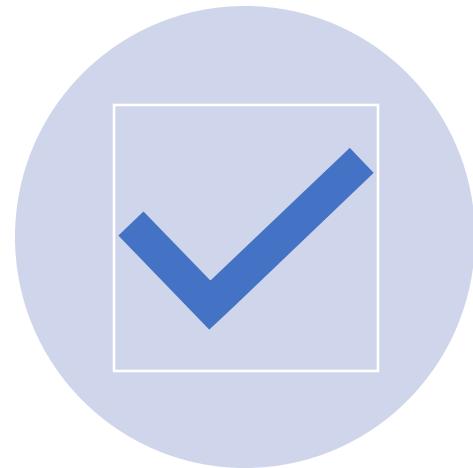


CONTAINERS



WEEK 1

Agenda

Benefits of Containerized Applications

Introduction to Containers

What is Docker

Docker Architecture

Docker Engine

Run Docker Containers

Create Docker Images

Publish Docker Images with Amazon ECR

Networks in Docker

Docker Storage

Docker Concepts in Depth

Build and Push to Amazon ECR from Cloud9

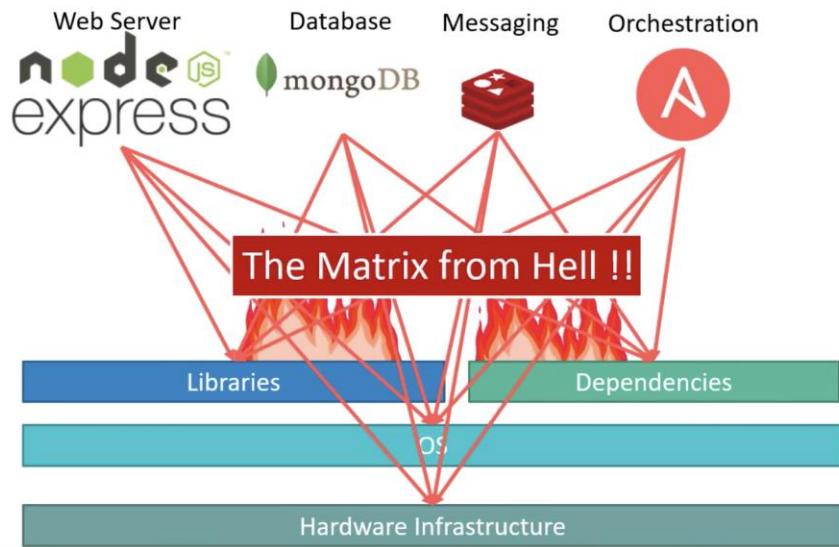
Build and Push to Amazon ECR with GitHub Actions

*Tell me and I forget.
Teach me and I remember.
Involve me and I learn.*

Misattributed to Benjamin Franklin

(Probably inspired by Chinese Confucian philosopher Xunzi)

Why Containers?

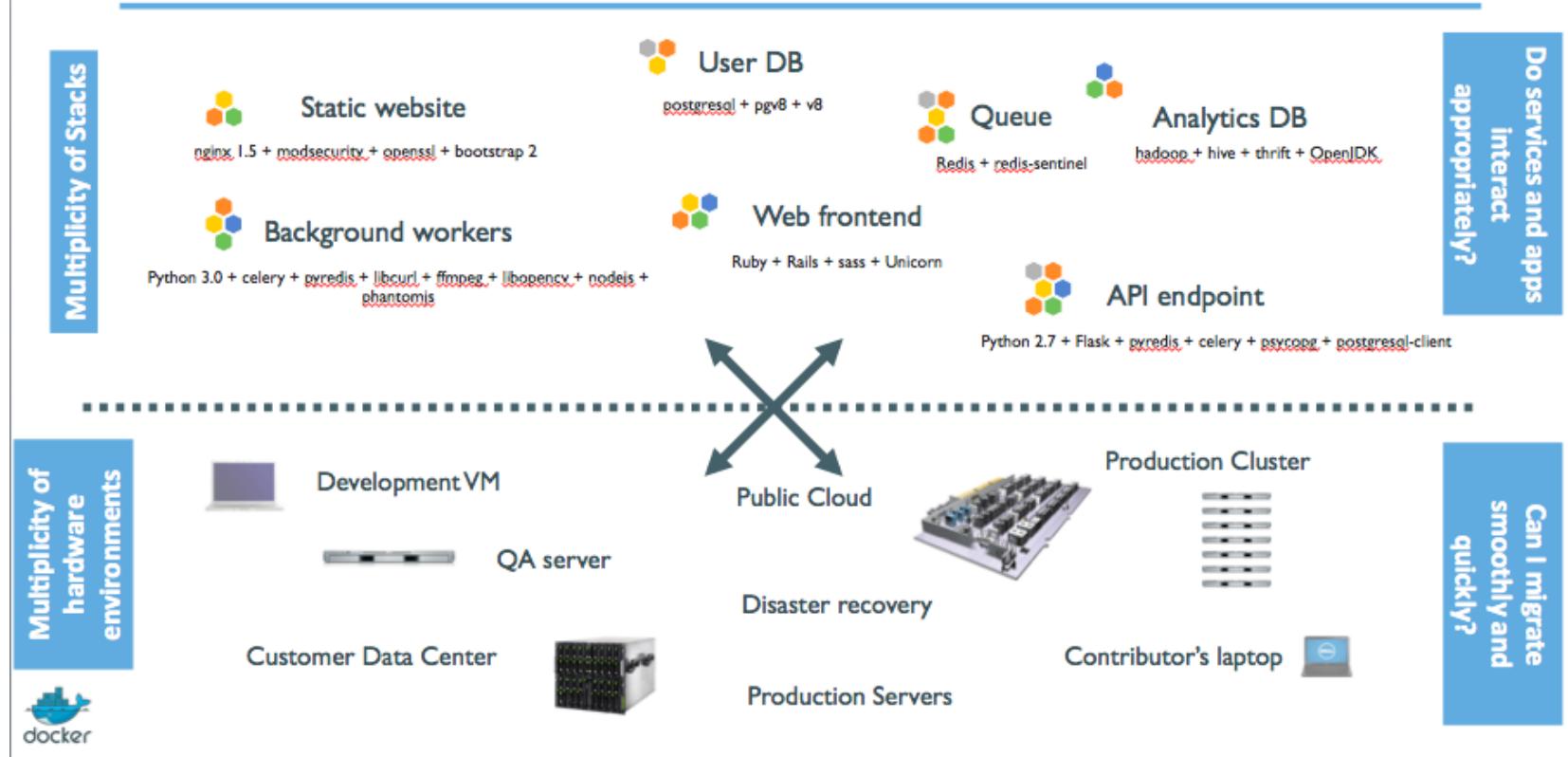


- The software industry has changed
- Before:
 - monolithic applications
 - long development cycles
 - single environment
 - slowly scaling up
- Now:
 - decoupled services
 - fast, iterative improvements
 - multiple environments
 - quickly scaling out

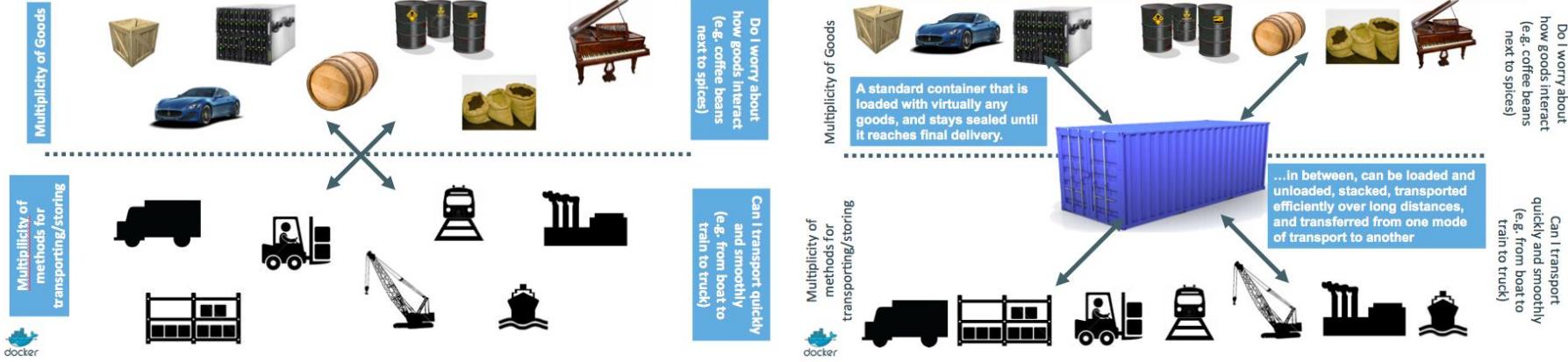
Deployment becomes very complex

- Many different stacks:
 - languages
 - frameworks
 - databases
- Many different targets:
 - individual development environments
 - pre-production, QA, staging...
 - production: on prem, cloud, hybrid

The Deployment Problem



The parallel with the shipping industry

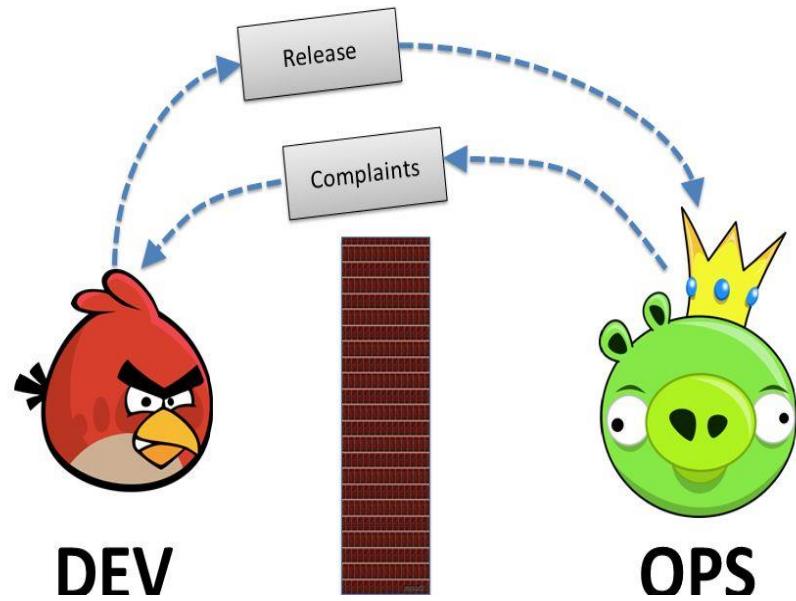


Escape dependency hell

- Write installation instructions into an INSTALL.txt file
- Using this file, write an install.sh script that works *for you*
- Turn this file into a Dockerfile, test it on your machine
- If the Dockerfile builds on your machine, it will build *anywhere*
- Rejoice as you escape dependency hell and "works on my machine"
- Never again "worked in dev - ops problem now!"

Devs vs Ops, before Docker

- Drop a tarball (or a commit hash) with instructions.
- Dev environment very different from production.
- Ops don't always have a dev environment themselves ...
- ... and when they do, it can differ from the devs'.
- Ops have to sort out differences and make it work ...
- ... or bounce it back to devs.
- Shipping code causes frictions and delays.



Devs vs Ops, after Docker

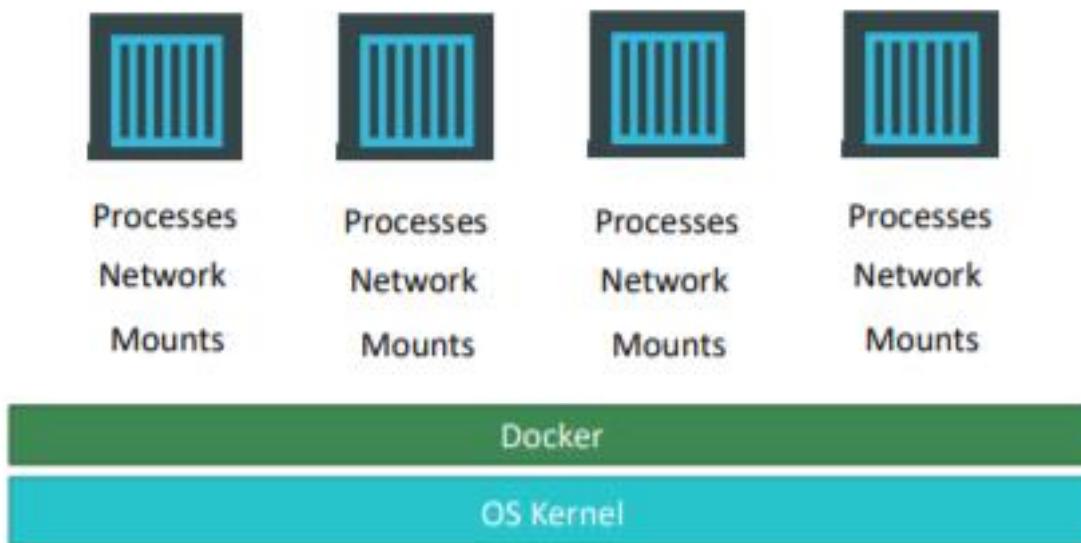
- Drop a container image or a Compose file.
- Ops can always run that container image.
- Ops can always run that Compose file.
- Ops still have to adapt to prod environment, but at least they have a reference point.
- Ops have tools allowing to use the same image in dev and prod.
- Devs can be empowered to make releases themselves more easily.

What are Containers

Containers

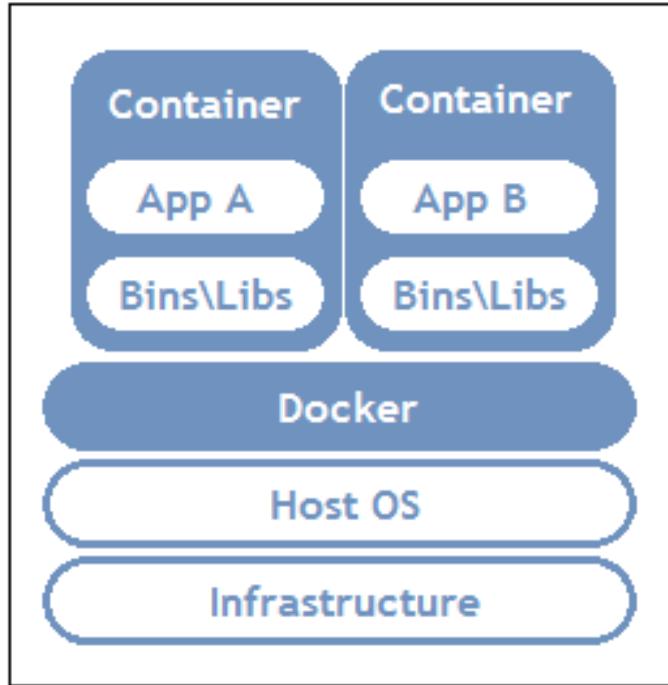
- LXC (Linux Containers) is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a control host using a single Linux kernel.
- The Linux kernel provides the cgroups functionality that allows limitation and prioritization of resources (CPU, memory, block I/O, network, etc.) without the need for starting any virtual machines, and namespace isolation functionality that allows complete isolation of an applications' view of the operating environment, including process trees, networking, user IDs and mounted file systems

What are containers?



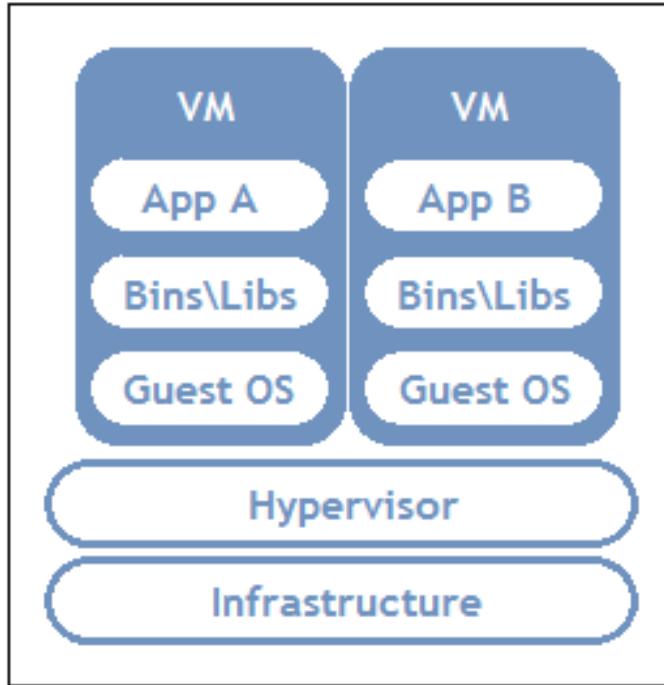
Containers vs VMs

Container Based Implementation



Small and Fast
Portable

Virtual Machine Implementation

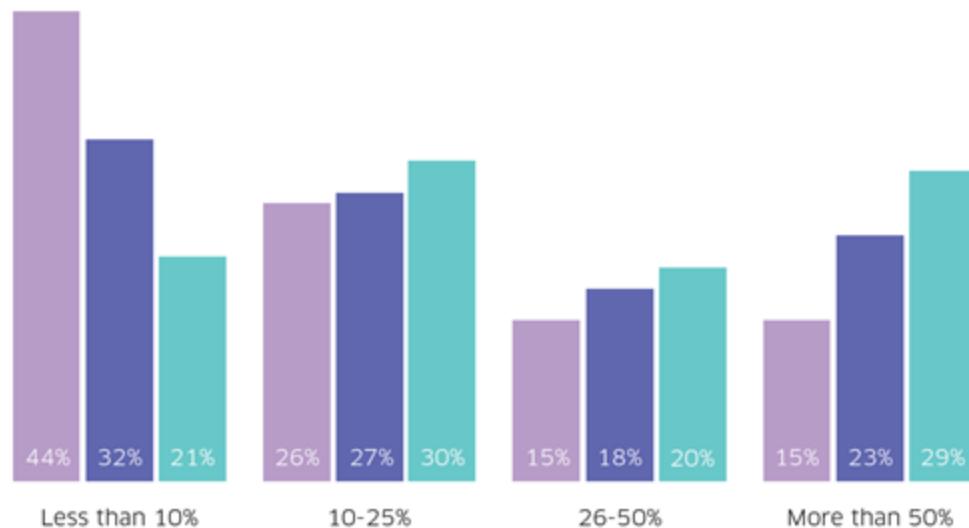


Heavy weight, non portable
Relies on OS package manager

Containers Adoption Rate

What percentage of your apps are containerized today?

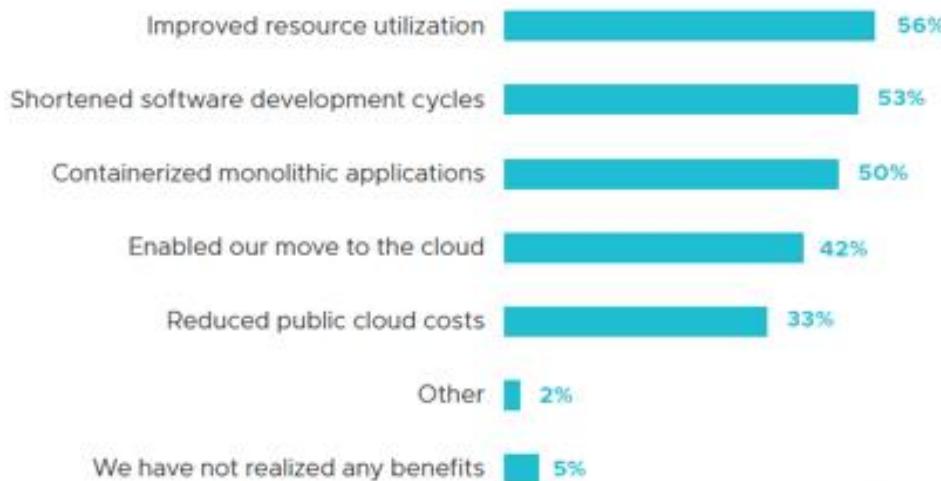
- Fall 2018
- Spring 2019
- Winter 2020



Perceived Benefits from containers adoption

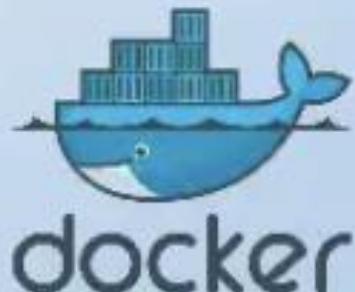
What benefits has your organization realized from operating Kubernetes?

Choose all that apply.



What is Docker

- Docker is an open-source project that automates the deployment of applications inside software container
- Docker containers wrap up a piece of software in a complete file system that contains everything it needs to run: code, runtime, system tools, system libraries - anything you can install on a server.
- This guarantees that it will always run the same, regardless of the environment it is running in.



What is Docker?

- "Installing Docker" really means "Installing the Docker Engine and CLI".
- The Docker Engine is a daemon (a service running in the background).
- This daemon manages containers, the same way that a hypervisor manages VMs.
- We interact with the Docker Engine by using the Docker CLI.
- The Docker CLI and the Docker Engine communicate through an API.
- There are many other programs and client libraries which use that API.

What makes docker containers so popular



Build once, run everywhere



Dependencies isolation



Ease of use with
docker command
line tool



Scalability with the
orchestration tools



Up and running in
seconds



Massive scale



Developers and Admins: Separation of concerns

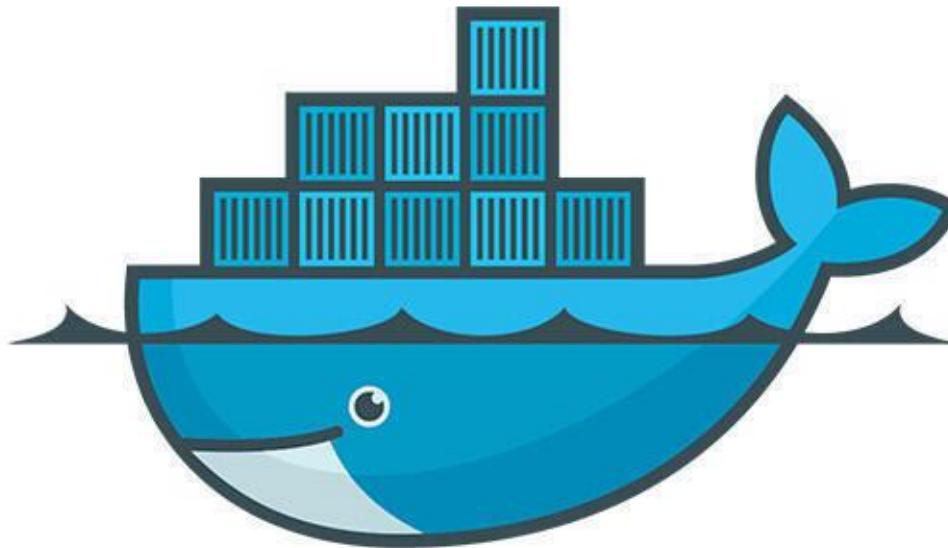
The Developer

- Responsible for the "inside" of the container
 - Code
 - Libraries
 - Package manager
 - Application
 - Data
- All Linux servers look **the same**

The Administrator

- } Responsible for the "outside" of the container
 - Logging
 - Monitoring
 - Remote Access
 - Networking
 - Security
- } Start, stop, attach, migrate all the containerized apps **the same way**

Lab1 – Run Docker Containers



docker

Run Docker Containers on Cloud9

We will use Cloud 9 to run the containers locally. We can run on our local machines as well, yet it will put strain on our laptop resources.

Use "sudo" when running Docker commands

```
$ docker version
```

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

```
hello world
```



Run Ubuntu on Amazon Linux VM

```
$ docker run -it ubuntu
root@04c0bb0a6c07:/#
root@04c0bb0a6c07:/# figlet CLO830
bash: figlet: command not found

# Install figlet
root@04c0bb0a6c07:/# apt-get update
root@04c0bb0a6c07:/# apt-get install figlet
root@04c0bb0a6c07:/# figlet CLO830 # Runs successfully

root@04c0bb0a6c07:/# exit
$ figlet CLO830 # not installed on the host!

# Re-run the container – will figlet work?
$ docker run -it ubuntu
root@04c0bb0a6c07:/# figlet CLO830



- 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.

```

Run the Hello-World Docker container on Amazon EC2

```
# Create ssh key pair and Amazon Linux EC2 in a public subnet in default VPC.  
  
# Use the provided Terraform code to deploy the EC2 instance  
  
$ ssh-keygen -t rsa -f week1-dev  
$ tf init  
$ tf apply --auto-approve  
  
# Log into the instance  
$ ssh -i week1-dev <elastic IP>  
  
# Is docker running?  
# Install Docker and run Hello-World container  
  
$ sudo yum update -y  
$ sudo yum install docker -y  
$ sudo systemctl start docker  
$ sudo systemctl status docker  
$ docker ps # Why does it fail?
```

Run the Hello-World Docker container on Amazon EC2

```
# Who has access to this socket?
```

```
$ ls -lra /var/run/docker.sock
```

```
# Add ec2-user to the docker group to enable docker cli use w/o sudo in the next session
```

```
$ sudo usermod -a -G docker ec2-user
```

```
# Log out of EC2 and log in again
```

```
$ docker info
```

```
$ docker run -it -p 80:80 hello-world
```

```
# List all the running containers
```

```
$ docker ps
```

```
# List all the images
```

```
$ docker images
```

```
[ec2-user@ip-10-1-0-102 ~]$ docker run -t -i -p 80:80 hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:c1c5c5b292d8525effc0f89cb299f1804f3a725c8d05e158653a563f15e4f685
Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

```
To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
```

```
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
```

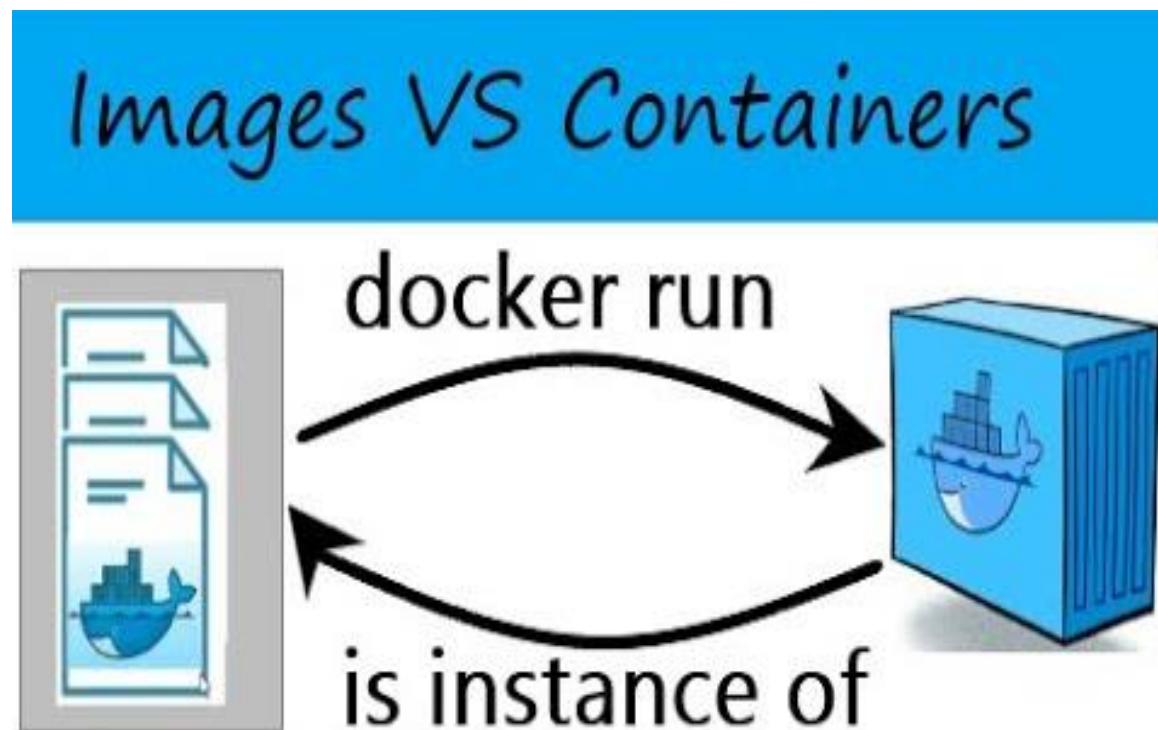
```
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.
```

```
To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash
```

```
Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/
```

```
For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

Where's my container?



Background and foreground

- The distinction between foreground and background containers is arbitrary.
- From Docker's point of view, all containers are the same.
- All containers run the same way, whether there is a client attached to them or not.
- It is always possible to detach from a container, and to reattach to a container.
- Analogy: attaching to a container is like plugging a keyboard and screen to a physical server.

Docker CLI – Working with Containers

```
# Run the container in the background (detached)
```

```
$ docker run -d nginx
```

```
# Mapping host port to container port
```

```
$ docker run -p 8001:80 -d nginx
```

```
# Volume Mapping
```

```
$ docker run -v /tmp:/var/lib -d nginx
```

```
# Inspect Container
```

```
$ docker inspect <container id>
```

```
# List running containers
```

```
$ docker ps
```

```
# View the logs
```

```
$ docker logs <container id>
```

```
# Stop the running container
```

```
$ docker stop <container id>
```

Docker CLI – Working with Containers - Cont

```
# View the logs
```

```
$ docker logs <container id>
```

```
# Stop the running container
```

```
$ docker stop <container id>
```

```
# Restarting a container
```

```
$ docker start <container id>
```

```
# Executing command inside the container
```

```
$ docker exec <container name> cat /etc/hosts
```

```
$ docker attach <container id>
```

```
# List ALL containers
```

```
$ docker ps -a
```

```
# Remove the stopped container
```

```
$ docker rm <container id>
```

Docker CLI – Working with Environment Variables

```
# Define environment variable $K8S_COLOR with value GREEN in your host session  
$ export K8S_COLOR=GREEN  
# Pass environment variable value BLUE to the Docker container and attempt to print it out  
  
$ docker run -e K8S_COLOR=BLUE ubuntu echo $K8S_COLOR  
  
# Explain the reason the command above did not print the environment variable value GREEN. Refer to the link in the presenter notes  
  
# Re-run the command in the formats below  
$ docker run -e K8S_COLOR=BLUE ubuntu printenv  
$ docker run -e K8S_COLOR=BLUE ubuntu /bin/sh -c 'echo $K8S_COLOR'  
  
# Explain the reason the commands above are successfully printing out the value of $K8S_COLOR as BLUE
```

Docker CLI – Working with Docker Images

```
# Listing the images
```

```
$ docker images
```

```
# Remove Images
```

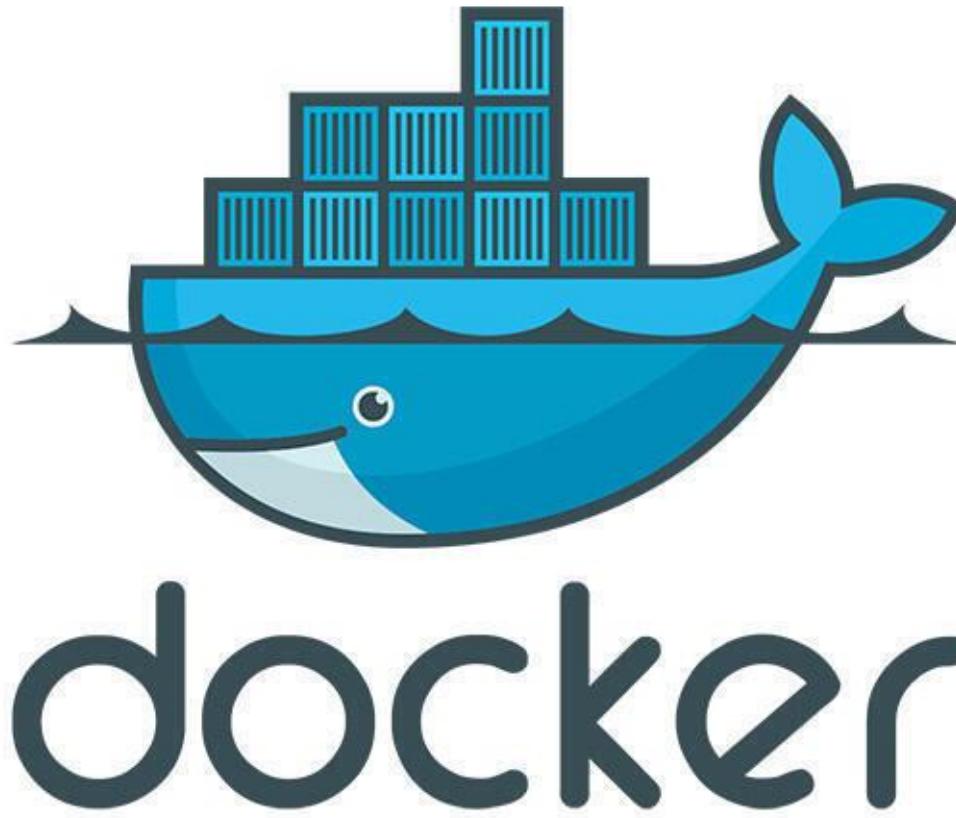
```
$ docker rmi ubuntu
```

```
# Download an image
```

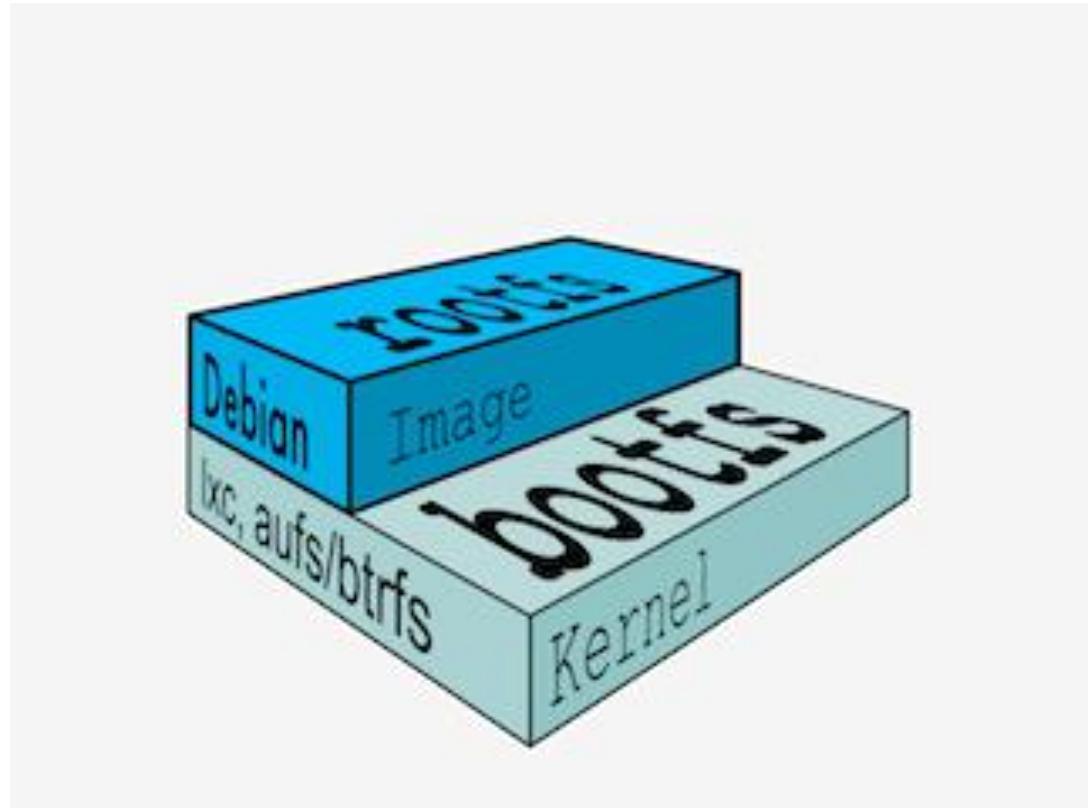
```
$ docker run nginx
```

```
$ docker pull nginx
```

Lab 1 – The end



Understanding Docker Images



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.



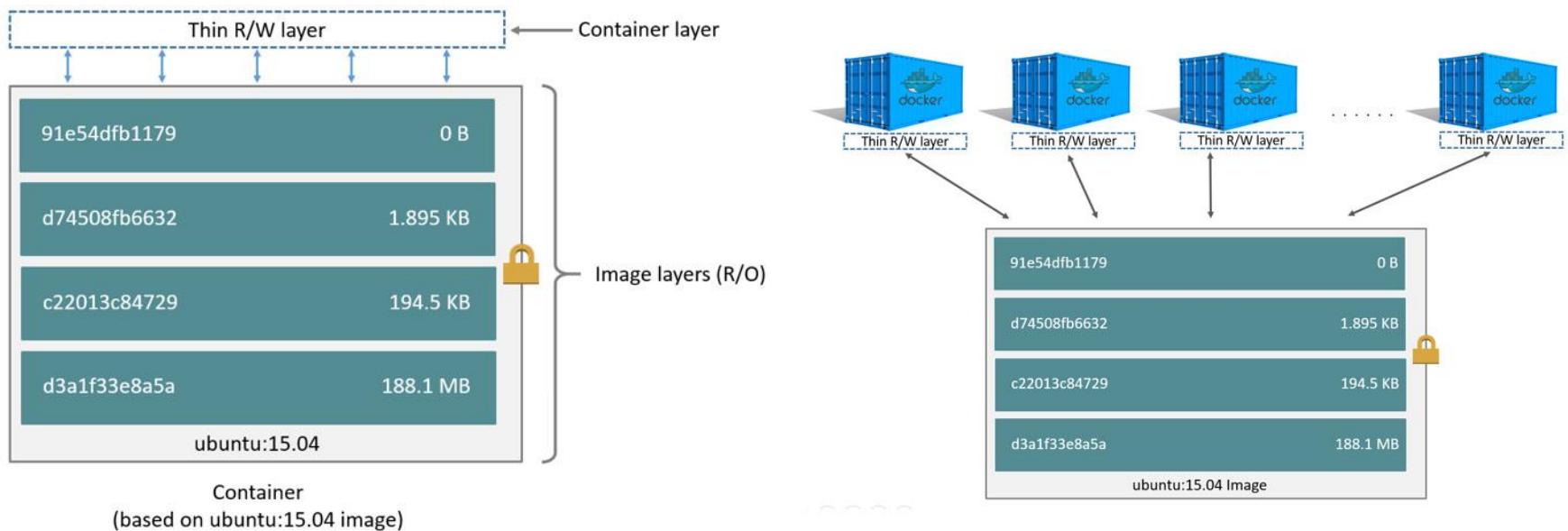
Image Layers Example for Java Web Applications

- 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.)

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.



Differences between containers and images



An image is a read-only filesystem.

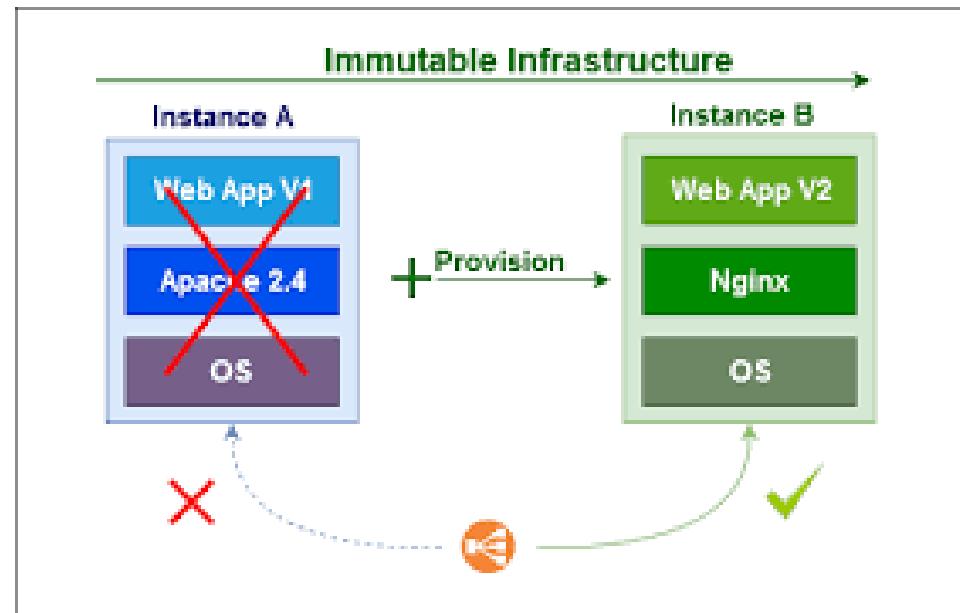
A container is an encapsulated set of processes, running in a read-write copy of that filesystem.

