Docker Fundamentals

TEKsystems Global Services

WORKFORCE DEVELOPMENT

PARTICIPANT GUIDE

Data Persistence

Docker Container Volumes

Discussion

What are some examples where Container Data Persistence might be required?

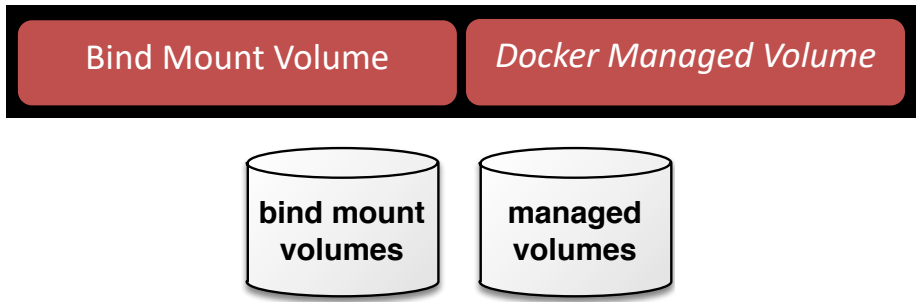
Discussion

What are some examples where Container Data Persistence might be required?

- Where Data Persistence is important:
  - Database Records (e.g. MySQL, MariaDB, Cassandra)
- Shared data from host to containers
  - build tools (e.g. Jenkins jobs)
  - automation tools (e.g. Ansible playbooks)
- Share data between containers
  - Container-A creates data, Container-B uses created data
- Access to Docker host's data
  - /var/log, /etc/hosts, /var/lib/, etc

Docker Volume Types

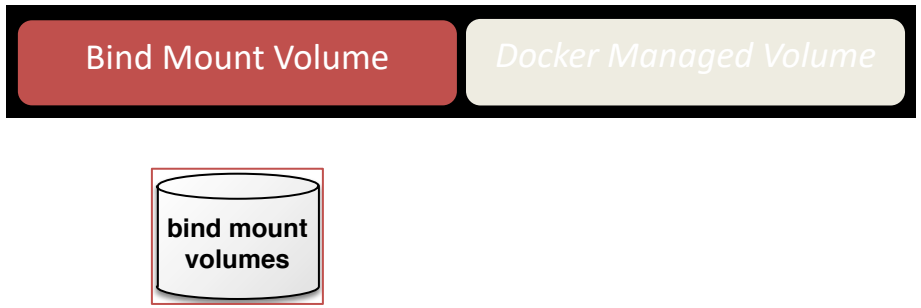
Volume Types



Provides Data:

- Accessibility by container and non-containers
- Persistence beyond a container's lifecycle
- Sharing between Containers
- Segmentation between Containers

Bind-mount volumes

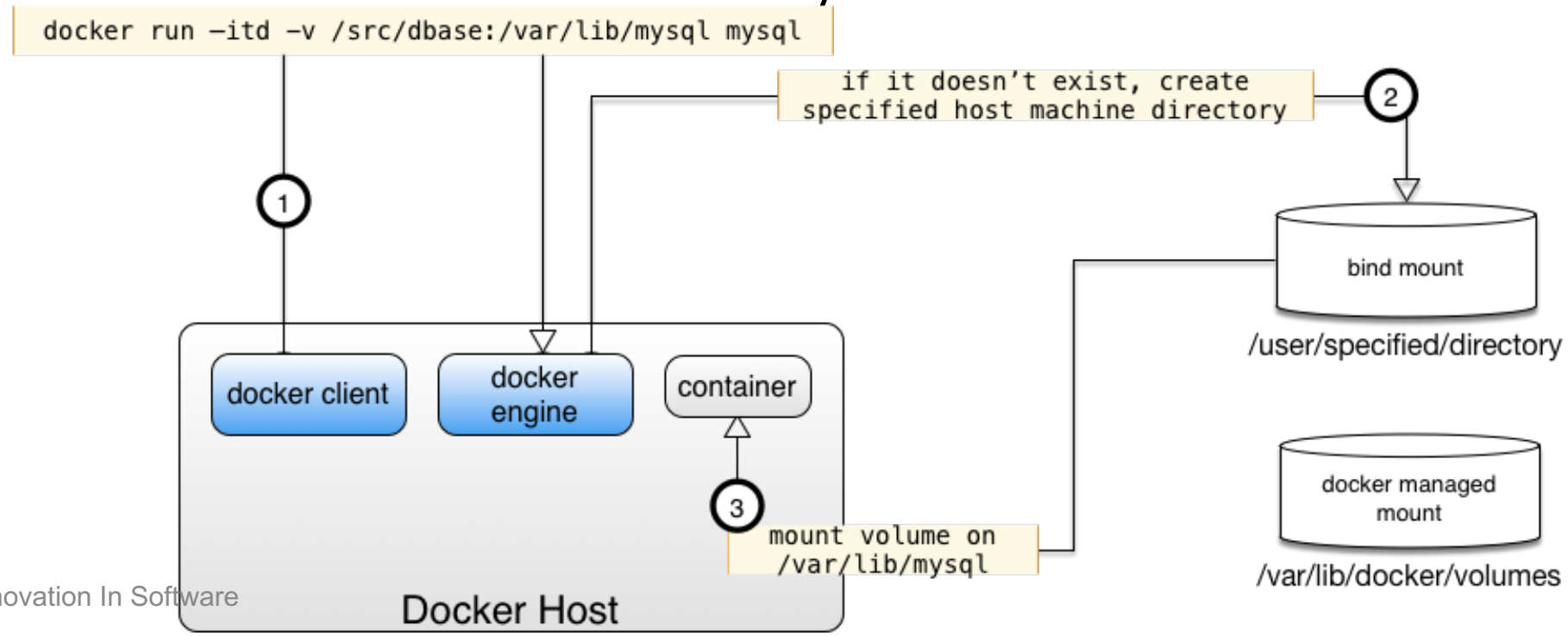


- User Defined
- Provides Data:
  - Accessible by containers and non-containers
  - Creates the directories, if required, in the volume path
  - Absolute paths must be used
  - Can not be automatically cleaned up by Docker-Engine
  - Managed by User

Bind-mount volumes

Bind-Mount Volume:

- user defined
- resides outside the Docker-managed space
- can be shared between many containers



Create a bind-mount volume

Create a bind-mount volume:

- Creates the /data, if it doesn't exist, mounts on container

```
$ docker run -itd -v /data:/data busybox top
```

Mounts are found in `docker inspect` output:

```
$ docker inspect $(docker ps -lq)
"Mounts": [
            {
                "Source": "/data",
                "Destination": "/data",
                "Mode": "",
                "RW": true
}
```

Create multiple bind-mount volumes

Create a bind-mount volume:

- Creates the directory, if it doesn't exist, mounts on container

```
$ docker run -itd --name share \
 -v /peanut:/peanut -v /butter:/butter \
 -v /jelly:/jelly busybox top
```

Mounts are found in `docker inspect` output:

```
$ docker inspect $(docker ps -lq)
"Mounts": [
            {
                "Source": "/peanut",
                "Source": "/butter",
                "Source": "/jelly",
}
```

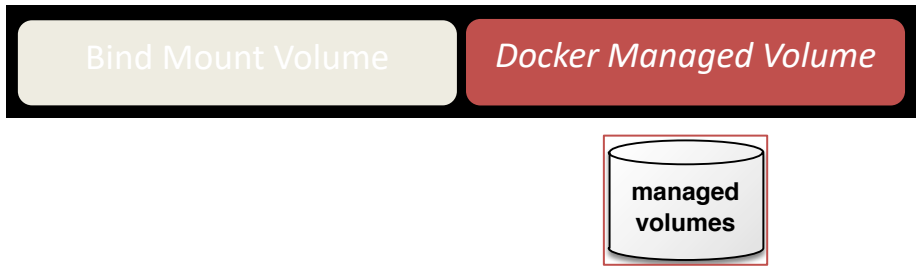
Create a bind-mount volume: new verbose method

Create a bind-mount volume using the new --mount option:

```
docker run --itd --mount type=bind,src=/data,dst=/data busybox top
```

The --mount option was originally only for swarm services but has been enabled for stand alone containers. The order of the arguments does not matter and it is easier to understand with the variable names. This is the suggested method for new docker containers by the official docker docs.

Docker managed volumes



- Docker Defined Mount Point
- Provides Data:
  - Accessible by container and non-containers
  - Creates the volume in /var/lib/docker/volumes/
  - Can be automatically cleaned up by Docker-Engine
  - Can be queried for orphaned volumes
  - Managed by Docker

```
$ docker run -itd -v data:/data busybox top
```

```
$ docker run -itd -v /data busybox top
```

Create a managed volume: new verbose method

Create a bind-mount volume using the new --mount option:

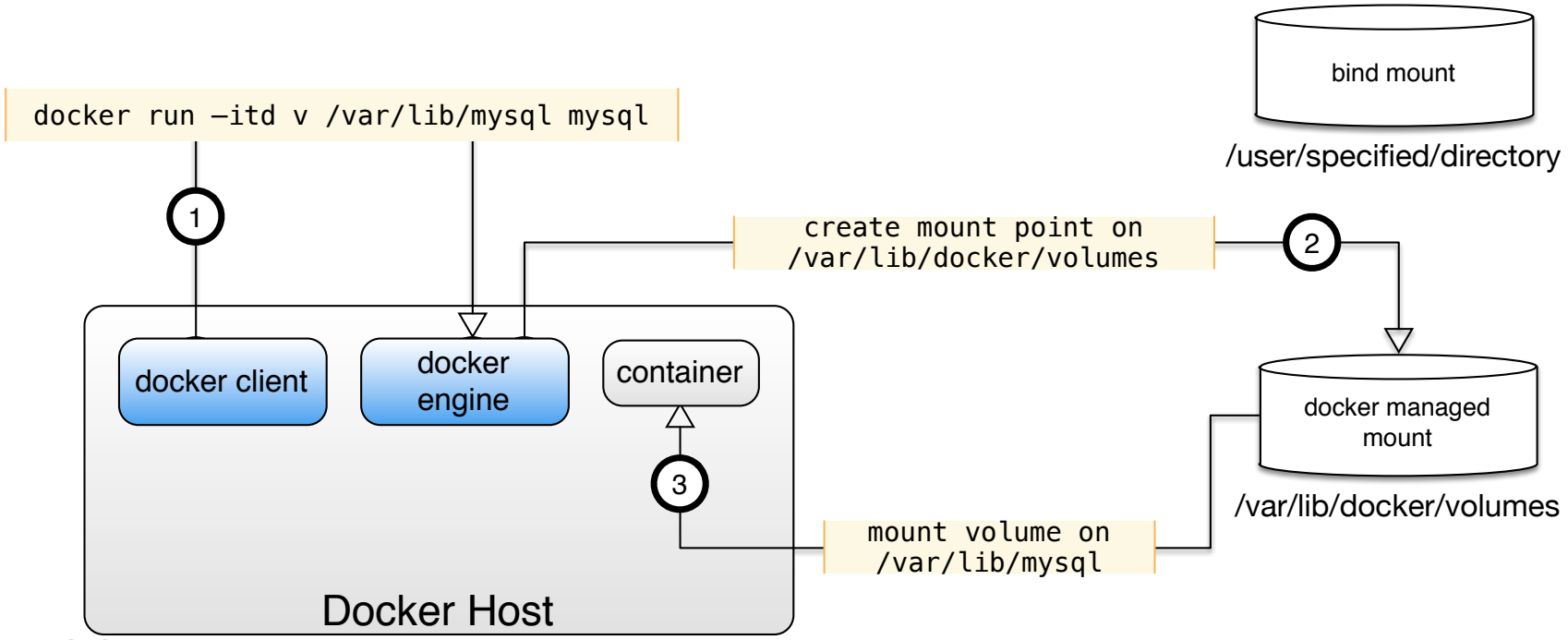
```
docker run --itd --mount src=data,dst=/data busybox top
```

The --mount option also works for Docker managed volumes.

Managed Volumes

Managed Volume:

- defined by docker-engine
- resides inside the Docker-managed space
- can be shared between many containers



Docker Volume– Use Cases

Docker Volume Use Cases

- Share Mount Points
- Container Troubleshooting
- Build Software on Mount Points
- Load Data from Images
- Data Protection

Scenario 1: Sharing Mount Points

Sharing Container mount points:

1. Copies the source container's mount-point endpoints
2. Can be viewed in the volume's metadata

Why use this?

- Sequential builds of content in the mount-point
- Sequential access to the content of the mount-point
- Data injection from a container of the mount-point
- Provides an easy method to migrate database
- Troubleshooting

Share Docker Volumes

Create volume mounts from a container:

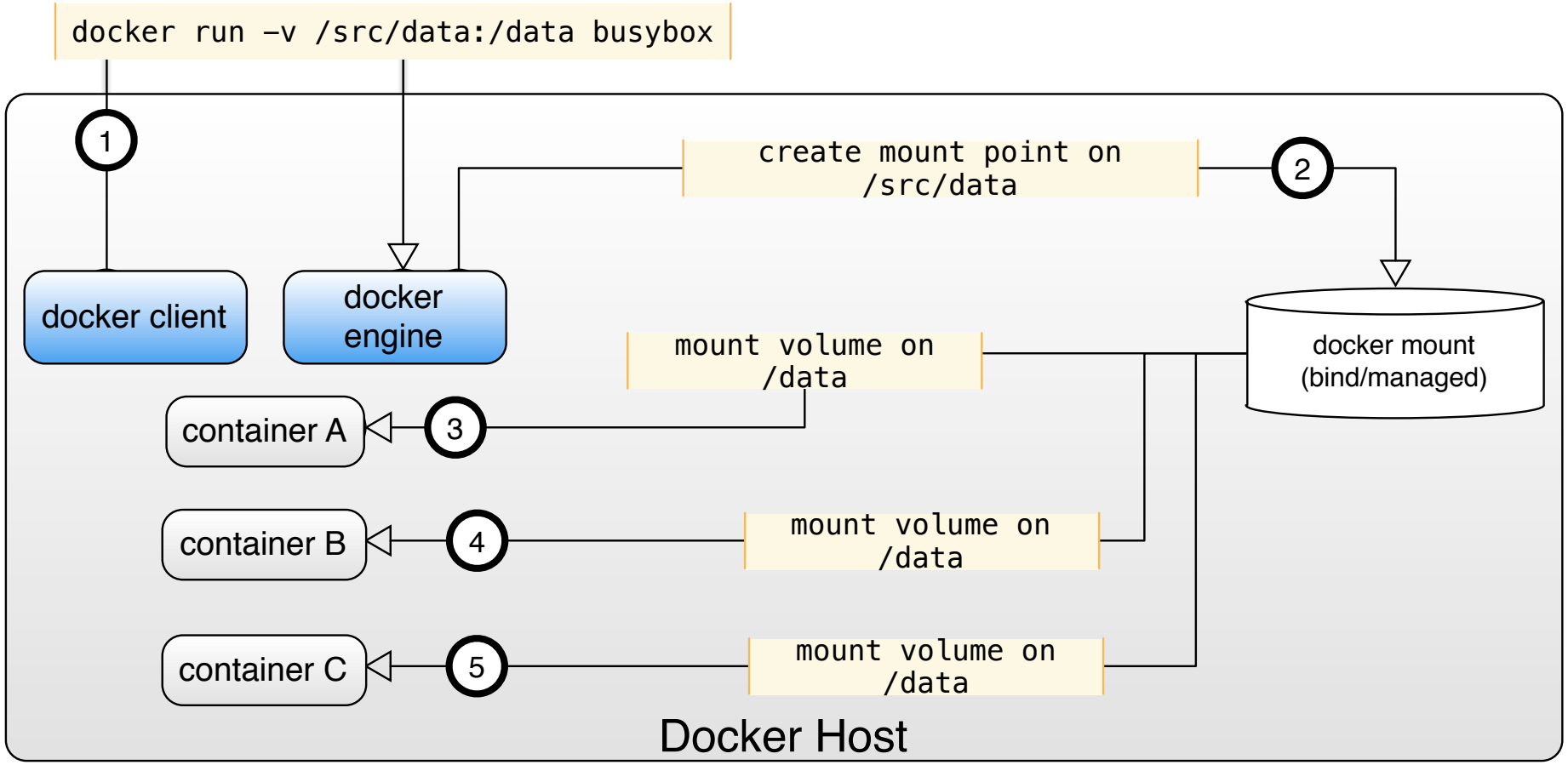
- Creates the directory, if it doesn't exist, mounts on container

```
$ docker run -itd --volumes-from share \
busybox top
```

Mounts is in `docker inspect` output:

```
$ docker inspect $(docker ps -lq)
"Mounts": [
            {
                "Source": "/peanut",
                "Source": "/butter",
                "Source": "/jelly",
}
```

Scenario 1: Sharing Mount Points



Scenario 2: Container Troubleshooting

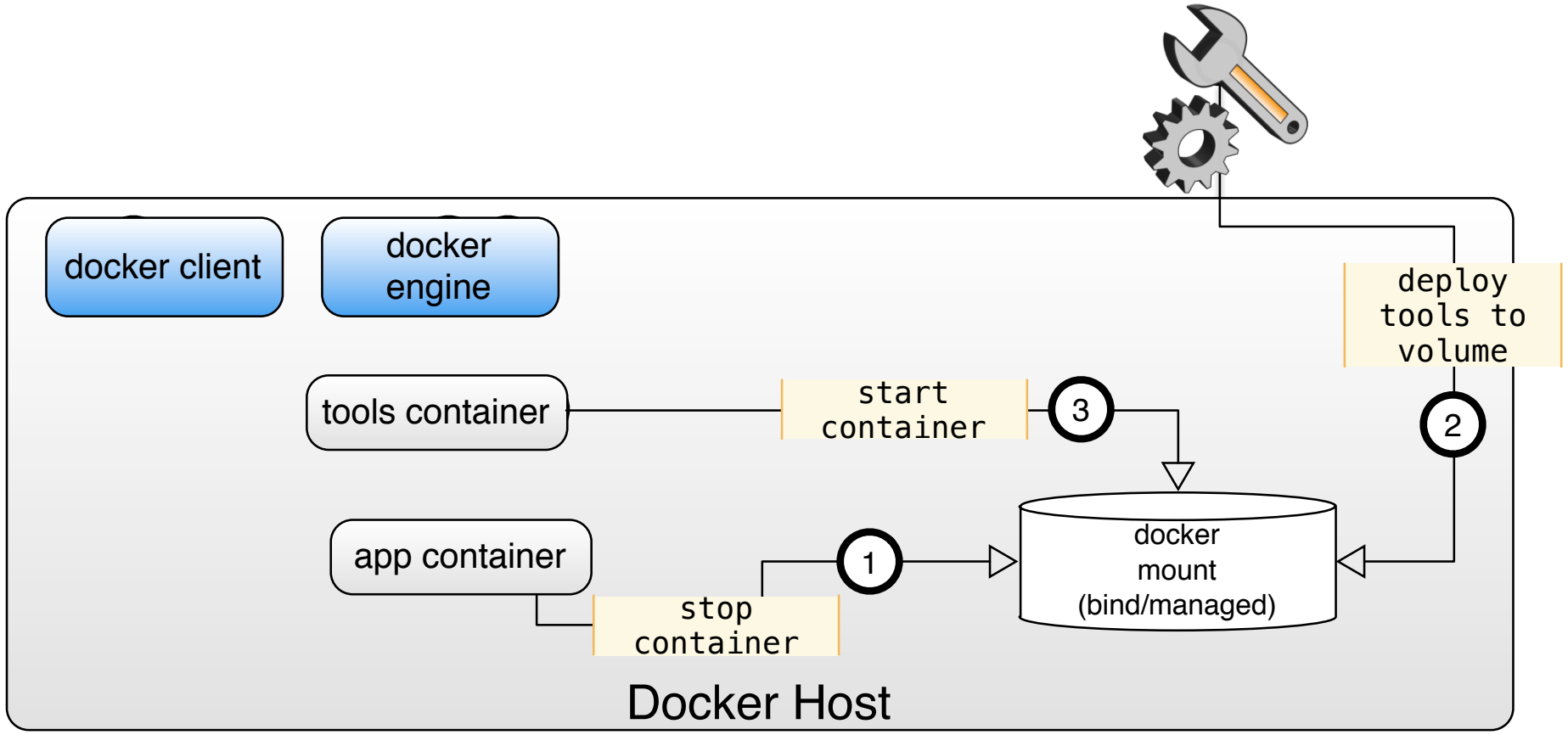
Troubleshooting containers using Volumes:

1. Stop the troublesome container
2. Copy tools into the volume
3. Run the original container or a second container

Why use this?

- Reduces the size of the app container's base image
- Allows the tools container access to the app's data
- Troubleshooting is made easier as additional tools don't need to be installed in the app's base image

Scenario 2: Container Troubleshooting



Scenario 3: Build Software on Mount Points

Building on volume mount-points:

1. First Build Container installs the application
2. Second Build Container configures the application
3. Third Container runs the application's software

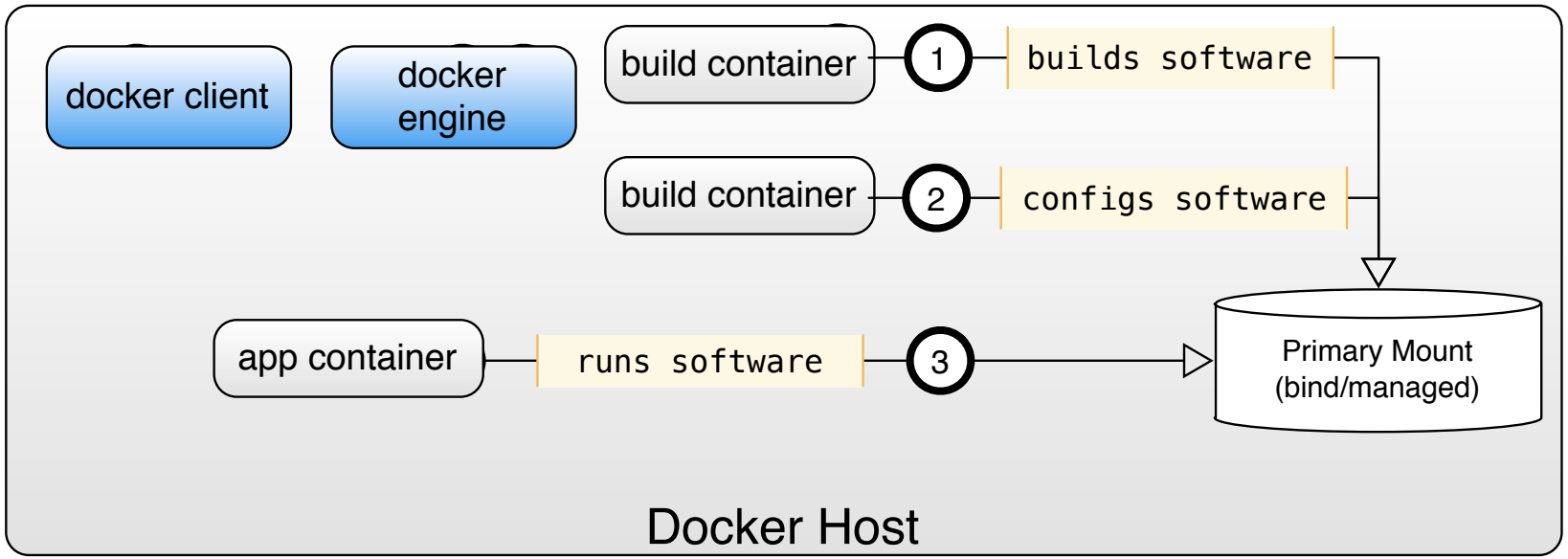
Why use this?

