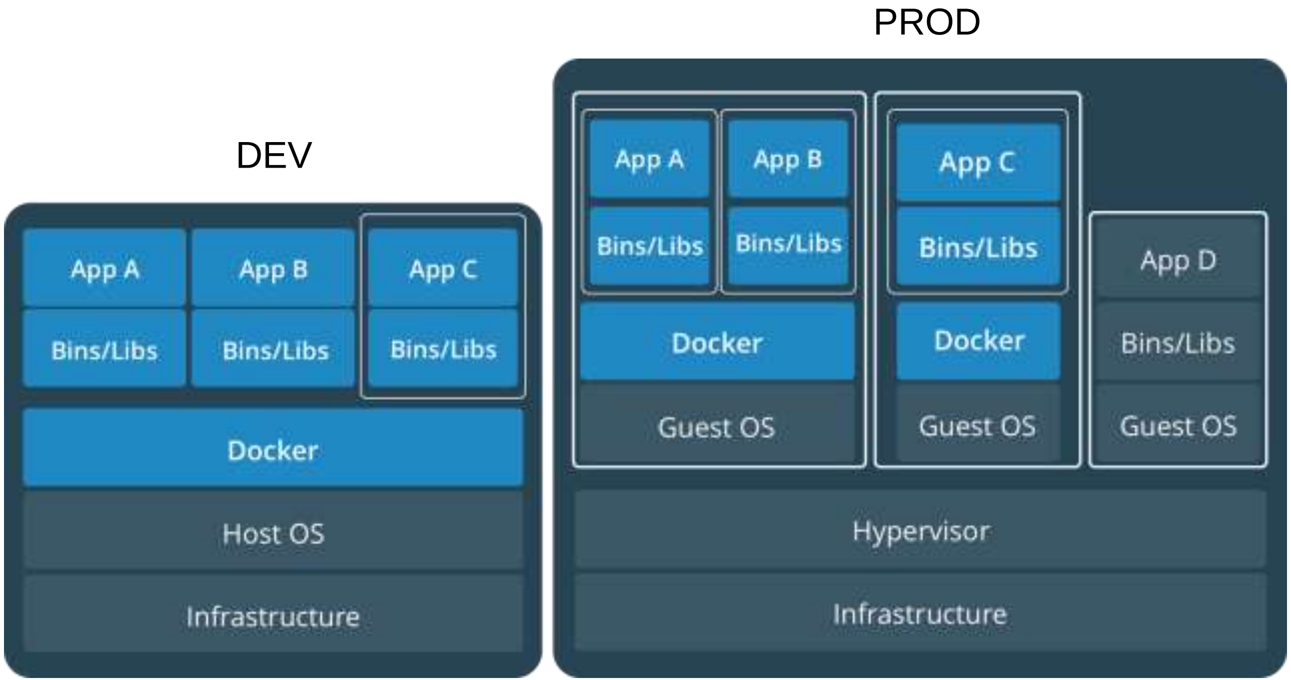




Containers are an app level construct

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

Containers and VMs together



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

Key Benefits of Docker Containers

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

Container Solutions & Landscape

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.

