

NobleProg

Docker & Kubernetes



The World's Local Training Provider

NobleProg® Limited 2017
All Rights Reserved

Overview

This Docker and Kubernetes training course is designed to provide attendees with a comprehensive foundation of Docker and Kubernetes technologies. The entire core features of Kubernetes is covered including, Pods, Labels, Volumes, Replication Controllers, Services and more. The course takes attendees from installation to management and usage through a combination of lecture and hands-on lab exercises where they will gain experience configuring and managing Kubernetes objects. In addition, this course will give attendees a solid understanding of Kubernetes including advanced features and security to run safely a Kubernetes cluster in production.

TARGET PUBLIC

This course is for people and teams that want to learn Kubernetes foundation and containers orchestration requirements to build services that are secure, robust, highly available, resilient and self-healing:

- Developers
- Operations
- DevOps
- QA Engineers
- IT Project Managers

Course Outline

Day 1

- Docker and Linux container technology
- Docker containers lifecycle
- Working with Docker images
- Network communication between containers
- Persistence of data in containers
- Container orchestration requirements and available options
- Introduction to Kubernetes and other orchestration systems
- Kubernetes core concepts: Pods, Labels, Controllers, Services, Secrets, Persistent Data Volumes, Claims, Namespaces, Quotas

Course Outline

Day 2

- Kubernetes reference architecture and its main components
- Containers network model in Kubernetes
- Service discovery, scaling and load balancing
- DNS for service discovery
- Ingress controller and reverse proxy
- Persistence of application state and the data volume model in Kubernetes
- Storage backend in Kubernetes: local, NFS, GlusterFS, Ceph
- Cluster management
- Deployment of applications and services on a Kubernetes cluster

Course Outline

Day 3

- Advanced controllers: Daemon Sets and Stateful Sets
- Job and Cron jobs
- Standalone pods
- Storage Classes and Dynamic Storage provisioning
- Network policies
- Securing a Kubernetes cluster
- Authentication, Authorization and Access Control
- Control Plane High Availability
- Auto Scaling
- Cluster monitoring
- Troubleshooting

Why should customers care about containers and microservices?

In reality, they shouldn't...

They do care about cloud native applications



“Unlimited” Scale



Global reach

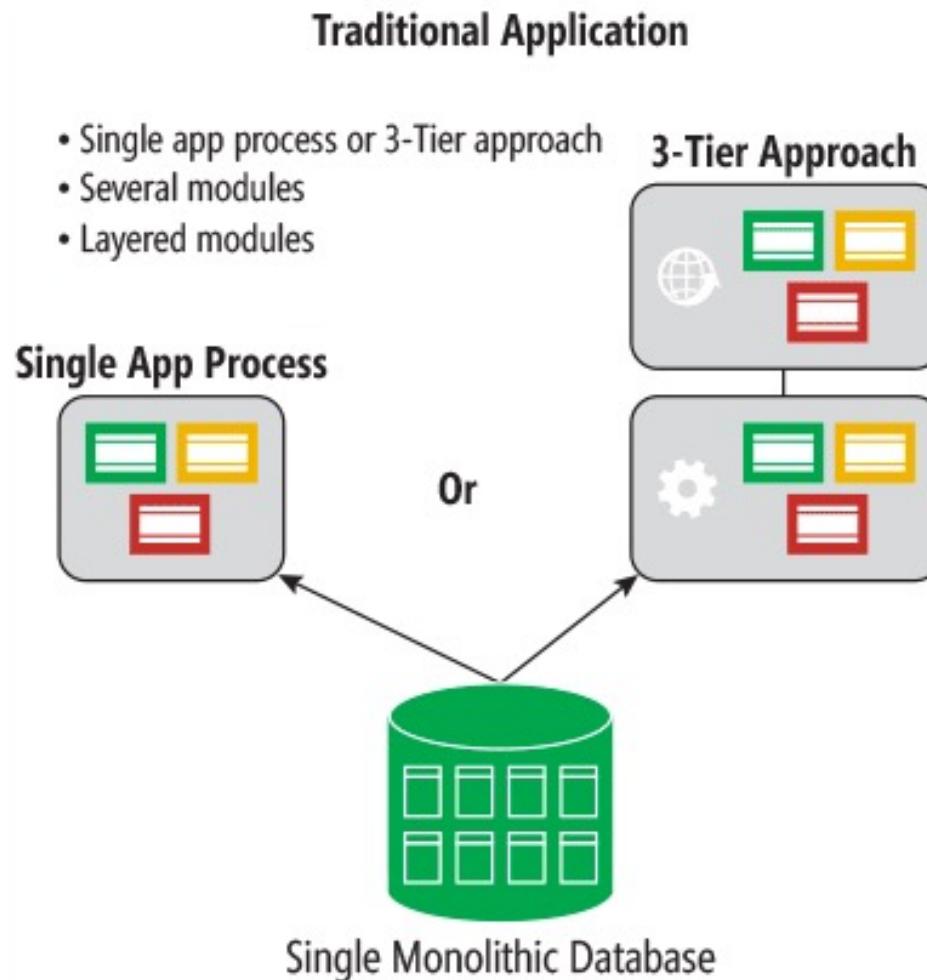
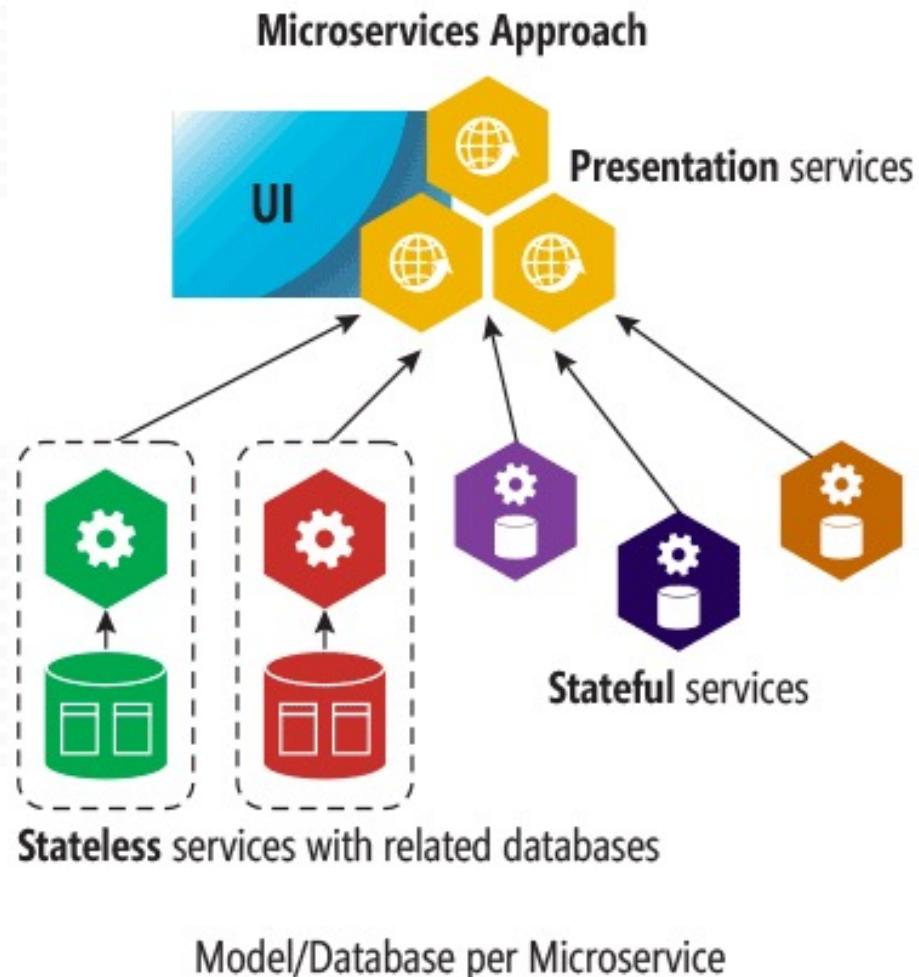


Rapid innovation
-> time to market



“Distributed apps are sufficiently complicated that they need to be flown by the instruments”

Microservices



microservices ≠ containers

microservices is an architectural
design approach

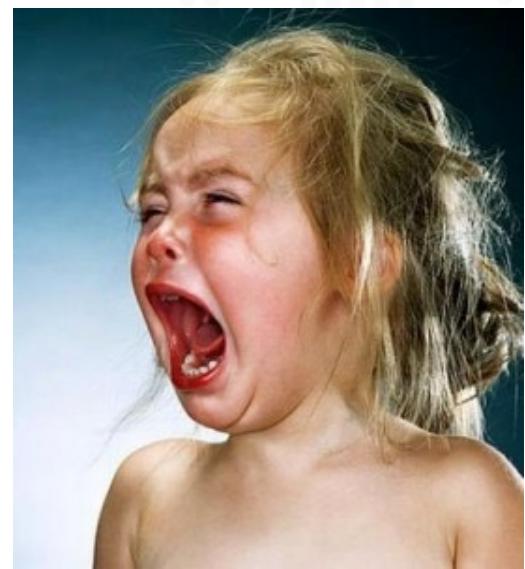
containers are an implementation
detail that often helps

Microservices Benefits

- ✓ Independent deployments
- ✓ Enables continuous delivery
- ✓ No downtime upgrades
- ✓ Improved scale and resource utilization per service
- ✓ Fault isolation
- ✓ Security isolation
- ✓ Services can be distributed across multiple servers or environments
- ✓ Multiple languages / diversity
- ✓ Smaller, focused teams
- ✓ Code can be organized around business capabilities
- ✓ Autonomous developer teams

Microservices – The Hard Part

- ✓ Deployment is complex
- ✓ Testing is difficult
- ✓ Debugging is difficult
- ✓ Monitoring/Logging is difficult
- ✓ New service versions must support old/new API contracts
- ✓ Distributed databases make transactions hard
- ✓ Cluster and orchestration tools overhead
- ✓ Distributed services adds more network communication
 - ✓ Increased network hops
 - ✓ Requires failure/recovery code
 - ✓ Need service discovery solution
- ✓ Advanced DevOps capability:
short-term pain for long-term gain



12-Factor Apps



THE TWELVE-FACTOR APP

The World's Local Training Provider

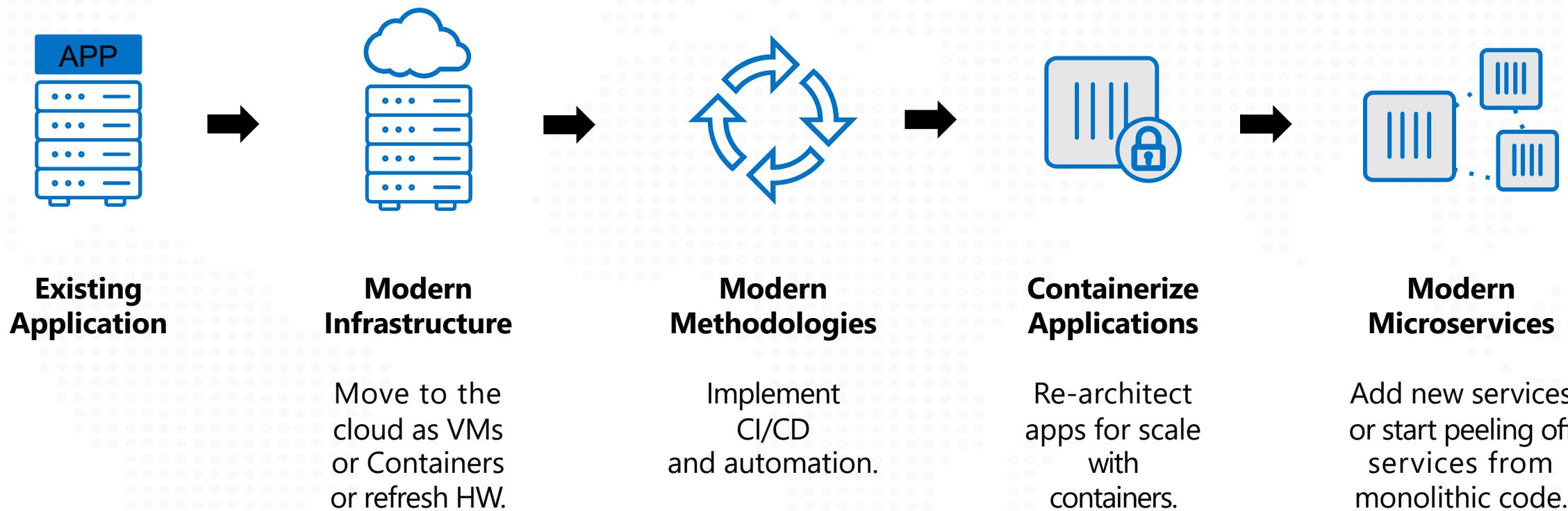
12-Factor Apps (1-5)

1. Single root repo; don't share code with another app
2. Deploy dependent libs with app
3. No config in code; read from environment vars
4. Handle unresponsive app dependencies robustly
5. Strictly separate build, release, & run steps
 - Build: Builds a version of the code repo & gathers dependencies
 - Release: Combines build with config Releaseld (immutable)
 - Run: Runs app in execution environment

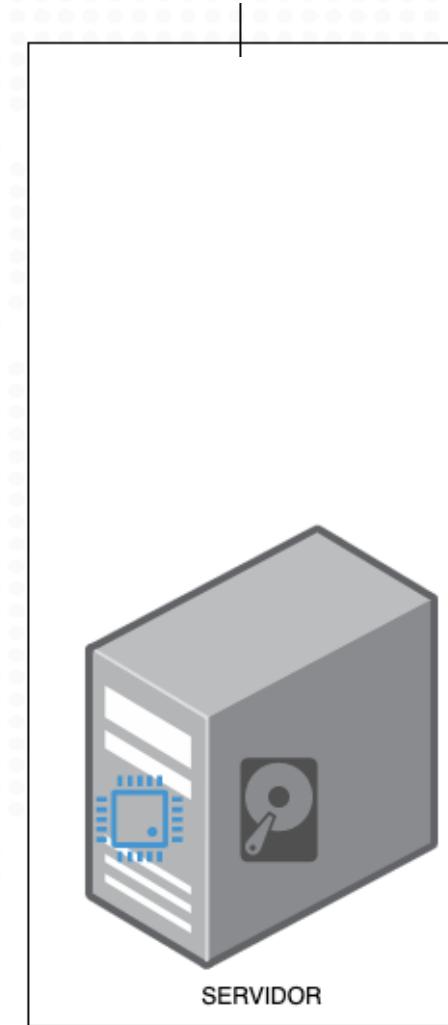
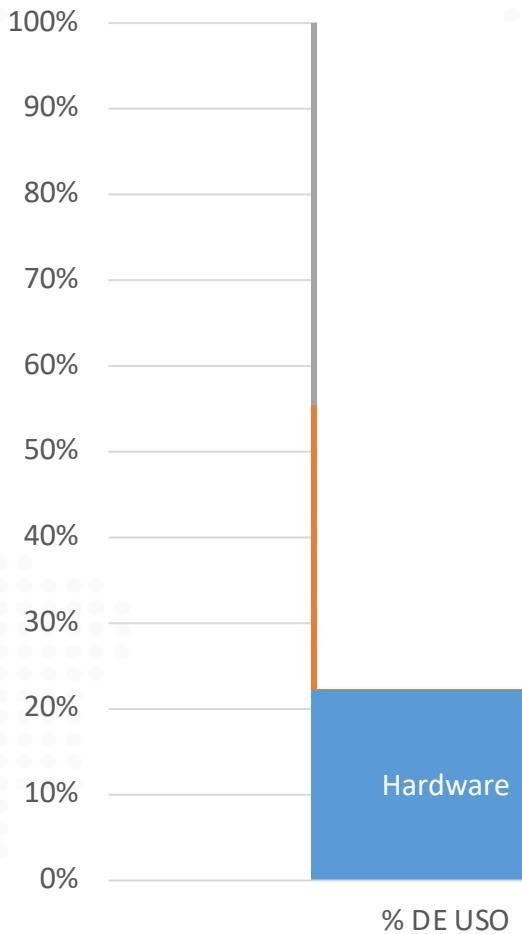
12-Factor Apps (6-12)

6. App executes as 1+ stateless process & shares nothing
7. App listens on ports; avoid using (web) host
8. Use processes for isolation; multiple for concurrency
9. Processes can crash/be killed quickly & start fast
10. Keep dev, staging, & prod environments similar
11. Log to stdout (dev=console; prod=file & archived)
12. Deploy & run admin tasks (scripts) as processes

From traditional app to modern app



¿Qué es un **contenedor**?



What is a container?

Slice up the OS to run multiple apps on a single VM

Every container has an isolated view and gets

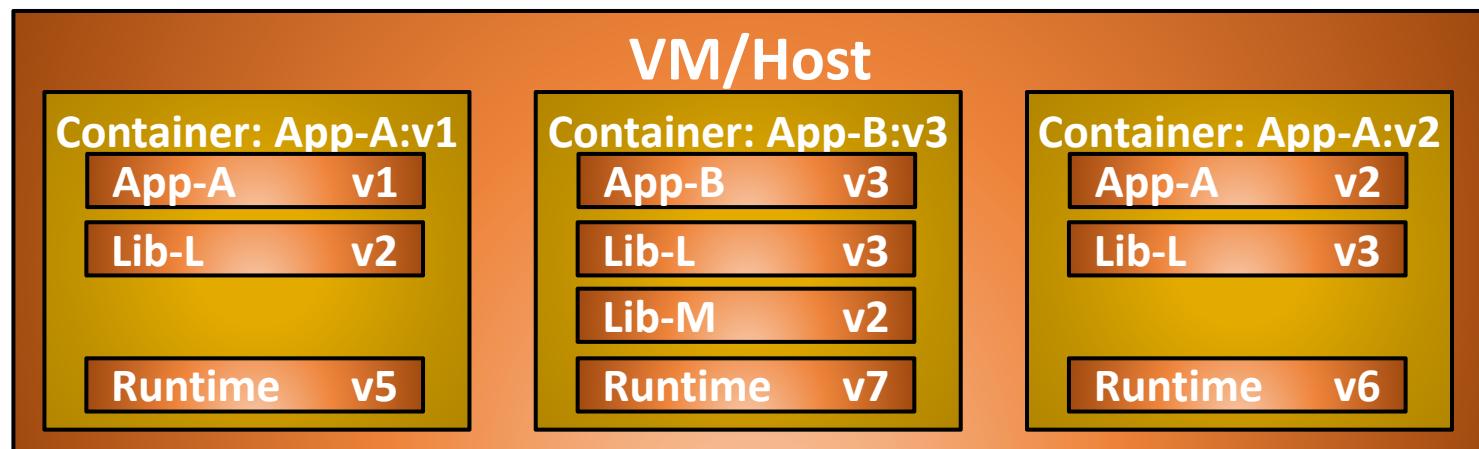
- it's own file system

- it's own PID0 and eth0 network interface

Container and apps share lifecycle

Shared kernel, very fast start-up, and repeatable execution

Cannot mix OS



Why containers?

Repeatable execution

immutable environment

reusable and portable code (“Build, Ship, and Run”)

Consistency across development, test, & production

Fast & agile app deployment; instant startup

Cloud portability

Density, partitioning, scale

Diverse developer framework support

Promotes microservices



NobleProg

Containers deliver speed, flexibility, and savings

Availability

62%

Report reduction in MTTR

10X

Cost reduction in maintaining existing applications

Hyper-scale

41%

Move workloads across private/public clouds

Eliminate

"works on my machine" issues

Agility

13X

More software releases

65%

Reduction in developer onboarding time

Docker, Docker, Docker

Containers have been around for many years

Linux kernel: cgroups, namespaces

Docker Inc. did not invent them

They created open source software to build and manage containers



Docker makes containers easy

Super easy. Fast learning curve

Docker is a container format and a set of tools

Docker CLI, Docker Engine, Docker Swarm, Docker Compose,
Docker Machine

Docker

- Docker is
 - ephemeral (short-lived)
 - isomorphic (unchanging)
 - deterministic (same every time)

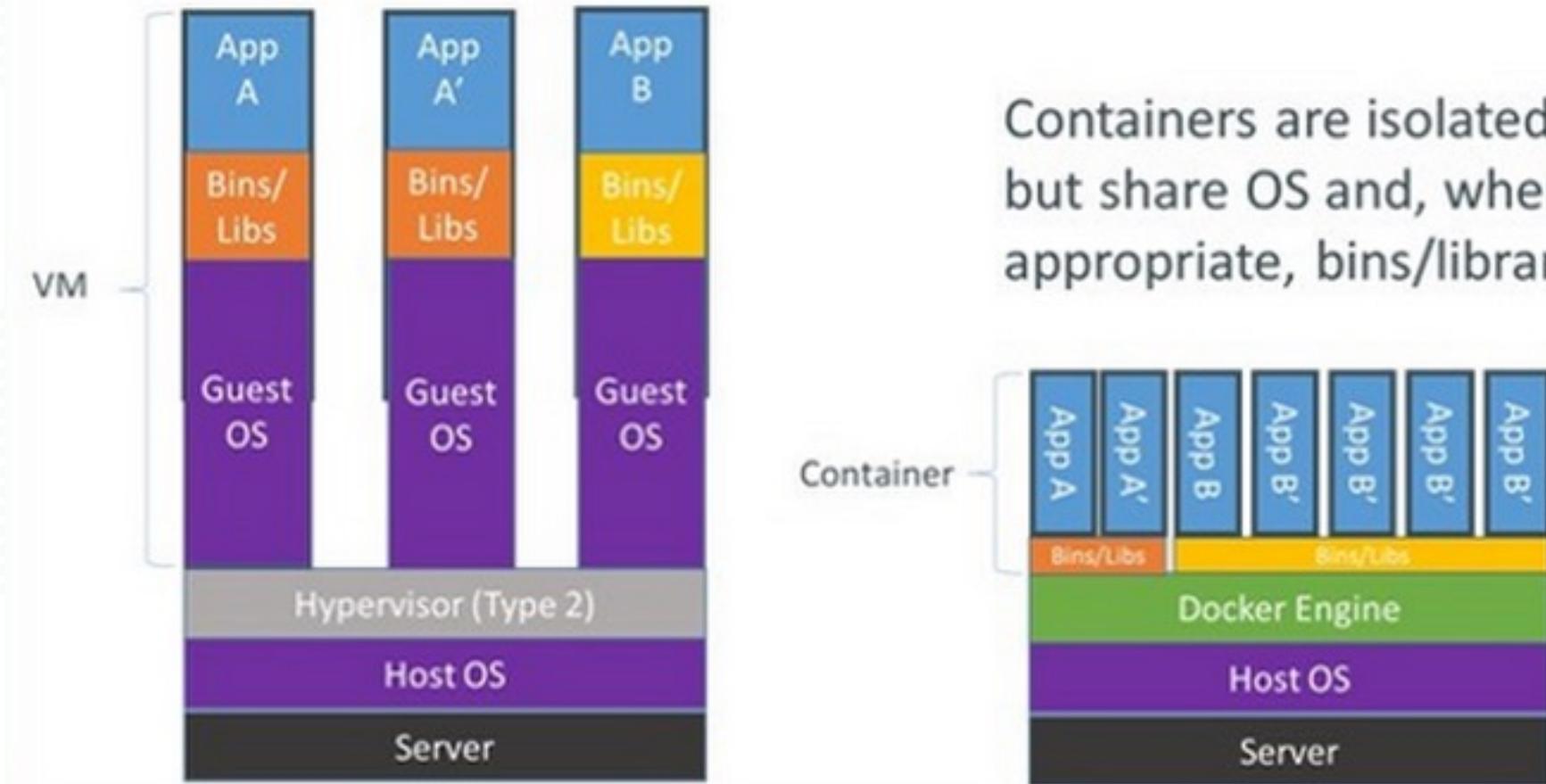
Therefore Docker is

- ideal platform for dev & ops
- clearly delineates duties
- clean communication strategy

Docker

- **What is Docker?**
- Docker is an ecosystem around Container Virtualization
- **What are Containers?**
- Light-weight kernel virtualization
- **What is Docker?**
- A suite of command-line tools for creating, running, and managing containers

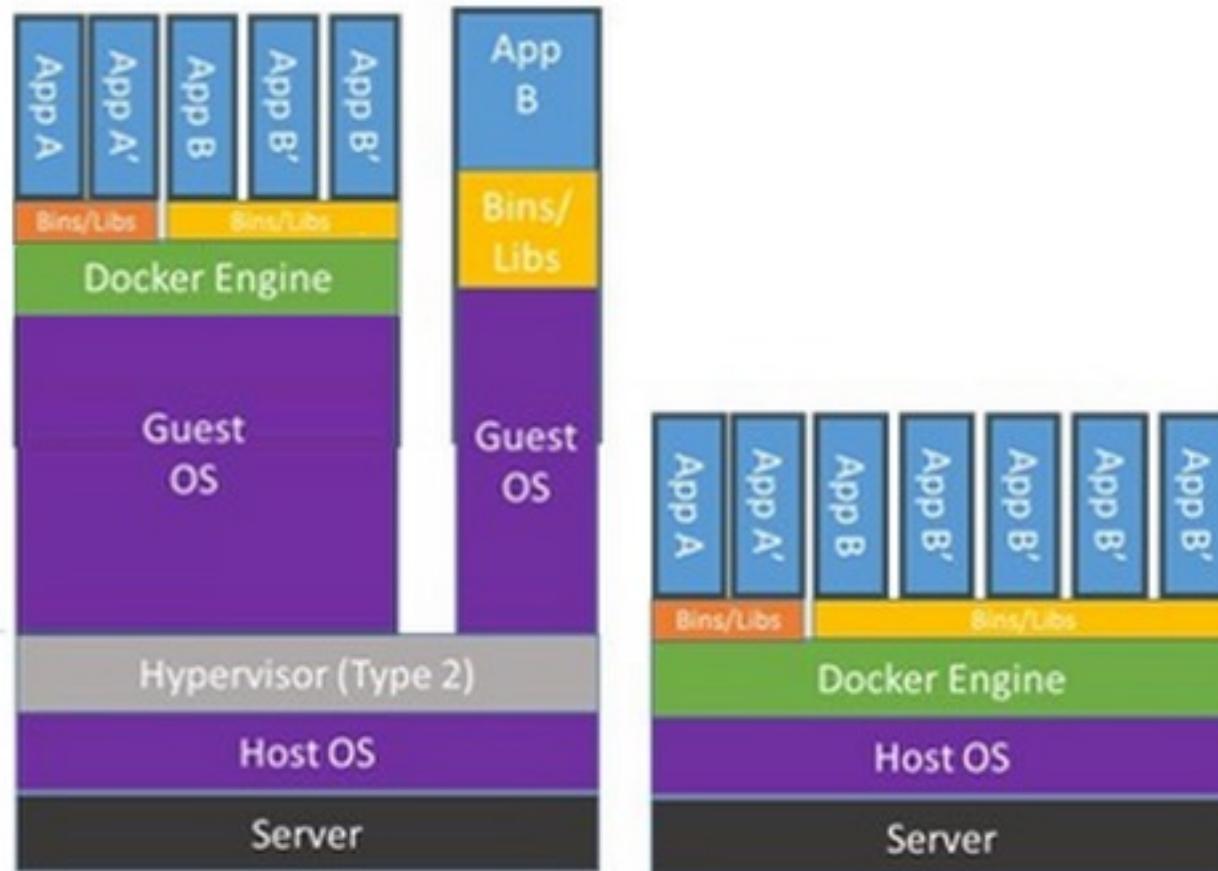
Containers vs VMs



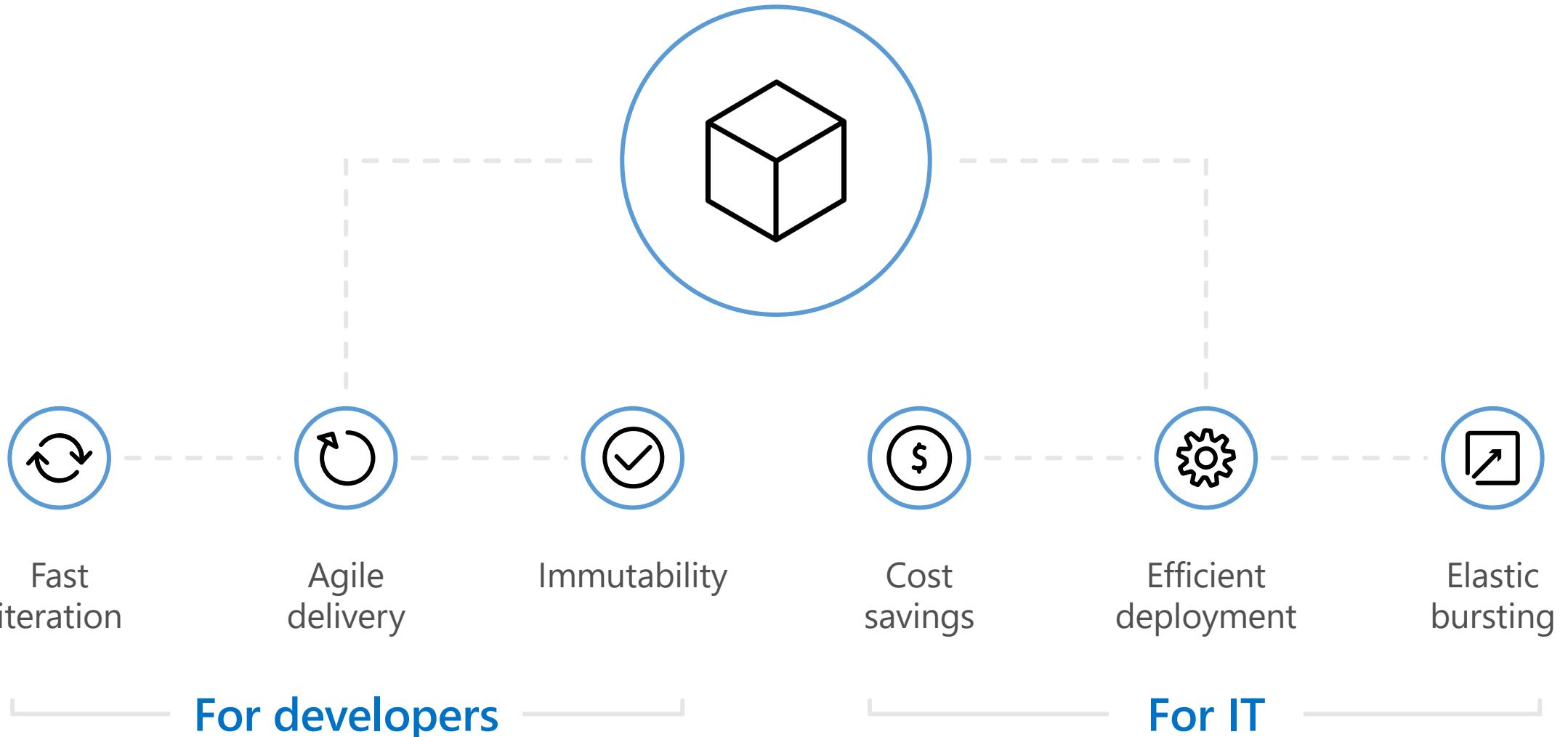
Containers

- **Containers**
- Containers virtualize and share the host kernel
- Containers must run on the kernel for which they were built:
- Linux containers run on a Linux host
- Windows containers run on Windows Server host

Host Docker in a VM



The container advantage



Why containers?

Repeatable execution

immutable environment

reusable and portable code (“Build, Ship, and Run”)

Consistency across development, test, & production

Fast & agile app deployment; instant startup

Cloud portability

Density, partitioning, scale

Diverse developer framework support

Promotes microservices

The elements of orchestration



Scheduling



Affinity/anti-affinity



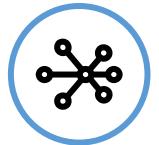
Health monitoring



Failover



Scaling



Networking



Service discovery



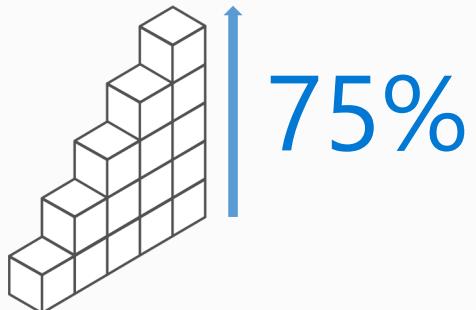
Coordinated app upgrades

Containers and Kubernetes **momentum**

"By 2020, more than **50%** of enterprises will run **mission-critical, containerized cloud-native applications** in production."

Gartner

The average size of a container deployment has grown 75% in one year.¹



Half of container environment is orchestrated.¹

77% of companies² who use container orchestrators choose Kubernetes.

77%

Larger companies are leading the adoption.¹

Nearly 50% of organizations¹ running 1000 or more hosts have adopted containers.