- Reduces the size of the app container's base image
- Provides a common software build
- Provides multiple configurations on the same base install

Scenario 3: Build Software on Mount Points

Using Volumes to build software:

- Multiple containers can interact with volumes
- Multiple applications can access the same data
- Containers can be aware of {{variables}} from previous build containers



Scenario 4: Load Data from Images

Load Data from Data Packed Containers:

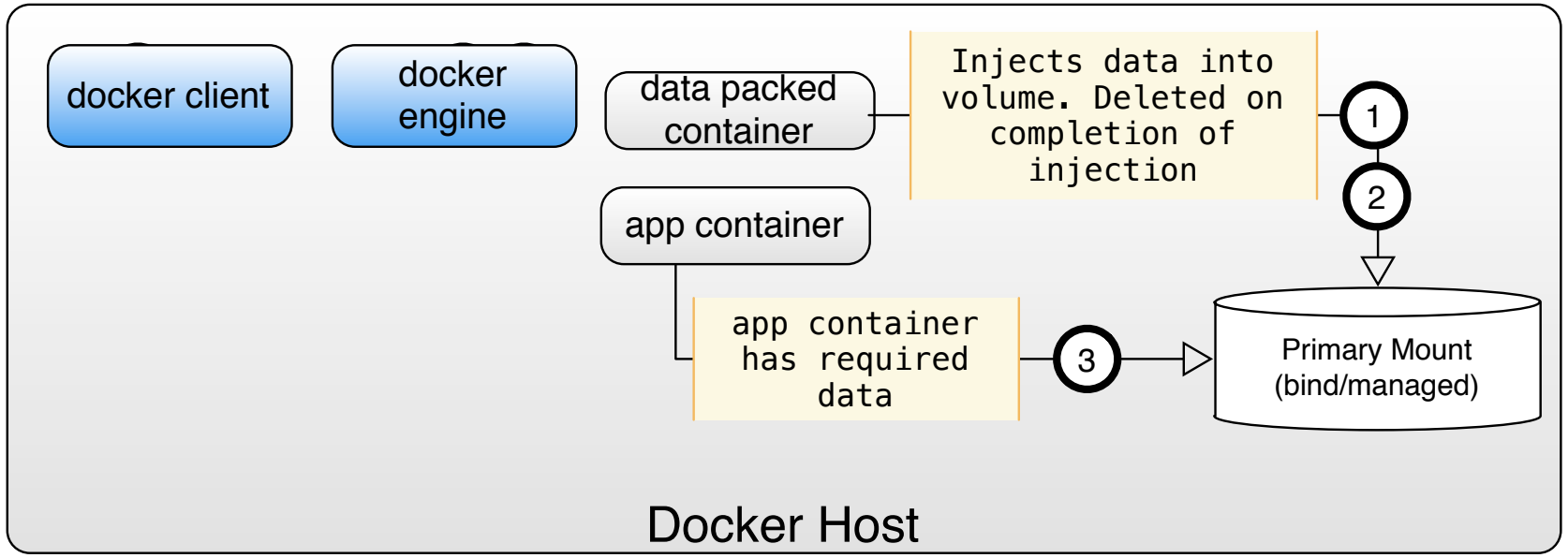
1. Images have a consistent start point
2. Copy tools into the volume
3. Run the original container or a second container

Why use this?

- Reduces the size of the app container’s base image
- Reusable consistent data from image
- Reduce overall image storage consumption
- Offload data writes to shared storage

Data packed container:

- Data Packed Container has data on images to inject
- Data Packed Container injects data into volume
- App Container is deployed and has injected data in volume
- Data Packed Container is automatically deleted



Scenario 5: Data Protection

Data Protection of Volumes:

1. Volumes can be backed up by regular host utilities
2. Volumes can be added to scheduled back ups for host
3. Volumes can be restored

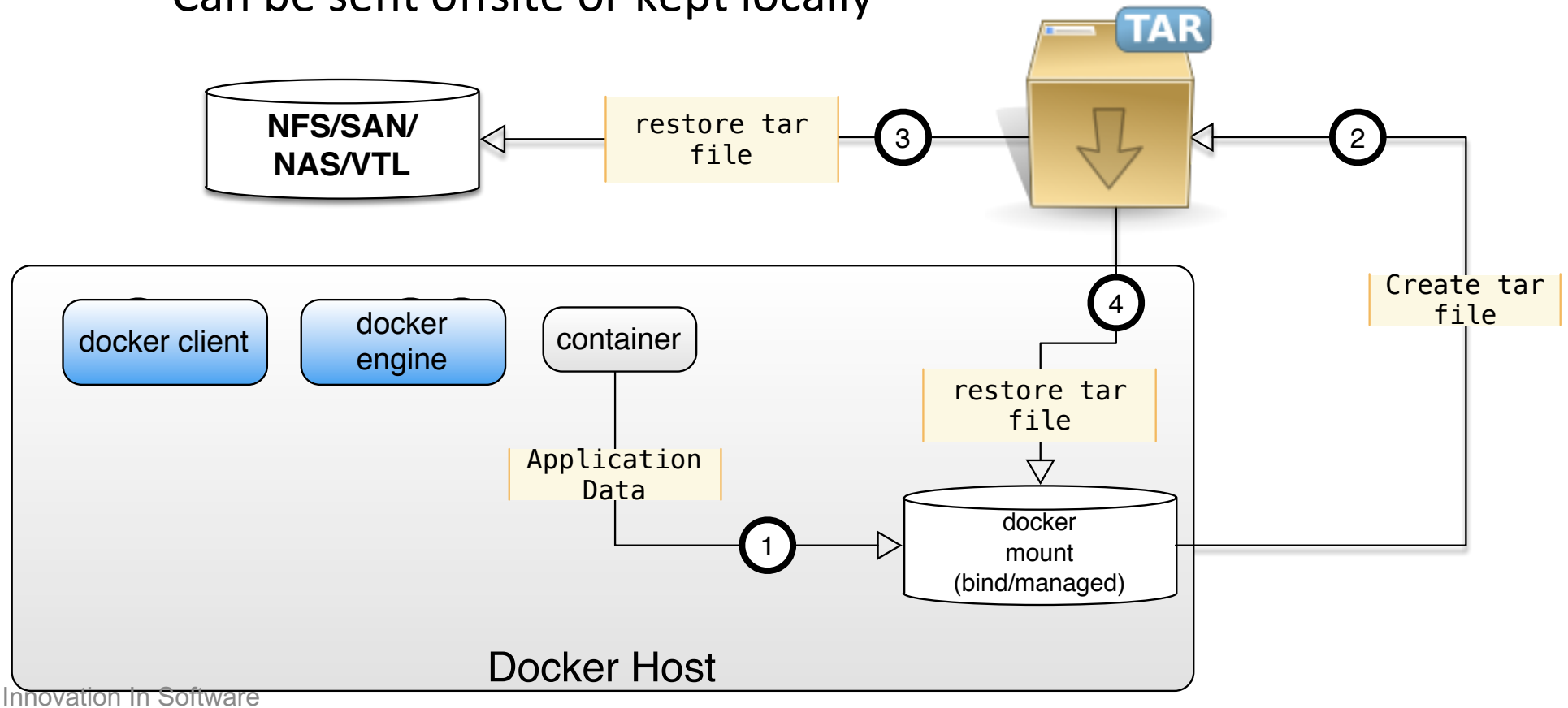
Why use this?

- App Data has:
  - a defined Recovery-Time-Objective
  - a defined Recovery-Point-Objective
  - regulatory requirements (e.g. SEC, BASEL)
  - importance to business functions

Volume backup using TAR

Data Protection using TAR:

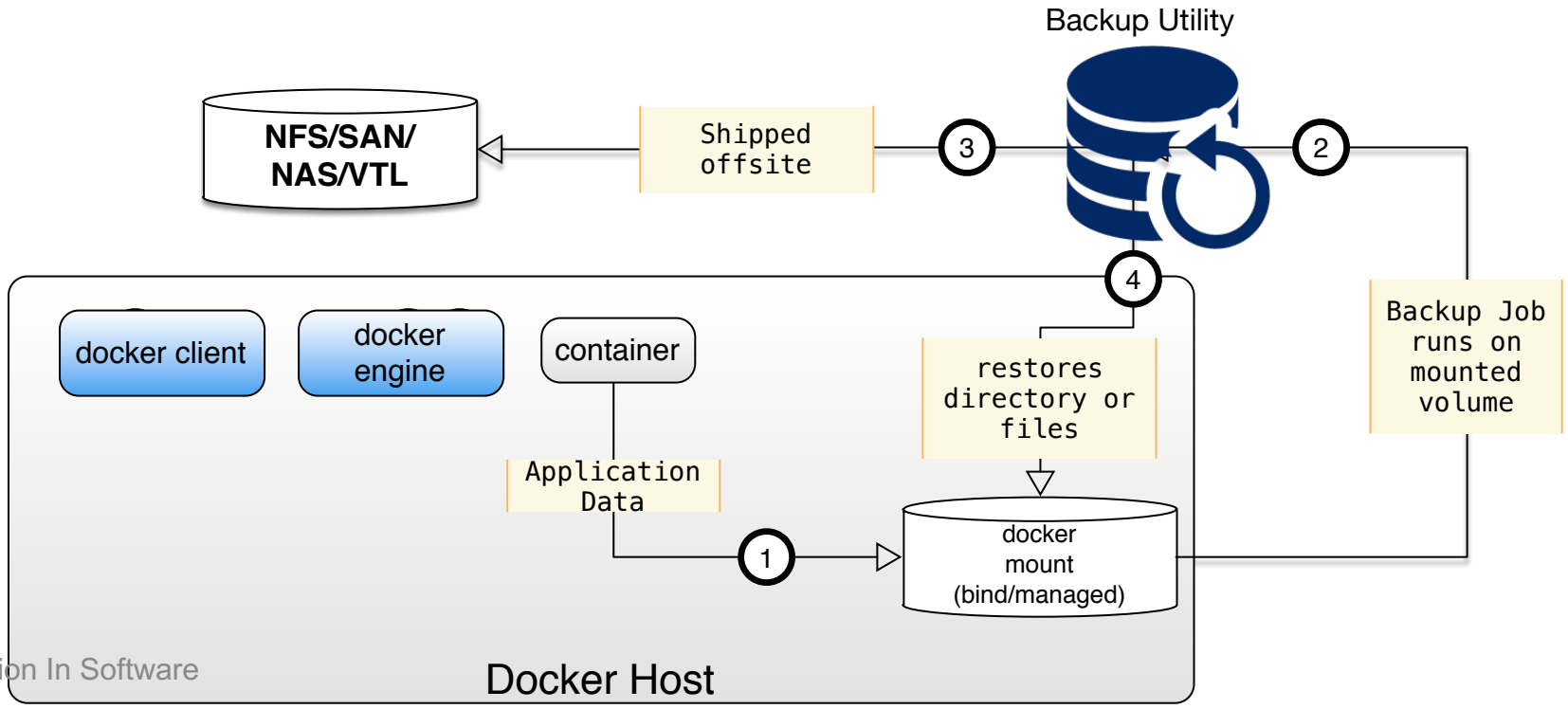
- Creates TAR of docker host's bind-mount volume (dir)
- Can be sent offsite or kept locally



Volume protection using backup utility

Data Protection using Backup Utility:

- mount-point added to backup schedule
- Backups stored locally and shipped offsite
- Backup Utility provides recoveries



Orphaned Volumes

Cleaning up volumes

- Attempts to clean up a volume

```
$ docker rm -v <container>
```

- If volume is in use, the volume is not deleted

Identify for dangling volumes:

```
$ docker volume ls -f dangling=true
```

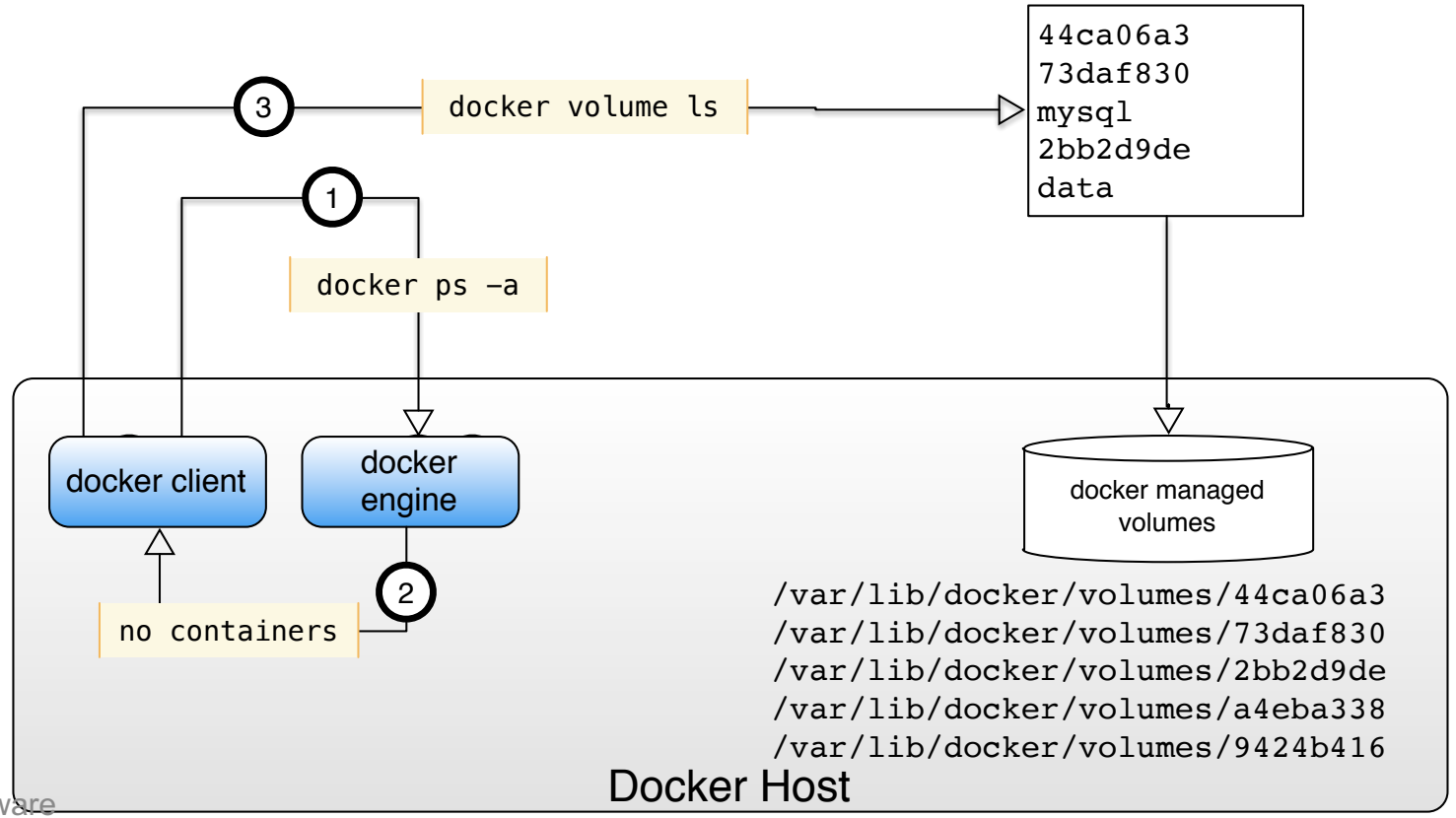
Orphaned Volume:

- Consumes storage space
- Data is no longer being used by a container
- Data may still be needed

Dangling Volumes

Dangling Volume:

- Orphaned volume
- Data is no longer being used



Docker Volume Metadata

Volume Metadata

What is it – volume metadata describes what the volume is through the use of key value pairs

- **docker volume inspect <id>**
  - Queries the Docker Engine, a json formatted output is returned

```
$ docker volume inspect <volume>
```

```
        "Name":
         "Driver":
         "Mountpoint":
         "Labels":
         "Scope":
```

Lab: Data Persistence

Docker Compose (Container Orchestration)

Provisioning Containers

Two ways to deploy containers:

Manual

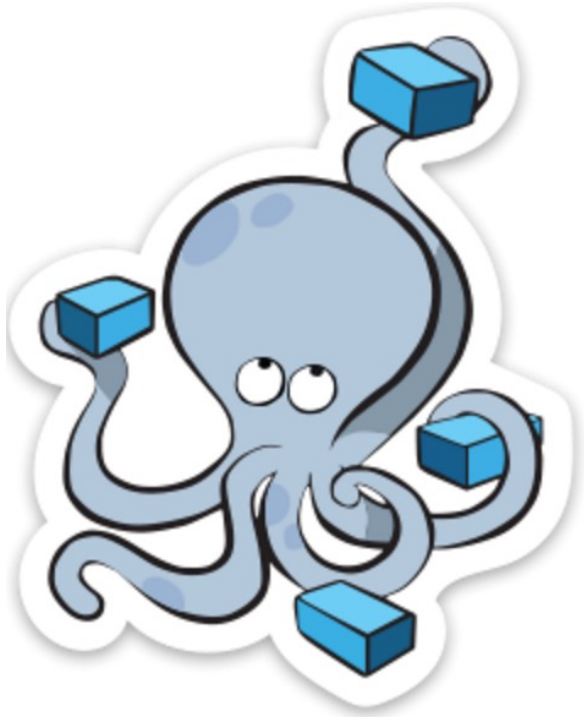
- Slow
- Doesn't Scale
- Error Prone
- Not what Docker was intended to do

Automated

- Fast
- Scales
- Repeatable
- Aligned with Docker's design intentions

Docker Compose

- Automates the building of applications (services, networks, and volumes) defined in a single file
- Each deployment is the same as the last
  - Compose files can be versioned along with the application source code



Docker Compose Overview

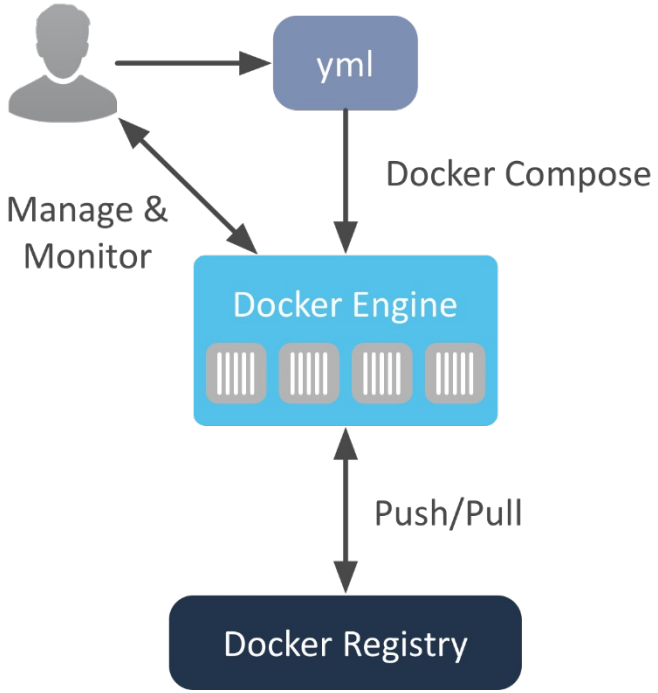
Docker Compose

Tool to create and manage multi-container applications

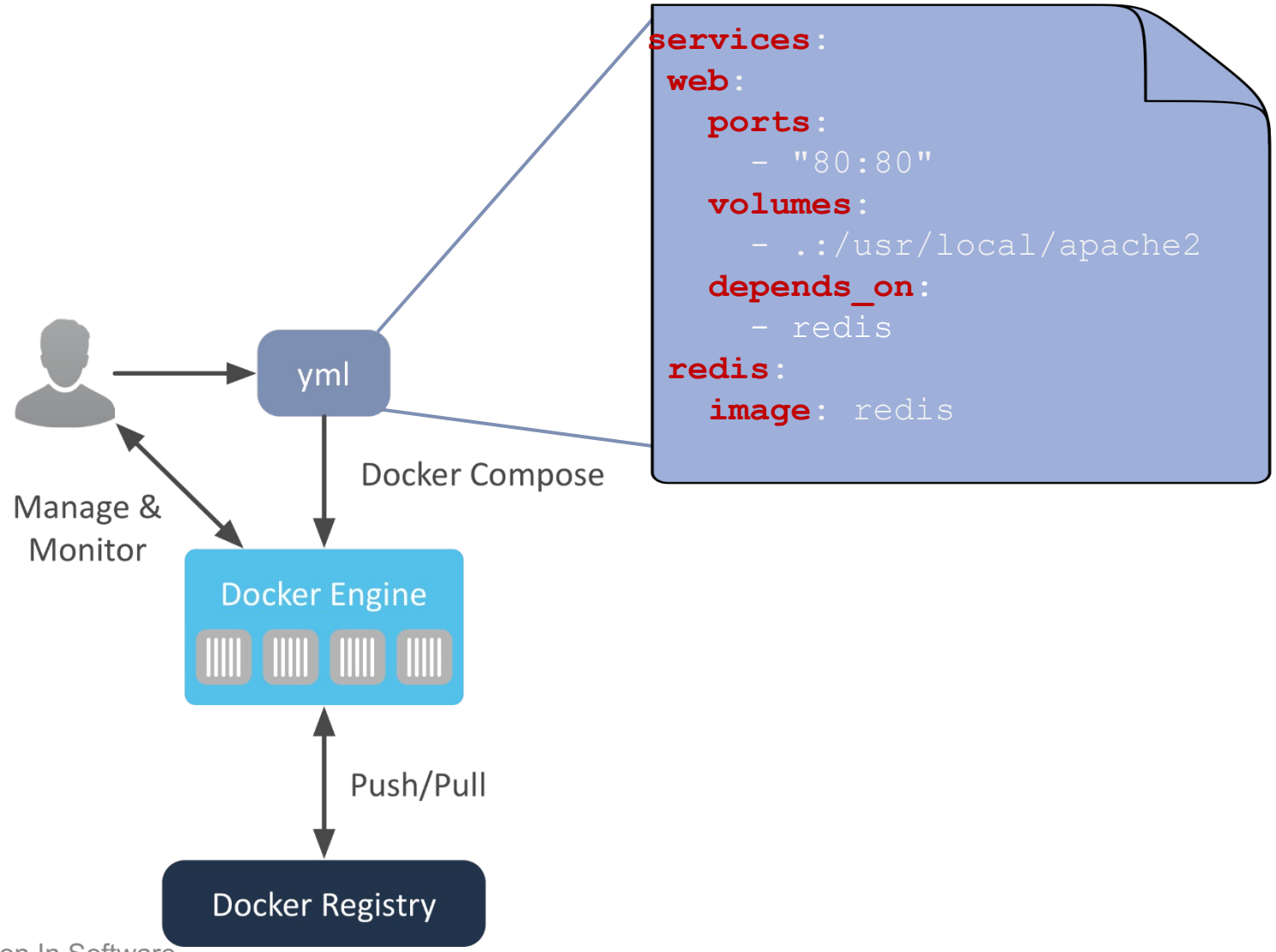
- Applications (services, networks, and volumes) defined in a single file:

  docker-compose.yml

- Transforms applications into individual containers that are linked together
- Compose will start all containers in a single command



Docker Compose



What is a Docker Composition

*A Docker Composition is a **YAML file** where all the top level keys are the names of a service and the values are the service definition.*

As with **`docker run`**, options specified in the Dockerfile are respected by default and do not need to be specified again in the **docker-compose.yml**

- **Default name and location for a compose file is:**

  ./docker-compose.yml

Example docker-compose.yml

```yaml
version: '2'

services:

  wordpress:
    image: wordpress
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_PASSWORD: example

  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: example
```


https://docs.docker.com/compose/compose-file/

Docker compose CLI

```
$ docker-compose --help
Define and run multi-container applications with Docker.
Usage:
  docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]
  docker-compose -h|--help

Options:
  -f, --file FILE             Specify an alternate compose file (default: docker-
compose.yml)
  -p, --project-name NAME     Specify an alternate project name (default:
directory name)
  --verbose                   Show more output
  -v, --version               Print version and exit
  -H, --host HOST             Daemon socket to connect to
```

Docker Compose Features

- Multiple isolated environments on a single host
  - Default project name is the basename of the project directory
  - Set a custom project name by using the **-p** command line option or the **COMPOSE_PROJECT_NAME** environment variable
- Preserve volume data between runs
- Recreate only changed containers
- Variable substitution

Docker Compose Features

- Multiple isolated environments on a single host
- Preserve volume data between runs
  - Finds any containers from previous runs
  - Copies the volumes from the old containers to the new containers
- Recreate only changed containers
- Variable substitution

Docker Compose Features

- Multiple isolated environments on a single host
- Preserve volume data between runs
- Recreate only changed containers
  - Caches the configuration used to create a container
- Variable substitution

Docker Compose Features

- Multiple isolated environments on a single host
- Preserve volume data between runs
- Recreate only changed containers
- Variable substitution
  - Supports variables in the Compose file
  - Can pass environment variables at runtime

Docker Compose Use Cases

Use Cases

- **Multiple development environments**
- Continuous Integration environments
- Shared Host environments
- Automated testing

Create multiple copies of a single environment

Example:

- Copy for
  - The stable branch
  - The current branch
  - Release candidate branch

Use Cases

- Multiple development environments
- **Continuous Integration environments**
- Shared Host environments
- Automated testing

On a shared build environment server

- Can set the project name prefix to a unique build number
- Keeps builds from interfering with each other

Use Cases

- Multiple development environments
- Continuous Integration environments
- **Shared Host environments**
- Automated testing

Allows projects that may use the same service names from interfering with each other

Use Cases

- Multiple development environments
- Continuous Integration environments
- Shared Host environments
- **Automated testing**

Run automated test scripts against a repeatable environment

Docker Compositions

Compose Service Definition

This is the top level key, this is the Service Definition

Each defined service must specify exactly one **image**

Other keys are optional and analogous to their docker run command-line counter parts

```
wordpress:
  image: wordpress:latest
  ports:
    - "443:443"
  environment:
    - ADMIN_PASS=xxx
mysql:
  image:mysql
  volumes:
    - mysql-data:/var/lib/mysql
  ports:
    - "3306:3306"
  links:
    - wordpress

<continued>
```

Compose Service Definition

ports: Publish a container's port(s) to the host

volumes: Bind mount a volume

environment: Set environment variables

devices: Add a host device to the container

dns: set custom DNS server

docker run --help for more information

```
mariadb:
  image: mariadb:latest
  ports:
    - "3306:3306"
  volumes:
    - wp-data:/var/wp-data
  environment:
    - DB_ADMIN_PASS=xxx
  devices:
    - "/dev/ttyUSB0:/dev/ttyUSB0"
  dns:
    - 8.8.8.8
    - 9.9.9.9
```

Linking multiple containers

Service: Wordpress

links:
Add link to another container. Provides a secure channel via which Docker containers can communicate with one another.