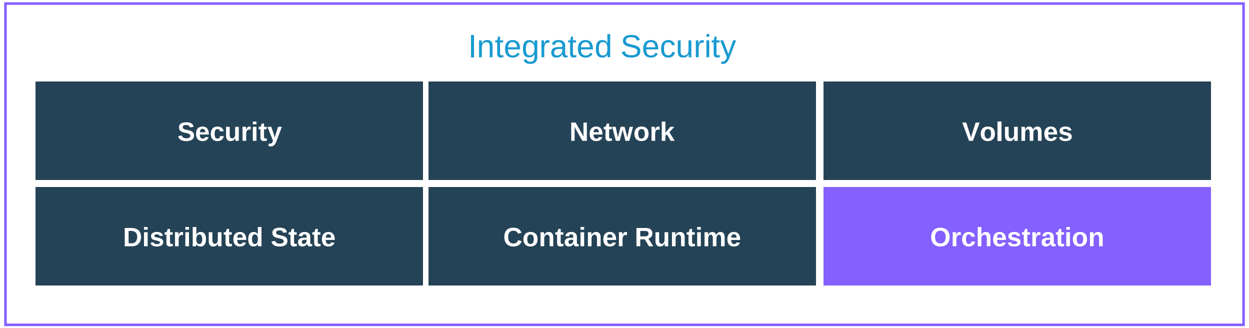
Registry

Stores, distributes and manages Docker images

Control Plane

Management plane for container and cluster orchestration

Foundation: Docker Engine



Integrated Security

Security

Network

Volumes

Distributed State

Container Runtime

Orchestration

Docker Engine

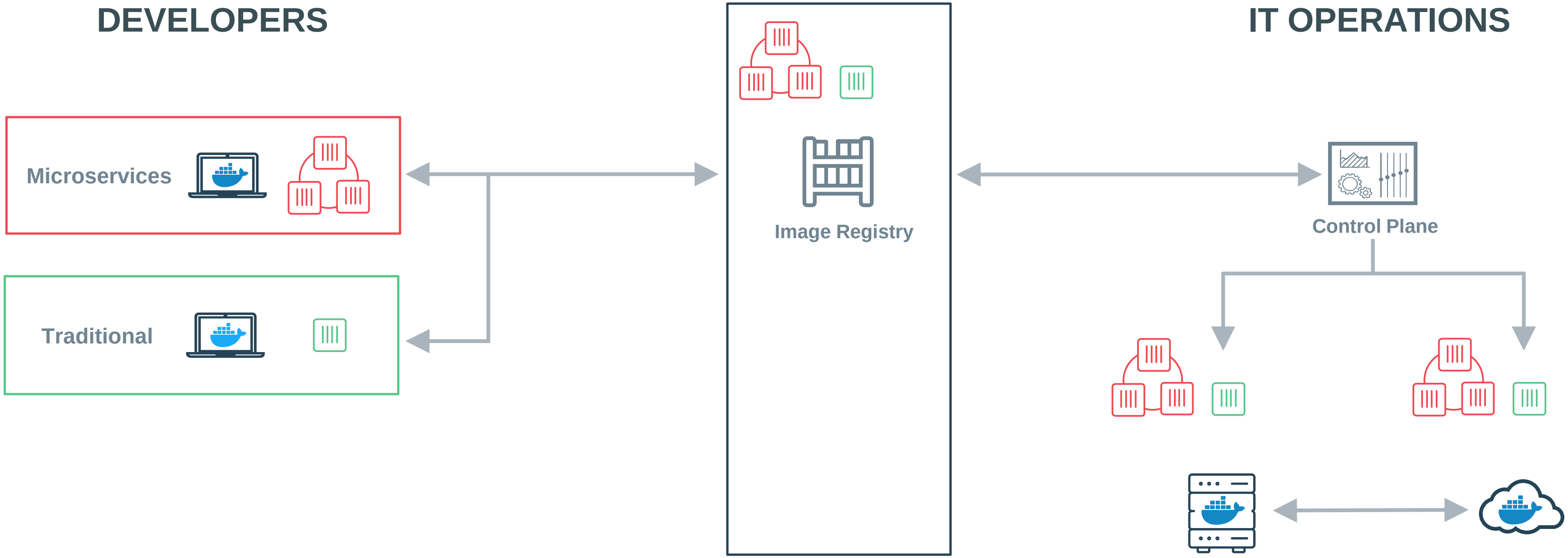
DEVELOPERS

IT OPERATIONS

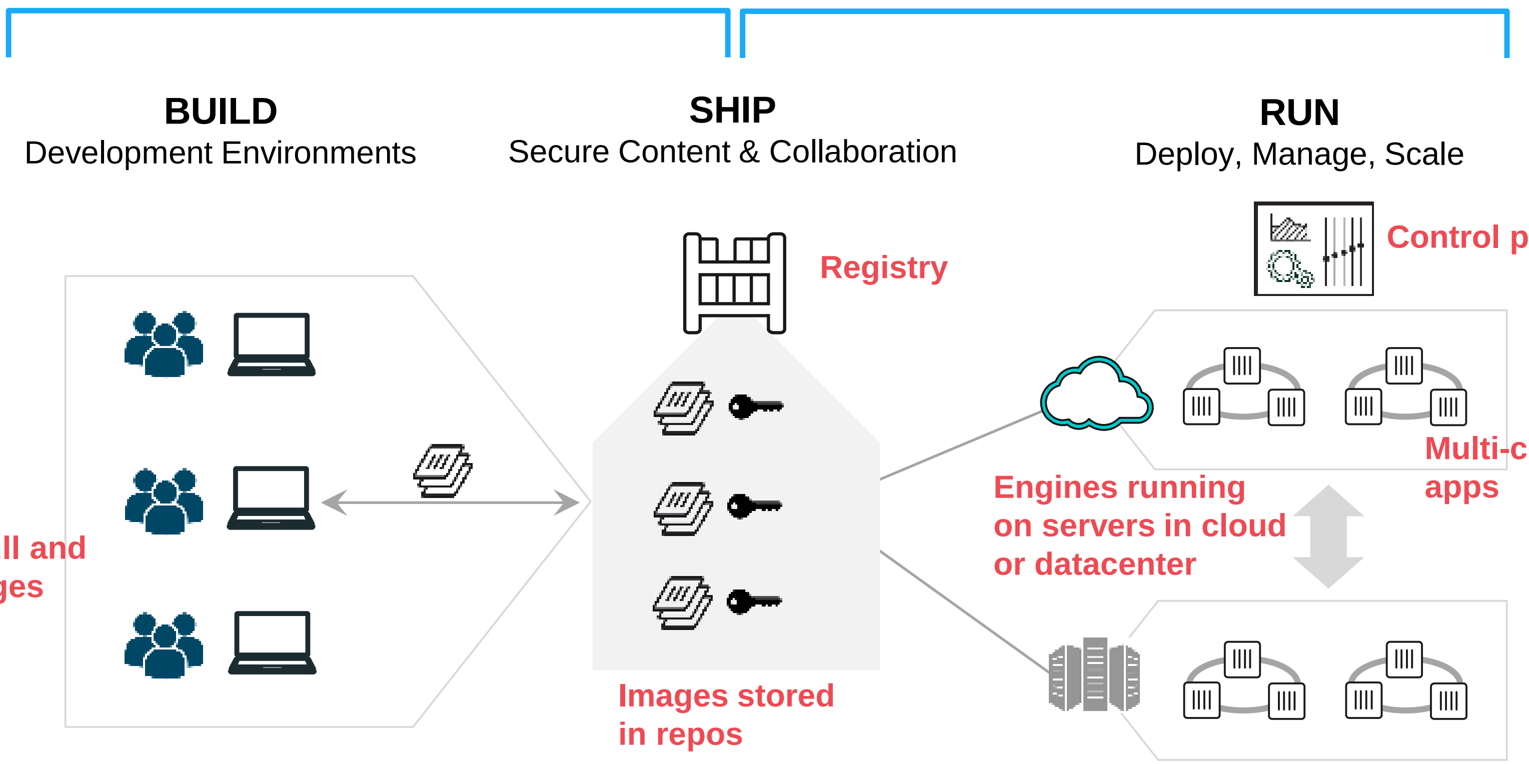
Microservices

Traditional

Building a Software Supply Chain



Containers as a Service



Developers

IT Operations

BUILD
Development Environments

SHIP
Secure Content & Collaboration

RUN
Deploy, Manage, Scale

Control plane

Registry

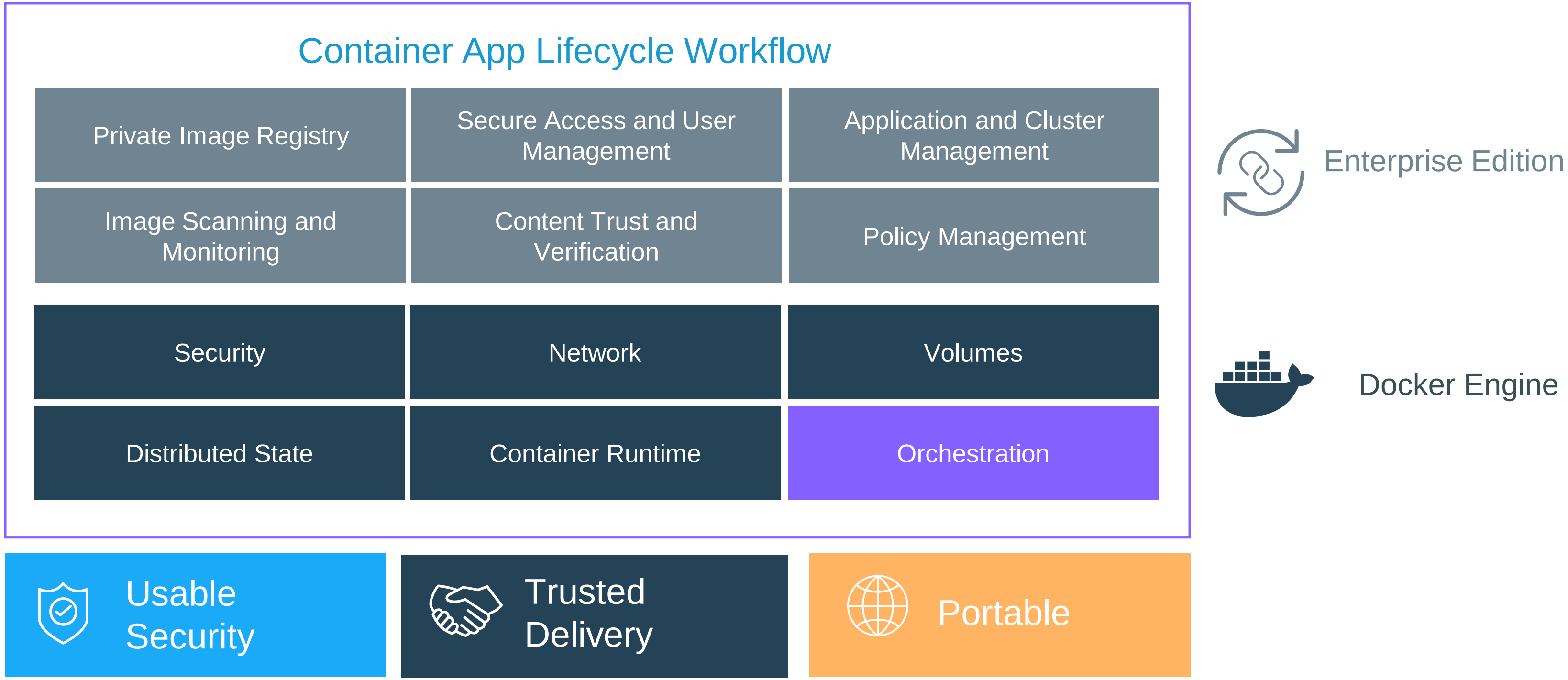
Multi-container apps

Engines running on servers in cloud or datacenter

Clients pull and push images

Images stored in repos

Building a Secure Supply Chain



Container App Lifecycle Workflow