50%

¹ Datadog report: 8 Surprising Facts About Real Docker Adoption

² CNCF survey: cloud-native-technologies-scaling-production-applications

VM Host

Container



Dependencies



Container



Dependencies



Container



Dependencies

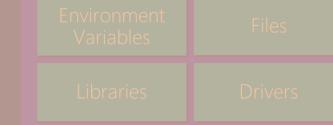


VM Host

Container



Dependencies



Container



Dependencies



Container



Dependencies

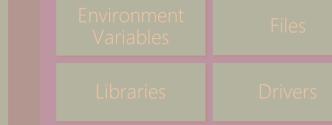


VM Host

Container



Dependencies



Container



Dependencies



Container



Dependencies



VM Host

Container



Dependencies



Container



Dependencies



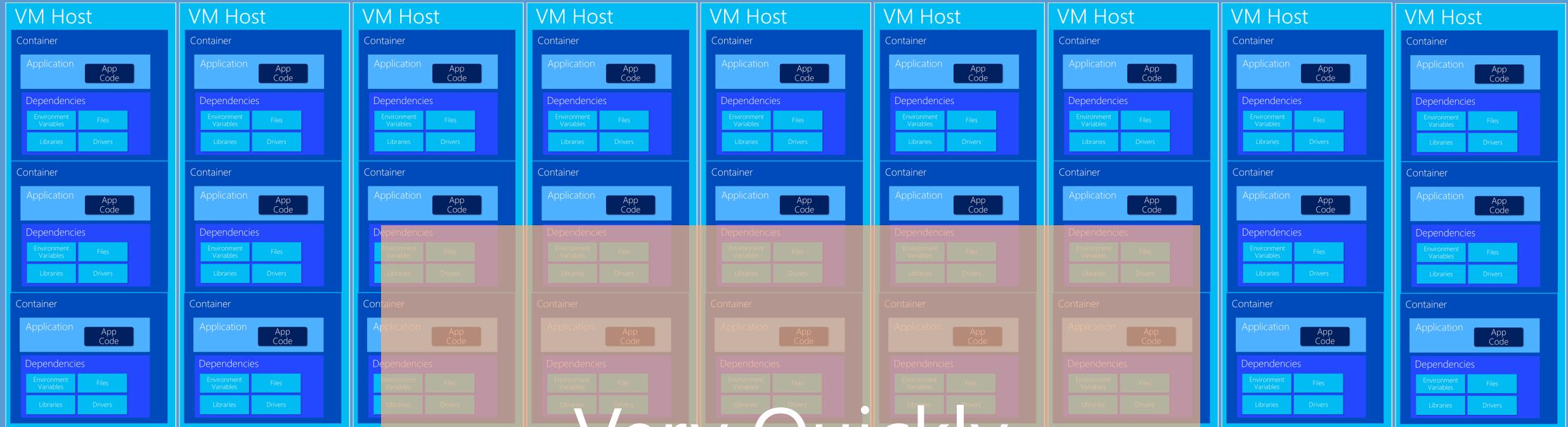
Container



Dependencies



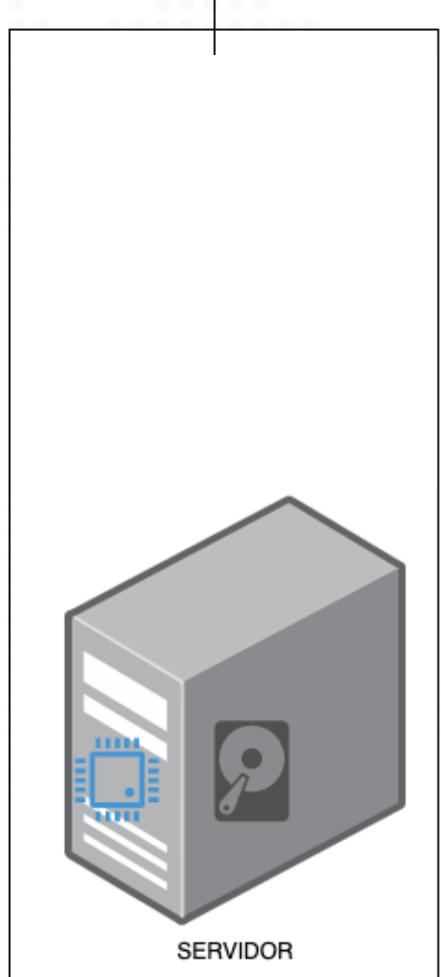
But things get
complicated quickly



Very Quickly



¿Qué es Kubernetes?



Container Management at Scale

Cluster Management: deploy and manage cluster resources

Scheduling: where containers run

Lifecycle & Health: keep containers running despite failure

Service Discovery & Load Balancing: where are my containers and how to

Security: control access to cluster resources and protect secrets

Scaling: make sets of containers elastic in number

Image repository: centralized, secure Docker container images

Continuous Delivery: CI/CD pipeline and workflow

Logging & Monitoring: track what's happening in containers and cluster

Storage volumes: persistent data for containers

What is Kubernetes?

- Background
 - "Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications"
 - Schedules and runs application containers across a cluster of machines
 - Kubernetes v1.0 released on July 21, 2015. Joe Beda, Brendan Burns, & Craig McLuckie
- Key features
 - Declarative infrastructure
 - Self-healing
 - Horizontal scaling
 - Automated rollouts and rollbacks
 - Service discovery and load balancing
 - Automatic bin packing
 - Storage orchestration
 - Secret and configuration management
 - Not a PaaS platform



Kubernetes en la vida real...

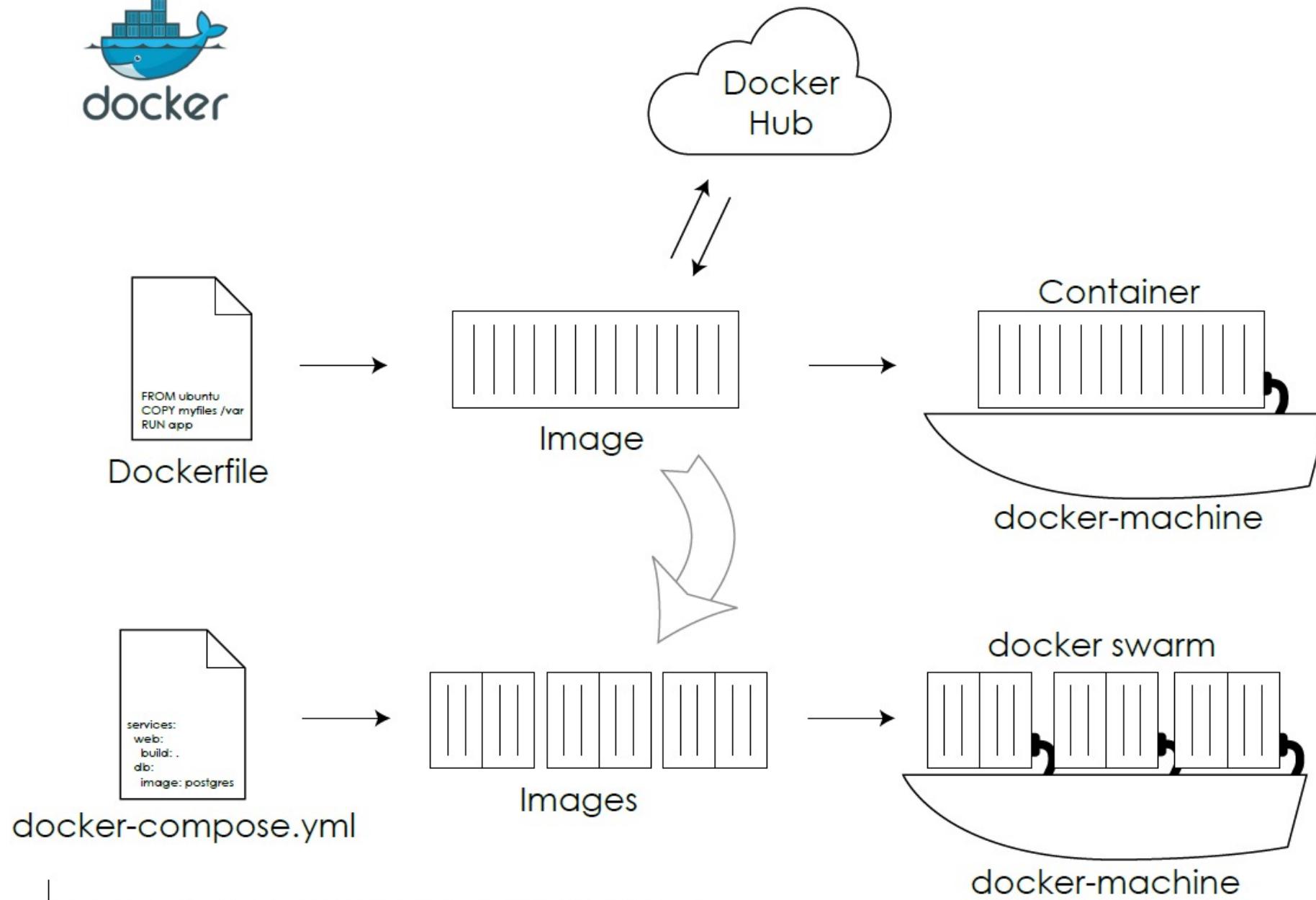
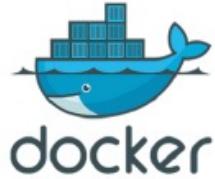


U.S. Air Force Photo by Airman 1st Class Luis A. Ruiz-Vazquez

Famed spy plane gets AI upgrade via Kubernetes

BY [SUSAN MILLER](#) | OCT 07, 2020

The Air Force has equipped a legacy U-2 surveillance aircraft with machine learning thanks to Kubernetes, an open-source container-orchestration system that automates the application deployment, scaling and management.



Installation

- **Installing Docker**
- Linux, Mac, or Windows 10:
<https://docs.docker.com/engine/install/>
- Windows Server:
<https://docs.microsoft.com/en-us/virtualization/windowscontainers/quick-start/set-up-environment>

Docker Desktop

- **Docker Desktop**
- Automatically spins up a Linux VM
- Connects docker command-line to the VM
- Proxies localhost traffic to the VM's containers

<https://docs.docker.com/desktop/>

Our first containers



Colorful plastic tubs

Objectives

At the end of this lesson, you will have:

- Seen Docker in action.
- Started your first containers.

Hello World

In your Docker environment, just run the following command:

```
$ docker run busybox echo hello world  
hello world
```

(If your Docker install is brand new, you will also see a few extra lines, corresponding to the download of the **busybox** image.)

That was our first container!

- We used one of the smallest, simplest images available: **busybox**.
- **busybox** is typically used in embedded systems (phones, routers...)
- We ran a single process and echo'ed `hello world`.

A more useful container

Let's run a more exciting container:

```
$ docker run -it ubuntu  
root@04c0bb0a6c07:/#
```

- This is a brand new container.
- It runs a bare-bones, no-frills `ubuntu` system.
- `-it` is shorthand for `-i -t`.
 - `-i` tells Docker to connect us to the container's stdin.
 - `-t` tells Docker that we want a pseudo-terminal.

Do something in our container

Try to run `figlet` in our container.

```
root@04c0bb0a6c07:/# figlet hello  
bash: figlet: command not found
```

Alright, we need to install it.

Install a package in our container

We want `figlet`, so let's install it:

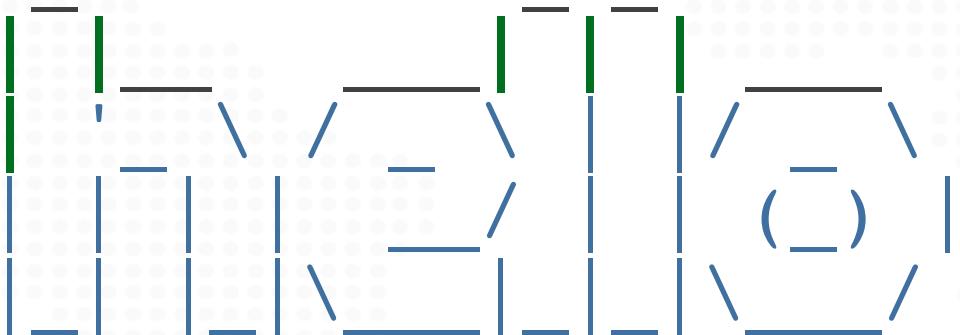
```
root@04c0bb0a6c07:/# apt-get update  
...  
Fetched 1514 kB in 14s (103 kB/s)  
Reading package lists... Done  
root@04c0bb0a6c07:/# apt-get install figlet  
Reading package lists... Done  
...
```

One minute later, `figlet` is installed!

Try to run our freshly installed program

The `figlet` program takes a message as parameter.

```
root@04c0bb0a6c07:/# figlet hello
```



Beautiful! 😍

Counting packages in the container

Let's check how many packages are installed there.

```
root@04c0bb0a6c07:/# dpkg -l | wc -l  
97
```

- `dpkg -l` lists the packages installed in our container
- `wc -l` counts them

How many packages do we have on our host?

Counting packages on the host

Exit the container by logging out of the shell, like you would usually do.

(E.g. with ^D or `exit`)

```
root@04c0bb0a6c07:/# exit
```

Now, try to:

- run `dpkg -l | wc -l`. How many packages are installed?
- run `figlet`. Does that work?

Comparing the container and the host

Exit the container by logging out of the shell, with ^D or `exit`.
Now try to run `figlet`. Does that work?

(It shouldn't; except if, by coincidence, you are running on a machine where `figlet` was installed before.)

Host and containers are independent things

- We ran an `ubuntu` container on an Linux/Windows/macOS host.
- They have different, independent packages.
- Installing something on the host doesn't expose it to the container.
- And vice-versa.
- Even if both the host and the container have the same Linux distro!
- We can run *any container on any host*.
(One exception: Windows containers can only run on Windows hosts; at least for now.)

Where's our container?

- Our container is now in a *stopped* state.
- It still exists on disk, but all compute resources have been freed up.
- We will see later how to get back to that container.

Starting another container

What if we start a new container, and try to run `figlet` again?

```
$ docker run -it ubuntu
root@b13c164401fb:/# figlet
bash: figlet: command not found
```

- We started a *brand new container*.
- The basic Ubuntu image was used, and `figlet` is not here.

Where's my container?

- Can we reuse that container that we took time to customize?

We can, but that's not the default workflow with Docker.

- What's the default workflow, then?

Always start with a fresh container. If we need something installed in our container, build a custom image.

- That seems complicated!

We'll see that it's actually pretty easy!

- And what's the point?

This puts a strong emphasis on automation and repeatability. Let's see why ...

Pets vs. Cattle

- In the “pets vs. cattle” metaphor, there are two kinds of servers.
- Pets:
 - have distinctive names and unique configurations
 - when they have an outage, we do everything we can to fix them
- Cattle:
 - have generic names (e.g. with numbers) and generic configuration
 - configuration is enforced by configuration management, golden images ...
 - when they have an outage, we can replace them immediately with a new server
- What's the connection with Docker and containers?

Local development environments

- When we use local VMs (with e.g. VirtualBox or VMware), our workflow looks like this:
 - create VM from base template (Ubuntu, CentOS...)
 - install packages, set up environment
 - work on project
 - when done, shut down VM
 - next time we need to work on project, restart VM as we left it
 - if we need to tweak the environment, we do it live
- Over time, the VM configuration evolves, diverges.
- We don't have a clean, reliable, deterministic way to provision that environment.

Local development with Docker

- With Docker, the workflow looks like this:
 - create container image with our dev environment
 - run container with that image
 - work on project
 - when done, shut down container
 - next time we need to work on project, start a new container
 - if we need to tweak the environment, we create a new image
- We have a clear definition of our environment, and can share it reliably with others.
- Let's see in the next chapters how to bake a custom image with **figlet!**

Dockerfile

```
FROM node
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
ENV NODE_ENV production
ENV PORT 3000
EXPOSE 3000
CMD ["npm", "start"]
```

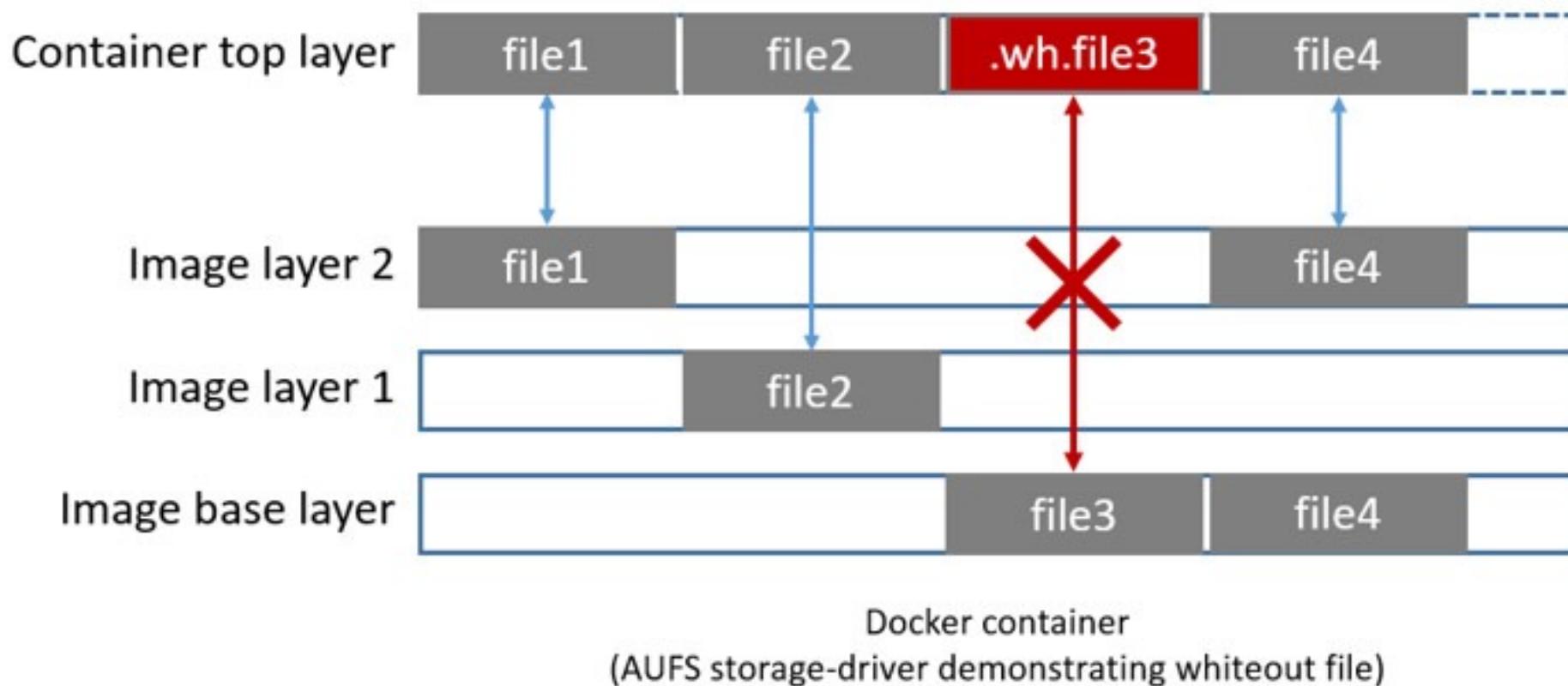
Dockerfile

- FROM ...image we start from
- RUN ...run setup content
- COPY ...copy content into the container
- CMD [...]the thing to start

Filesystem

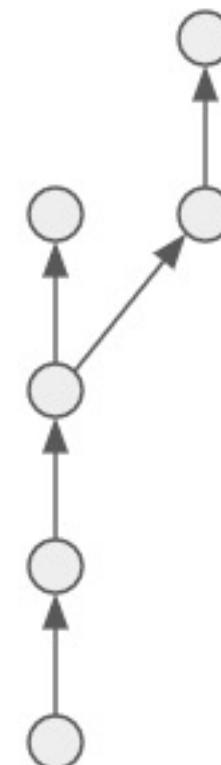
- **Layered Filesystem**
- Each line in the Dockerfile creates a layer
- Layer is isomorphic
- Layer hash is SHA256 of content and parent hash

Filesystem



Layered Filesystem

- Only downloads each layer to disk once
- because layers don't change



Local development workflow with Docker



Construction site

Objectives

At the end of this section, you will be able to:

- Share code between container and host.
- Use a simple local development workflow.

Local development in a container

We want to solve the following issues:

- “Works on my machine”
- “Not the same version”
- “Missing dependency”

By using Docker containers, we will get a consistent development environment.

Working on the “namer” application

- We have to work on some application whose code is at:
<https://github.com/jpetazzo/namer>
- What is it? We don't know yet!
- Let's download the code.

```
$ git clone https://github.com/jpetazzo/namer
```

Looking at the code

```
$ cd namer
$ ls -1
company_name_generator.rb
config.ru
docker-compose.yml
Dockerfile
Gemfile
```

—
Aha, a `Gemfile`! This is Ruby. Probably. We know this. Maybe?

Looking at the Dockerfile

```
FROM ruby

COPY . /src
WORKDIR /src
RUN bundle install

CMD ["rackup", "--host", "0.0.0.0"]
EXPOSE 9292
```

- This application is using a base `ruby` image.
- The code is copied in `/src`.
- Dependencies are installed with `bundler`.
- The application is started with `rackup`.
- It is listening on port 9292.

Building and running the “namer” application

- Let’s build the application with the Dockerfile!

—
\$ docker build -t namer .

-
- Then run it. *We need to expose its ports.*

—
\$ docker run -dP namer

-
- Check on which port the container is listening.

—
\$ docker ps -l

Connecting to our application

- Point our browser to our Docker node, on the port allocated to the container.
 -
 - Hit “reload” a few times.
 -
 - This is an enterprise-class, carrier-grade, ISO-compliant company name generator!
(With 50% more bullshit than the average competition!)
(Wait, was that 50% more, or 50% less? Anyway!)
- web application 1

Making changes to the code

Option 1:

- Edit the code locally
- Rebuild the image
- Re-run the container

Option 2:

- Enter the container (with `docker exec`)
- Install an editor
- Make changes from within the container

Option 3:

- Use a *bind mount* to share local files with the container
- Make changes locally
- Changes are reflected in the container

Our first volume

We will tell Docker to map the current directory to `/src` in the container.

```
$ docker run -d -v $(pwd) :/src -P namer
```

- `-d`: the container should run in detached mode (in the background).
- `-v`: the following host directory should be mounted inside the container.
- `-P`: publish all the ports exposed by this image.
- `namer` is the name of the image we will run.
- We don't specify a command to run because it is already set in the Dockerfile via `CMD`.

Note: on Windows, replace `$(pwd)` with `%cd%` (or `$(pwd)` if you use PowerShell).

Mounting volumes inside containers

The `-v` flag mounts a directory from your host into your Docker container.

The flag structure is:

`[host-path] : [container-path] : [rw | ro]`

- `[host-path]` and `[container-path]` are created if they don't exist.
- You can control the write status of the volume with the `ro` and `rw` options.
- If you don't specify `rw` or `ro`, it will be `rw` by default.

Hold your horses... and your mounts

- The `-v /path/on/host:/path/in/container` syntax is the “old” syntax
- The modern syntax looks like this:
`--mount type=bind,source=/path/on/host,target=/path/in/container`
- `--mount` is more explicit, but `-v` is quicker to type
- `--mount` supports all mount types; `-v` doesn’t support `tmpfs` mounts
- `--mount` fails if the path on the host doesn’t exist; `-v` creates it

With the new syntax, our command becomes:

```
docker run --mount=type=bind,source=$(pwd),target=/src  
-dP namer
```

Testing the development container

- Check the port used by our new container.

```
$ docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	NAMES
STATUS	PORtS			
045885b68bc5	namer	rackup	3 seconds ago	Up
...	0.0.0.0:32770->9292/tcp	...		

- Open the application in your web browser.

Making a change to our application

Our customer really doesn't like the color of our text. Let's change it.

```
$ vi company_name_generator.rb
```

And change

```
color: royalblue;
```

To:

```
color: red;
```

Viewing our changes

- Reload the application in our browser.

—

- The color should have changed.

web application 2

Understanding volumes

- Volumes are *not* copying or synchronizing files between the host and the container
- Changes made in the host are immediately visible in the container (and vice versa)
- When running on Linux:
 - volumes and bind mounts correspond to directories on the host
 - if Docker runs in a Linux VM, these directories are in the Linux VM
- When running on Docker Desktop:
 - volumes correspond to directories in a small Linux VM running Docker
 - access to bind mounts is translated to host filesystem access (a bit like a network filesystem)

Docker Desktop caveats

- When running Docker natively on Linux, accessing a mount = native I/O
- When running Docker Desktop, accessing a bind mount = file access translation
 - That file access translation has relatively good performance *in general* (watch out, however, for that big `npm install` working on a bind mount!)
- There are some corner cases when watching files (with mechanisms like inotify)
- Features like “live reload” or programs like `entr` don’t always behave properly
 - (due to e.g. file attribute caching, and other interesting details!)

Trash your servers and burn your code

(This is the title of a [2013 blog post](#) by Chad Fowler, where he explains the concept of immutable infrastructure.)

- - Let's majorly mess up our container.
(Remove files or whatever.)
 - Now, how can we fix this?
- - Our old container (with the blue version of the code) is still running.
 - See on which port it is exposed:
`docker ps`
 - Point our browser to it to confirm that it still works fine.

Immutable infrastructure in a nutshell

- Instead of *updating* a server, we deploy a new one.
- This might be challenging with classical servers, but it's trivial with containers.
- In fact, with Docker, the most logical workflow is to build a new image and run it.
- If something goes wrong with the new image, we can always restart the old one.
- We can even keep both versions running side by side.

If this pattern sounds interesting, you might want to read about *blue/green deployment* and *canary deployments*.

Recap of the development workflow

1. Write a Dockerfile to build an image containing our development environment. (Rails, Django, ... and all the dependencies for our app)
2. Start a container from that image. Use the `-v` flag to mount our source code inside the container.
3. Edit the source code outside the container, using familiar tools. (vim, emacs, textmate...)
4. Test the application. (Some frameworks pick up changes automatically. Others require you to Ctrl-C + restart after each modification.)
5. Iterate and repeat steps 3 and 4 until satisfied.
6. When done, commit+push source code changes.

Debugging inside the container

Docker has a command called `docker exec`.

It allows users to run a new process in a container which is already running.

If sometimes you find yourself wishing you could SSH into a container: you can use `docker exec` instead.

You can get a shell prompt inside an existing container this way, or run an arbitrary process for automation.

Detaching from a container (Linux/macOS)

- If you have started an *interactive* container (with option `-it`), you can detach from it.
- The “detach” sequence is `^P^Q`.
- Otherwise you can detach by killing the Docker client.
(But not by hitting `^C`, as this would deliver `SIGINT` to the container.)

What does `-it` stand for?

- `-t` means “allocate a terminal.”
- `-i` means “connect stdin to the terminal.”

Attaching to a container

You can attach to a container:

```
$ docker attach <containerID>
```

- The container must be running.
- There *can* be multiple clients attached to the same container.
- If you don't specify `--detach-keys` when attaching, it defaults back to `^P^Q`.

Try it on our previous container:

```
$ docker attach $(docker ps -lq)
```

Check that `^x x` doesn't work, but `^P ^Q` does.

Restarting a container

When a container has exited, it is in stopped state.

It can then be restarted with the `start` command.

```
$ docker start <yourContainerID>
```

The container will be restarted using the same options you launched it with.

You can re-attach to it if you want to interact with it:

```
$ docker attach <yourContainerID>
```

Use `docker ps -a` to identify the container ID of a previous `jpetazzo/clock` container, and try those commands.

docker exec example

```
$ # You can run ruby commands in the area the  
app is running and more!  
$ docker exec -it <yourContainerId> bash  
root@5ca27cf74c2e:/opt/namer# irb  
irb(main):001:0> [0, 1, 2, 3, 4].map { |x| x **  
2}.compact  
=> [0, 1, 4, 9, 16]  
irb(main):002:0> exit
```

Stopping the container

Now that we're done let's stop our container.

```
$ docker stop <yourContainerID>
```

And remove it.

```
$ docker rm <yourContainerID>
```

A non-interactive container

We will run a small custom container.

This container just displays the time every second.

```
$ docker run jpetazzo/clock  
Fri Feb 20 00:28:53 UTC 2015  
Fri Feb 20 00:28:54 UTC 2015  
Fri Feb 20 00:28:55 UTC 2015  
...
```

- This container will run forever.
- To stop it, press ^C.
- Docker has automatically downloaded the image `jpetazzo/clock`.
- This image is a user image, created by `jpetazzo`.
- We will hear more about user images (and other types of images) later.

Why is PID 1 special?

- PID 1 has some extra responsibilities:
 - it starts (directly or indirectly) every other process
 - when a process exits, its processes are “reparented” under PID 1
- When PID 1 exits, everything stops:
 - on a “regular” machine, it causes a kernel panic
 - in a container, it kills all the processes
- We don't want PID 1 to stop accidentally
- That's why it has these extra protections

Run a container in the background

Containers can be started in the background, with the `-d` flag (daemon mode):

```
$ docker run -d jpetazzo/clock  
47d677dcfba4277c6cc68fcaa51f932b544cab1a187c853  
b7d0caf4e8debe5ad
```

- We don't see the output of the container.
- But don't worry: Docker collects that output and logs it!
- Docker gives us the ID of the container.

List running containers

How can we check that our container is still running?

With `docker ps`, just like the UNIX `ps` command, lists running processes.

```
$ docker ps
CONTAINER ID        IMAGE               ...      CREATED             STATUS
...                jpetazzo/clock   ...    2 minutes ago   Up  2
47d677dcfba4      jpetazzo/clock   ...    2 minutes ago   Up  2
minutes          ...
```

Docker tells us:

- The (truncated) ID of our container.
- The image used to start the container.
- That our container has been running (`Up`) for a couple of minutes.
- Other information (`COMMAND`, `PORTS`, `NAMES`) that we will explain later.

Starting more containers

Let's start two more containers.

```
$ docker run -d jpetazzo/clock  
57ad9bdxfc06bb4407c47220cf59ce21585dce9a1298d7a6  
7488359aeaea8ae2a
```

```
$ docker run -d jpetazzo/clock  
068cc994ffd0190bbe025ba74e4c0771a5d8f14734af772  
ddee8dc1aaaf20567d
```

Check that `docker ps` correctly reports all 3 containers.

Viewing only the last container started

When many containers are already running, it can be useful to see only the last container that was started.

This can be achieved with the `-1` ("Last") flag:

```
$ docker ps -1
CONTAINER ID        IMAGE               ...      CREATED
STATUS              ...
068cc994ffd0      jpetazzo/clock     ...      2 minutes
ago    Up 2 minutes  ...
```

View the logs of a container

We told you that Docker was logging the container output.
Let's see that now.

```
$ docker logs 068  
Fri Feb 20 00:39:52 UTC 2015  
Fri Feb 20 00:39:53 UTC 2015  
...
```

- We specified a *prefix* of the full container ID.
- You can, of course, specify the full ID.
- The `logs` command will output the *entire* logs of the container.
(Sometimes, that will be too much. Let's see how to address that.)

Follow the logs in real time

Just like with the standard UNIX command `tail -f`, we can follow the logs of our container:

```
$ docker logs --tail 1 --follow 068
Fri Feb 20 00:57:12 UTC 2015
Fri Feb 20 00:57:13 UTC 2015
^C
```

- This will display the last line in the log file.
- Then, it will continue to display the logs in real time.
- Use `^C` to exit.

Stop our container

There are two ways we can terminate our detached container.

- Killing it using the `docker kill` command.
- Stopping it using the `docker stop` command.

The first one stops the container immediately, by using the `KILL` signal.

The second one is more graceful. It sends a `TERM` signal, and after 10 seconds, if the container has not stopped, it sends `KILL`.

Reminder: the `KILL` signal cannot be intercepted, and will forcibly terminate the container.

List stopped containers

We can also see stopped containers, with the `-a` (`--all`) option.

```
$ docker ps -a
CONTAINER ID  IMAGE          ...  CREATED        STATUS
068cc994ffd0  jpetazzo/clock ...  21 min. ago   Exited
(137) 3 min. ago
57ad9bdxfc06b  jpetazzo/clock ...  21 min. ago   Exited
(137) 3 min. ago
47d677dcfba4  jpetazzo/clock ...  23 min. ago   Exited
(137) 3 min. ago
5c1dfd4d81f1  jpetazzo/clock ...  40 min. ago   Exited
(0) 40 min. ago
b13c164401fb  ubuntu         ...  55 min. ago   Exited
(130) 53 min. ago
```

Limiting RAM usage

```
some_str = '' * 512000000
```

Example:

```
docker run -ti --memory 100m python
```

If the container tries to use more than 100 MB of RAM, and swap is available:

- the container will not be killed,
- memory above 100 MB will be swapped out,
- in most cases, the app in the container will be slowed down (a lot).

If we run out of swap, the global OOM killer still intervenes.

Limiting both RAM and swap usage

Example:

```
docker run -ti --memory 100m --memory-swap 100m  
python
```

If the container tries to use more than 100 MB of memory, it is killed.

On the other hand, the application will never be slowed down because of swap.

Limiting CPU usage

- There are no less than 3 ways to limit CPU usage:
 - setting a relative priority with `--cpu-shares`,
 - setting a CPU% limit with `--cpus`,
 - pinning a container to specific CPUs with `--cpuset-cpus`.
- They can be used separately or together.

Setting relative priority

- Each container has a relative priority used by the Linux scheduler.
- By default, this priority is 1024.
- As long as CPU usage is not maxed out, this has no effect.
- When CPU usage is maxed out, each container receives CPU cycles in proportion of its relative priority.
- In other words: a container with `--cpu-shares 2048` will receive twice as much than the default.

Setting a CPU% limit

- This setting will make sure that a container doesn't use more than a given % of CPU.
- The value is expressed in CPUs; therefore:
 - cpus 0.1 means 10% of one CPU,
 - cpus 1.0 means 100% of one whole CPU,
 - cpus 10.0 means 10 entire CPUs.

Pinning containers to CPUs

- On multi-core machines, it is possible to restrict the execution on a set of CPUs.
- Examples:
 - cpuset-cpus 0 forces the container to run on CPU 0;
 - cpuset-cpus 3,5,7 restricts the container to CPUs 3, 5, 7;
 - cpuset-cpus 0-3,8-11 restricts the container to CPUs 0, 1, 2, 3, 8, 9, 10, 11.
- This will not reserve the corresponding CPUs!
(They might still be used by other containers, or uncontainerized processes.)

Limiting disk usage

- Most storage drivers do not support limiting the disk usage of containers.
(With the exception of devicemapper, but the limit cannot be set easily.)
- This means that a single container could exhaust disk space for everyone.
- In practice, however, this is not a concern, because:
 - data files (for stateful services) should reside on volumes,
 - assets (e.g. images, user-generated content...) should reside on object stores or on volume,
 - logs are written on standard output and gathered by the container engine.
- Container disk usage can be audited with `docker ps -s` and `docker diff`.

Default names

When we create a container, if we don't give a specific name, Docker will pick one for us.

It will be the concatenation of:

- A mood (furious, goofy, suspicious, boring...)
- The name of a famous inventor (tesla, darwin, wozniak...)

Examples: `happy_curie`, `clever_hopper`,
`jovial_lovelace` ...

Specifying a name

You can set the name of the container when you create it.