In this example, the link provides secure communication between the Wordpress plug-in and Wordpress application.

```
wordpress:
  image: wordpress:latest
  ports:
    - "443:443"
  environment:
    - ADMIN_PASS=xxx
mysql:
  image: mysql
  volumes:
    - mysql-data:/var/lib/mysql
  ports:
    - "3306:3306"
  links:
    - wordpress
```

Container Environment Variables

Service: Wordpress

environment:

Define environment variables set in container's runtime environment

```
wordpress:
  image: wordpress:latest
  ports:
    - "443:443"
  environment:
    - ADMIN_PASS=xxx
```

To see what environment variables are available to a service, run:

```
# $ docker-compose run SERVICE env
```

Using the wordpress service as an example, the following slide displays the output of the command:

Show Available Environment Variables

```
# $ docker-compose run wordpress env

# HOSTNAME=f836bf313aed
# TERM=xterm
# CA_CERTIFICATES_JAVA_VERSION=20140324
# PATH=/usr/share/wordpress/bin:/usr/local/sbin:/usr/
  local/bin:/usr/sbin:/usr/bin:/sbin:/bin
# PWD=/
# JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre
# LANG=C.UTF-8
# JAVA_VERSION=8u66
# SHLVL=0
# HOME=/root
# JAVA_DEBIAN_VERSION=8u66-b17-1~bpo8+1
```

Environment Variable Substitution

- You are able to pass variable values from the shell environment to docker-compose
- The environment variable POSTGRES_VERSION=9.4 might be set within the shell. You could then supply the following configuration:

```
db:
  image: "postgres:${POSTGRES_VERSION}"
```

Docker Compose CLI

Running in Detached Mode

Running **`docker-compose`** with **`-d`** runs the Docker containers in the background

```
# $ docker-compose up
# 02multiple_web_1 is up-to-date
# Attaching to web_1
# web_1 | => Configuring NGINX ...
# web_1 | => ... Starting Services
# Gracefully stopping... (press Ctrl+C
  again to force)
# Stopping 02multiple_web_1 ... done
# $ docker-compose up -d
# Starting 02multiple_web_1
# $
```

Displaying logs for docker-compose

`docker-compose logs` displays log output from services

```
# $ docker-compose logs
# Attaching to web_1
# web_1 | => Configuring NGINX ...
# web_1 | => Using default NGINX
  configuration
# web_1 | => Configuring PHP-FPM ...
# web_1 | => Done!
# web_1 | => ... Starting Services
```

Scaling docker-compose containers

`docker-compose scale [SERVICE=NUM]`
sets the number of containers for a service

```
# $ docker-compose scale web=3
# Creating and starting 2 ... done
# Creating and starting 3 ... done
# $ docker-compose scale web=1
# Stopping 02multiple_web_2 ... done
# Stopping 02multiple_web_3 ... done
# Removing 02multiple_web_3 ... done
# Removing 02multiple_web_2 ... Done
# $
```

Running a Selected Single Service

`docker-compose run`, runs a one-time command against a service

Docker-compose run overrides the commands specified in the `docker-compose` service configuration

```
# $ hostname
# docker-workstation
# $ docker-compose run web bash
# [root@9ad9eec462fd /]# hostname
# 9ad9eec462fd
# [root@8f3f8001f8df /]# exit
# exit
# $ hostname
# docker-workstation
```

Stopping a Selected Single Service

`docker-compose stop SERVICE` shuts down the service gracefully

`docker-compose kill SERVICE` immediately kills the service

```
# $ docker-compose run --rm -d web
# Starting web_1
# $ docker-compose kill web
# Killing web_1 ... done
# $ docker-compose ps
# Name    Command  State    Ports
# web_1   /bin/run Exit     137
# $
```

Lab: Compose

Docker Swarm Overview

Docker Swarm Features

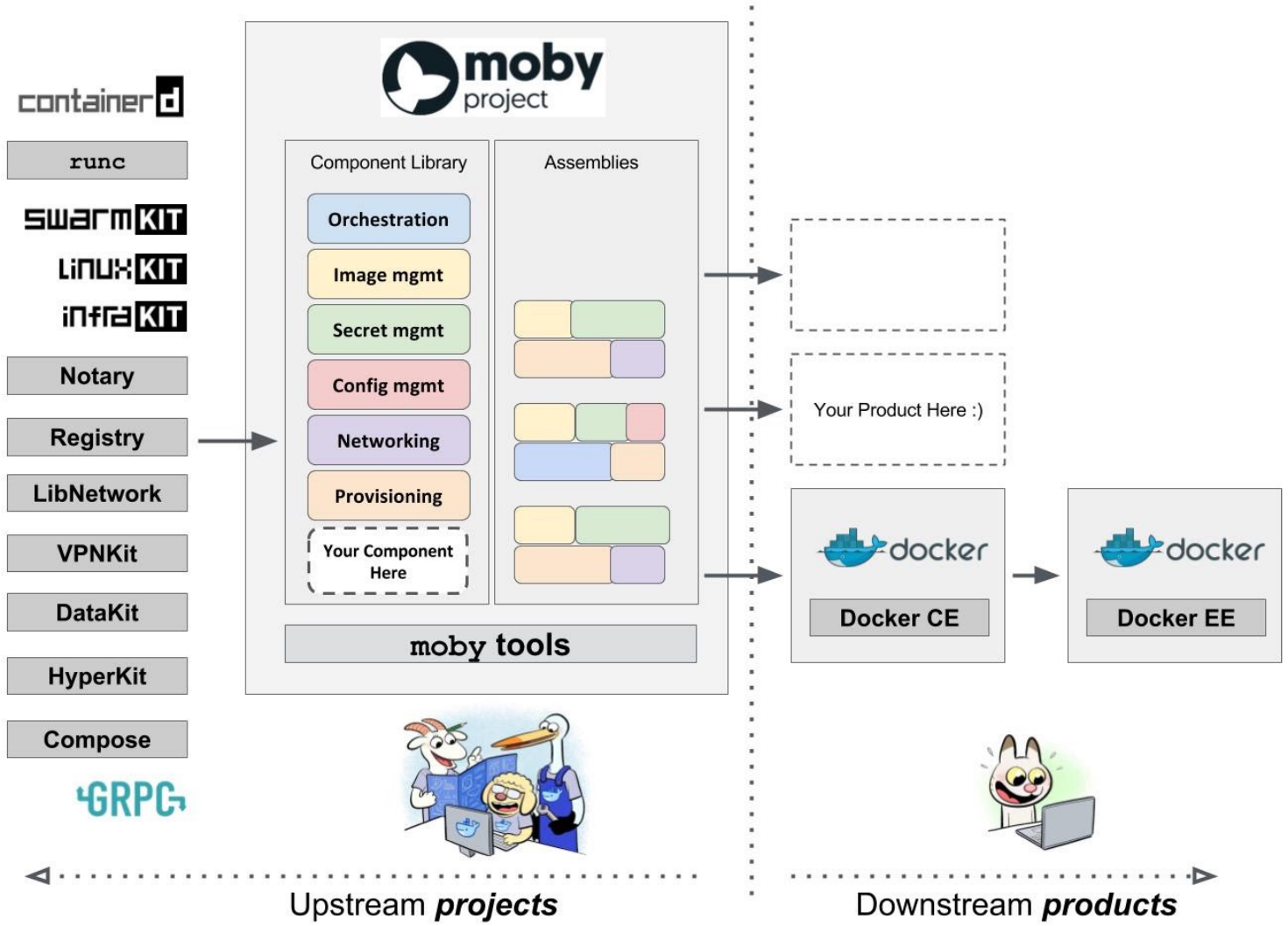
- Cluster management integrated in Docker Engine
- Decentralized Design
- Declarative Service Model
- Scaling
- Desired State Reconciliation
- Multi-Host Networking
- Service Discovery
- Load Balancing
- Secure by Default
- Rolling Updates

Docker Swarm

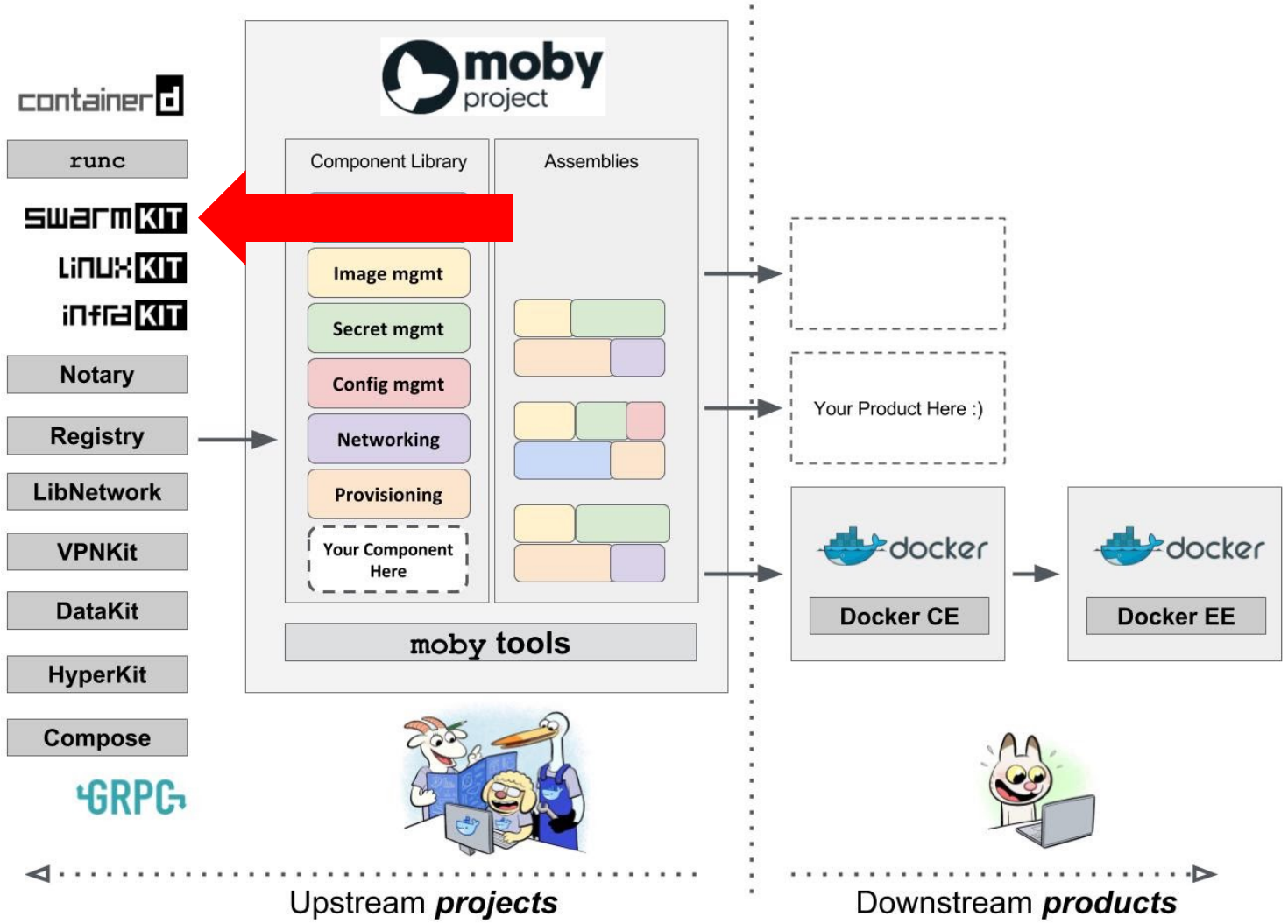
What is a Swarm?

- A Swarm is a cluster of Docker Engines (nodes) where services are deployed.
- ***Cluster management*** and ***orchestration*** are built into Docker Engine using the Swarm Kit.
- Swarm mode is enabled using ***swarm init*** or ***swarm join***.
- Runs ***Container commands*** and ***Swarm commands*** at the same time
- Swarm mode enables the orchestration of services

Docker Swarm



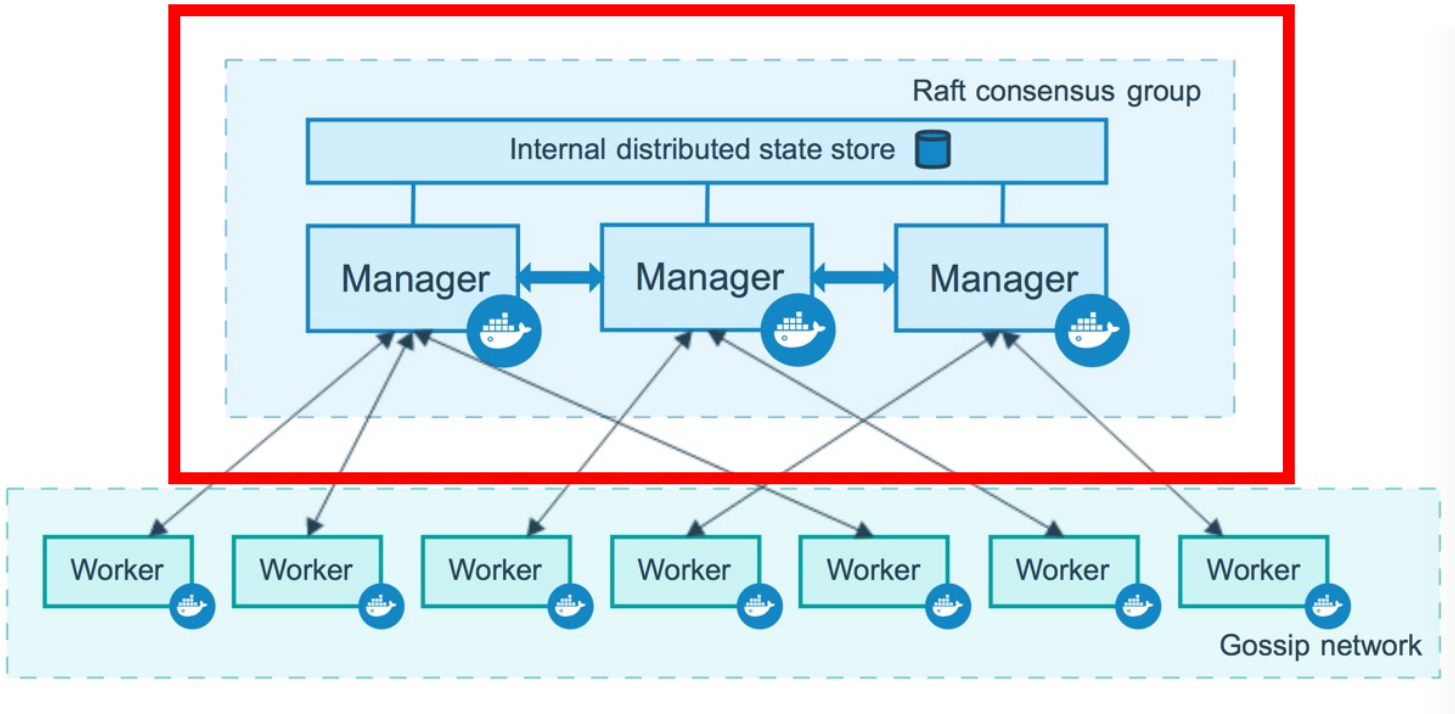
Docker Swarm



Docker Swarm (Swarm Nodes)

Manager Nodes

- Maintains Cluster State
- Schedules Services
- Serves up swarm mode HTTP API endpoints



Docker Swarm (Nodes)

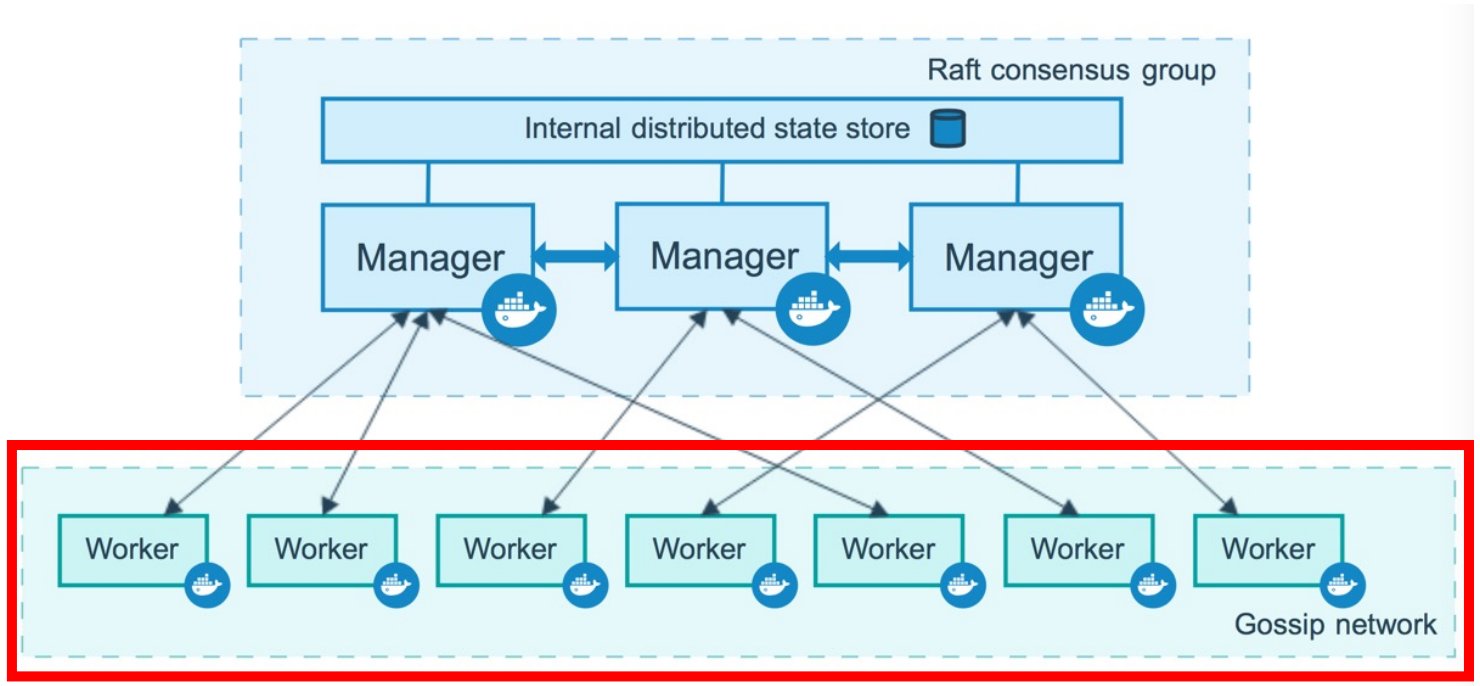
What are Nodes?

- An instance of the Docker Engine running in Swarm Mode
- Can run standalone (is that really a cluster?) or distributed across many machines
- Nodes are manager nodes or worker nodes
- Manager Nodes elect a single “Leader”
- Manager Nodes dispatch tasks to worker nodes
- Worker Nodes receive and execute tasks
- A Manager Node can also be a Worker Node

Docker Swarm (Worker Nodes)

Worker Nodes

- Executes containers (**Tasks**)
- Serve the swarm mode HTTP API
- Requires a Manager Node to function



Docker Swarm (Services and Tasks)

What are Services and Tasks?

- Service is a definition of the tasks to execute on the workers
- Primary root of user interaction with Swarm
- Two Types of **Services**:
  - Replicated Services - Distributed to a specific number of workers
  - Global Services - Distributed to all workers
- A **Task** is the atomic scheduling unit of swarm
- A **Task** can only run on one worker node at anytime
- A **Task** can only run or fail

Docker Swarm (Load Balancing)

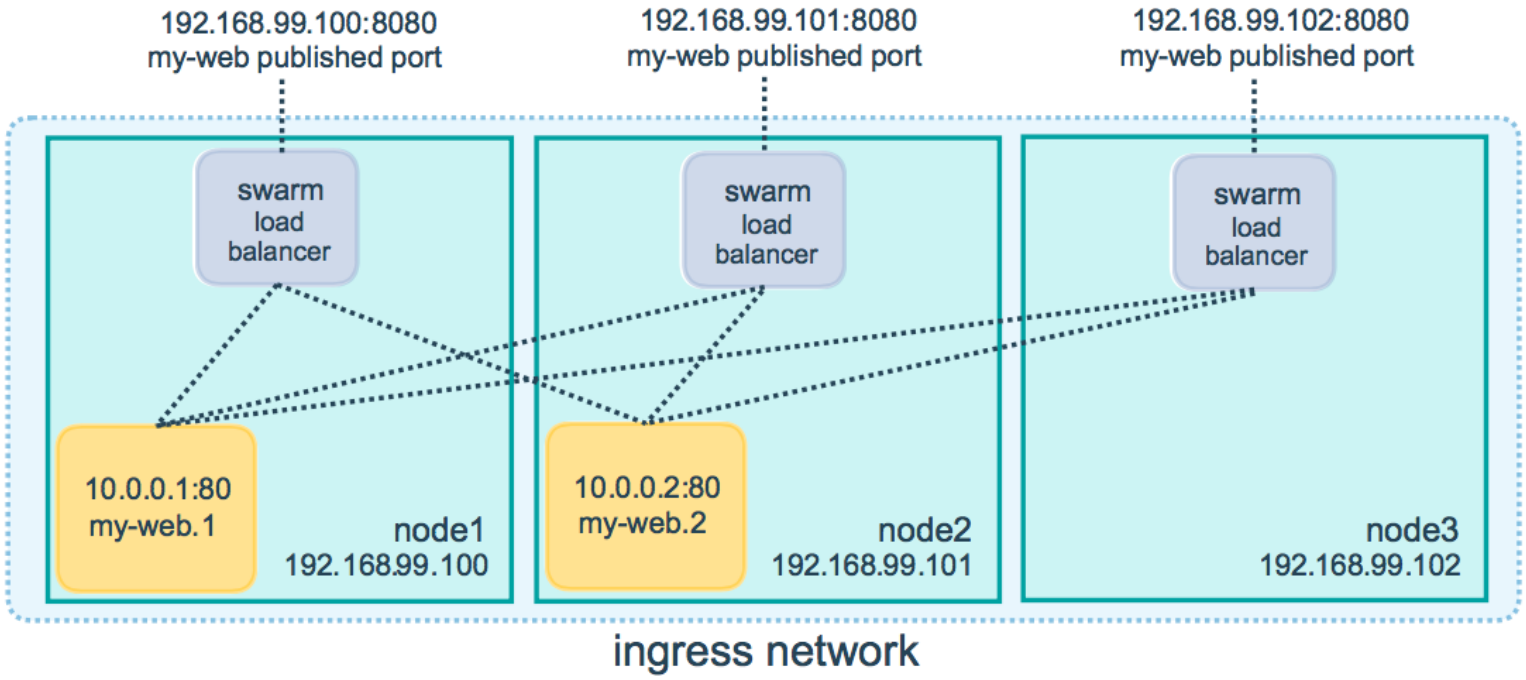
What is Load Balancing?