Private Image Registry

Secure Access and User Management

Application and Cluster Management

Image Scanning and Monitoring

Content Trust and Verification

Policy Management

Security

Network

Volumes

Distributed State

Container Runtime

Orchestration

Enterprise Edition

Docker Engine

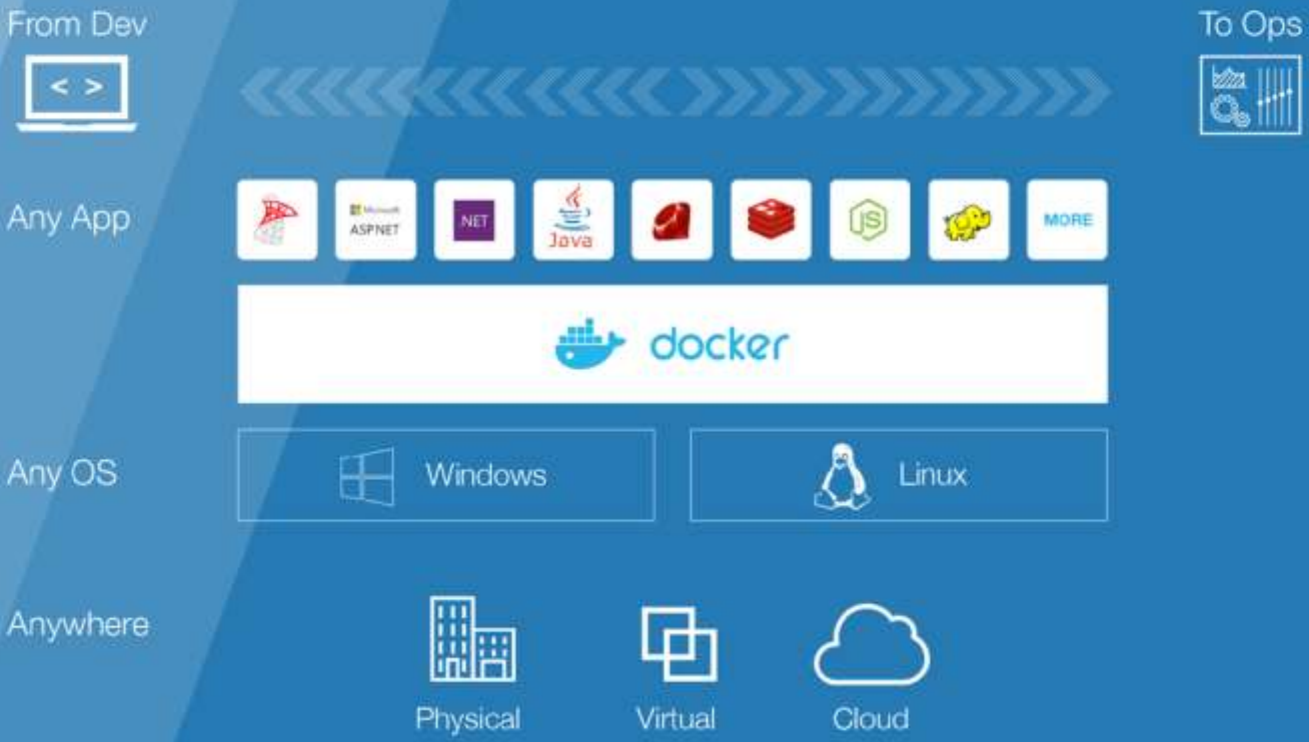
Usable Security

Trusted Delivery

Portable

Build, Ship, and Run

Build, Ship, Run, Any App Anywhere



From Dev

To Ops

Any App

Any OS

Windows

Linux

Anywhere

Physical

Virtual

Cloud

Some Docker vocabulary

Docker Image

The basis of a Docker container. Represents a full application

Docker Container

The standard unit in which the application service resides and executes

Docker Engine

Creates, ships and runs Docker containers deployable on a physical or virtual, host locally, in a datacenter or cloud service provider