```
$ docker run --name ticktock jpetazzo/clock
```

If you specify a name that already exists, Docker will refuse to create the container.

This lets us enforce unicity of a given resource.

Renaming containers

- You can rename containers with `docker rename`.
- This allows you to “free up” a name without destroying the associated container.

Inspecting a container

The `docker inspect` command will output a very detailed JSON map.

```
$ docker inspect <containerID>
[ {
  ...
  (many pages of JSON here)
  ...
}
```

There are multiple ways to consume that information.

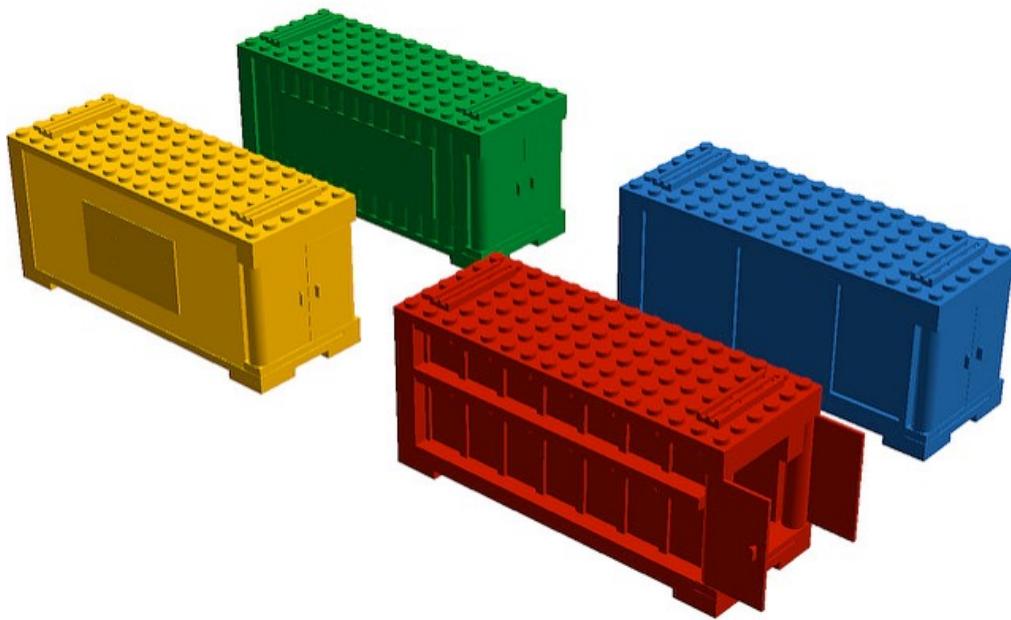
Parsing JSON with the Shell

- You could grep and cut or awk the output of `docker inspect`.
- Please, don't.
- It's painful.
- If you really must parse JSON from the Shell, use JQ! (It's great.)

```
$ docker inspect <containerID> | jq .
```

- We will see a better solution which doesn't require extra tools.

Building Docker images with a Dockerfile



Construction site with containers

Objectives

We will build a container image automatically, with a **Dockerfile**.

At the end of this lesson, you will be able to:

- Write a **Dockerfile**.
- Build an image from a **Dockerfile**.

Dockerfile overview

- A **Dockerfile** is a build recipe for a Docker image.
- It contains a series of instructions telling Docker how an image is constructed.
- The `docker build` command builds an image from a **Dockerfile**.

Writing our first Dockerfile

Our Dockerfile must be in a **new, empty directory**.

1. Create a directory to hold our Dockerfile.

```
$ mkdir myimage
```

2. Create a Dockerfile inside this directory.

```
$ cd myimage  
$ vim Dockerfile
```

Of course, you can use any other editor of your choice.

Type this into our Dockerfile...

```
FROM ubuntu
```

```
RUN apt-get update
```

```
RUN apt-get install figlet
```

- FROM indicates the base image for our build.
- Each RUN line will be executed by Docker during the build.
- Our RUN commands **must be non-interactive**. (No input can be provided to Docker during the build.)
- In many cases, we will add the -y flag to apt-get.

Build it!

Save our file, then execute:

```
$ docker build -t figlet .
```

- `-t` indicates the tag to apply to the image.
- `.` indicates the location of the *build context*.

We will talk more about the build context later.

To keep things simple for now: this is the directory where our Dockerfile is located.

Sending the build context to Docker

Sending build context to Docker daemon 2.048 kB

- The build context is the . directory given to `docker build`.
- It is sent (as an archive) by the Docker client to the Docker daemon.
- This allows to use a remote machine to build using local files.
- Be careful (or patient) if that directory is big and your link is slow.
- You can speed up the process with a [.dockerignore](#) file
 - It tells docker to ignore specific files in the directory
 - Only ignore files that you won't need in the build context!

Executing each step

Step 2/3 : RUN apt-get update

---> Running in e01b294dbffd

(...output of the RUN command...)

Removing intermediate container e01b294dbffd

---> eb8d9b561b37

- A container (`e01b294dbffd`) is created from the base image.
- The `RUN` command is executed in this container.
- The container is committed into an image (`eb8d9b561b37`).
- The build container (`e01b294dbffd`) is removed.
- The output of this step will be the base image for the next one.

The caching system

If you run the same build again, it will be instantaneous. Why?

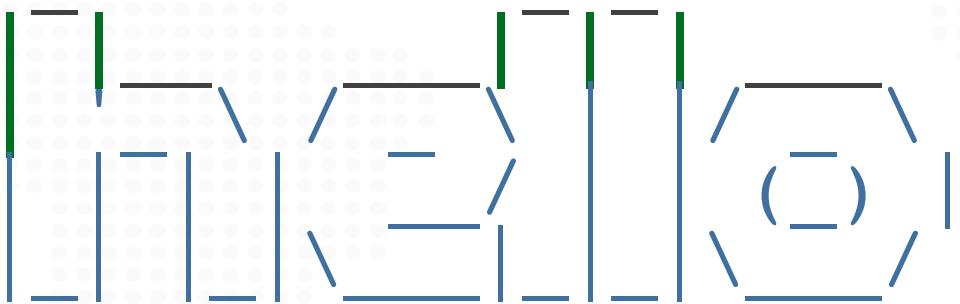
- After each build step, Docker takes a snapshot of the resulting image.
- Before executing a step, Docker checks if it has already built the same sequence.
- Docker uses the exact strings defined in your Dockerfile, so:
 - `RUN apt-get install figlet cowsay` is different from `RUN apt-get install cowsay figlet`
 - `RUN apt-get update` is not re-executed when the mirrors are updated

You can force a rebuild with `docker build --no-cache . . .`

Running the image

The resulting image is not different from the one produced manually.

```
$ docker run -ti figlet  
root@91f3c974c9a1:/# figlet hello
```



Yay! 

Using image and viewing history

The `history` command lists all the layers composing an image.

For each layer, it shows its creation time, size, and creation command.

When an image was built with a Dockerfile, each layer corresponds to a line of the Dockerfile.

\$ docker history figlet	IMAGE	CREATED	CREATED BY	SIZE
	f9e8f1642759	About an hour ago	/bin/sh -c apt-get install figlet	1.627 MB
	7257c37726a1	About an hour ago	/bin/sh -c apt-get update	21.58 MB
	07c86167cdc4	4 days ago	/bin/sh -c #(nop) CMD ["/bin/sh"]	0 B
	<missing>	4 days ago	/bin/sh -c sed -i 's/^#\s*/(/	1.895 kB
	<missing>	4 days ago	/bin/sh -c echo '#!/bin/sh'	194.5 kB
	<missing>	4 days ago	/bin/sh -c #(nop) ADD file:b	187.8 MB

Shell syntax vs exec syntax

Dockerfile commands that execute something can have two forms:

- plain string, or *shell syntax*: RUN apt-get install figlet
- JSON list, or *exec syntax*: RUN ["apt-get", "install", "figlet"]

We are going to change our Dockerfile to see how it affects the resulting image.

Using exec syntax in our Dockerfile

Let's change our Dockerfile as follows!

```
FROM ubuntu
RUN apt-get update
RUN [ "apt-get", "install", "figlet" ]
```

Then build the new Dockerfile.

```
$ docker build -t figlet .
```

History with exec syntax

Compare the new history:

```
$ docker history figlet
IMAGE           CREATED
27954bb5faaf  10 seconds ago
7257c37726a1  About an hour ago
07c86167cdc4  4 days ago
<missing>      4 days ago
<missing>      4 days ago
<missing>      4 days ago
```

CREATED BY	SIZE
apt-get install figlet	1.627 MB
/bin/sh -c apt-get update	21.58 MB
/bin/sh -c #(nop) CMD ["/bin	0 B
/bin/sh -c sed -i 's/^#\s*/(1.895 kB
/bin/sh -c echo '#!/bin/sh'	194.5 kB
/bin/sh -c #(nop) ADD file:b	187.8 MB

- Exec syntax specifies an *exact* command to execute.
- Shell syntax specifies a command to be wrapped within /bin/sh -c "...".

When to use exec syntax and shell syntax

- shell syntax:
 - is easier to write
 - interpolates environment variables and other shell expressions
 - creates an extra process (`/bin/sh -c ...`) to parse the string
 - requires `/bin/sh` to exist in the container
- exec syntax:
 - is harder to write (and read!)
 - passes all arguments without extra processing
 - doesn't create an extra process
 - doesn't require `/bin/sh` to exist in the container

Pro-tip: the `exec` shell built-in

POSIX shells have a built-in command named `exec`.
`exec` should be followed by a program and its arguments.

From a user perspective:

- it looks like the shell exits right away after the command execution,
- in fact, the shell exits just *before* command execution;
- or rather, the shell gets *replaced* by the command.

Example using exec

CMD exec figlet -f script hello

In this example, sh -c will still be used, but figlet will be PID 1 in the container.

The shell gets replaced by figlet when figlet starts execution.

This allows to run processes as PID 1 without using JSON.

Adding **CMD** to our Dockerfile

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
CMD figlet -f script hello
```

- **CMD** defines a default command to run when none is given.
- It can appear at any point in the file.
- Each **CMD** will replace and override the previous one.
- As a result, while you can have multiple **CMD** lines, it is useless.

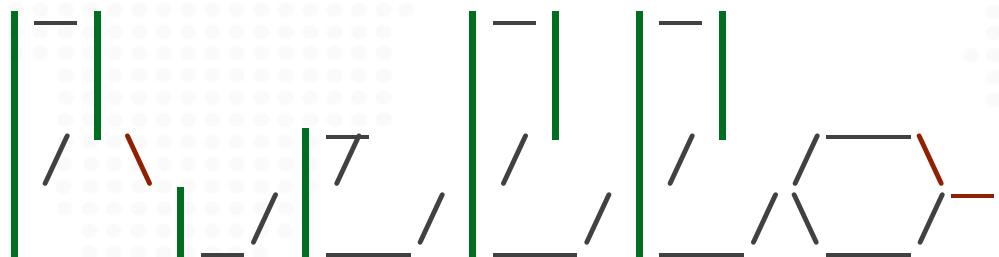
Build and test our image

Let's build it:

```
$ docker build -t figlet .  
.  
Successfully built 042dff3b4a8d  
Successfully tagged figlet:latest
```

And run it:

```
$ docker run figlet
```



Overriding CMD

If we want to get a shell into our container (instead of running `figlet`), we just have to specify a different program to run:

```
$ docker run -it figlet bash  
root@7ac86a641116:/#
```

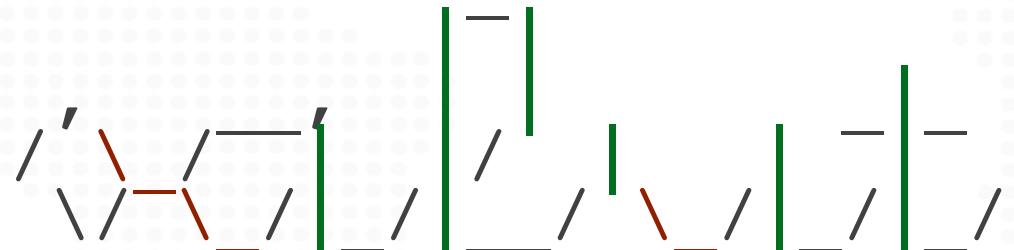
- We specified `bash`.
- It replaced the value of `CMD`.

Using **ENTRYPOINT**

We want to be able to specify a different message on the command line, while retaining `figlet` and some default parameters.

In other words, we would like to be able to do this:

```
$ docker run figlet salut
```



We will use the `ENTRYPOINT` verb in Dockerfile.

Adding **ENTRYPOINT** to our Dockerfile

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
ENTRYPOINT ["figlet", "-f", "script"]
```

- **ENTRYPOINT** defines a base command (and its parameters) for the container.
- The command line arguments are appended to those parameters.
- Like **CMD**, **ENTRYPOINT** can appear anywhere, and replaces the previous value.

Why did we use JSON syntax for our **ENTRYPOINT**?

Implications of JSON vs string syntax

- When CMD or ENTRYPOINT use string syntax, they get wrapped in `sh -c`.
- To avoid this wrapping, we can use JSON syntax.

What if we used ENTRYPOINT with string syntax?

```
$ docker run figlet salut
```

This would run the following command in the `figlet` image:

```
sh -c "figlet -f script" salut
```

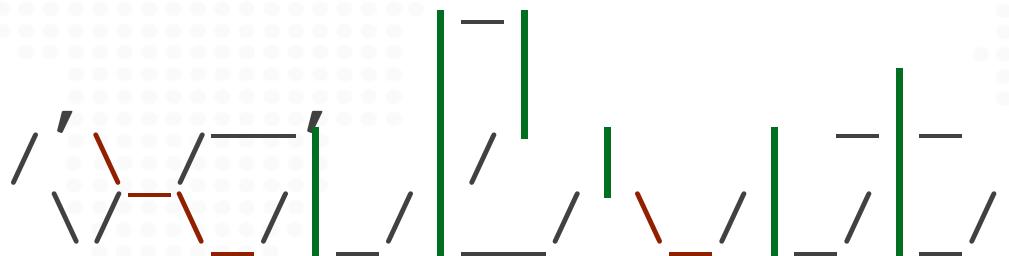
Build and test our image

Let's build it:

```
$ docker build -t figlet .
.
.
Successfully built 36f588918d73
Successfully tagged figlet:latest
```

And run it:

```
$ docker run figlet salut
```



Using **CMD** and **ENTRYPOINT** together

What if we want to define a default message for our container?

Then we will use **ENTRYPOINT** and **CMD** together.

- **ENTRYPOINT** will define the base command for our container.
- **CMD** will define the default parameter(s) for this command.
- They *both* have to use JSON syntax.

CMD and ENTRYPOINT together

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
ENTRYPOINT ["figlet", "-f", "script"]
CMD ["hello world"]
```

- ENTRYPOINT defines a base command (and its parameters) for the container.
- If we don't specify extra command-line arguments when starting the container, the value of CMD is appended.
- Otherwise, our extra command-line arguments are used instead of CMD.

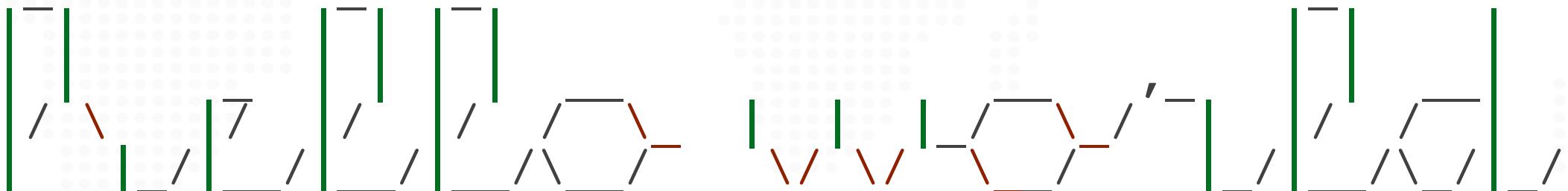
Build and test our image

Let's build it:

```
$ docker build -t myfiglet .  
...  
Successfully built 6e0b6a048a07  
Successfully tagged myfiglet:latest
```

Run it without parameters:

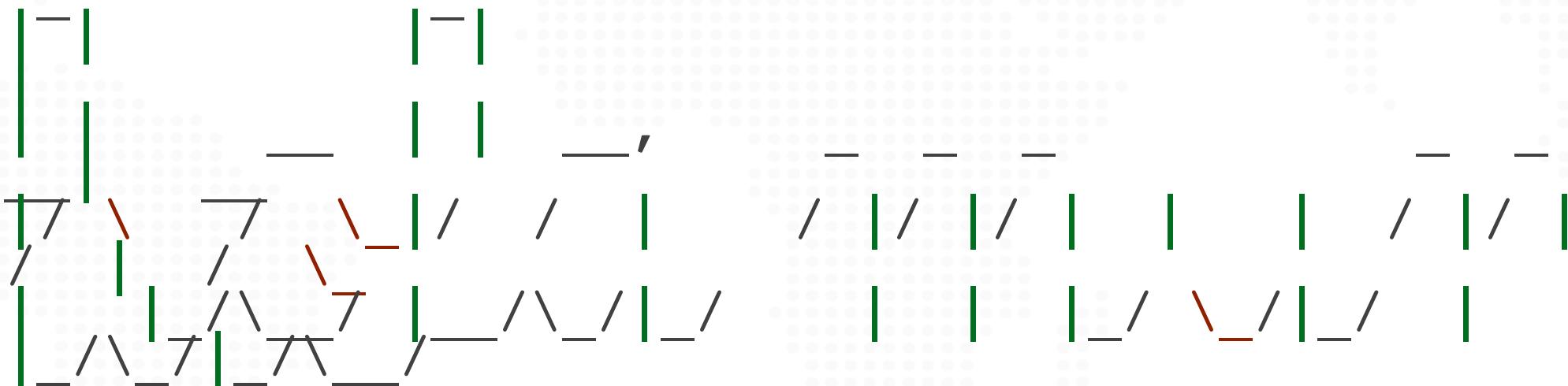
```
$ docker run myfiglet
```



Overriding the image default parameters

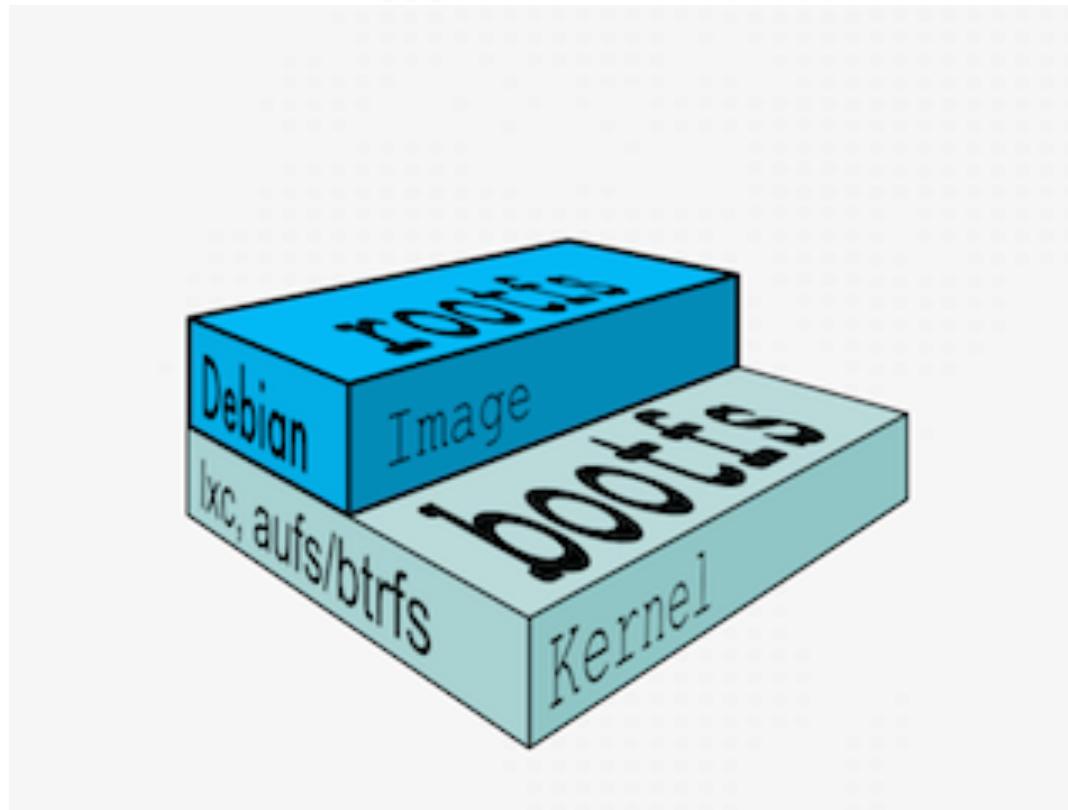
Now let's pass extra arguments to the image.

```
$ docker run myfiglet hola mundo
```



We overrode `CMD` but still used `ENTRYPOINT`.

Understanding Docker images



image

Objectives

In this section, we will explain:

- What is an image.
- What is a layer.
- The various image namespaces.
- How to search and download images.
- Image tags and when to use them.

What is an image?

- Image = files + metadata
- These files form the root filesystem of our container.
- The metadata can indicate a number of things, e.g.:
 - the author of the image
 - the command to execute in the container when starting it
 - environment variables to be set
 - etc.
- Images are made of *layers*, conceptually stacked on top of each other.
- Each layer can add, change, and remove files and/or metadata.
- Images can share layers to optimize disk usage, transfer times, and memory use.

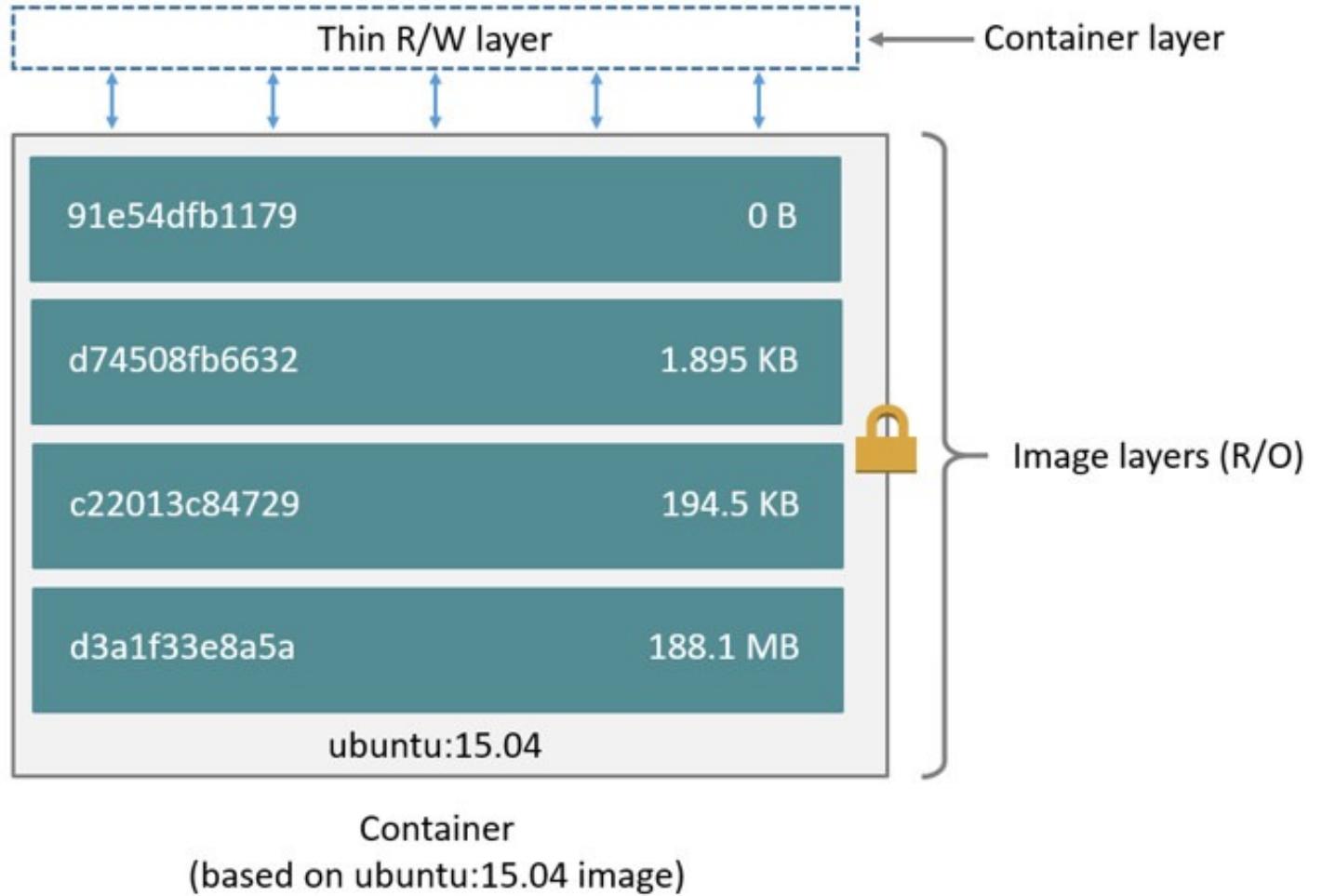
Example for a Java webapp

Each of the following items will correspond to one layer:

- CentOS base layer
- Packages and configuration files added by our local IT
- JRE
- Tomcat
- Our application's dependencies
- Our application code and assets
- Our application configuration

(Note: app config is generally added by orchestration facilities.)

The read-write layer

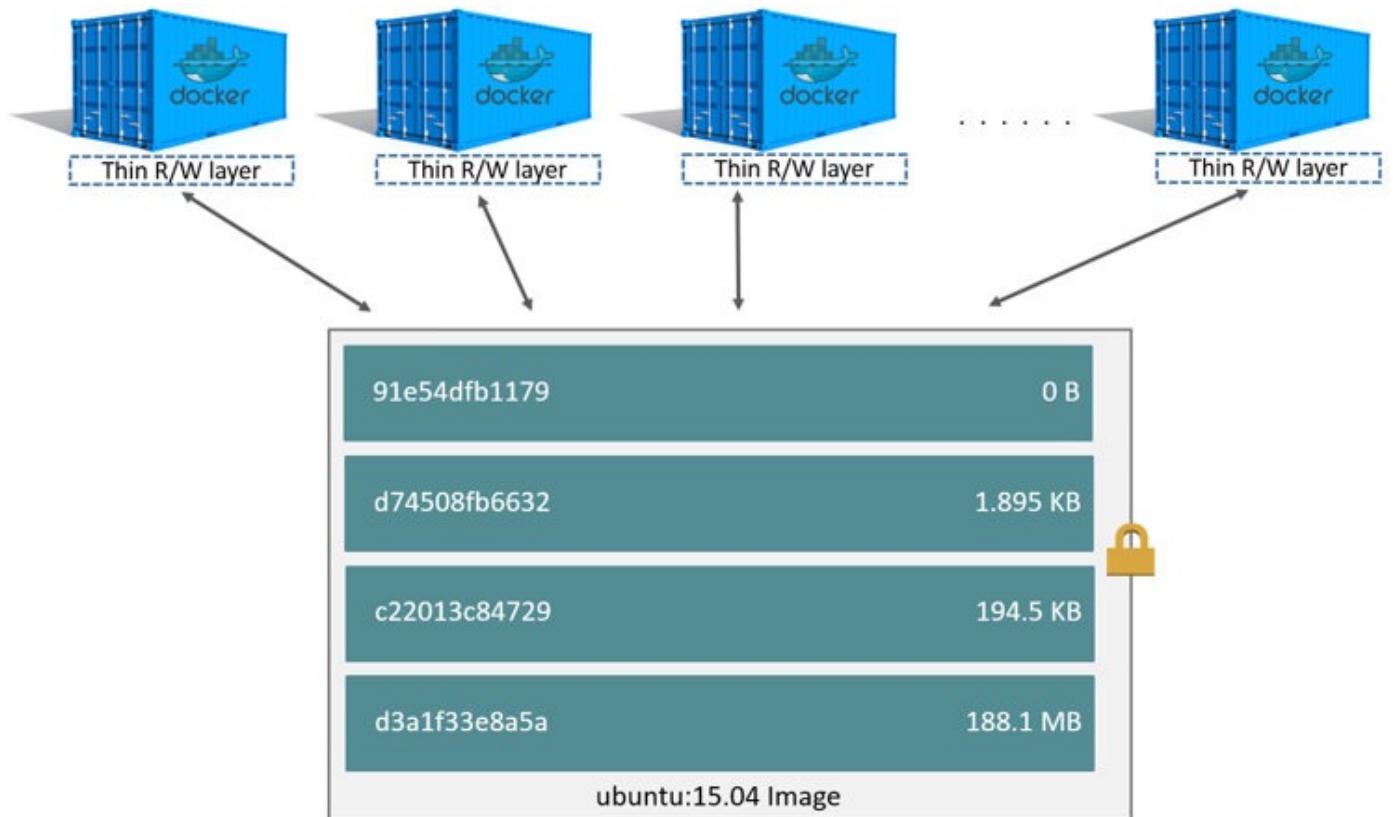


layers

Differences between containers and images

- An image is a read-only filesystem.
- A container is an encapsulated set of processes, running in a read-write copy of that filesystem.
- To optimize container boot time, *copy-on-write* is used instead of regular copy.
- `docker run` starts a container from a given image.

Multiple containers sharing the same image



layers

Wait a minute...

If an image is read-only, how do we change it?

- We don't.
- We create a new container from that image.
- Then we make changes to that container.
- When we are satisfied with those changes, we transform them into a new layer.
- A new image is created by stacking the new layer on top of the old image.

A chicken-and-egg problem

- The only way to create an image is by “freezing” a container.
- The only way to create a container is by instantiating an image.
- Help!

Images namespaces

There are three namespaces:

- Official images
 - e.g. `ubuntu`, `busybox` ...
- User (and organizations) images
 - e.g. `jpetazzo/clock`
- Self-hosted images
 - e.g. `registry.example.com:5000/my-private/image`

Let's explain each of them.

Root namespace

The root namespace is for official images.

They are gated by Docker Inc.

They are generally authored and maintained by third parties.

Those images include:

- Small, “swiss-army-knife” images like busybox.
- Distro images to be used as bases for your builds, like ubuntu, fedora...
- Ready-to-use components and services, like redis, postgresql...
- Over 150 at this point!

User namespace

The user namespace holds images for Docker Hub users and organizations.

For example:

`jpetazzo/clock`

The Docker Hub user is:

`jpetazzo`

The image name is:

`clock`

Self-hosted namespace

This namespace holds images which are not hosted on Docker Hub, but on third party registries.

They contain the hostname (or IP address), and optionally the port, of the registry server.

For example:

`localhost:5000/wordpress`

- `localhost:5000` is the host and port of the registry
- `wordpress` is the name of the image

Other examples:

`quay.io/coreos/etcd`

`gcr.io/google-containers/hugo`

How do you store and manage images?

Images can be stored:

- On your Docker host.
- In a Docker registry.

You can use the Docker client to download (pull) or upload (push) images.

To be more accurate: you can use the Docker client to tell a Docker Engine to push and pull images to and from a registry.

Showing current images

Let's look at what images are on our host now.

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
fedora	latest	ddd5c9c1d0f2	3 days ago	204.7 MB
centos	latest	d0e7f81ca65c	3 days ago	196.6 MB
ubuntu	latest	07c86167cdc4	4 days ago	188 MB
redis	latest	4f5f397d4b7c	5 days ago	177.6 MB
postgres	latest	afe2b5e1859b	5 days ago	264.5 MB
alpine	latest	70c557e50ed6	5 days ago	4.798 MB
debian	latest	f50f9524513f	6 days ago	125.1 MB
busybox	latest	3240943c9ea3	2 weeks ago	1.114 MB
training/namer	latest	902673acc741	9 months ago	289.3 MB
jpetazzo/clock	latest	12068b93616f	12 months ago	2.433 MB

Searching for images

We cannot list *all* images on a remote registry, but we can search for a specific keyword:

\$ docker search marathon	NAME	DESCRIPTION	STARS	OFFICIAL
	AUTOMATED			
	mesosphere/marathon	A cluster-wide init and co...	105	
	[OK]			
	mesoscloud/marathon	Marathon	31	
	[OK]			
	mesosphere/marathon-lb	Script to update haproxy b...	22	
	[OK]			
	tobilg/mongodb-marathon	A Docker image to start a ...	4	
	[OK]			

- “Stars” indicate the popularity of the image.
- “Official” images are those in the root namespace.
- “Automated” images are built automatically by the Docker Hub. (This means that their build recipe is always available.)

Downloading images

There are two ways to download images.

- Explicitly, with `docker pull`.
- Implicitly, when executing `docker run` and the image is not found locally.

Pulling an image

```
$ docker pull debian:jessie
Pulling repository debian
b164861940b8: Download complete
b164861940b8: Pulling image (jessie) from debian
d1881793a057: Download complete
```

- As seen previously, images are made up of layers.
- Docker has downloaded all the necessary layers.
- In this example, :jessie indicates which exact version of Debian we would like.

It is a *version tag*.

Image and tags

- Images can have tags.
- Tags define image versions or variants.
- `docker pull ubuntu` will refer to `ubuntu:latest`.
- The `:latest` tag is generally updated often.