- Swarm uses ingress load balancing
- Swarm automatically assigns PublishedPorts
- Swarm can dynamically assign ports (30000 - 32767)
- External Load Balancers access PublishedPorts of Nodes
- ALL nodes route ingress traffic to the correct Task
- Swarm has internal DNS (routing mesh)

Docker Swarm (Load Balancing)

Swarm Load Balancing Routing Mesh

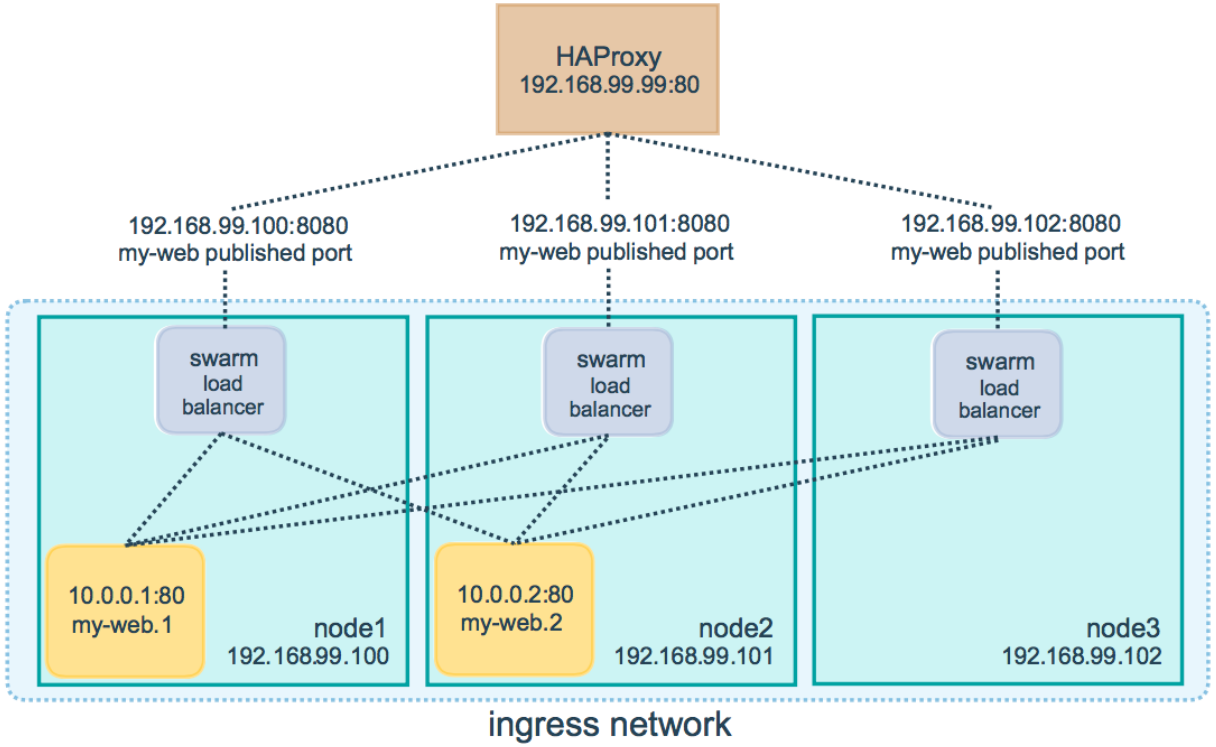
- All nodes are aware of **TASK** locations in Swarm
- Can send traffic to any node
- Traffic automatically routed to correct node



Docker Swarm (Load Balancing)

Using an External Load Balancer

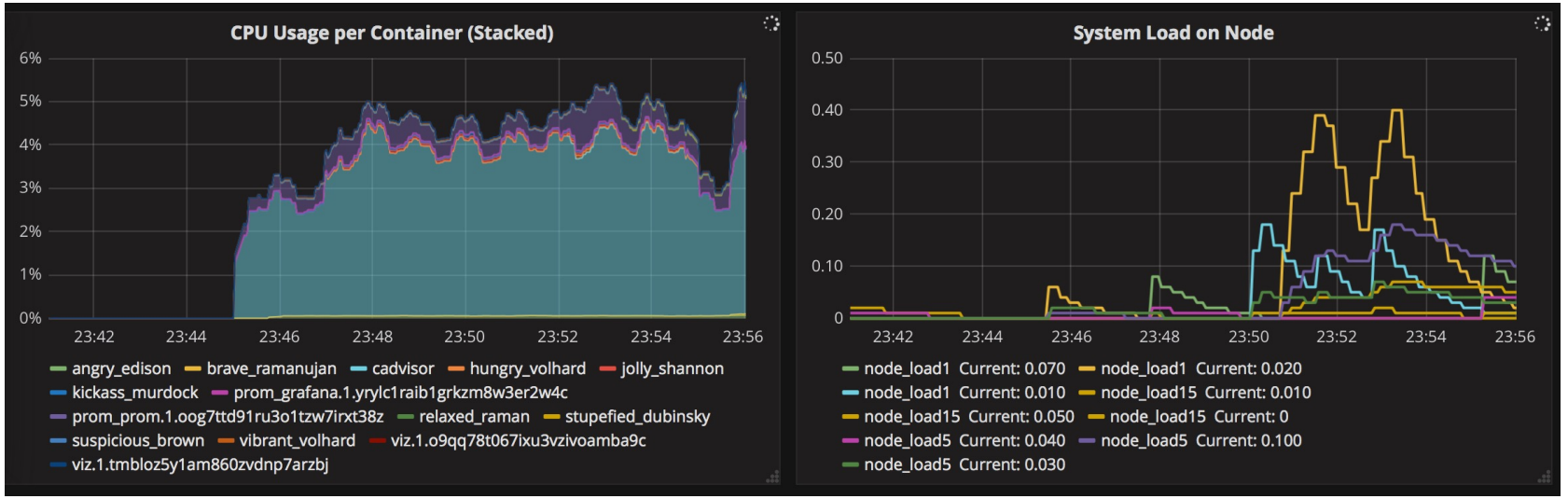
- External Load Balancer on TCP 80
- Internal Swarm Routing Mesh on TCP 8080
- All nodes are aware of **TASK** locations in Swarm



Swarm Scheduler

Swarm Scheduler - Resource Availability

Resource Availability is tracked by the Docker Engine on each node. Swarm Scheduler is aware of available resources on all Worker Nodes



Swarm Scheduler - Labels

Labels can be specified to record attributes of a Swarm node (e.g. role = manager, env = dev, storage = ssd)

```
# $ docker node update [OPTIONS] NODE
```

```
# $ docker node update --label-add env=dev node-1
```

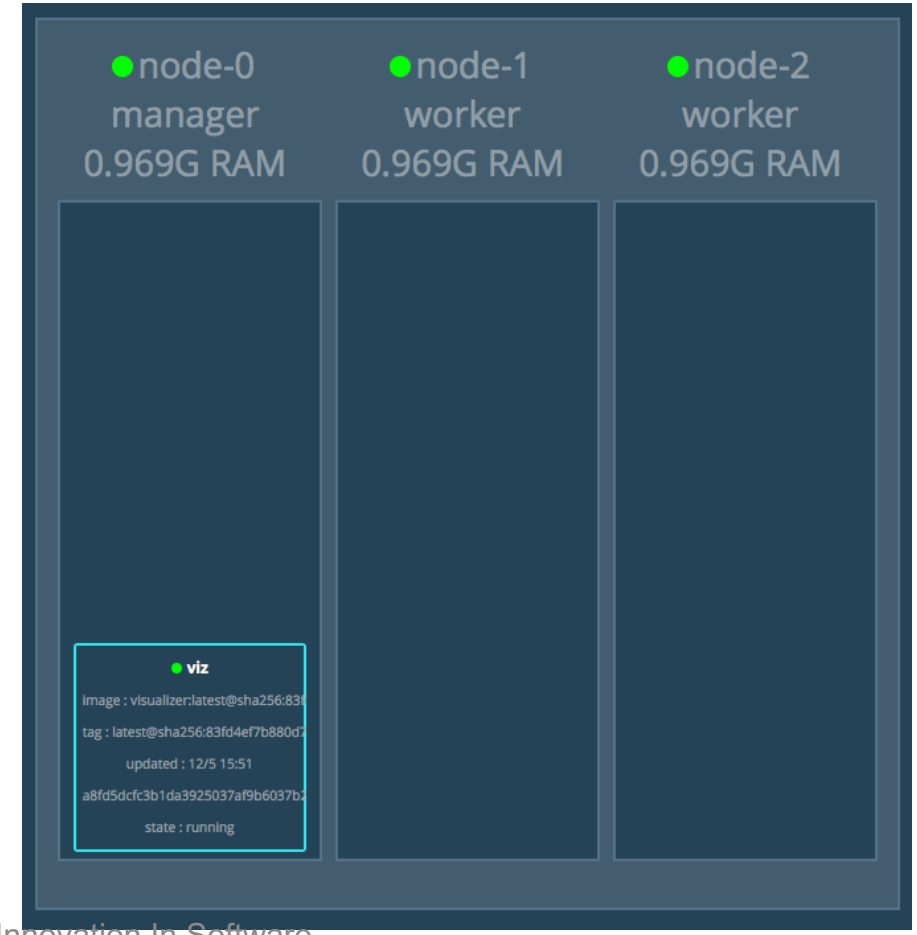
```
# $ docker node inspect node-1
```

```
# $ docker node inspect --pretty node-1
```

```
# $ docker node update --label-add disk=ssd node-1
```

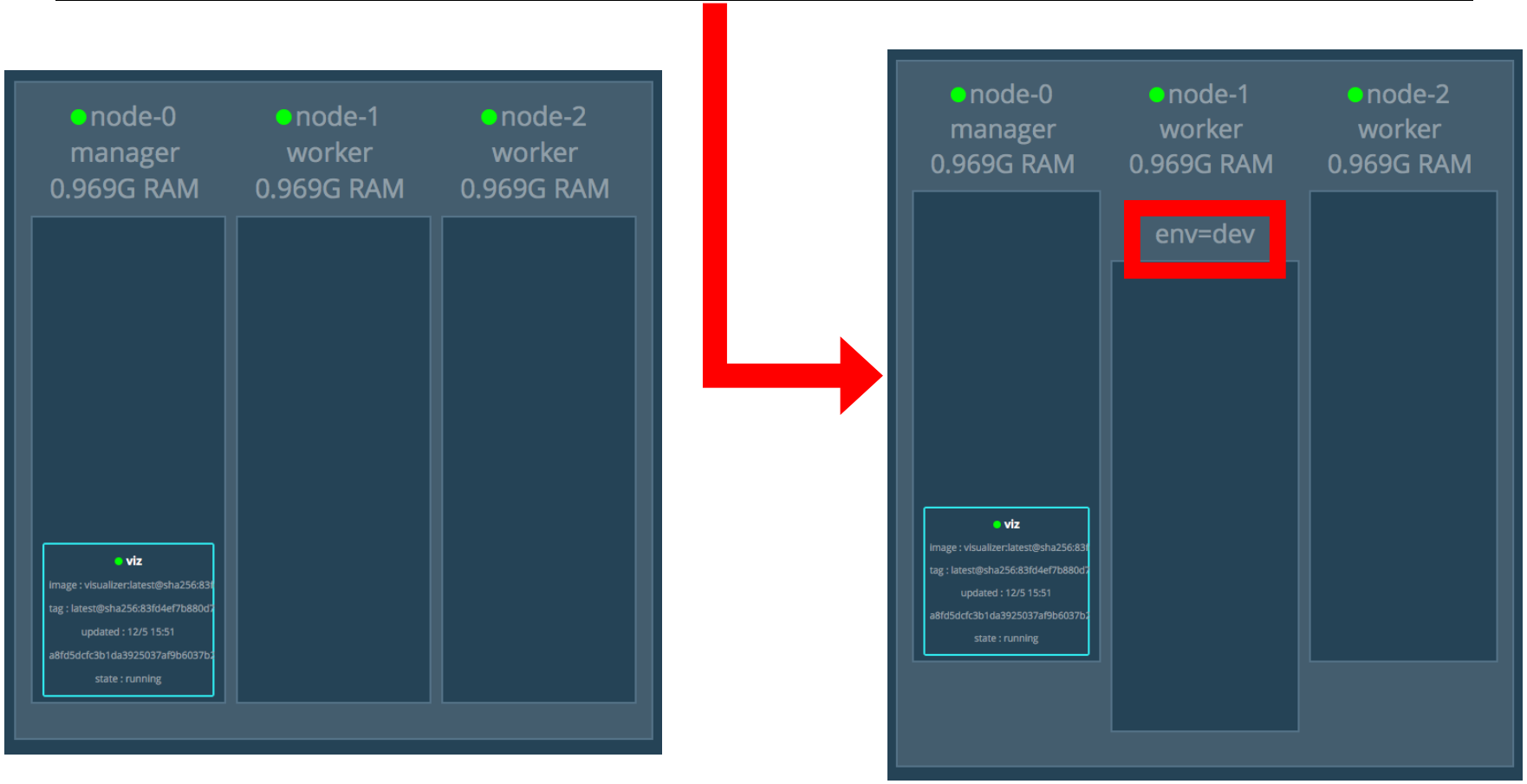
Swarm Scheduler - Labels

```
# $ docker node update --label-add env=dev node-1
```



Swarm Scheduler - Labels

```
# $ docker node update --label-add env=dev node-1
```



Swarm Scheduler - Constraints

Constraints specified on a Service to restrict task scheduling to nodes with certain attributes, e.g. nodes with specific label

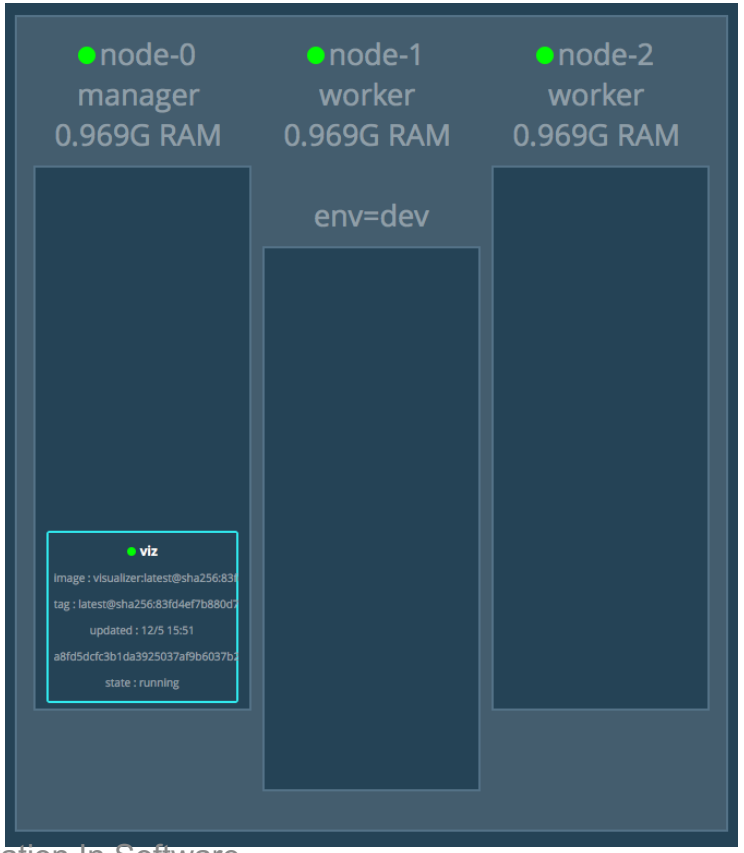
```
# $ docker service create --name nginx-dev \
#         --constraint 'node.labels.env == dev' \
#         nginx:1.11-alpine
```

```
# $ docker node inspect --pretty node-1
#    Labels:
       - disk = ssd
       - env = dev
#    Hostname:              node-1
```


https://docs.docker.com/swarm/scheduler/strategy/

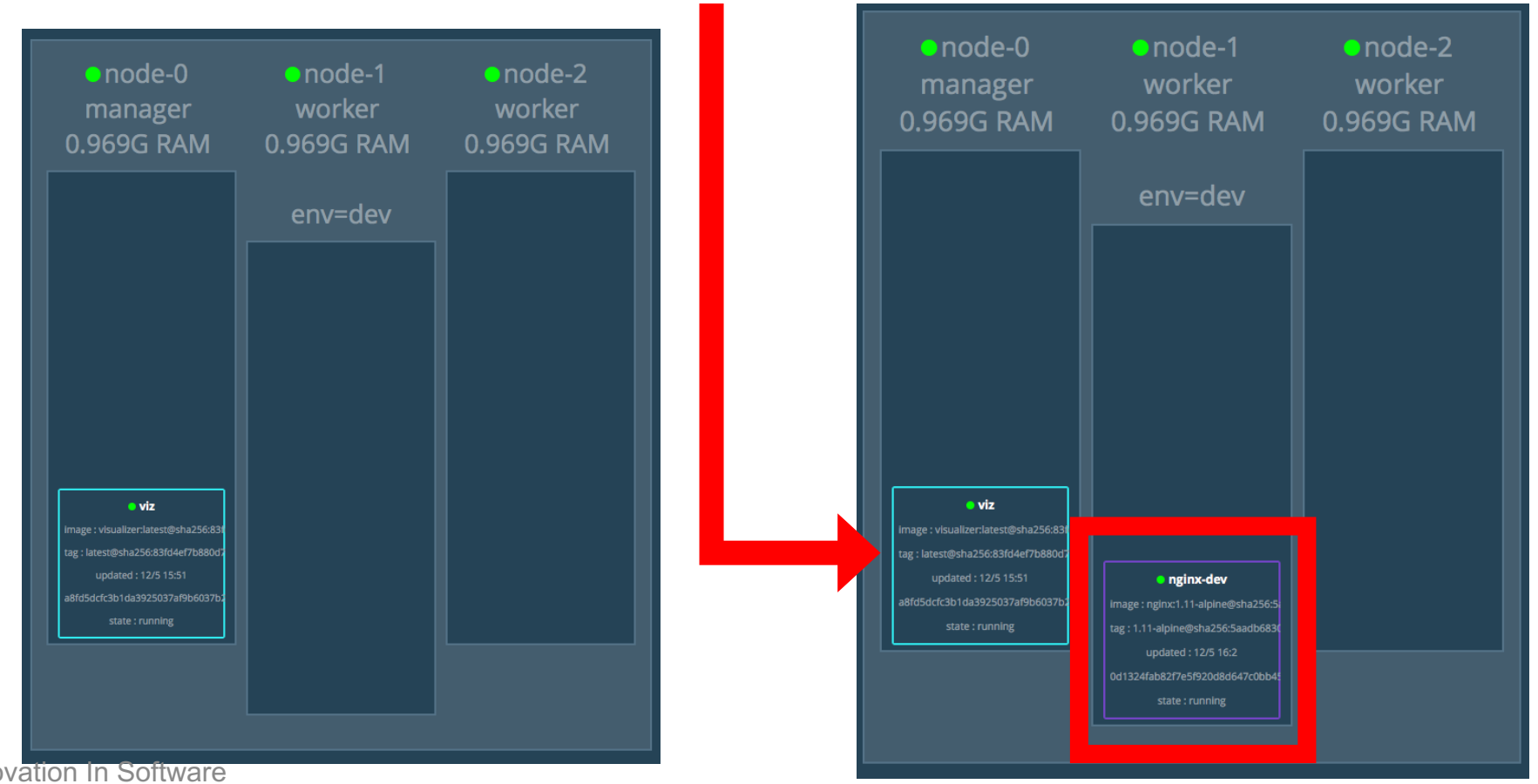
Swarm Scheduler - Constraints

```
# $ docker service create --name nginx-dev \
  --constraint 'node.labels.env == dev' \
  nginx:1.11-alpine
```



Swarm Scheduler - Constraints

```
# $ docker service create --name nginx-dev \
  --constraint 'node.labels.env == dev' \
  nginx:1.11-alpine
```



Swarm Scheduler - Pre-Defined Attributes

Constraints can reference a range of other pre-defined attributes on nodes, in addition to user-defined labels

node attribute	matches	example
node.id	Node ID	node.id == 2ivku8v2gvtg4
node.hostname	Node hostname	node.hostname != node-2
node.role	Node role	node.role == manager
node.labels	user defined node labels	node.labels.security == high
engine.labels	Docker Engine's labels	engine.labels.operatingsystem == ubuntu 14.04

Deploy & Configure Swarm

Deploy & Configure Swarm

Prerequisites

Run Swarm Init

Run Swarm Join

- 1 Docker Nodes (minimum)
  - Swarm Manage/Node
- Docker Engine 17.03CE+ installed

Deploy & Configure Swarm

Prerequisites

Run Swarm Init

Run Swarm Join

```
# $ docker swarm init

# Swarm initialized: current node (p0lpp0knm362nipb6jxlq06a6) is now a
  manager.

  To add a worker to this swarm, run the following command:
     docker swarm join \
      --token SWMTKN-1-5bf0v9rruiau0a4a4ov2d4927abe9w31zyegr \
      10.0.22.10:2377

  To add a manager to this swarm, run 'docker swarm join-token manager' and
  follow the instructions.

# $
```

Source: https://docs.docker.com/swarm/install-manual/

Deploy & Configure Swarm

- Prerequisites
- Run Swarm Init
- Run Swarm Join

```
$ docker swarm join
This node joined a swarm as a worker.
```

Deploy a Service with Swarm

Deploy a Service with Swarm

Deploy an nginx application to the Swarm.

Deploy service on manager:

```
$ docker service create --name nginx \
    --replicas 2 nginx
```

Deploy a Service with Swarm

List containers in service

```
$ docker service ps nginx

ID      NAME      IMAGE          NODE    DESIRED STATE
rzybb   nginx.1   nginx:latest   node-2  Running
agoth   nginx.2   nginx:latest   node-0  Running
```

Scale a Service with Swarm

Scale nginx application to the Swarm.