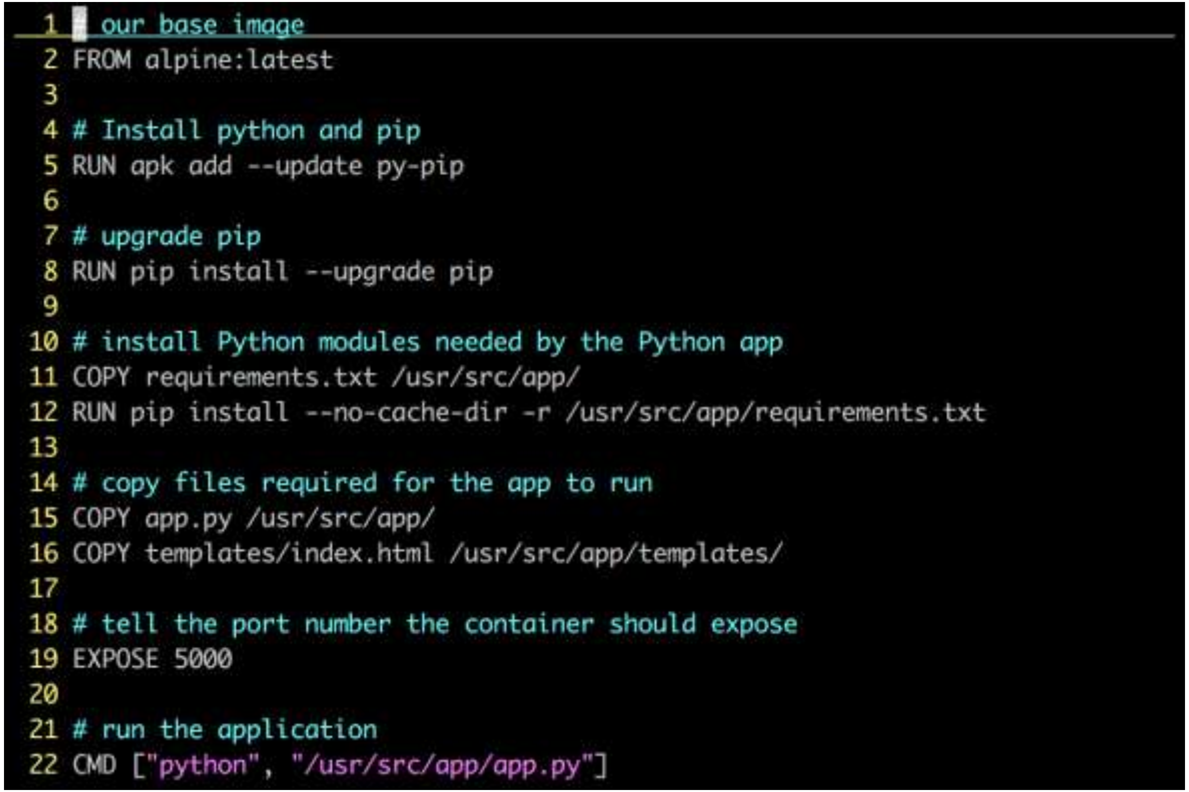
Registry Service (Docker Hub or Docker Trusted Registry)

Cloud or server based storage and distribution service for your images

Basic Docker Commands

```
$ docker pull mikegcoleman/catweb:latest

$ docker images

$ docker run -d -p 5000:5000 --name catweb mikegcoleman/catweb:latest

$ docker ps

$ docker stop catweb (or <container id>)

$ docker rm catweb (or <container id>)

$ docker rmi mikegcoleman/catweb:latest (or <image id>)
```