To optimize container boot time, *copy-on-write* is used instead of regular copy.

docker run starts a container from a given image.

Docker Images Are Immutable. So how can we change them?

- 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.



Images Namespaces

There are three namespaces:

- Official images- gated by Docker Inc, authored and maintained by third parties e.g. ubuntu, busybox ...
- User (and organizations) images for DockerHub users e.g. igeiman/exampleapp
- Self-hosted images – not hosted in DockerHub
 - e.g. registry.example.com:5000/my-private/image

Docker File



Docker can build images by reading the instructions provided in Dockerfile



A Dockerfile is a text document that contains all the commands a user could call to assemble an image



Using docker build users can create an automated build that executes a set of command line instructions in succession



The docker build command builds an image from a Docker file and a context

Docker File: Example

- # Each instruction in this file generates a new layer that gets pushed to your local image cache
- # Lines preceded by # are regarded as comments and ignored
- # The line below states we will base our new image on the Latest Official Ubuntu
- FROM ubuntu:latest
- # Identify the maintainer of an image
- LABEL mymame@somecompany.com
- # Update the image to the latest packages
- RUN apt-get update && apt-get upgrade -y
- # Install NGINX to test.
- RUN apt-get install nginx -y
- # Expose port 80
- EXPOSE 80
- # Last is the actual command to start up NGINX within our Container
- CMD ["nginx", "-g", "daemon off;"]

- \$ docker build (used 99% of the time)
- Performs a repeatable build sequence.
- This is the preferred method!
- Images can be stored on your Docker host or in docker registry
- # Adding tags
- \$ docker tag <image id> nginx

Dockerfile usage summary

- Dockerfile instructions are executed in order.
- Each instruction creates a new layer in the image.
- Docker maintains a cache with the layers of previous builds.
- When there are no changes in the instructions and files making a layer, the builder re-uses the cached layer, without executing the instruction for that layer.
- The FROM instruction MUST be the first non-comment instruction.
- Lines starting with # are treated as comments.
- Some instructions (like CMD or ENTRYPOINT) update a piece of metadata.
- (As a result, each call to these instructions makes the previous one useless.)

The RUN instruction

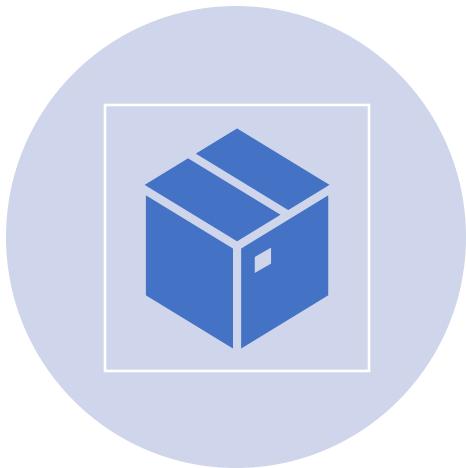
The RUN instruction can be specified in two ways.

With shell wrapping, which runs the specified command inside a shell, with /bin/sh -c:

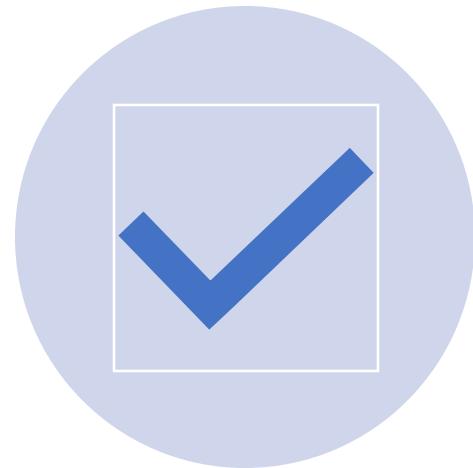
RUN apt-get update

Or using the exec method, which avoids shell string expansion, and allows execution in images that don't have /bin/sh:

RUN ["apt-get", "update"]



CONTAINERS



WEEK 2

Agenda

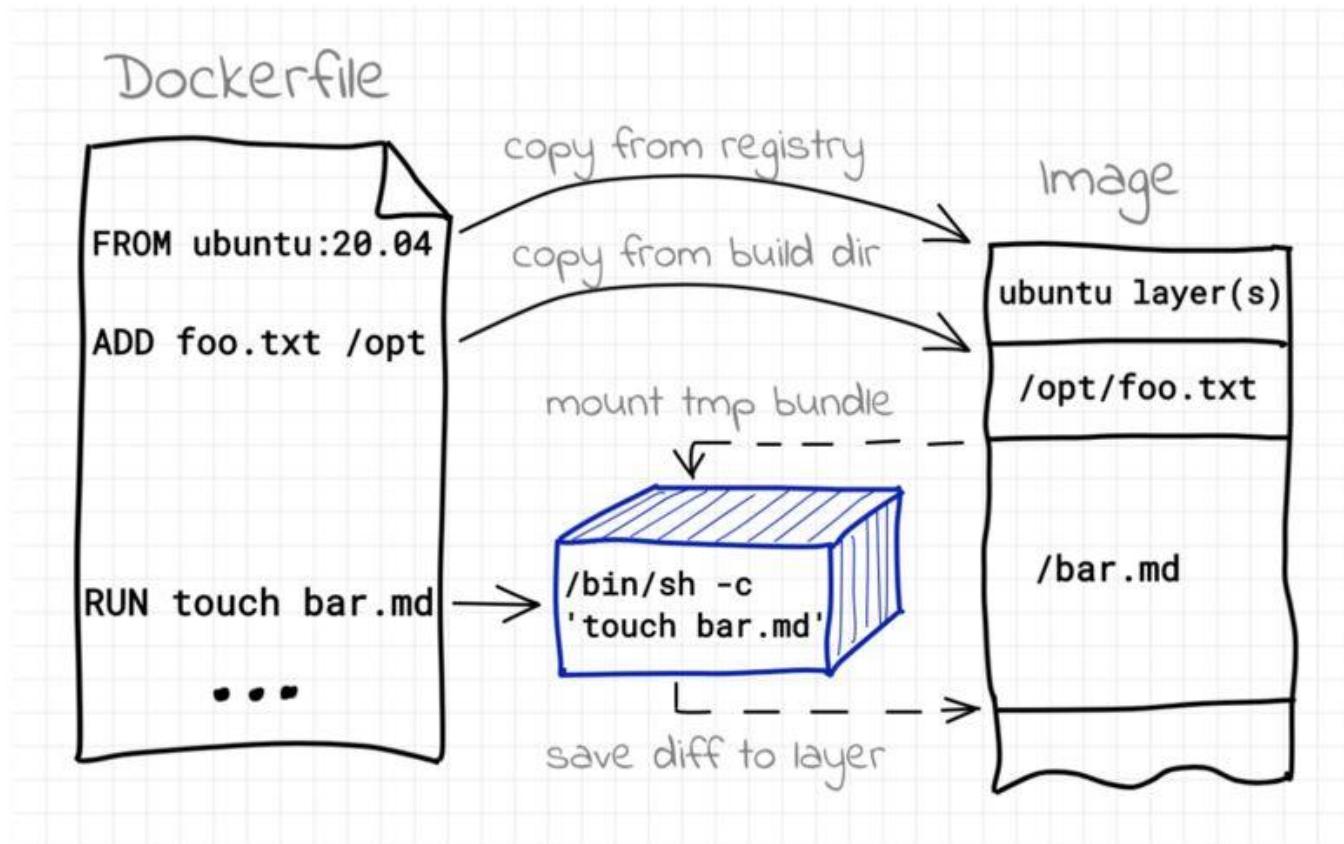
Docker Images

Dockerfile

Create Docker Images

Publish Docker Images with Amazon ECR

Building Docker Image



Building image using Dockerfile



The EXPOSE instruction

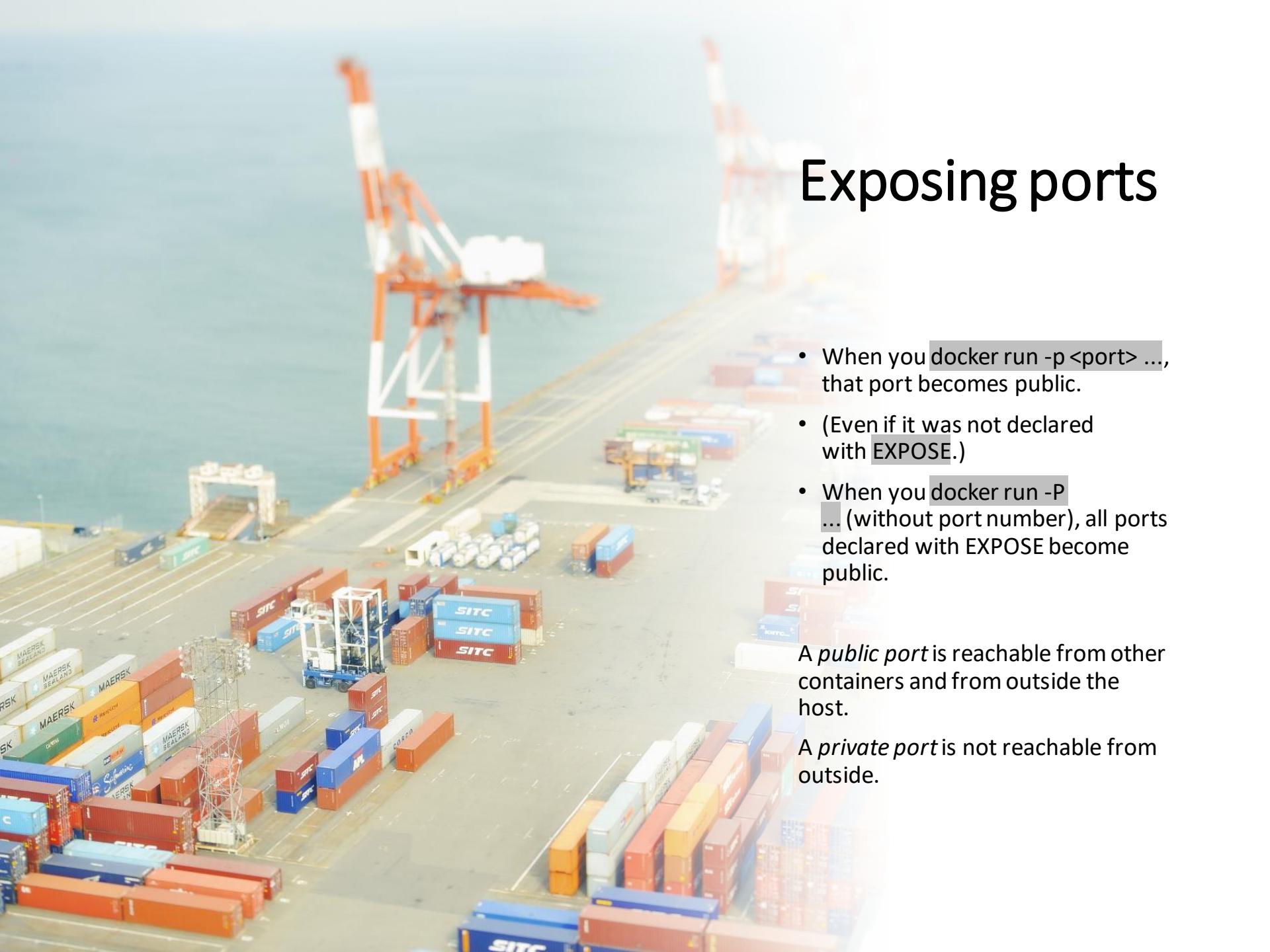
The **EXPOSE** instruction tells Docker what ports are to be published in this image.

EXPOSE 8080

EXPOSE 80 443

EXPOSE 53/tcp 53/udp

- All ports are private by default.
- Declaring a port with EXPOSE is not enough to make it public.
- The Dockerfile doesn't control on which port a service gets exposed.

An aerial photograph of a large port facility. In the foreground, there are numerous shipping containers stacked in various colors (red, blue, white, green) on a concrete dock. Industrial cranes with orange and white striped booms are positioned along the edge of the water, some with white containers suspended from them. The water is a light blue-green color. The overall scene is a complex network of industrial activity.

Exposing ports

- When you `docker run -p <port> ...`, that port becomes public.
- (Even if it was not declared with `EXPOSE`.)
- When you `docker run -P ...` (without port number), all ports declared with `EXPOSE` become public.

A *public port* is reachable from other containers and from outside the host.

A *private port* is not reachable from outside.

CMD and ENTRYPPOINT



Adding CMD to Dockerfile

- 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.

```
FROM ubuntu
RUN apt-get update
CMD sleep 20
```

Adding ENTRYPOINT to the Dockerfile

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
ENTRYPOINT ["sleep"]
```

- 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?

CMD vs Entrypoint

The default process invoked by Ubuntu container is **bash**.

If we specify the command at a runtime, the original CMD instruction is **replaced** if CMD is used in Dockerfile.

If we specify the command at a runtime, the command line parameter is **appended** to the ENTRYPOINT instruction in the Dockerfile.

```
#  
# Ubuntu Dockerfile  
#  
# https://github.com/dockerfile/ubuntu  
#  
  
# Pull base image.  
FROM ubuntu:14.04  
  
# Install.  
RUN \  
    sed -i 's/# \(.*multiverse$\)/\1/g' /etc/apt/sources.list && \  
    apt-get update && \  
    apt-get -y upgrade && \  
    apt-get install -y build-essential && \  
    apt-get install -y software-properties-common && \  
    apt-get install -y byobu curl git htop man unzip vim wget && \  
    rm -rf /var/lib/apt/lists/*  
  
# Add files.  
ADD root/.bashrc /root/.bashrc  
ADD root/.gitconfig /root/.gitconfig  
ADD root/.scripts /root/.scripts  
  
# Set environment variables.  
ENV HOME /root  
  
# Define working directory.  
WORKDIR /root  
  
# Define default command.  
CMD ["bash"]
```

Using Environment Variables



Example of MySQL image on DockerHub

https://hub.docker.com/_/mysql

Start a mysql server instance

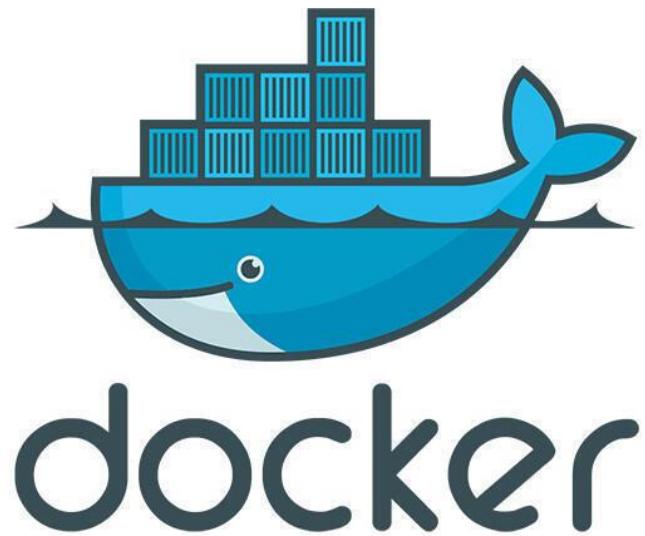
Starting a MySQL instance is simple:

```
$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d  
mysql:tag
```

Where:

- `some-mysql` is the name you want to assign to your container,
- `my-secret-pw` is the password to be set for the MySQL root user
- `tag` is the tag specifying the MySQL version you want.

Week 2, Lab1– Environment Variables, CMD and ENTRYPOINT



Why and How are we going to create a Docker Image

- Why? We want to containerize custom Python Flask application
- How?
 1. Start from the vanilla Ubuntu image
 2. Update apt
 3. Use apt to install relevant packages
 4. Use pip to install Python application dependencies
 5. Copy source code to /opt folder
 6. Run the simple web server using `flask` command

Create Dockerfile and build Docker image

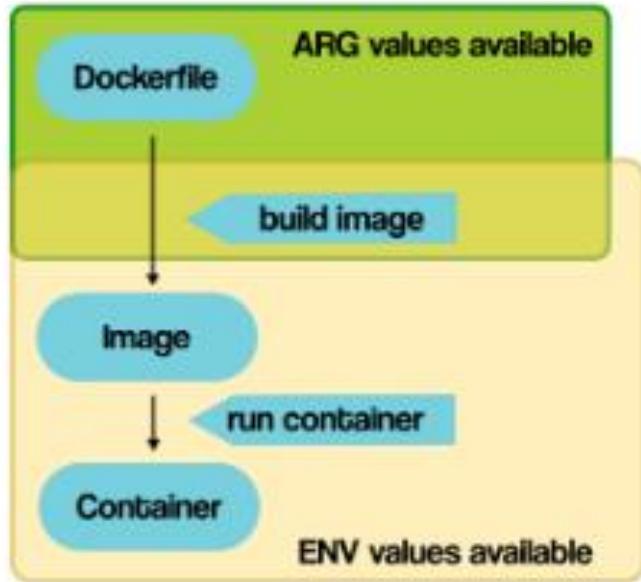
```
git clone https://github.com/mmumshad/simple-webapp-color.git  
cd simple-webapp-color/
```

```
# Create Dockerfile with the content below  
FROM ubuntu:18.04  
RUN apt-get update && apt-get install -y python python-pip  
RUN pip install flask  
# Mind the space between . and /opt/  
COPY . /opt/  
ENV APP_COLOR=red  
ENTRYPOINT FLASK_APP=/opt/app.py flask run --host=0.0.0.0 --port=8080
```

Explain:

What is the purpose of COPY directive in the Dockerfile above?

Why are the advantages of using ENTRYPOINT in place of CMD in the Dockerfile above?



Dockerfile content:

```

ARG required_var      # expects a value
ARG var_name=def_value # set default ARG value
ENV foo=other_value   # set default ENV value
ENV bar=${var_name}    # set default ENV value from ARG

```

Override Dockerfile ARG values on build:

```
$ docker build --build-arg var_name=value
```

Override Docker image ENV values on run:

```

$ docker run -e "foo=other_value" [...]
$ docker run -e foo [...] # pass from host
$ docker run --env-file=env_file_name [...] # pass from file

```

<https://vsupalov.com/docker-env-vars>

```

# Build docker image
$ docker build -t color_app .
# List docker images
$ docker images
# Create a container – can we access the app?
$ docker run -d -e APP_COLOR=blue color_app
# Create a container and publish the port
$ docker run -d -e APP_COLOR=blue -p 80:8080 color_app
# Verify that you can access the application locally via "curl localhost" command and from your browser using public IP of your Cloud9 environment
# Make sure to open relevant port in the Security Group
# Add screenshots of application responses to your report

```

Provide Values to Container using Environment Variables

Environment Variables of a Running Container

```
# Examine environment variables of the  
container using "docker inspect <container  
id> command  
$ docker inspect <container id>
```

Explain using the color_app example:

- How is the value of `APP_COLOR` environment variable made available to flask application?
- Is it enough specifying the environment variable in the Docker file or should it be mentioned in the application code as well?

Environment Variables of the Official Image

```
# Try running mysql container using official image  
$ docker pull mysql  
$ docker run -d mysql
```

```
# Adding environment variables, re-running  
$ docker run --name some-mysql -e  
MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql
```

```
# Checking the image configuration  
$ docker ps  
$ docker inspect <container id>
```

CMD vs Entrypoint, continued

```
# Overriding the CMD instruction
```

```
$ docker run ubuntu ls
```

```
# Create a new with ENTRYPOINT instruction
```

```
FROM ubuntu:16.04
```

```
ENTRYPOINT ["ls"]
```

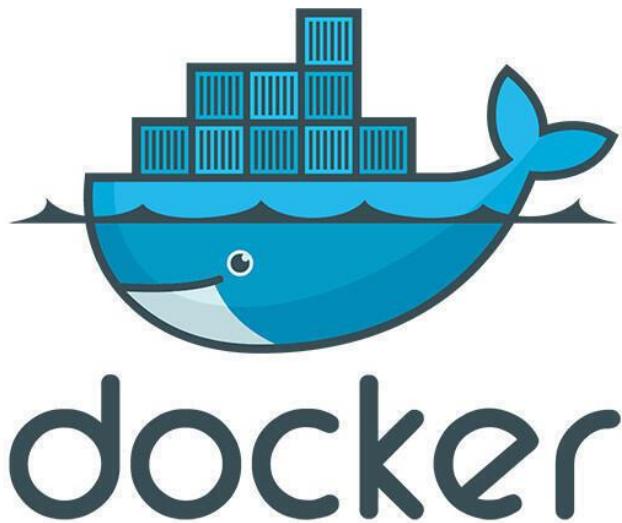
```
$ docker build -t ubuntu-ls .
```

```
$ docker run ubuntu-ls /etc
```

```
$ docker run ubuntu-ls ls /etc
```

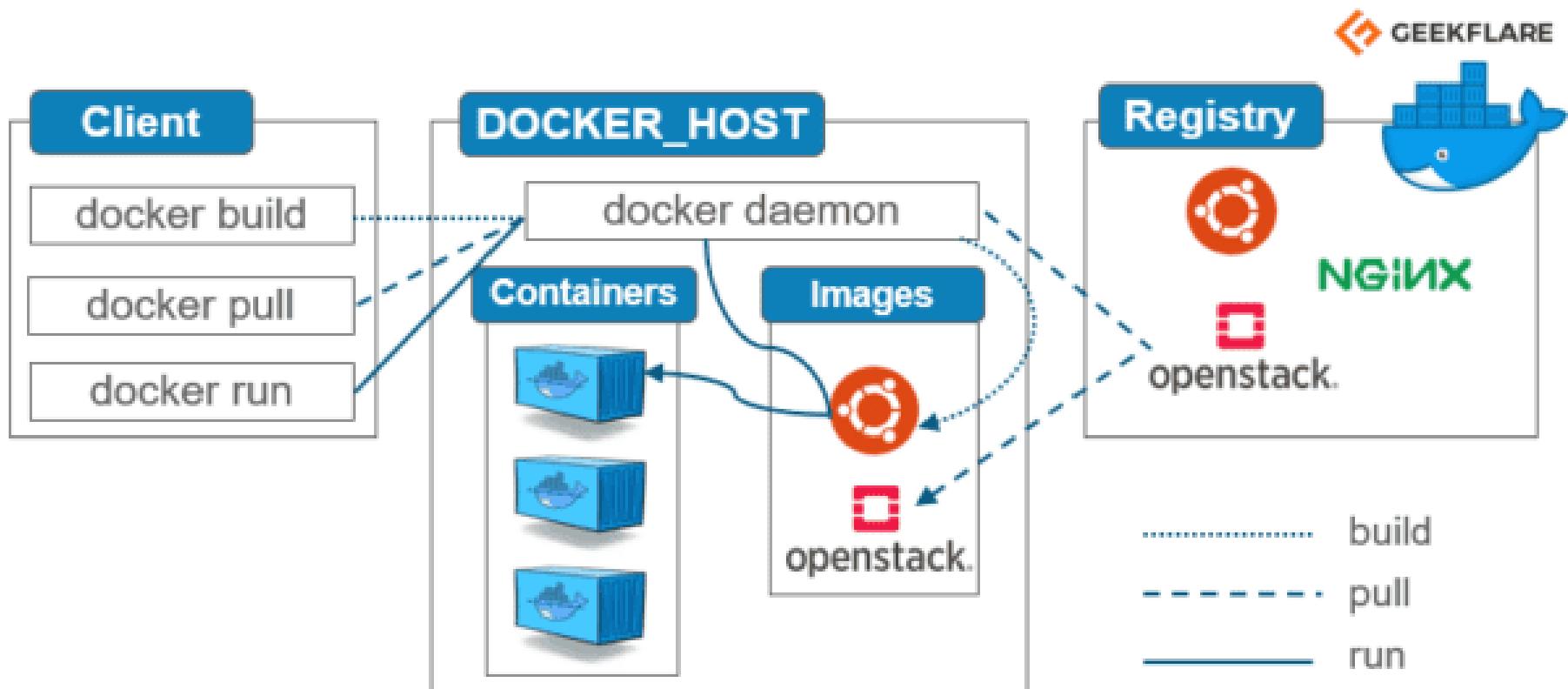
```
# Explain the reason this command reports an error
```

```
# Would the command error out if we replace ENTRYPOINT with CMD? Explain.
```



Week2, Lab1 – The End

Publishing Docker Images



Docker Registry

Docker registries hold images

These are public or private stores used to upload or download docker images

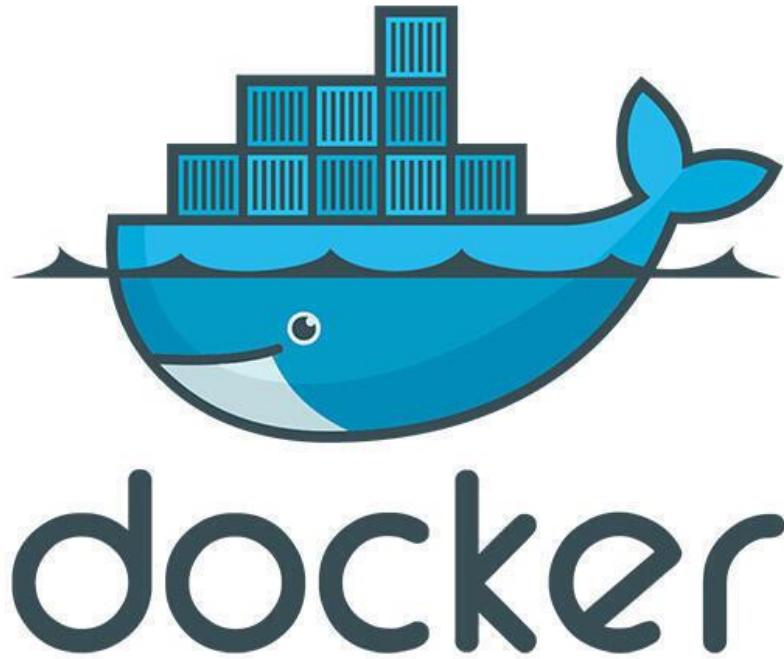
It serves a collection of docker images created and shared by you or other docker registry users

Docker registries are the distribution component of Docker

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?
- `ubuntu` is, in fact, `library/ubuntu`, i.e. `index.docker.io/library/ubuntu`

```
# Tagging and publishing Docker image
docker tag <registry>/<user>/<image>:clo835-week2
docker push <registry>/<user>/<image>:clo835-week2
```



Week 2, Lab2– Building and Publishing Docker Images

Workshop Flow

1

Build docker images for web server, app server and PostgreSQL db

2

Run each of the components locally as docker containers

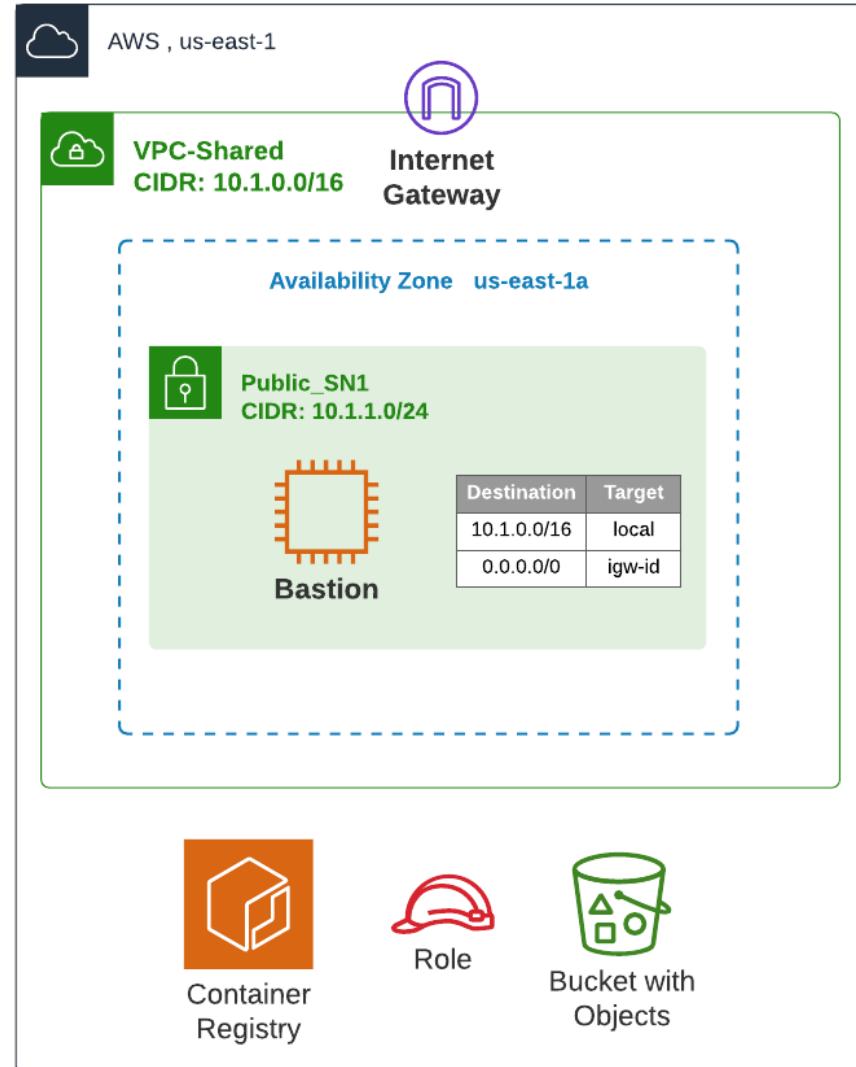
3

Create Amazon ECR repositories

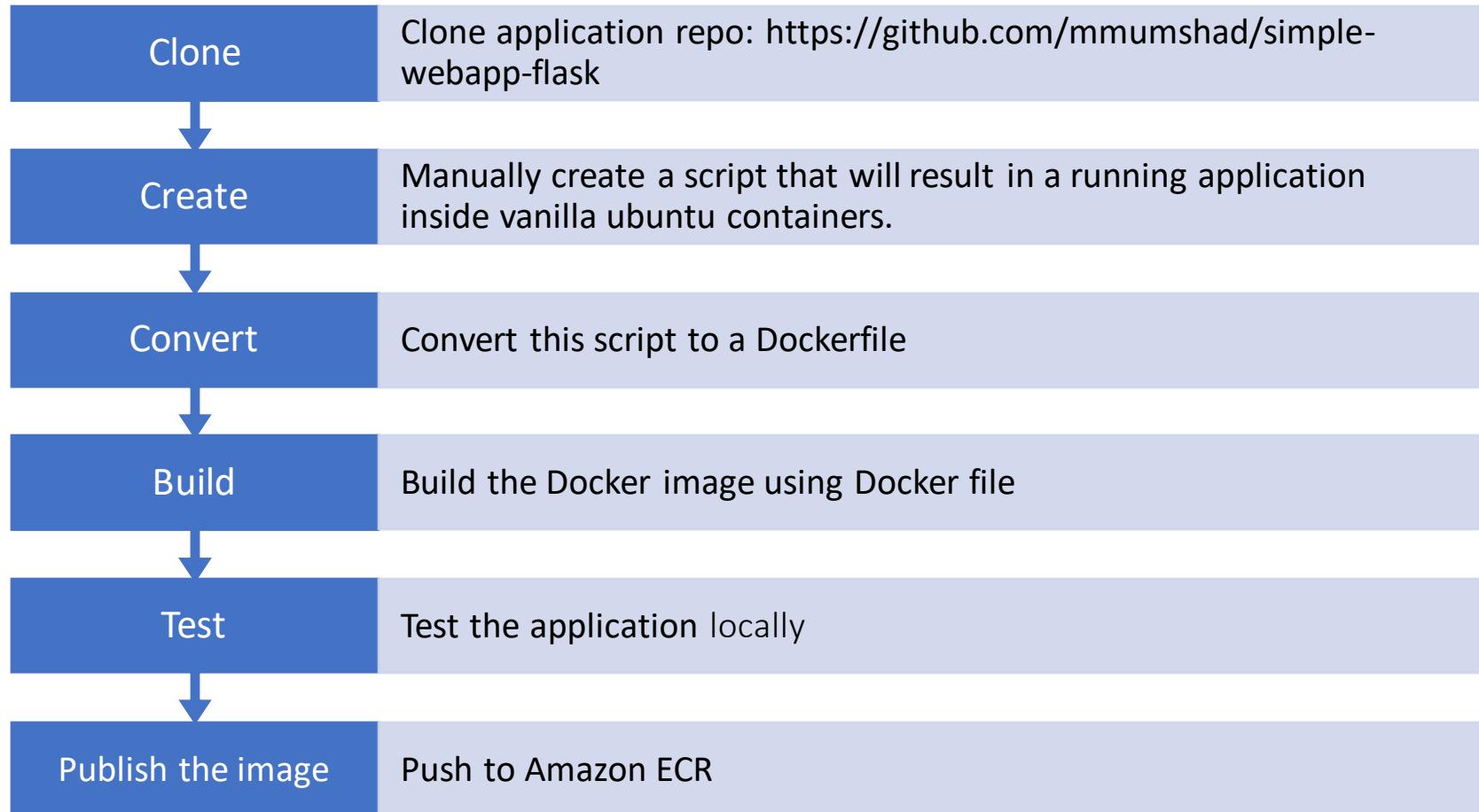
4

Push the images to Amazon ECR

Architecture Diagram



Creating a Simple Flask Web Application



Clone the Application Repo and Create a Script

```
# Install git
$ sudo yum install git -y

# Clone the repo
$ git clone https://github.com/mmumshad/simple-webapp-flask && cd simple-webapp-flask/

# Run ubuntu container in an interactive mode
$ docker run -it ubuntu bash

# Install python and pip inside the ubuntu container. Notice that you run as a root user
root@3ed51fd4362c:/# apt-get update
root@3ed51fd4362c:/# apt-get install python-pip -y
root@3ed51fd4362c:/# apt-get install pip -y

# Install flask using python package manager
root@3ed51fd4362c:/# pip install flask

# Copy the code of app.py into /opt folder and start the webserver. Use CTRL+D to save and exit
root@2946eb4d0d81:/# cat > /opt/app.py
root@2946eb4d0d81:/# cd /opt
root@2946eb4d0d81:/# FLASK_APP=app.py flask run --host=0.0.0.0

# Open another terminal and send HTTP request
$ curl http://172.17.0.2:5000
```

```
[ec2-user@ip-172-31-93-152 ~]$ curl http://172.17.0.2:5000
Welcome! [ec2-user@ip-172-31-93-152 ~]$
[ec2-user@ip-172-31-93-152 ~]$ curl http://172.17.0.2:5000
```

```
* Running on http://172.17.0.2:5000 (Press CTRL+C to quit)
172.17.0.1 - - [16/Apr/2022 02:46:36] "GET / HTTP/1.1" 200 -
```

Converting the Script to a Dockerfile. Build Docker image

```
# Try building the original Dockerfile from the repo. Is it successful?  
# Let's fix it with the new Dockerfile definition  
FROM ubuntu:22.04  
RUN apt-get update && apt-get install -y python-pip  
RUN apt-get install -y pip  
RUN pip install flask  
COPY app.py /opt/  
ENTRYPOINT FLASK_APP=/opt/app.py flask run --host=0.0.0.0 --port=8080
```

```
# Build Docker image  
$ docker build . -t clo835-week2  
# List images  
$ docker images  
# Run the container  
$ docker run clo835-week2  
# Test locally using container IP  
$ curl 172.17.0.2:8080
```

```
[ec2-user@ip-172-31-93-152 simple-webapp-flask]$ docker images  
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE  
clo835-week2    latest   d499aabc969a  2 minutes ago  456MB  
<none>          <none>   8a37ee1f1206  4 minutes ago  432MB  
ubuntu          latest   825d55fb6340  10 days ago   72.8MB  
hello-world     latest   feb5d9fea6a5  6 months ago  13.3kB  
ubuntu          16.04    b6f507652425  7 months ago  135MB  
[ec2-user@ip-172-31-93-152 simple-webapp-flask]$ █
```

```
[ec2-user@ip-172-31-93-152 simple-webapp-flask]$ docker run clo835-week2  
* Serving Flask app '/opt/app.py' (lazy loading)  
* Environment: production  
  WARNING: This is a development server. Do not use it in a production deployment.  
  Use a production WSGI server instead.  
* Debug mode: off  
* Running on all addresses (0.0.0.0)  
  WARNING: This is a development server. Do not use it in a production deployment.  
* Running on http://127.0.0.1:8080  
* Running on http://172.17.0.2:8080 (Press CTRL+C to quit)  
[ec2-user@ip-172-31-93-152 ~]$ █
```

```
[ec2-user@ip-172-31-93-152 ~]$ curl http://172.17.0.2:8080  
Welcome! [ec2-user@ip-172-31-93-152 ~]$ █
```

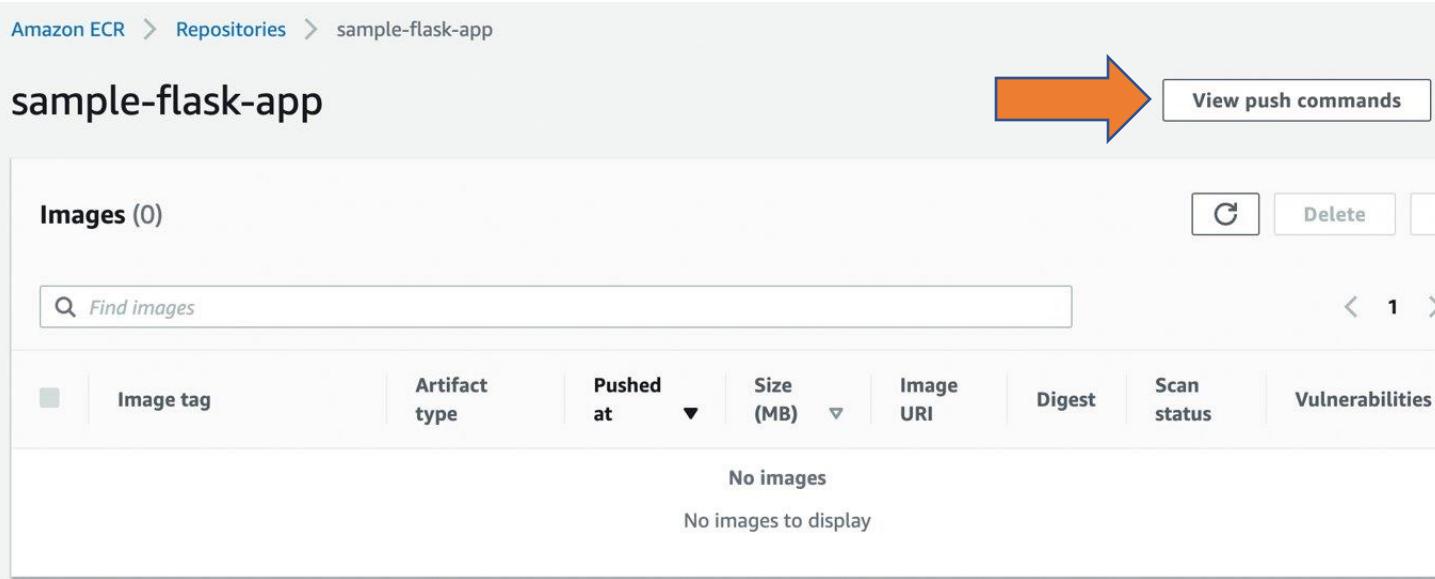
Publish Docker image to Amazon ECR

```
aws ecr create-repository --repository-name sample-flask-app --image-scanning-configuration scanOnPush=true  
# Find the docker repo name  
aws ecr describe-repositories  
  
# Export docker repo name, replace with your account id  
export ECR="417374112702.dkr.ecr.us-east-1.amazonaws.com/sample-flask-app"  
export REPO_NAME=sample-flask-app
```

```
ddd_v1_w_IN3_1136420@runweb53512:~$ aws ecr describe-repositories  
{  
    "repositories": [  
        {  
            "repositoryArn": "arn:aws:ecr:us-east-1:417374112702:repository/sample-flask-app",  
            "registryId": "417374112702",  
            "repositoryName": "sample-flask-app",  
            "repositoryUri": "417374112702.dkr.ecr.us-east-1.amazonaws.com/sample-flask-app",  
            "createdAt": "2022-04-15T20:40:19-07:00",  
            "imageTagMutability": "MUTABLE",  
            "imageScanningConfiguration": {  
                "scanOnPush": true  
            },  
            "encryptionConfiguration": {  
                "encryptionType": "AES256"  
            }  
        }  
    ]  
}
```

Authenticate to Amazon ECR