Scale nginx service up

```
$ docker service update nginx --replicas 3
```

Scale a Service with Swarm

New container created for service

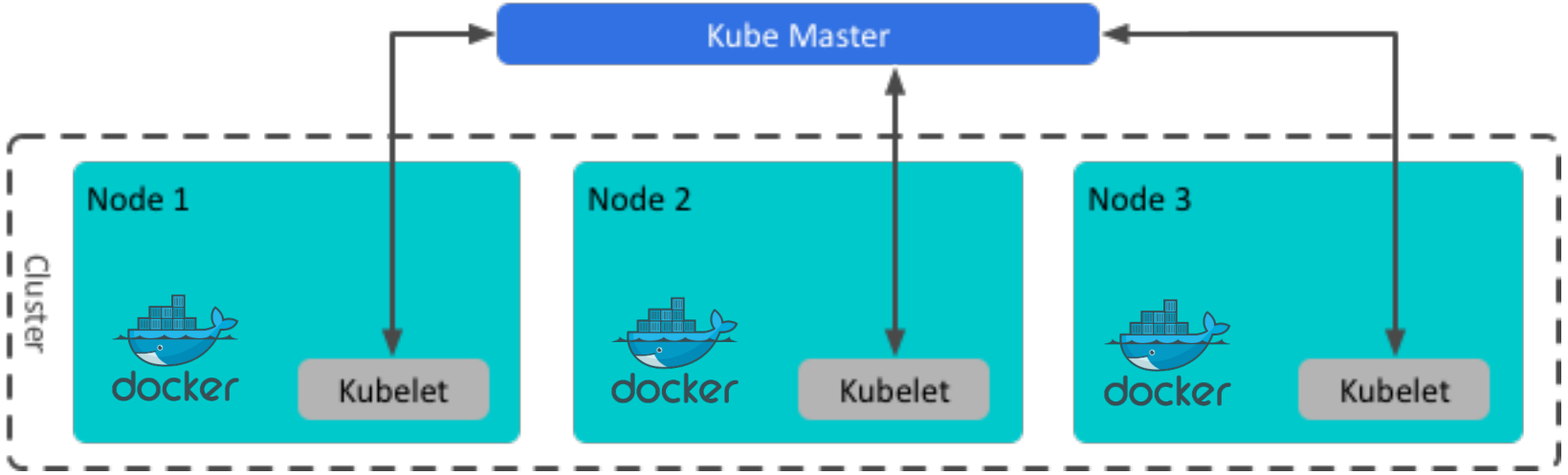
```
$ docker service ps nginx

ID      NAME      IMAGE          NODE    DESIRED STATE
rzybb   nginx.1   nginx:latest   node-2  Running
agoth   nginx.2   nginx:latest   node-0  Running
0yfp0   nginx.3   nginx:latest   node-1  Running
```

Lab: Swarm

Kubernetes – Containers at scale

Kubernetes: Containers at Scale

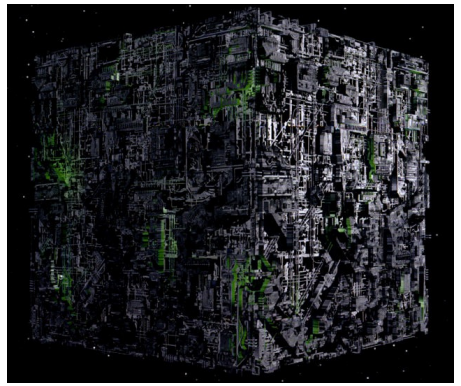
Kubernetes provides the infrastructure for container-centric deployment and operation of applications



- Request load balancing
- Application health checking
- Log access and resource monitoring

History of Kubernetes

- Google adopted containers as an application deployment standard over a decade ago
  - Contributed cgroups to Linux kernel in 2007
- Google developed generations of container management systems, scaling to thousands of hosts per cluster
  - First was Borg, treated as a trade secret until 2015*
  - Omega built on concepts of Borg, also Google-internal
  - Kubernetes inspired by observed needs of Google Cloud Platform customers, open source
- All major Google services run on Borg
- Other cluster management frameworks, like Apache Mesos, inspired by Borg



Star Trek Borg Cube

A hegemonizing swarm

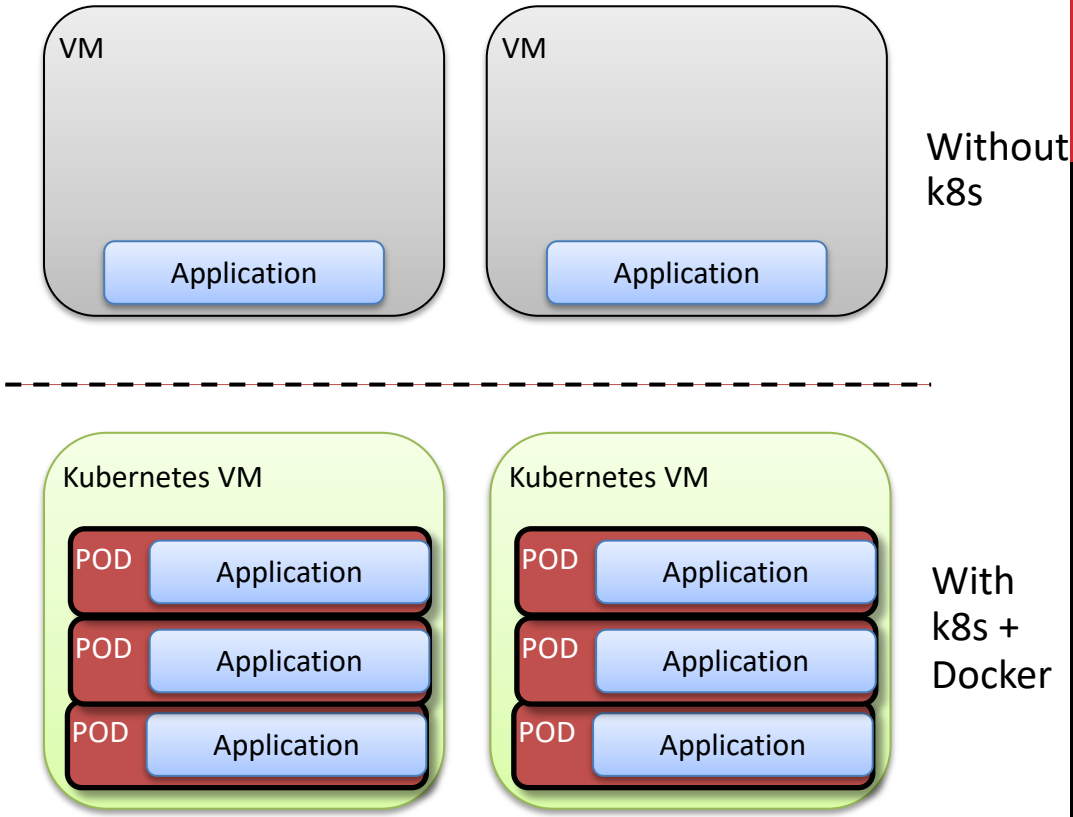
Original project name:

Seven of Nine

(the friendly Borg)

Kubernetes: Google Cloud Platform

- After launch of Google Compute Engine, utilization of VM's by customers was observed to be very low
  - Running apps in whole VM's leading to stranded resources
- GCE itself already running on Borg, which solved utilization through efficient scheduling of containerized applications
- Google engineers pushed to found Kubernetes, as open source project
  - K8s available to GCE customers
  - Not tied to Google or GCP infrastructure



Kubernetes today – CNCF Open Source Project

- Cloud Native Computing Foundation (CNCF) founded in 2015 as Linux Foundation Collaborative Project
- Kubernetes contributed by Google to CNCF at same time, still the flagship project
  - One of the most active projects on GitHub
- CNCF membership includes growing range of major industry vendors, including Cisco
- CNCF also governs the major container engine projects, thanks to recent donations
  - containerd (core runtime of Docker Engine)
  - rkt

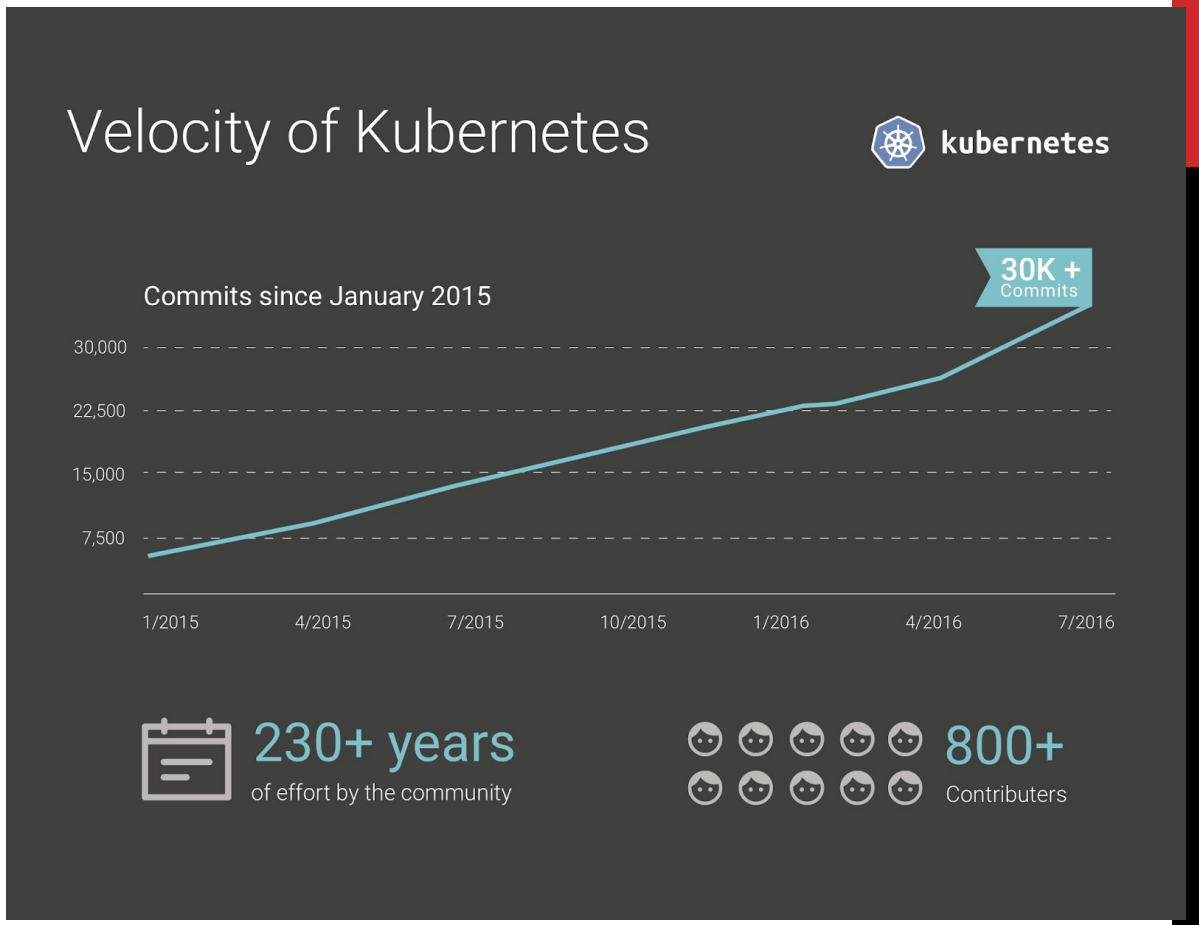


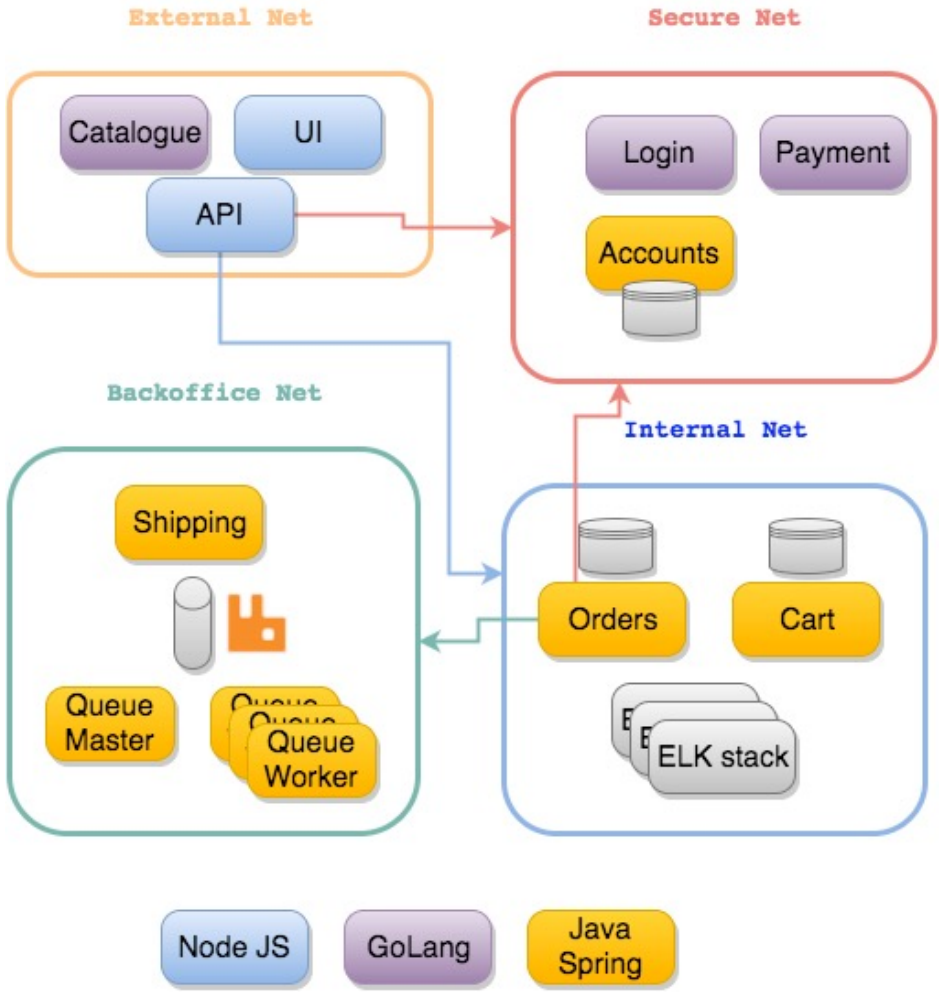
Image Source: https://cloudplatform.googleblog.com/2016/07/from-Google-to-the-world-the-Kubernetes-origin-story.html

Kubernetes Use Cases

Kubernetes:Cloud-Native Application Deployment

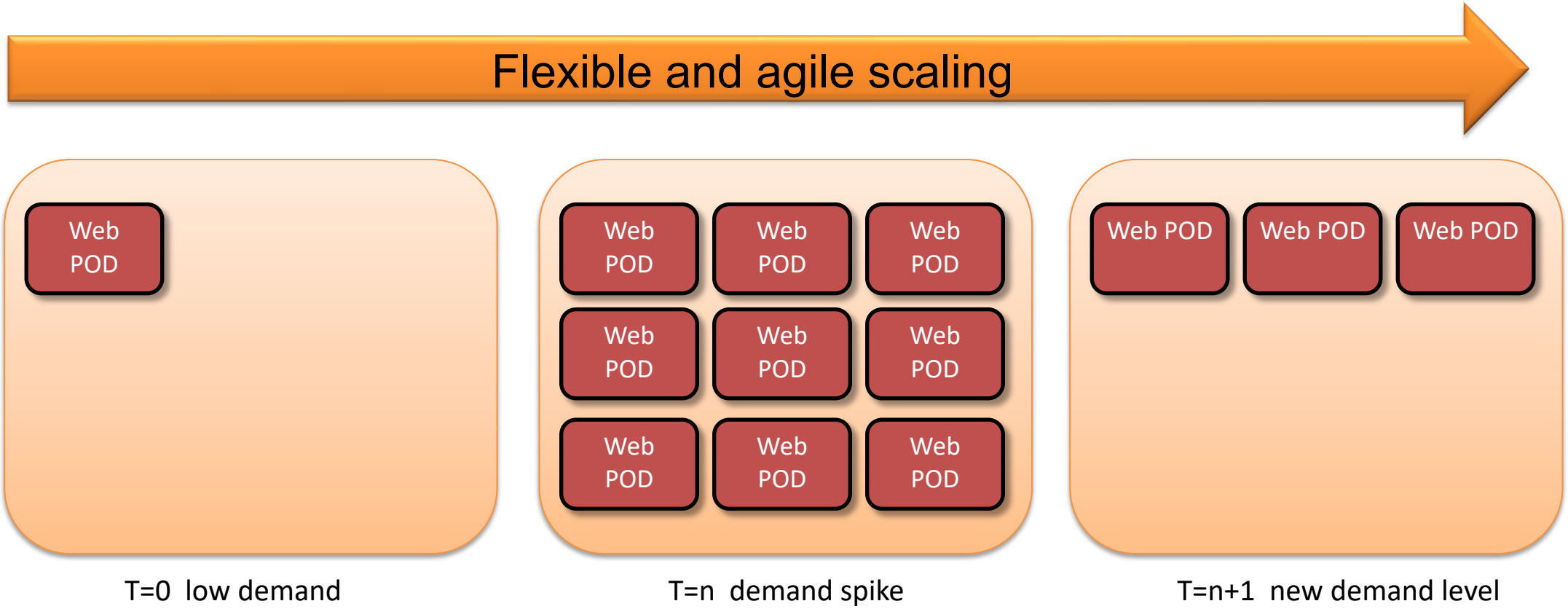
Cloud-native applications, aka microservice-based or 12-factor apps

- Cloud-native applications are composed of small, independently-deployable, loosely-coupled services
- Kubernetes makes it easy to deploy, update, and coordinate operations between multiple containerized service components
- Kubernetes project actively enhancing features to better support and manage stateful applications



Kubernetes: Elastic Services

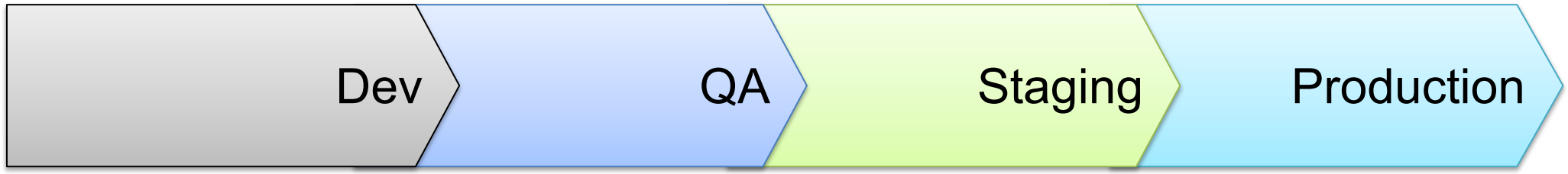
Kubernetes supports manual and automated scaling of application services based on demand for resources



Kubernetes: CI/CD Pipelines

Managing execution environments in a CI/CD pipeline

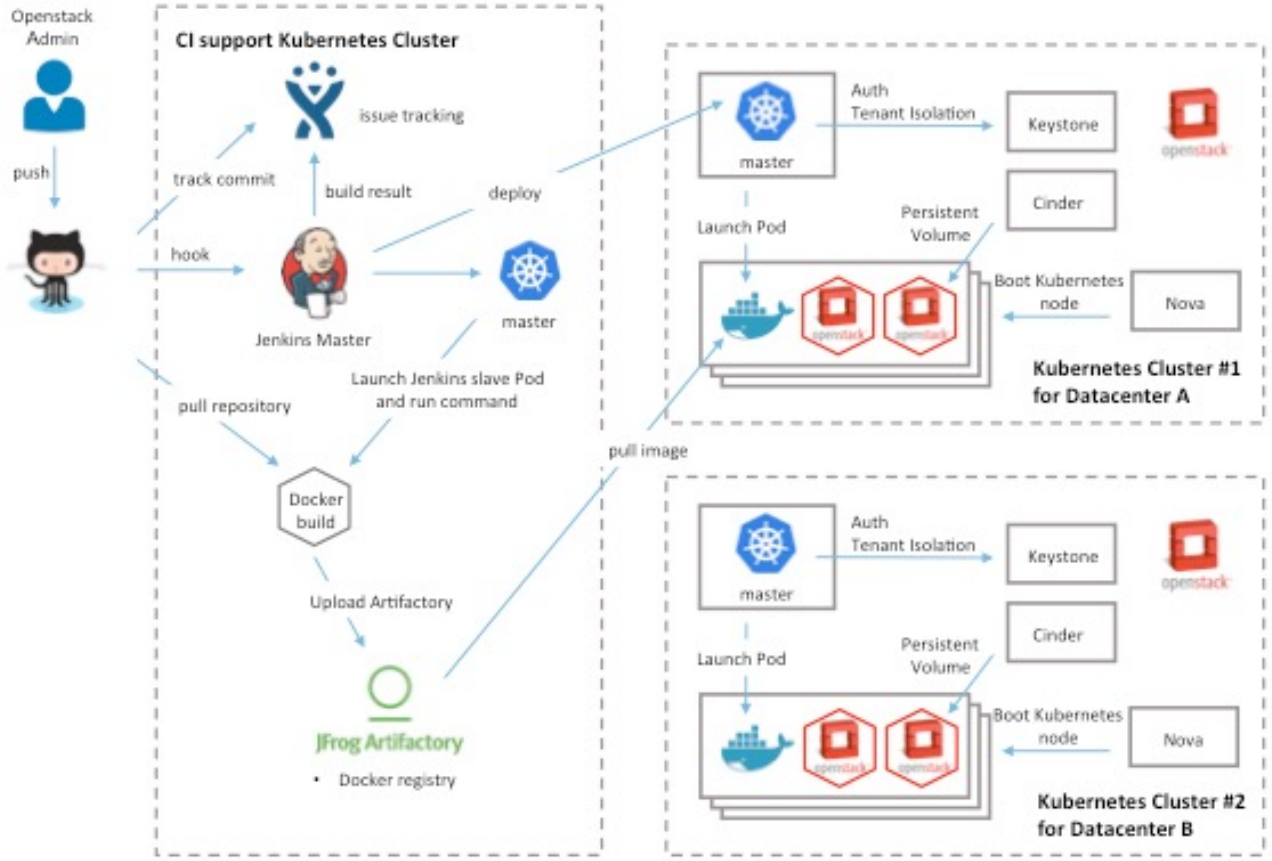
- Automated systems can push to logical environments in the same cluster, or to different clusters, using the same control plane API
- Kubernetes labels and annotations allow users to organize resources into separate environments without changing code or functionality
- Label selectors target specific sets of resources for control and exposure



Example: Kubernetes CI in Production at Scale

YAHOO! JAPAN

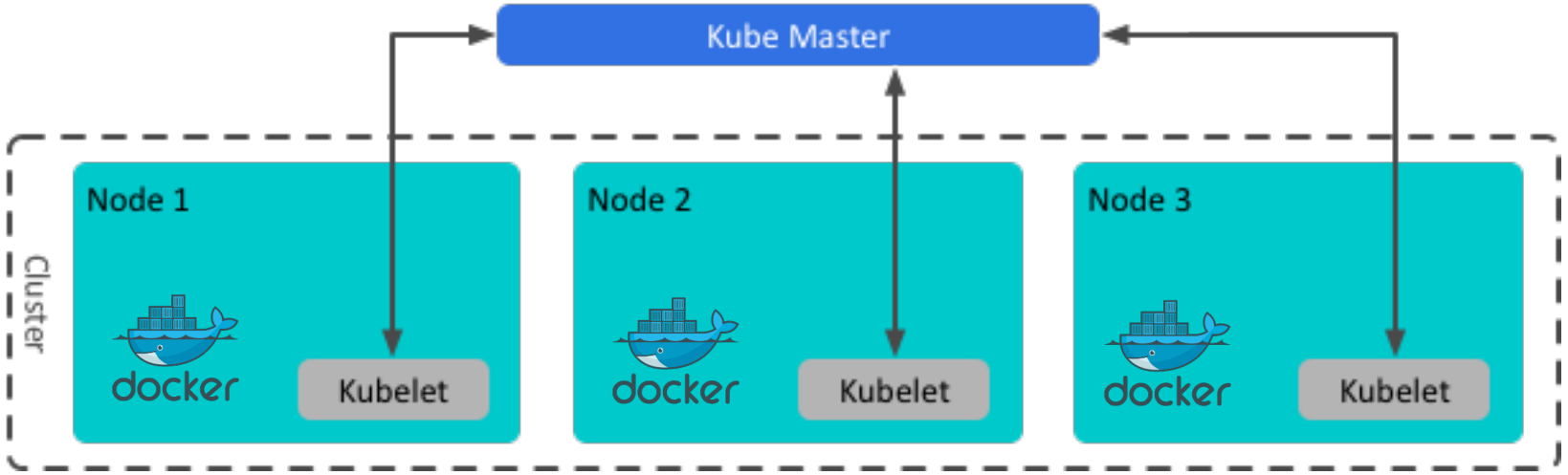
- Built automated system to build and deploy containerized applications on Kubernetes
- Kubernetes clusters built from VM's in private OpenStack environments
  - 1,000 nodes
  - 42,000 Containers
  - ~2,500 Applications