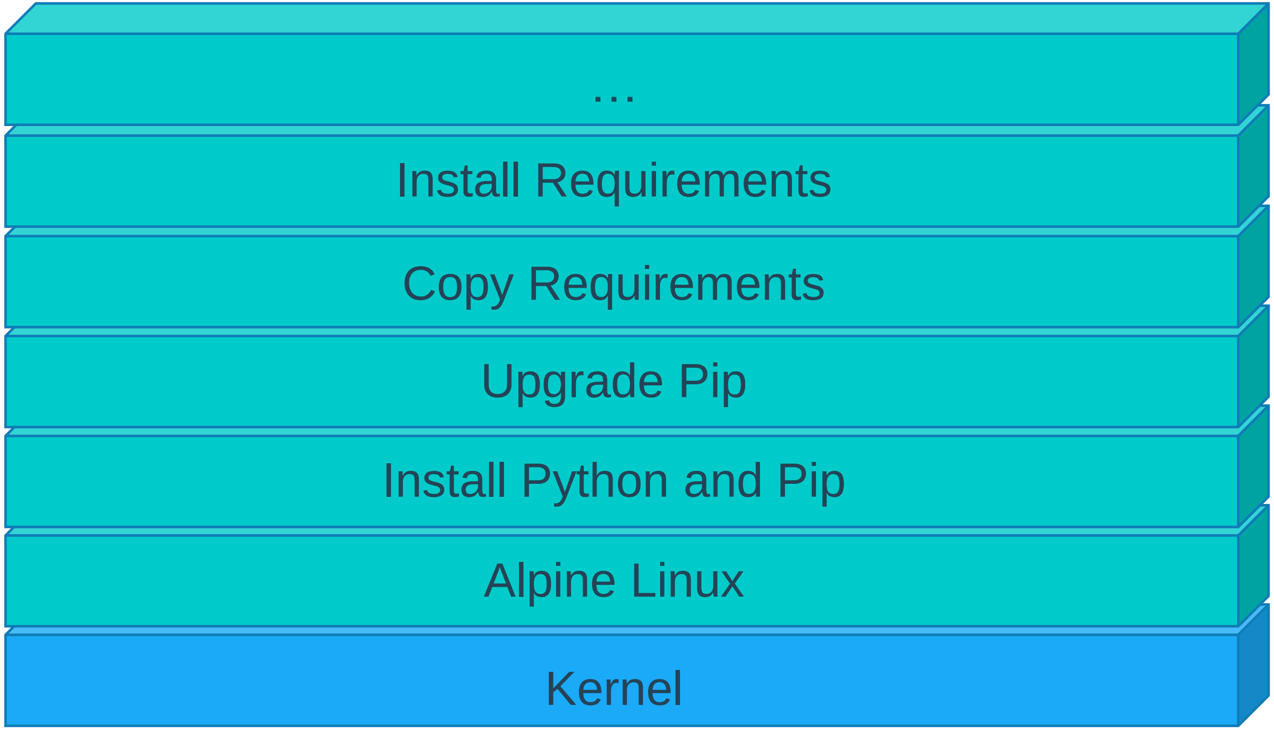
Dockerfile – Linux Example



```
# our base image
FROM alpine:latest

# Install python and pip
RUN apk add --update py-pip

# upgrade pip
RUN pip install --upgrade pip

# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

# copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

# tell the port number the container should expose
EXPOSE 5000

# run the application
CMD ["python", "/usr/src/app/app.py"]
```

- Instructions on how to build a Docker image
- Looks very similar to “native” commands
- Important to optimize your Dockerfile

Image Layers



...

Install Requirements

Copy Requirements

Upgrade Pip

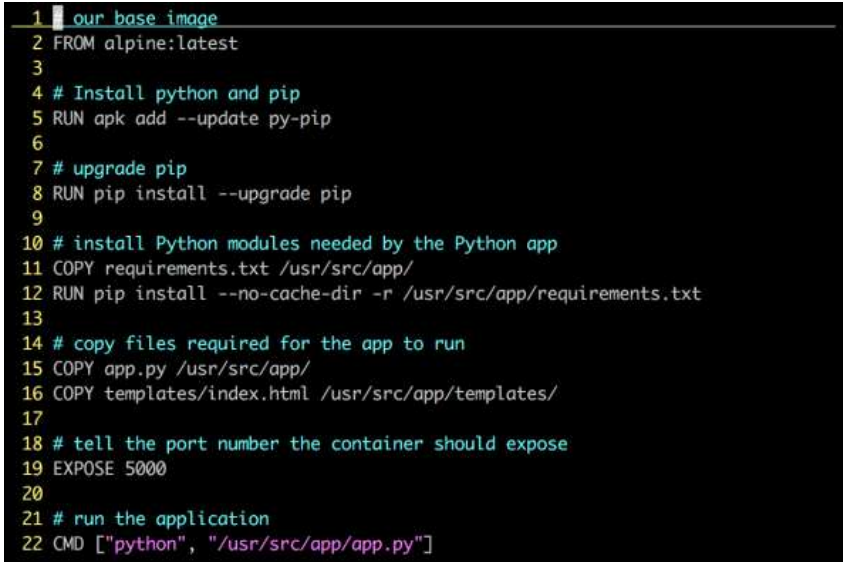
Install Python and Pip

Alpine Linux

Kernel

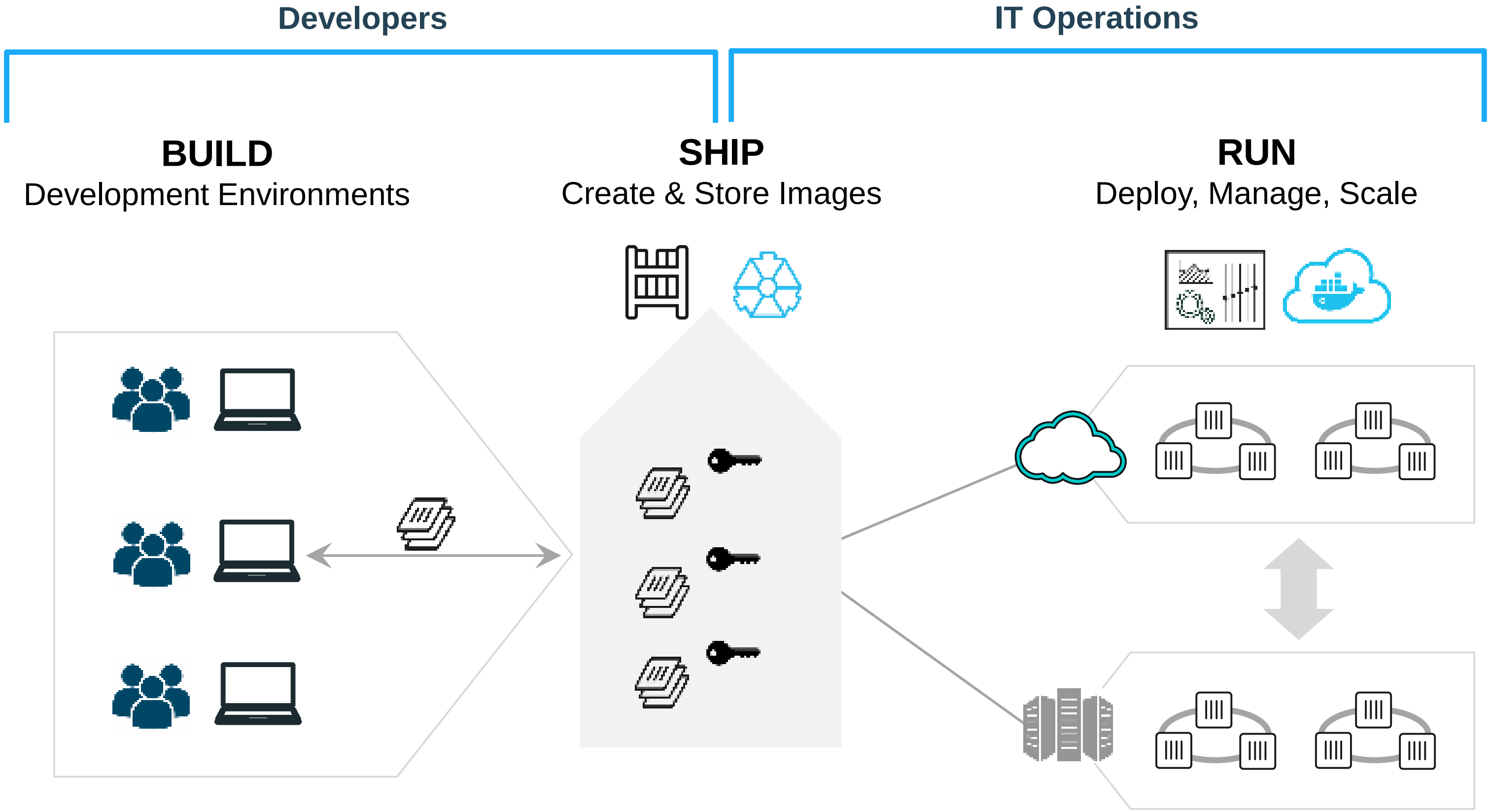
Basic Docker Commands

```
$ docker build –t mikegcoleman/catweb:2.0 .

$ docker push mikegcoleman/catweb:2.0
```



```
# our base image
FROM alpine:latest

# Install python and pip
RUN apk add --update py-pip

# upgrade pip
RUN pip install --upgrade pip

# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

# copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

# tell the port number the container should expose
EXPOSE 5000

# run the application
CMD ["python", "/usr/src/app/app.py"]
```