```
# Authenticate to Amazon ECR, replace with your Account id  
# The easiest way is to copy this command from the ECR blade in AWS Management Console  
  
aws ecr get-login-password --region us-east-1 | docker login -u AWS ${ECR} --password-stdin
```



Amazon ECR > Repositories > sample-flask-app

sample-flask-app

View push commands

Images (0)

Find images

No images

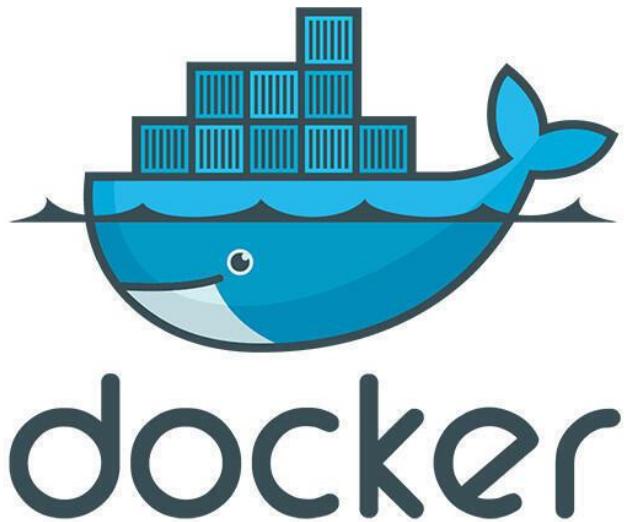
No images to display

```
[ec2-user@ip-172-31-93-152 ~]$ export ECR="417374112702.dkr.ecr.us-east-1.amazonaws.com/sample-flask-app"  
[ec2-user@ip-172-31-93-152 ~]$ aws ecr get-login-password --region us-east-1 | docker login -u AWS ${ECR} --password-stdin  
WARNING! Your password will be stored unencrypted in /home/ec2-user/.docker/config.json.  
Configure a credential helper to remove this warning. See  
https://docs.docker.com/engine/reference/commandline/login/#credentials-store  
Login Succeeded
```

Publish the Image To Amazon ECR

- # Tag the docker image
\$ docker tag clo835-week2 "\${ECR}:v1.0"
- # List docker images in your local environment
\$ docker images
- # Publish docker image in Amazon ECR
\$ docker push "\${ECR}:v1.0"
- # List images in Amazon ECR repo to verify your image was published
\$ aws ecr list-images --repository-name \${REPO_NAME}

```
vocabs:~/environment $ docker push "${ECR}:v1.0"
The push refers to repository [657756357702.dkr.ecr.us-east-1.amazonaws.com/sample-flask-app]
feb5565a868f: Pushed
36b0ea76dfde: Pushed
589fe07308c3: Pushed
02152cbf4db8: Pushed
7f5cbd8cc787: Pushed
v1.0: digest: sha256:2766946e42a0cd902a9c22fa3cfcab8745995b900a93ef88a2ab3e7a863929cb size: 1372
```

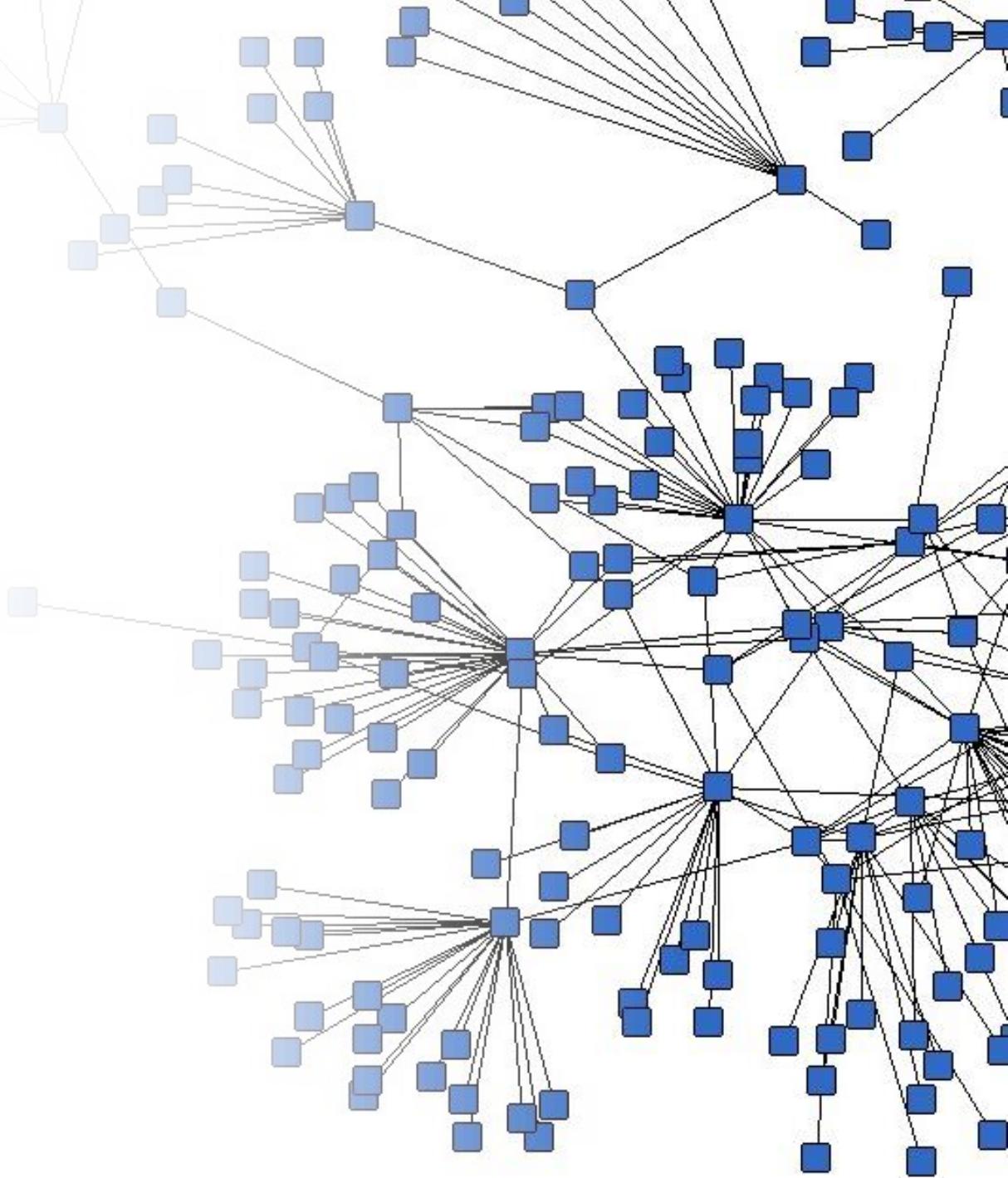


Week 2, Lab2 – The End

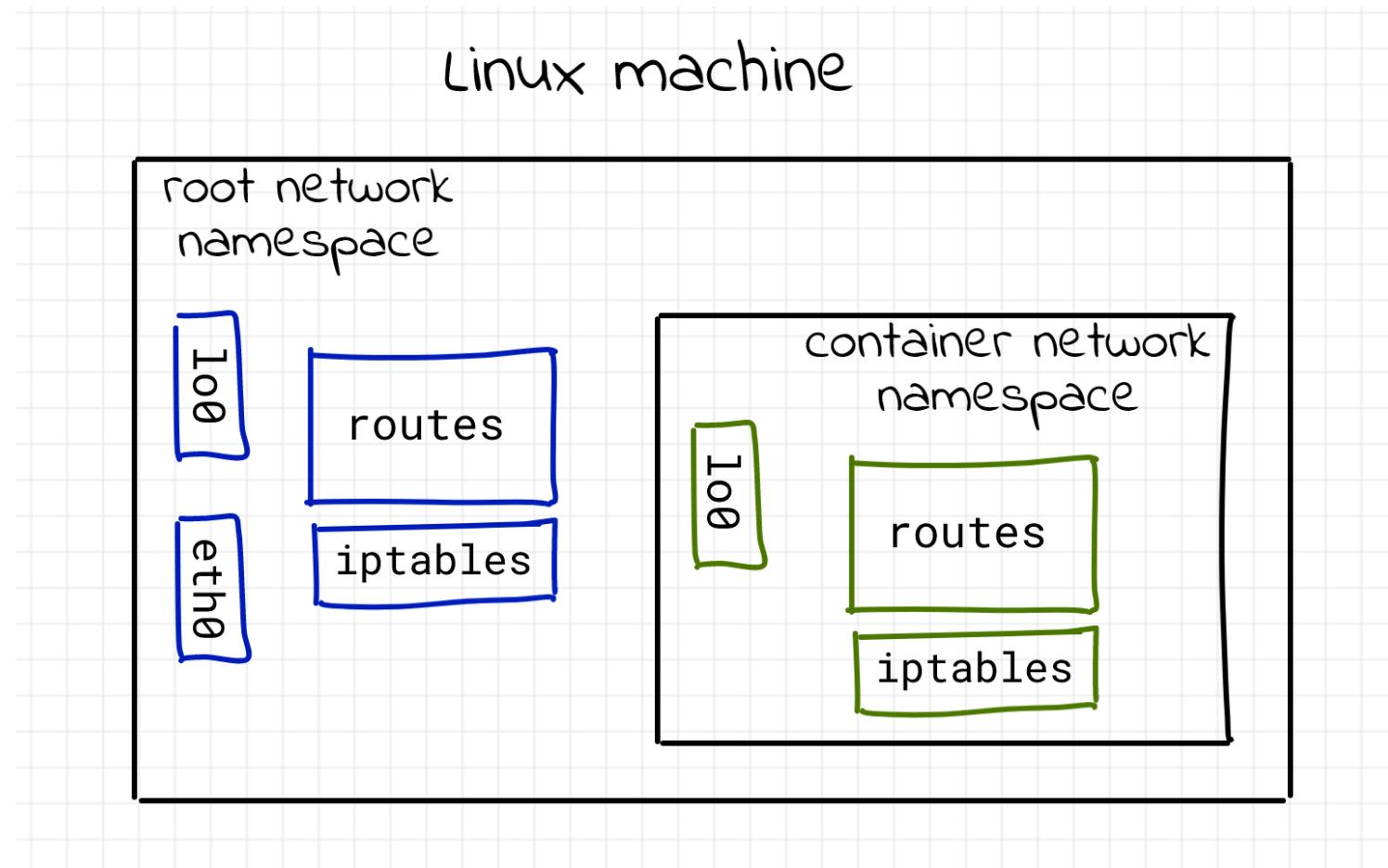


Container Networking Basics

Containers
Networking is
Simple!
Just kidding, it's not..



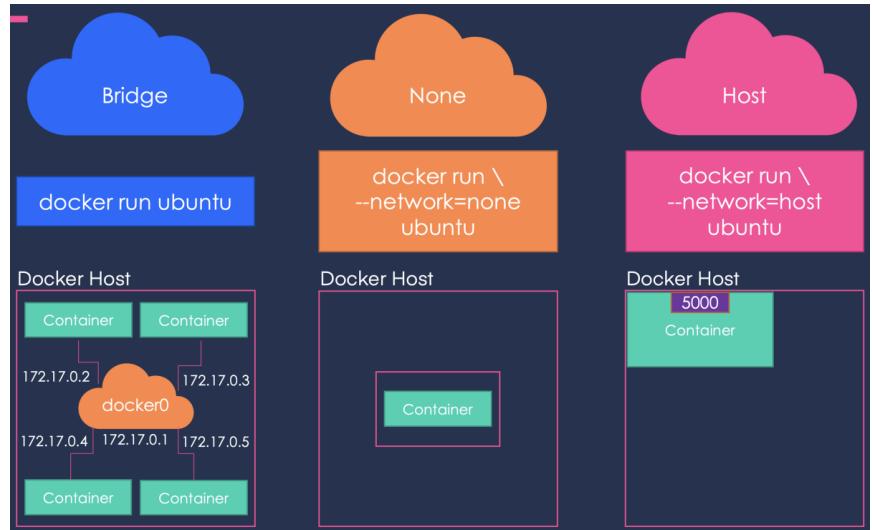
Container Network Namespace



Containers

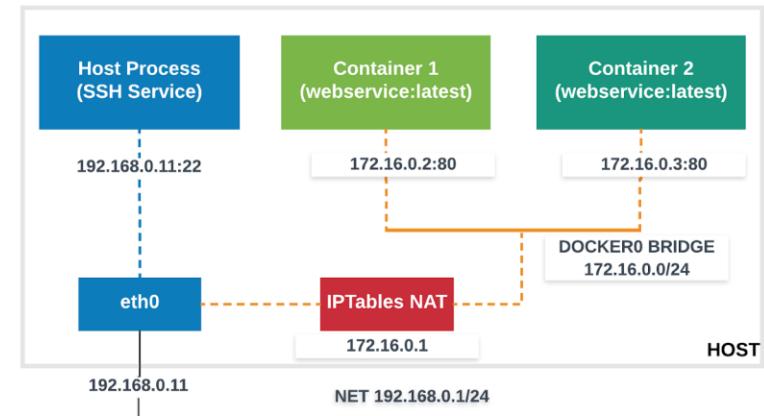
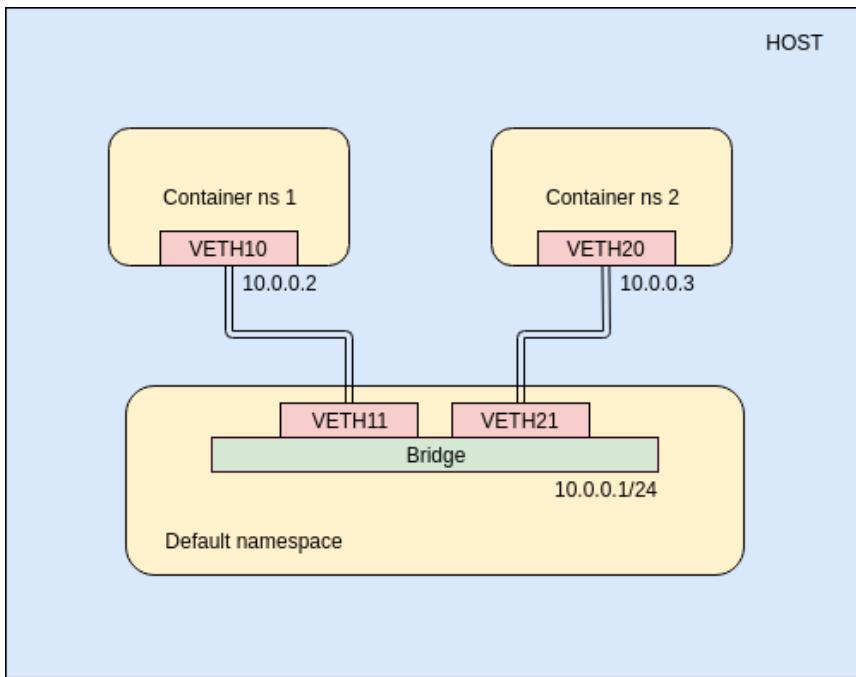
Network Drivers

- When we install Docker, it comes with the drivers that support the networks below
 - bridge (default)
 - null
 - host
- New network is created with `docker network create`
- The network is selected with `docker run --net`
- Each network is managed by a driver.



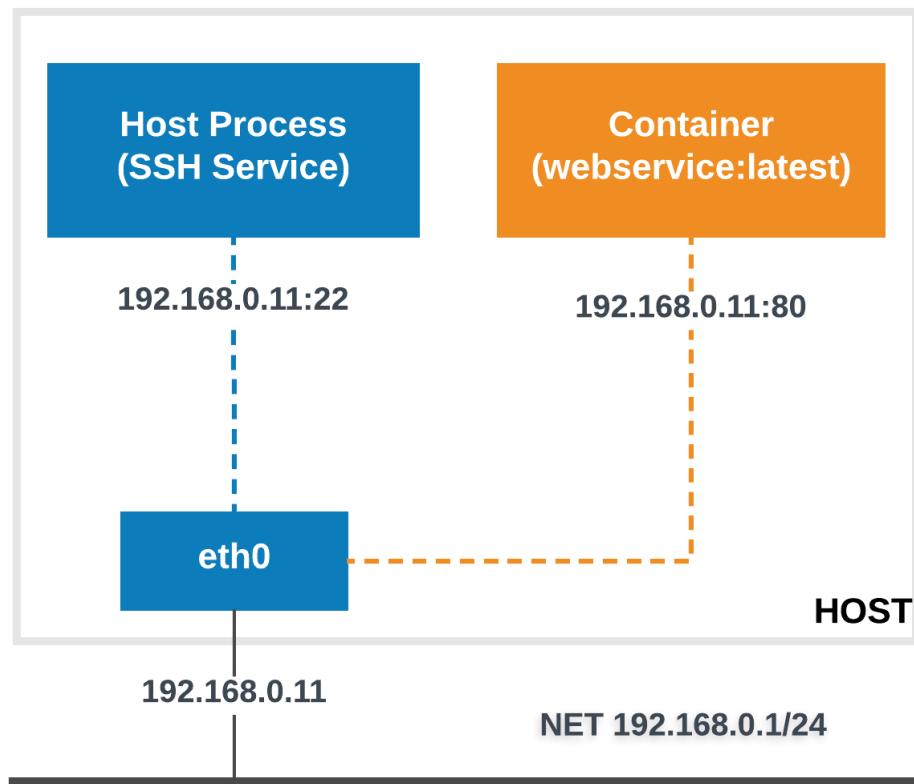
The Default Bridge

- By default, the container gets a virtual `eth0` interface.
(In addition to its own private `lo` loopback interface.)
- That interface is provided by a `veth` pair.
- It is connected to the Docker bridge.
(Named `docker0` by default; configurable with `--bridge`.)
- Addresses are allocated on a private, internal subnet.
(Docker uses `172.17.0.0/16` by default; configurable with `--bip`.)
- The container can have its own routes, iptables rules, etc.



The Host Driver

- Container is started with `docker run --net host ...`
- It sees (and can access) the network interfaces of the host.
- It can bind any address, any port (for ill and for good).
- Network traffic doesn't have to go through NAT, bridge, or veth.
- Performance = native!
- Use cases:
- Performance sensitive applications (VOIP, gaming, streaming...)



What is port mapping in Docker?

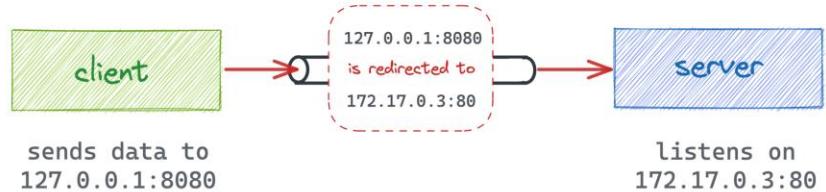
"`docker run -p 80:8080 <image>`"

Port forwarding or, as it's also called, *port mapping* is just a way to say that data addressed to one socket is redirected to another by an intermediary (for instance, by a network router or a proxy process).

"Direct" Sockets



Port Forwarding

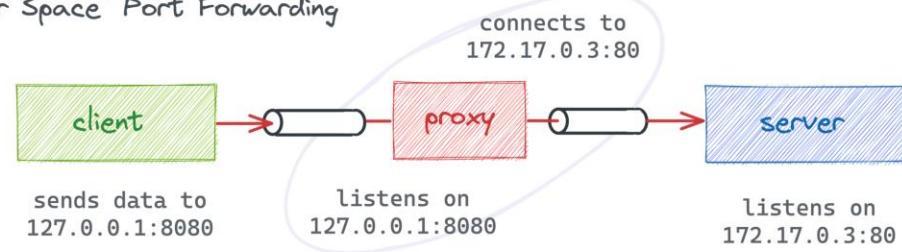


Proxy vs Kernel Port Forwarding

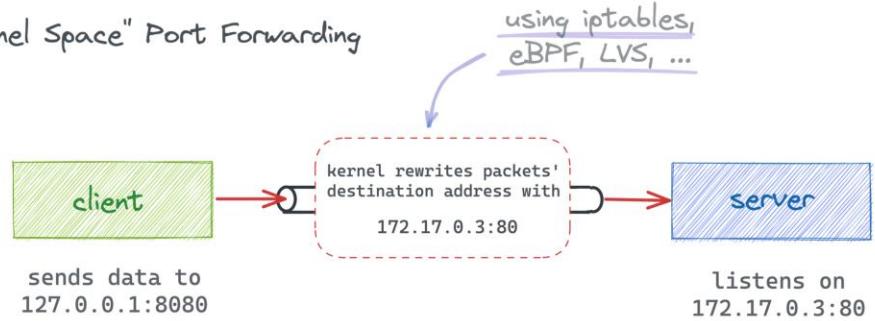
There are two common ways to redirect network data (hence, implement port forwarding):

- By sneakily modifying the destination address of packets.
- By explicitly putting a proxy between the client and the server

"User Space" Port Forwarding

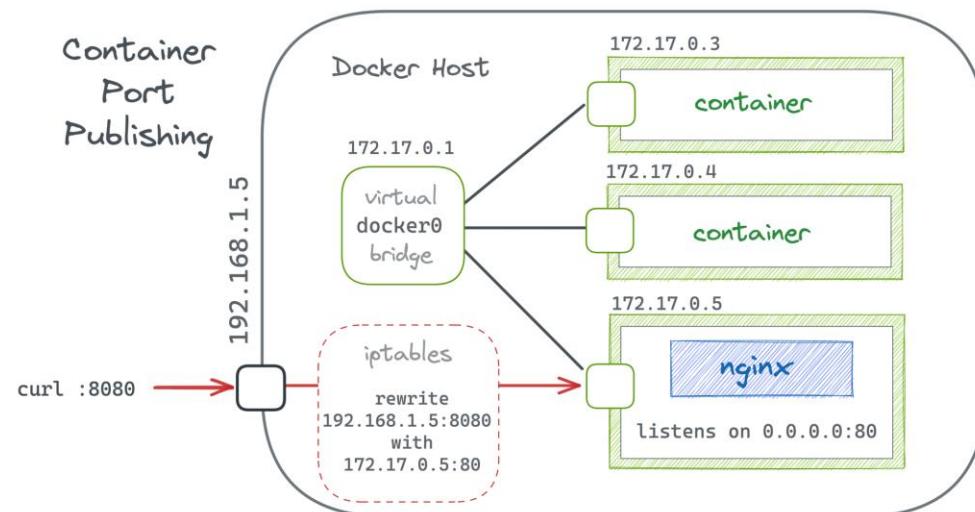


"Kernel Space" Port Forwarding



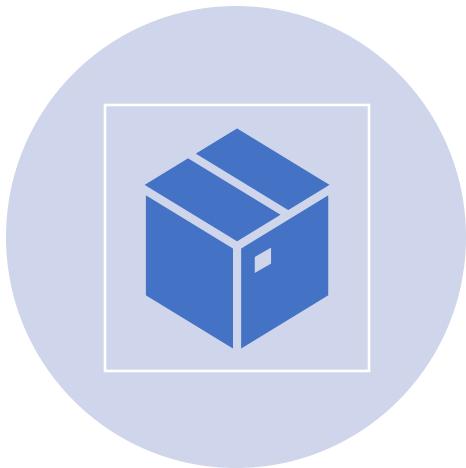
Containers and Port Forwarding

- docker run --publish 8080:80
nginx command creates a **regular port forwarding** from the host's 0.0.0.0:8080 to the container's \$CONT_IP:80
- in the case of Docker Engine, there is not much difference between **port publishing** and good old **kernel space port forwarding**

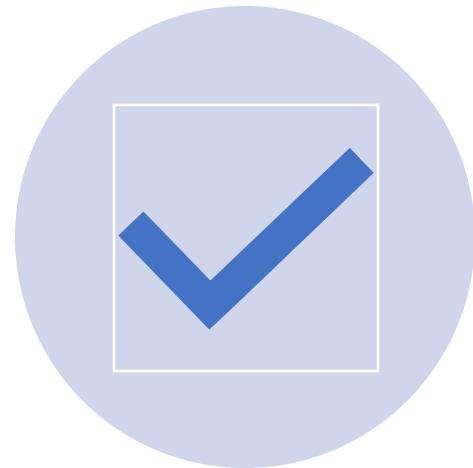


References

- <https://iximiuz.com/en/posts/docker-publish-container-ports/>
- <https://iximiuz.com/en/posts/container-networking-is-simple/>



CONTAINERS



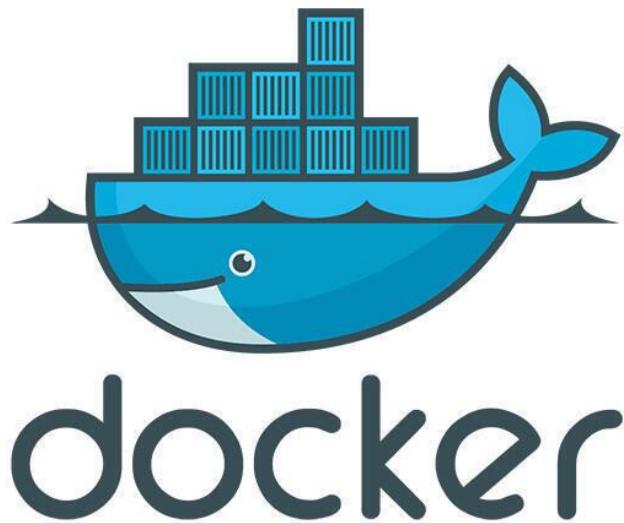
WEEK 3

Agenda

Docker Networks Lab

Docker Storage Lab

Build and Push to Amazon ECR with GitHub Actions Lab



Lab 1– Docker Networks

List the networks

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

```
# Create a new network  
$ docker network create [network name]
```

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")

Basic Connectivity and DNS

```
$ docker run -it -d --name ubuntu1 ubuntu
$ docker run -it -d --name ubuntu2 ubuntu
$ docker inspect ubuntu1 # What network is this container
using

# Try pinging ubuntu1 from the host using it's IP and DNS
$ ping 172.12.0.3
$ ping ubuntu2

# Try the same from inside the container ubuntu1
$ docker exec -it ubuntu2 /bin/sh
# ping 172.12.0.3
# ping ubuntu2
```

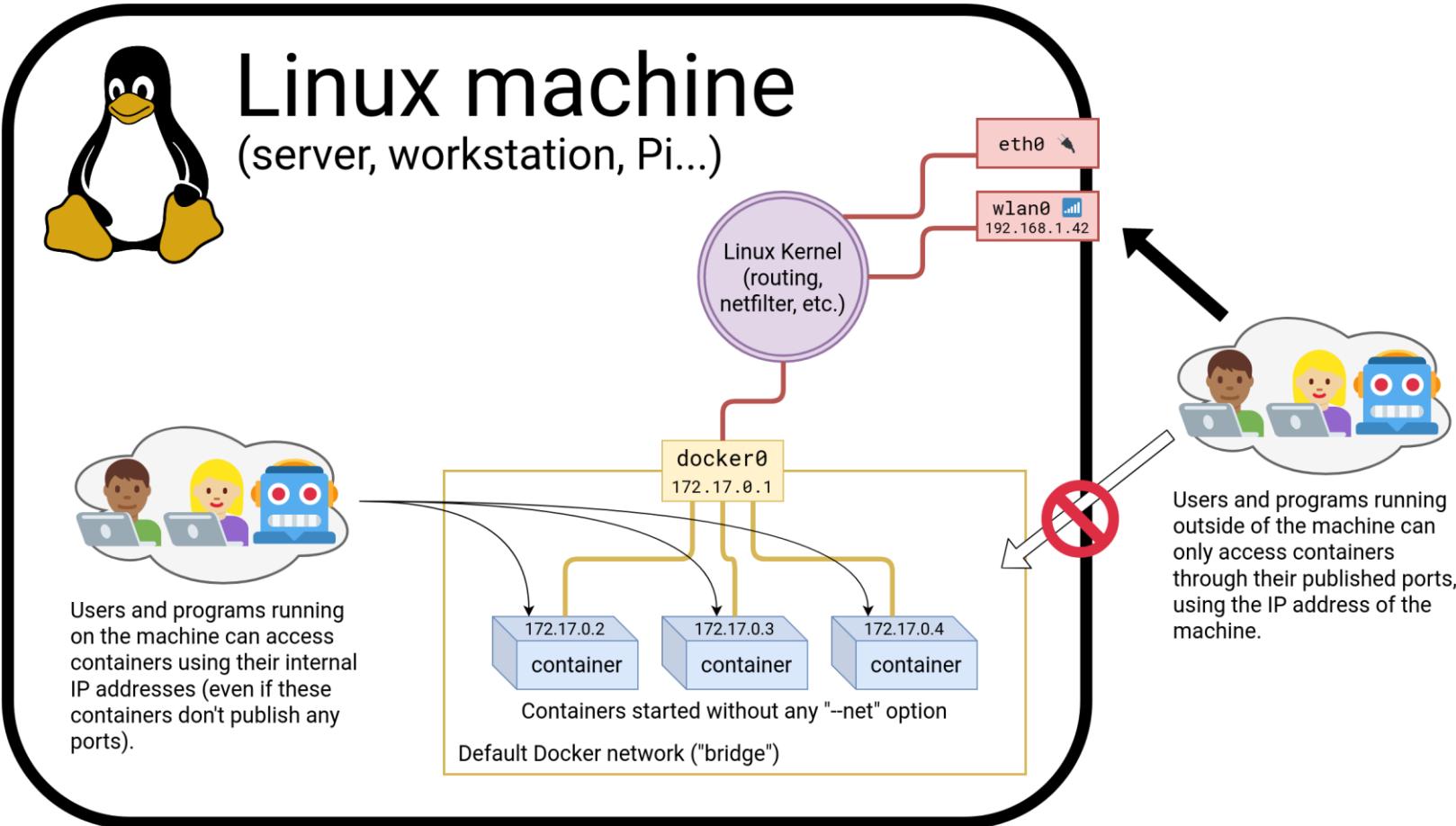
```
# Create a new network
docker network create -d bridge --subnet 182.18.0.1/24 --
gateway 182.18.0.1 new-network

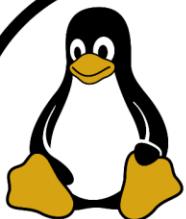
# Deploy a mysql database using the mysql:5.6 image and
name it mysql-db. Attach it to the newly created
network new-network
Set the database password to use db_pass123. The
environment variable to set is MYSQL_ROOT_PASSWORD
$ docker run -d -e MYSQL_ROOT_PASSWORD=db_pass123 --name
mysql-db --network new-network mysql:5.6

# Run a simple app and link it to DB container
$ docker run --network=new-network -e DB_Host=mysql-db -e
DB_Password=db_pass123 -p 38080:8080 --name webapp --link
mysql-db:mysql-db -d kodekloud/simple-webapp-mysql

# Can containers on a default “bridge” network connect to
containers on “new-network”?
```

Connectivity Explained



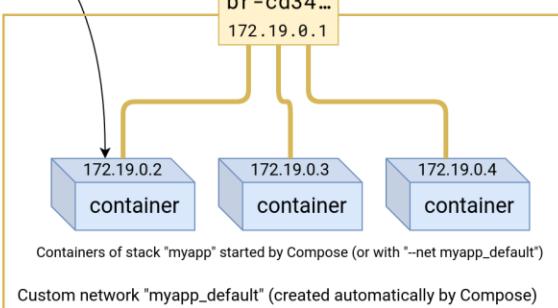
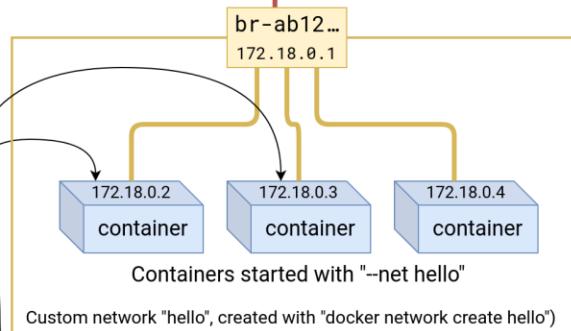
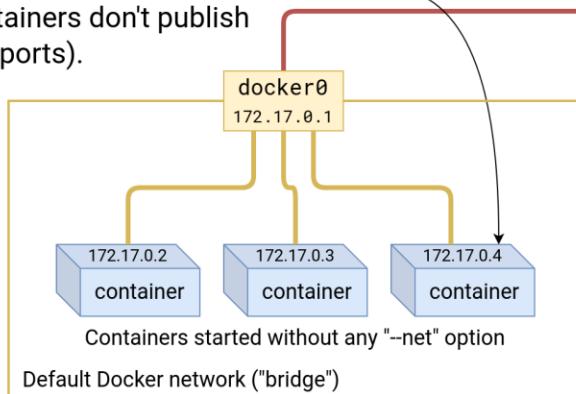


Linux machine

(server, workstation, Pi...)



Users and programs running on the machine can access containers using their internal IP addresses (even if these containers don't publish any ports).



There is *no* direct communication between containers belonging to *different* networks.

Containers must use published ports.

(Note: containers can be connected to *multiple* networks if needed!)



Users and programs running outside of the machine can only access containers through their published ports, using the IP address of the machine.

CNM vs CNI

- CNM is the model used by Docker
- Kubernetes uses a different model, architecture 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)

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 "new-network" network created before.



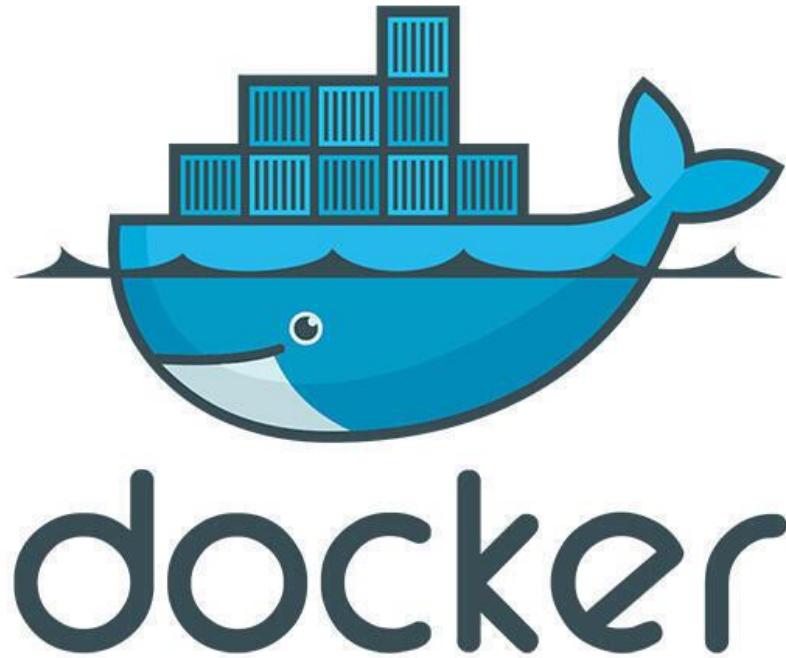
```
$ docker run --net new-network -d -P jpetazzo/trainingwheels
# Check the port that has been allocated to it:

$ docker ps -l

# If we connect to the application now, we will see an error
page
# Start the container:
$ docker run --net new-network --net-alias redis -d redis

# That container must be on the same network as the web
server.

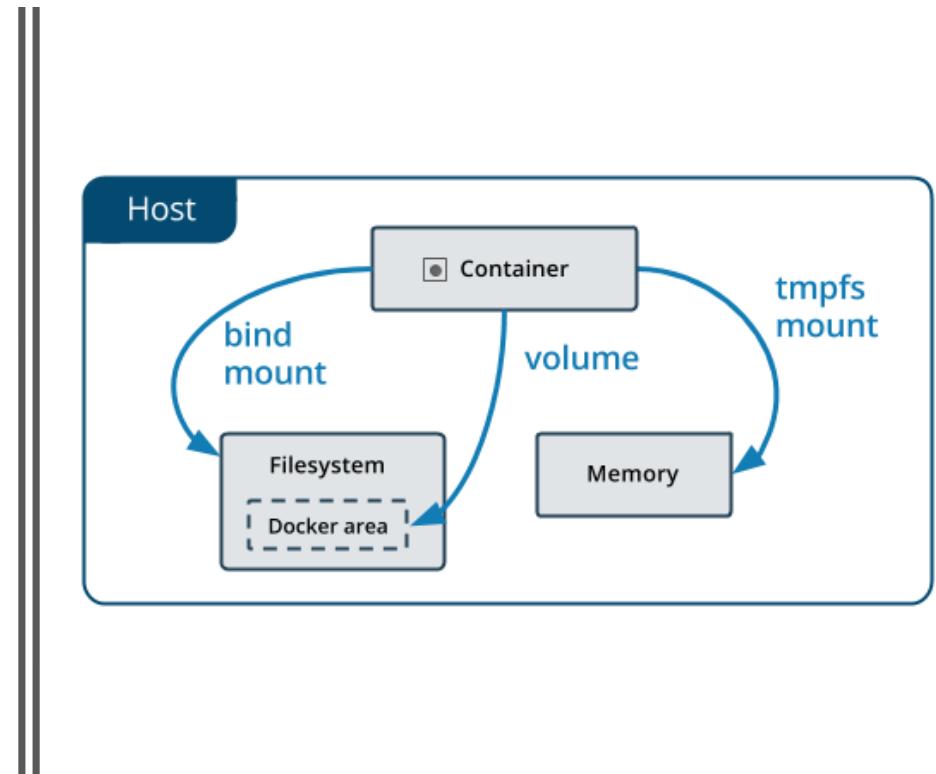
# It must have the right network alias (redis) so the
application can find it. Mind the dot!
$ docker run --net new-network --rm alpine nslookup redis.
```



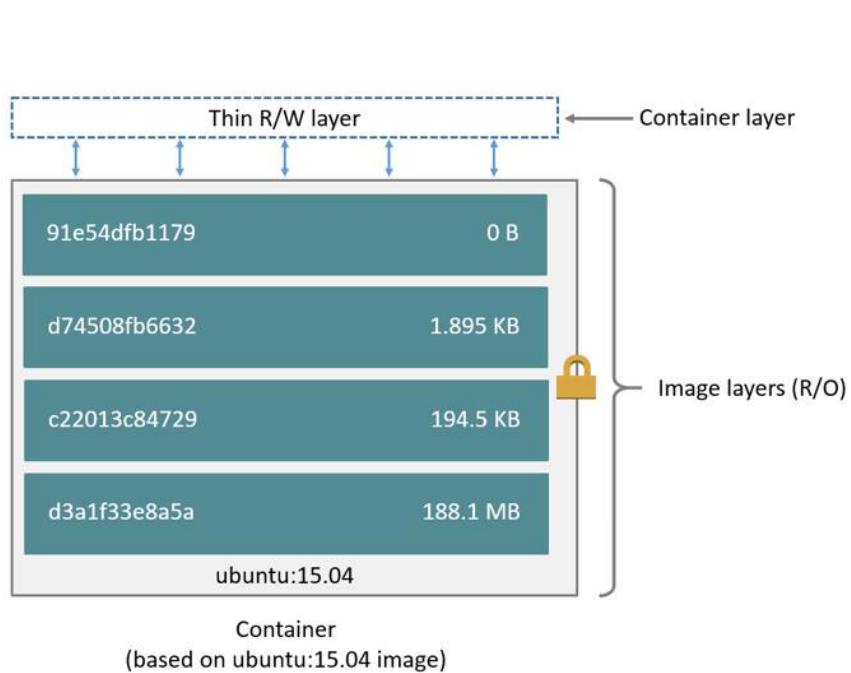
Lab 1– The End

Docker Storage

```
/var/lib/docker
└── containers
└── image
└── volumes
    └── data_volume
```



Docker Layers



Making changes to the code

Option 1:

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

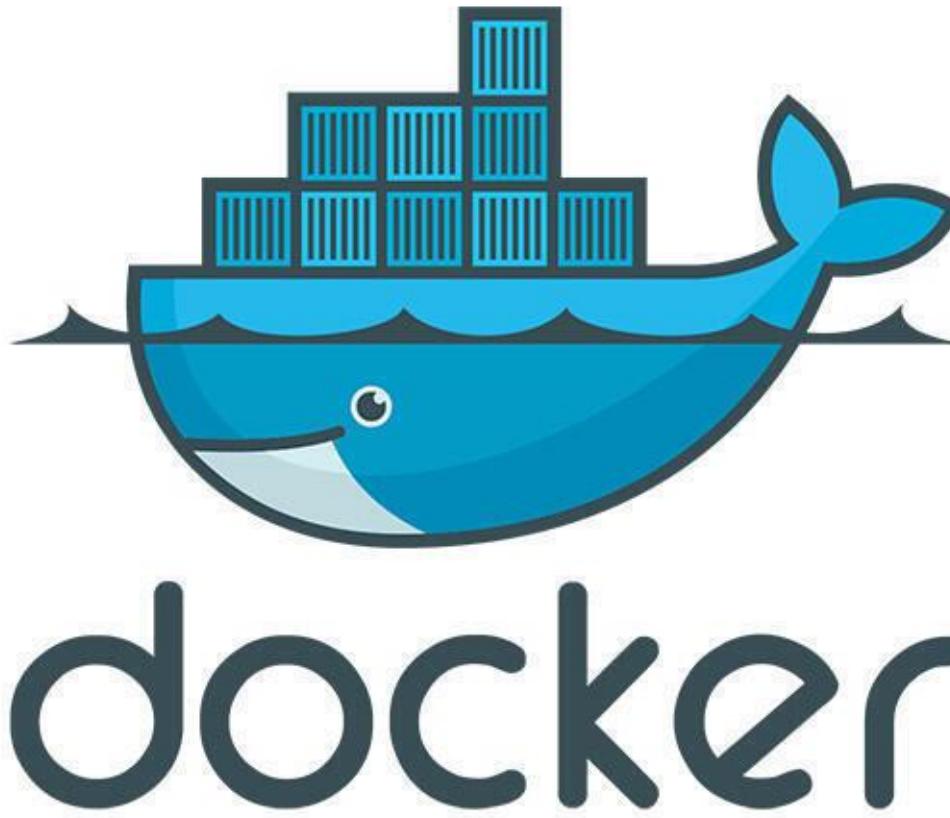
Option 2:

- 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

Lab 2 – Docker Storage



Running MySQL DB as a container