When to (not) use tags

Don't specify tags:

- When doing rapid testing and prototyping.
- When experimenting.
- When you want the latest version.

Do specify tags:

- When recording a procedure into a script.
- When going to production.
- To ensure that the same version will be used everywhere.
- To ensure repeatability later.

This is similar to what we would do with `pip install`, `npm install`, etc.

Publishing images to the Docker Hub

We have built our first images.

We can now publish it to the Docker Hub!

You don't have to do the exercises in this section, because they require an account on the Docker Hub, and we don't want to force anyone to create one.

Note, however, that creating an account on the Docker Hub is free (and doesn't require a credit card), and hosting public images is free as well.

Logging into our Docker Hub account

- This can be done from the Docker CLI:

```
docker login
```

.warning[When running Docker for Mac/Windows, or Docker on a Linux workstation, it can (and will when possible) integrate with your system's keyring to store your credentials securely. However, on most Linux servers, it will store your credentials in `~/.docker/config`.]

<https://docs.docker.com/docker-hub/>



Repositories > Create

Using 0 of 1 private repositories. [Get more](#)

Create Repository

mobythewhale

my-private-repo

Description

Visibility

Using 0 of 1 private repositories. [Get more](#) Public

Public repositories appear in Docker Hub search results

 Private

Only you can view private repositories

Build Settings (optional)

Autobuild triggers a new build with every git push to your source code repository. [Learn More](#).

Please re-link a GitHub or Bitbucket account

We've updated how Docker Hub connects to GitHub and Bitbucket. You'll need to re-link a GitHub or Bitbucket account to create new automated builds. [Learn More](#)

Disconnected



Disconnected

[Cancel](#)[Create](#)[Create & Build](#)<https://docs.docker.com/docker-hub/>

Image tags and registry addresses

- Docker images tags are like Git tags and branches.
- They are like *bookmarks* pointing at a specific image ID.
- Tagging an image doesn't *rename* an image: it adds another tag.
- When pushing an image to a registry, the registry address is in the tag.
Example: `registry.example.net:5000/image`
- What about Docker Hub images?
–
 - `jpetazzo/clock` is, in fact, `index.docker.io/jpetazzo/clock`
 - `ubuntu` is, in fact, `library/ubuntu`,
i.e. `index.docker.io/library/ubuntu`

Tagging an image to push it on the Hub

- Let's tag our `figlet` image (or any other to our liking):

```
docker tag figlet jpetazzo/figlet
```

- And push it to the Hub:

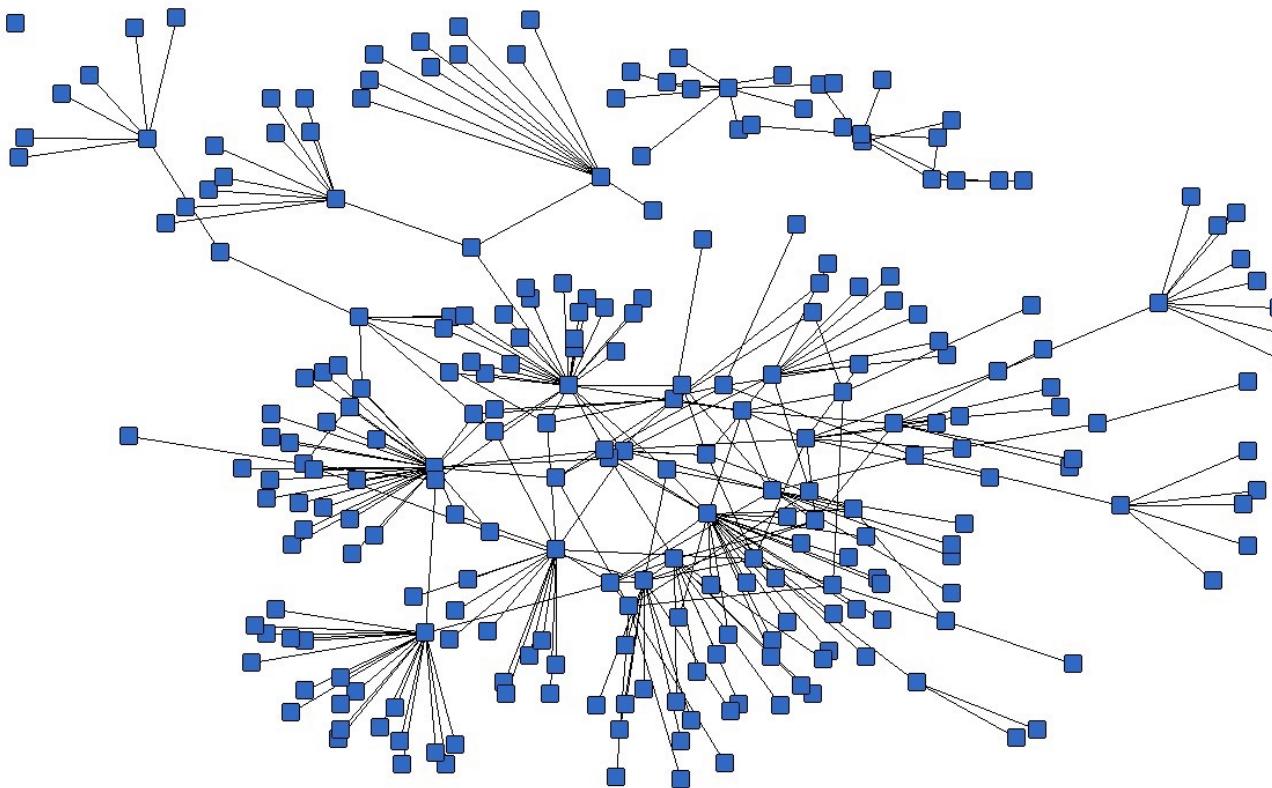
```
docker push jpetazzo/figlet
```

- That's it!

—

- Anybody can now `docker run jpetazzo/figlet` anywhere.

Container networking basics



A dense graph network

Objectives

We will now run network services (accepting requests) in containers.

At the end of this section, you will be able to:

- Run a network service in a container.
- Connect to that network service.
- Find a container's IP address.

Running a very simple service

- We need something small, simple, easy to configure
(or, even better, that doesn't require any configuration at all)
- Let's use the official NGINX image (named `nginx`)
- It runs a static web server listening on port 80
- It serves a default "Welcome to nginx!" page

Runing an NGINX server

```
$ docker run -d -P nginx  
66b1ce719198711292c8f34f84a7b68c3876cf9f67015e7  
52b94e189d35a204e
```

- Docker will automatically pull the `nginx` image from the Docker Hub
- `-d / --detach` tells Docker to run it in the background
- `P / --publish-all` tells Docker to publish all ports
(publish = make them reachable from other computers)
- ...OK, how do we connect to our web server now?

Finding our web server port

- First, we need to find the *port number* used by Docker
(the NGINX container listens on port 80, but this port will be *mapped*)

- We can use `docker ps`:

```
$ docker ps
CONTAINER ID  IMAGE      ...      PORTS          ...
e40ffb406c9e  nginx      ...      0.0.0.0:12345->80/tcp  ...
```

- This means:
port 12345 on the Docker host is mapped to port 80 in the container
- Now we need to connect to the Docker host!

Finding the address of the Docker host

- When running Docker on your Linux workstation:
use localhost, or any IP address of your machine
- When running Docker on a remote Linux server:
use any IP address of the remote machine
- When running Docker Desktop on Mac or Windows:
use localhost
- In other scenarios (`docker-machine`, local VM...):
use the IP address of the Docker VM

Connecting to our web server (GUI)

Point your browser to the IP address of your Docker host, on the port shown by `docker ps` for container port 80.



Screenshot

Connecting to our web server (CLI)

You can also use `curl` directly from the Docker host.
Make sure to use the right port number if it is different from the example below:

```
$ curl localhost:12345
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...

```

How does Docker know which port to map?

- There is metadata in the image telling “this image has something on port 80”.
- We can see that metadata with `docker inspect`:

```
$ docker inspect --format '{{.Config.ExposedPorts}}' nginx
map[80/tcp:{ }]
```

- This metadata was set in the Dockerfile, with the `EXPOSE` keyword.
- We can see that with `docker history`:

```
$ docker history nginx
IMAGE                  CREATED          CREATED BY
7f70b30f2cc6          11 days ago    /bin/sh -c #(nop)
CMD ["nginx" "-g" "..."]           /bin/sh -c #(nop)
<missing>              11 days ago    /bin/sh -c #(nop)
STOP SIGNAL [SIGTERM]           11 days ago    /bin/sh -c #(nop)
<missing>              11 days ago    /bin/sh -c #(nop)
EXPOSE 80/tcp
```

Why can't we just connect to port 80?

- Our Docker host has only one port 80
- Therefore, we can only have one container at a time on port 80
- Therefore, if multiple containers want port 80, only one can get it
- By default, containers *do not* get “their” port number, but a random one
 - (not “random” as “crypto random”, but as “it depends on various factors”)
- We’ll see later how to force a port number (including port 80!)

Using multiple IP addresses

Hey, my network-fu is strong, and I have questions...

- Can I publish one container on 127.0.0.2:80, and another on 127.0.0.3:80?
- My machine has multiple (public) IP addresses, let's say A.A.A.A and B.B.B.B. Can I have one container on A.A.A.A:80 and another on B.B.B.B:80?
- I have a whole IPV4 subnet, can I allocate it to my containers?
- What about IPV6?

You can do all these things when running Docker directly on Linux.
(On other platforms, *generally not*, but there are some exceptions.)

Finding the web server port in a script

Parsing the output of `docker ps` would be painful.

There is a command to help us:

```
$ docker port <containerID> 80  
0.0.0.0:12345
```

Manual allocation of port numbers

If you want to set port numbers yourself, no problem:

```
$ docker run -d -p 80:80 nginx
$ docker run -d -p 8000:80 nginx
$ docker run -d -p 8080:80 -p 8888:80 nginx
```

- We are running three NGINX web servers.
- The first one is exposed on port 80.
- The second one is exposed on port 8000.
- The third one is exposed on ports 8080 and 8888.

Note: the convention is **port-on-host:port-on-container**.

Plumbing containers into your infrastructure

There are many ways to integrate containers in your network.

- Start the container, letting Docker allocate a public port for it. Then retrieve that port number and feed it to your configuration.
- Pick a fixed port number in advance, when you generate your configuration. Then start your container by setting the port numbers manually.
- Use an orchestrator like Kubernetes or Swarm. The orchestrator will provide its own networking facilities.

Orchestrators typically provide mechanisms to enable direct container-to-container communication across hosts, and publishing/load balancing for inbound traffic.

Finding the container's IP address

We can use the `docker inspect` command to find the IP address of the container.

```
$ docker inspect --format '{{  
    .NetworkSettings.IPAddress  
}}' <yourContainerID>  
172.17.0.3
```

- `docker inspect` is an advanced command, that can retrieve a ton of information about our containers.
- Here, we provide it with a format string to extract exactly the private IP address of the container.

Pinging our container

Let's try to ping our container *from another container*.

```
docker run alpine ping <ipaddress>
PING 172.17.0.x (172.17.0.x): 56 data bytes
64 bytes from 172.17.0.x: seq=0 ttl=64 time=0.106
ms
64 bytes from 172.17.0.x: seq=1 ttl=64 time=0.250
ms
64 bytes from 172.17.0.x: seq=2 ttl=64 time=0.188
ms
```

When running on Linux, we can even ping that IP address directly!
(And connect to a container's ports even if they aren't published.)

How often do we use -p and -P ?

- When running a stack of containers, we will often use Compose
- Compose will take care of exposing containers (through a `ports:` section in the `docker-compose.yml` file)
- It is, however, fairly common to use `docker run -P` for a quick test
- Or `docker run -p ...` when an image doesn't `EXPOSE` a port correctly

Section summary

We've learned how to:

- Expose a network port.
- Connect to an application running in a container.
- Find a container's IP address.

The Container Network Model

Docker has “networks”.

We can manage them with the `docker network` commands; for instance:

\$ docker network ls	NETWORK ID	NAME	DRIVER
	6bde79dfcf70	bridge	bridge
	8d9c78725538	none	null
	eb0eeab782f4	host	host
	4c1ff84d6d3f	blog-dev	overlay
	228a4355d548	blog-prod	overlay

New networks can be created (with `docker network create`).

(Note: networks `none` and `host` are special; let’s set them aside for now.)

What's a network?

- Conceptually, a Docker “network” is a virtual switch
(we can also think about it like a VLAN, or a WiFi SSID, for instance)
- By default, containers are connected to a single network
(but they can be connected to zero, or many networks, even dynamically)
- Each network has its own subnet (IP address range)
- A network can be local (to a single Docker Engine) or global (span multiple hosts)
- Containers can have *network aliases* providing DNS-based service discovery
(and each network has its own “domain”, “zone”, or “scope”)

Service discovery

- A container can be given a network alias
 - (e.g. with `docker run --net some-network --net-alias db ...`)
- The containers running in the same network can resolve that network alias
 - (i.e. if they do a DNS lookup on `db`, it will give the container's address)
- We can have a different `db` container in each network
 - (this avoids naming conflicts between different stacks)
- When we name a container, it automatically adds the name as a network alias
 - (i.e. `docker run --name xyz ...` is like `docker run --net-alias xyz ...`)

Network isolation

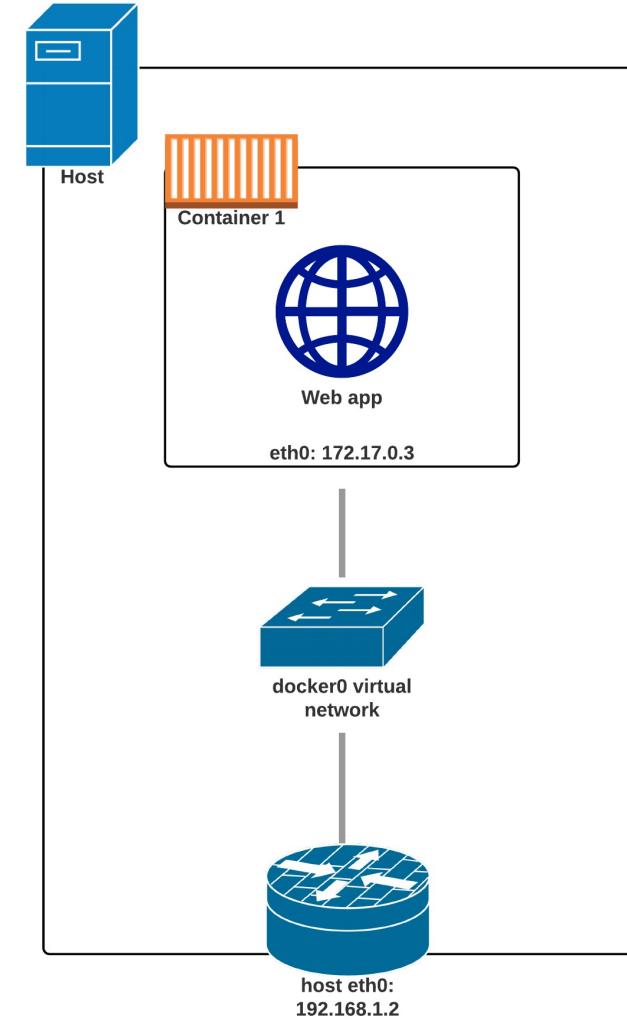
- Networks are isolated
- By default, containers in network A cannot reach those in network B
- A container connected to both networks A and B can act as a router or proxy
- Published ports are always reachable through the Docker host address

(`docker run -P ...` makes a container port available to everyone)

How to use networks

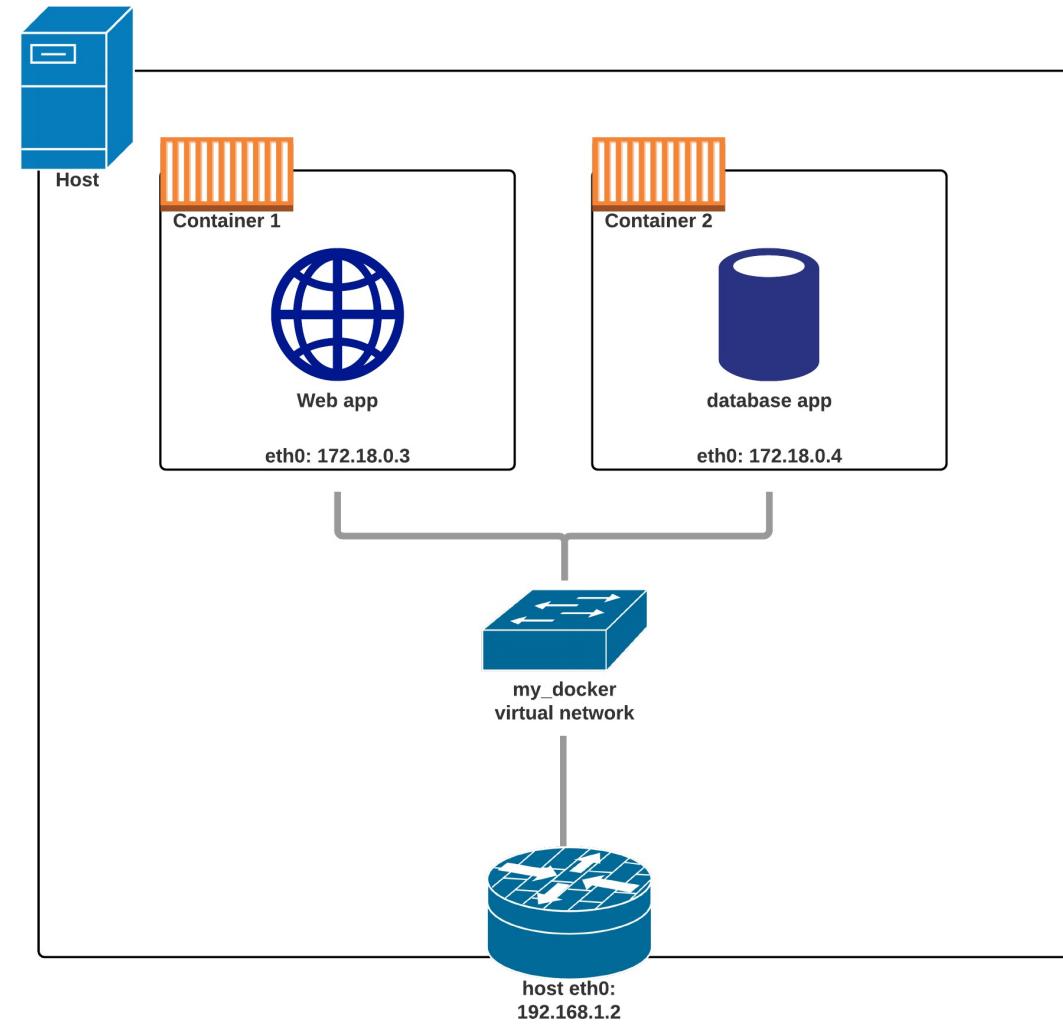
- We typically create one network per “stack” or app that we deploy
- More complex apps or stacks might require multiple networks (e.g. `frontend`, `backend`, ...)
- Networks allow us to deploy multiple copies of the same stack (e.g. `prod`, `dev`, `pr-442`,)
- If we use Docker Compose, this is managed automatically for us

Single container in a Docker network



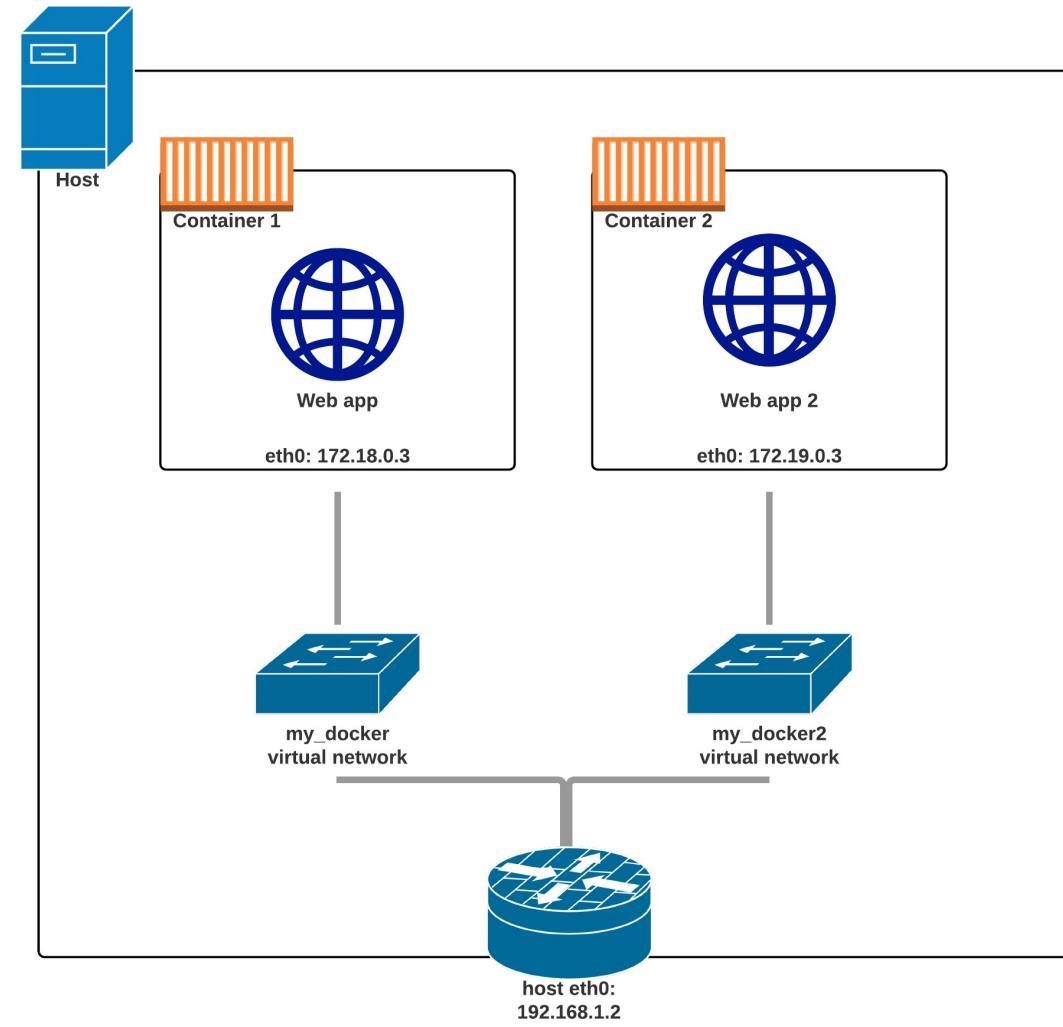
bridge0

Two containers on a single Docker network



bridge2

Two containers on two Docker networks



bridge3

CNM vs CNI

- CNM is the model used by Docker
- Kubernetes uses a different model, architected around CNI
(CNI is a kind of API between a container engine and *CNI plugins*)
- Docker model:
 - multiple isolated networks
 - per-network service discovery
 - network interconnection requires extra steps
- Kubernetes model:
 - single flat network
 - per-namespace service discovery
 - network isolation requires extra steps (Network Policies)

Creating a network

Let's create a network called dev.

```
$ docker network create dev  
4c1ff84d6d3f1733d3e233ee039cac276f425a9d5228a4355d  
54878293a889ba
```

The network is now visible with the `network ls` command:

NETWORK ID	NAME	DRIVER
6bde79dfcf70	bridge	bridge
8d9c78725538	none	null
eb0eeab782f4	host	host
4c1ff84d6d3f	dev	bridge

Placing containers on a network

We will create a *named* container on this network.
It will be reachable with its name, es.

```
$ docker run -d --name es --net dev  
elasticsearch:2  
8abb80e229ce8926c7223beb69699f5f34d6f1d438bfc56  
82db893e798046863
```

Communication between containers

Now, create another container on this network.

- \$ docker run -ti --net dev alpine sh
- root@0ecccdfa45ef:/#

From this new container, we can resolve and ping the other one, using its assigned name:

- / # ping es
- PING es (172.18.0.2) 56(84) bytes of data.
- 64 bytes from es.dev (172.18.0.2): icmp_seq=1 ttl=64 time=0.221 ms
- 64 bytes from es.dev (172.18.0.2): icmp_seq=2 ttl=64 time=0.114 ms
- 64 bytes from es.dev (172.18.0.2): icmp_seq=3 ttl=64 time=0.114 ms
- ^C
- --- es ping statistics ---
- 3 packets transmitted, 3 received, 0% packet loss, time 2000ms
- rtt min/avg/max/mdev = 0.114/0.149/0.221/0.052 ms
- root@0ecccdfa45ef:/#

Service discovery with containers

- Let's try to run an application that requires two containers.
- The first container is a web server.
- The other one is a redis data store.
- We will place them both on the `dev` network created before.

Running the web server

- The application is provided by the container image `jpetazzo/trainingwheels`.
- We don't know much about it so we will try to run it and see what happens!

Start the container, exposing all its ports:

```
$ docker run --net dev -d -P  
jpetazzo/trainingwheels
```

Check the port that has been allocated to it:

```
$ docker ps -l
```

Test the web server

If we connect to the application now, we will see an error page:



Trainingwheels error

- This is because the Redis service is not running.
 - This container tries to resolve the name `redis`.
- Note: we're not using a FQDN or an IP address here; just `redis`.

Start the data store

- We need to start a Redis container.
- That container must be on the same network as the web server.
- It must have the right network alias (`redis`) so the application can find it.

Start the container:

```
$ docker run --net dev --net-alias redis -d  
redis
```

Test the web server again

If we connect to the application now, we should see that the app is working correctly:

Training wheels

This request was served by f927b966d8e5.
f927b966d8e5 served 1 request so far.

The current ladder is:

- f927b966d8e5 → 1 request

Trainingwheels OK

- When the app tries to resolve `redis`, instead of getting a DNS error, it gets the IP address of our Redis container.

A few words on scope

- Container names are unique (there can be only one `--name redis`)
- Network aliases are not unique
- We can have the same network alias in different networks:

```
docker run --net dev --net-alias redis ...
docker run --net prod --net-alias redis ...
```
- We can even have multiple containers with the same alias in the same network
(in that case, we get multiple DNS entries, aka “DNS round robin”)

Names are *local* to each network

Let's try to ping our `es` container from another container, when that other container is *not* on the `dev` network.

```
$ docker run --rm alpine ping es  
ping: bad address 'es'
```

Names can be resolved only when containers are on the same network.

Containers can contact each other only when they are on the same network (you can try to ping using the IP address to verify).

Network aliases

We would like to have another network, `prod`, with its own `es` container. But there can be only one container named `es`!

We will use *network aliases*.

A container can have multiple network aliases.

Network aliases are *local* to a given network (only exist in this network).

Multiple containers can have the same network alias (even on the same network).

Since Docker Engine 1.11, resolving a network alias yields the IP addresses of all containers holding this alias.

Creating containers on another network

Create the prod network.

```
$ docker network create prod  
5a41562fecf2d8f115bedc16865f7336232a04268bdf2bd816aecca01b68  
d50c
```

We can now create multiple containers with the `es` alias on the new `prod` network.

```
$ docker run -d --name prod-es-1 --net-alias es --net prod  
elasticsearch:2  
38079d21caf0c5533a391700d9e9e920724e89200083df73211081c8a356  
d771  
$ docker run -d --name prod-es-2 --net-alias es --net prod  
elasticsearch:2  
1820087a9c600f43159688050dcc164c298183e1d2e62d5694fd46b10ac3  
bc3d
```

Resolving network aliases

Let's try DNS resolution first, using the `nslookup` tool that ships with the `alpine` image.

```
$ docker run --net prod --rm alpine nslookup es
Name:      es
Address 1: 172.23.0.3 prod-es-2.prod
Address 2: 172.23.0.2 prod-es-1.prod
```

(You can ignore the `can't resolve '(null)'` errors.)

Good to know ...

- Docker will not create network names and aliases on the default **bridge** network.
- Therefore, if you want to use those features, you have to create a custom network first.
- Network aliases are *not* unique on a given network.
- i.e., multiple containers can have the same alias on the same network.
- In that scenario, the Docker DNS server will return multiple records.
(i.e. you will get DNS round robin out of the box.)
- Enabling *Swarm Mode* gives access to clustering and load balancing with IPVS.
- Creation of networks and network aliases is generally automated with tools like Compose.

A few words about round robin DNS

Don't rely exclusively on round robin DNS to achieve load balancing.

Many factors can affect DNS resolution, and you might see:

- all traffic going to a single instance;
- traffic being split (unevenly) between some instances;
- different behavior depending on your application language;
- different behavior depending on your base distro;
- different behavior depending on other factors (sic).

It's OK to use DNS to discover available endpoints, but remember that you have to re-resolve every now and then to discover new endpoints.

Network drivers

- A network is managed by a *driver*.
- The built-in drivers include:
 - `bridge` (default)
 - `none`
 - `host`
 - `macvlan`
 - `overlay` (for Swarm clusters)
- More drivers can be provided by plugins (OVS, VLAN...)
- A network can have a custom IPAM (IP allocator).

Overlay networks

- The features we've seen so far only work when all containers are on a single host.
- If containers span multiple hosts, we need an *overlay* network to connect them together.
- Docker ships with a default network plugin, `overlay`, implementing an overlay network leveraging VXLAN, *enabled with Swarm Mode*.
- Other plugins (Weave, Calico...) can provide overlay networks as well.
- Once you have an overlay network, *all the features that we've used in this chapter work identically across multiple hosts*.

Compose for development stacks

Dockerfile = great to build one container image.

What if we have multiple containers?

What if some of them require particular `docker run` parameters?

How do we connect them all together?

... Compose solves these use-cases (and a few more).

Life before Compose

Before we had Compose, we would typically write custom scripts to:

- build container images,
- run containers using these images,
- connect the containers together,
- rebuild, restart, update these images and containers.

Life with Compose

Compose enables a simple, powerful onboarding workflow:

1. Checkout our code.
2. Run `docker-compose up`.
3. Our app is up and running!

```
https://docs.docker.com/compose/install/
```

```
/usr/libexec/docker/cli-plugins/docker-compose
```

```
sudo ln -s /usr/libexec/docker/cli-plugins/docker-compose /usr/local/bin/
```

class: pic



composeup

Life after Compose

(Or: when do we need something else?)

- Compose is *not* an orchestrator
- It isn't designed to need to run containers on multiple nodes
 - (it can, however, work with Docker Swarm Mode)
- Compose isn't ideal if we want to run containers on Kubernetes
 - it uses different concepts (Compose services ≠ Kubernetes services)
 - it needs a Docker Engine (althought containerd support might be coming)

First rodeo with Compose

1. Write Dockerfiles
2. Describe our stack of containers in a YAML file called `docker-compose.yml`
3. `docker-compose up` (or `docker-compose up -d` to run in the background)
4. Compose pulls and builds the required images, and starts the containers
5. Compose shows the combined logs of all the containers
(if running in the background, use `docker-compose logs`)
6. Hit Ctrl-C to stop the whole stack
(if running in the background, use `docker-compose stop`)

Iterating

After making changes to our source code, we can:

1. `docker-compose build` to rebuild container images
2. `docker-compose up` to restart the stack with the new images

We can also combine both with `docker-compose up --build`.
Compose will be smart, and only recreate the containers that have changed.

When working with interpreted languages:

- don't rebuild each time
- leverage a `volumes` section instead

Launching Our First Stack with Compose

First step: clone the source code for the app we will be working on.

```
git clone  
https://github.com/jpetazzo/trainingwheels  
cd trainingwheels
```

Second step: start the app.

```
docker-compose up
```

Watch Compose build and run the app.

That Compose stack exposes a web server on port 8000; try connecting to it.

Launching Our First Stack with Compose

We should see a web page like this:

Training wheels

This request was served by 5457fb09c174.

5457fb09c174 served 1 request so far.

The current ladder is:

- 5457fb09c174 → 1 request

composeapp

Each time we reload, the counter should increase.

Stopping the app

When we hit Ctrl-C, Compose tries to gracefully terminate all of the containers.

After ten seconds (or if we press ^C again) it will forcibly kill them.

The `docker-compose.yml` file

Here is the file used in the demo:
`.small[]`

```
version: "2"
services:
  www:
    build: www
    ports:
      - 8000:5000
    user: nobody
    environment:
      DEBUG: 1
    command: python
    counter.py
    volumes:
      - ./www:/src
    redis:
      image: redis
```

Compose file structure

A Compose file has multiple sections:

- **version** is mandatory. (Typically use “3”.)
- **services** is mandatory. Each service corresponds to a container.
- **networks** is optional and indicates to which networks containers should be connected. (By default, containers will be connected on a private, per-compose-file network.)
- **volumes** is optional and can define volumes to be used and/or shared by the containers.

Compose file versions

- Version 1 is legacy and shouldn't be used.
(If you see a Compose file without `version` and `services`, it's a legacy v1 file.)
- Version 2 added support for networks and volumes.
- Version 3 added support for deployment options (scaling, rolling updates, etc).
- Typically use `version: "3"`.

The [Docker documentation](#) has excellent information about the Compose file format if you need to know more about versions.

Containers in docker-compose.yml

Each service in the YAML file must contain either `build`, or `image`.

- `build` indicates a path containing a Dockerfile.
- `image` indicates an image name (local, or on a registry).
- If both are specified, an image will be built from the `build` directory and named `image`.

The other parameters are optional.

They encode the parameters that you would typically add to `docker run`.

Sometimes they have several minor improvements.

Container parameters

- **command** indicates what to run (like `CMD` in a Dockerfile).
- **ports** translates to one (or multiple) `-p` options to map ports. You can specify local ports (i.e. `x:y` to expose public port `x`).
- **volumes** translates to one (or multiple) `-v` options. You can use relative paths here.