Put it all together: Build, Ship, Run Workflow



What about data persistence?

- Volumes allow you to specify a directory in the container that exists outside of the docker file system structure
- Can be used to share (and persist) data between containers
- Directory persists after the container is deleted
  - Unless you explicitly delete it
- Can be created in a Dockerfile or via CLI

WHAT IS DOCKER

- Allows you ship code along with all its dependencies in a self-contained manner
- Dockerfile like a manifest allows you to describe these dependencies and steps to set it up
- Spin up many instances of this image as you want (container)
- Cloud ready

WHY USE IT

- So many many libraries, so many many versions
- Dependency Install nightmare, be shielded from inadvertent upgrades
- Simplify and speed up focus on actual ML problem not supporting infrastructure

STEP 1

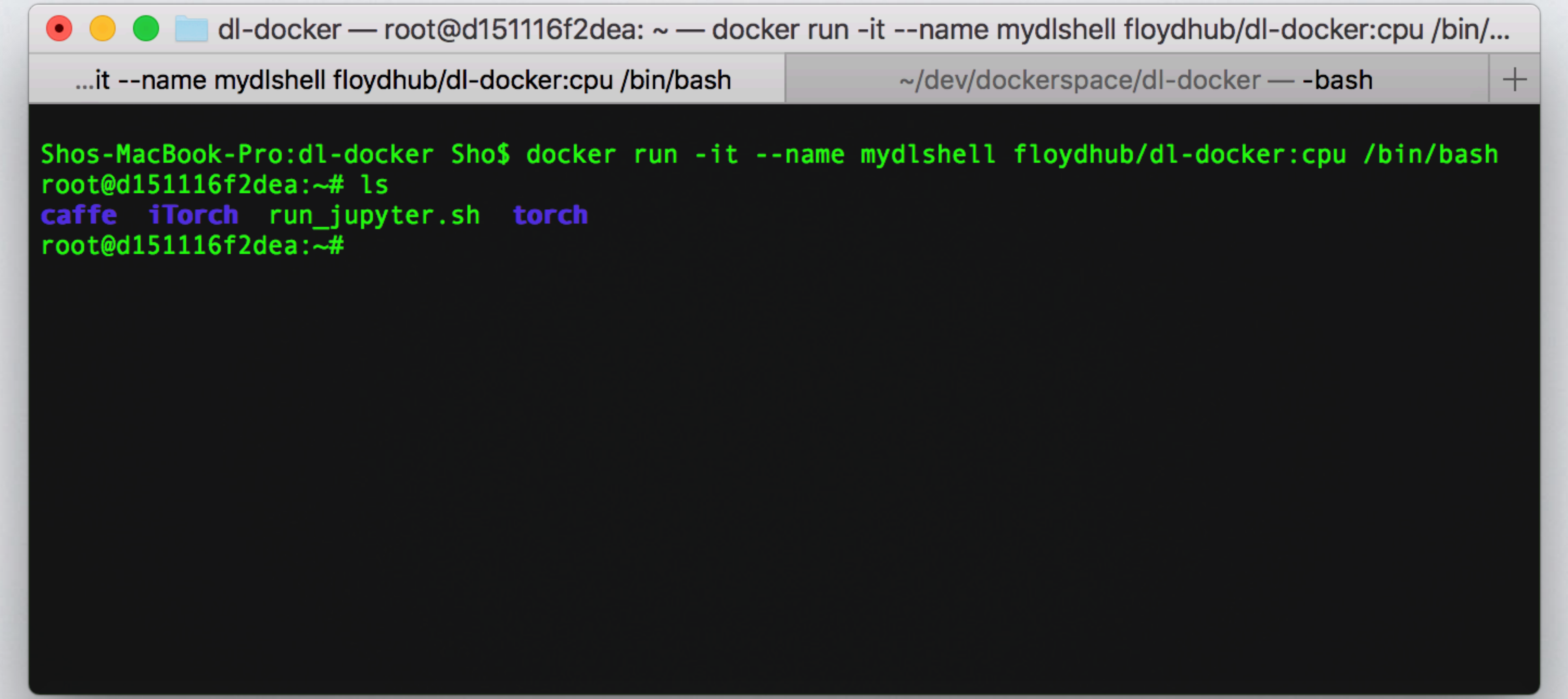
Download the image of choice from Docker Hub

```
$ docker pull floydhub/dl-docker:cpu
```