```
# Run MySQL DB as a container
$ docker run --name mysql-db -d -e MYSQL_ROOT_PASSWORD=db_clo835 mysql

# Add some data to the DB and print it out
$ docker exec -it mysql-db /bin/bash

# mysql -pdb_clo835
# create database foo;
# use foo;
# create table myTable (name VARCHAR(20), owner VARCHAR(20), species VARCHAR(20), sex CHAR(1), birth DATE, death DATE);
# INSERT INTO myTable VALUES ('Puffball','Diane','hamster','f','1999-03-30',NULL);
# select * from myTable;

$ docker exec mysql-db mysql -pdb_clo835 -e 'use foo; select * from myTable'

# Stop and delete the container
$ docker stop <container id>
$ docker rm  <container id>
# Explain: What happened to the data when we removed the container and why?
```

Running MySQL DB as a Container with a mounted volume

```
# Add persistent volume binding and re-run
$ docker run --name mysql-db -d -e MYSQL_ROOT_PASSWORD=db_pass123 -v
/opt/data:/var/lib/mysql mysql

# Create DB and populate it with some entries
$ docker exec mysql-db /bin/sh
..

# Stop and delete the container
$ docker stop <container id>
$ docker rm  <container id>

# Explain: What happened to the data and why?
$ docker run --name mysql-db -d --env MYSQL_ROOT_PASSWORD=db_pass123 -v
/opt/data:/var/lib/mysql mysql
$ docker exec mysql-db mysql -pdb_clo835 -e 'use foo; select * from myTable'
```

Running MySQL DB as a Container with a mounted volume

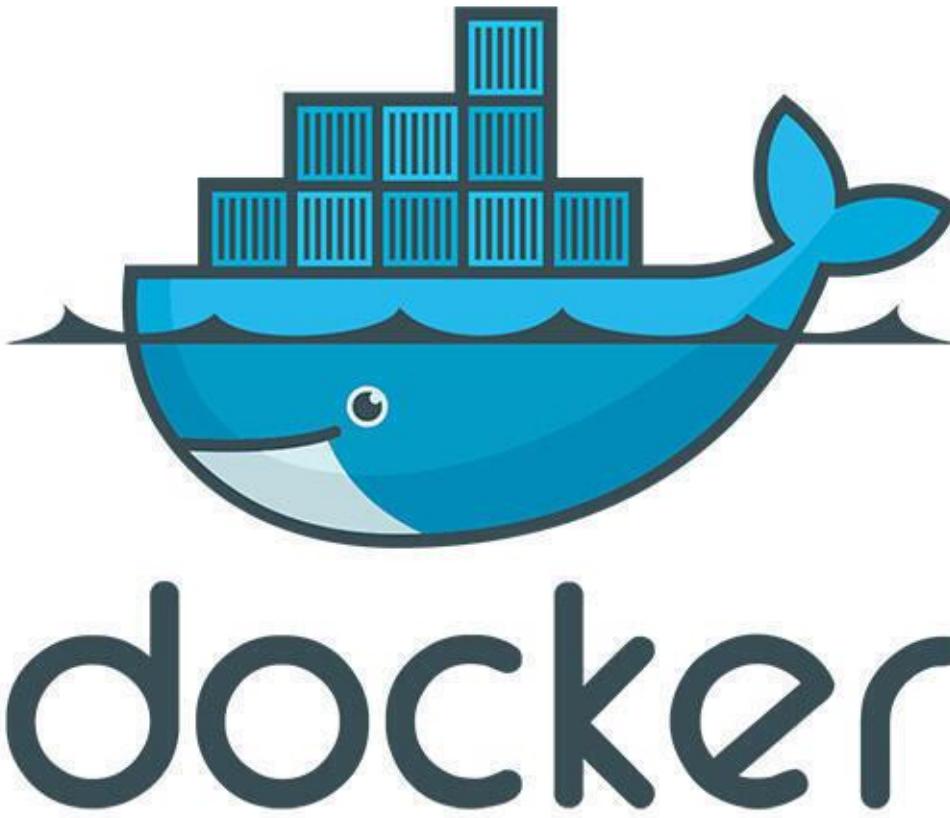
The `-v /path/on/host:/path/in/container` syntax is the "old" syntax

The modern syntax looks like this:

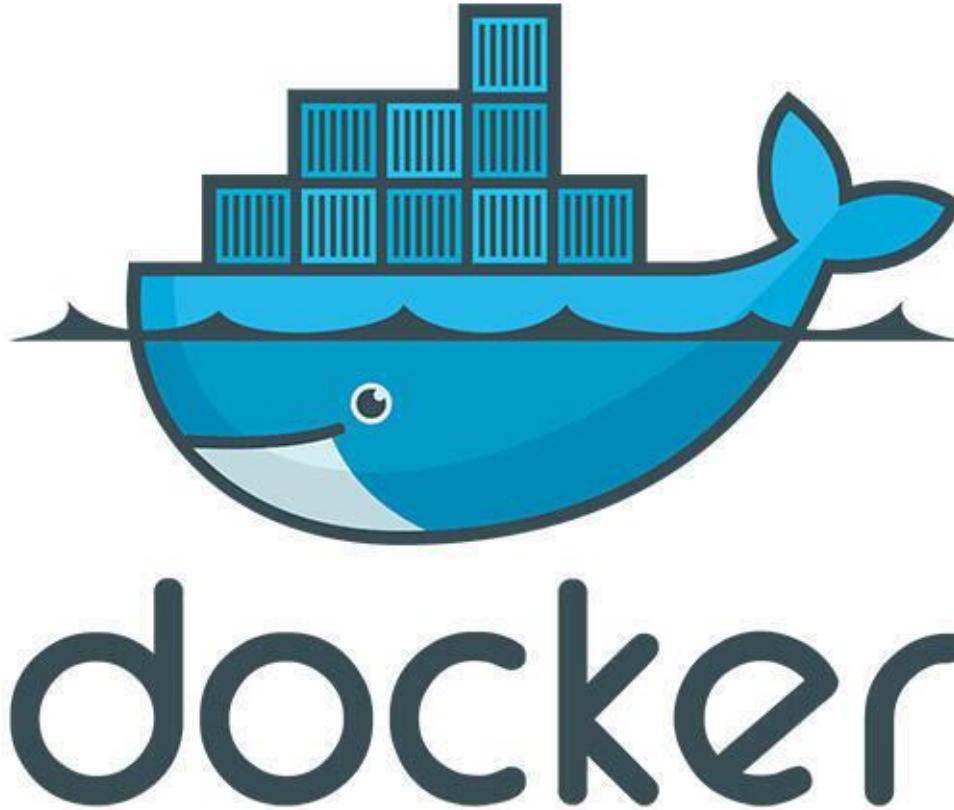
- `--mount type=bind,source=/path/on/host,target=/path/in/container`
- `--mount` is more explicit, but `-v` is quicker to type
- `--mount` supports all mount types; `-v` doesn't support tmpfs mounts
- `--mount` fails if the path on the host doesn't exist; `-v` creates it
- With the new syntax, our command becomes:

```
$ docker run --mount=type=bind,source=/opt/data,target=/var/lib/mysql -d mysql-db
```

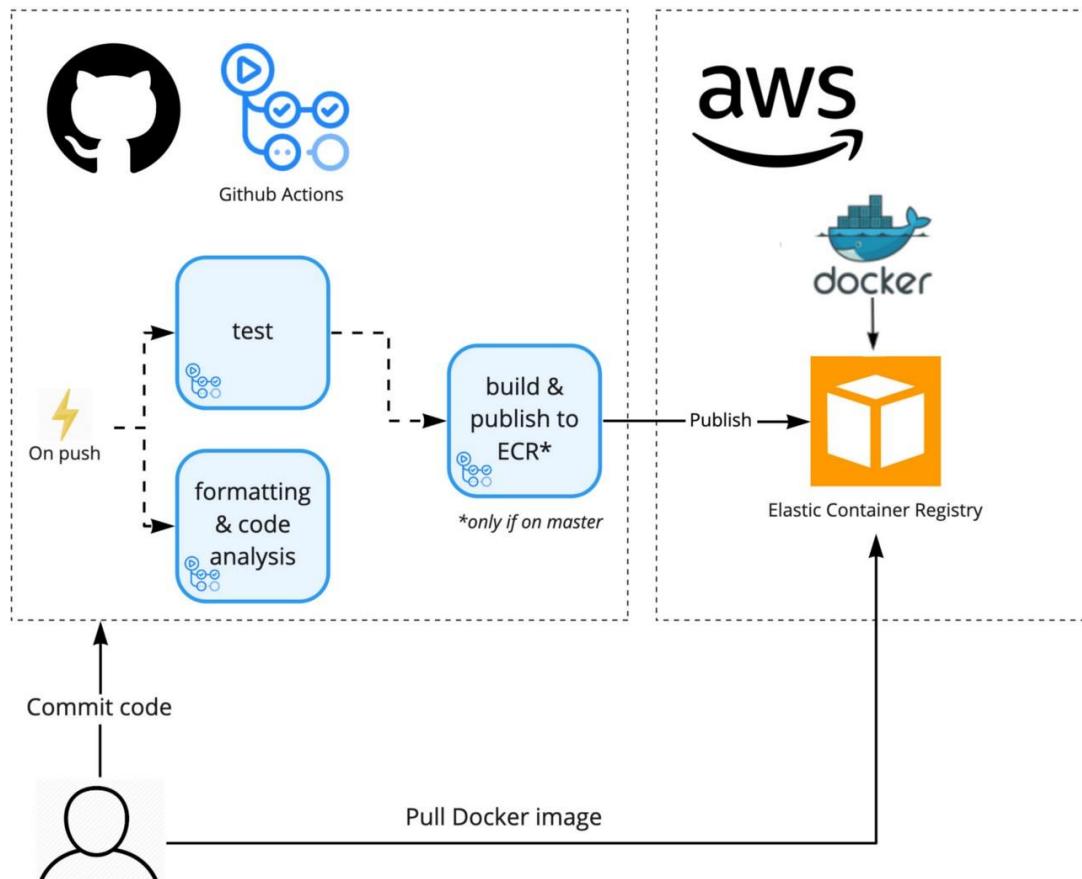
Lab 2 – The End



Lab 3 – Build Docker Image and Publish to Amazon ECR with GitHub Actions



Build Docker Images with GitHub Actions



Create a Fork of the Application Repo

The screenshot shows the GitHub interface for creating a fork of the 'kubernetes-in-action' repository. At the top, there's a navigation bar with links for Issues (11), Pull requests (11), Actions, Projects, Wiki, Security, and Insights. Below the navigation, a heading says 'Create a new fork'. A subtext explains that a fork is a copy of a repository, allowing experimentation without affecting the original project, with a link to 'View existing forks'. The 'Owner' field is set to 'igelman13', and the 'Repository name' field contains 'kubernetes-in-action' with a green checkmark. A note below states that forks are named like their parent by default but can be customized. An optional 'Description' field contains 'Code from the Kubernetes in Action book'. A note at the bottom indicates the fork is being created in the user's personal account. A prominent green 'Create fork' button is at the bottom.

Fork the repo

<https://github.com/luksa/kubernetes-in-action/fork>

Create a Repository for Your Images

```
# Create an Amazon ECR repository using CLI from your AWS Academy or Cloud9 Terminal
```

```
$ aws ecr create-repository --repository-name clo835-week3
```

```
# Retrieve temporary credentials from AWS Academy. Select AWS Details and then Click on "Show" under AWS CLI.
```

The screenshot shows the AWS Academy interface with the following details:

- Header: AWS • Used \$2.9 of \$100, 03:54, Start Lab, End Lab, AWS Details, Readme, Reset, Close.
- Section: 3 Service Unavailable
- Message: "No server is available to handle this request."
- Section: Cloud Access
- Text: "AWS CLI: Copy and paste the following into ~/.aws/credentials"
- Code block (highlighted): [default]
aws_access_key_id=ASIA4TLXENSKIR6K37UT
aws_secret_access_key=W93tFnPDPLGp2x0BdiuX6gAriyZC1xKcfocSLP8
aws_session_token=FwoGZXIVYXdzEHsoDHHlkRbocwXDzFvh1CK9AaGh1J9Hz0XcjZQ1bcw76vJFFWkoy0B1Oj3kYT98z5gpNMf0monQUu0LLG5zp0f9L2mWr0bd1YFN+1tYapi6mVjb/HZyNTMyFUHX9zT8NF/3znsMgmQ3Wfr7lsHW60CcivQv0Pw8g789Dc+wciCibCob+oviqta1KFFqv4v4d5ZnCbsc+AR18CH778tDQR1tBh2zr0wGN0l5/VgddcxZSpSdssOWW7PvmoAKvqsXNNmtxW06UEb4jTeW/RSe5iav6uUBjIt1DVVddqPefg6Ps18CPN6aPd1ebvYCl+xMaboeJ7hA1zD3P2qhncyp07o4nqh

Update Your Repo with AWS Credentials

In your repository, select Settings=>Secrets, click on "New repository secrets" and add the secrets below.

The screenshot shows the 'Repository secrets' section of a GitHub repository settings page. It lists three secrets:

Secret	Last Updated	Action	Action
AWS_ACCESS_KEY_ID	Updated 4 minutes ago	Update	Remove
AWS_SECRET_ACCESS_KEY	Updated 5 minutes ago	Update	Remove
AWS_SESSION_TOKEN	Updated 5 minutes ago	Update	Remove

Create GitHub Actions Workflow

The screenshot shows a GitHub repository page for `igeiman13/kubernetes-in-action`, which is public and forked from `luksa/kubernetes-in-action`. The page features a navigation bar with links for Code, Pull requests, Actions (which is highlighted with an orange underline), Projects, Wiki, and Issues. Below the navigation bar, there's a large heading "Get started with GitHub Actions" followed by the text "Build, test, and deploy your code. Make code reviews, branch management". A blue link "Skip this and set up a workflow yourself →" is present, with a red arrow pointing to it. At the bottom, there's a search bar labeled "Search workflows".

In the forked repo, click on Actions and select "set up workflow yourself"

Copy the workflow from [the GitHub link](#)

Verify that the Docker Image is available in Registry

The screenshot shows the GitHub Actions interface for a repository named 'kubernetes-in-action'. The 'Actions' tab is selected. A single job, 'Update push_to_ecr.yml Deploy to ECR #8', is shown with a green checkmark indicating success. The 'Build Image' step is expanded, showing the following steps:

- > Set up job
- > Check out code
- > Login to Amazon ECR
- > Build, tag, and push image to Amazon ECR
- > Post Login to Amazon ECR
- > Post Check out code
- > Complete job

The status message indicates the build succeeded 16 seconds ago in 40s.

The screenshot shows the Amazon ECR console. The user is viewing the 'Images' section for the repository 'clo835-week3'. One image is listed:

Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest
v1.1	Image	May 22, 2022, 23:24:10 (UTC-04)	263.67	Copy URI	sha256:b826f79932f231...

Run the container in Cloud9

```
# Export environment variable
$ export ECR=866222632084.dkr.ecr.us-east-1.amazonaws.com/clo835-week3

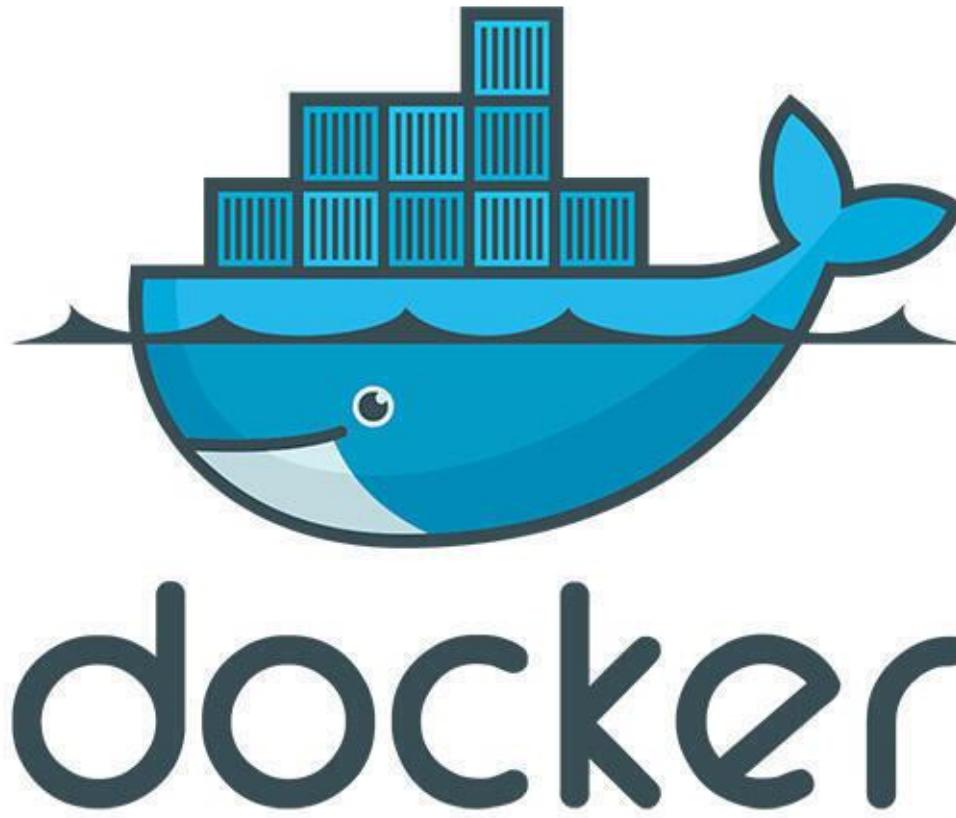
# Log into docker registry to get permissions to pull an image
$ aws ecr get-login-password --region us-east-1 | docker login -u AWS ${ECR} --
password-stdin