Overview of Kubernetes Clusters

Kubernetes Clusters

- Kubernetes deployed on a set of physical or virtual hosts – K8s nodes
- Hosts run host OS that supports Linux containers, e.g. Docker or rkt hosts
- Kubernetes runs well in both private and public IaaS environments
- Users and admins control Kubernetes resources via REST API on K8s master



Kubernetes Components: Master

Main Components:

Leader Node:

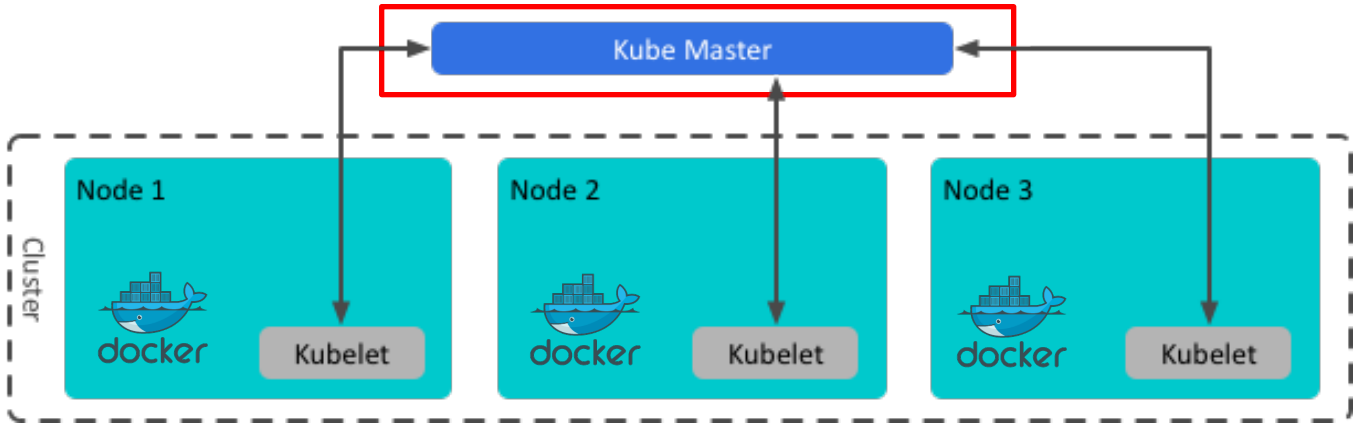
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

Worker Nodes:

- Run user workloads as directed by the K8s master
- The master may also serve as a worker node

Clusters:

- A collection of nodes bound to a Master and managed as a single logical unit of capacity



Kubernetes Components: Nodes

Main Components:

Leader Node:

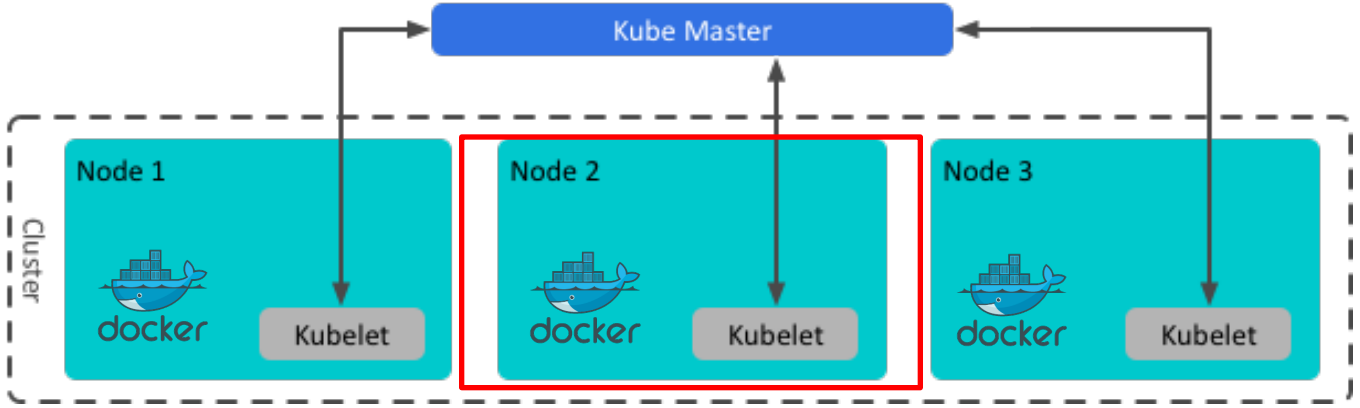
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

Worker Nodes:

- Run user workloads as directed by the K8s master
- The master may also serve as a worker node

Clusters:

- A collection of nodes bound to a Master and managed as a single logical unit of capacity



Kubernetes Cluster Components

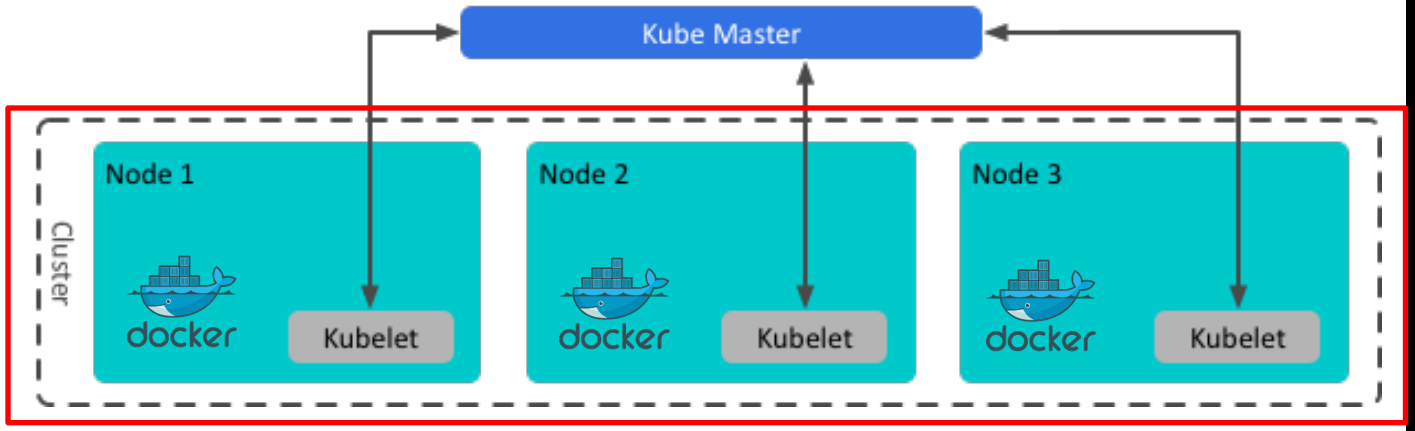
Main Components:

Leader Node:

- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

Worker Nodes:

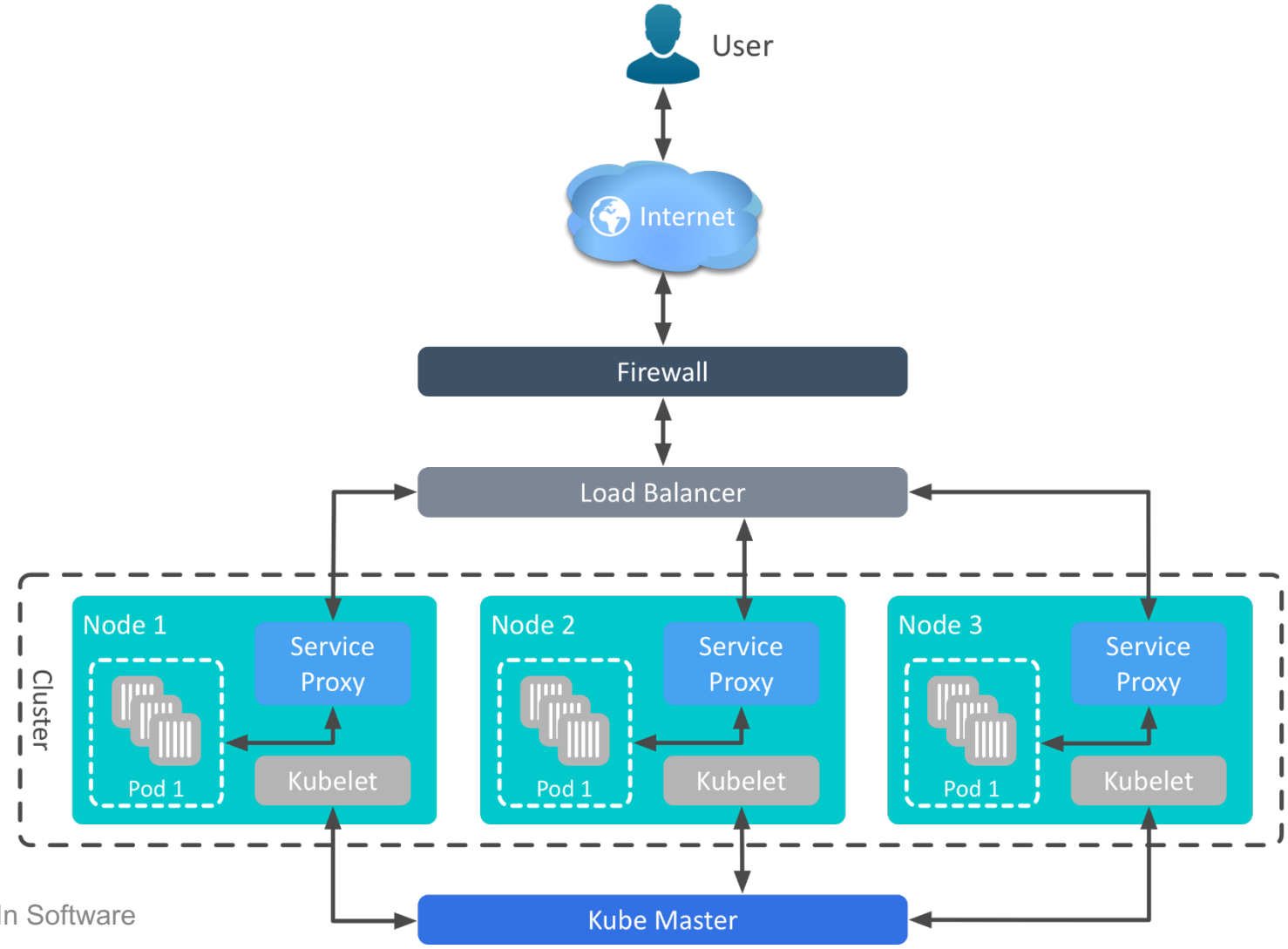
- Run user workloads as directed by the K8s master
- The master may also serve as a worker node



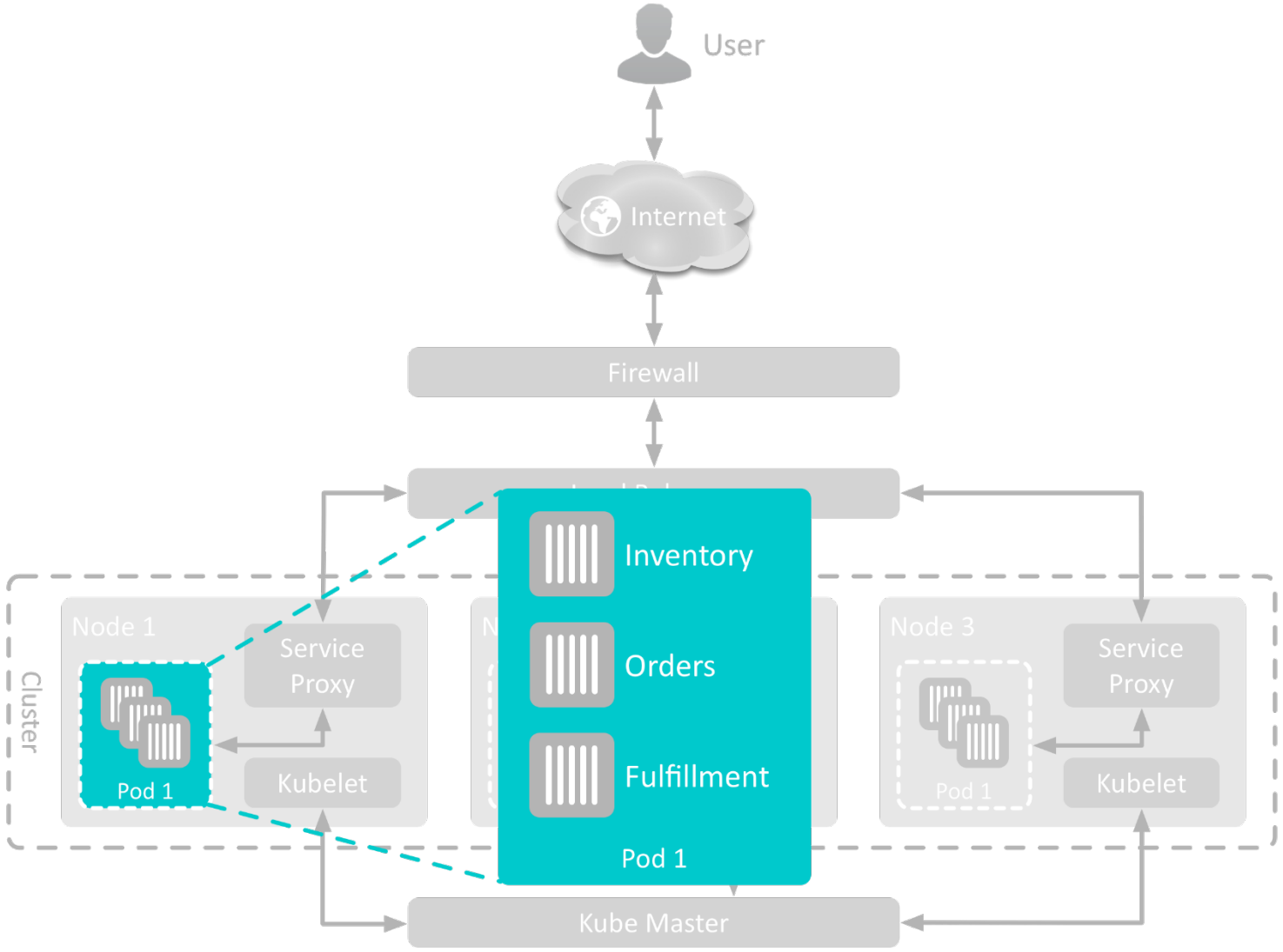
Clusters:

- A collection of nodes bound to a Master and managed as a single logical unit of capacity

Kubernetes Architecture



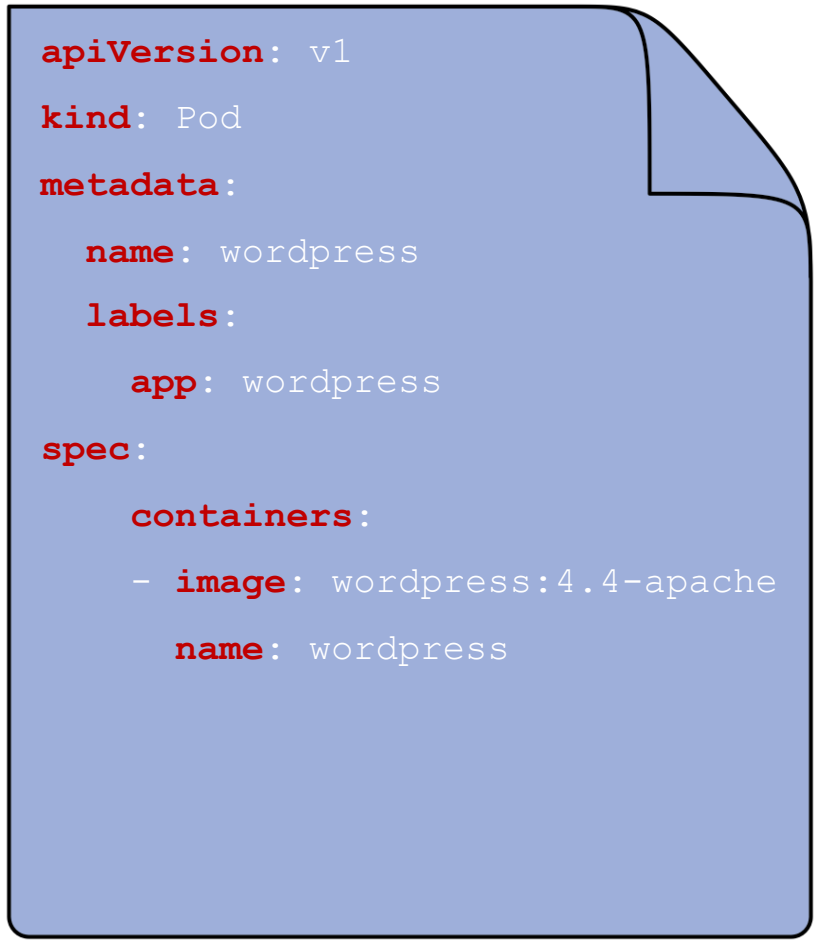
Work Units - Pods



Work Units - Pods

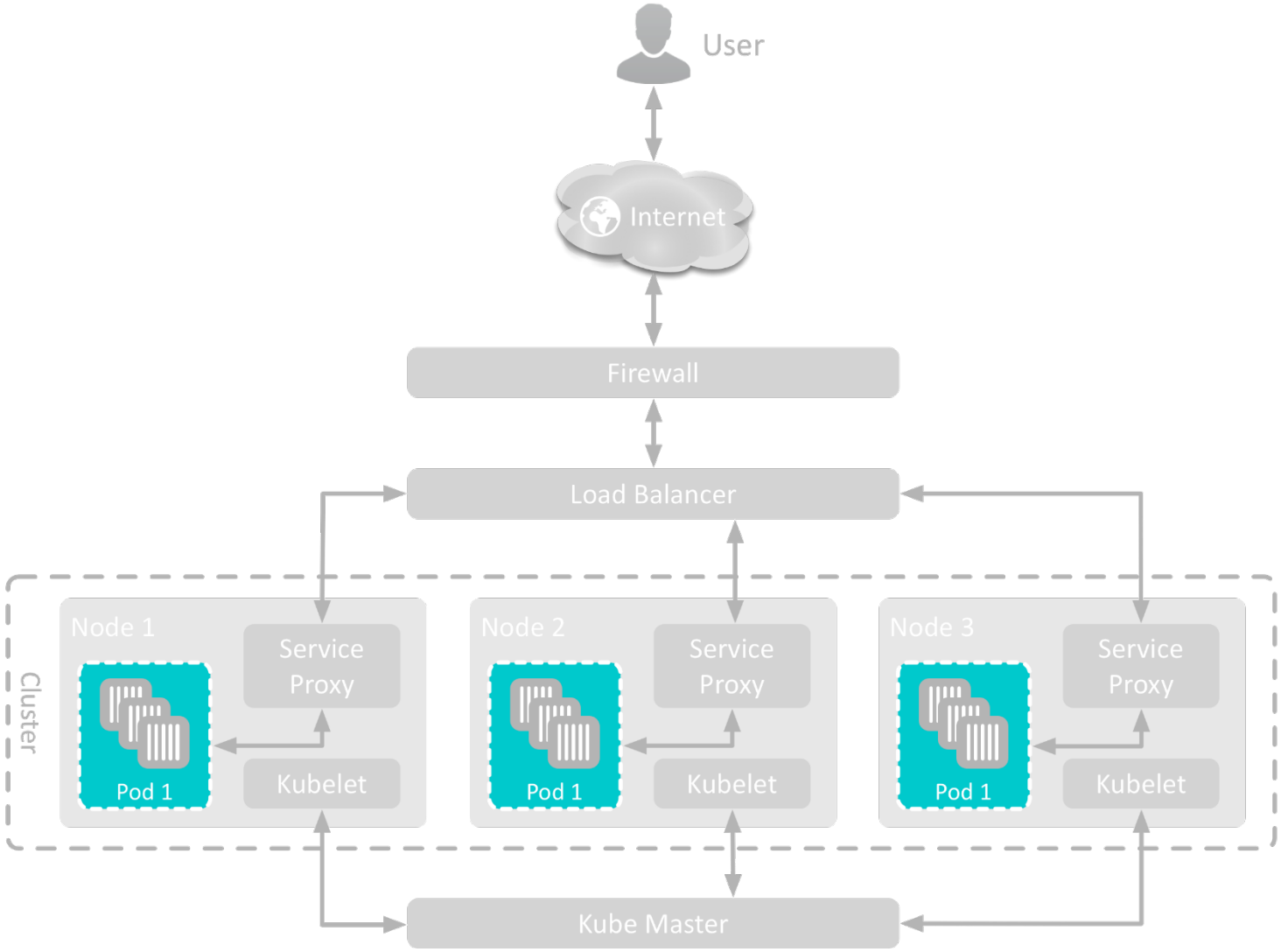
The basic work unit in Kubernetes

- One or more containers to be orchestrated together
- All containers in a pod share environment
  - IP
  - Volumes
- Pod definitions are declarative