STEP 2

Start container with that image

$: docker run -it --name mydlshell floydhub/dl-docker:cpu /bin/bash



dl-docker — root@d151116f2dea: ~ — docker run -it --name mydlshell floydhub/dl-docker:cpu /bin/...
...it --name mydlshell floydhub/dl-docker:cpu /bin/bash
~/dev/dockerspace/dl-docker — -bash

```
Shos-MacBook-Pro:dl-docker Sho$ docker run -it --name mydlshell floydhub/dl-docker:cpu /bin/bash
root@d151116f2dea:~# ls
caffe  iTorch  run_jupyter.sh  torch
root@d151116f2dea:~#
```

STEP 2B

Another Way to Start Container … Using Assigned Label

```
$: docker start -ia mydlshell
```

STEP 3

Interact with the container to perform various tasks

Approach 1: Copy files into Container

```
$: docker cp ~/dev/dockerspace/census_keras.py dl-docker/ mydlshell:/root/test/census_keras.py
```

STEP 3B

Or Share a Volume (my preferred method)

```
$: docker run -it -v ~/dev/dockerspace/dl-docker:/projects/dl-docker --name
mydlspace floydhub/dl-docker:cpu
```

```
$: docker start mydlspace
```

```
$: docker exec -it mydlspace python /projects/dl-docker/census_keras.py
```

“HOW CAN IT BE THIS EASY ?”

Docker Compose

Defining and running multi-container Docker applications

What is Docker Compose?

1 A tool for defining and running multi-container Docker applications

2 With Compose, you use a YAML file to configure your application's services.

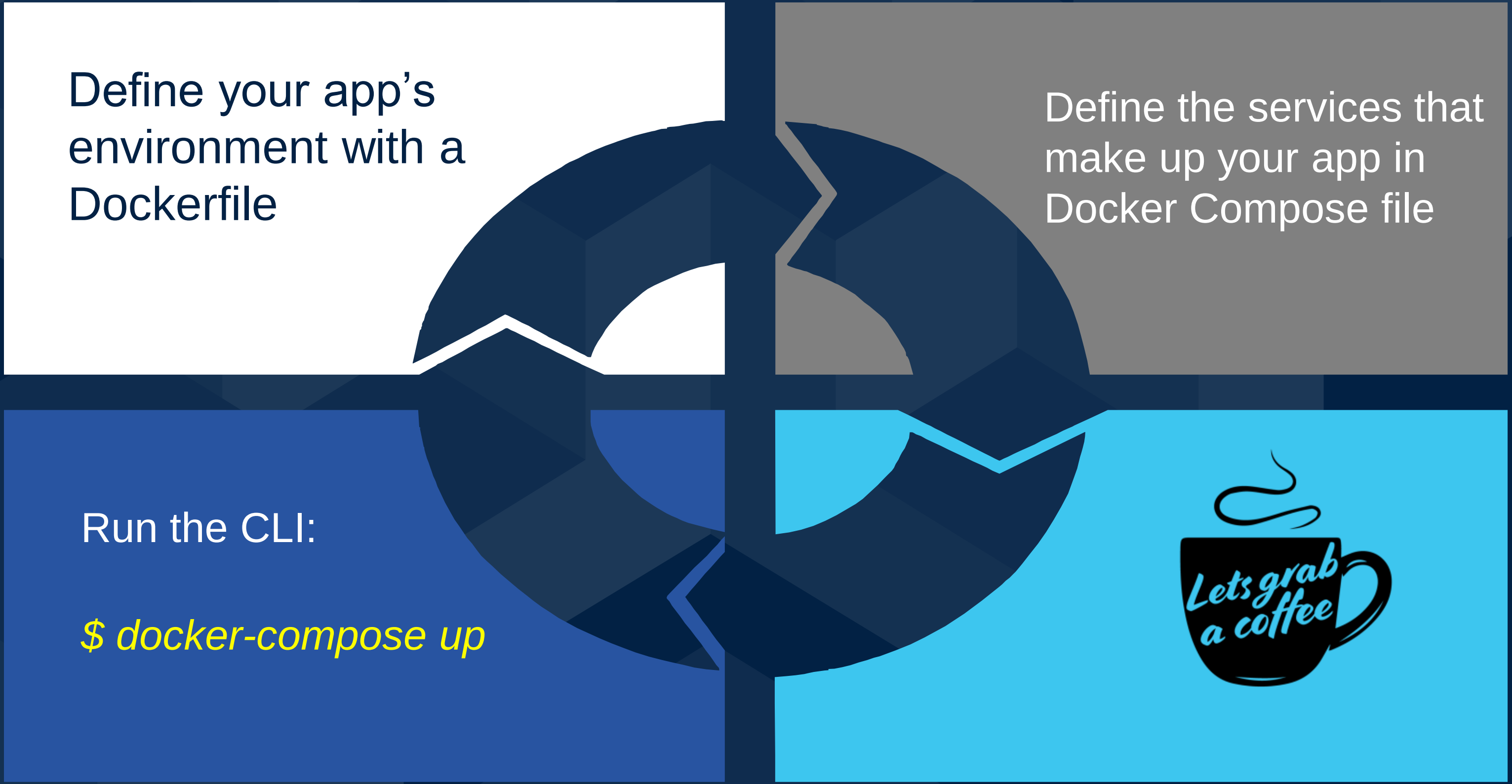
3 Compose works in all environments: production, staging, development, testing, as well as CI workflows.