# Start the webserver
$ docker run -d -p 80:8080 866222632084.dkr.ecr.us-east-1.amazonaws.com/clo835-
week3:v1.1

# Verify that the application is running
$ curl localhost

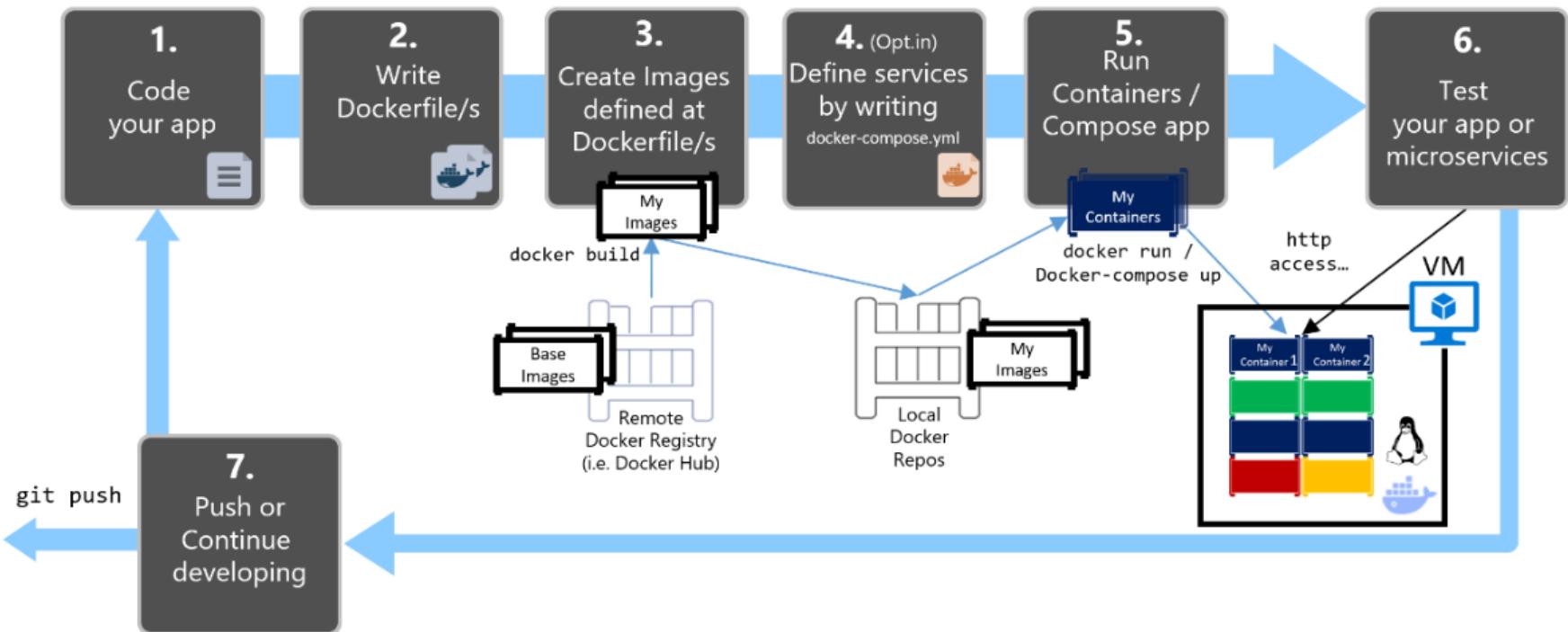
# Explain: how would you run two instances of the same application on the Cloud9
machine? Update the message printed by the application, build a new image and show it
in the browser. The new and the old version of the application should run
simultaneously.
```

Lab 3 – The End



Lifecycle

Inner-Loop development workflow for Docker apps



Docker Architecture: Namespaces



Docker takes advantage of Linux namespaces to provide isolated workspace we call "container"

Some namespaces on Linux:

PID namespace for processes isolation (PID: Process ID)

Net namespace for network isolations, manages network interfaces

IPC namespace: inter-process communication

Mnt namespace to manage mount points

Docker Architecture: Control Groups



Docker engine uses another Linux technology called cgroups or control groups



Cgroups are responsible for limiting resources used by the container



Cgroups ensures that containers are good multi-tenant citizens on the host



Cgroups allow Docker engine to share available VM resources between containers and set limits and constraints, if required

Docker Engine

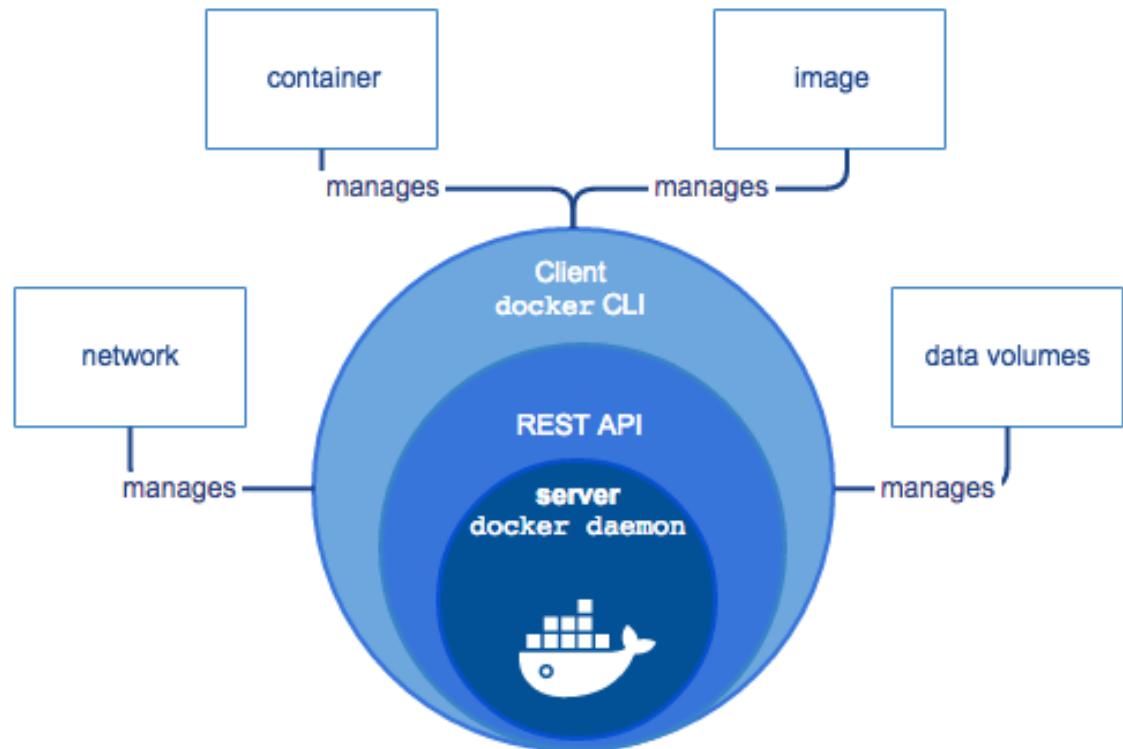
Docker Engine is a client server application with the main components below

A server which is a long-running process called Docker daemon

REST API that defines interface that docker client can use to talk to the daemon and instruct it what to do

CLI client

Docker Engine



Docker commands

Container management commands

command	description
<code>docker create image [command]</code> <code>docker run image [command]</code>	create the container = <code>create + start</code>
<code>docker start container...</code>	start the container
<code>docker stop container...</code>	graceful ² stop
<code>docker kill container...</code>	kill (SIGKILL) the container
<code>docker restart container...</code>	= <code>stop + start</code>
<code>docker pause container...</code>	suspend the container
<code>docker unpause container...</code>	resume the container
<code>docker rm [-f³] container...</code>	destroy the container

²send SIGTERM to the main process + SIGKILL 10 seconds later

³-f allows removing running containers (= `docker kill + docker rm`)

Useful links

<https://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html>

<https://ecs-cats-dogs.workshop.aws/en/ecr/docker.html>

[https://blog.aquasec.com/docker-security-best-practices? _ga=2.100580638.1058616926.1651978295-1119178887.1645428243](https://blog.aquasec.com/docker-security-best-practices?_ga=2.100580638.1058616926.1651978295-1119178887.1645428243)

<https://github.com/luksa/kubernetes-in-action>

<https://towardsaws.com/build-push-docker-image-to-aws-ecr-using-github-actions-8396888a8f9e>

<https://iximiuz.com/en/posts/container-networking-is-simple/>

Introduction to Kubernetes

Week 4

Agenda



Business need in containers orchestration tool



Kubernetes timeline



Kubernetes High Level Architecture



Setup dev K8s cluster



Running "Hello World" application in K8s

The Need in Containers Orchestration Tool

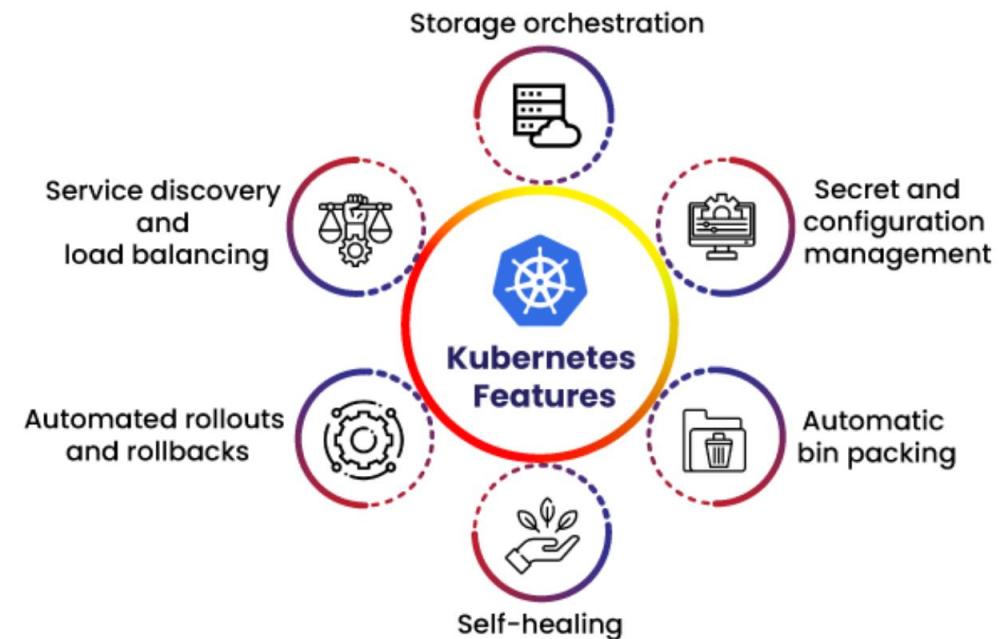
Containers take care of packaging, running and sharing application images

The topics below are not covered by containerization:

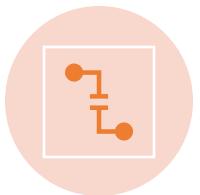
- Application components synchronization (web, app, db, messaging .. or 1000s of components in case of Google applications and services)
- Application components capacity scaling
- Underlying host capacity scaling
- Support of high availability
- Support of application SDLC

What is K8s?

- Kubernetes is a container management system
- It runs and manages containerized applications on a cluster



Main Advantages of using K8s



Application is now highly available - hardware failures do not bring your application down



Health checking and self-healing



The user traffic is load balanced across the various containers.



Application scales in a matter of seconds

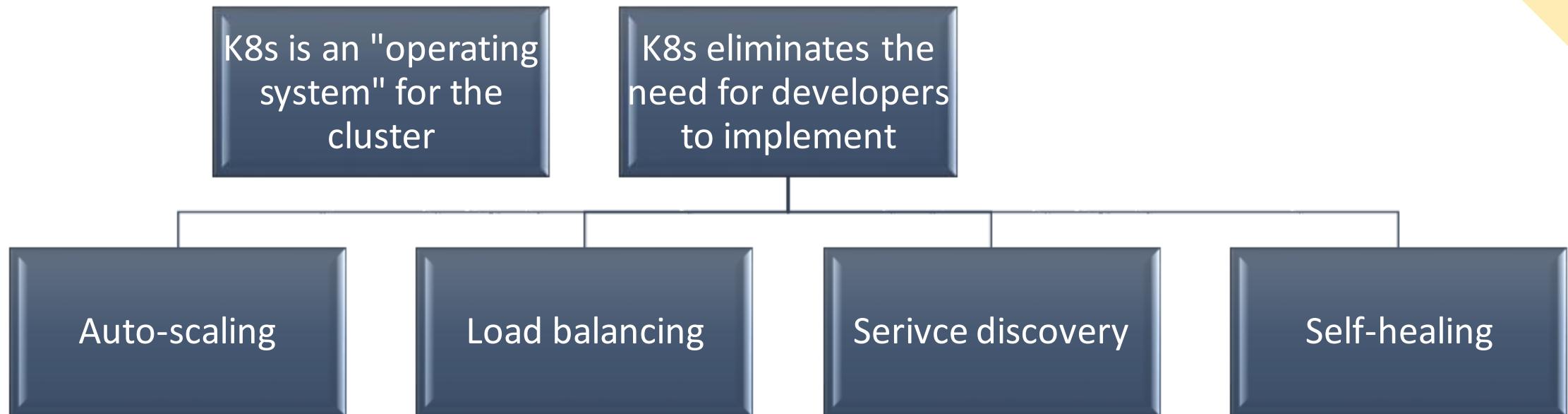


The number of nodes scales up/down without affecting application availability



The desired state defined in a declarative manner with YAML

From the Developers Perspective



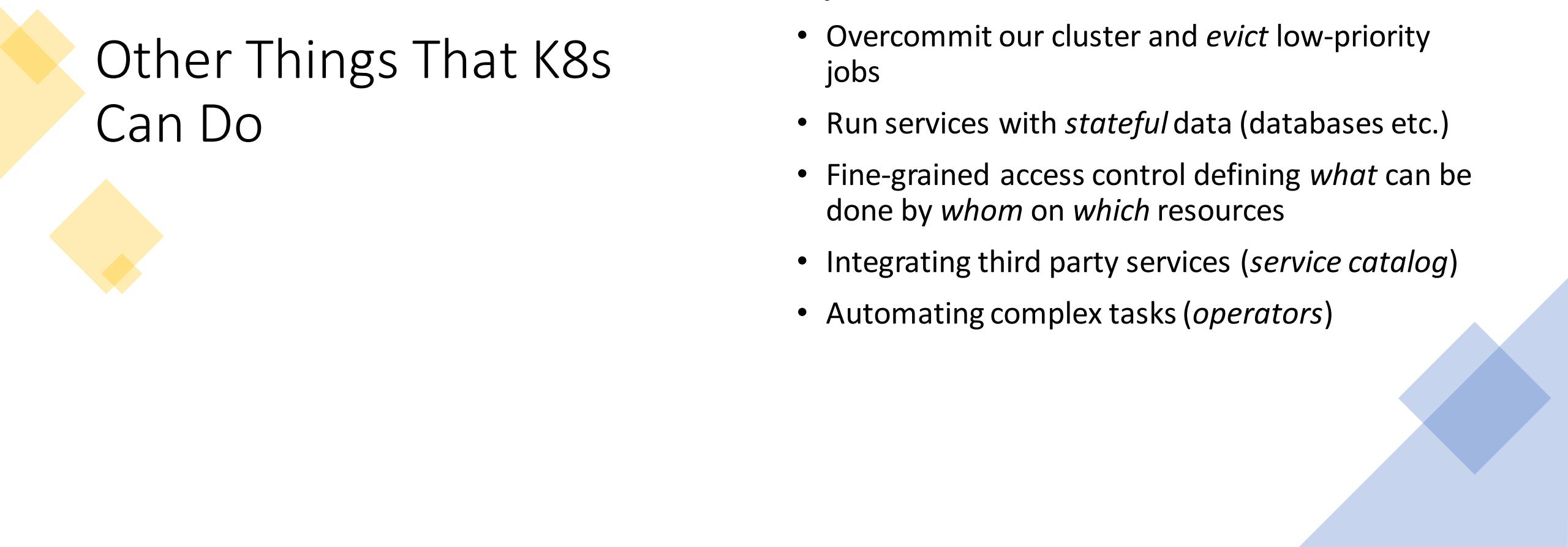
From the Operations Perspective

Better resources utilization

Application re-allocation in case of failing nodes

Cluster and application auto-scaling

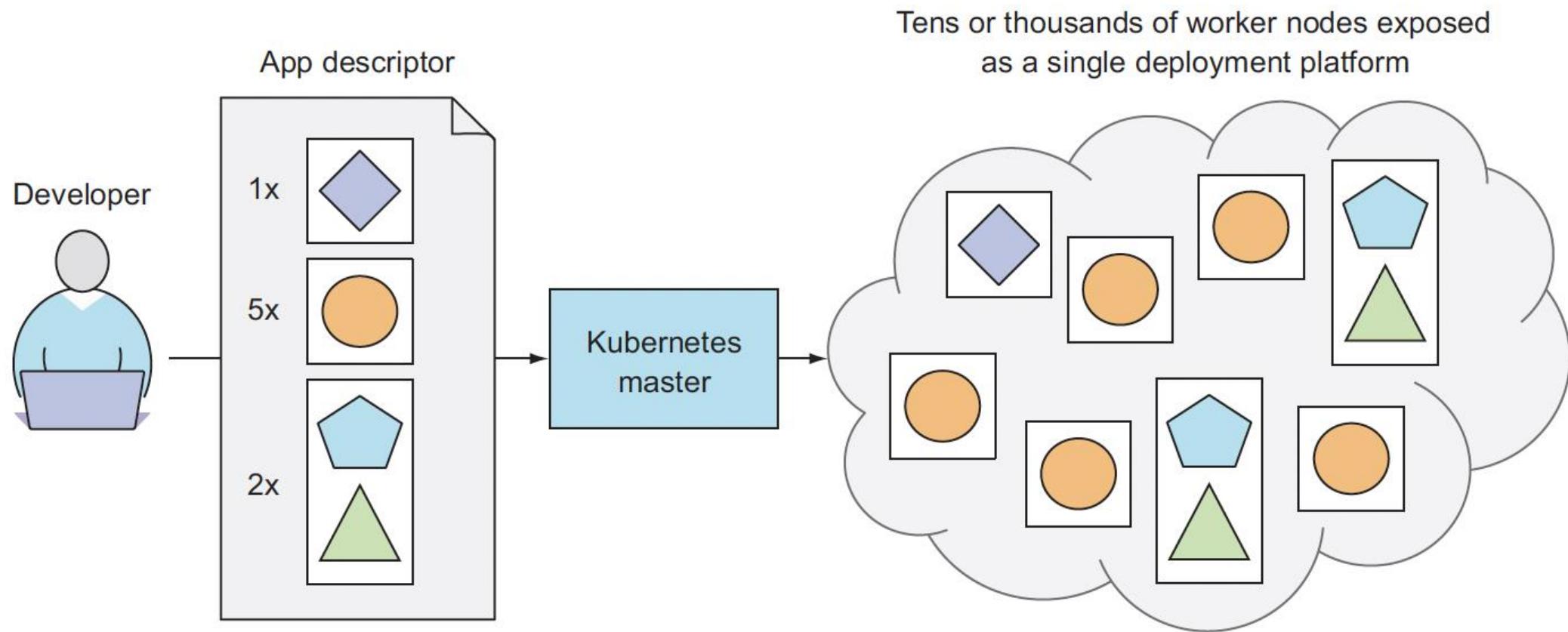
K8s maintains the desired state and will keep the applications running



Other Things That K8s Can Do

- Basic autoscaling
- Blue/green deployment, canary deployment
- Long running services, but also batch (one-off) jobs
- Overcommit our cluster and *evict* low-priority jobs
- Run services with *stateful* data (databases etc.)
- Fine-grained access control defining *what* can be done by *whom* on *which* resources
- Integrating third party services (*service catalog*)
- Automating complex tasks (*operators*)

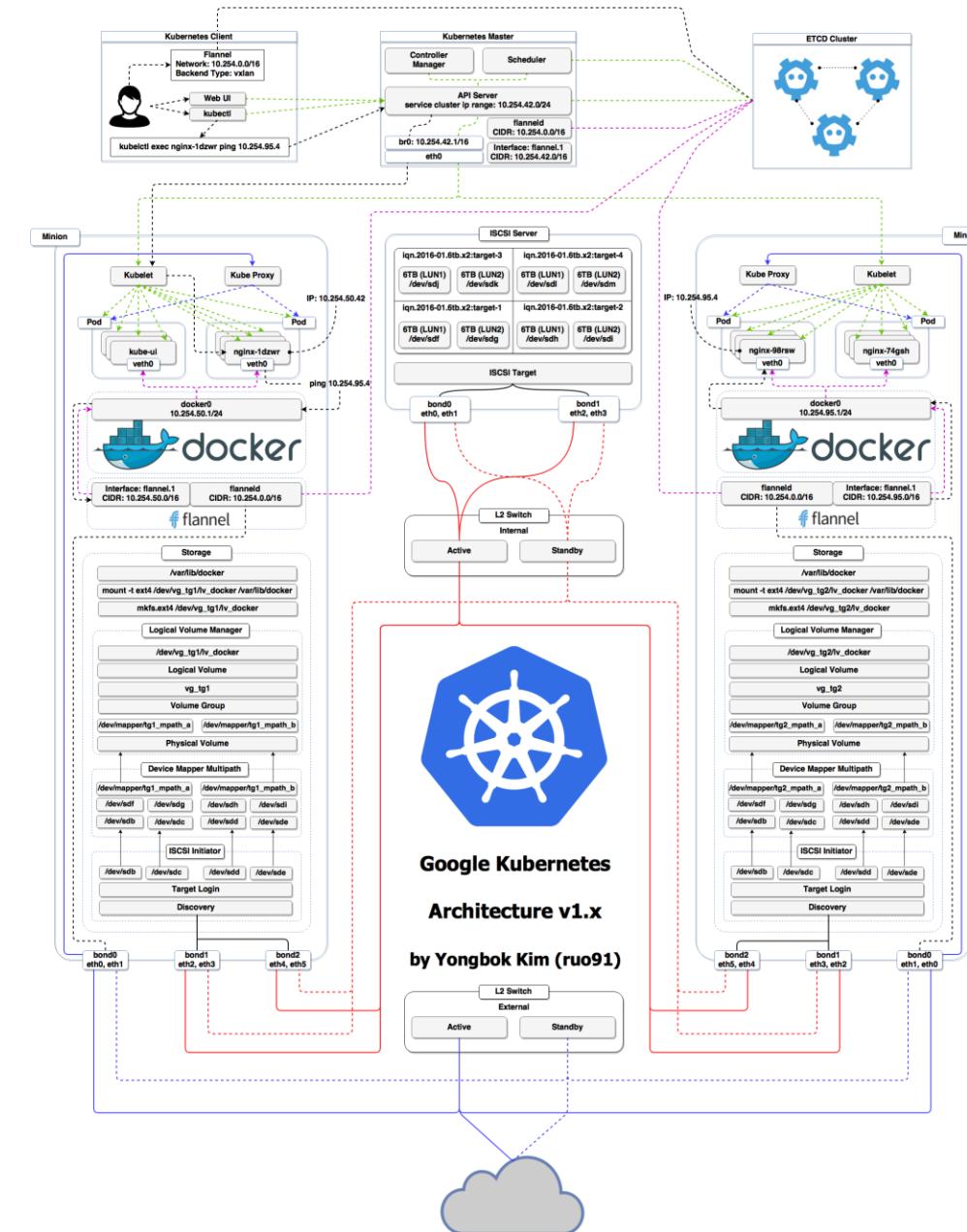
K8s Architecture from 10,000 Feet



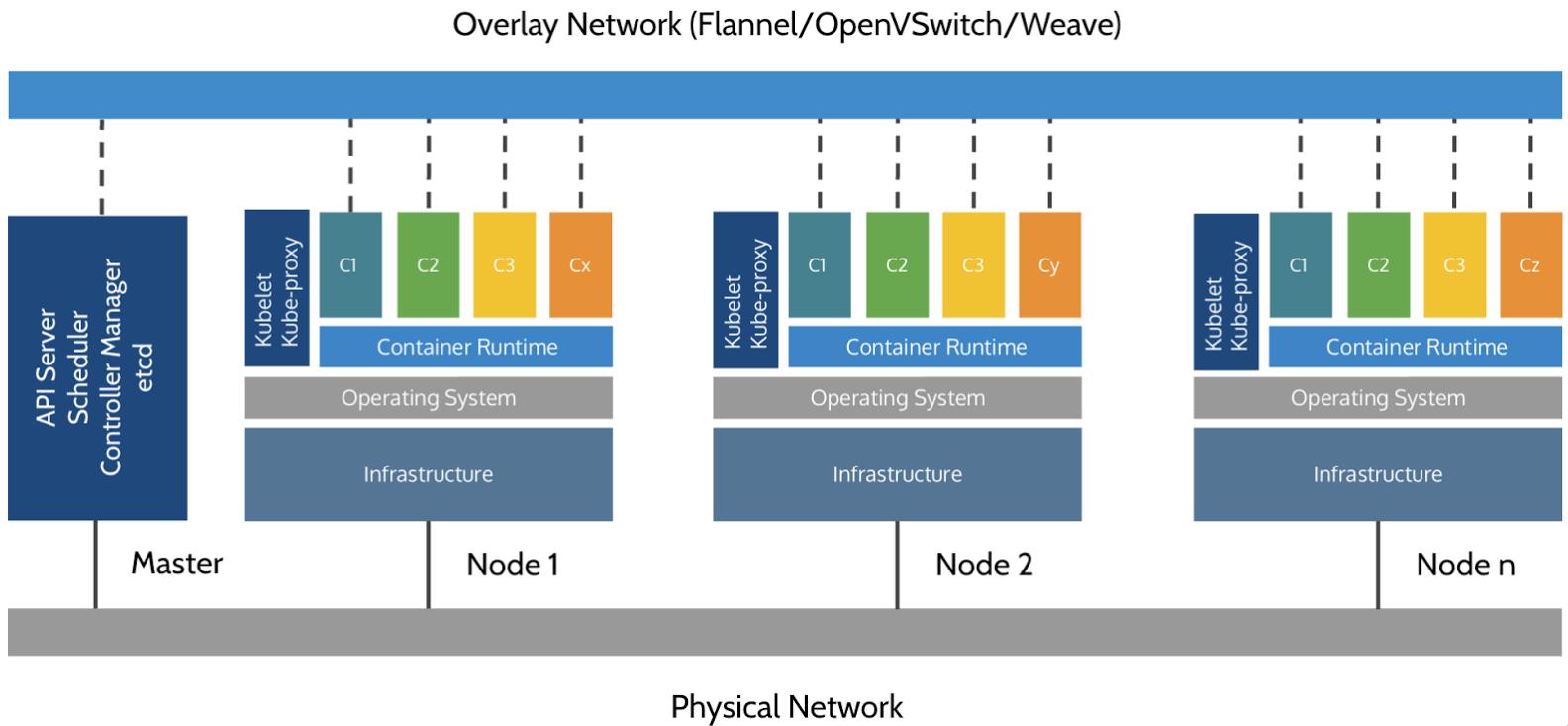
K8s Architecture

Master Nodes – Control plane

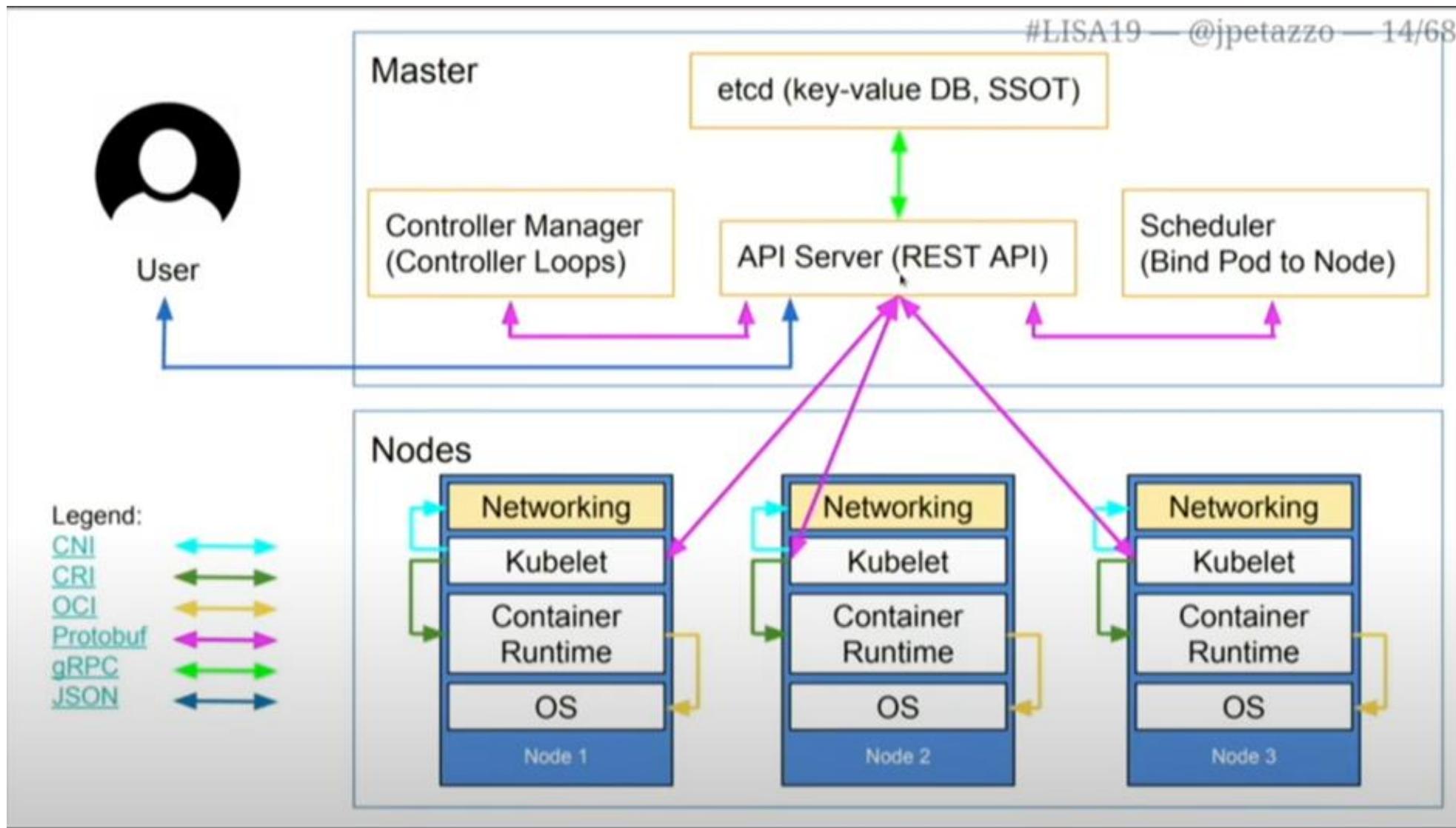
Worker Nodes (or Nodes) - Data plane

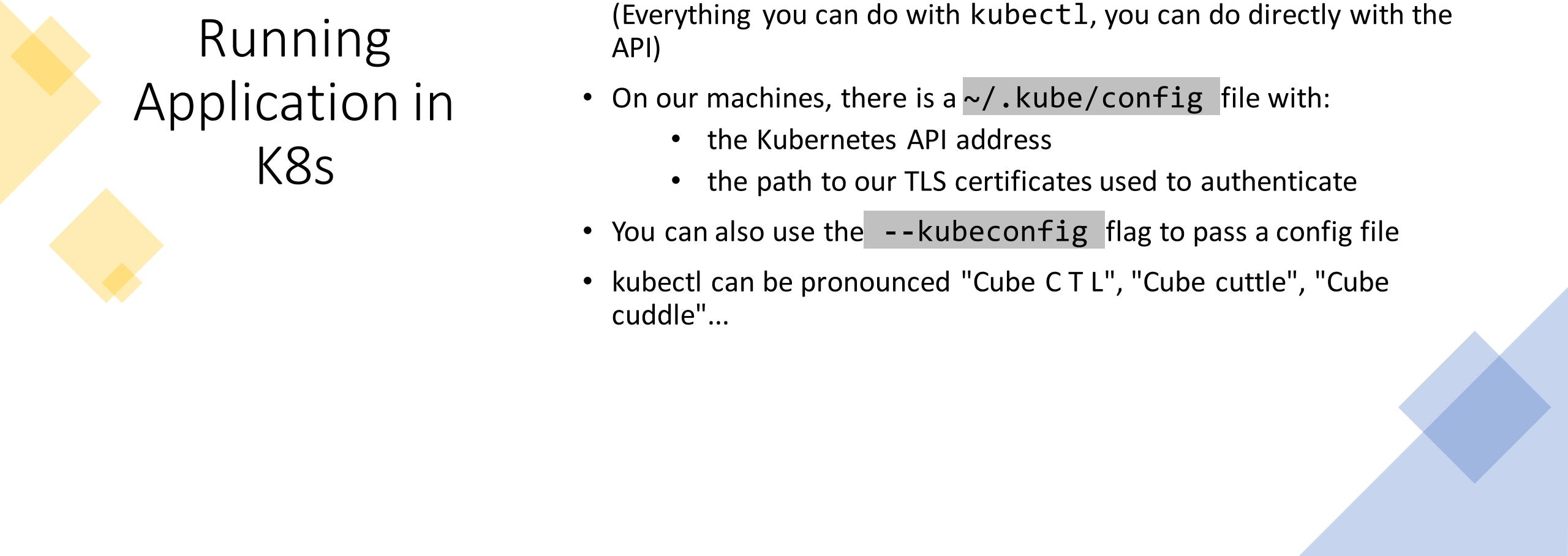


Kidding :)
Basic K8s
Architecture
Diagram



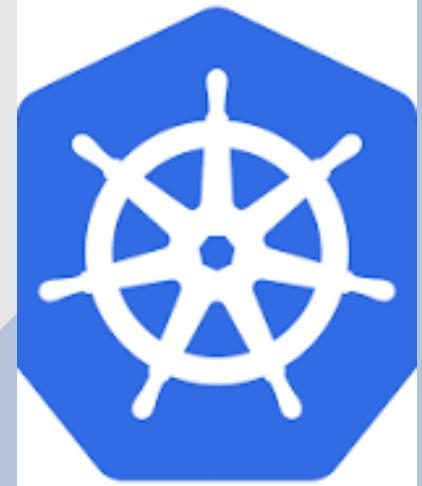
Components and Interfaces in K8s Cluster





Running Application in K8s

- `kubectl` is (almost) the only tool we'll need to talk to Kubernetes
- It is a rich CLI tool around the Kubernetes API
(Everything you can do with `kubectl`, you can do directly with the API)
- On our machines, there is a `~/.kube/config` file with:
 - the Kubernetes API address
 - the path to our TLS certificates used to authenticate
- You can also use the `--kubeconfig` flag to pass a config file
- `kubectl` can be pronounced "Cube C T L", "Cube cuttle", "Cube cuddle"...



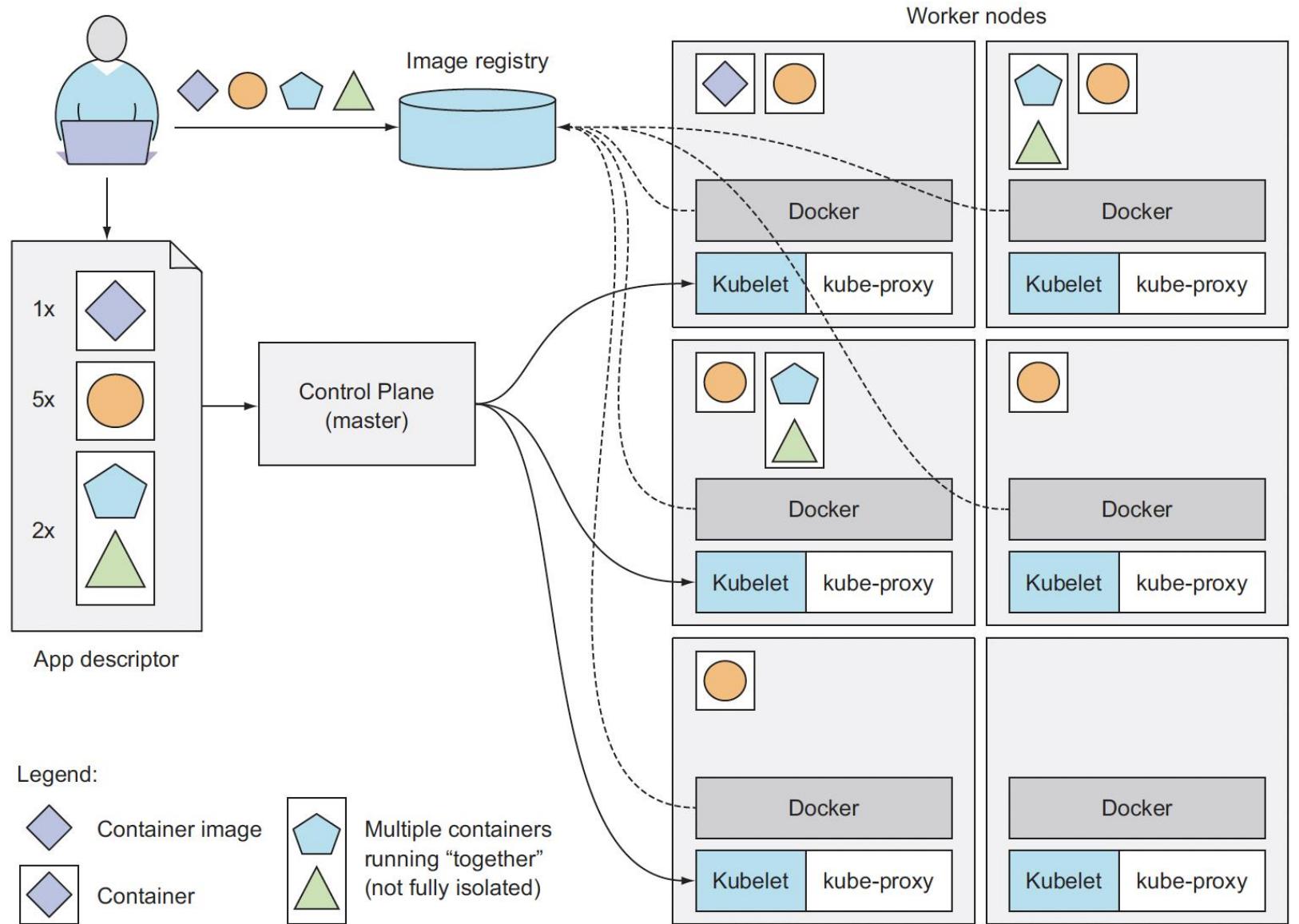
Deploying Application in K8s - Prerequisites

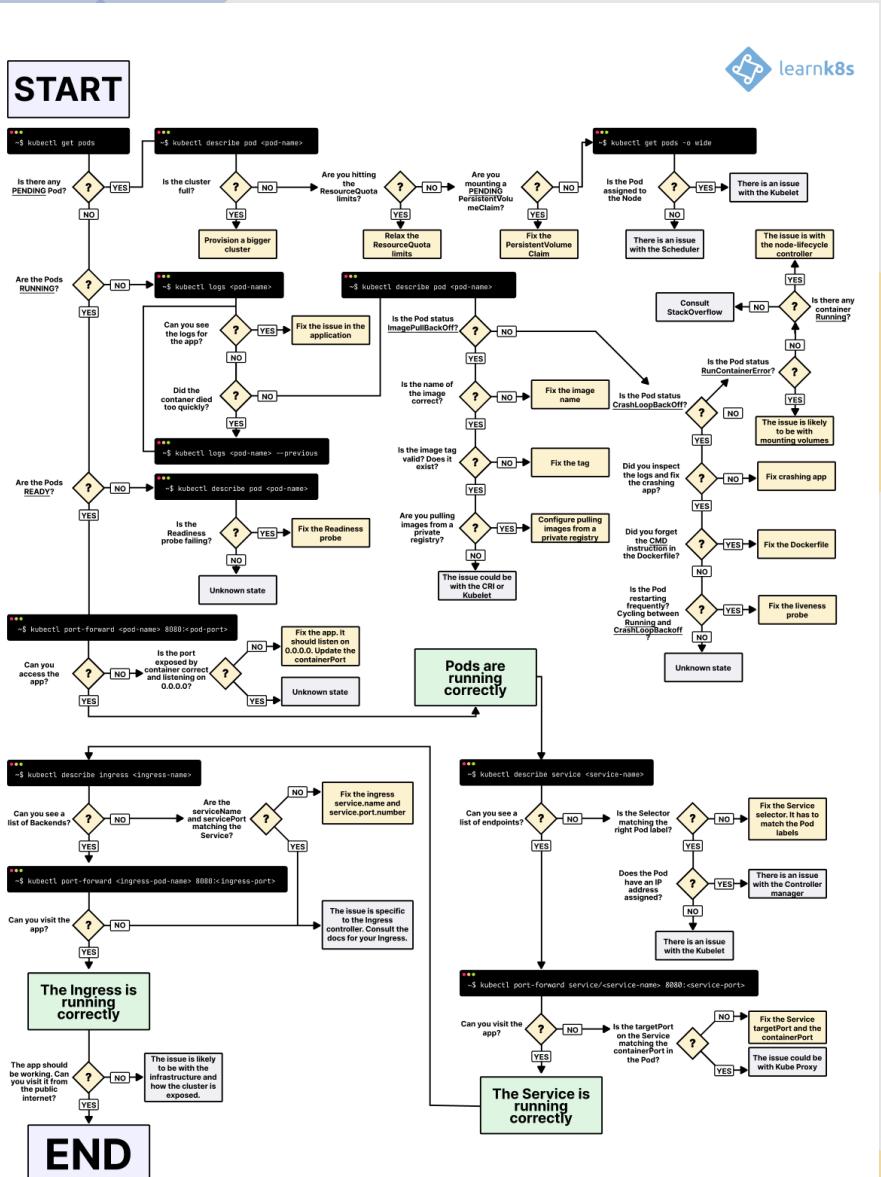


Running Application in K8s, Continued

- Package and publish your app in Docker registry
- Post a description of your app to K8s API
 - The description will indicate the location of the application images
 - How these images are related to each other
 - The resources requirements
 - The number of replicas to run
 - Internally or externally discoverable services

Application Deployment Flow in K8s





Ready for a Kubernetes deployment

K8s Network Model

- TL,DR:
- *Our cluster (nodes and pods) is one big flat IP network.*
- In detail:
 - all nodes must be able to reach each other, without NAT
 - all pods must be able to reach each other, without NAT
 - pods and nodes must be able to reach each other, without NAT
 - each pod is aware of its IP address (no NAT)
- Kubernetes doesn't mandate any particular implementation



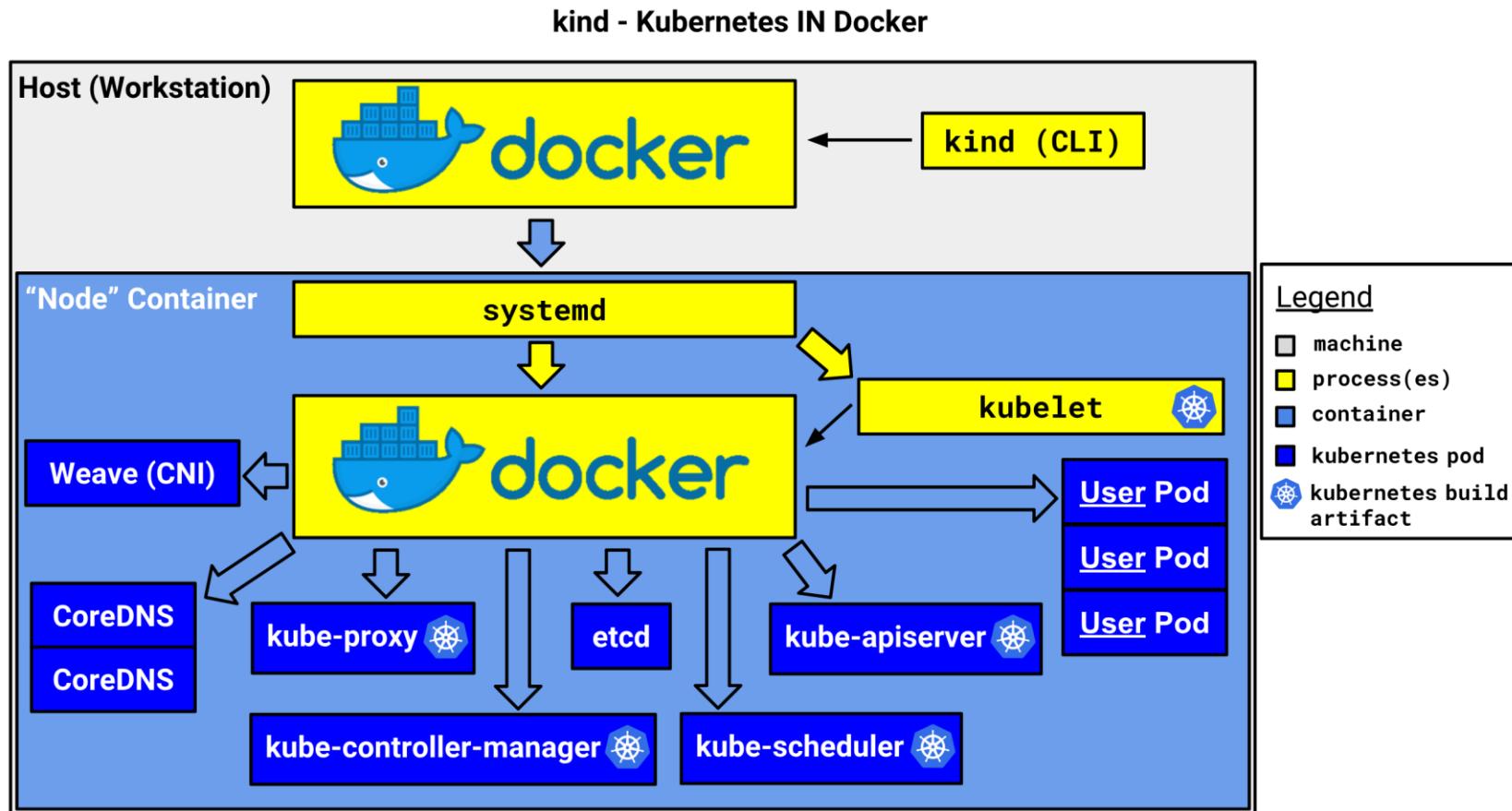
Hosting K8s Cluster

K8s cluster can run anywhere

- On prem
- In public cloud, self-hosted
- In public cloud, as a managed service
- In a private cloud, Openstack
- On virtualized VMs
- On bare metal hosts
- OpenShift – commercial offering by IBM



Workshop 1 – Setting up K8s cluster with kind



Relevant Links

<https://kind.sigs.k8s.io/docs/user/quick-start/>

https://www.eksworkshop.com/intermediate/200_migrate_to_eks/create-kind-cluster/

Pre-requisites

```
# remove all the docker images in Cloud9  
docker rmi -f $(docker images -aq)
```

```
# install kind  
curl -sLo kind https://kind.sigs.k8s.io/dl/v0.11.0/kind-linux-amd64  
# Type the command below. DO NOT copy/paste it!  
sudo install -o root -g root -m 0755 kind /usr/local/bin/kind  
rm -f ./kind
```

```
# Install kubectl – important! It should match cluster version 1.21  
# https://docs.aws.amazon.com/eks/latest/userguide/install-kubectl.html
```

```
curl -o kubectl https://s3.us-west-2.amazonaws.com/amazon-eks/1.21.2/2021-07-05/bin/linux/amd64/kubectl  
chmod +x ./kubectl  
mkdir -p $HOME/bin && cp ./kubectl $HOME/bin/kubectl && export PATH=$PATH:$HOME/bin  
echo 'export PATH=$PATH:$HOME/bin' >> ~/ .bashrc  
kubectl version --short --client
```

Create Cluster Description and Deploy Cluster

```
# You can copy from this link # https://www.eksworkshop.com/intermediate/200\_migrate\_to\_eks/create-kind-cluster/
```

```
# delete all the docker images apart from kind from your Cloud9
```

```
cat > kind.yaml <<EOF
```

```
kind: Cluster
```

```
apiVersion: kind.x-k8s.io/v1alpha4
```

```
nodes:
```

```
- role: control-plane
```

```
image: kindest/node:v1.19.11@sha256:07db187ae84b4b7de440a73886f008cf903fcf5764ba8106a9fd5243d6f32729
```

```
extraPortMappings:
```

```
- containerPort: 30000
```

```
    hostPort: 30000
```

```
- containerPort: 30001
```

```
    hostPort: 30001
```

```
EOF
```

```
# Create the cluster
```

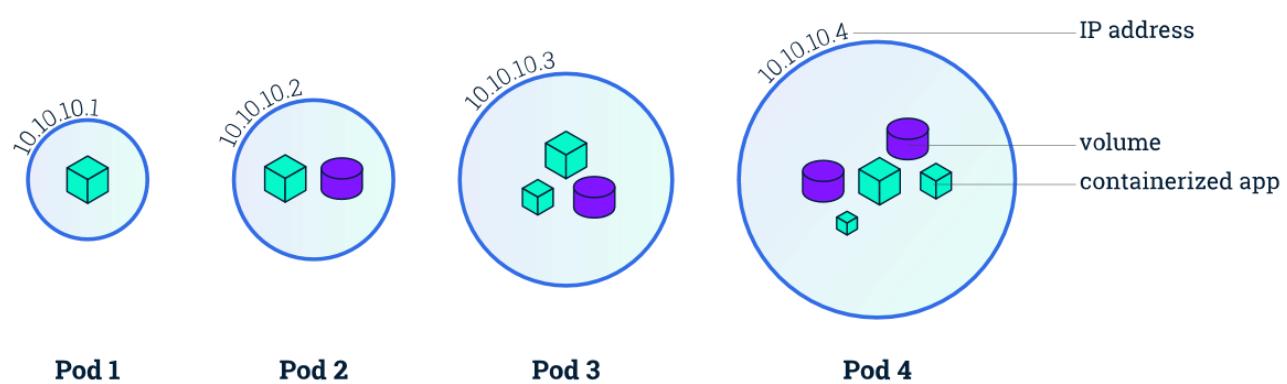
```
kind create cluster --config kind.yaml
```

Deploy our first application

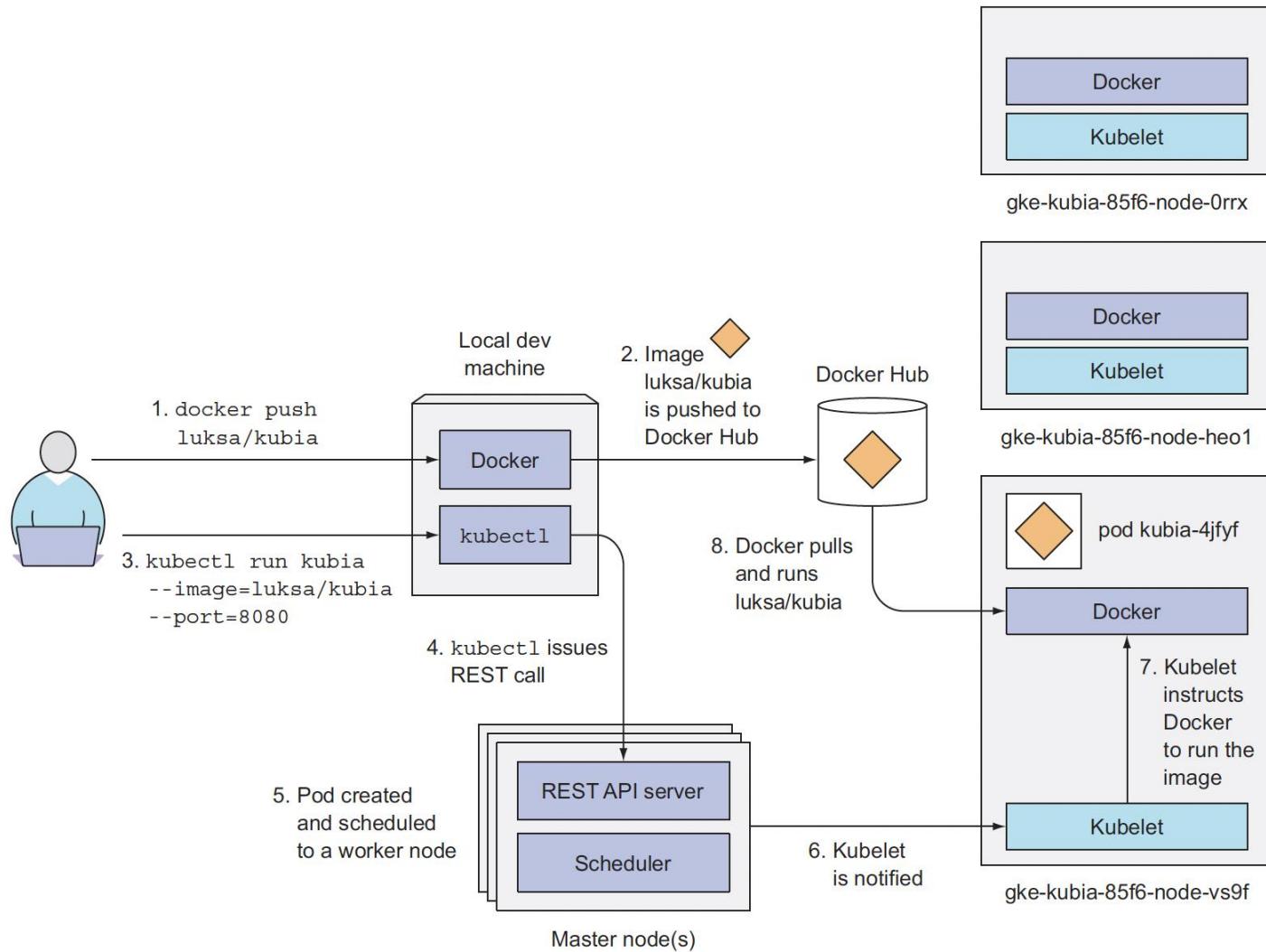
```
$ kubectl run nginx --image=nginx  
--port=80
```

```
# List the pods  
$ kubectl get pods
```

```
# What are pods?
```



Understanding the flow of "kubectl run" command



Expose your application

```
$ k expose pod nginx --type=NodePort --name nginx-http
```

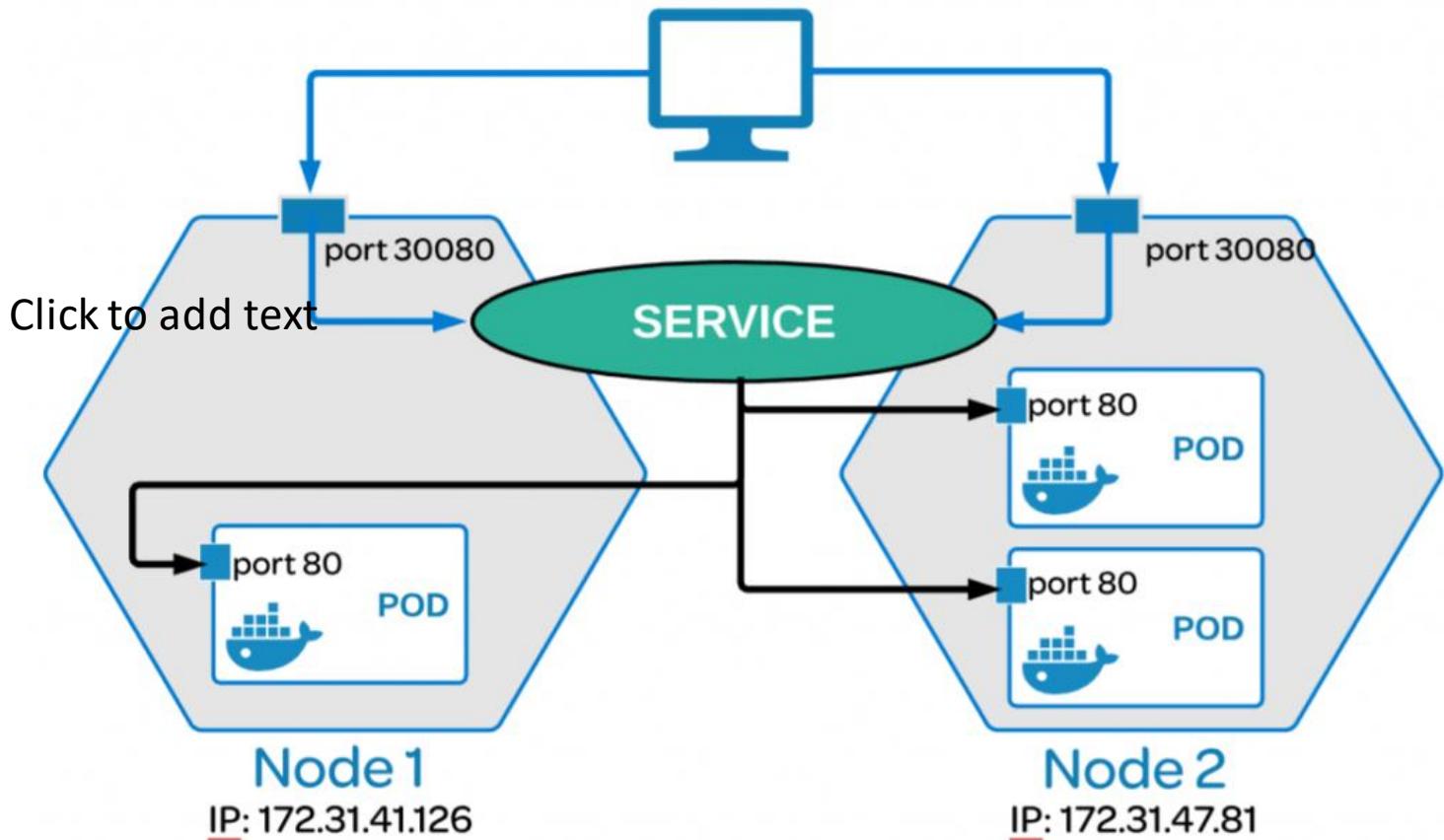
```
# List services
```

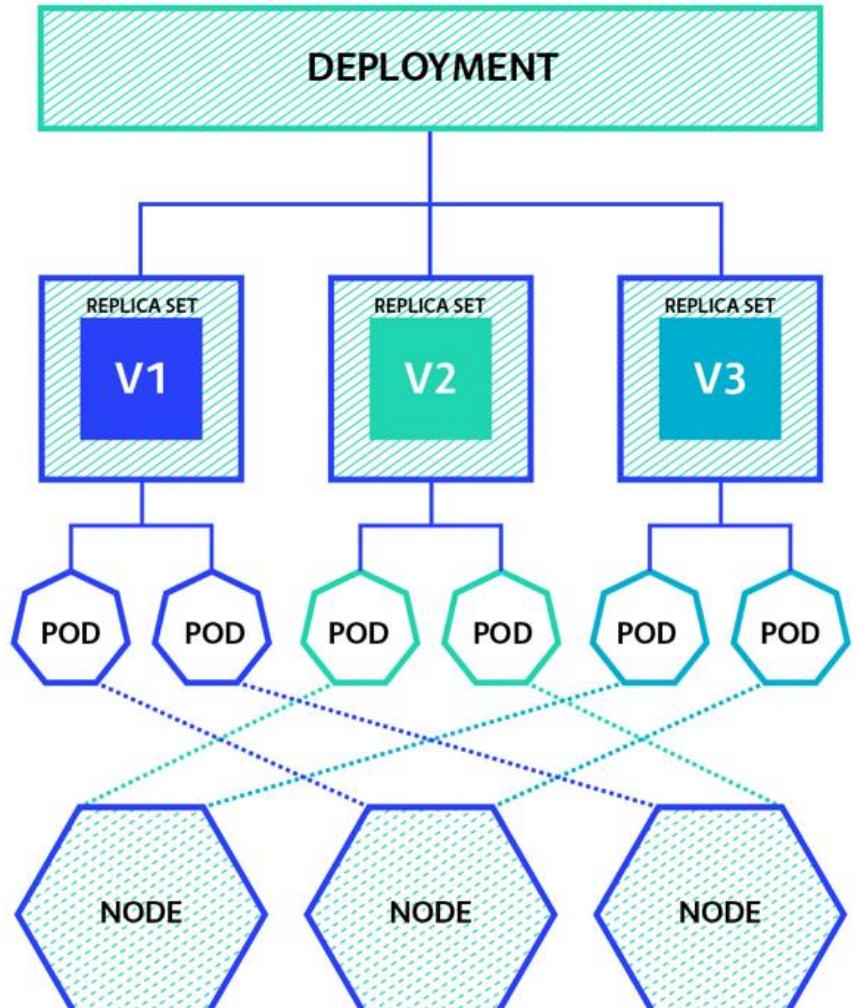
```
$ kubectl get services
```