For the full list, check:

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

Environment variables

- We can use environment variables in Compose files
(like `$THIS` or `${THAT}`)
- We can provide default values, e.g. `${PORT-8000}`
- Compose will also automatically load the environment file
`.env`
(it should contain `VAR=value`, one per line)
- This is a great way to customize build and run parameters
(base image versions to use, build and run secrets, port numbers...)

Running multiple copies of a stack

- Copy the stack in two different directories, e.g. `front` and `frontcopy`
- Compose prefixes images and containers with the directory name:
`front_www`, `front_www_1`, `front_db_1`
`frontcopy_www`, `frontcopy_www_1`, `frontcopy_db_1`
- Alternatively, use `docker-compose -p frontcopy`
(to set the `--project-name` of a stack, which default to the dir name)
- Each copy is isolated from the others (runs on a different network)

Checking stack status

We have ps, docker ps, and similarly, docker-compose ps:

Name	Command
State	Ports
<hr/>	
<hr/>	
trainingwheels_redis_1	/entrypoint.sh red Up
6379/tcp	
trainingwheels_www_1	python counter.py Up
0.0.0.0:8000->50007tcp	

Shows the status of all the containers of our stack.
Doesn't show the other containers.

Cleaning up (1)

If you have started your application in the background with Compose and want to stop it easily, you can use the `kill` command:

```
$ docker-compose kill
```

Likewise, `docker-compose rm` will let you remove containers (after confirmation):

```
$ docker-compose rm
Going to remove trainingwheels_redis_1,
trainingwheels_www_1
Are you sure? [yN] -y
Removing trainingwheels_redis_1...
Removing trainingwheels_www_1...
```

Cleaning up (2)

Alternatively, `docker-compose down` will stop and remove containers.

It will also remove other resources, like networks that were created for the application.

```
$ docker-compose down
Stopping trainingwheels_www_1 ... done
Stopping trainingwheels_redis_1 ... done
Removing trainingwheels_www_1 ... done
Removing trainingwheels_redis_1 ... done
```

Use `docker-compose down -v` to remove everything including volumes.

Special handling of volumes

- When an image gets updated, Compose automatically creates a new container
- The data in the old container is lost...
- ... Except if the container is using a *volume*
- Compose will then re-attach that volume to the new container
 - (and data is then retained across database upgrades)
- All good database images use volumes
 - (e.g. all official images)

Exercise — writing Dockerfiles

Let's write Dockerfiles for an existing application!

1. Check out the code repository
2. Read all the instructions
3. Write Dockerfiles
4. Build and test them individually

Code repository

Clone the repository available at:

<https://github.com/jpetazzo/wordsmith>

It should look like this:

```
├── LICENSE
├── README
├── db/
│   └── words.sql
├── web/
│   ├── dispatcher.go
│   └── static/
└── words/
    ├── pom.xml
    └── src/
```

Instructions

The repository contains instructions in English and French. For now, we only care about the first part (about writing Dockerfiles). Place each Dockerfile in its own directory, like this:

```
└── LICENSE
└── README
└── db/
    ├── `Dockerfile`
    └── words.sql
└── web/
    ├── `Dockerfile`
    ├── dispatcher.go
    └── static/
└── words/
    ├── `Dockerfile`
    ├── pom.xml
    └── src/
```

Exercise — writing a Compose file

Let's write a Compose file for the wordsmith app!

The code is at: <https://github.com/jpetazzo/wordsmith>

Build and test

Build and run each Dockerfile individually.

For `db`, we should be able to see some messages confirming that the data set was loaded successfully (some `INSERT` lines in the container output).

For `web` and `words`, we should be able to see some message looking like "server started successfully".

That's all we care about for now!

Bonus question: make sure that each container stops correctly when hitting Ctrl-C.

???

Test with a Compose file

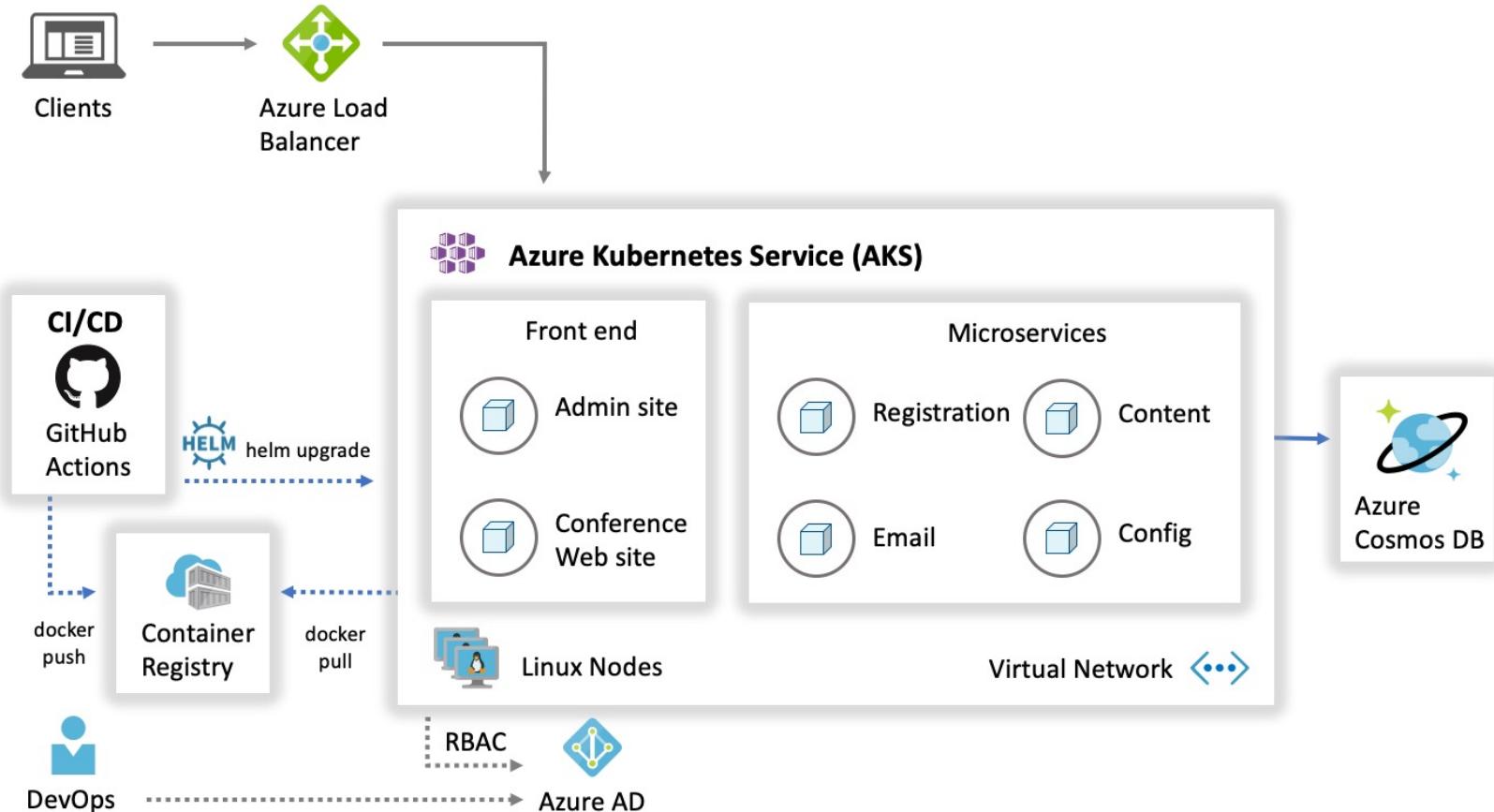
Place the following Compose file at the root of the repository:

```
version: "3"
services:
  db:
    build: db
  words:
    build: words
  web:
    build: web
    ports:
      - 8888:80
```

Test the whole app by bringin up the stack and connecting to port 8888.

MODERNIZACIÓN APLICACIÓN

- Pruebas Aplicación Legacy
- Migración a Docker
- Publicación en Registro
- Despliegue en AKS
- Pruebas y navegación
- Debugging/Logging



Orchestration, an overview

In this chapter, we will:

- Explain what is orchestration and why we would need it.
- Present (from a high-level perspective) some orchestrators.

class: pic

What's orchestration?



Joana Carneiro (orchestra conductor)

What's orchestration?

According to Wikipedia:

*Orchestration describes the **automated** arrangement, coordination, and management of complex computer systems, middleware, and services.*

—
[...] orchestration is often discussed in the context of **service-oriented architecture**, **virtualization**, provisioning, Converged Infrastructure and **dynamic datacenter** topics.

—
What does that really mean?

TL,DR

- Scheduling with multiple resources (dimensions) is hard.
- Don't expect to solve the problem with a Tiny Shell Script.
- There are literally tons of research papers written on this.

But our orchestrator also needs to manage ...

- Network connectivity (or filtering) between containers.
- Load balancing (external and internal).
- Failure recovery (if a node or a whole datacenter fails).
- Rolling out new versions of our applications.
(Canary deployments, blue/green deployments...)

Some orchestrators

We are going to present briefly a few orchestrators.

There is no “absolute best” orchestrator.

It depends on:

- your applications,
- your requirements,
- your pre-existing skills...

Nomad

- Open Source project by Hashicorp.
- Arbitrary scheduler (not just for containers).
- Great if you want to schedule mixed workloads.
(VMs, containers, processes...)
- Less integration with the rest of the container ecosystem.

Mesos

- Open Source project in the Apache Foundation.
- Arbitrary scheduler (not just for containers).
- Two-level scheduler.
- Top-level scheduler acts as a resource broker.
- Second-level schedulers (aka “frameworks”) obtain resources from top-level.
- Frameworks implement various strategies.
(Marathon = long running processes; Chronos = run at intervals; ...)
- Commercial offering through DC/OS by Mesosphere.

Rancher

- Rancher 1 offered a simple interface for Docker hosts.
- Rancher 2 is a complete management platform for Docker and Kubernetes.
- Technically not an orchestrator, but it's a popular option.

Swarm

- Tightly integrated with the Docker Engine.
- Extremely simple to deploy and setup, even in multi-manager (HA) mode.
- Secure by default.
- Strongly opinionated:
 - smaller set of features,
 - easier to operate.

Kubernetes

- Open Source project initiated by Google.
- Contributions from many other actors.
- *De facto* standard for container orchestration.
- Many deployment options; some of them very complex.
- Reputation: steep learning curve.
- Reality:
 - true, if we try to understand *everything*;
 - false, if we focus on what matters.

NobleProg

Kubernetes intro & advanced

The World's Local Training Provider

NobleProg® Limited 2017
All Rights Reserved

Kubernetes: the industry leading orchestrator



Portable

Public, private, hybrid,
multi-cloud



Extensible

Modular, pluggable,
hookable, composable



Self-healing

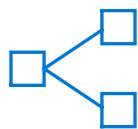
Auto-placement, auto-restart,
auto-replication, auto-scaling

Work how you want with opensource tools and APIs

	Development	DevOps	Monitoring	Networking	Storage	Security
Take advantage of services and tools in the Kubernetes ecosystem	 	    	    	 	 	   RBAC



Lift and shift to containers



Microservices



Machine learning

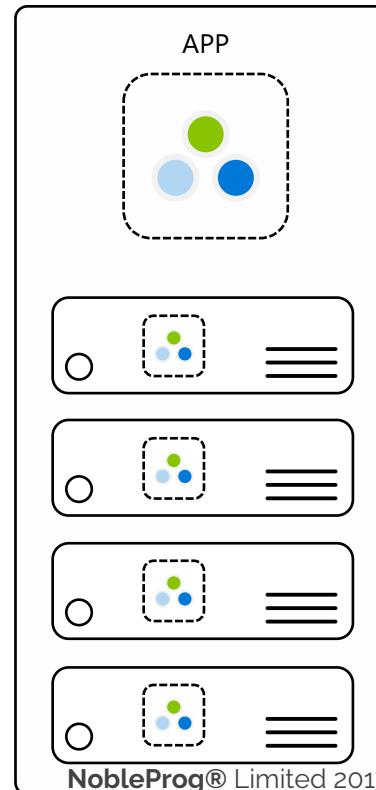


IoT

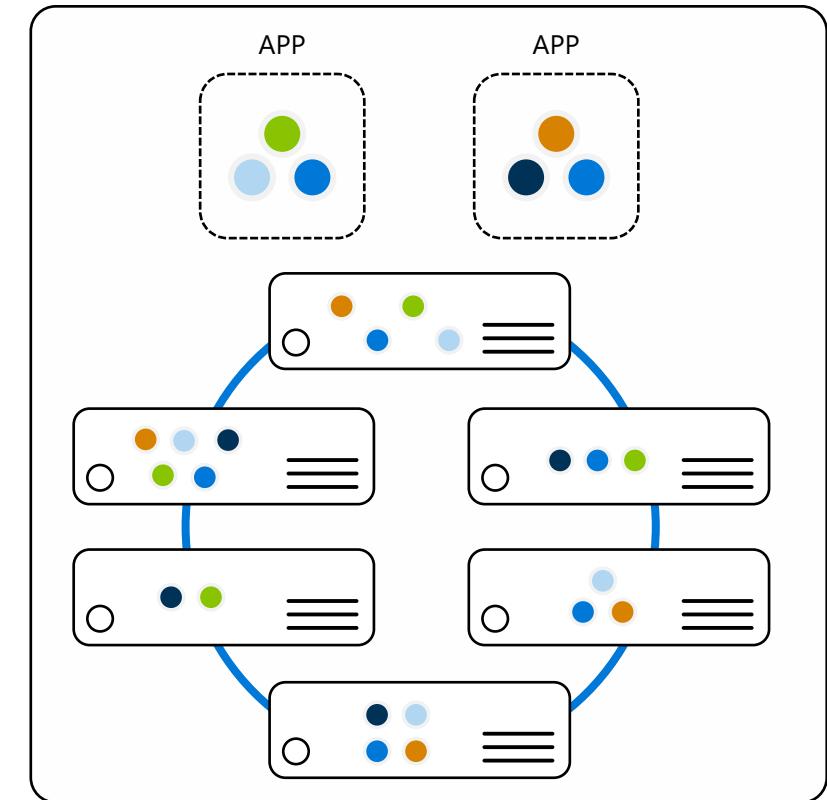
Microservices: for faster app development

- Independent deployments
- Improved scale and resource utilization per service
- Smaller, focused teams

Monolithic
Large, all-inclusive app



Microservices
Small, independent services



What is Kubernetes?

Background

- "Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications"
- Schedules and runs application containers across a cluster of machines
- Kubernetes v1.0 released on July 21, 2015. Joe Beda, Brendan Burns, & Craig McLuckie

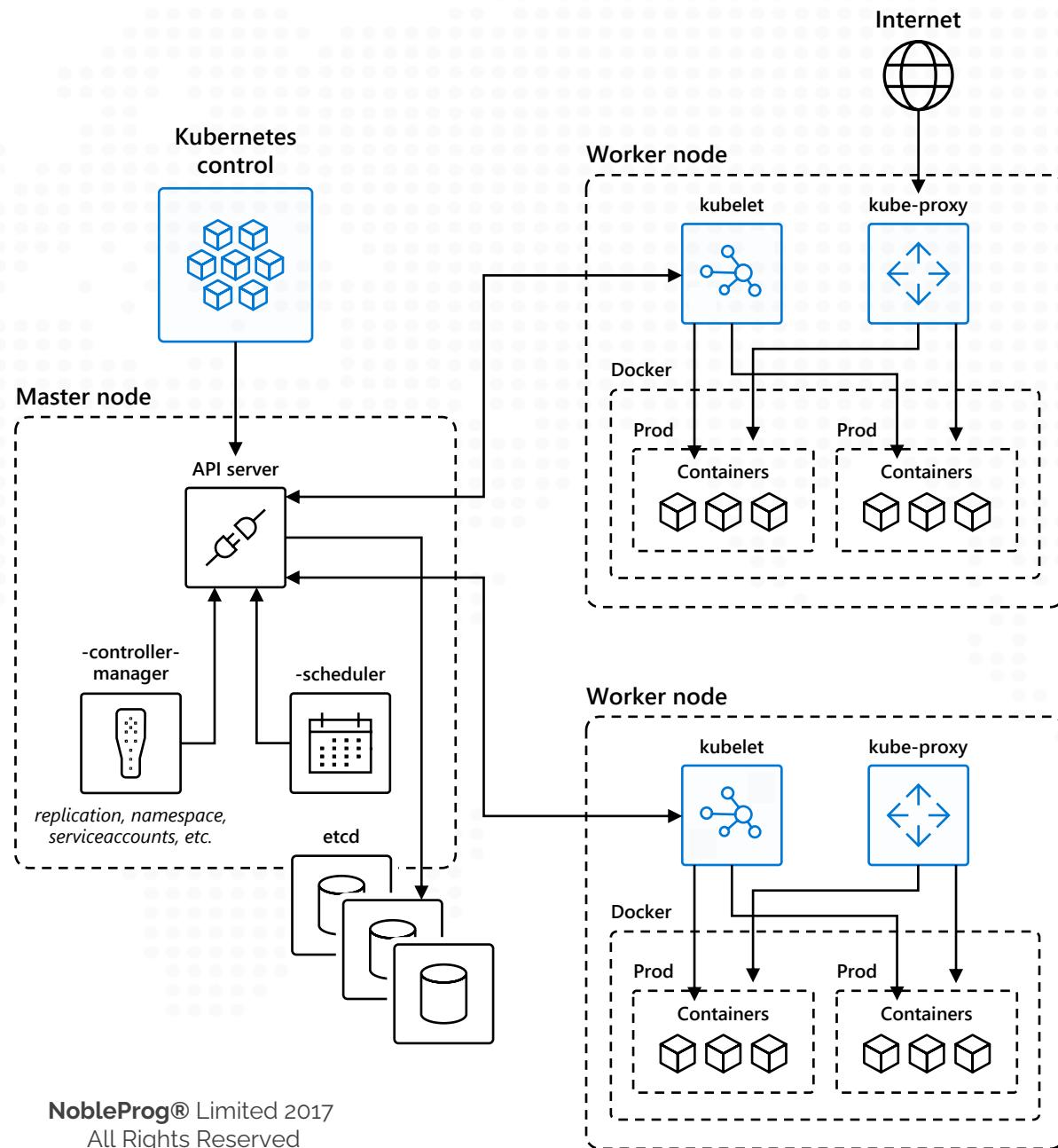
Key features

- Declarative infrastructure
- Self-healing
- Horizontal scaling
- Automated rollouts and rollbacks
- Service discovery and load balancing
- Automatic bin packing
- Storage orchestration
- Secret and configuration management
- Not a PaaS platform

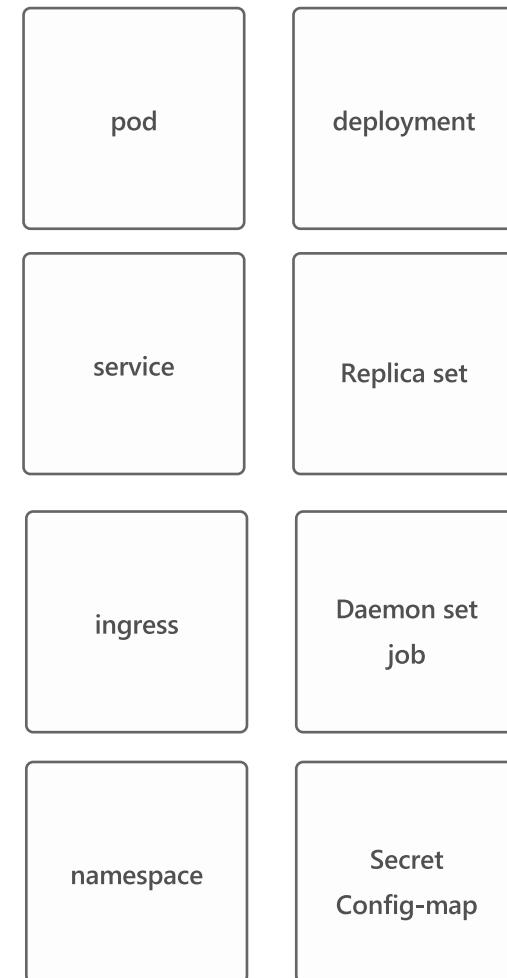


Kubernetes 101

1. Kubernetes users communicate with API server and apply desired state
2. Master nodes actively enforce desired state on worker nodes
3. Worker nodes support communication between containers
4. Worker nodes support communication from the Internet



Kubernetes Resources



Kubernetes Resources: Namespaces

Provide grouping or Kubernetes resources

to enable:

- RBAC
- Affinity
- Quotas
- Policy (Cluster & Network)

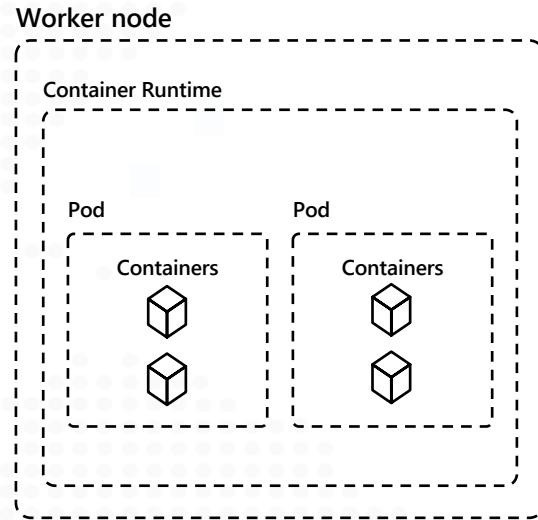
Default K8s Namespaces:

- default
- kube-node-lease
- kube-public
- kube-system

```
griffith ➔ ~/temp ➔
└─ kubectl get namespaces
  NAME          STATUS   AGE
  azure-arc      Active  12d
  default        Active  12d
  falco         Active  32h
  kube-node-lease Active  12d
  kube-public    Active  12d
  kube-system    Active  12d
  monitoring     Active  32h
```

Kubernetes Resources: Pods

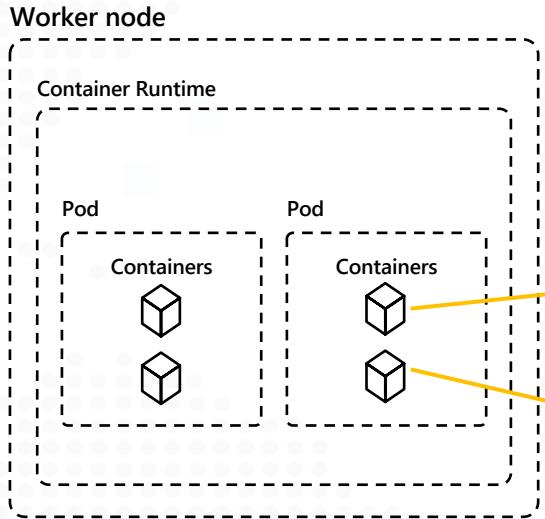
Zero to Many Containers Per Pod



Multi-Container Common Patterns:

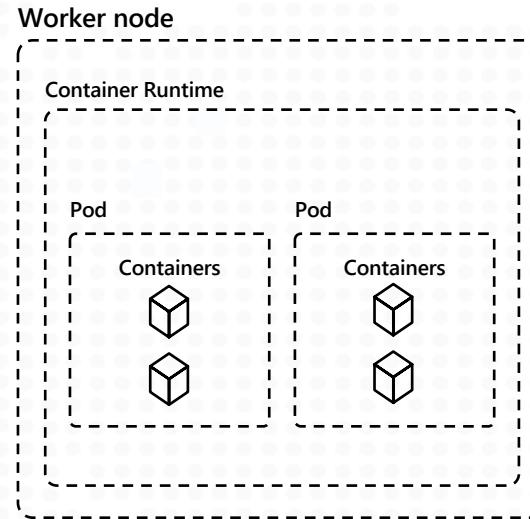
- Sidecar: Enhance or extend a container
- Ambassador: Proxy network calls
- Adapter: Transform output

Kubernetes Resources: Pods - Adapter Example



```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: sidecar-demo
    name: sidecar-demo
spec:
  containers:
    - image: nginx
      name: webserver
      volumeMounts:
        - mountPath: /var/log/nginx/
          name: log-volume
      resources: {}
    - image: busybox
      name: logaggregator
      args: [/bin/sh, -c, 'tail -f /var/log/nginx/error.log>/var/log/nginx/nginx.errors']
      volumeMounts:
        - mountPath: /var/log/nginx/
          name: log-volume
      resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
  volumes:
    - name: log-volume
      emptyDir: {}
```

Kubernetes Resources: Pods – Adapter/Sidecar Example



```
griffith ~ ~
k run -it busybox --rm --image=busybox --restart=Never -- /bin/sh
If you don't see a command prompt, try pressing enter.
/ # wget -O- http://10.244.2.39/all-the-fails
Connecting to 10.244.2.39 (10.244.2.39:80)
wget: server returned error: HTTP/1.1 404 Not Found
/ #
```

```
griffith ~ ~
k exec -it sidecar-demo --container=logaggregator -- tail -f /var/log/nginx/nginx.errors
2020/02/24 22:56:34 [error] 7#7: *3 open() "/usr/share/nginx/html/fail" failed (2: No such file or directory), client: 10.244.0.42, server: localhost, request: "GET /fail HTTP/1.1", host: "10.244.2.39"
2020/02/24 22:56:40 [error] 7#7: *4 open() "/usr/share/nginx/html/fail" failed (2: No such file or directory), client: 10.244.0.42, server: localhost, request: "GET /fail HTTP/1.1", host: "10.244.2.39"
2020/02/24 23:09:54 [error] 7#7: *5 open() "/usr/share/nginx/html/failmore" failed (2: No such file or directory), client: 10.244.0.46, server: localhost, request: "GET /failmore HTTP/1.1", host: "10.244.2.39"
2020/02/24 23:11:14 [error] 7#7: *6 open() "/usr/share/nginx/html/failagain" failed (2: No such file or directory), client: 10.244.0.47, server: localhost, request: "GET /failagain HTTP/1.1", host: "10.244.2.39"
2020/02/24 23:17:05 [error] 7#7: *7 open() "/usr/share/nginx/html/all-the-fails" failed (2: No such file or directory), client: 10.244.0.50, server: localhost, request: "GET /all-the-fails HTTP/1.1", host: "10.244.2.39"
```

Kubernetes Resources: DaemonSet

Runs a given pod on every node

Typically used for monitoring agents or system processes you want running on every node

```
griffith ~ 
└── kubectl get nodes
NAME           STATUS  ROLES   AGE    VERSION
aks-nodepool1-30239456-vmss000000  Ready   agent   12d   v1.14.8
aks-nodepool1-30239456-vmss000001  Ready   agent   12d   v1.14.8
aks-nodepool1-30239456-vmss000002  Ready   agent   12d   v1.14.8

griffith ~ 
└── kubectl get daemonsets --all-namespaces
NAMESPACE      NAME        DESIRED  CURRENT  READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR          AGE
falco         sysdig-falco  3         3         3         3           3           <none>          32h
kube-system   kube-proxy   3         3         3         3           3           beta.kubernetes.io/os=linux  12d
kube-system   omsagent    3         3         3         3           3           beta.kubernetes.io/os=linux  5d6h

griffith ~ 
└── kubectl get pods -l dsName=omsagent-ds -n kube-system -o wide
NAME        READY  STATUS    RESTARTS  AGE     IP          NODE
omsagent-cjg5m  1/1    Running   0        5d6h   10.244.1.15  aks-nodepool1-30239456-vmss000001
omsagent-hmqdr  1/1    Running   0        5d6h   10.244.2.26  aks-nodepool1-30239456-vmss000002
omsagent-kc5sn  1/1    Running   0        5d6h   10.244.0.26  aks-nodepool1-30239456-vmss000000
```

Kubernetes Resources: Jobs/CronJobs

A task that runs to completion, possibly on a schedule via Cron

```
griffith ➜ ~/temp
└── kubectl get cronjobs
  NAME      SCHEDULE      SUSPEND      ACTIVE      LAST SCHEDULE      AGE
  hello    */1 * * * *    False        0           20s          6m45s

griffith ➜ ~/temp
└── kubectl get jobs
  NAME              COMPLETIONS      DURATION      AGE
  hello-1582600080  1/1            7s           2m15s
  hello-1582600140  1/1            5s           75s
  hello-1582600200  1/1            2s           14s

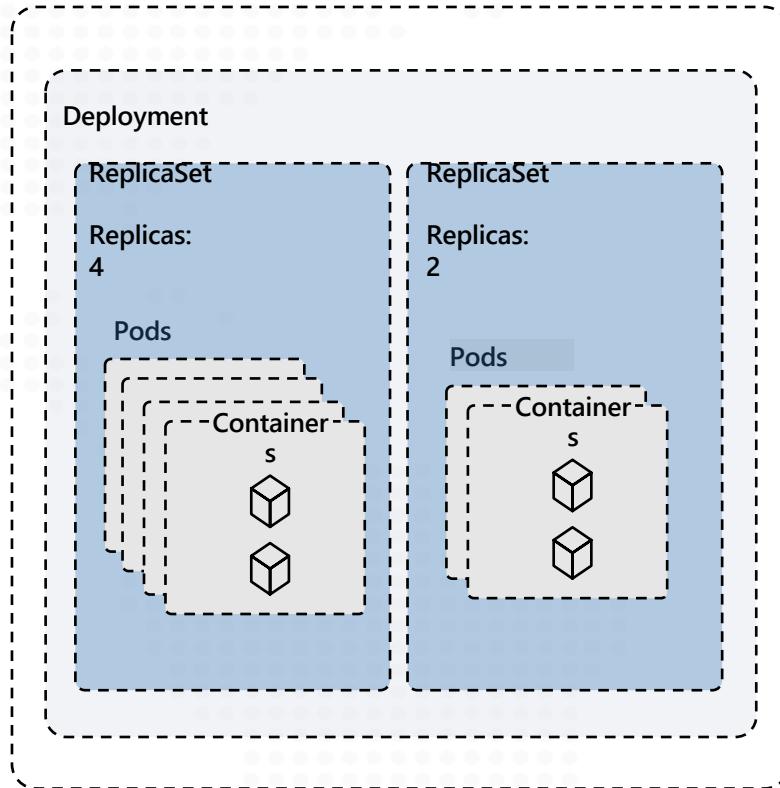
griffith ➜ ~/temp
└── kubectl get pods
  NAME          READY      STATUS      RESTARTS      AGE
  hello-1582600080-xwchd  0/1      Completed   0           2m19s
  hello-1582600140-6vff7  0/1      Completed   0           79s
  hello-1582600200-gfnpd  0/1      Completed   0           18s

griffith ➜ ~/temp
└── kubectl logs hello-1582600200-gfnpd
Tue Feb 25 03:10:12 UTC 2020
Hello from the Kubernetes cluster
```

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
        restartPolicy: OnFailure
```

Kubernetes Resources: Replica Sets

Worker node

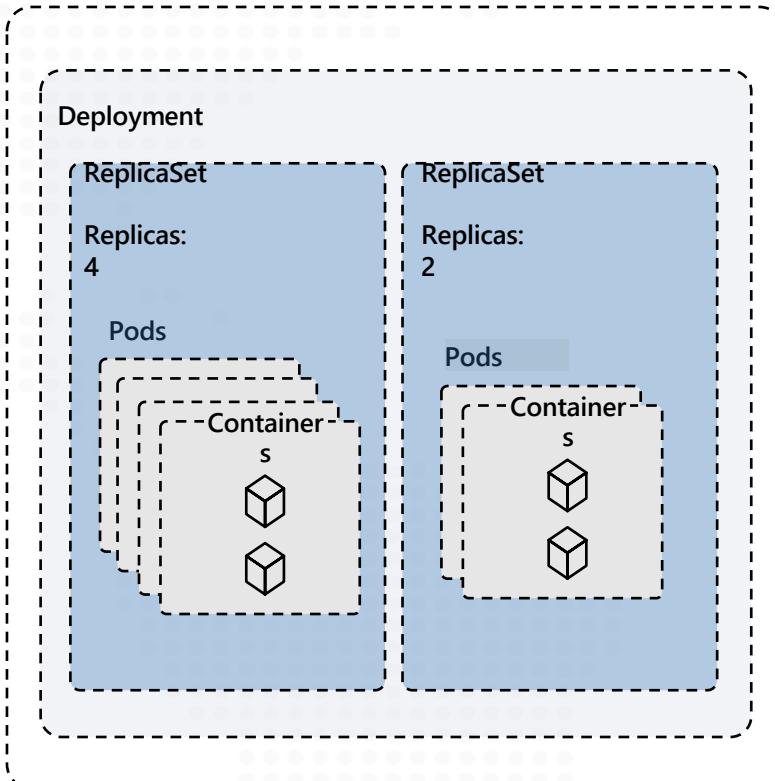


```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: rs-demo
    name: rs-demo
spec:
  replicas: 10
  selector:
    matchLabels:
      run: rs-demo
  strategy: {}
  template:
    metadata:
      labels:
        run: rs-demo
    spec:
      containers:
        - image: nginx
          name: rs-demo
      resources: {}
```

NAME	READY	STATUS	RESTARTS	AGE
replicaset.extensions/rs-demo-6749fd79c6	10	10	10	13s
pod/rs-demo-6749fd79c6-4n5x6	1/1	Running	0	13s
pod/rs-demo-6749fd79c6-8qwgw	1/1	Running	0	13s
pod/rs-demo-6749fd79c6-f9xv4	1/1	Running	0	13s
pod/rs-demo-6749fd79c6-fk4tb	1/1	Running	0	13s
pod/rs-demo-6749fd79c6-h6lbx	1/1	Running	0	13s
pod/rs-demo-6749fd79c6-hbk8s	1/1	Running	0	13s
pod/rs-demo-6749fd79c6-jg2ts	1/1	Running	0	13s
pod/rs-demo-6749fd79c6-pxrcf	1/1	Running	0	13s
pod/rs-demo-6749fd79c6-rk6ll	1/1	Running	0	13s
pod/rs-demo-6749fd79c6-whvpj	1/1	Running	0	13s

Kubernetes Resources: Deployments

Worker node