4 With a single command, you create and start all the services from your configuration

BUT

- Binding to different ports on the host
- Setting environment variables differently
- Specifying a restart policy
- Adding extra services

Docker Compose is a 3 Steps Process



Define your app’s environment with a Dockerfile

Define the services that make up your app in Docker Compose file

Run the CLI:

$ docker-compose up

dockerfile

```
FROM python:2.7
ADD . /code
WORKDIR /code
RUN pip install -r requirements.txt
CMD python app.py
```

docker-compose.yml

```
web:
  build: .
  ports:
   - "5000:5000"
  volumes:
   - .:/code
  links:
   - redis
redis:
  image: redis
```

docker-compose up

```
$ docker-compose up
Pulling image redis...
Building web...
Starting composetest_redis_1...
Starting composetest_web_1...
redis_1 | [8] 02 Jan 18:43:35.576 # Server started, Redis version 2.8.3
web_1   |  * Running on http://0.0.0.0:5000/
```

docker compose cli

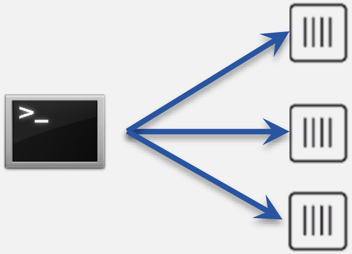
commands

```
build

logs

run

scale

up
```

Docker Compose: Multi Container Applications

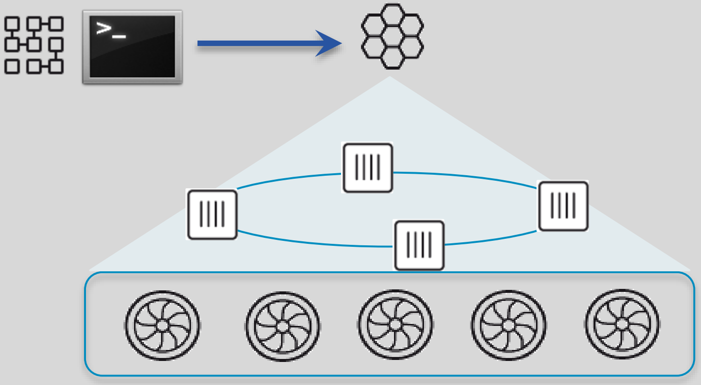
Without Compose

- Build and run one container at a time
- Manually connect containers together
- Must be careful with dependencies and start up order



With Compose

- Define multi container app in compose.yml file
- Single command to deploy entire app
- Handles container dependencies
- Works with Docker Swarm, Networking, Volumes, Universal Control Plane



Multiple container application in Docker

```
$ docker pull mysql

$ docker pull wordpress

$ docker run -d --name=db -e MYSQL_ROOT_PASSWORD=root mysql

$ docker run --name=wp -p 8000:80 --link db:db \
      -e WORDPRESS_DB_HOST=db \
      -e WORDPRESS_DB_PASSWORD=root wordpress
```

Docker Compose - YAML

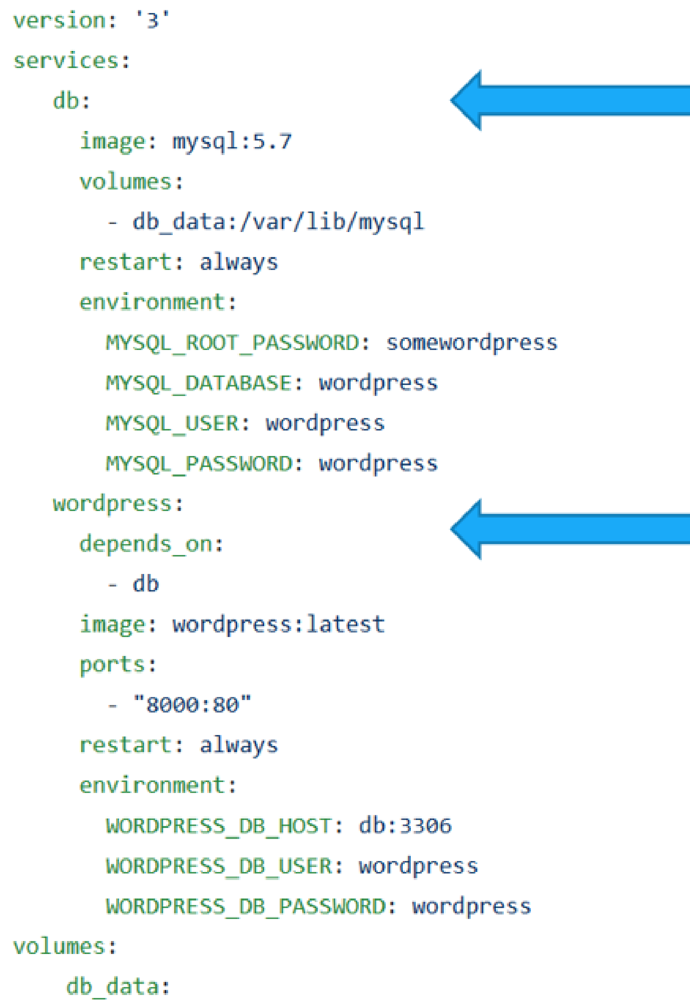
```
$ docker pull mysql

$ docker pull wordpress

$ docker run -d --name=db
        -e MYSQL_ROOT_PASSWORD=root mysql

$ docker run --name=wp  \
        -p 8000:80 \
        --link db:db \
        -e WORDPRESS_DB_HOST=db \
        -e WORDPRESS_DB_PASSWORD=root  \
        wordpress
```

```
version: '2'
services:
  db:
   image: mysql
   environment:
    MYSQL_ROOT_PASSWORD: root
  wp:
   depends_on:
    - db
   image: wordpress
   ports:
    - "8000:80"
   environment:
    WORDPRESS_DB_HOST: db
    WORDPRESS_DB_PASSWORD: root
```

Docker Compose - YAML

```
$ docker-compose up

$ docker-compose ps

$ docker-compose stop
```

```
version: '2'
services:
  db:
   image: mysql
   environment:
    MYSQL_ROOT_PASSWORD: root
  wp:
   depends_on:
    - db
   image: wordpress
   ports:
    - "8000:80"
   environment:
    WORDPRESS_DB_HOST: db
    WORDPRESS_DB_PASSWORD: root
```



```
version: '3'
services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: somewordpress
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD: wordpress
   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     restart: always
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
       WORDPRESS_DB_PASSWORD: wordpress
volumes:
    db_data:
```

Backend Service

Specify Volumes/Network

Environmental variables

Frontend Service

Specify Volumes/Network

Environmental variables