```
# What are services?
```

Kubernetes Service

A service allows you to dynamically access a group of replica pods.





All the components:
Deployment, Service,
Pods, Nodes



Workshop 1 – The End

References

- <https://pycon2019.container.training/#1>
- <https://github.com/jpetazzo/container.training/tree/main/dockercoins>
- https://www.eksworkshop.com/intermediate/200_migrate_to_eks/deploy-counter-app-kind/
- <https://github.com/luksa/kubernetes-in-action>
- <https://kind.sigs.k8s.io/>

Kubernetes Core Concepts: Pods, ReplicaSets, Deployments

Week 5

Agenda



Pods vs Containers



Pods Use Cases



Deploying Pods

Using kubectl run

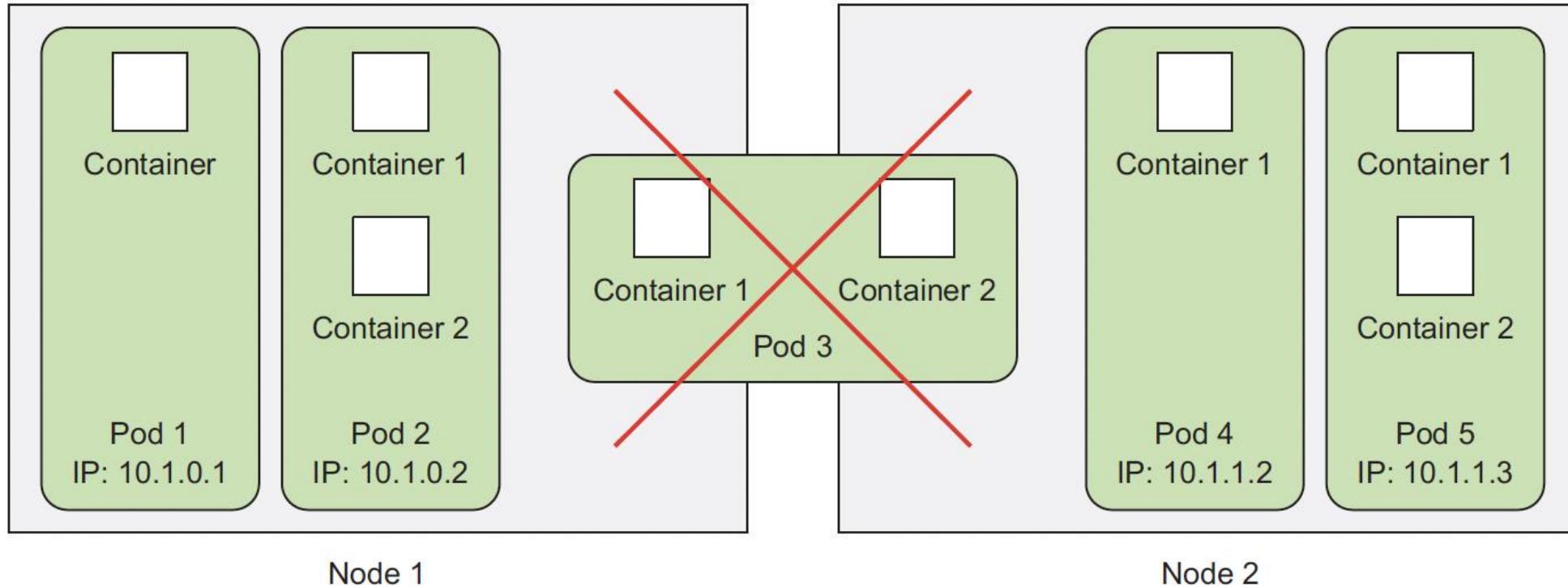
Using YAML file



ReplicaSets



Deploying ReplicaSet



Pods vs Containers

Pods are co-located groups of containers

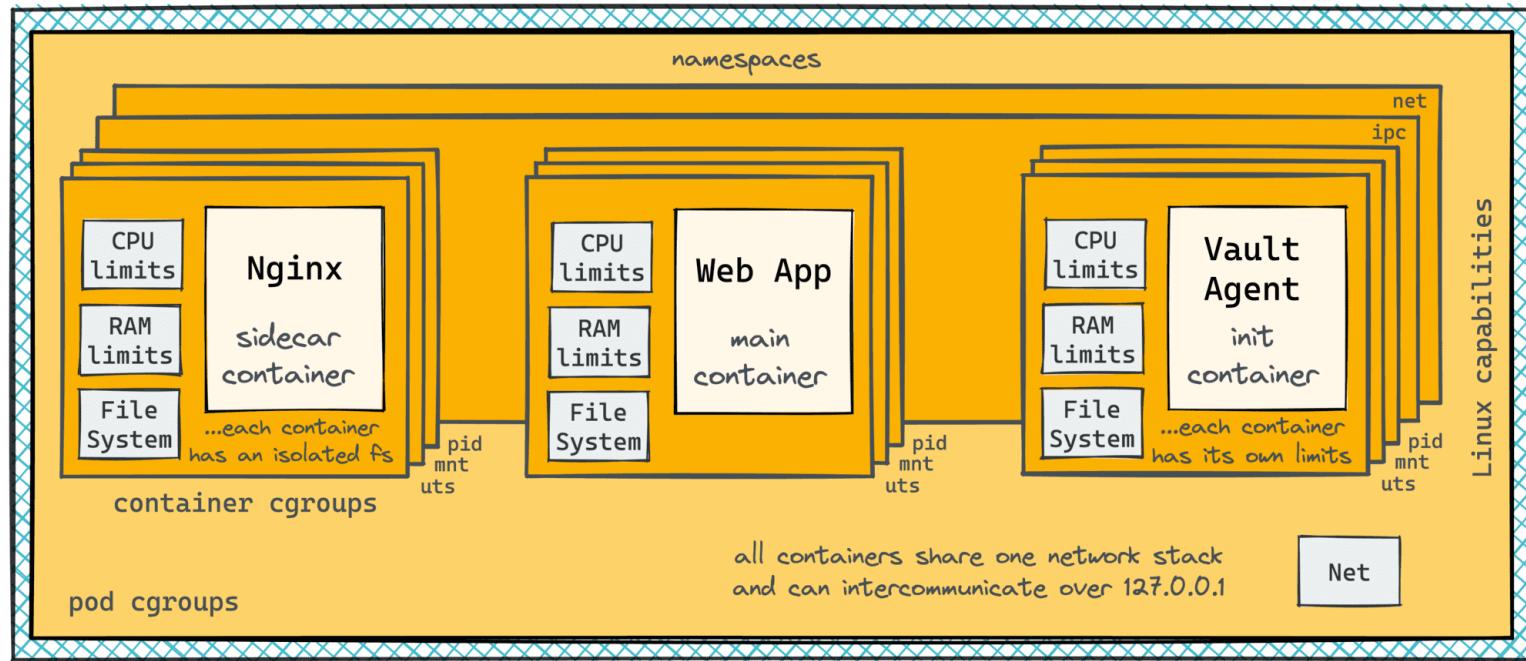
Pods (and not containers) represent the minimal deployment unit in K8s

Containers in a pods are scheduled, started and terminated as one unit

Containers in a pod share the same network, processes and can share the same filesystem

Pods – The best of VMs and Containers World

Kubernetes Pod - a multitenant "box"



Pod - a group of "semi-fused" containers

all containers in a Pod are addressable via a single IP address. E.g. 10.0.0.3



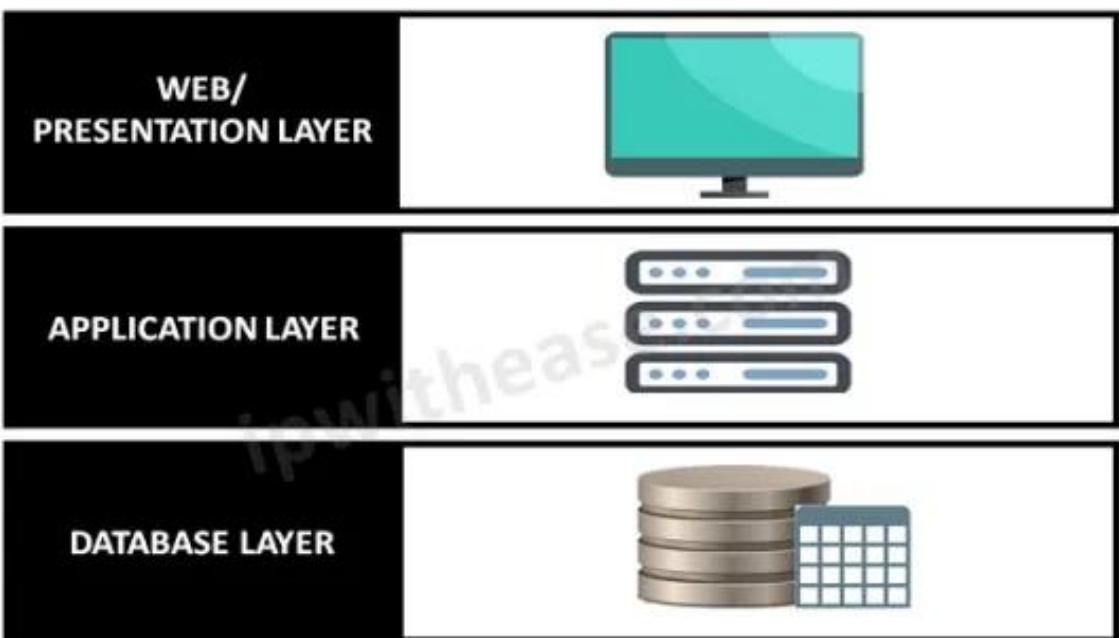
Pods Use Cases

Containers are designed to run a single process – what if we need to run multiple related processes the way we run them on a VM

- Pods are logical hosts used to run tightly coupled application components
 - Running multiple processes in a container adds a lot of complexity
 - Running related processes as multiple isolated containers is not trivial
 - Keeping track which containers are related
 - Start and stop them simultaneously
 - Scale them simultaneously
 - Pod provides the best of both worlds
 - Containers in the pod *share* Linux namespaces (UTS, Network IPC); docker containers have their own individual namespaces
 - Containers in the pod are *somewhat* isolated – each container in the pod has its own filesystem
 - Containers share the same loopback interface (127.0.0.1)

Splitting Application into Multiple Pods

THREE-TIER ARCHITECTURE IN APPLICATION



<https://ipwitthease.com>

Do you think a multi-tier application consisting of a frontend web server, application server and a backend database should be configured as a single pod or as two pods?

Running Multiple Containers in a Pod

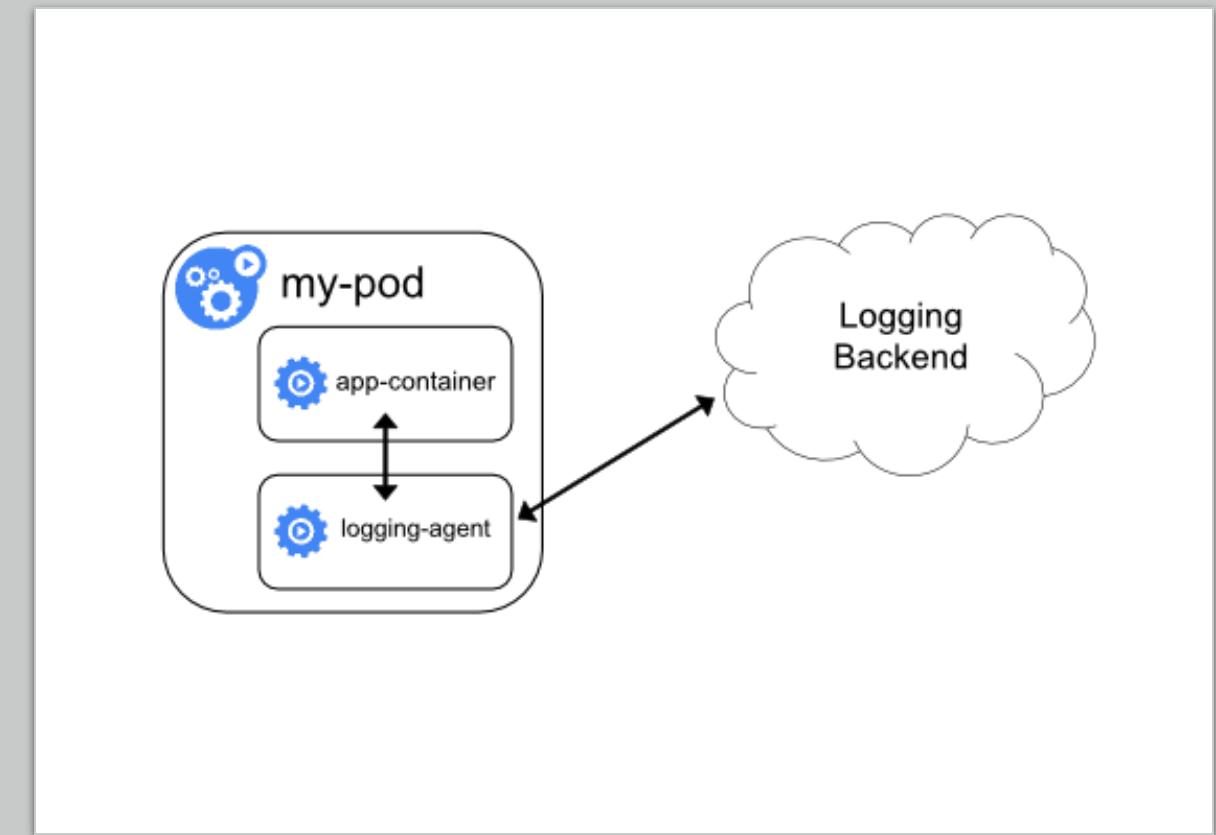
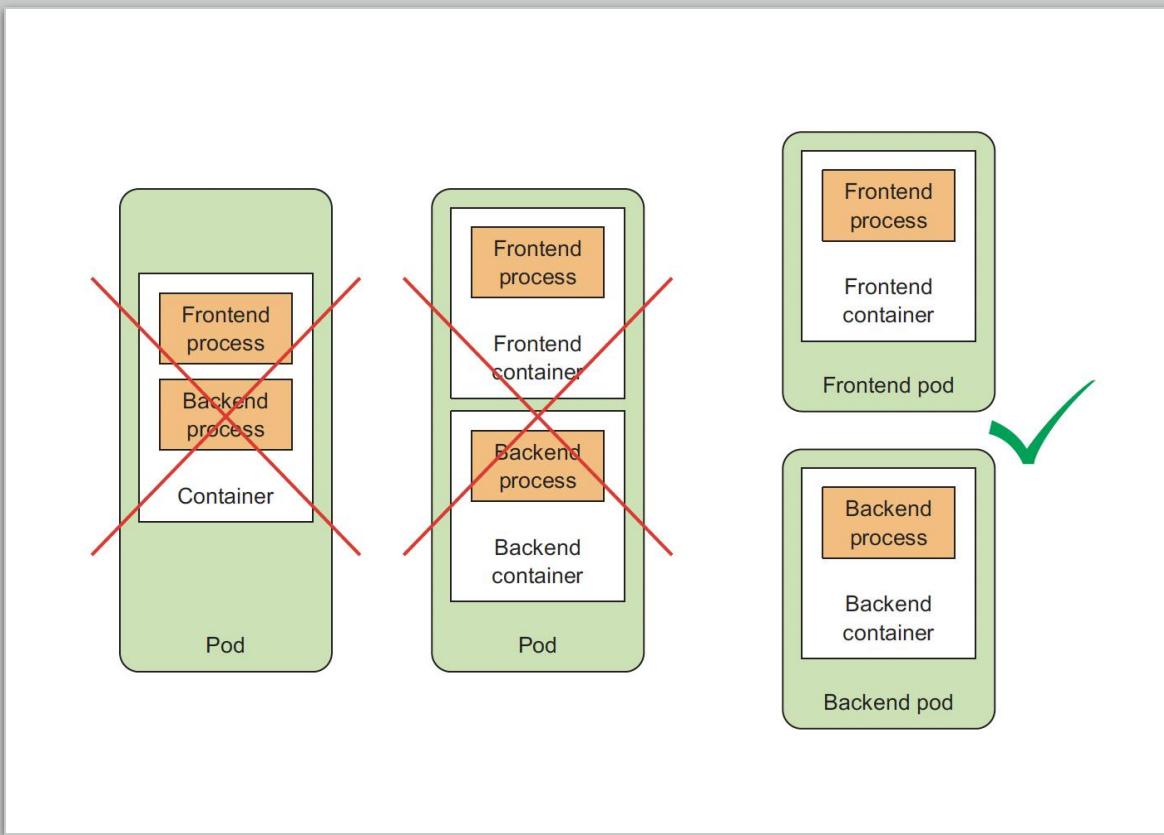


- Do they need to be run together or can they run on different hosts?
- Do they represent a single whole or are they independent components?
- Must they be scaled together or individually?

Collocating Processes in the Pod

Main process and a side car pattern

- user input validation sidecar
- logging rotation sidecar
- content enrichment sidecar



Introducing Pod Descriptor

- Metadata includes the name, namespace, labels, and other information about the pod.
- Spec contains the actual description of the pod's contents, such as the pod's containers, volumes, and other data. Pod specification/contents (list of pod's containers, volumes, and so on)

```
$ kubectl get po kubia-zxzij -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: ...
  creationTimestamp: 2016-03-18T12:37:50Z
  generateName: kubia-
  labels:
    run: kubia
    name: kubia-zxzij
  namespace: default
  resourceVersion: "294"
  selfLink: /api/v1/namespaces/default/pods/kubia-zxzij
  uid: 3a564dc0-ed06-11e5-ba3b-42010af00004
spec:
  containers:
  - image: luksa/kubia
    imagePullPolicy: IfNotPresent
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
    resources:
      requests:
        cpu: 100m
```

Kubernetes API version used in this YAML descriptor

Type of Kubernetes object/resource

Pod metadata (name, labels, annotations, and so on)

Pod specification/contents (list of pod's containers, volumes, and so on)

YAML – Pod Descriptor

```
$ kubectl get po kubia-zxzij -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: ...
  creationTimestamp: 2016-03-18T12:37:50Z
  generateName: kubia-
  labels:
    run: kubia
  name: kubia-zxzij
  namespace: default
  resourceVersion: "294"
  selfLink: /api/v1/namespaces/default/pods/kubia-zxzij
  uid: 3a564dc0-ed06-11e5-ba3b-42010af00004
spec:
  containers:
    - image: luksa/kubia
      imagePullPolicy: IfNotPresent
      name: kubia
      ports:
        - containerPort: 8080
          protocol: TCP
      resources:
        requests:
          cpu: 100m
```

Kubernetes API version used in this YAML descriptor

Type of Kubernetes object/resource

Pod metadata (name, labels, annotations, and so on)

Pod specification/contents (list of pod's containers, volumes, and so on)

Creating Pods from Descriptor

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual
spec:
  containers:
    - image: luksa/kubia
      name: kubia
      ports:
        - containerPort: 8080
          protocol: TCP
```

Descriptor conforms to version v1 of Kubernetes API

You're describing a pod.

The name of the pod

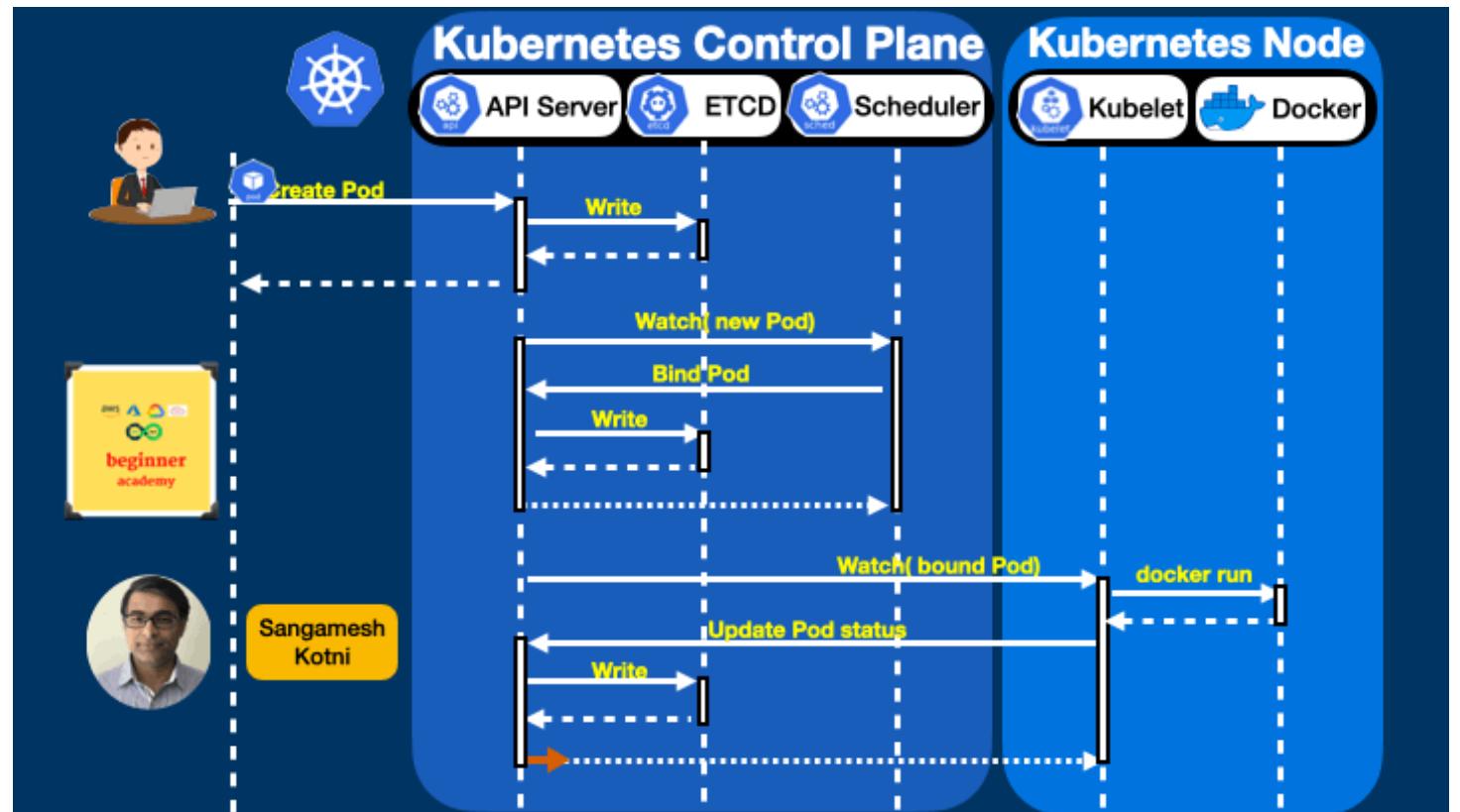
Container image to create the container from

Name of the container

The port the app is listening on

Kubernetes Pod Scheduling Flow

1. The user creates a Pod via the API Server
2. The API server writes it to etcd
3. The scheduler notices an “unbound” Pod and decides which node to run that Pod on
4. Scheduler writes that binding back to the API Server
5. The Kubelet notices a change in the set of Pods that are bound to its node
6. Kubelet, in turn, runs the container via the container runtime (i.e. Docker)
7. The Kubelet monitors the status of the Pod via the container runtime
8. As things change, the Kubelet will reflect the current status back to the API Server



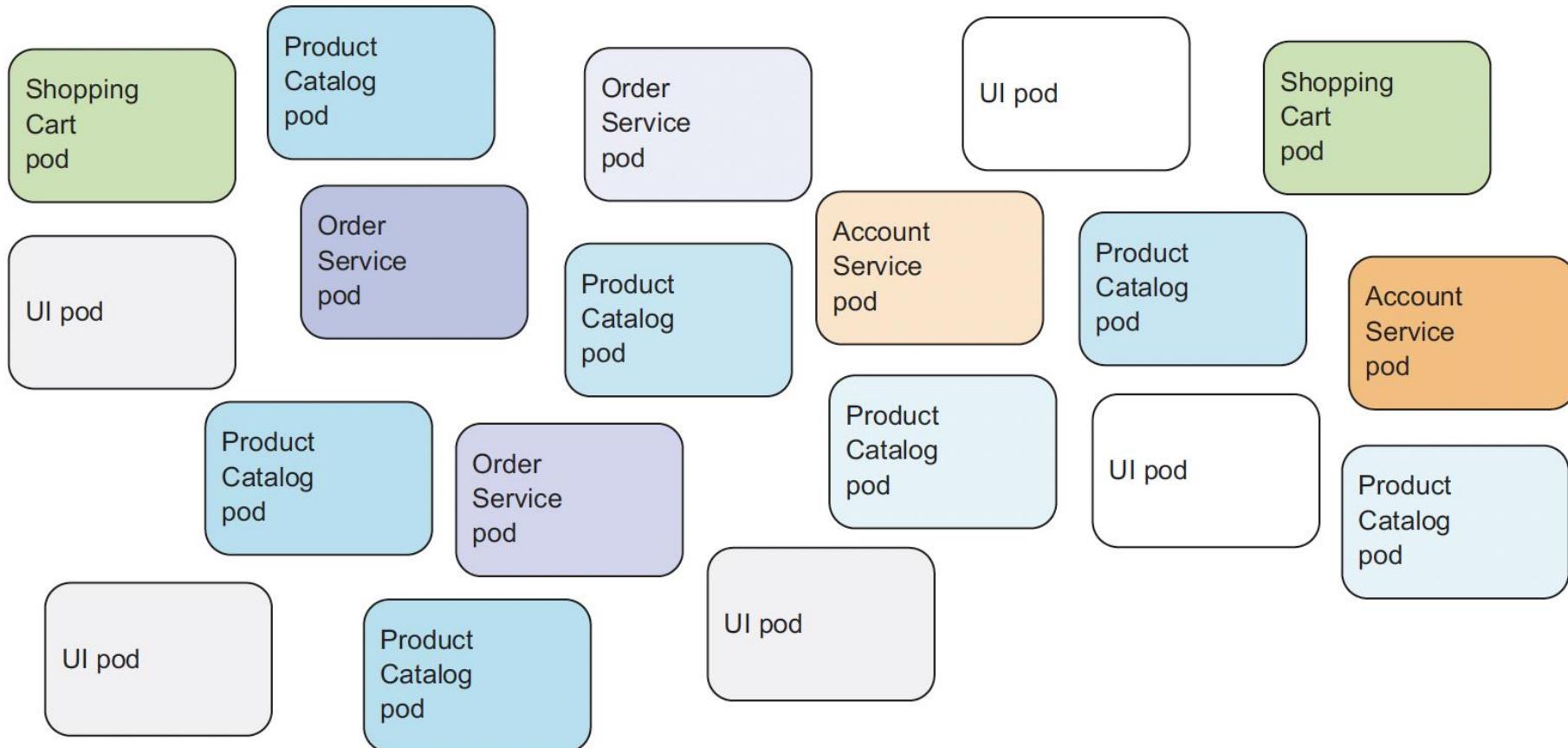
Pods: Summary

- Can have one or multiple containers
- Runs on a single node
- (Pod cannot "straddle" multiple nodes)
- Pods cannot be moved (e.g. in case of node outage)
- Pods cannot be scaled horizontally (except by manually creating more Pods)

Pods: Summary Continued

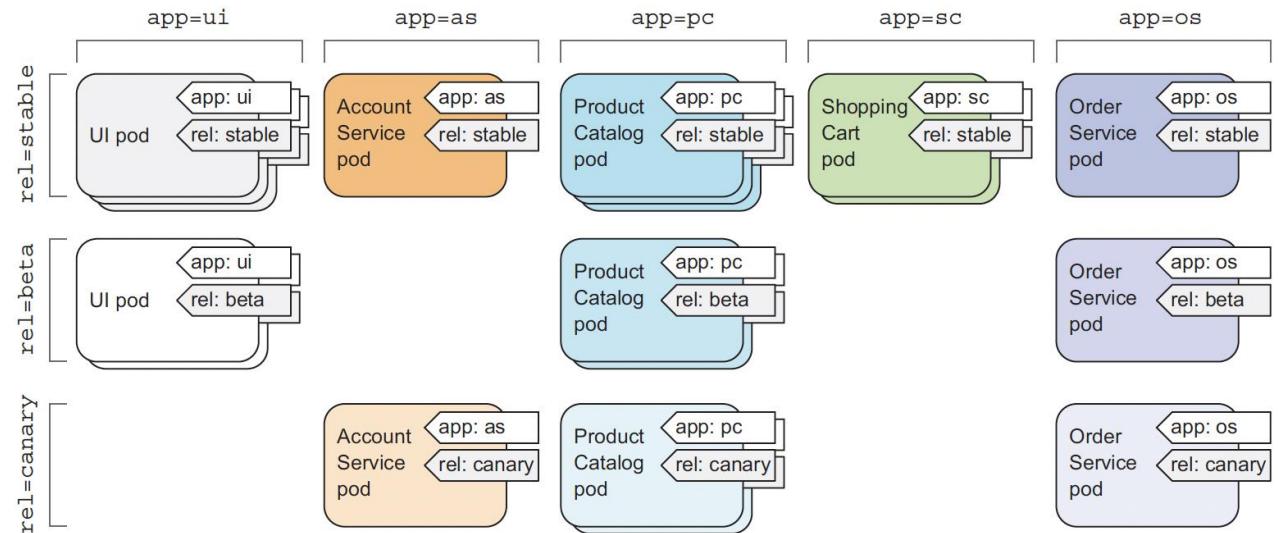
- Pod is not a process; it's an environment for containers
 - it cannot be "restarted"
 - it cannot "crash"
- The containers in a Pod can crash
- They may or may not get restarted
 - (depending on Pod's restart policy)
- If all containers exit successfully, the Pod ends in "Succeeded" phase
- If some containers fail and don't get restarted, the Pod ends in "Failed" phase

Organizing K8s Objects with Labels



Labels in K8s

- Arbitrary key-value pairs added to resource manifest
- Used by label selectors
- Added to metadata section of the manifest
- Example
 - App
 - Release



Scaling Pods

Task: We need to
create multiple
containers of the
same type

We can run the
commands below
multiple times

`kubectl run kubia –image luksa/kubia`

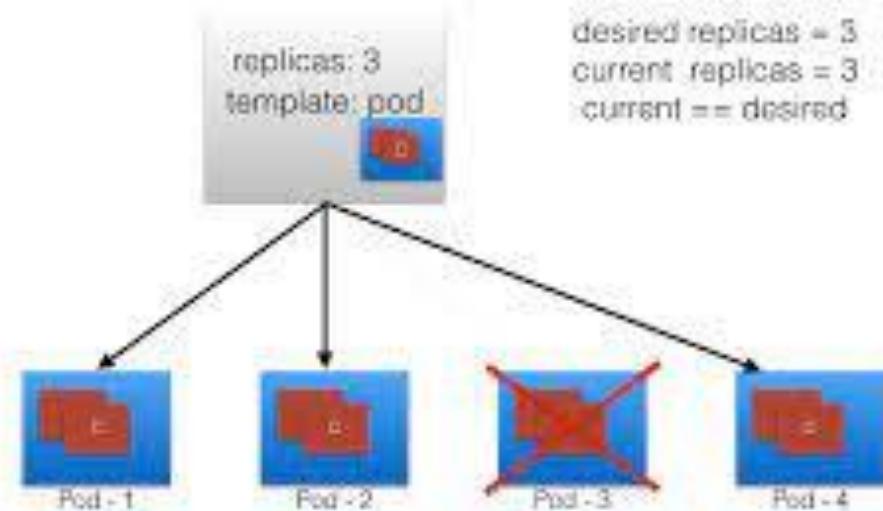
`kubectl create –f kubia-descriptor.yaml`

Is it the right way
to scale?

Scaling Pods - ReplicaSet

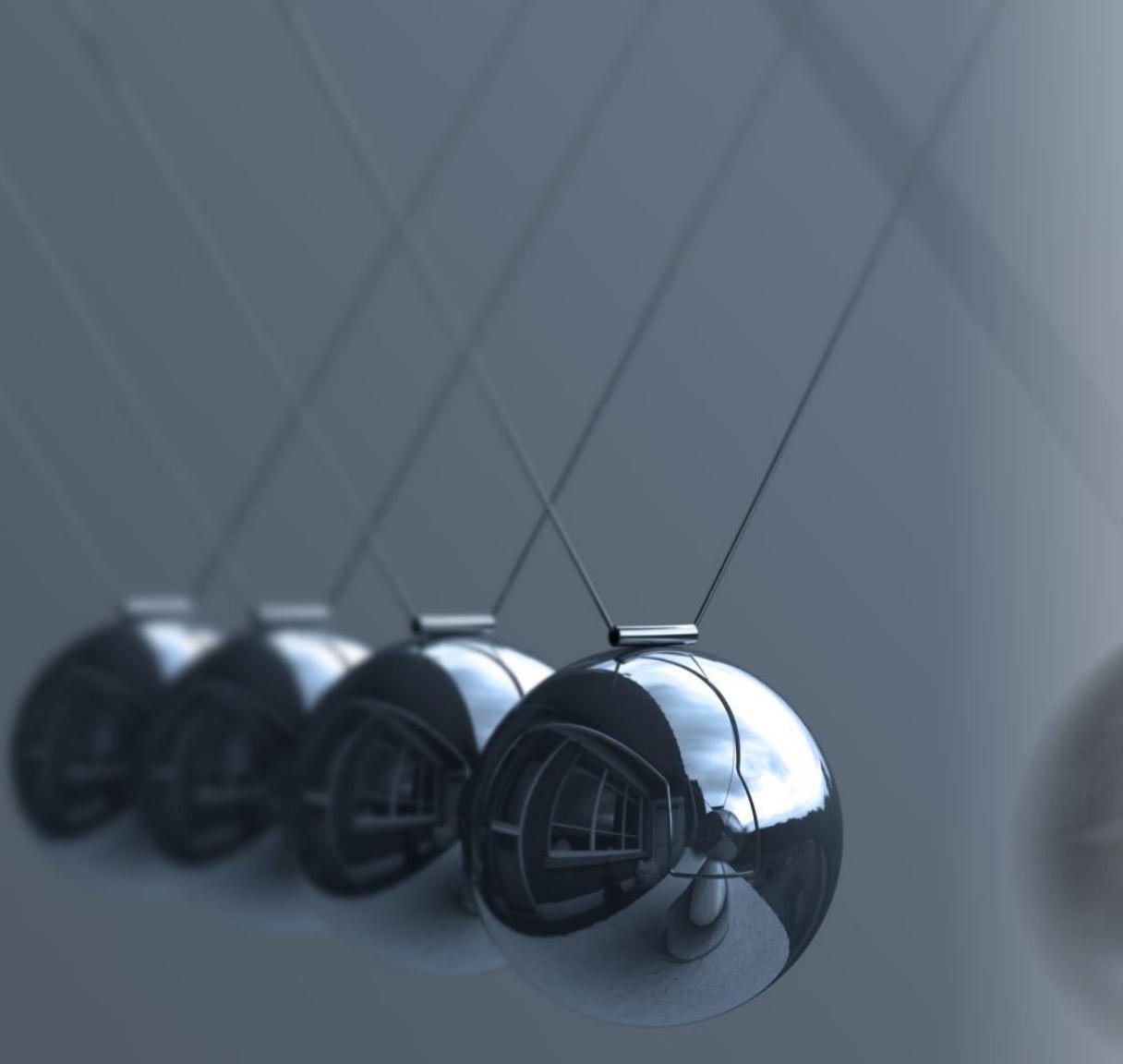
- Set of identical (replicated) Pods
- Defined by a pod template + number of desired replicas
- If there are not enough Pods, the Replica Set creates more (e.g. in case of node outage; or simply when scaling up)
- If there are too many Pods, the Replica Set deletes some (e.g. if a node was disconnected and comes back; or when scaling down)
- We can scale up/down a Replica Set
 - we update the manifest of the Replica Set
 - as a consequence, the Replica Set controller creates/deletes Pods

Replica Set



Creating ReplicaSet from Descriptor

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  annotations:
    strategy.spinnaker.io/max-version-history: '2'
    traffic.spinnaker.io/load-balancers: '["service my-service"]'
  labels:
    tier: frontend
  name: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - image: 'gcr.io/google_samp.../gb-frontend:v3'
          name: frontend
```



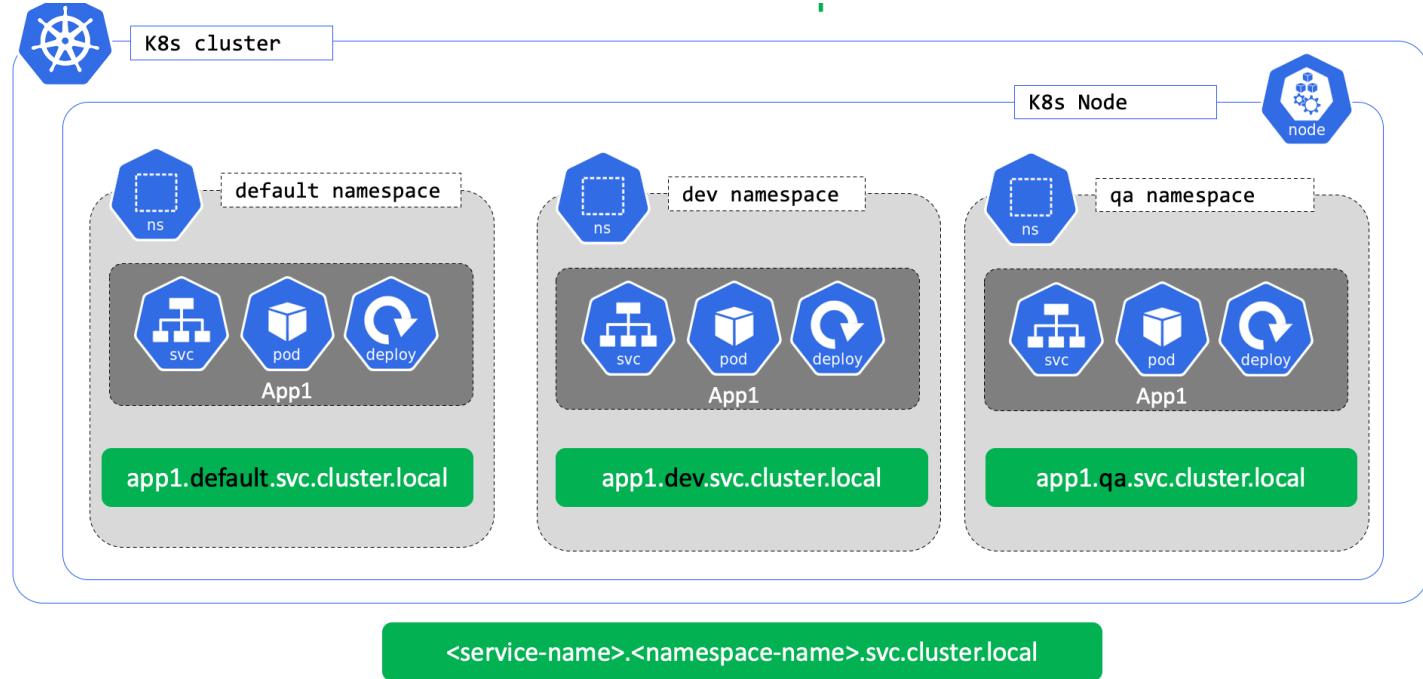
Pods -> ReplicaSets -> Deployments

- Replica Sets control *identical* Pods
- Deployments are used to roll out different Pods (different image, command, environment variables, ...)
- When we update a Deployment with a new Pod definition:
 - a new Replica Set is created with the new Pod definition
 - that new Replica Set is progressively scaled up
 - meanwhile, the old Replica Set(s) is(are) scaled down
- This is a *rolling update*, minimizing application downtime
- When we scale up/down a Deployment, it scales up/down its Replica Set

Organizing Pods

- Namespaces

- Provide scope for object names
- Allows to organize complex systems into distinct groups
- Not all resources are namespaced
 - Nodes
 - Cluster Roles



Workshop 1 – Deployment of K8s Pod



kubernetes



Deployment of K8s Pod - Prerequisites



RUNNING K8S CLUSTER –
KIND



CONTAINER IMAGE
ACCESSIBLE BY K8S NODES

Preparing the cluster

```
# Deploy EC2 instance

ssh-keygen -f week5

terraform init

terraform apply --auto-approve

# Copy the installation files to the instance

scp -i week5 init_kind.sh kind.yaml 44.196.58.89:/tmp

# Log into the instance

ssh -i week5 <public ip>

# Update permissions

cd /tmp

chmod 777 init_kind.sh

# Run installation script. If the command errors out, exit ssh session, ssh to the instance again and re-run
./init_kind.sh

./init_kind.sh

# Verify the cluster is running

kubectl get nodes
```

Nginx pod deployment

```
# Creating nginx pod
$ kubectl run nginx --image nginx
# List all pods in a default namespace
$ kubectl get pods
# List all pods in all namespaces
$ kubectl get pods --all-namespaces
# Retrieve pod's descriptor
$ kubectl get pod nginx -o yaml
# Access nginx web server running in a pod
$ kubectl port-forward nginx 8080:80
$ curl localhost:8080 # Send this command in a new terminal session
```



Create pod manifest

```
# Create a manifest file  
$ alias k=kubectl  
  
k run kubia --image luksa/kubia --  
dry-run -o yaml > kubia-manual.yaml  
  
# Edit the manifest and create the  
pod  
  
k apply -f kubia-manual.yaml
```

```
apiVersion: v1  
kind: Pod  
metadata:  
  labels:  
    run: kubia  
    name: kubia  
spec:  
  containers:  
  - image: luksa/kubia  
    name: kubia  
    resources: {}  
  dnsPolicy: ClusterFirst  
  restartPolicy: Always
```

Kubia deployment with YAML manifest

```
$ kubectl explain pods
```

```
$ kubectl explain pod.spec
```

```
$ kubectl create -f kubia-manual.yaml
```

```
$ kubectl get po kubia-manual -o yaml
```

```
$ kubectl logs kubia-manual
```

```
$ kubectl logs kubia-manual -c kubia
```

```
$ kubectl port-forward kubia-manual  
8888:8080
```

```
$ curl localhost:8888
```

The diagram illustrates a Kubernetes Pod YAML manifest with annotations explaining its components:

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual
spec:
  containers:
    - image: luksa/kubia
      name: kubia
      ports:
        - containerPort: 8080
          protocol: TCP
```

- Descriptor conforms to version v1 of Kubernetes API**: Points to the `apiVersion: v1` field.
- You're describing a pod.**: Points to the `kind: Pod` field.
- The name of the pod**: Points to the `name: kubia-manual` field.
- Container image to create the container from**: Points to the `image: luksa/kubia` field.
- Name of the container**: Points to the `name: kubia` field.
- The port the app is listening on**: Points to the `containerPort: 8080` field.