Pod definition

Work Unit - Deployments



Work Unit - Deployments

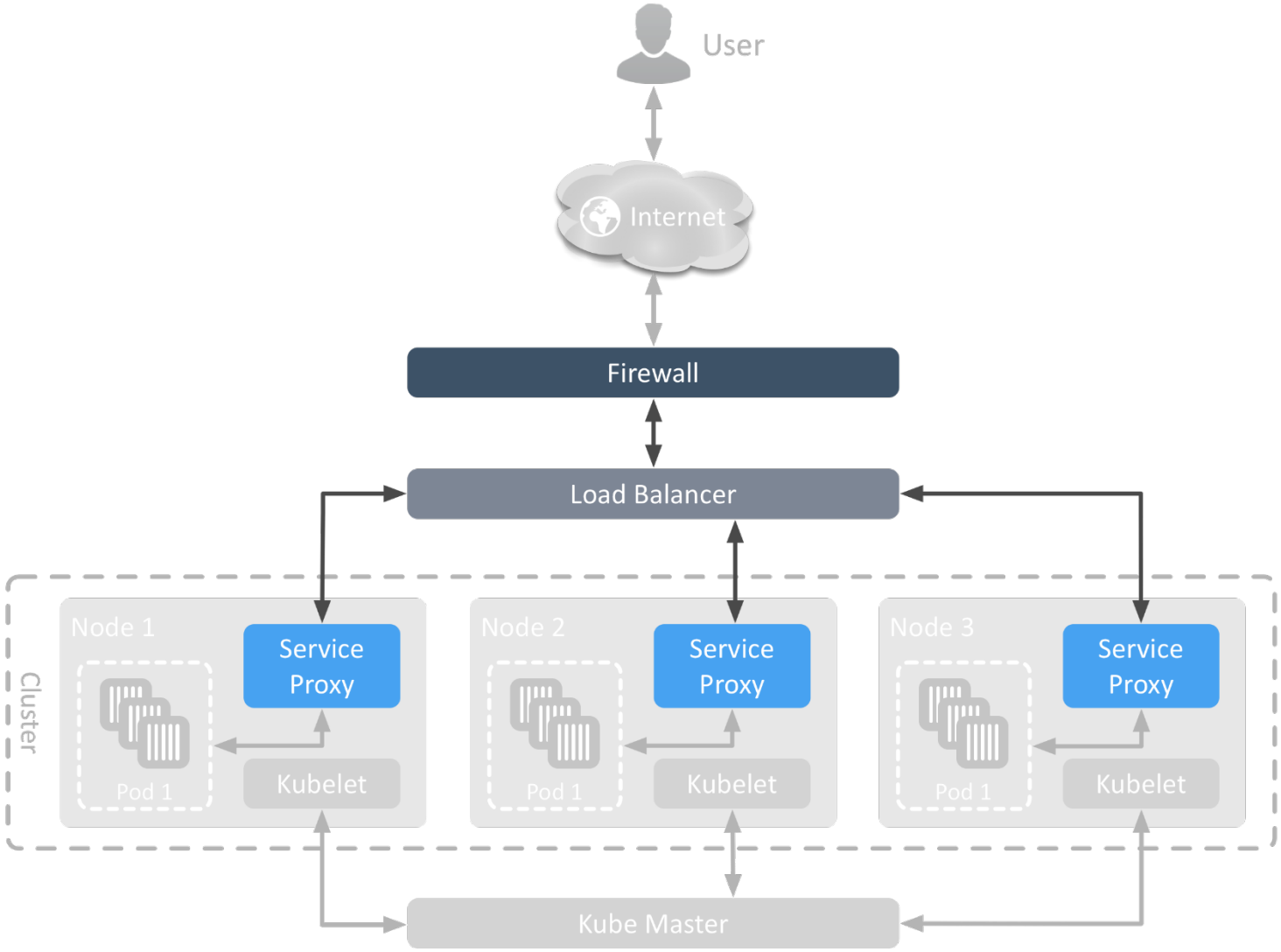
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: Wordpress
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
      - name: wordpress
        image: wordpress:4.4-apache
        ports:
         - containerPort: 80
```

Deployment Definition

Deployments control a **set of pod replicas**

- Meant to be horizontally scaled
- Deployments will maintain desired number of copies
- If Pods go down, they are recreated automatically

Work Unit - Services



Work Unit - Services

Acts as an abstraction between layers of your application

- Tracks all pods matching specified label selector
- Provides stable IP as named load balancer for pod endpoints

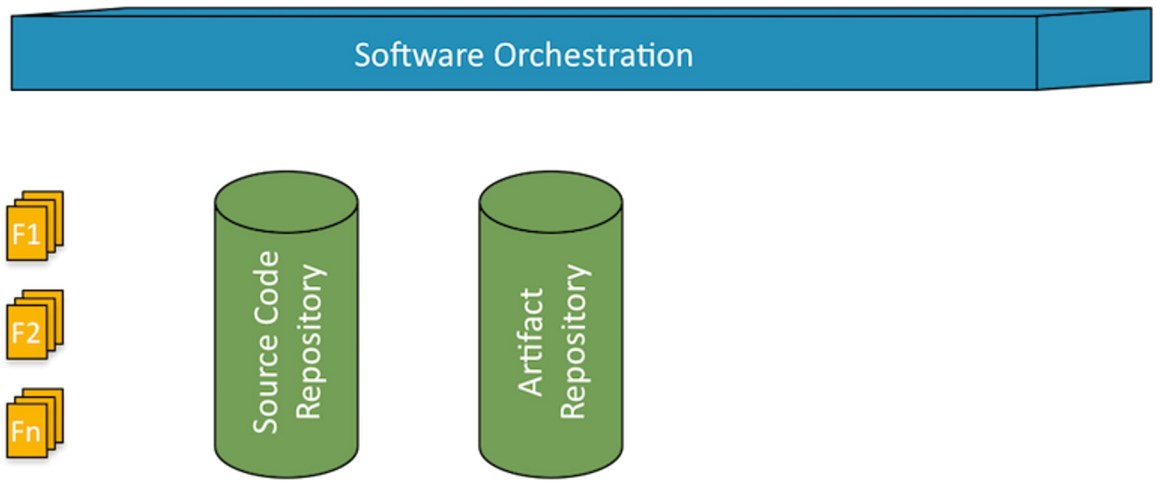
```yaml
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
    tier: frontend
  type: LoadBalancer
```

Service Definition

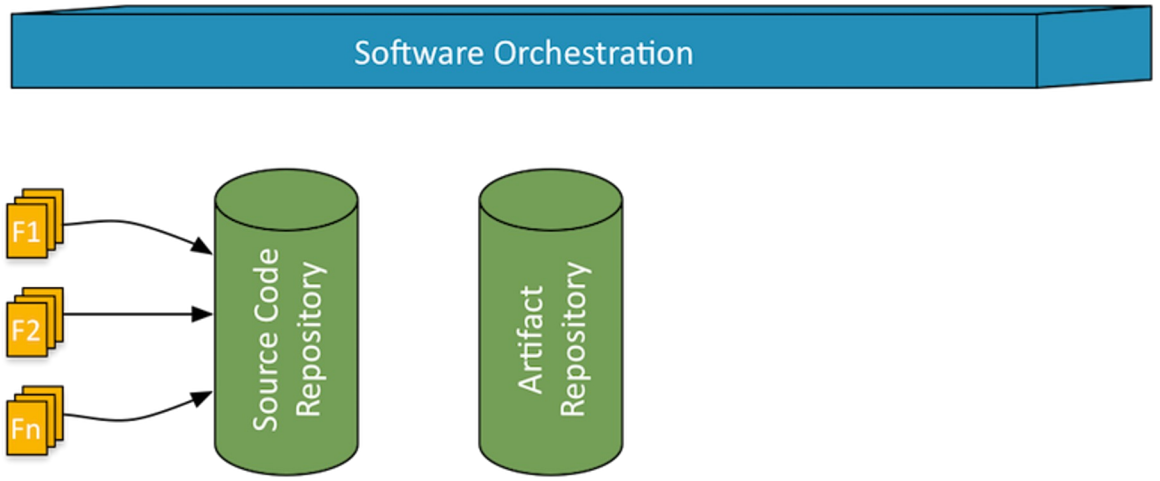
Docker for CI & CD

What is CI & CD?

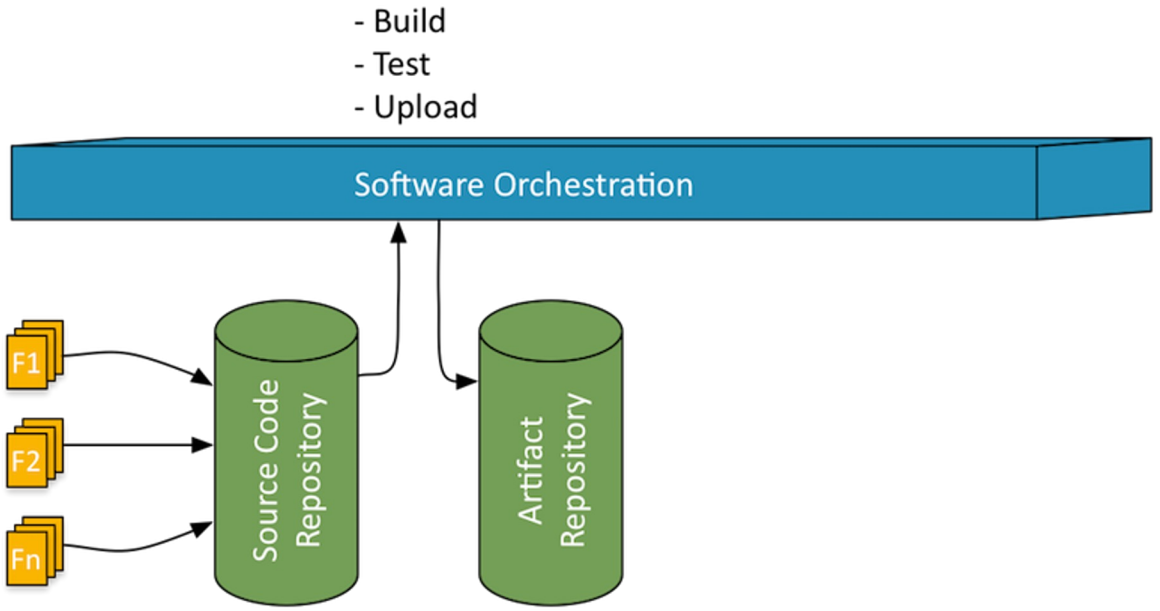
What is Continuous Integration (CI)?



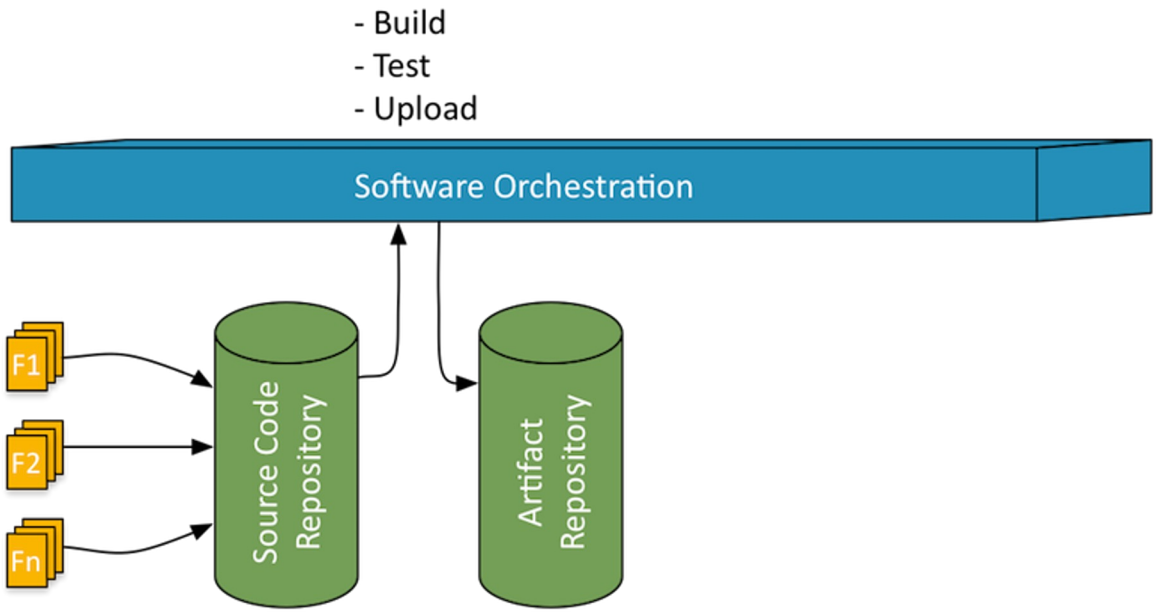
What is Continuous Integration (CI)?



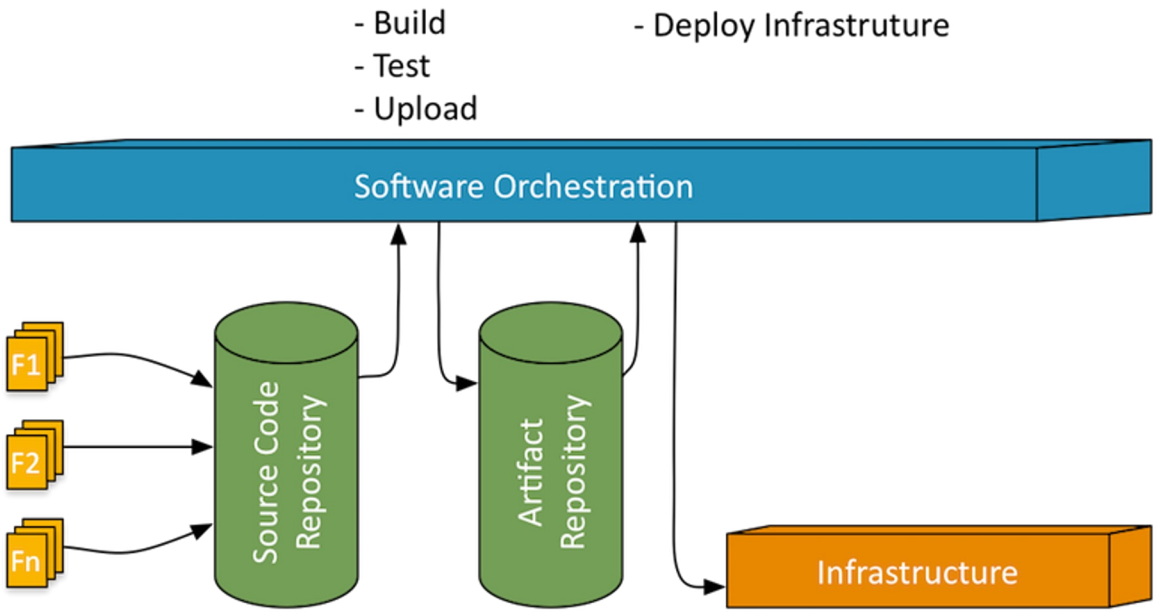
What is Continuous Integration (CI)?



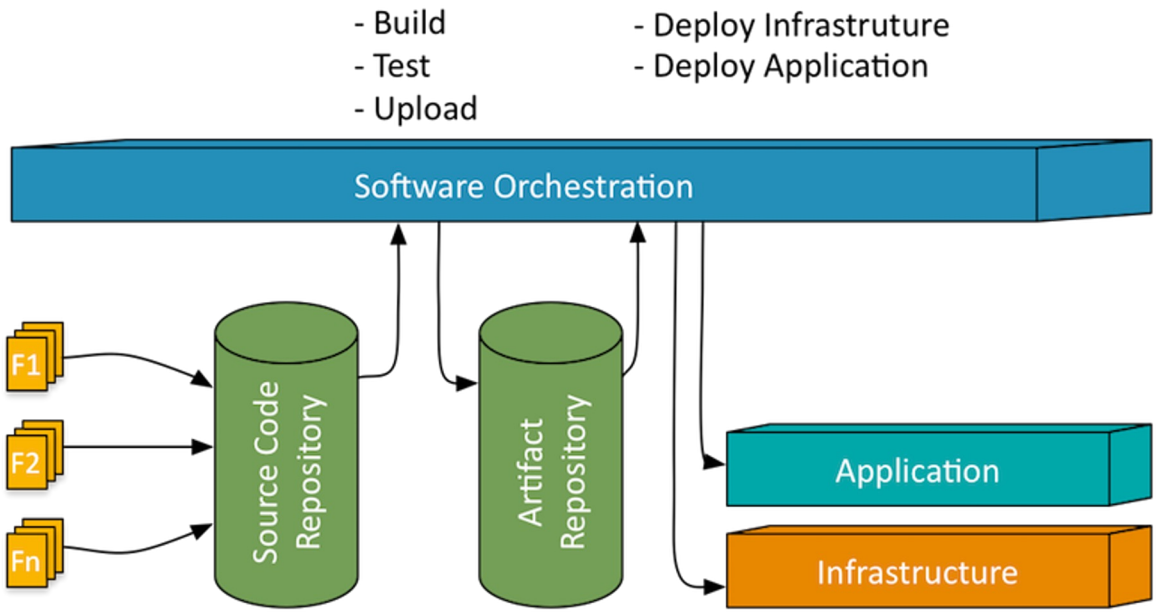
What is Continuous Delivery (CD)?



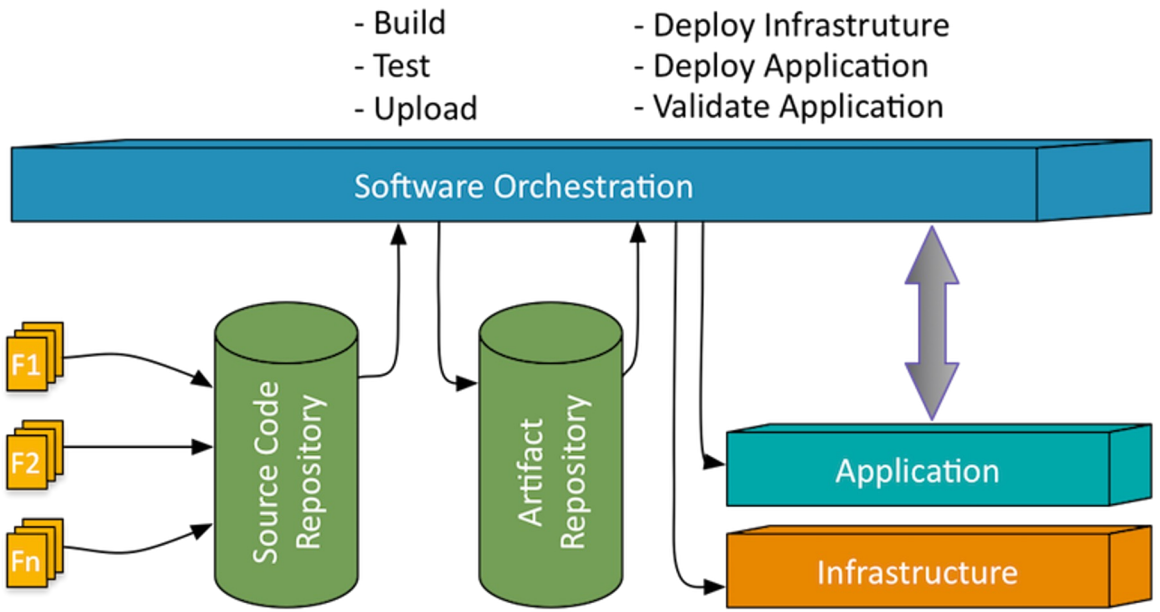
What is Continuous Delivery (CD)?



What is Continuous Delivery (CD)?



What is Continuous Delivery (CD)?



What's the Difference?

Continuous Integration:

- Frequent integrations of the code base

  Small, iterative change sets
- Automated builds for each commit
- Automated testing for validation
- Automated packaging/storage of applications

Continuous Delivery:

- Frequent deliveries

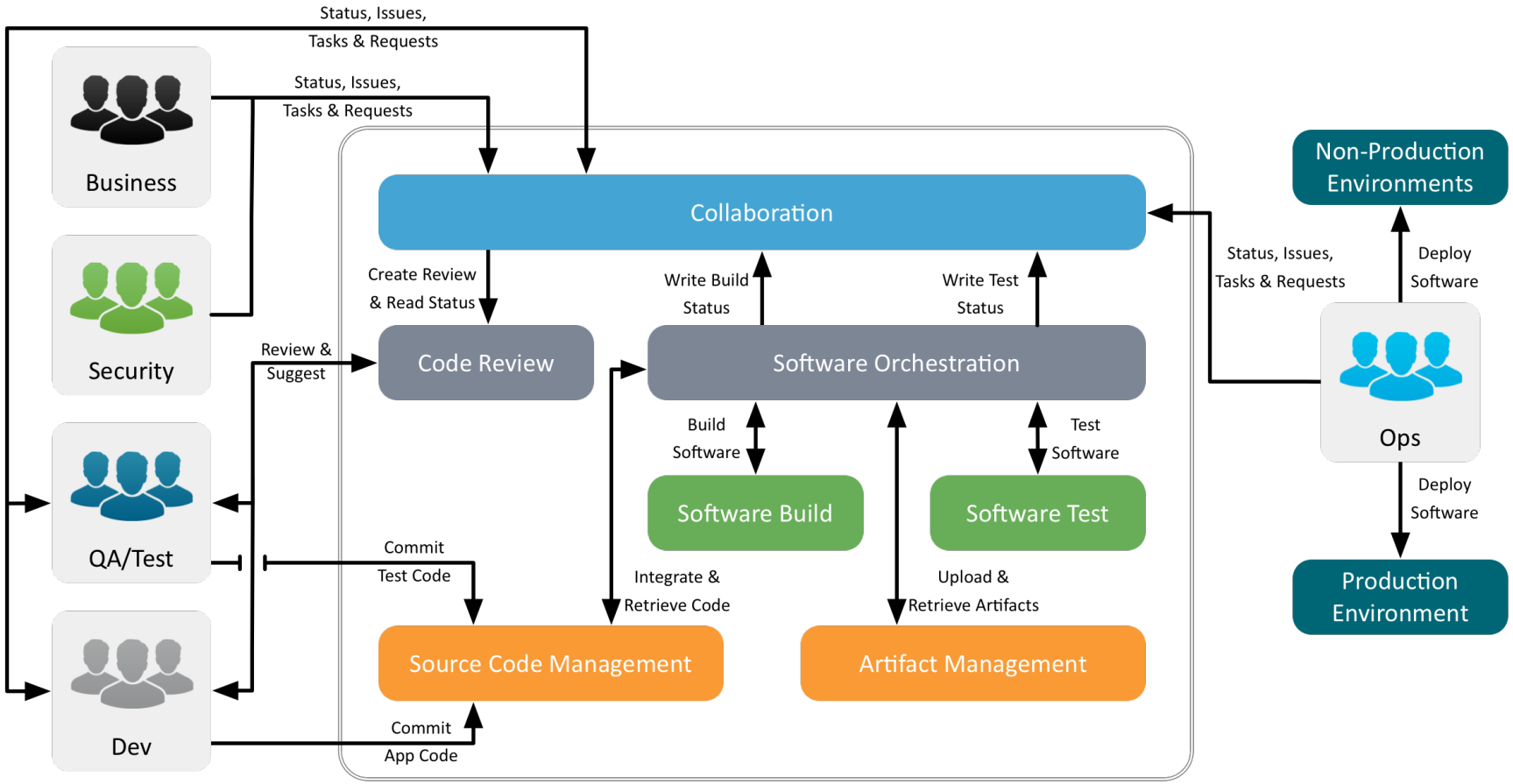
  Small, integrated change sets
- Automated deployment of Application Infrastructure
- Automated deployment of Applications
- Automated Application Testing/Validation

Docker Containers for CI

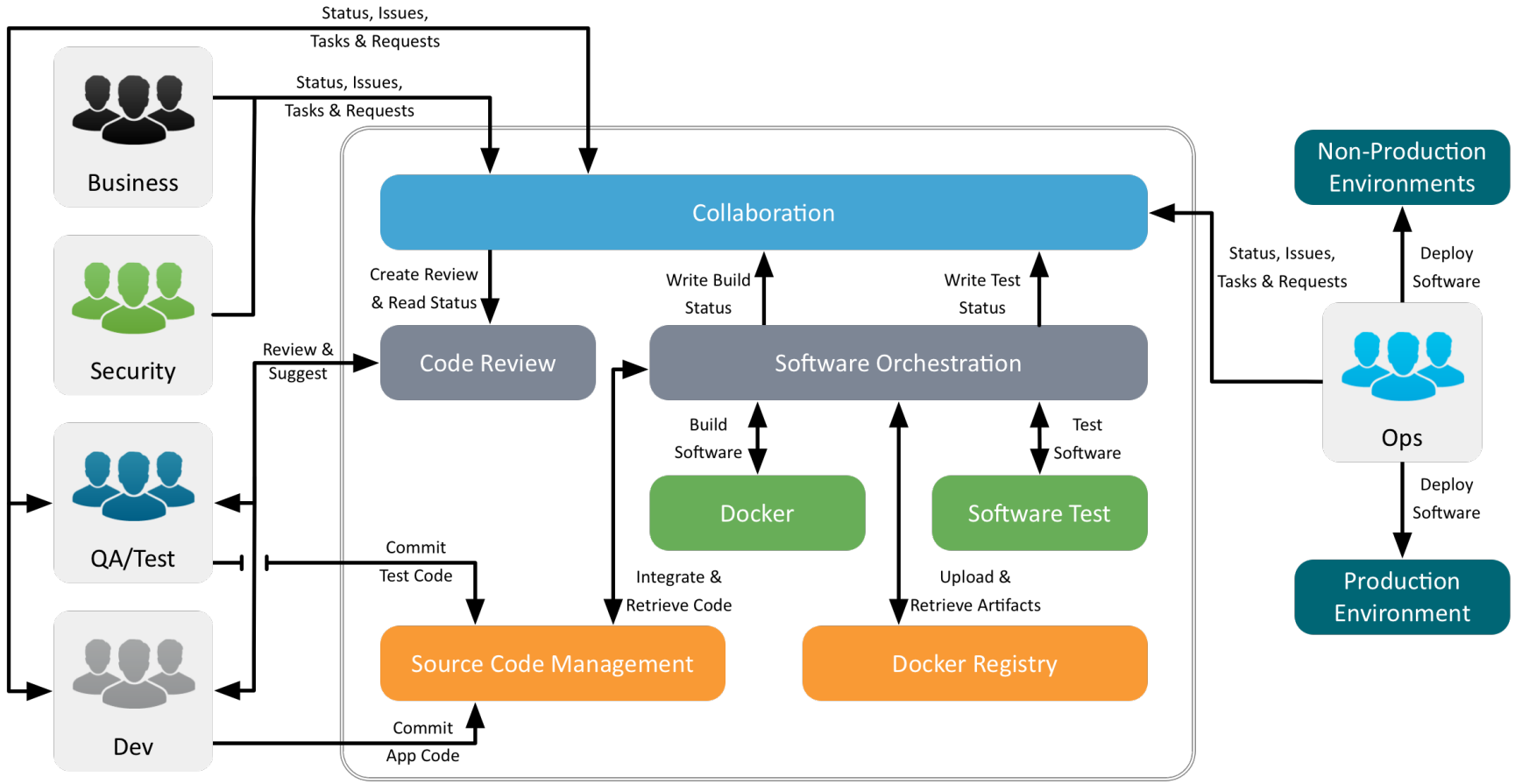
Docker for Continuous Integration

- Frequent deliveries
  - Small, integrated change sets
- Automated deployment of Application Infrastructure
- Automated deployment of Applications
- **Automated Application Testing/Validation**

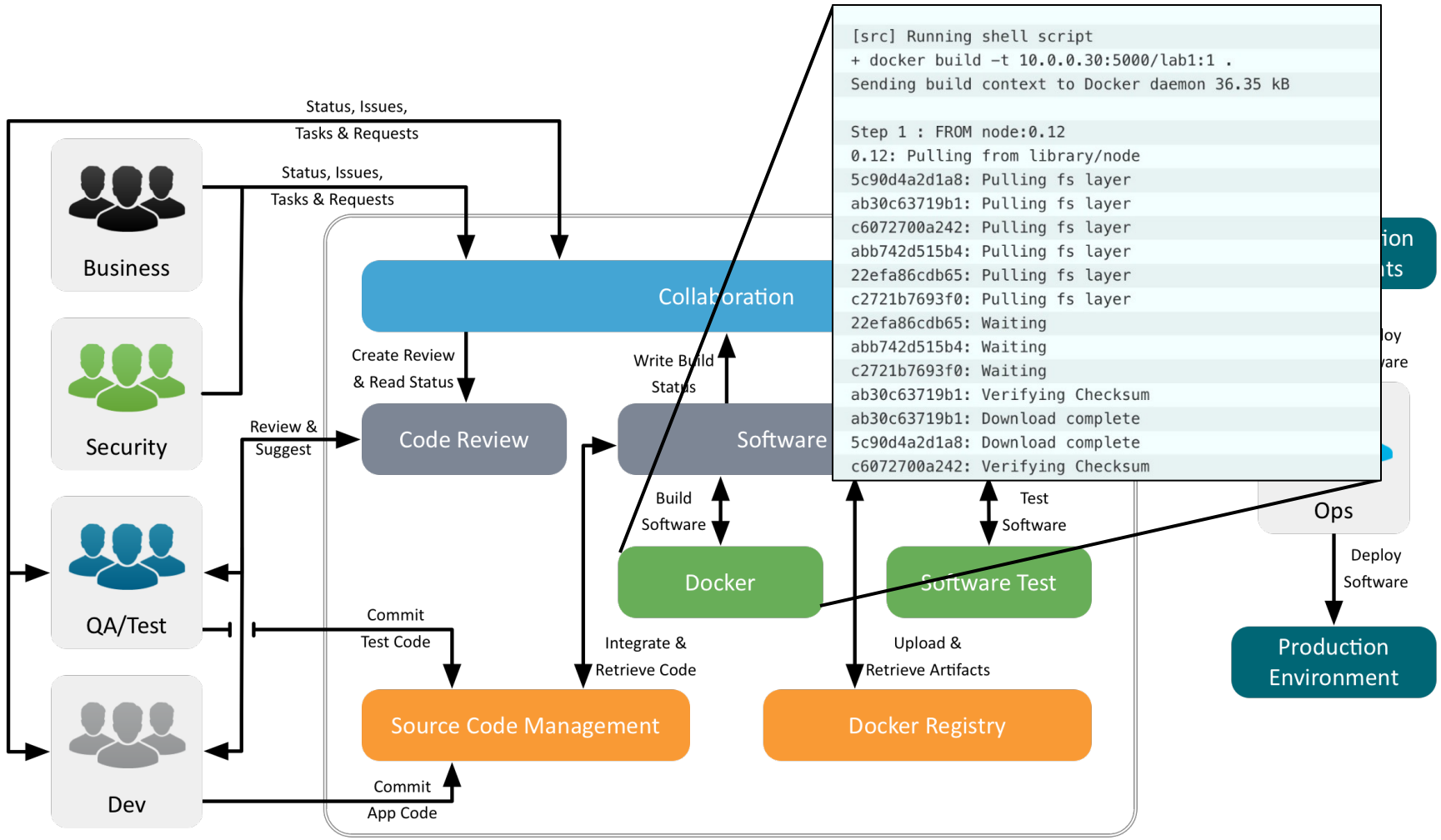
Docker for Continuous Integration



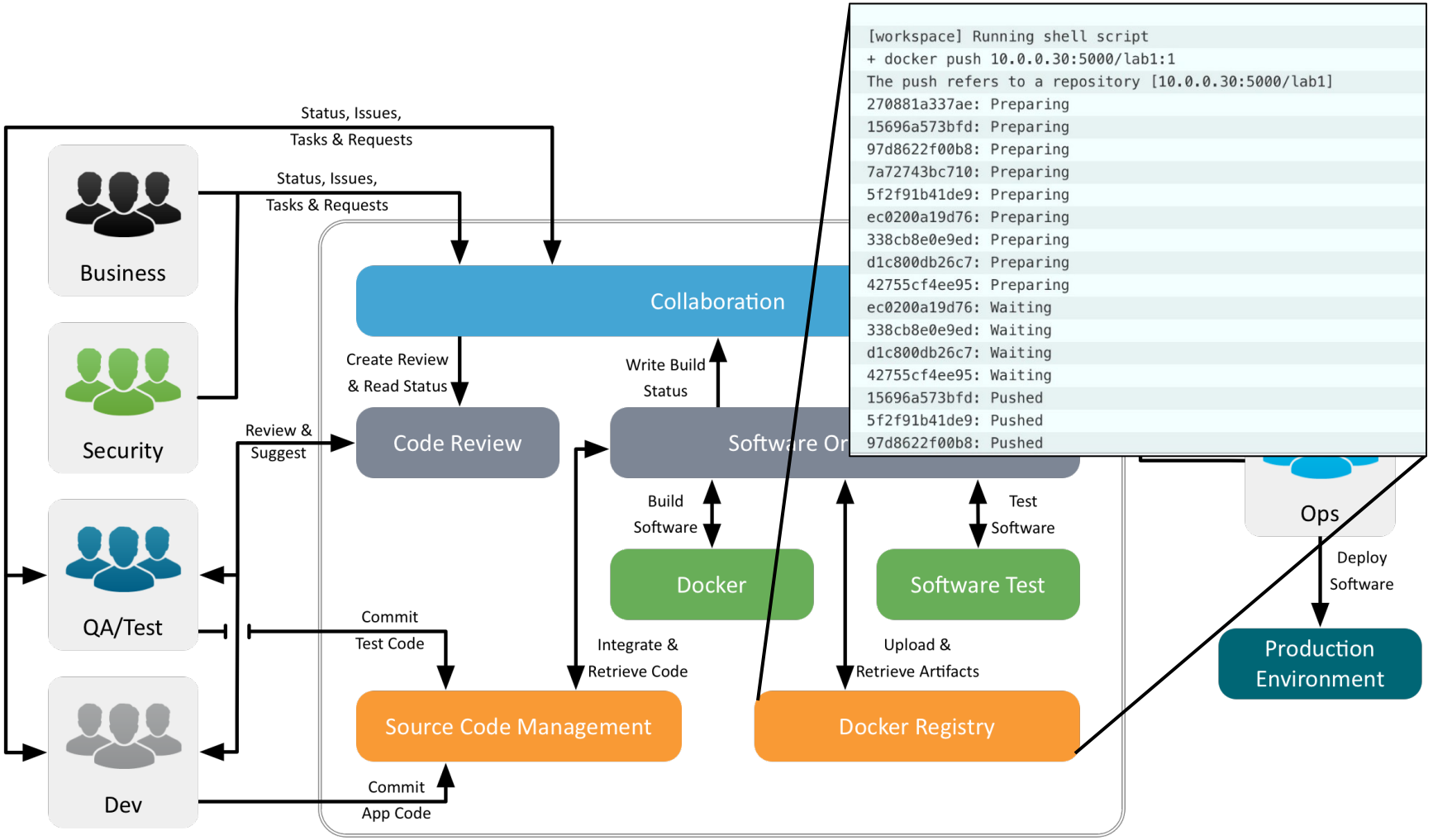
Docker for Continuous Integration



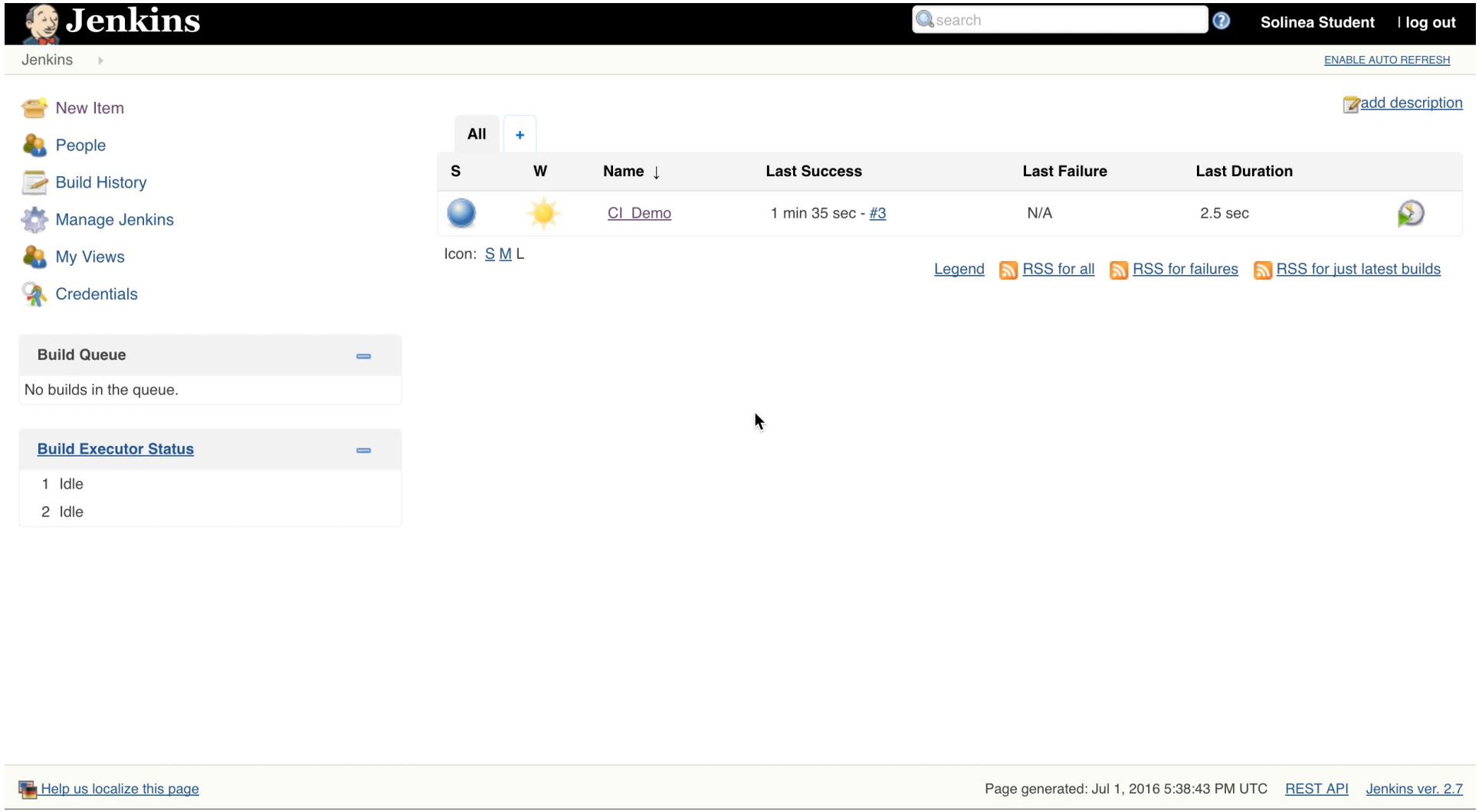
Docker for Continuous Integration



Docker for Continuous Integration



Docker for Continuous Integration

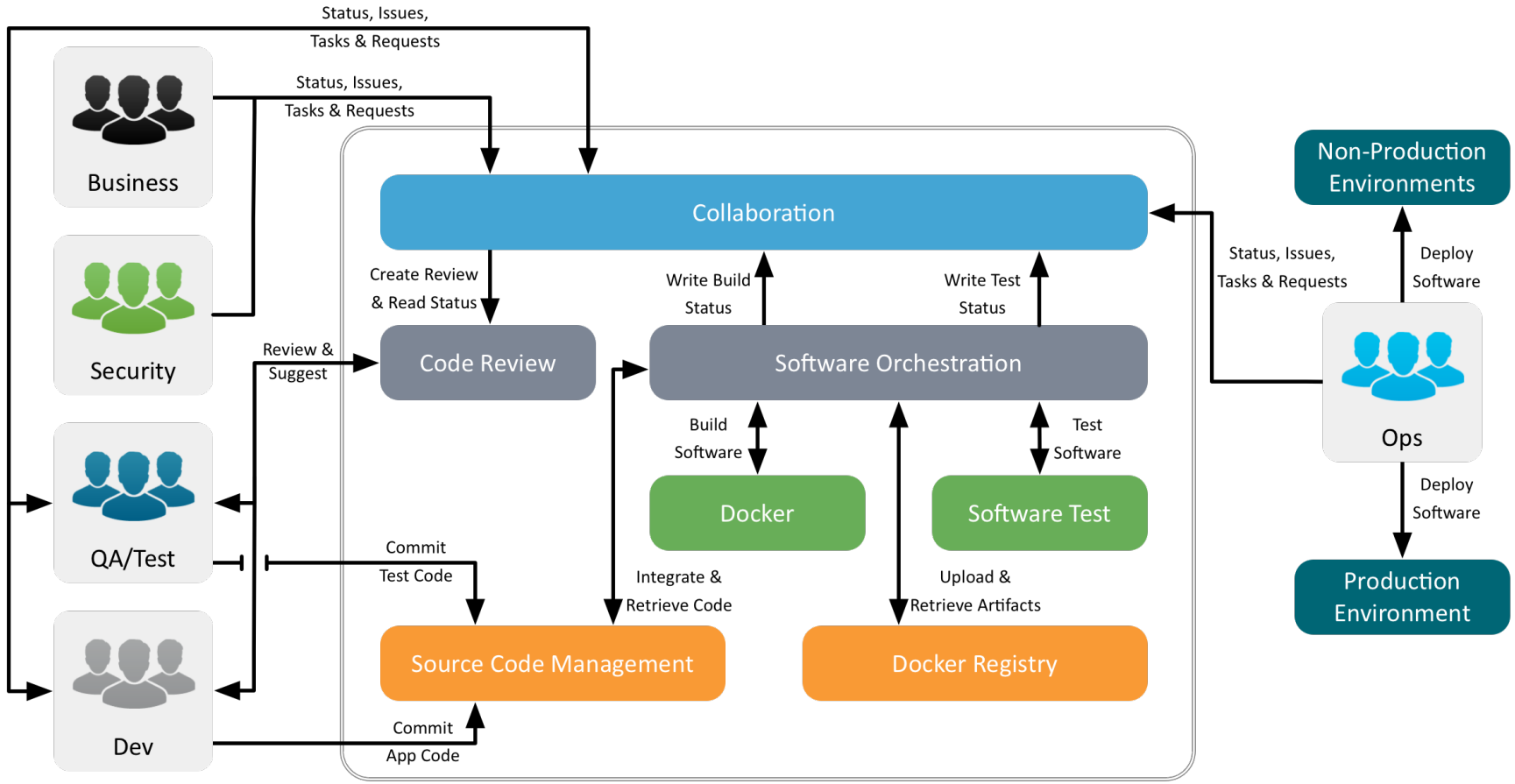


Docker Containers for CD

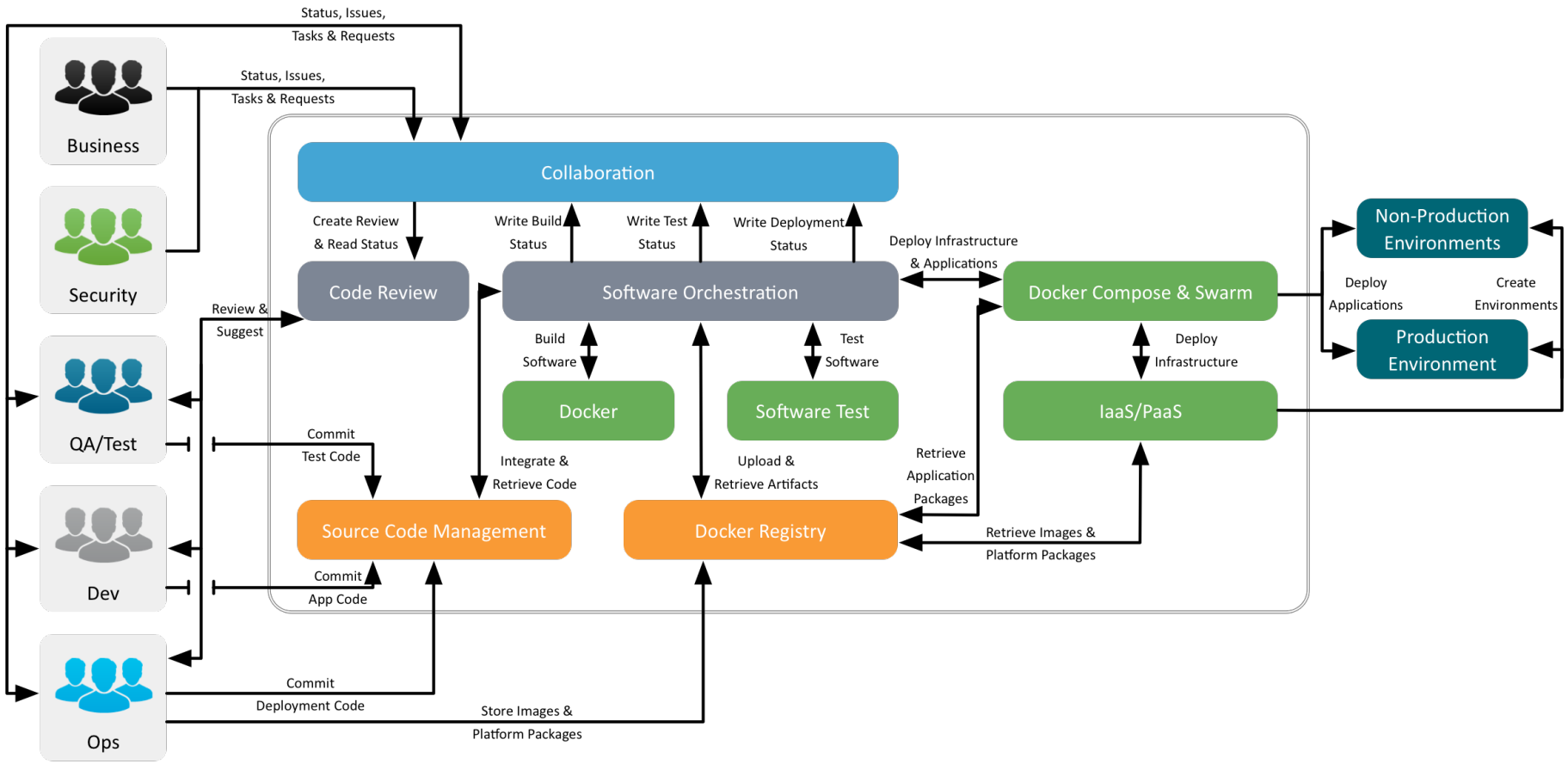
Docker for Continuous Delivery

- Frequent deliveries
  - Small, integrated change sets
- Automated deployment of Application Infrastructure
- **Automated deployment of Applications**
- Automated Application Testing/Validation

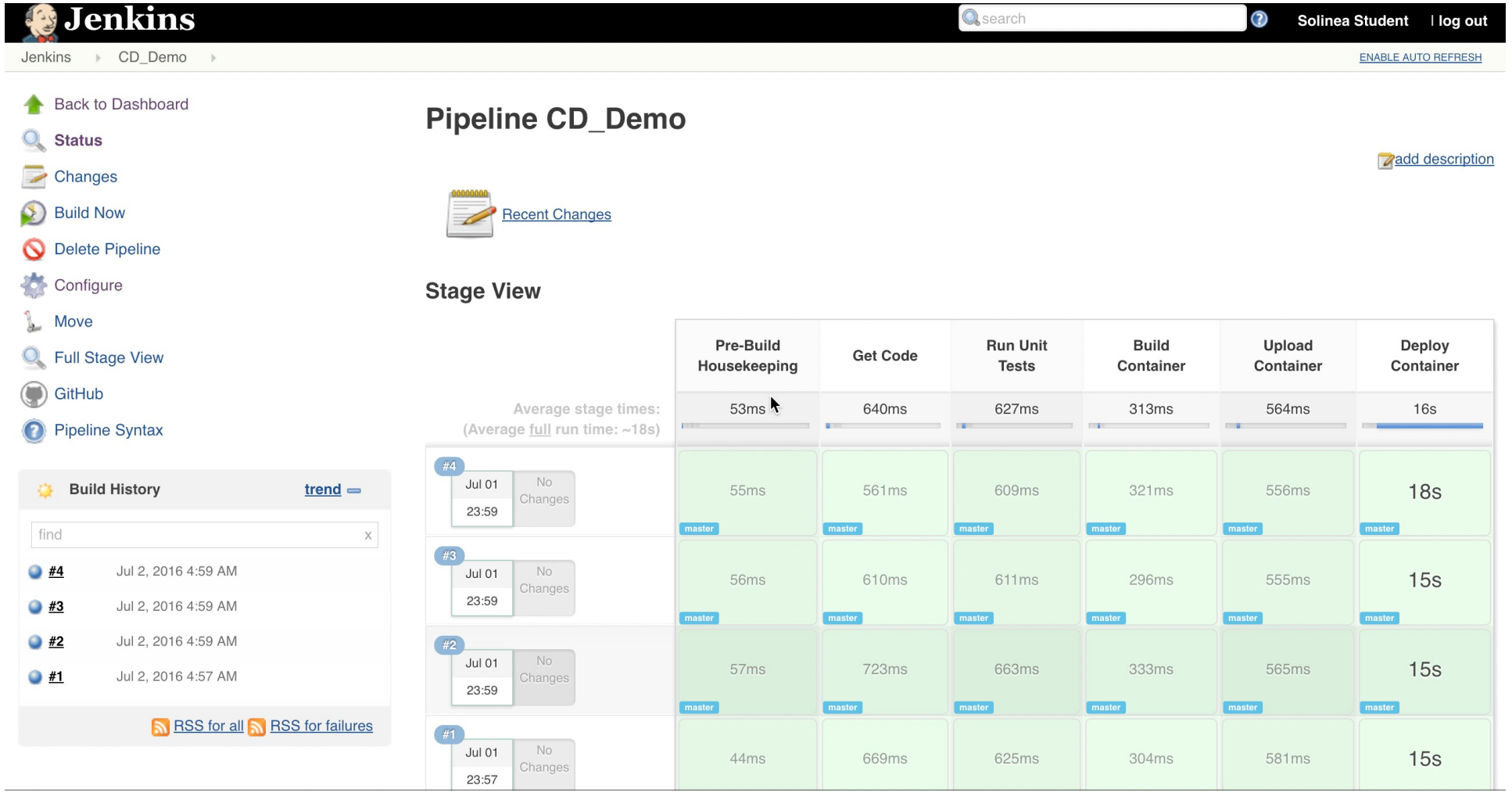
Docker for Continuous Delivery



Docker for Continuous Delivery



Docker for Continuous Delivery



Summary, Review, & Q&A

Lab: Capstone