```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: rs-demo
  name: rs-demo
spec:
  replicas: 10
  selector:
    matchLabels:
      run: rs-demo
  strategy: {}
  template:
    metadata:
      labels:
        run: rs-demo
    spec:
      containers:
        - image: nginx
          name: rs-demo
      resources: {}
```

```
griffith ➜ ~/temp
└── kubectl get deployments
  NAME      READY   UP-TO-DATE   AVAILABLE   AGE
  rs-demo   10/10   10           10          7m26s

griffith ➜ ~/temp
└── kubectl scale deployment rs-demo --replicas=5
deployment.extensions/rs-demo scaled

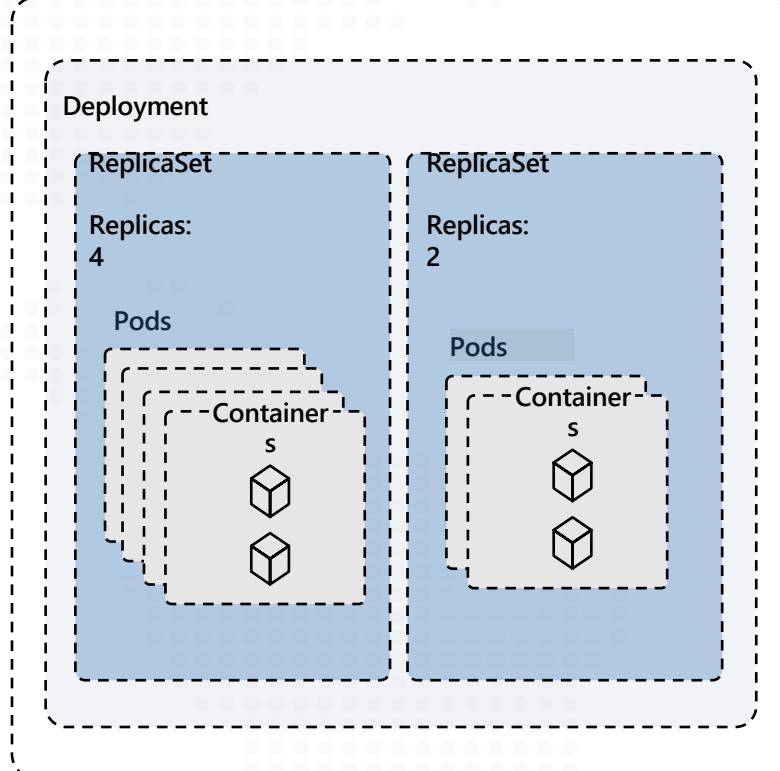
griffith ➜ ~/temp
└── kubectl get deployments,rs,pods
  NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
  deployment.extensions/rs-demo   5/5     5           5          7m45s

  NAME                           DESIRED  CURRENT   READY   AGE
  replicaset.extensions/rs-demo-6749fd79c6   5       5         5       7m45s

  NAME          READY   STATUS    RESTARTS   AGE
  pod/rs-demo-6749fd79c6-4n5x6   1/1     Running   0          7m47s
  pod/rs-demo-6749fd79c6-fk4tb   1/1     Running   0          7m47s
  pod/rs-demo-6749fd79c6-h6lbx   1/1     Running   0          7m47s
  pod/rs-demo-6749fd79c6-hbk8s   1/1     Running   0          7m47s
  pod/rs-demo-6749fd79c6-pxrcf   1/1     Running   0          7m47s
```

Kubernetes Resources: Deployment History and Rollback

Worker node



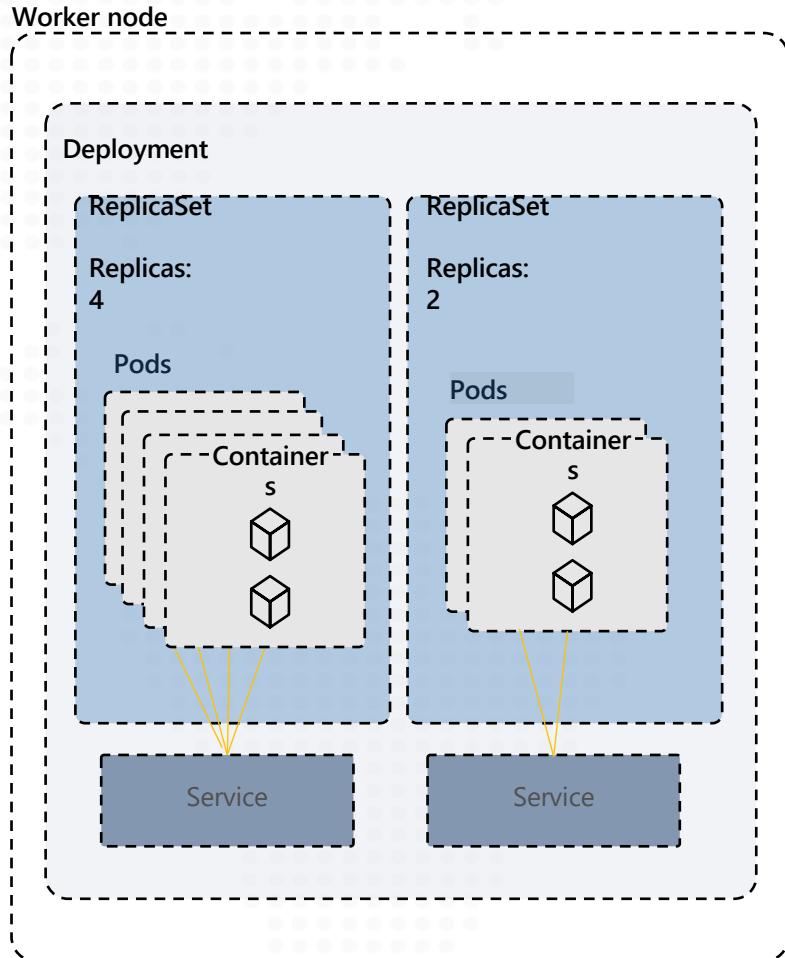
```
griffith ➤ ~/temp
kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
rs-demo   8/8     8           8           23s

griffith ➤ ~/temp
kubectl rollout history deployment rs-demo
deployment.extensions/rs-demo
REVISION  CHANGE-CAUSE
1         kubectl apply --filename=rsdemo.yaml --record=true

griffith ➤ ~/temp
kubectl set image deployment.extensions/rs-demo rs-demo=nginx:1.17.8 --record=true
deployment.extensions/rs-demo image updated
griffith ➤ ~/temp
kubectl rollout history deployment rs-demo
deployment.extensions/rs-demo
REVISION  CHANGE-CAUSE
1         kubectl apply --filename=rsdemo.yaml --record=true
2         kubectl set image deployment.extensions/rs-demo rs-demo=nginx:1.17.8 --record=true

griffith ➤ ~/temp
kubectl rollout undo deployment rs-demo --to-revision=1
deployment.extensions/rs-demo rolled back
griffith ➤ ~/temp
kubectl rollout history deployment rs-demo
deployment.extensions/rs-demo
REVISION  CHANGE-CAUSE
2         kubectl set image deployment.extensions/rs-demo rs-demo=nginx:1.17.8 --record=true
3         kubectl apply --filename=rsdemo.yaml --record=true
```

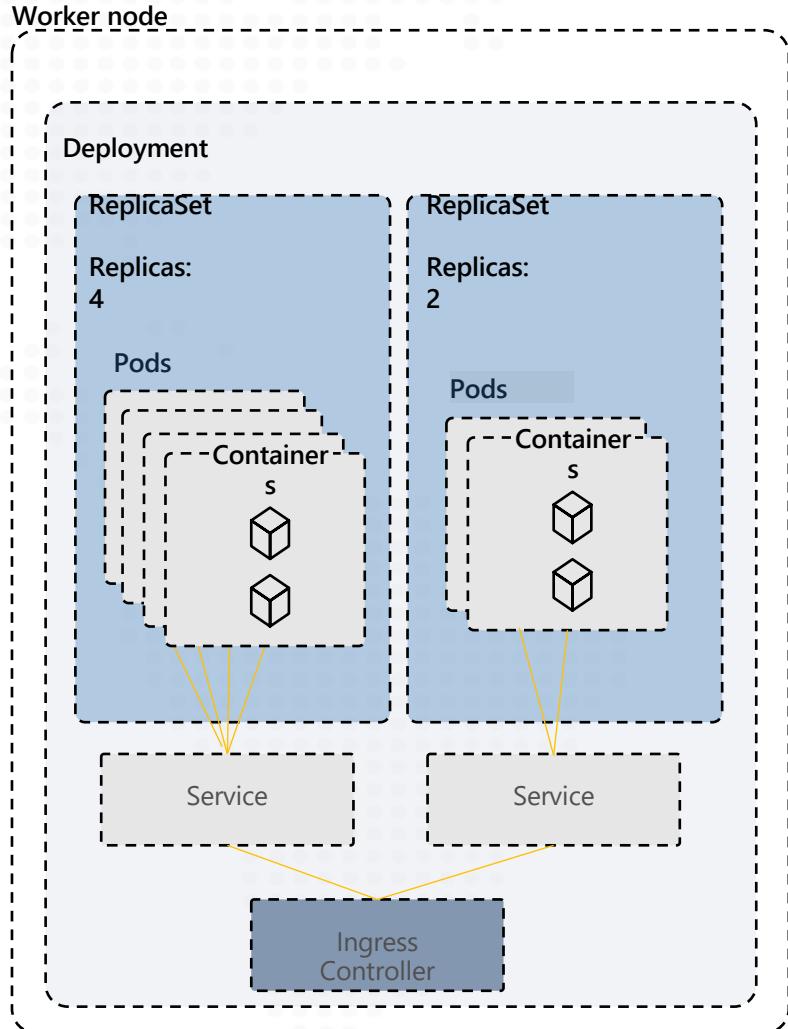
Kubernetes Resources: Service



Provides **Layer 4 Load Balancing**
Types:

- **ClusterIP:** Service is provided an IP internal to the cluster
- **NodePort:** Allocates a port used to access the service across all nodes
- **LoadBalancer:** Exposes the service via a cloud provider loadbalancer (ex. Azure internal or external LB)
- **ExternalName:** Expose the service via cluster DNS mapping

Kubernetes Resources: Ingress



Provides Layer 7 Load Balancing

Kubernetes provides the basic API spec but third parties provide implementations, often adding features via Custom Resource Definitions

Common Ingress Controller Implementations

- Nginx
- Traefik
- Gloo
- Kong
- Azure App Gateway
- etc

Common Features:

- SSL Offload
- Routing (including Canary & A/B)
- WAF

Kubernetes Architecture Components

api-server

etcd

controller-manager

scheduler

kubelet

kube-proxy

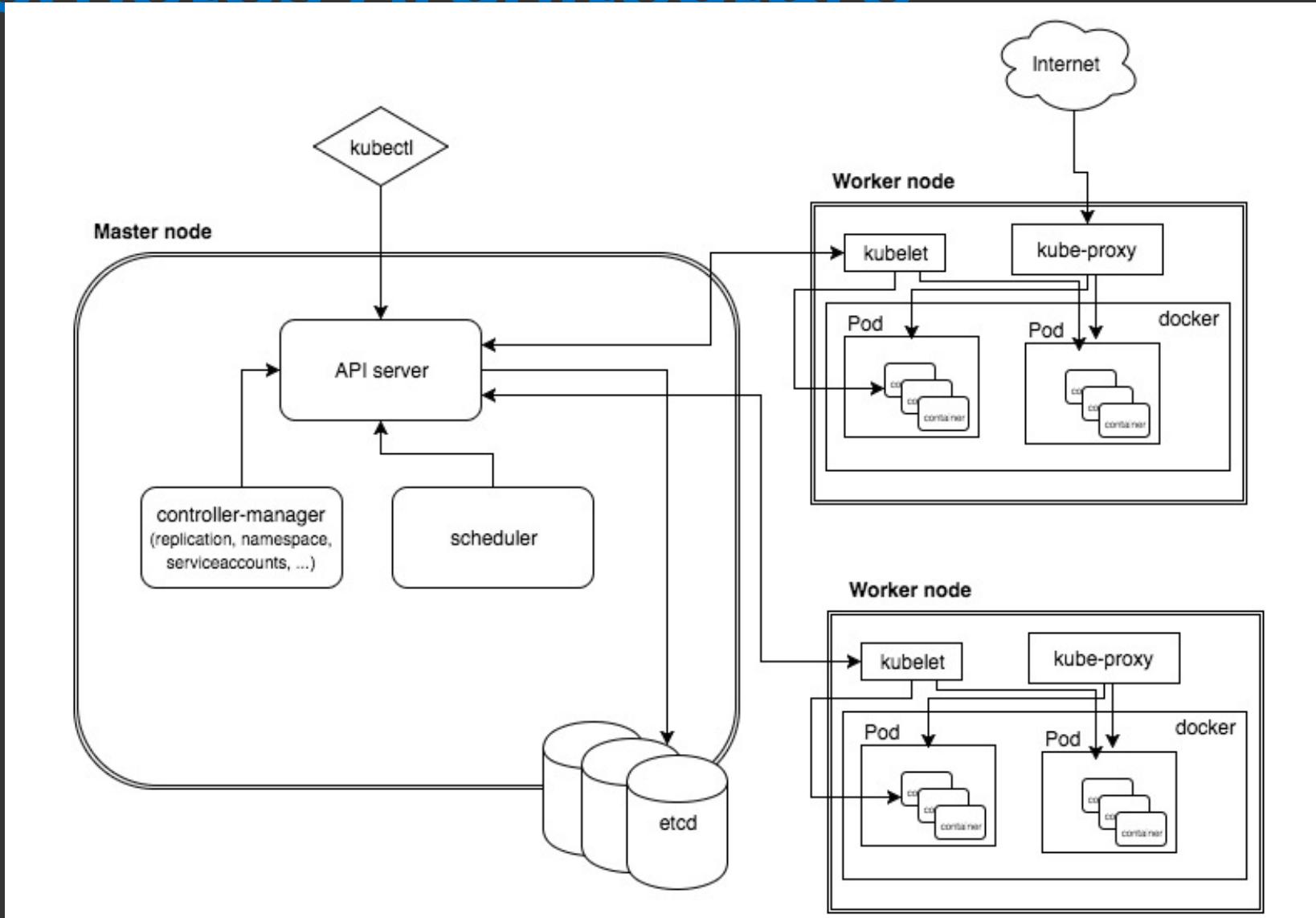
docker

dns

master
components

node
components

Kubernetes Architecture



Master Components

- api-server: Front-end control plane. Exposes API
- etcd: Cluster database. Distributed, highly available
- controller-manager. Runs controllers, eg. replication controller, node controller
- scheduler: assigns pods to nodes
- Add-ons
 - DNS
 - Heapster. enables monitoring and performance analysis
 - Dashboard
 - Logging

Worker Node Components

- kubelet:
 - Primary node agent
 - Watches and runs assigned pods
 - Executes health probes and reports status
- kube-proxy: enables network services
- docker: container engine (rkt supported experimentally)

kubectl: CLI to run commands against a Kubernetes cluster

- Swiss Army Knife: run deployments, exec into containers, view logs, etc.
- Syntax largely the same as docker
- Pronounced “koob sea tee el”...
 - Or “koob cuddle”

Kubernetes Resources

pod

deployment

service

replica set

ingress

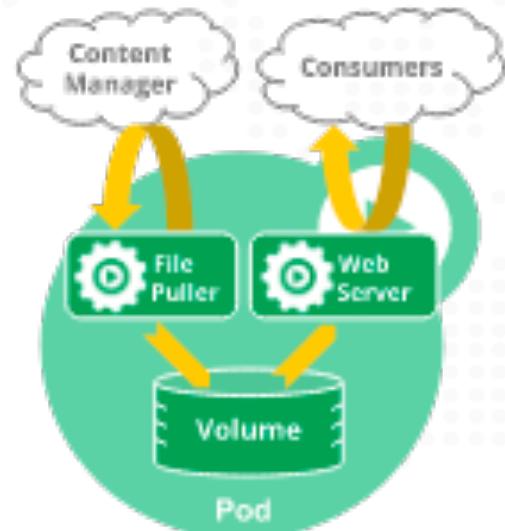
daemon set, job

namespace

secret, config-map

What is a pod?

- Pod is the basic building block in kubernetes
- Pods are how containers are delivered
- Can be multiple containers (eg - side car)
- Encapsulates container(s), storage, network IP, and options on how to run
- Use Deployment resources to deploy
 - ReplicaSet
 - StatefulSet
 - DaemonSet
 - Job
 - InitContainer



Kubernetes manifest: Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: redis-django
  labels:
    app: web
spec:
  containers:
    - name: key-value-store
      image: redis
      ports:
        - containerPort: 6379
    - name: frontend
      image: django
      ports:
        - containerPort: 8000
```

Interact with pods

```
$ kubectl get po --all-namespaces
```

```
$ kubectl describe po/my-pod
```

```
$ kubectl logs my-pod
```

```
# Run bash in container
```

```
$ kubectl exec -it my-pod bash
```

Kubernetes Services

- Defines a logical set of pods (your *microservice*)
- Essentially a virtual load balancer in front of pods

Service Types

- **ClusterIP:**

- Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster

- **NodePort:**

- Exposes the service on each Node's IP at a static port (the NodePort)
 - Connect from outside the cluster by requesting <NodeIP>:<NodePort>

- **LoadBalancer:**

- Exposes the service externally using a cloud provider's load balancer

Kubernetes manifest: Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: web
  type: ClusterIP
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

Deployment

- Provides declarative updates for Pods and Replica Sets
- Deployment describes "desired state"
- Can:
 - Create deployment to rollout ReplicaSet
 - Declare new state for pods (eg – new imageTag)
 - Rollback to earlier state
 - Scale up/down
 - Check rollout history
 - Clean-up