Retrieving logs - Pingpong

```
# Get logs of the pod. By default, the command will only show the logs of the first container in the pod
```

```
$ kubectl run pingpong --image alpine ping 127.0.0.1
```

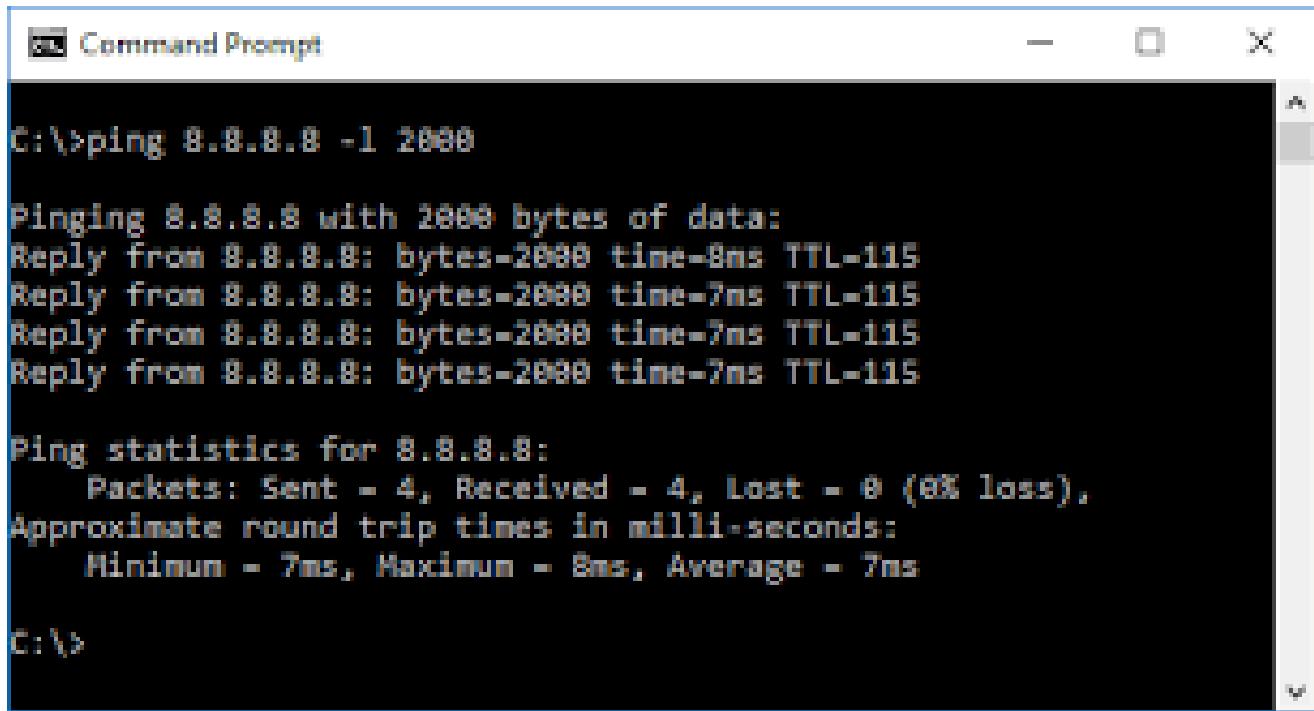
```
$ kubectl logs pingpong
```

```
$ kubectl logs pingpong -c pingpong
```

```
$ kubectl logs pingpong --tail 1 --follow
```

```
# Full definition of the running pod
```

```
$ kubectl get po pingpong -o yaml
```



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered is "C:\>ping 8.8.8.8 -l 2000". The output displays four successful replies from the IP address 8.8.8.8, each with 2000 bytes of data, a time of 8ms, and a TTL of 115. It then provides ping statistics for the target host, showing 4 sent packets, 4 received packets, 0 lost packets (0% loss), and approximate round-trip times of 7ms, with a minimum of 7ms, a maximum of 8ms, and an average of 7ms.

```
C:\>ping 8.8.8.8 -l 2000

Pinging 8.8.8.8 with 2000 bytes of data:
Reply from 8.8.8.8: bytes=2000 time=8ms TTL=115
Reply from 8.8.8.8: bytes=2000 time=7ms TTL=115
Reply from 8.8.8.8: bytes=2000 time=7ms TTL=115
Reply from 8.8.8.8: bytes=2000 time=7ms TTL=115

Ping statistics for 8.8.8.8:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
        Minimum = 7ms, Maximum = 8ms, Average = 7ms

C:\>
```

Can we scale the pod? Deployments => ReplicaSets => Pods

```
# Let's try scaling the pod  
$ kubectl scale pod pingpong --replicas=3  
  
# Let's create a deployment of the pingpong in  
it's own namespace  
alias k=kubectl  
k create ns pingpong  
kubectl create deployment pingpong --  
image=alpine:3.14 -n pingpong -- ping 127.0.0.1  
kubectl get all -n pingpong
```

```
[ec2-user@ip-172-31-12-180 tmp]$ k get all -n pingpong  
NAME                                     READY   STATUS    RESTARTS   AGE  
pod/pingpong-85bddc666f-dqfv6   0/1     Error     2          18s  
  
NAME                               READY   UP-TO-DATE   AVAILABLE   AGE  
deployment.apps/pingpong   0/1     1           0          18s  
  
NAME                           DESIRED   CURRENT   READY   AGE  
replicaset.apps/pingpong-85bddc666f   1         1         0      18s  
[ec2-user@ip-172-31-12-180 tmp]$ █
```

```
[ec2-user@ip-172-31-12-180 tmp]$ k get deployment pingpong
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "2"
  creationTimestamp: "2022-06-07T01:44:36Z"
  generation: 2
  labels:
    app: pingpong
    name: pingpong
    namespace: pingpong
    resourceVersion: "2218"
  selfLink: /apis/apps/v1/namespaces/pingpong/deployments/pingpong
  uid: f65cadd3-1c53-4db1-ab51-485217ced220
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: pingpong
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
      type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: pingpong
    spec:
      containers:
        - command:
          - ping
          - 127.0.0.1
          image: alpine:3.15
          imagePullPolicy: IfNotPresent
          name: alpine
          resources: {}
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
        dnsPolicy: ClusterFirst
        restartPolicy: Always
        schedulerName: default-scheduler
        securityContext: {}
        terminationGracePeriodSeconds: 30
```

Descriptor: Deployment

- Responsible for multiple versions of application (replicaset)
- Defines the way to perform rolling forward and rollback updates
- Does it by using ReplicaSets
- `k get deployment pingpong -n pingpong -o yaml`

Descriptor: ReplicaSet



```
[ec2-user@ip-172-31-12-180 tmp]$ k get rs pingpong-6754b574d9 -n pingpong -o yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  annotations:
    deployment.kubernetes.io/desired-replicas: "1"
    deployment.kubernetes.io/max-replicas: "2"
    deployment.kubernetes.io/revision: "2"
  creationTimestamp: "2022-06-07T01:51:21Z"
  generation: 1
  labels:
    app: pingpong
    pod-template-hash: 6754b574d9
  name: pingpong-6754b574d9
  namespace: pingpong
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: Deployment
    name: pingpong
    uid: f65cadd3-1c53-4db1-ab51-485217ced220
  resourceVersion: "2209"
  selfLink: /apis/apps/v1/namespaces/pingpong/replicasets/pingpong-6754b574d9
  uid: 75aae3e4-fa12-4610-85eb-7794aa012a11
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pingpong
      pod-template-hash: 6754b574d9
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: pingpong
        pod-template-hash: 6754b574d9
    spec:
      containers:
      - command:
        - ping
        - 127.0.0.1
        image: alpine:3.15
        imagePullPolicy: IfNotPresent
        name: alpine
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
```

Descriptor: Pod



```
error from server (not found): pods "pingpong-6754b574d9-9lphv" not found
[ec2-user@ip-172-31-12-180 tmp]$ k get pod/pingpong-6754b574d9-9lphv -o yaml -n pingpong
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2022-06-07T01:51:21Z"
  generateName: pingoona-6754b574d9-
  labels:
    app: pingpong
    pod-template-hash: 6754b574d9
  name: pingpong-6754b574d9-9lphv
  namespace: pingpong
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: pingpong-6754b574d9
    uid: 75aae3e4-fa12-4610-85eb-7794aa012a11
  resourceVersion: "2208"
  selflink: /api/v1/namespaces/pingpong/pods/pingpong-6754b574d9-9lphv
  uid: 9af78bf3-b2d0-418e-b07e-6e102d6bbe70
spec:
  containers:
  - command:
    - ping
    - -127.0.0.1
    image: alpine:3.15
    imagePullPolicy: IfNotPresent
    name: alpine
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-bj2gf
      readOnly: true
    dnsPolicy: ClusterFirst
    enableServiceLinks: true
    nodeName: kind-control-plane
    preemptionPolicy: PreemptLowerPriority
    priority: 0
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    serviceAccount: default
    serviceAccountName: default
    terminationGracePeriodSeconds: 30
    tolerations:
    - effect: NoExecute
      key: node.kubernetes.io/not-ready
      operator: Exists
      tolerationSeconds: 300
    - effect: NoExecute
      key: node.kubernetes.io/unreachable
      operator: Exists
      tolerationSeconds: 300
    volumes:
    - name: default-token-bj2gf
      secret:
        defaultMode: 420
        secretName: default-token-bj2gf
```

```
deployment.apps/pingpong status
```

```
[ec2-user@ip-172-31-12-180 tmp]$ k get all -n pingpong
```

NAME	READY	STATUS	RESTARTS	AGE
pod/pingpong-6754b574d9-9lphv	1/1	Running	0	13m
pod/pingpong-6754b574d9-cbvs2	1/1	Running	0	14s
pod/pingpong-6754b574d9-chflc	1/1	Running	0	14s

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/pingpong	3/3	3	3	20m

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/pingpong-6754b574d9	3	3	3	13m
replicaset.apps/pingpong-85bddc666f	0	0	0	20m

Scaling

```
k scale deployment pingpong --replicas 3 -n pingpong  
k scale deployment pingpong --replicas 1 -n pingpong
```

Rolling out a new version – Updating alpine version to 3.15

```
[ec2-user@ip-172-31-12-180 tmp]$ k get all -n pingpong
NAME                               READY   STATUS        RESTARTS   AGE
pod/pingpong-6754b574d9-9lphv   1/1    Running      0          17m
pod/pingpong-7774544bf9-kbzfk  0/1    Terminating  3          69s
pod/pingpong-7bdd67fd57-9xmvz  0/1    ContainerCreating  0          1s

NAME               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/pingpong  1/1     1           1           24m

NAME                           DESIRED   CURRENT   READY   AGE
replicaset.apps/pingpong-6754b574d9  1         1         1         17m
replicaset.apps/pingpong-7774544bf9  0         0         0         69s
replicaset.apps/pingpong-7bdd67fd57  1         1         0         2s
replicaset.apps/pingpong-85bddc666f  0         0         0         24m
[ec2-user@ip-172-31-12-180 tmp]$ █
```

Maintaining the desired state - Resilience

```
watch kubectl get pods
```

```
kubectl delete pod pingpong-xxxxxxxxx-yyyyy
```

As soon as the pod is in "Terminating" state,
the Replica Set replaces it

What will happen if we terminate the "stand-alone" pod?





Workshop 1 – The End

Kubernetes Core Concepts: Deployment Rollouts, Labels and Selectors, Services

Week 6

Agenda



Rollouts



Deployments



Services

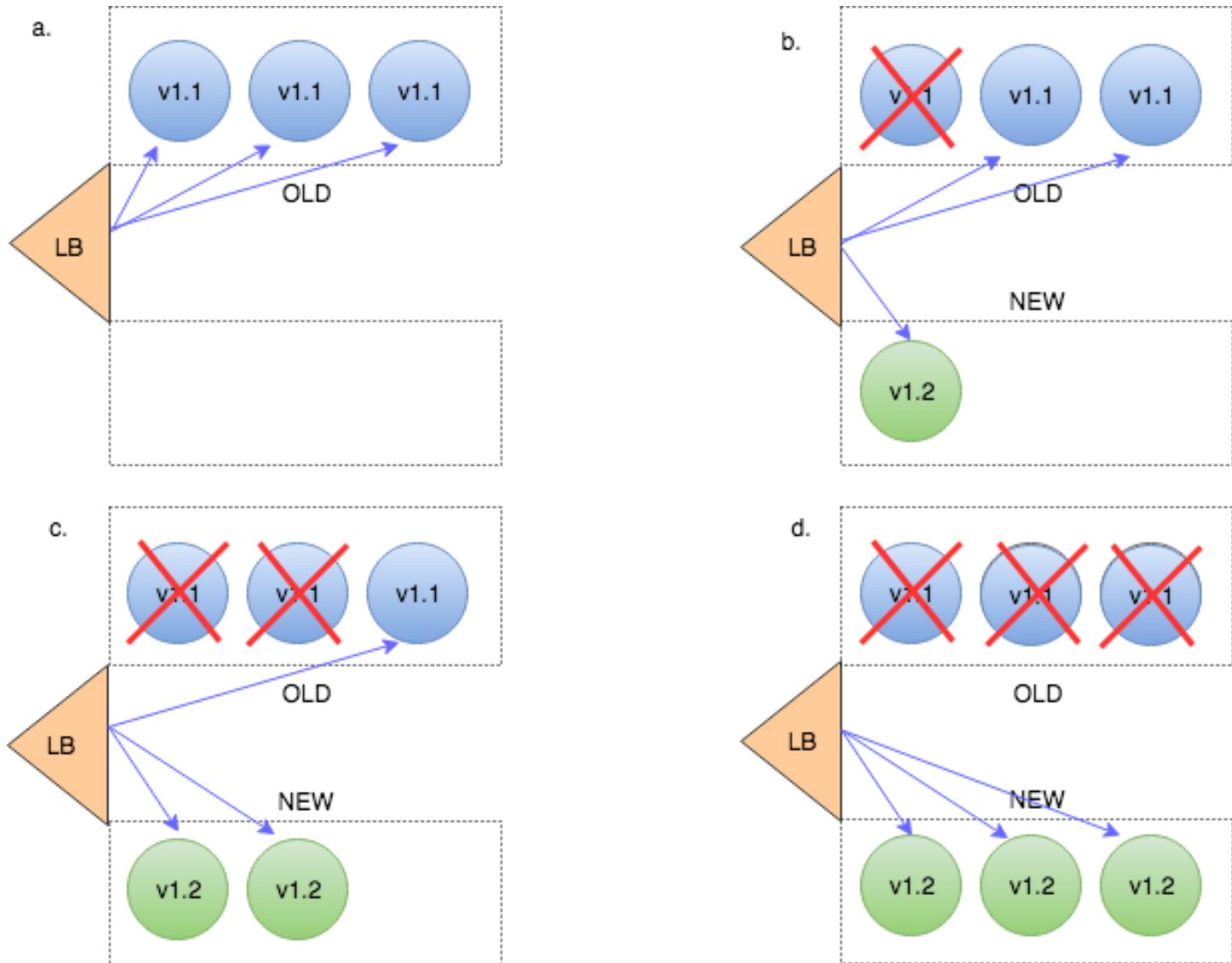


Deployment Rollouts

- Rollout is *a process* of creating or upgrading application containers
- Rollout happens when
 - We create a new deployment
 - Update images used in the existing deployment
- Rollout creates a new deployment revision
- Revisions help in keeping history and enable rollback

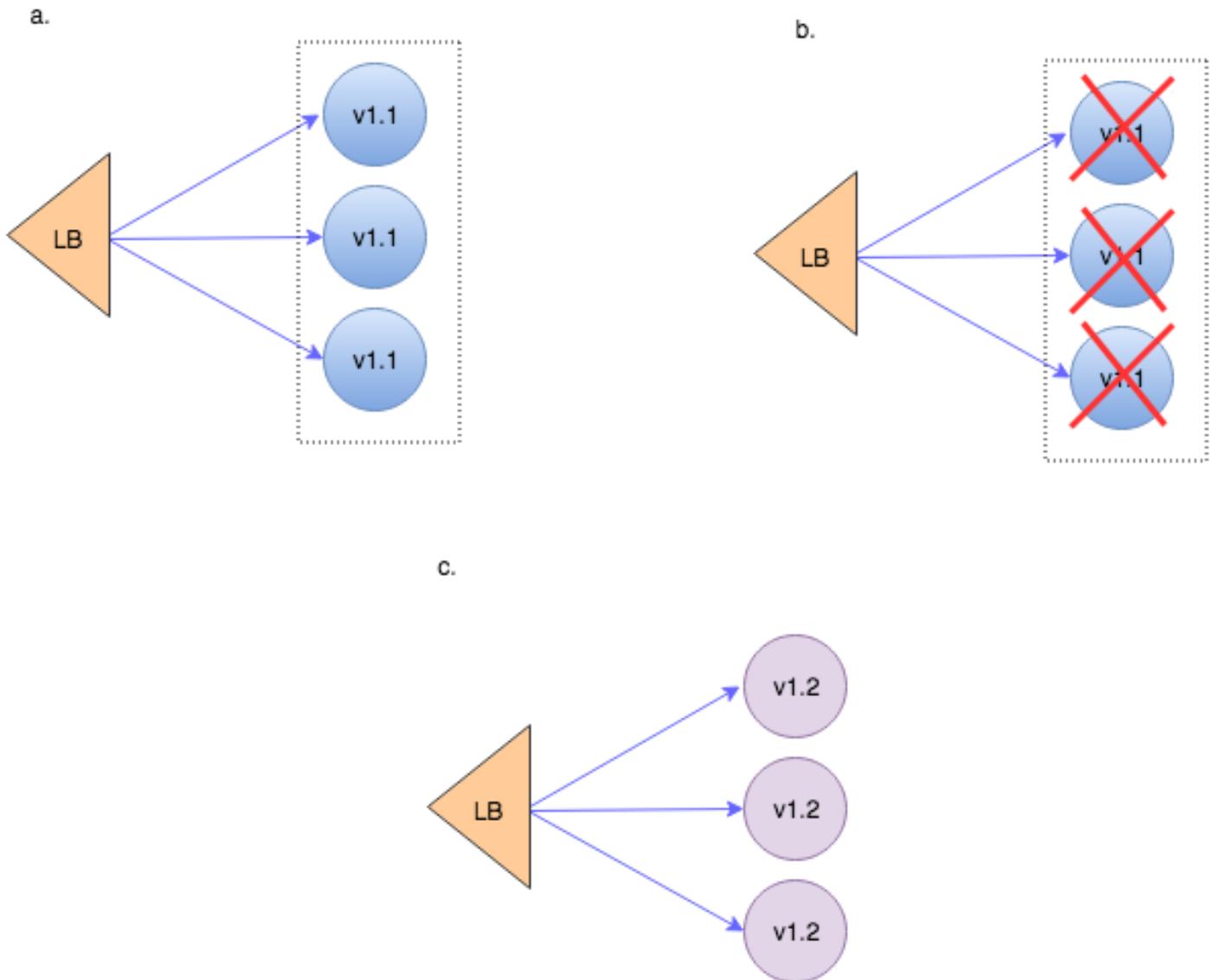
Rollout Strategies: Rollout

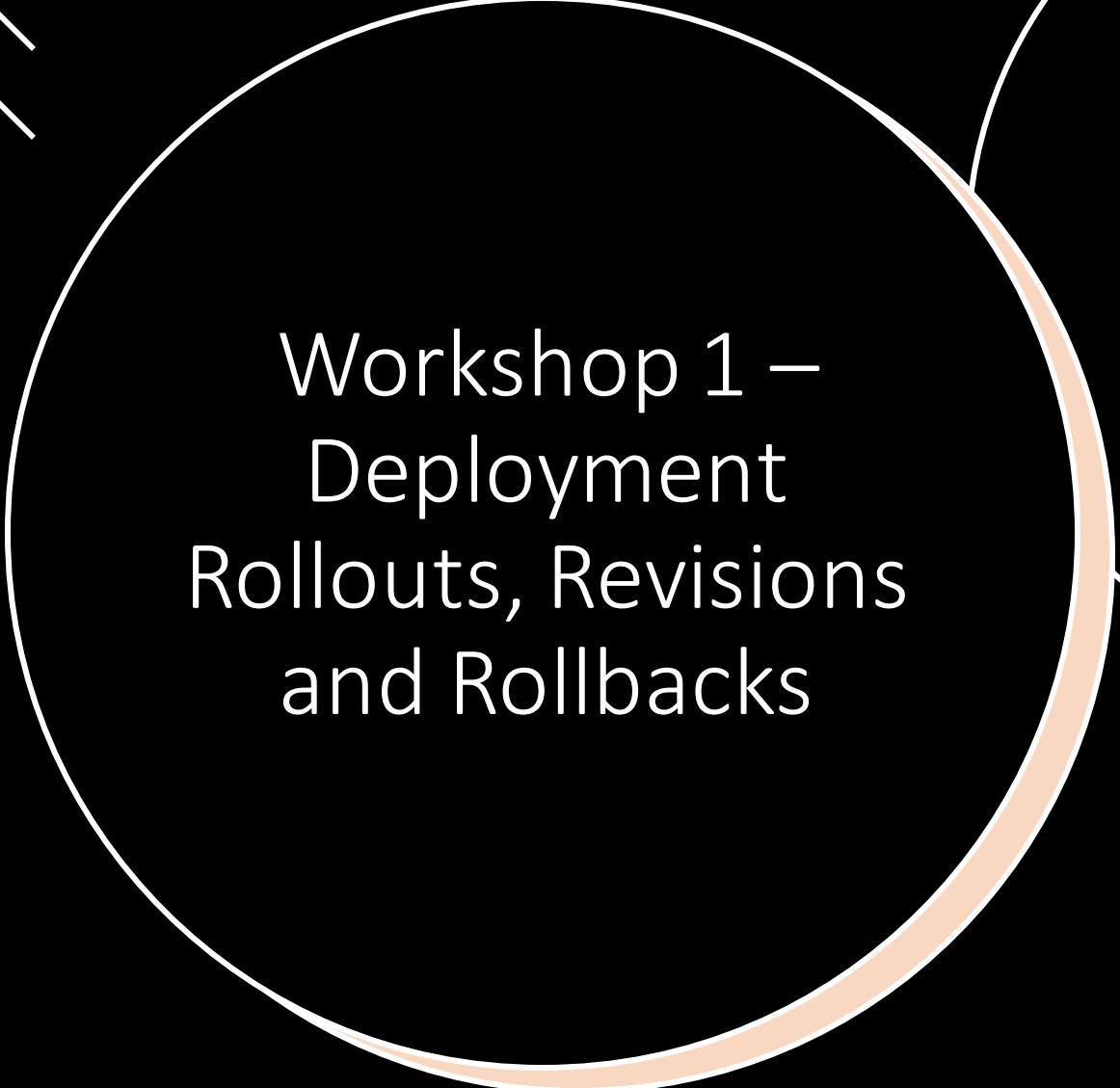
- Default strategy
- No downtime



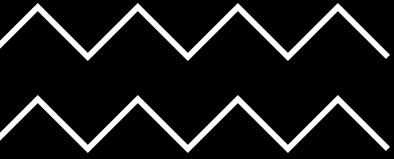
Rollout Strategies: Recreate

- Application downtime
- Should be specified explicitly





Workshop 1 – Deployment Rollouts, Revisions and Rollbacks



Working with deployment

```
# Copy deployment file to the machine running kind K8s cluster  
scp -i week6 week6Session1/deployment.yaml 44.201.36.90:/tmp  
  
# Log into EC2, start docker daemon on EC2  
sudo systemctl start docker  
  
# Create new namespace and create an nginx deployment  
k create ns nginx  
k apply -f /tmp/deployment.yaml  
  
# Let's see the status of our deployment  
k rollout status deployment.apps/deployment-nginx -n nginx  
k get all -n nginx  
  
# Why is there only one rollout?  
k rollout history deployment.apps/deployment-nginx -n nginx
```

Creating Revisions

```
# Re-create the deployment and record change history  
k apply -f /tmp/deployment.yaml --record  
k rollout history deployment.apps/deployment-nginx -n nginx  
# Change nginx image version in the YAML  
k apply -f /tmp/deployment.yaml --record  
K get pods -n nginx  
# Change again with "kubectl edit"  
k edit deployment.apps/deployment-nginx -n nginx  
k rollout status deployment.apps/deployment-nginx -n nginx
```

```
deployment.apps/deployment-nginx  
REVISION  CHANGE-CAUSE  
1        kubectl apply --filename=/tmp/deployment.yaml --record=true  
2        kubectl apply --filename=/tmp/deployment.yaml --record=true  
3        kubectl apply --filename=/tmp/deployment.yaml --record=true
```

Rollback

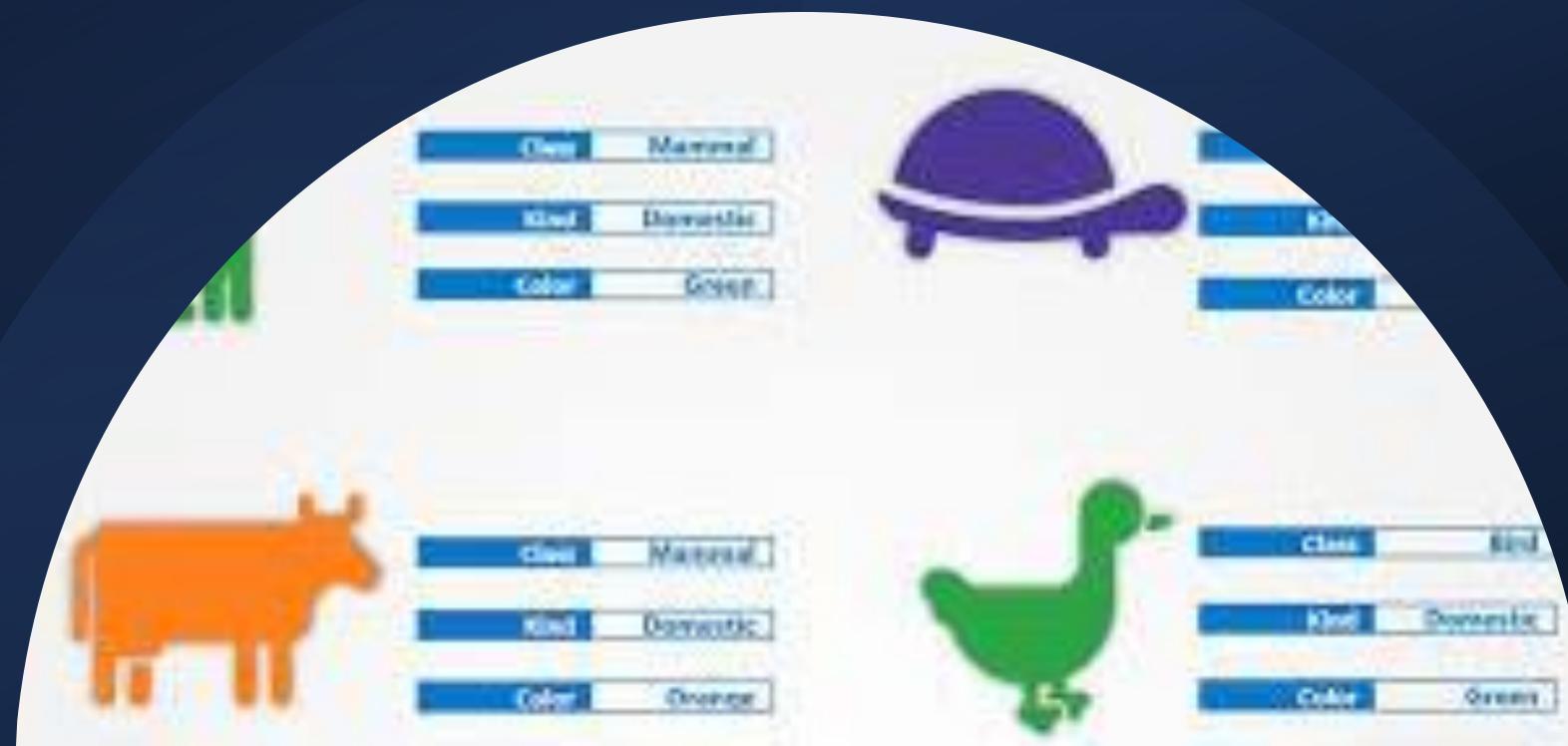
```
# Rollback one revision  
k rollout undo deployment.apps/deployment-nginx -n nginx  
k rollout history deployment.apps/deployment-nginx -n nginx  
  
# Update deployment manifest to point to non-existent version on nginx  
k edit deployment.apps/deployment-nginx -n nginx  
# Are the pods replaced with a new version?  
k get deployment.apps/deployment-nginx -n nginx  
  
# Are the previous pods terminated?  
k get pods -n nginx  
# Let's roll back the unsuccessful deployment  
k rollout undo deployment.apps/deployment-nginx -n nginx  
k get pods -n nginx
```

-bash-4.2\$ k get pods -n nginx	READY	STATUS	RESTARTS	AGE
NAME				
deployment-nginx-77d5dbb5f-fmflx	0/1	ImagePullBackOff	0	2m38s
deployment-nginx-77d5dbb5f-hk8vv	0/1	ImagePullBackOff	0	2m38s
deployment-nginx-77d5dbb5f-v4t27	0/1	ImagePullBackOff	0	2m37s
deployment-nginx-85f69c8b88-6tgzz	1/1	Running	0	5m18s
deployment-nginx-85f69c8b88-8mtxg	1/1	Running	0	5m18s
deployment-nginx-85f69c8b88-n42v7	1/1	Running	0	5m20s
deployment-nginx-85f69c8b88-p4bfb	1/1	Running	0	5m21s
deployment-nginx-85f69c8b88-qv9th	1/1	Running	0	5m21s

Workshop 1 – The End



Workshop 2 – Using Labels to Organize K8s Pods





Assigning Labels to Pods

```
$ kubectl create -f kubia-manual-with-labels.yaml  
  
$ kubectl get po --show-labels  
$ kubectl get po -L creation_method,env  
$ kubectl label po kubia-manual creation_method=manual  
$ kubectl label po kubia-manual-v2 env=debug --overwrite  
$ kubectl get po -L creation_method,env
```

Using Labels

```
$ kubectl get po -l  
creation_method=manual
```

```
$ kubectl get po -l env
```

```
$ kubectl get po -l '!env'
```

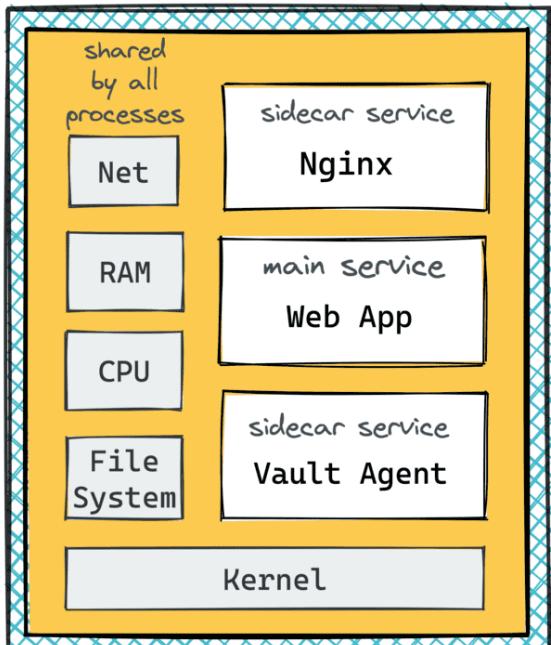




Workshop 2 – The End

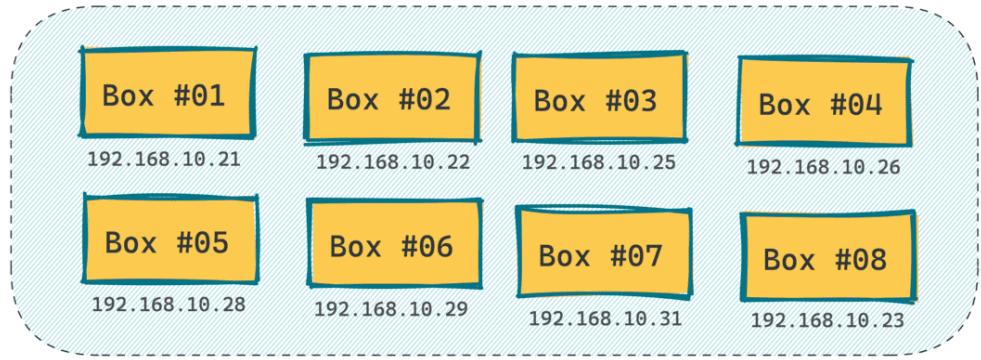
Services in the Traditional Environment

Virtual Machine - a "Box"
...or real!



e.g. 192.168.10.5

Service - a named group of identical "Boxes"
distributed!



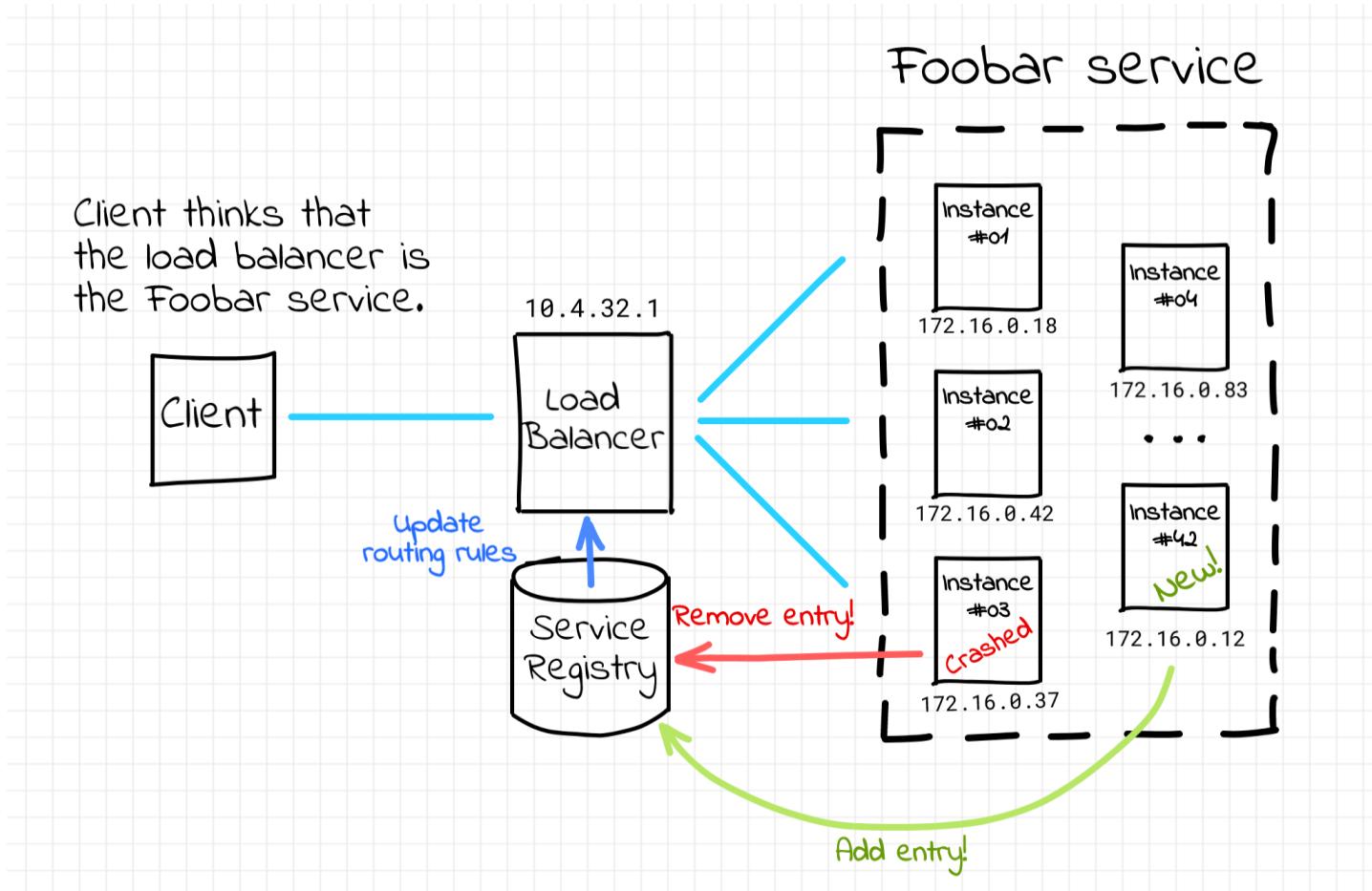
a roster of boxes
is maintained behind
a service name

foo.bar.svc.domain

clients access
boxes using the
service name

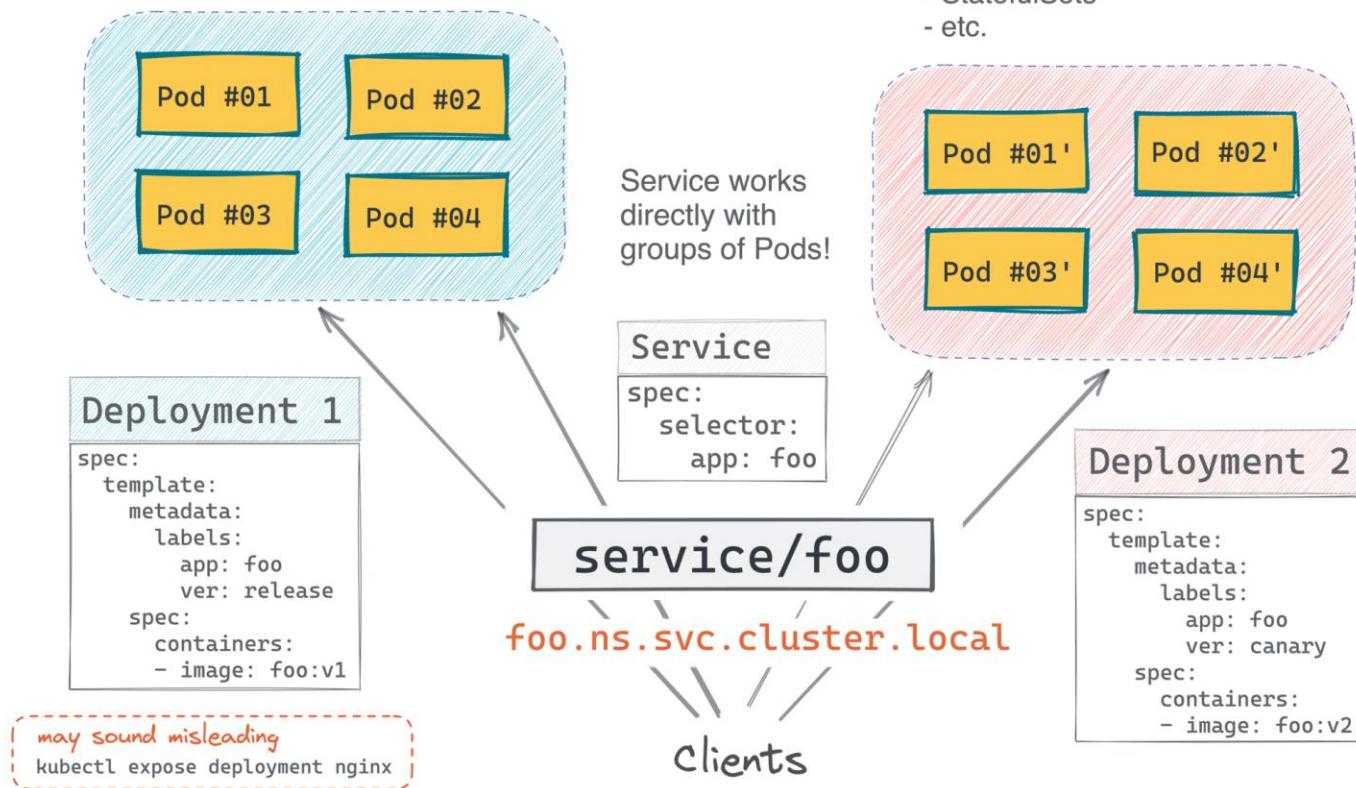
Clients

Traditional Service Discovery – Server-Side



K8s Service - Similarly Labeled Pods

Kubernetes Service - a named group of similarly-labeled Pods



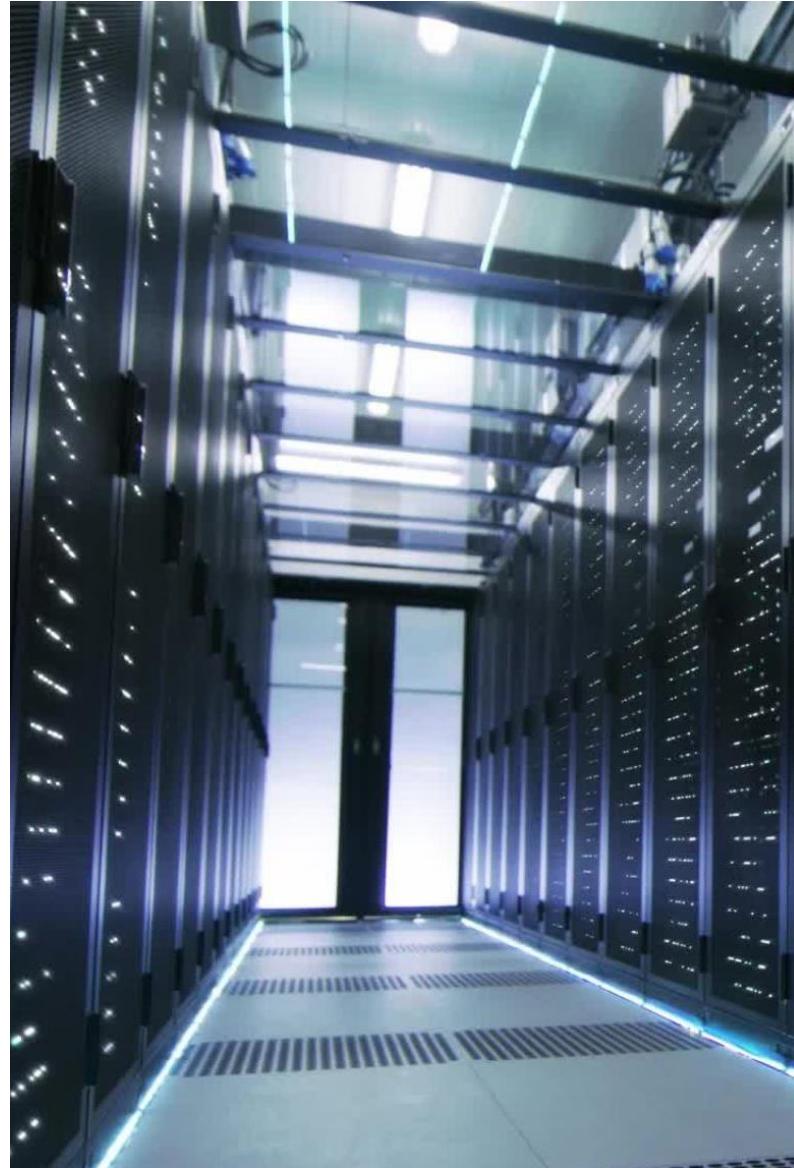
Exposing containers

Imagine standard client-server configuration. How would we configure the access points for the clients?