Kubernetes manifest: Deployment

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: smackweb-deploy
spec:
  replicas: 5
  template:
    metadata:
      labels:
        app: smackweb
    spec:
      containers:
        - name: smackweb
          image: chzbrgr71/smackweb
          ports:
            - containerPort: 8080
```

Namespaces

- Allow for multiple virtual clusters backed by the same physical cluster
- Logical separation
- Namespace used in FQDN of Kubernetes services
 - Eg - <service-name>.<namespace-name>.svc.cluster.local
- Every Kubernetes resource type is scoped to a namespace (except for nodes, persistentVolumes, etc.)
- Intended for environments with many users, teams, projects

Labels and Selectors

- Not related CSS...I promise
- Labels are key/value pairs for any API object in Kubernetes
- "Label selectors" == queries against labels to match objects
- Use cases:
 - Associating pods to a service
 - Pinning workloads to specific nodes
 - Selecting a subset of resources

Pod

Name: my-app-xaj712

Labels:

version=1

app=my-app

ReplicaSet

Replicas: 2

Label Selectors:

version=1

app=my-app

Pod

Name: my-app-xaj712

Labels:

version=1

app=my-app

Pod

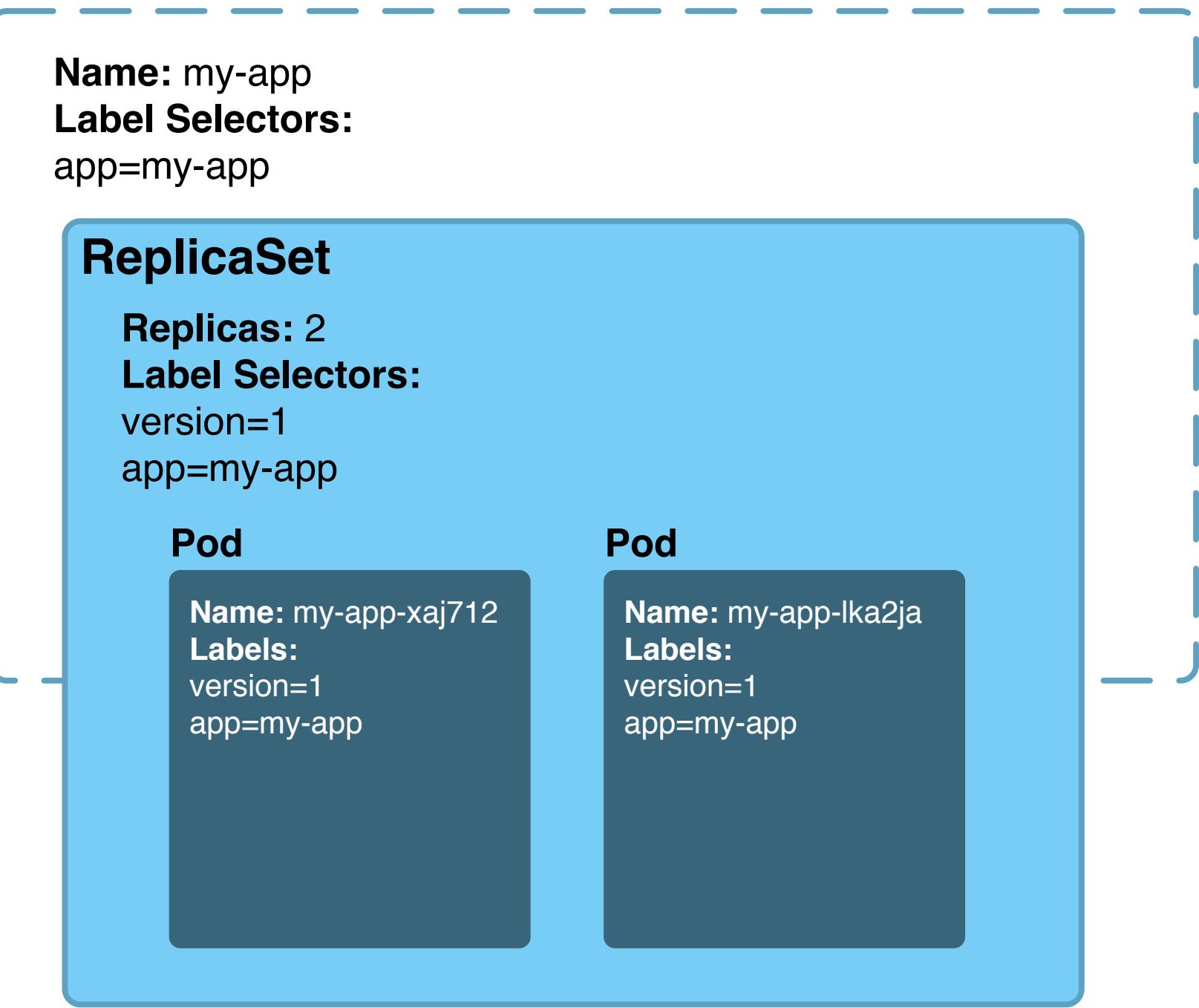
Name: my-app-lka2ja

Labels:

version=1

app=my-app

Service: My App



Name: my-app

Label Selectors:

app=my-app

ReplicaSet

Replicas: 2

Label Selectors:

version=1

app=my-app

Pod

Name: my-app-xaj712

Labels:

version=1

app=my-app

Pod

Name: my-app-lka2ja

Labels:

version=1

app=my-app

Service: My App

Name: my-app

Label Selectors:

app=my-app

ReplicaSet

Replicas: 2

Label Selectors:

version=1

app=my-app

Pod

Name: my-app-xaj712

Labels:

version=1

app=my-app

Pod

Name: my-app-lka2ja

Labels:

version=1

app=my-app

Pod

Name: my-app-123hfa

Labels:

version=canary

app=my-app

Service: My App

Name: my-app

Label Selectors:

app=my-app

ReplicaSet

Replicas: 2

Label Selectors:

version=1

app=my-app

Pod

Name: my-app-xaj712

Labels:

version=1

app=my-app

Pod

Name: my-app-lka2ja

Labels:

version=1

app=my-app

ReplicaSet

Replicas: 1

Label Selectors:

version=2

app=my-app

Service: My App

Name: my-app

Label Selectors:

app=my-app

ReplicaSet

Replicas: 2

Label Selectors:

version=1

app=my-app

Pod

Name: my-app-xaj712

Labels:

version=1

app=my-app

Pod

Name: my-app-lka2ja

Labels:

version=1

app=my-app

ReplicaSet

Replicas: 1

Label Selectors:

version=2

app=my-app

Pod

Name: my-app-19sdfd

Labels:

version=2

app=my-app

Service: My App

Name: my-app

Label Selectors:

app=my-app

ReplicaSet

Replicas: 1

Label Selectors:

version=1

app=my-app

Pod

Name: my-app-xaj712

Labels:

version=1

app=my-app

ReplicaSet

Replicas: 2

Label Selectors:

version=2

app=my-app

Pod

Name: my-app-19sdfd

Labels:

version=2

app=my-app

Pod

Name: my-app-xaj712

Labels:

version=2

app=my-app

Service: My App

Name: my-app

Label Selectors:

app=my-app

ReplicaSet

Replicas: 0

Label Selectors:

version=1

app=my-app

ReplicaSet

Replicas: 2

Label Selectors:

version=2

app=my-app

Pod

Name: my-app-19sdfd

Labels:

version=2

app=my-app

Pod

Name: my-

app-0q2a87

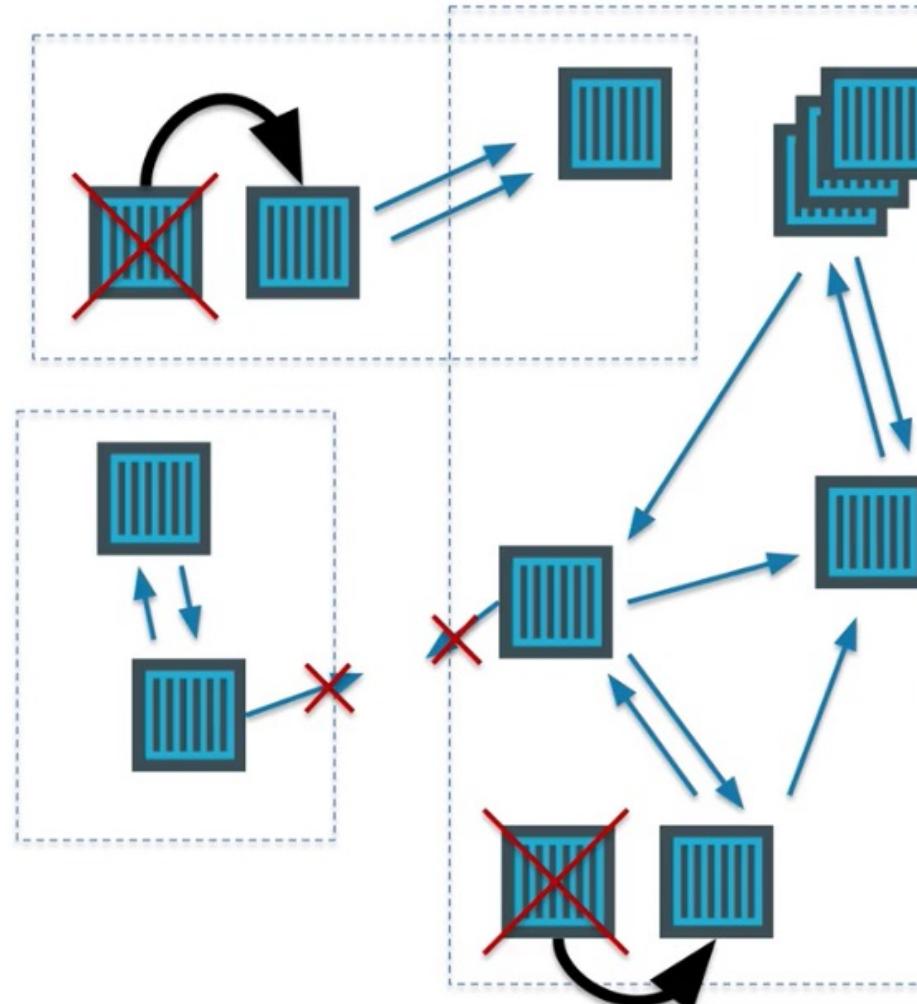
Labels:

version=2

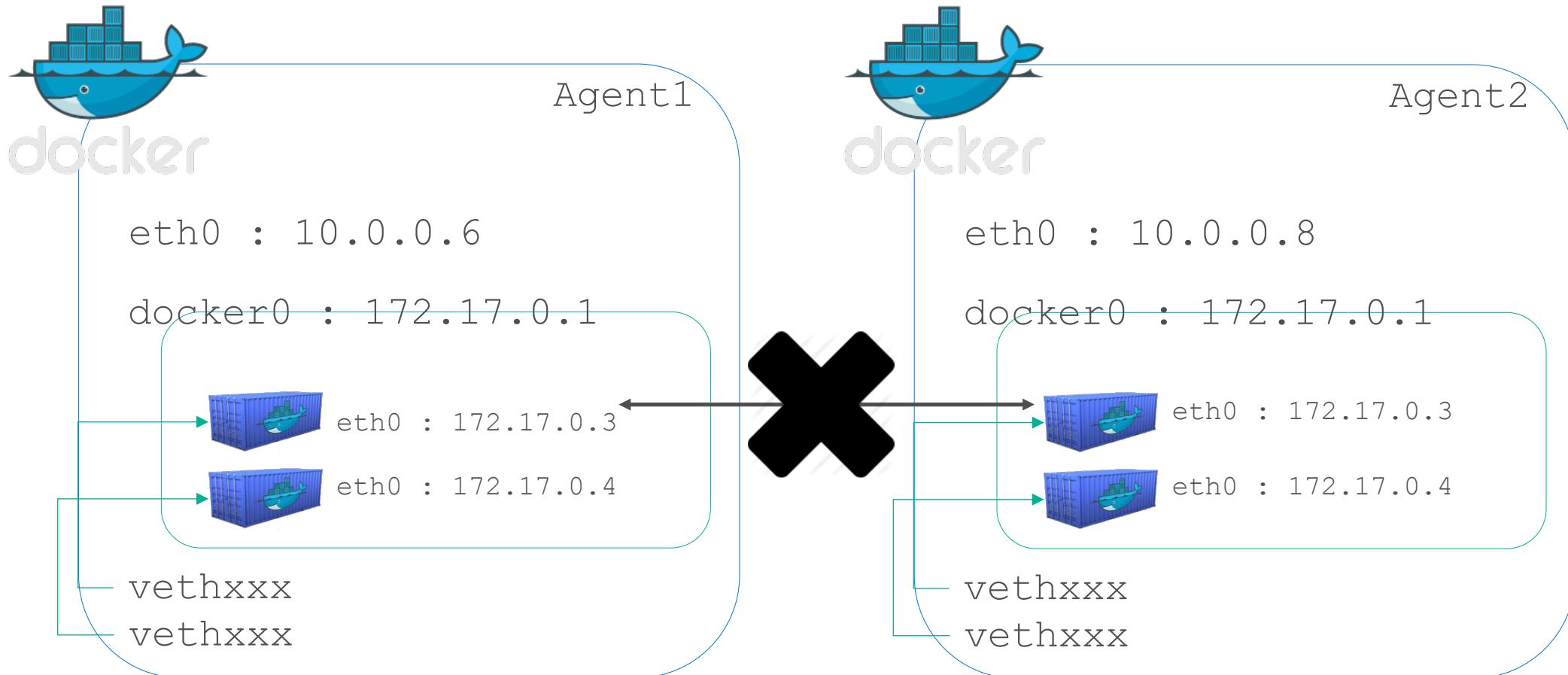
app=my-app

Networking

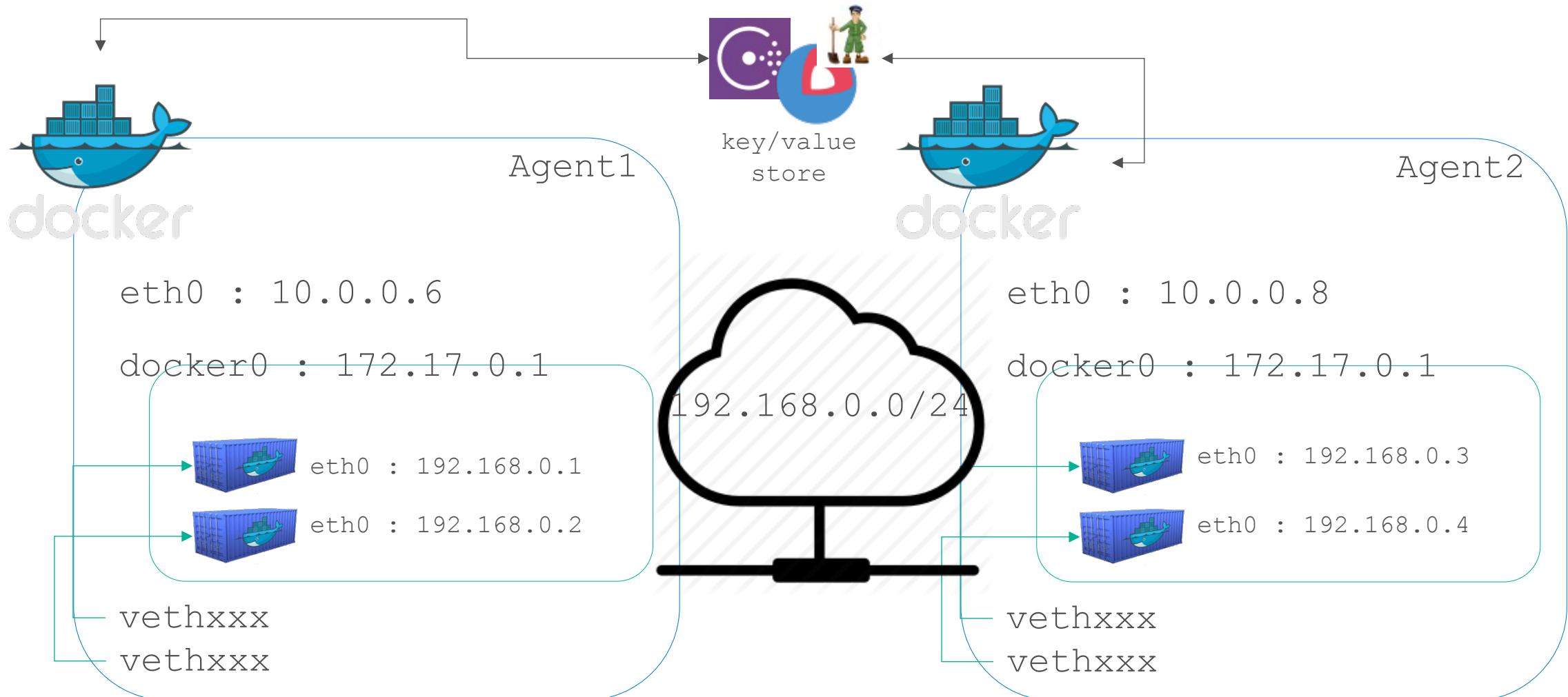
Networking in the Container World



Multi-Host Networking



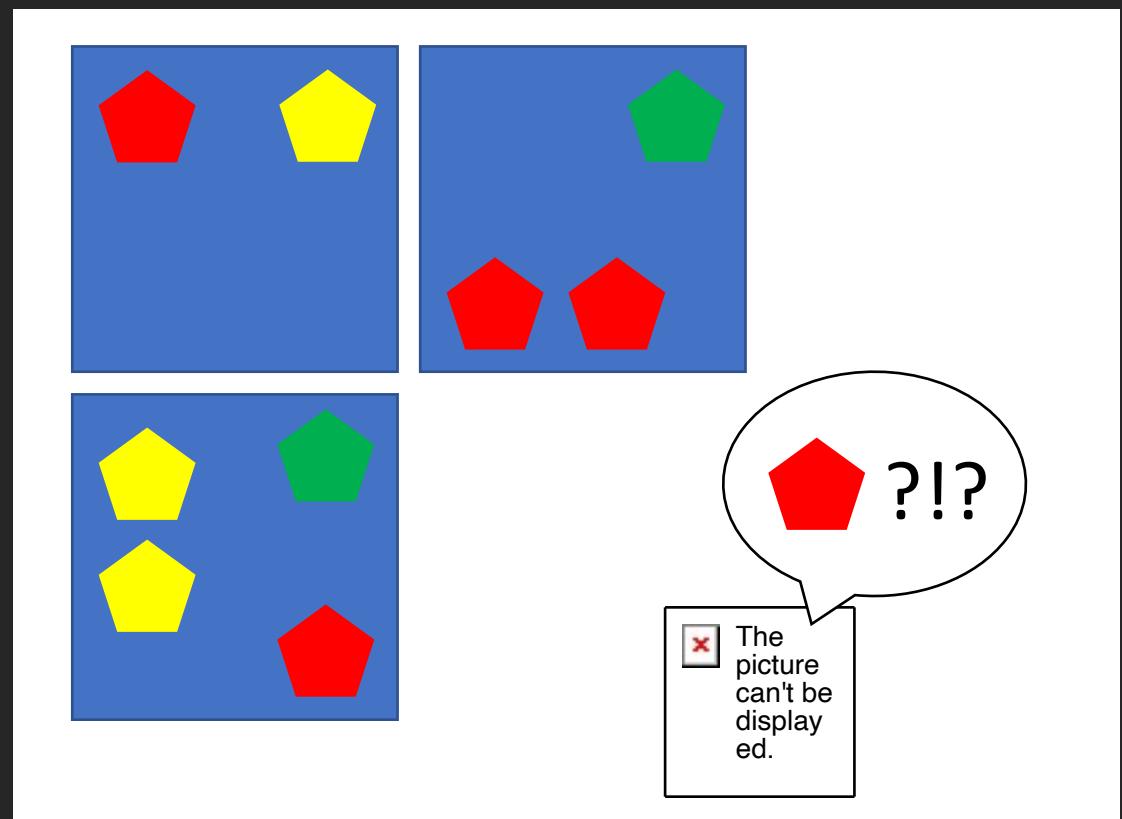
Multi-Host Networking



What is service discovery?

Problem: How service can find and communicate with another one running into the same cluster?

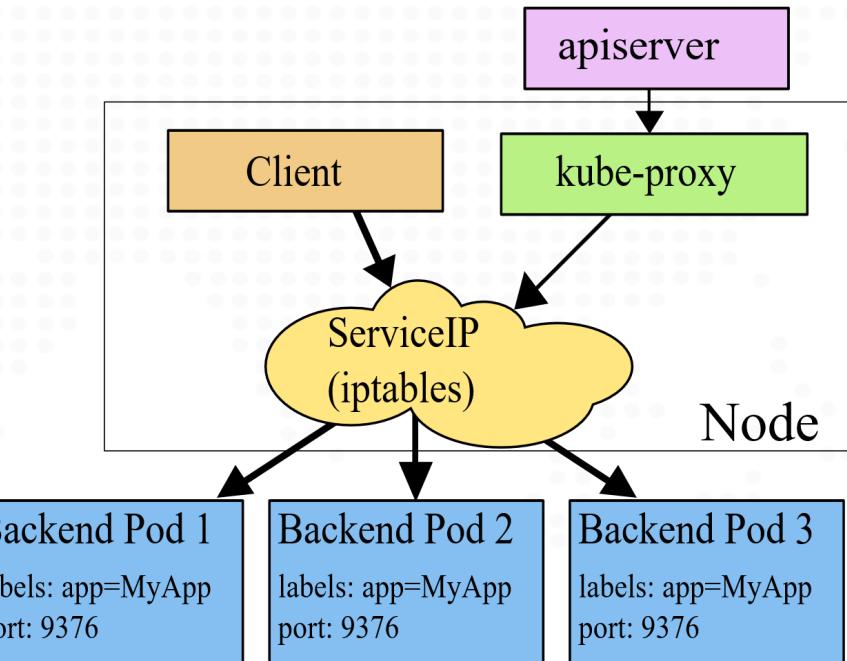
- Service Registry/Naming Service
- Service Announcement
- Lookup/Discovery
- Load Balancing



Networking in Kubernetes

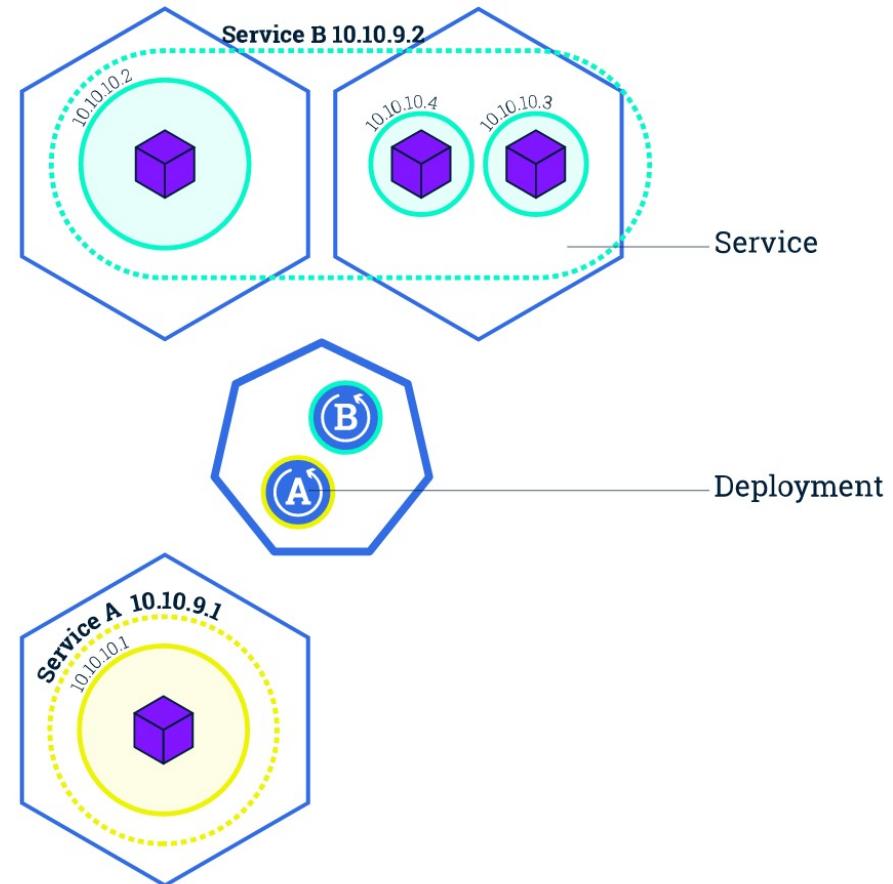
Kubernetes Networking model introduces 3 methods of communications:

- Pod-to-Pod communication directly by IP address. Kubernetes has a Pod-IP wide metric simplifying communication.
- Pod-to-Service Communication – Client traffic is directed to service virtual IP by iptables rules that are modified by the kube-proxy process (running on all hosts) and directed to the correct Pod.
- External-to-Internal Communication – external access is captured by an external load balancer which targets nodes in a cluster. The Pod-to-Service flow stays the same.



Service Discovery with Kubernetes

- Nodes, Deployments, Pods, Services
- Services
 - Expose the services
 - Works with platform Load Balancers to get public IPs
 - Internal load balancing between pods
- kube-dns (preferred)
 - Name service enabled in Kubernetes (Default in ACS)
- Environment Variables
 - Every Service is assigned environment variables with information of the service
 - REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11



Ingresses

- Typically, services and pods have IPs only routable by the cluster network
- All traffic that ends up at an edge router is either dropped or forwarded elsewhere
- Ingress is a collection of rules that allow inbound connections to reach the cluster services
- An Ingress controller is responsible for fulfilling the Ingress, usually with a loadbalancer, though it may also configure your edge router or additional frontends to help handle the traffic in an HA manner

Container Network Interface (CNI)

- CNI (a CNCF project) is a spec for plugins to configure Linux container network interfaces
- Wide range of support
 - Kubernetes, OpenShift, Cloud Foundry, Mesos, rkt, etc.
- Examples
 - Calico, Weave, Contiv, etc.
- Azure CNI plug-in integrates Azure VNET into kubernetes clusters
- Use acs-engine

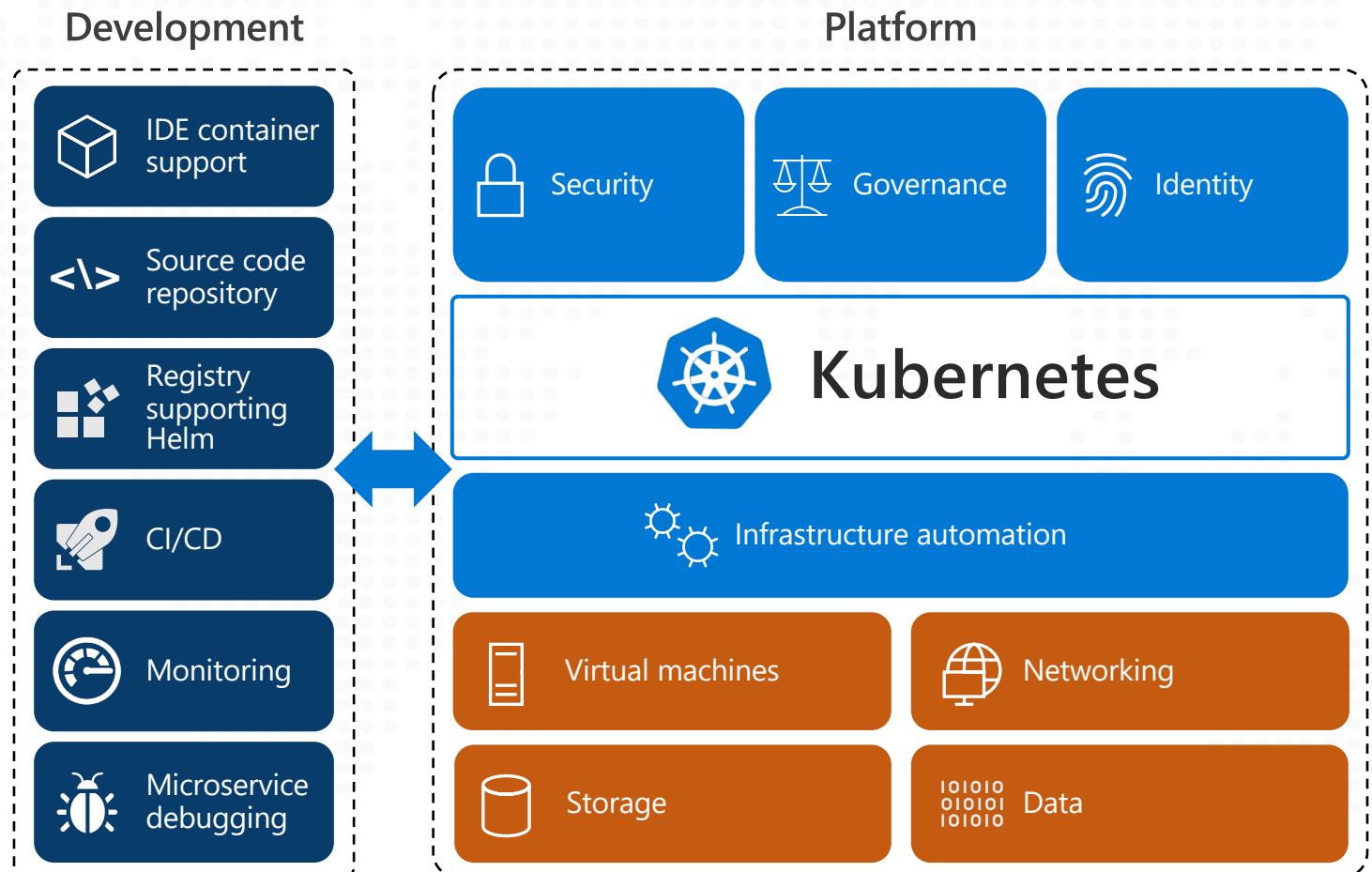
DevOps & CI / CD

Kubernetes on its own is **not enough**

Save time from infrastructure management and roll out updates faster without compromising security

Unlock the agility for containerized applications using:

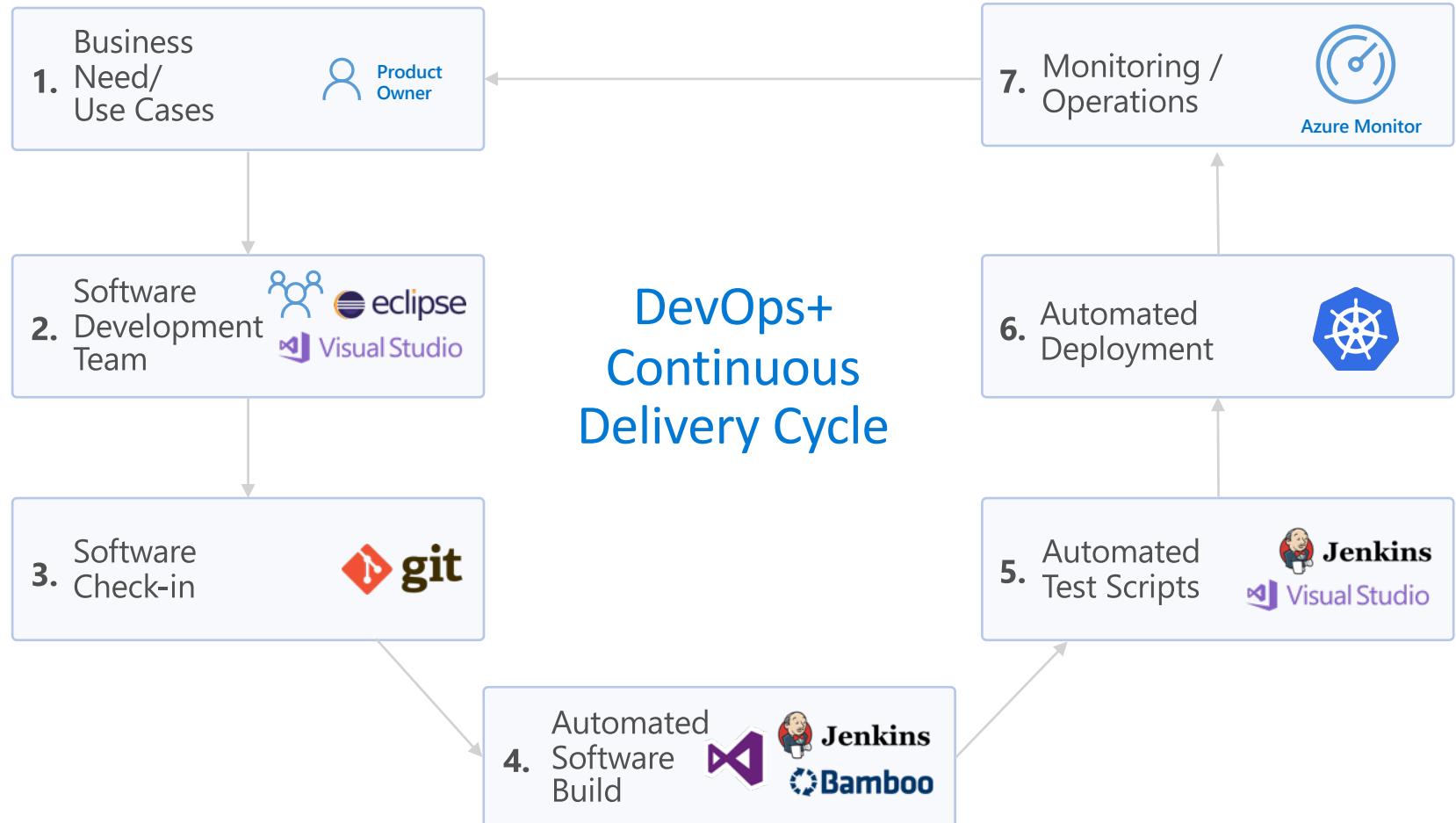
- Infrastructure automation that simplifies provisioning, patching, and upgrading
- Tools for containerized app development and CI/CD workflows
- Services that support security, governance, and identity and access management



DevOps Practices Arrive



Release
Automation Tools

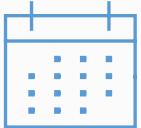


Why DevOps?

The benefits

46x

more frequent deployments



5x

lower change failure rate



440x

faster deployments



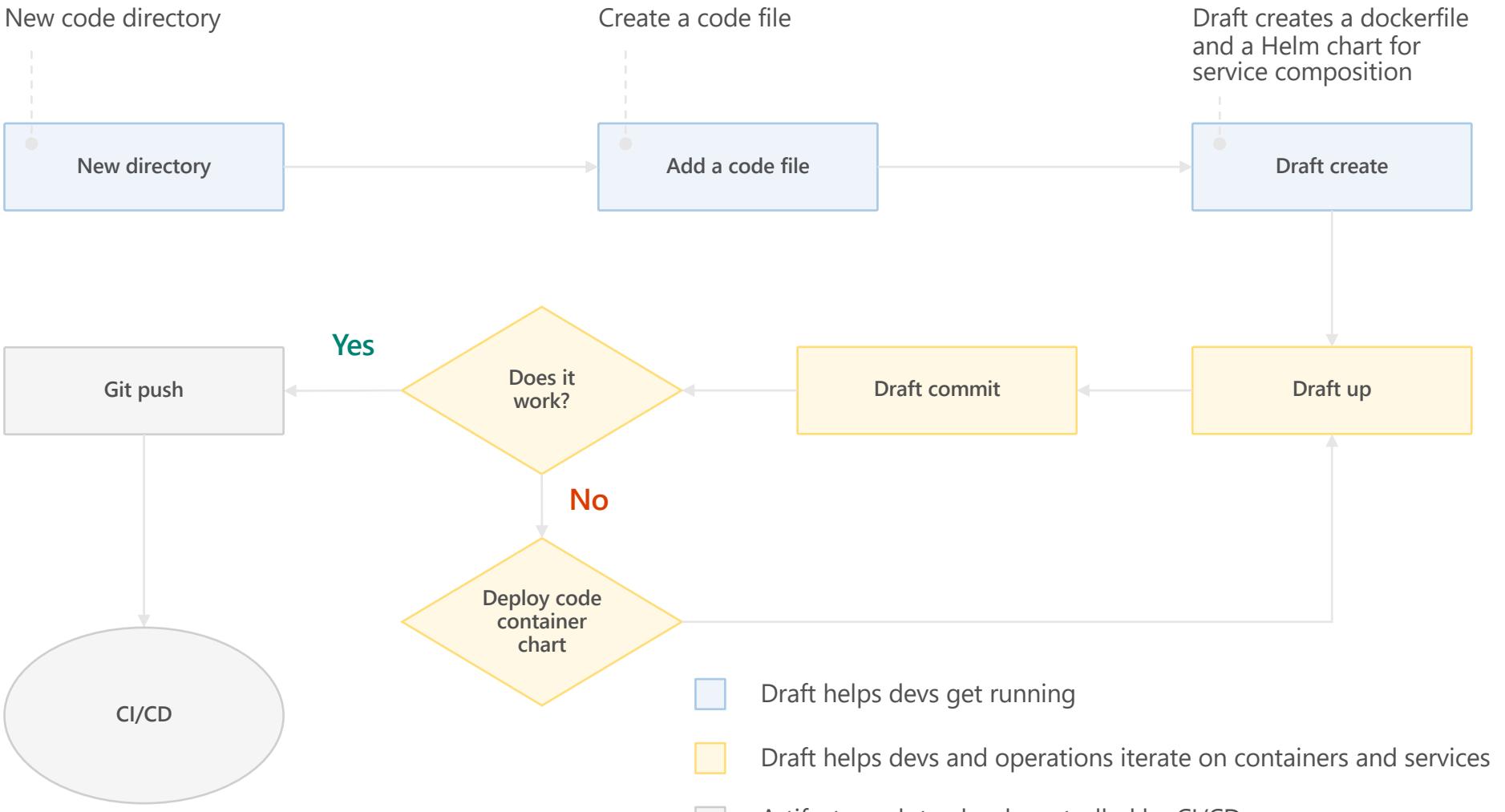
440x

shorter lead times



Release
Automation Tools

Release automation workflow



Release
Automation Tools

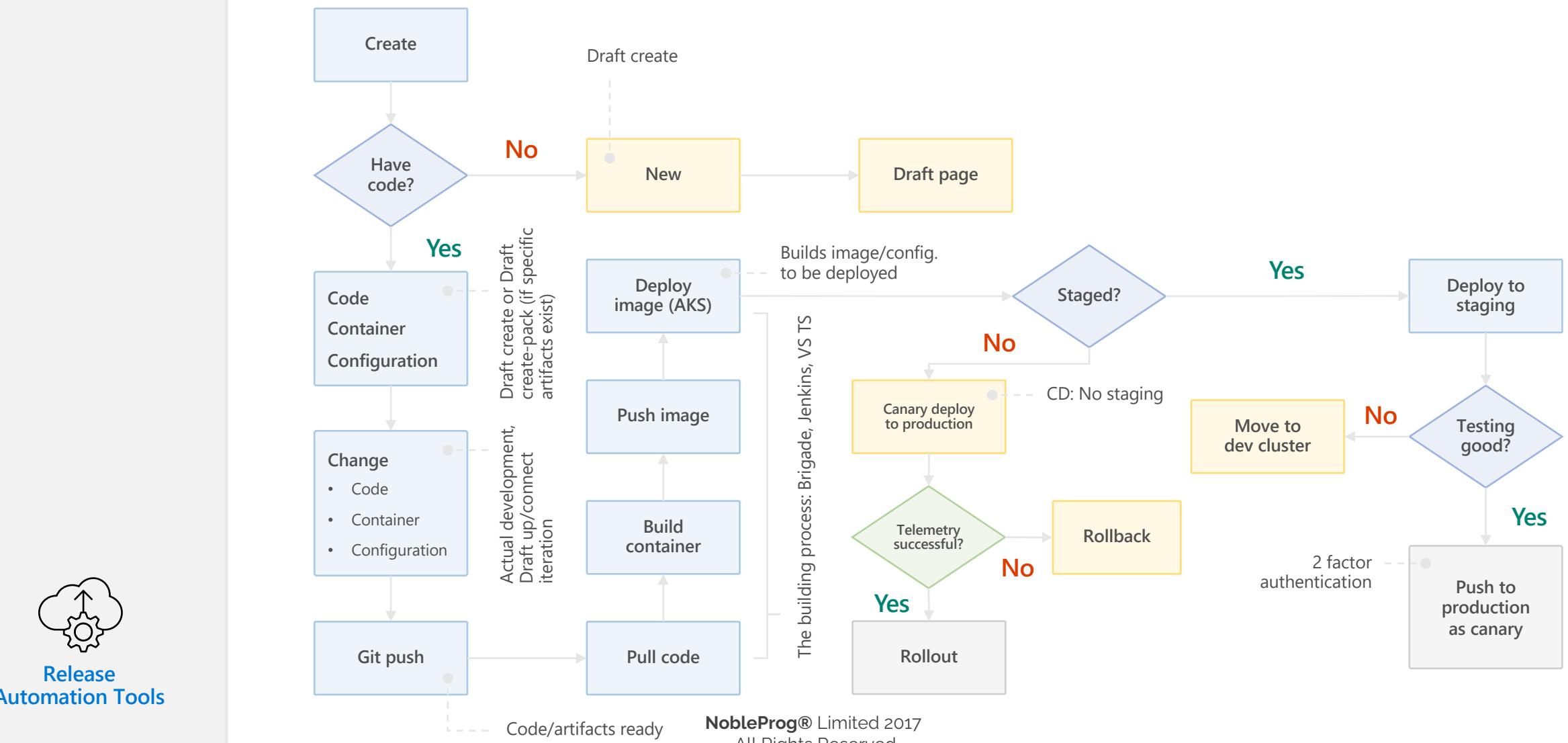
Release automation workflow

Once developers are up and running—or working on a service that is in a complex system—Draft ALSO helps devs ignore artifacts and focus on code



Release
Automation Tools

Release automation workflow



Release Automation Tools

Release automation tools

Simplifying the Kubernetes experience



Streamlined
Kubernetes
development



The package
manager for
Kubernetes



Event-driven
scripting for
Kubernetes

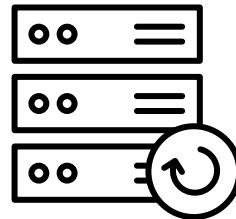
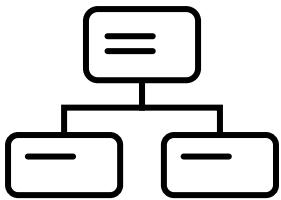


Release
Automation Tools



Helm

The best way to find, share, and use software built for Kubernetes



Manage complexity

Charts can describe complex apps; provide repeatable app installs, and serve as a single point of authority

Easy updates

Take the pain out of updates with in-place upgrades and custom hooks

Simple sharing

Charts are easy to version, share, and host on public or private servers

Rollbacks

Use `helm rollback` to roll back to an older version of a release with ease

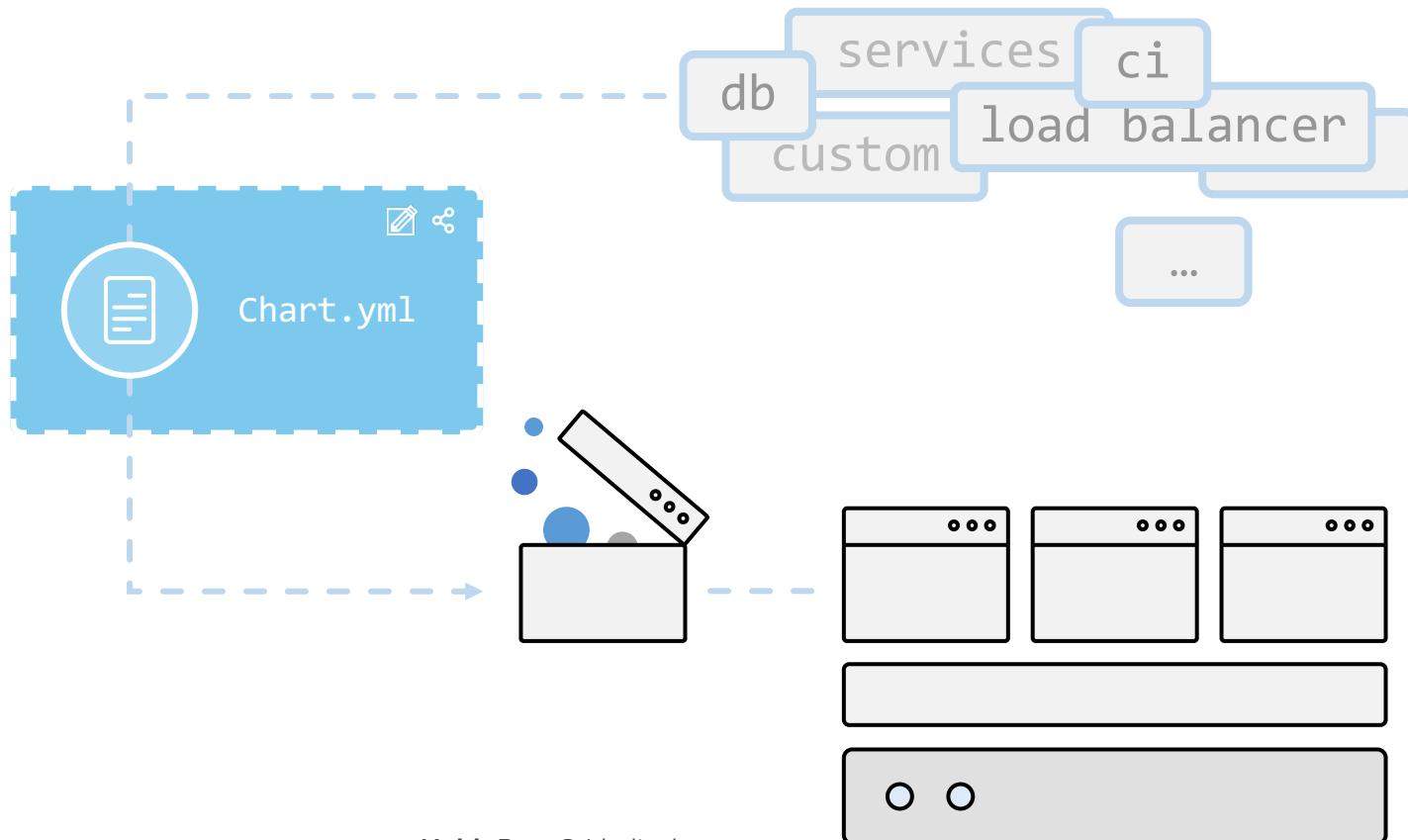


Release
Automation Tools



Helm

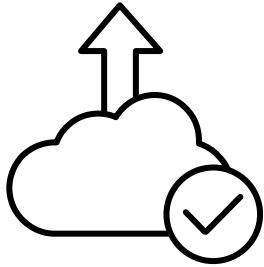
Helm Charts helps you define, install, and upgrade even the most complex Kubernetes application



Release
Automation Tools

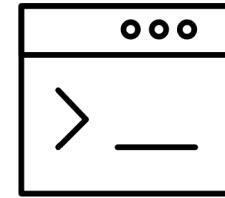
Draft

Simple app development and deployment – into any Kubernetes cluster



Simplified development

Using two simple commands, developers can now begin hacking on container-based applications without requiring Docker or even installing Kubernetes themselves



Language support

Draft detects which language your app is written in, and then uses packs to generate a Dockerfile and Helm Chart with the best practices for that language



Release
Automation Tools

[Get Started](#)

Draft

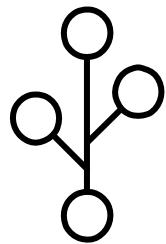
Draft in action



Release
Automation Tools

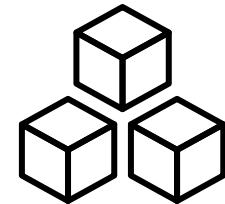
Brigade

Run scriptable, automated tasks in the cloud — as part of your Kubernetes cluster



Simple, powerful pipes

Each project gets a `brigade.js` config file, which is where you can write dynamic, interwoven pipelines and tasks for your Kubernetes cluster



Runs inside your cluster

By running Brigade as a service inside your Kubernetes cluster, you can harness the power of millions of available Docker images

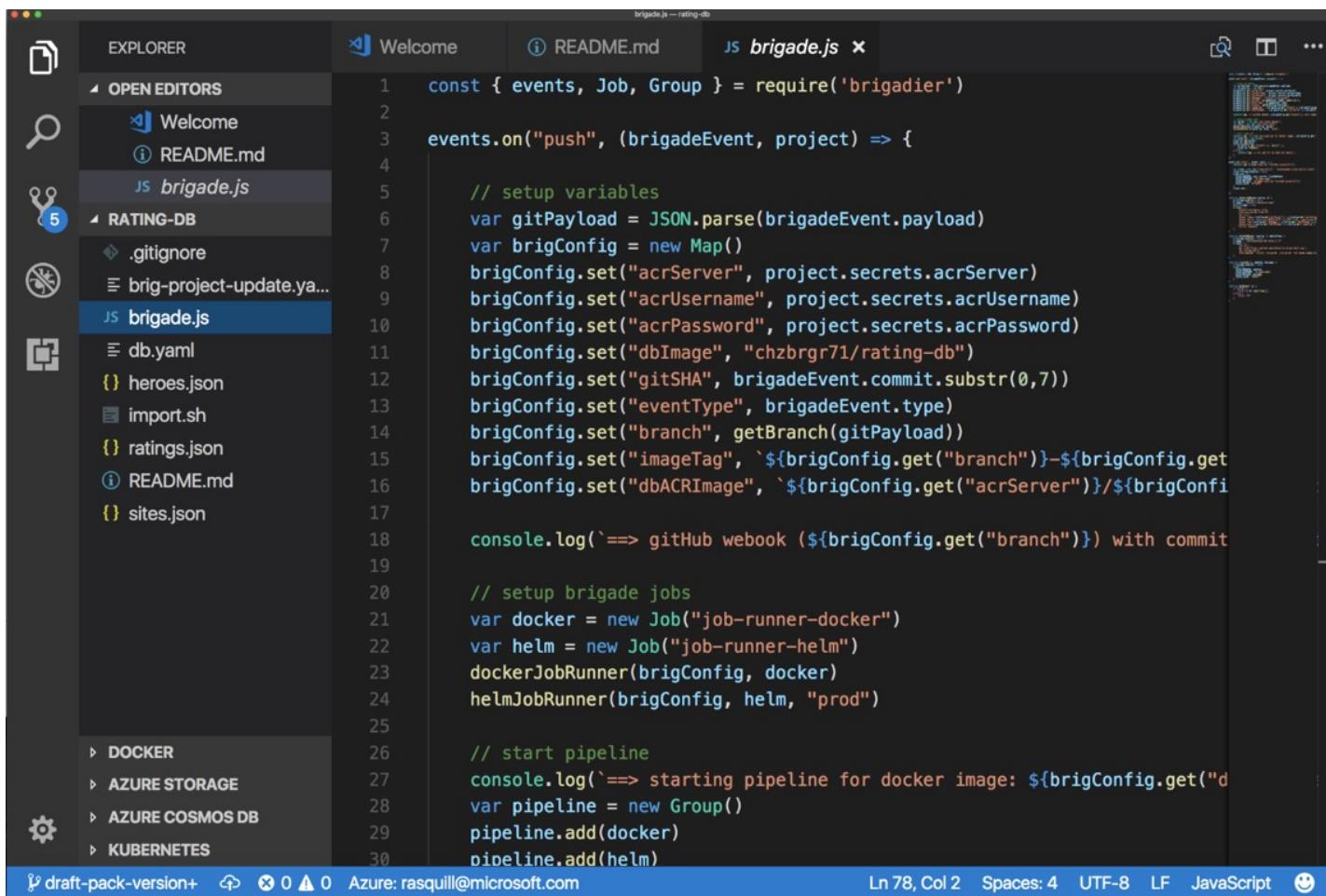


Release
Automation Tools



Brigade

Brigade in action



```
const { events, Job, Group } = require('brigadier')

events.on("push", (brigadeEvent, project) => {

  // setup variables
  var gitPayload = JSON.parse(brigadeEvent.payload)
  var brigConfig = new Map()
  brigConfig.set("acrServer", project.secrets.acrServer)
  brigConfig.set("acrUsername", project.secrets.acrUsername)
  brigConfig.set("acrPassword", project.secrets.acrPassword)
  brigConfig.set("dbImage", "chzbrgr71/rating-db")
  brigConfig.set("gitSHA", brigadeEvent.commit.substr(0,7))
  brigConfig.set("eventType", brigadeEvent.type)
  brigConfig.set("branch", getBranch(gitPayload))
  brigConfig.set("imageTag", `${brigConfig.get("branch")}-${brigConfig.get("dbACRImage")}`)
  brigConfig.set("dbACRImage", `${brigConfig.get("acrServer")}/${brigConfig.get("acrUsername")}`)

  console.log(`==> GitHub webhook ${brigConfig.get("branch")}) with commit ${gitPayload.sha}`)

  // setup brigade jobs
  var docker = new Job("job-runner-docker")
  var helm = new Job("job-runner-helm")
  dockerJobRunner(brigConfig, docker)
  helmJobRunner(brigConfig, helm, "prod")

  // start pipeline
  console.log(`==> starting pipeline for docker image: ${brigConfig.get("dbImage")}`)
  var pipeline = new Group()
  pipeline.add(docker)
  pipeline.add(helm)
})

function getBranch(payload) {
  if (payload.ref === "refs/heads/main") {
    return "main"
  }
  if (payload.ref === "refs/heads/master") {
    return "master"
  }
  return payload.ref.replace("refs/heads/", "")
}
```



Release
Automation Tools

Brigade UI

Dashboards for Brigade pipelines

Build #01C0HX5S1GH7A0TBJA2EYG7R1

Started at 2017-12-23T07:31:49Z | Finished at 2017-12-23T07:36:44Z

Passed

1 job ran inside this build:

```
brigade-cli : build started a build via commit #master
```

started at 12/22/2017 @ 11:31PM -0800

build

Image: node:8
ID: build-1514014319426-master
Log output:

```
yarn install v1.3.2
[1/5] Validating package.json...
[2/5] Resolving packages...
[3/5] Fetching packages...
info "fsevents@1.1.3: The platform "linux" is incompatible with this module.
info "fsevents@1.1.3" is an optional dependency and failed compatibility check. Excluding it from installation.
```

Builds dashboard

Azure/kashti

https://github.com/Azure/kashti.git
sidecar:
Namespace: default

Job Status	Task	Branch	ID	Timestamp	Duration	Details
Passed	brigade-cli	master	#01c0hx5s1gh7a0tzbja2eyg7r1	succeeded a month ago.	Ran for 295 seconds.	Details >
Failed	brigade-cli	master	#01c0hx0yxqa15t7fdcqxe7errw	Failed a month ago.	Ran for 268 seconds.	
Failed	brigade-cli	master	#01c0hmmh4h41x0mq7z9sr794cr	Failed a month ago.	Ran for 11 seconds.	
Failed	brigade-cli	master	#01c0hmj9cfxr6eb6cq818yac5n	Failed a month ago.	Ran for 35 seconds.	
Failed	brigade-cli	master	#01c0hmawnqw9332yg1ad4jqgah	Failed a month ago.	Ran for 68 seconds.	
Failed	brigade-cli	master	#01c0hm8ekarvbybpd93d475mt	Failed a month ago.	Ran for 49 seconds.	
Passed	brigade-cli	master	#01c0hm249jgz2jcje4wj2cf0sn	Succeeded a month ago.	Ran for 106 seconds.	
Failed	brigade-cli	master	#01c0hkxb91dkxzshp58tx18qcf	Failed a month ago.	Ran for 70 seconds.	
Failed	brigade-cli	master	#01c0hkwhmegs7ywvjjzbnngchb	Failed a month ago.	Ran for 8 seconds.	

Events log

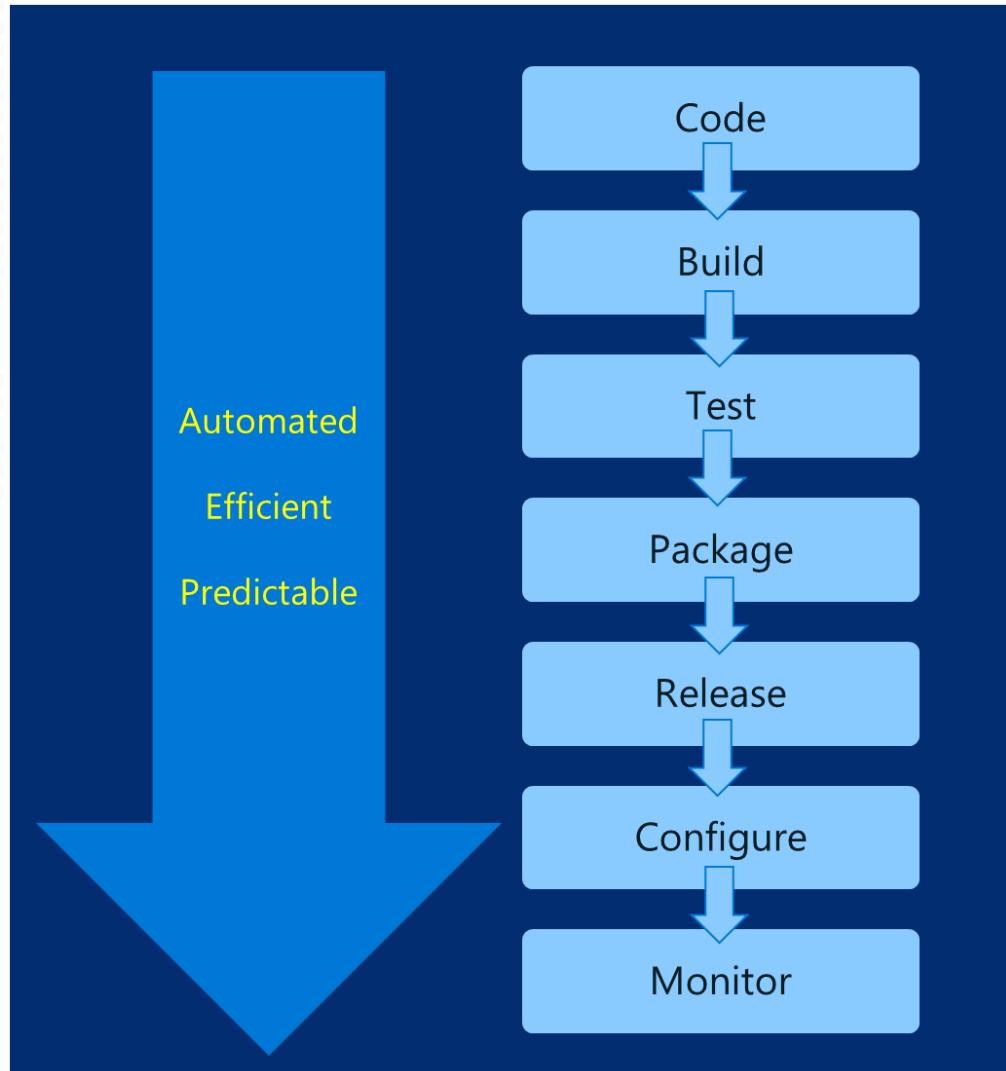


Release
Automation Tools

DevOps Practices Arrive

- Developers: Test, Build, Code, Plan
- Operations: Monitor, Release, Deploy, Operate
- DevOps Features
 - Speed of application delivery/updates
 - Faster time to value
 - Repeatable/consistent
 - Automated testing
 - Traceability of process
 - Applying developer patterns to infrastructure
 - Infrastructure as code
 - Ops teams embracing source control
 - Centralized monitoring, logging, debugging
- CI / CD – key aspect of quality DevOps

DevOps – Why you care



Deploy 200 times more frequently

Go from code check-in to production
2,555 times faster

Recover from failure 24 times faster

Spent 50% less time remediating
security challenges

Spent 22% less time on unplanned
work

2.2 times more likely to believe their
places a great place to work

Common Toolsets

- Jenkins
- Visual Studio Team Services
- Spinnaker and Netflix OSS
- Travis
- TeamCity
- CircleCI
- Additional utilities: code quality scanning, security, collaboration, etc.

Enter Helm: Kubernetes Package Manager



Birth of Helm

On October 15th, 2015

Hackathon project at company offsite

Could we take the ideas behind npm and Homebrew and build something for deploying apps into Kubernetes?

Installation tool for Deis Workflow

Announced at the first KubeCon in San Francisco 2015

Helm Charts

Application definition

Consists of:

- Metadata
- Kubernetes resource definitions
- Configuration
- Documentation

Stored in chart repository

- Any HTTP server that can house YAML/tar files (Azure, GitHub pages, etc.)
- Public repo with community supported charts (eg – Jenkins, Mongo, etc.)

Helm (CLI) + Tiller (server side)

Release: Instance of chart + values -> Kubernetes

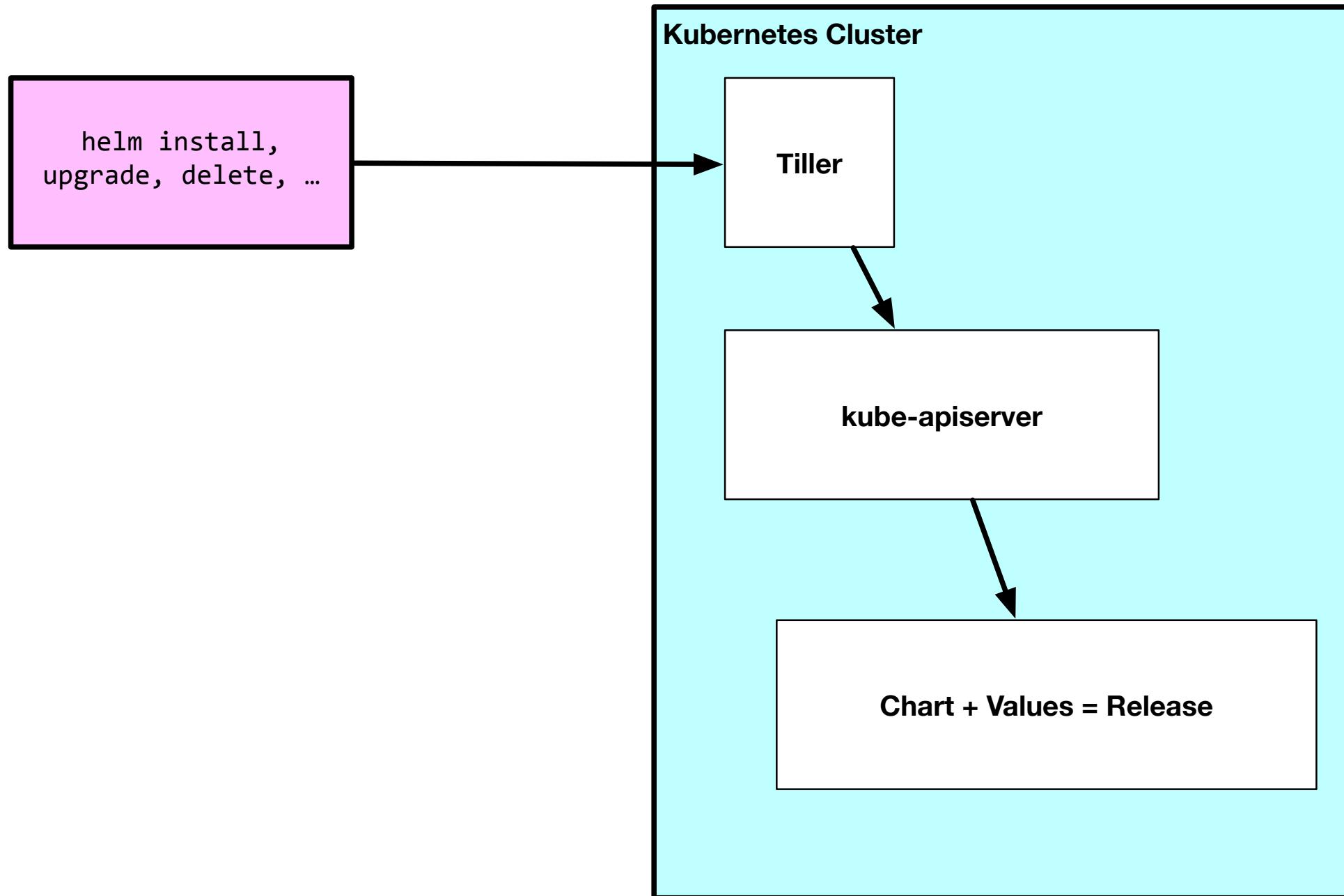


Chart structure

- Charts have structure
 - Set of conventions, including file and directory names
 - Charts can be packaged into tarballs for distribution

Chart structure

- Layout
 - Helm expects a strict chart structure

```
wordpress/
  Chart.yaml          # A YAML file containing information about the chart
  LICENSE            # OPTIONAL: A plain text file containing the license for the chart
  README.md          # OPTIONAL: A human-readable README file
  values.yaml        # The default configuration values for this chart
  charts/             # OPTIONAL: A directory containing any charts upon which this chart depends.
  templates/          # OPTIONAL: A directory of templates that, when combined with values,
                      # will generate valid Kubernetes manifest files.
  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

Chart.yaml

```
name: The name of the chart (required)
version: A SemVer 2 version (required)
description: A single-sentence description of this project (optional)
keywords:
  - A list of keywords about this project (optional)
home: The URL of this project's home page (optional)
sources:
  - A list of URLs to source code for this project (optional)
maintainers: # (optional)
  - name: The maintainer's name (required for each maintainer)
    email: The maintainer's email (optional for each maintainer)
engine: gotpl # The name of the template engine (optional, defaults to gotpl)
icon: A URL to an SVG or PNG image to be used as an icon (optional).
```

Helm values.yaml

- The knobs and dials:
 - A values.yaml file provided with the chart that contains **default** values
 - Use -f to provide your own values overrides
 - Use --set to override individual values

Helm Templates

- Built on Go's template language w/addition of 50 or so add-on template functions
- Almost anything goes! ;-)
- Also useful in generating random values (e.g. passwords)
 - Provides flow control (if/else, with, range, etc)
 - Named templates (partials)

Draft: Streamlined Kubernetes Development

What is draft?

Developer workstations cannot mimic production

As devs hack on code prior to committing to version control

Draft targets the "inner loop" of a developer's workflow

Works with other Deis tools such as Helm and Workflow

Developer Workstation

Dev (Inner Loop)

Production / Staging Cluster



Storage

Volumes

- On-cluster representation of physical storage
- Lifetime of Volume object itself tied to the pod that encloses it—when the pod is gone, the Volume is gone
- Can persist data
 - PersistentVolume: a disk
 - PersistentVolumeClaim: dynamically provision a disk
 - StorageClass: cluster consumers can request storage without needing intimate knowledge about underlying storage volume sources

Other neat volume-y things

- Mount a Secret or ConfigMap as a volume in a pod
 - Useful for passing around TLS certificates
- Mount a git repo as a volume

Supported Volume Types

emptyDir	fc (fibre channel)
hostPath	flocker
gcePersistentDisk	glusterfs
awsElasticBlockStore	rbd
nfs	Cephfs
iscsi	Quobyte
azureFileVolume	PortworxVolume
azureDisk	ScaleIO
vsphereVolume	

Scaling

Autoscaling Strategies

Infrastructure Level

Increase/decrease the number of nodes running in the cluster

Azure infrastructure function

VM Scalesets (DCOS/Docker), Availability Sets (Kubernetes)

No auto-scaling in ACS today

Application Level

Increase/decrease the number of containers for a given service

Handled by the orchestrator or the application itself

Kubernetes Horizontal Pod Autoscaling

Application Health and Readiness

- Kubelets on workers use liveness and readiness probes to know when to restart a container or start directing traffic
- Liveness check: when to restart
- Readiness check: when to forward Service traffic to a pod

Monitoring: What indicators matter in Kubernetes

- Cluster health: total number of nodes, pods, etc
- Node health: available compute resources, health
- Application health
- Tooling Examples
 - Datadog
 - Prometheus
 - Cloud-specific: OMS (Azure), CloudWatch (AWS)

Logging

- Pod logs: applications must log to standard out!
- Cluster-wide **event logs**
- **Logging forwarder solutions**
 - fluentd
 - logstash
- Log indexers
 - OMS Log Analytics
 - Splunk
 - ELK stack

ConfigMaps and Secrets

- Flexible configuration model for Kubernetes
- Both Secrets and ConfigMaps can be used as ENV vars or files
- Secrets
 - Use Secrets for things which are actually secret like API keys, credentials, etc
 - Secret values are base64 encoded and automatically decoded for pods
- ConfigMaps
 - ConfigMaps for configuration that doesn't have sensitive information. However, there are pros/cons with each
 - ConfigMaps designed to more conveniently support working with strings

Daemonsets

- Ensures that all (or some) nodes run a copy of a pod
- Doesn't care if a node is marked as unschedulable
- Can make pods even if the scheduler is not running

StatefulSets

- Used for workloads that require consistent, persistent hostnames
e.g. etcd-01, etcd-02
- One PersistentVolume storage per pod
- StatefulSet example - zookeeper
- StatefulSet example - cockroachdb

Jobs

- Creates one or more pods and ensures that a specified number of them successfully terminate
- 3 types of jobs
 - Non-parallel Jobs - e.g. single pod done when exit is successful
 - Parallel Jobs with a fixed completion count - e.g. one successful pod for each value in the range 1 to .spec.completions
 - Parallel Jobs with a work queue that coordinate with each other and terminate when one pod exits successfully

Role Based Access Control

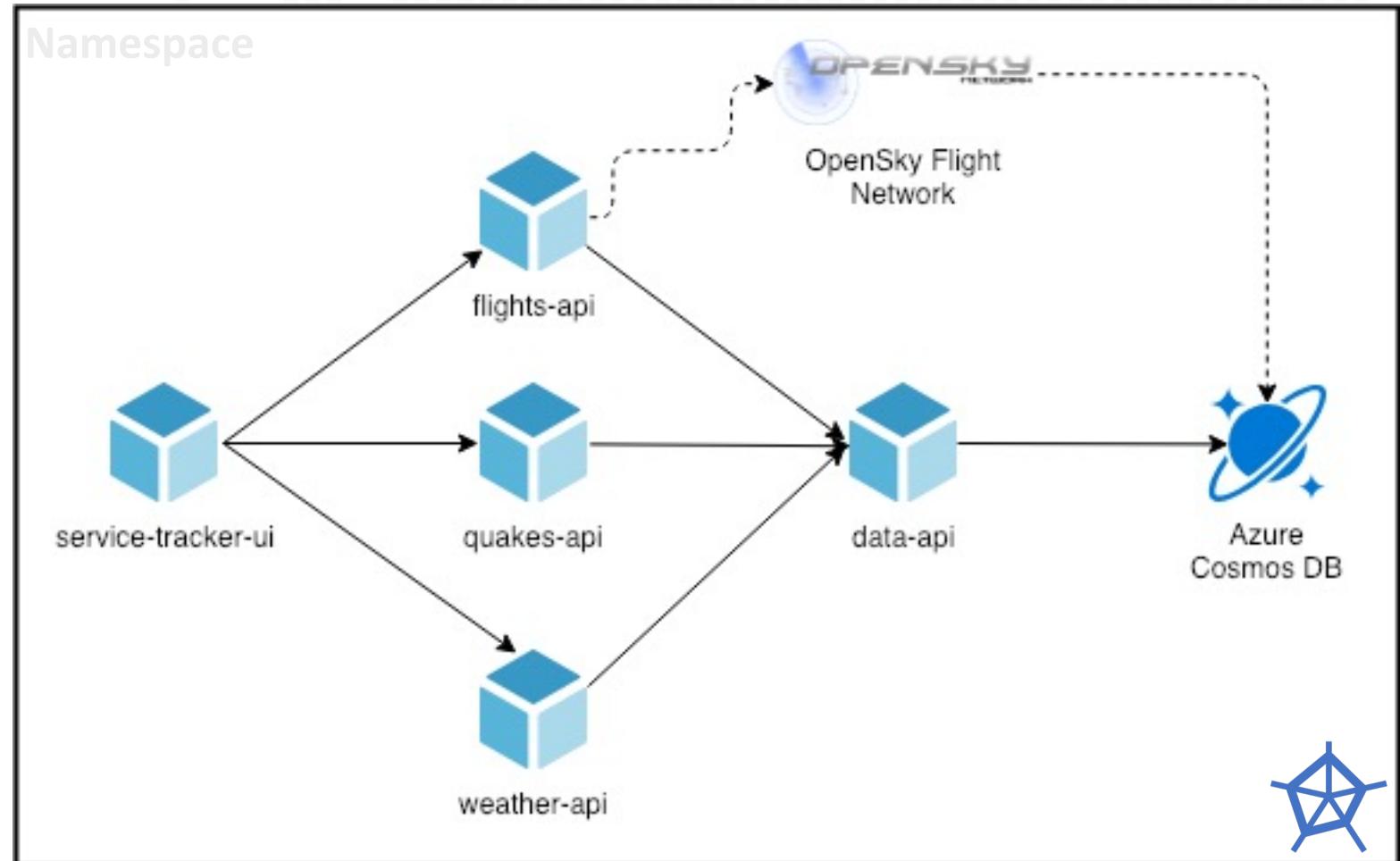
- Allows for dynamic policies against k8s API for authZ
- In version 1.6, this is a beta feature
- Can create both "Roles" and "ClusterRoles"
- Can configure api-server and kubectl to use Azure AD for access
- To grant permissions:
 - Use "RoleBindings" for within namespaces
 - Use "ClusterRoleBindings" for cluster-wide access
 - Can contain users, groups, service accounts
 - Control access to specific resources or API verbs

```
rules:  
- apiGroups: [ "" ]  
  resources: [ "pods" ]  
  verbs: [ "get", "list", "watch" ]  
- apiGroups: [ "batch", "extensions" ]  
  resources: [ "jobs" ]  
  verbs: [ "get", "list", "watch", "create", "update", "patch", "delete" ]
```

DESPLIEGUE DE APLICACIONES (1)

PRACTICA 2

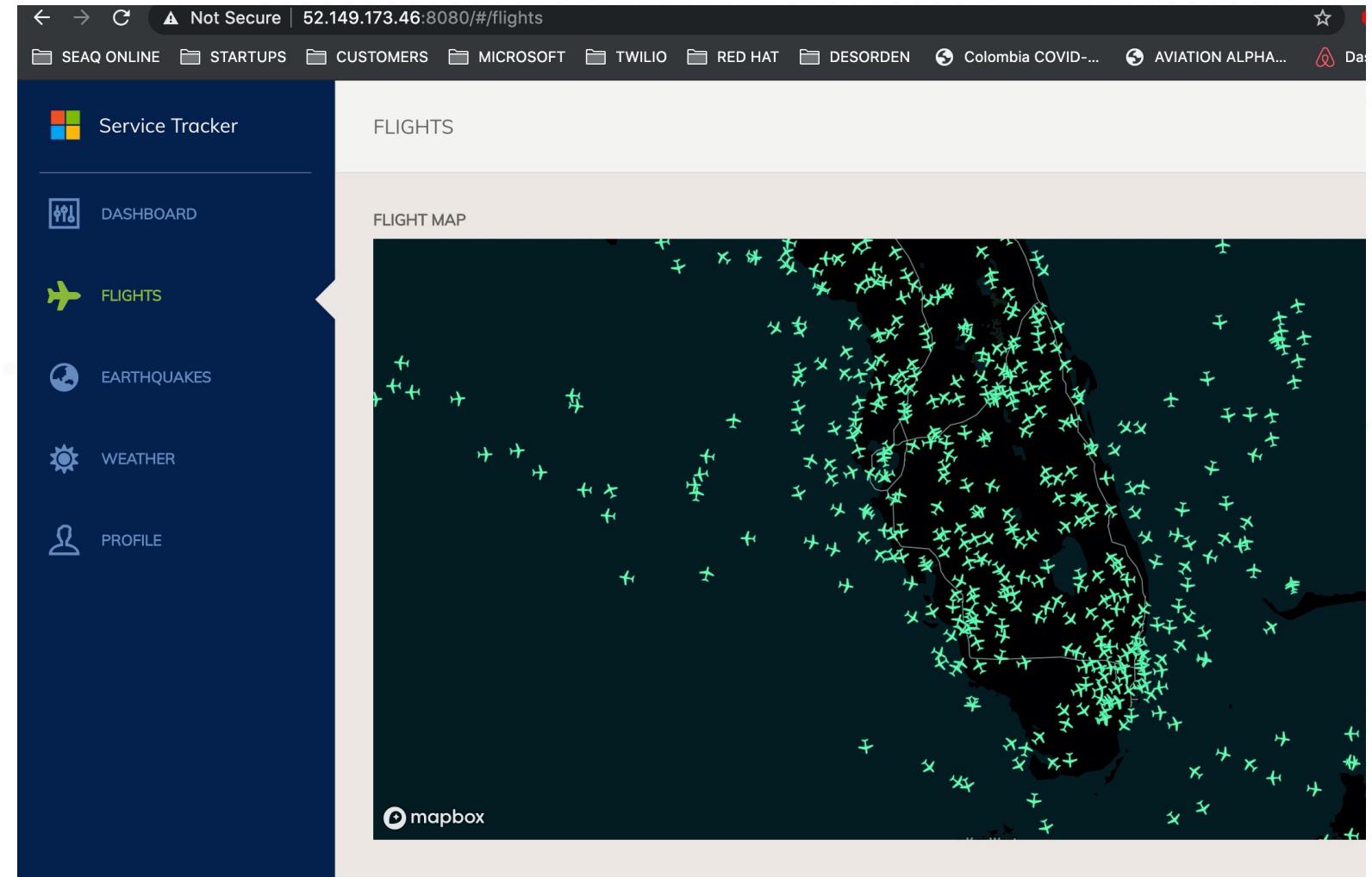
- Creación del Azure Container Registry
- Parametrización Aplicaciones
- Creación del App Insights
- Creación del Backend de Datos
- Creación de Docker Containers
- Debugging/Logging



DESPLIEGUE DE APLICACIONES (2)

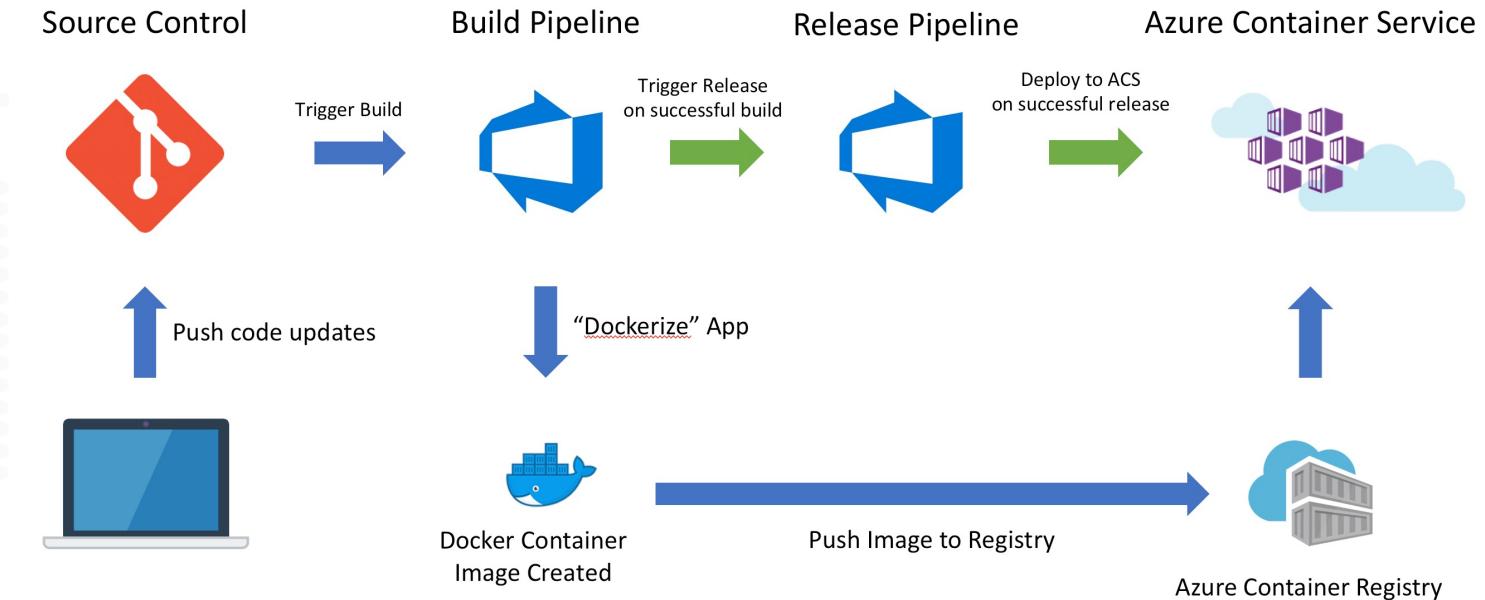
PRACTICA 3

- Parametrización despliegues
- Construcción application charts
- Despliegue application charts
- Inicialización de aplicaciones
- Acceso y navegación aplicaciones
- Debugging/Logging

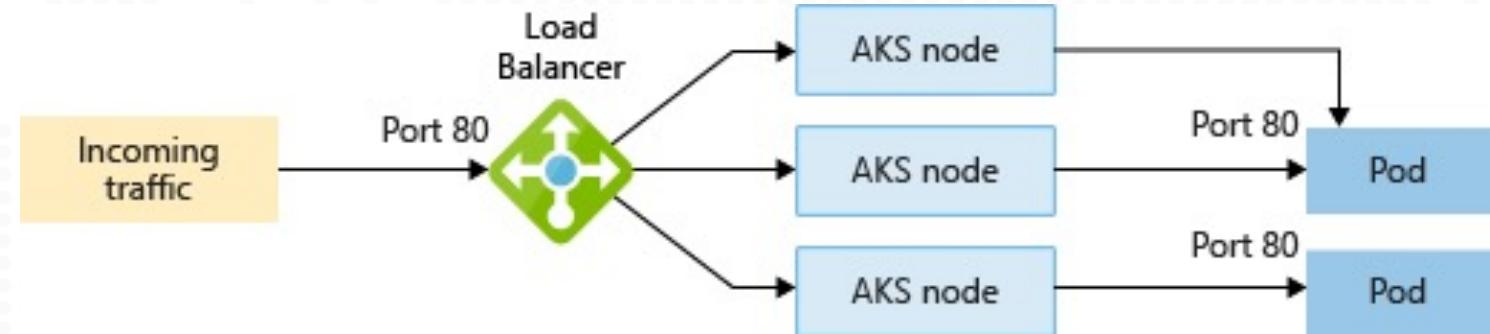


AUTOMATIZACIÓN CI/CD – Azure DevOps

- Creación Proyecto Azure DevOps
- Importar repositorio
- Crear Pipeline de compilación CI
- Crear Pipeline de despliegue CD
- Ejecutar una compilación y despliegue de pruebas
- Editar el Código y desplegar cambios en línea
- Debugging/Logging



- Parametrización networking
- Despliegue de NGINX
- Generación de certificados TLS
- Creación de ruta de acceso
- Pruebas y navegación
- Debugging/Logging



NobleProg

Kubernetes intro & advanced

The World's Local Training Provider

NobleProg® Limited 2017
All Rights Reserved