Challenges:

- We cannot configure pod's IP – pods are ephemeral
- Clients do not know server IP upfront
- Multiple pods can provide the same service – clients have no way to know

We need a single IP endpoint



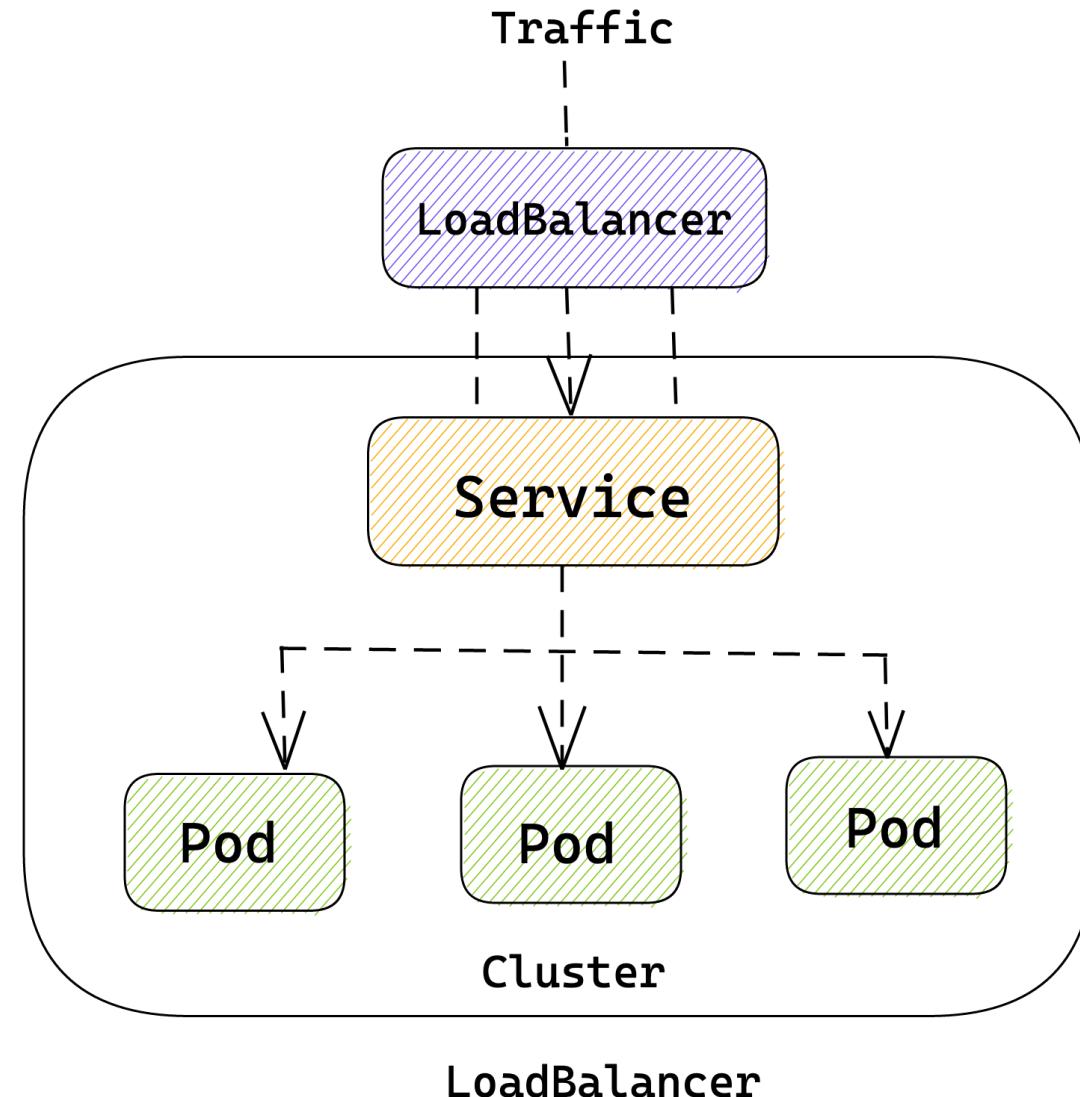


Exposing Containers in Kubernetes

- We can connect to our pods using their IP address
- Then we need to figure out a lot of things:
 - how do we look up the IP address of the pod(s)?
 - how do we connect from outside the cluster?
 - how do we load balance traffic?
 - what if a pod fails?
- Kubernetes has a resource type named *Service*
- Services address all these questions!

Introducing Kubernetes Services

-
- Immutable point of entry to a group of pods providing the same service
 - Clients open connections to this IP/port and client's requests are routed to pod backing the service
 - Services are automatically added to an internal DNS zone



Why Services?



We don't need to look up the IP address of the pod(s)



There are multiple service types; some of them allow external traffic



(e.g. LoadBalancer and NodePort)



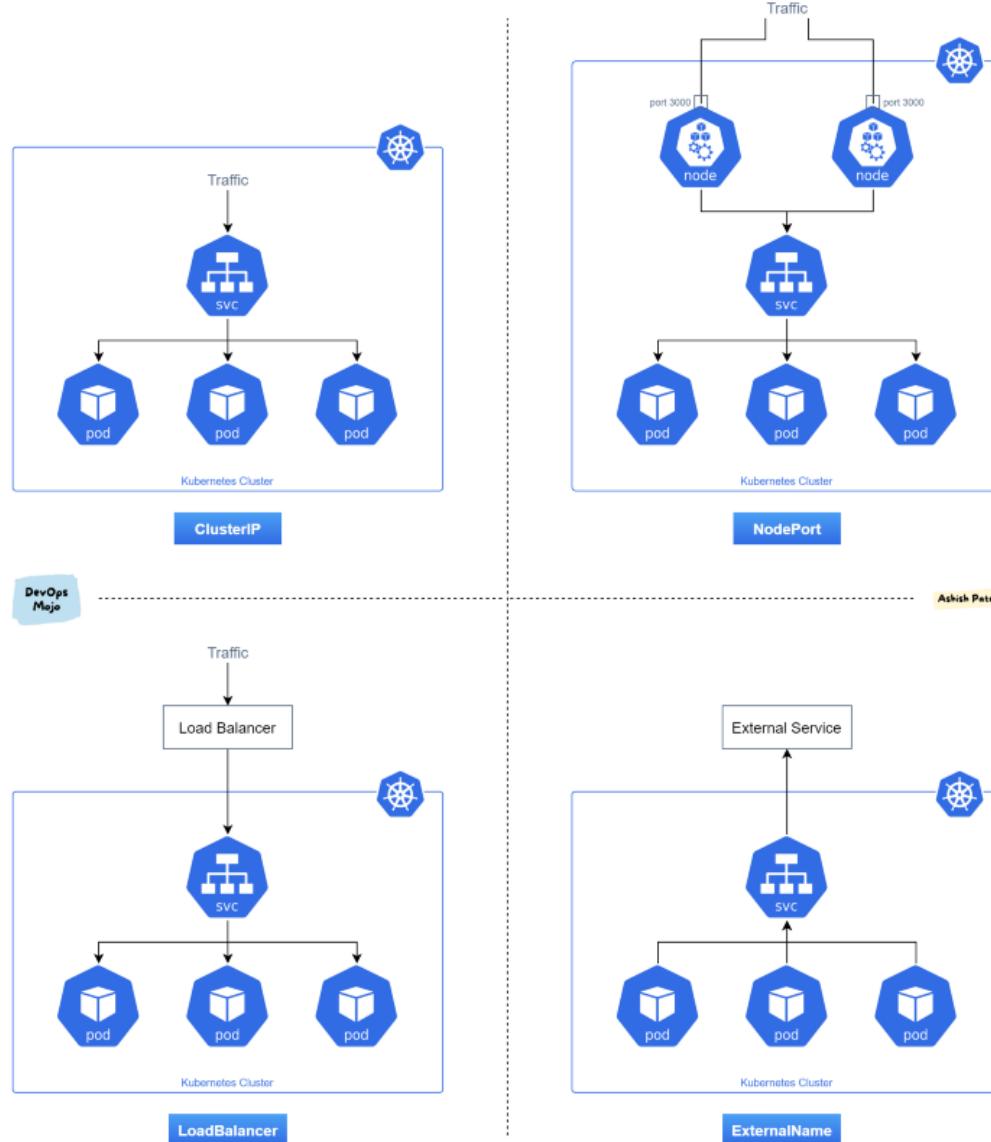
Services provide load balancing



Service addresses are independent from pods' addresses

Types of Services

- There are different types of services:
 - ClusterIP, NodePort, LoadBalancer, ExternalName
- There are also *headless services*
- Services can also have optional *external IPs*
- There is also another resource type called *Ingress*
- (specifically for HTTP services)



Creating Service Imperatively

```
# Create a deployment with single replica
```

```
kubectl create deploy nginx-deployment --image=nginx
```

```
# Scale the deployment to 3 replicas
```

```
kubectl scale deploy nginx-deployment --replicas=3  
deployment
```

```
# Expose the container on port 80
```

```
kubectl expose deployment nginx-deployment --port=80
```

Service Manifest

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: kubia
```

The port this service
will be available on

The container port the
service will forward to

All pods with the app=kubia
label will be part of this service.

```
kind: Pod
spec:
  containers:
    - name: kubia
      ports:
        - name: http
          containerPort: 8080
        - name: https
          containerPort: 8443
apiVersion: v1
kind: Service
spec:
  ports:
    - name: http
      port: 80
      targetPort: http
    - name: https
      port: 443
      targetPort: https
```

Container's port
8080 is called http

Port 8443 is called https.

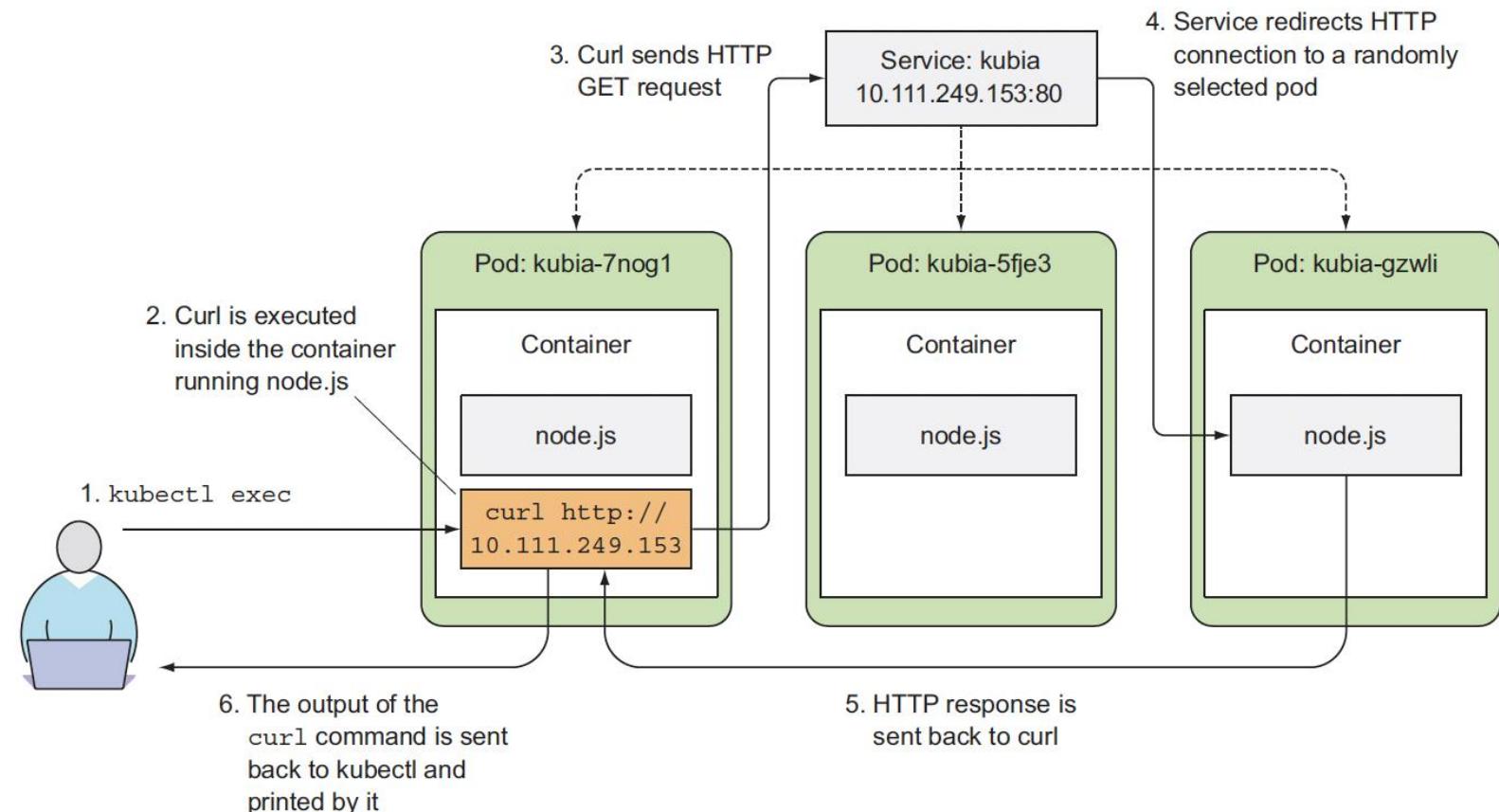
Port 80 is mapped to the
container's port called http.

Port 443 is mapped to the container's
port, whose name is https.

Using Named Ports

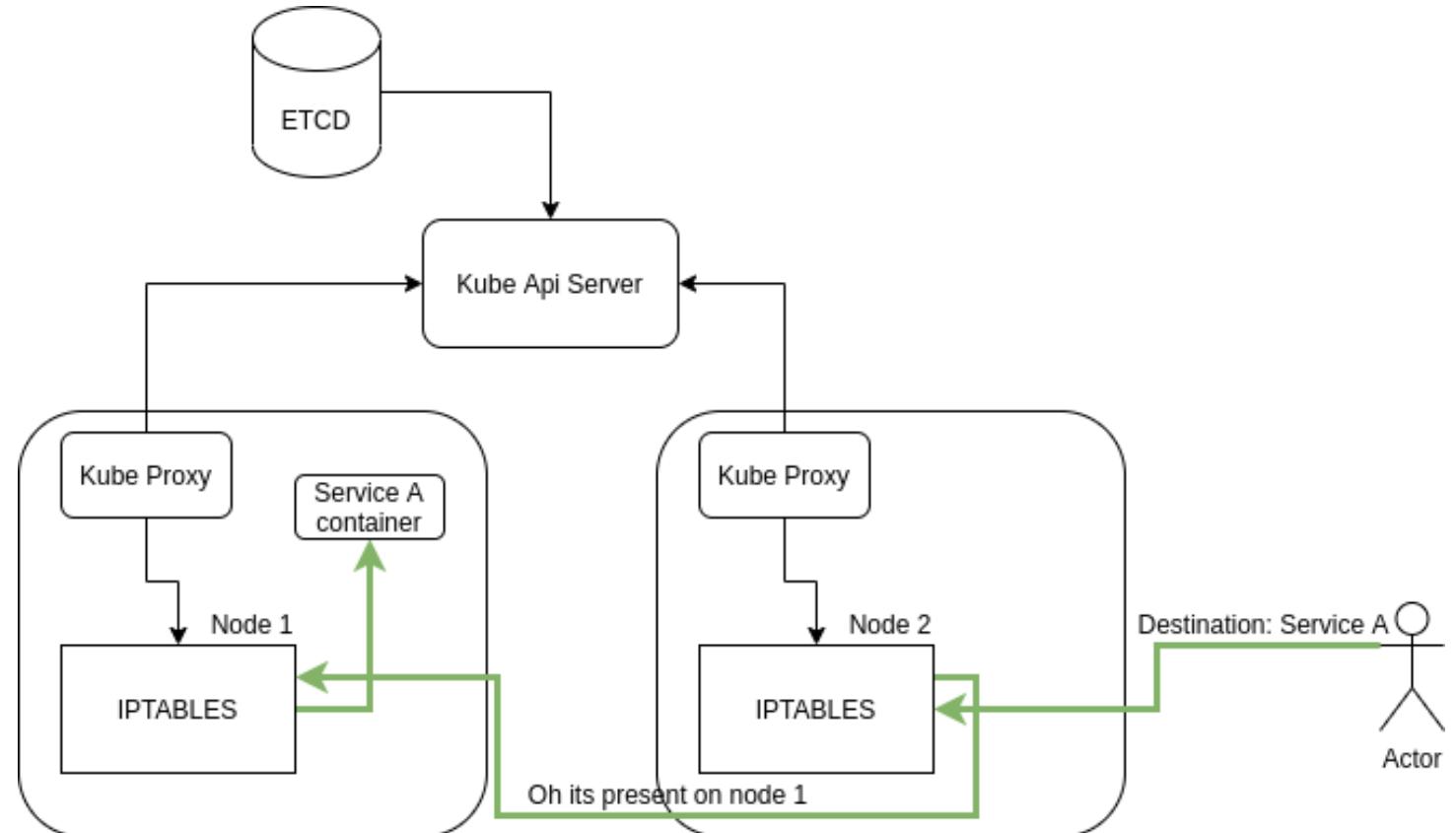
Accessing the Service

- Services of type ClusterIP are not externally accessible
- We can access this service in different ways
 - Run a pod in the cluster that will access this service
 - Execute `curl` with `kubectl exec` command
 - `ssh` into one of the Kubernetes nodes and use the `curl` command.



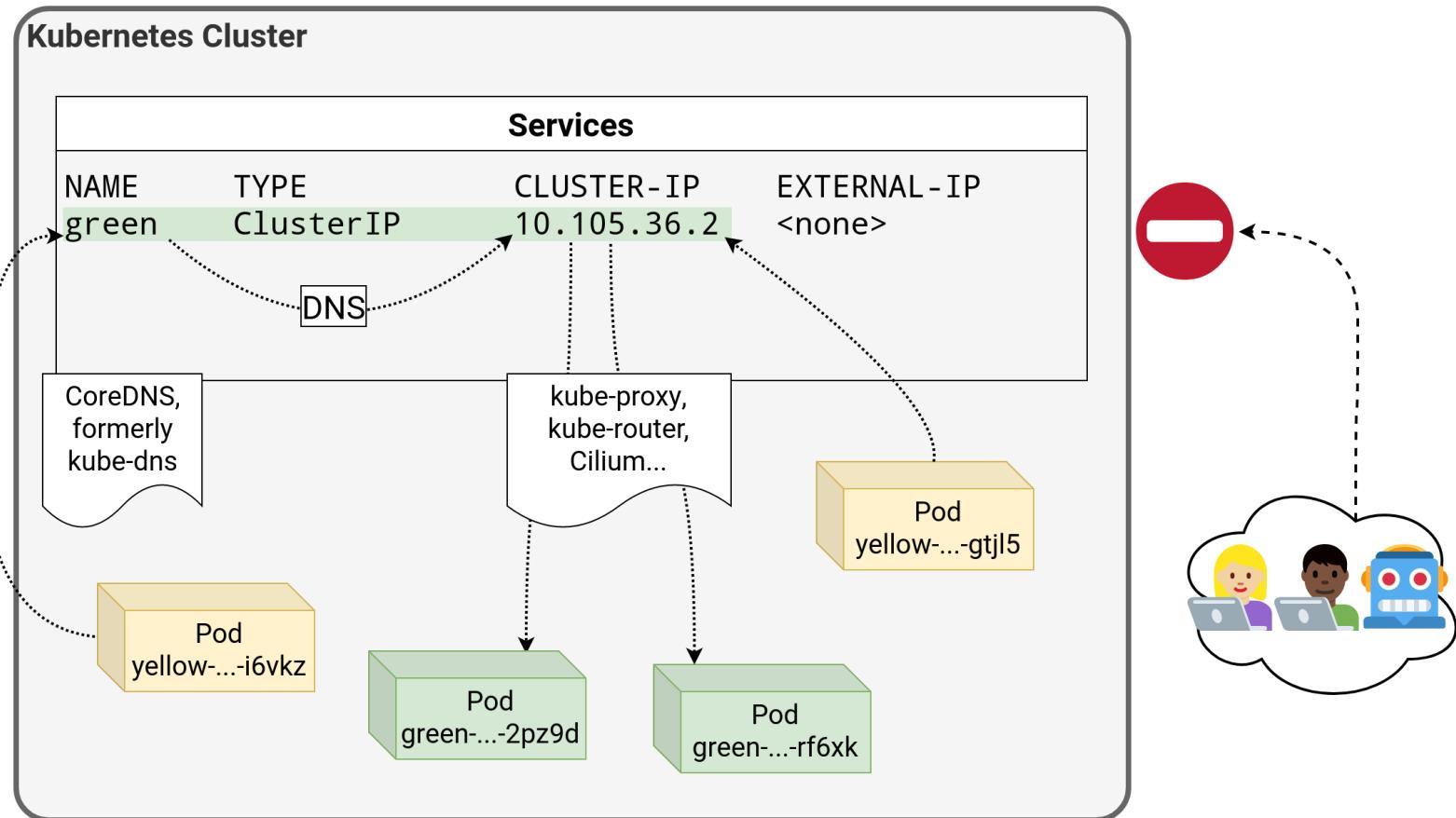
How does it work?

- Kube-proxy modifies IPTABLES on each node to create rules for the services we define.
- Kube-proxy does not *do* the actual routing



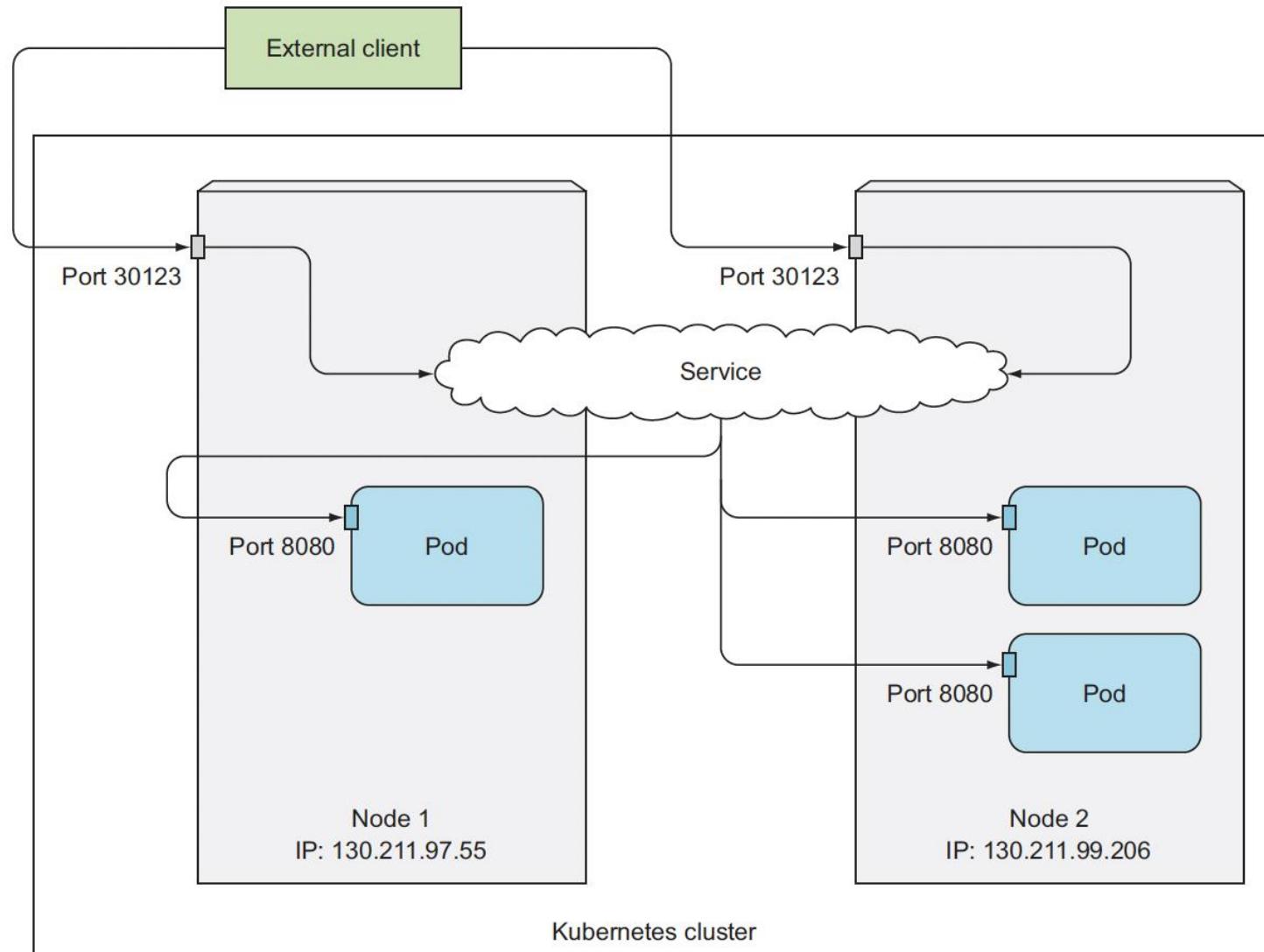
Service Type: ClusterIP

- It's the default service type
 - A virtual IP address is allocated for the service
 - (in an internal, private range; e.g. 10.96.0.0/12)
 - This IP address is reachable only from within the cluster (nodes and pods)
 - Our code can connect to the service using the original port number
 - Perfect for internal communication, within the cluster



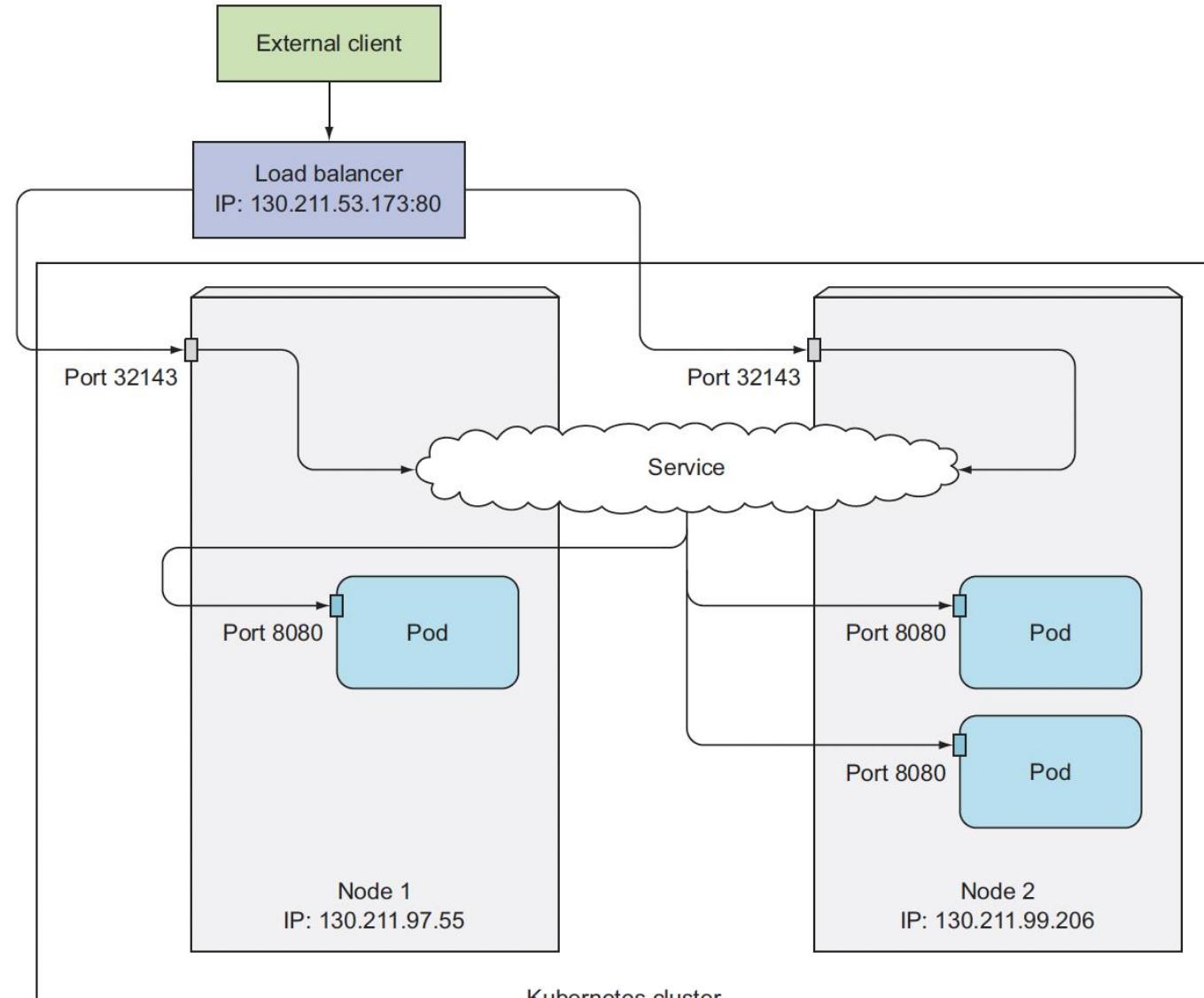
Service Type: NodePort

- A port number is allocated for the service
- (by default, in the 30000-32767 range)
- That port is made available *on all our nodes* and anybody can connect to it
- (we can connect to any node on that port to reach the service)
- Our code needs to be changed to connect to that new port number
- Under the hood: `kube-proxy` sets up a bunch of `iptables` rules on our nodes
- Sometimes, it's the only available option for external traffic (e.g. most clusters deployed with `kubeadm` or on-premises)
- Automatically will create a ClusterIP service

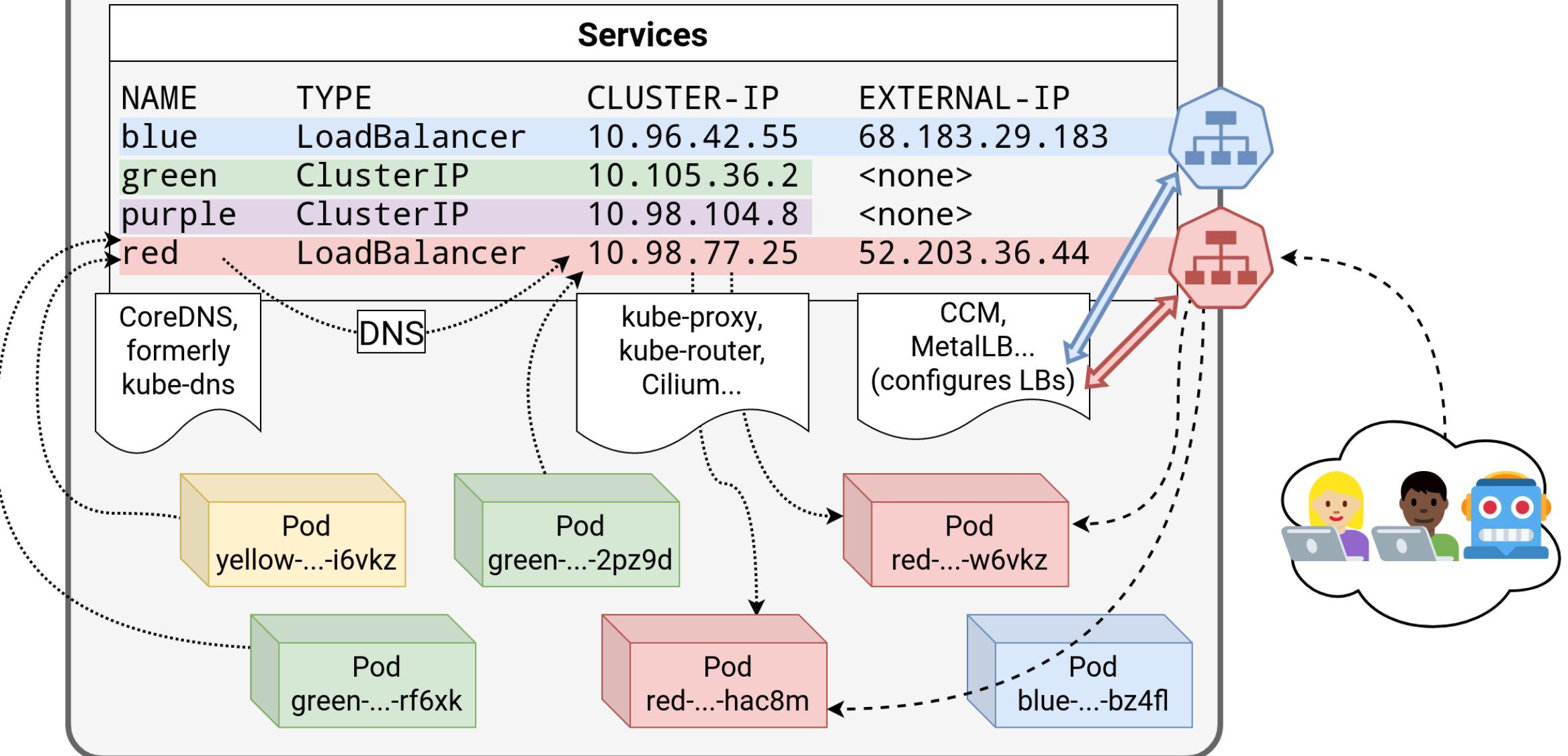


Service Type: LoadBalancer

- An external load balancer is allocated for the service
- This is available only when the underlying infrastructure provides some kind of "load balancer as a service"
- Each service of that type will typically cost a little bit of money
- Ideally, traffic would flow directly from the load balancer to the pods
- In practice, it will often flow through a NodePort first



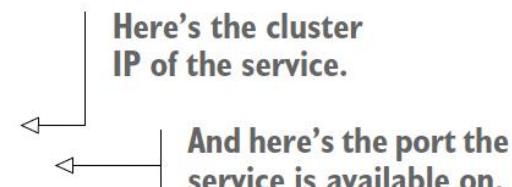
Kubernetes Cluster



Discovering Services

- Through Environment Variables
- Through DNS servers and FQDNs

```
$ kubectl exec -it kubia-3inly bash  
root@kubia-3inly:/#  
  
# Inside the container  
root@kubia-3inly:/# curl http://kubia.default.svc.cluster.local  
You've hit kubia-5asi2  
  
root@kubia-3inly:/# curl http://kubia.default.svc.cluster.local  
You've hit kubia-3inly  
  
root@kubia-3inly:/# curl http://kubia.default.svc.cluster.local  
You've hit kubia-8awf3  
  
root@kubia-3inly:/# cat /etc/resolv.conf  
search default.svc.cluster.local svc.cluster.local cluster.local ..  
  
$ kubectl exec kubia-3inly env  
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin  
HOSTNAME=kubia-3inly  
KUBERNETES_SERVICE_HOST=10.111.240.1  
KUBERNETES_SERVICE_PORT=443  
...  
KUBIA_SERVICE_HOST=10.111.249.153  
KUBIA_SERVICE_PORT=80  
...
```



Here's the cluster IP of the service.

And here's the port the service is available on.

Service Endpoints

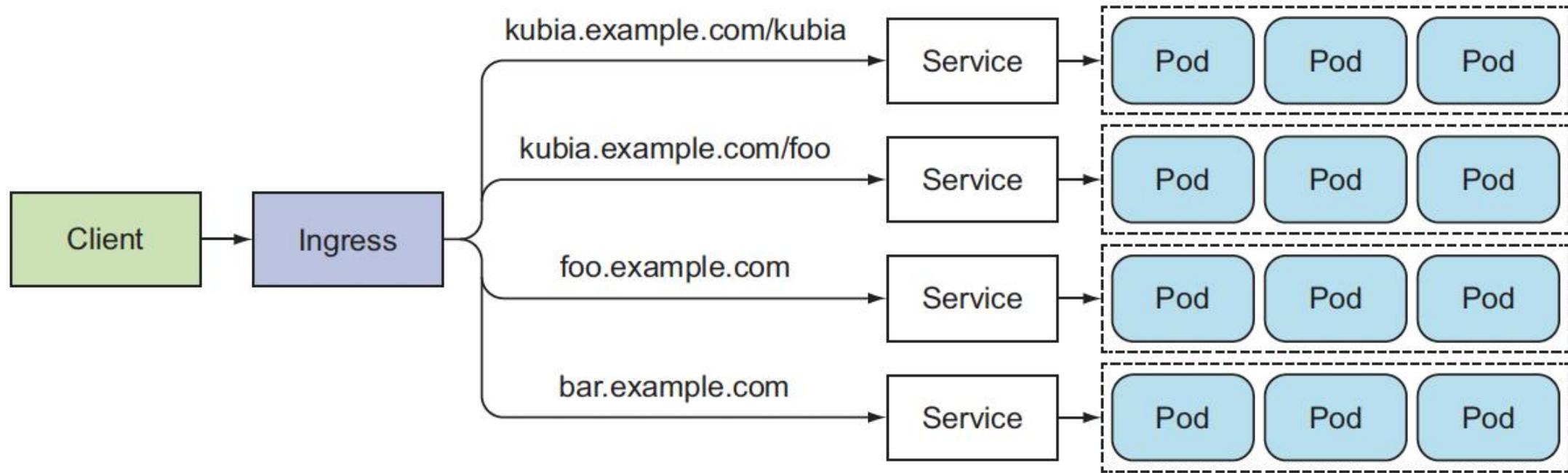
- An Endpoints resource (yes, plural) is a list of IP addresses and ports exposing a service.
- The selector is used to build a list of IPs and ports, which is then stored in the Endpoints resource.
- When a client connects to a service, the service proxy selects one of those IP and port pairs and redirects the incoming connection to the server listening at that location.

```
$ kubectl describe svc kubia
Name:           kubia
Namespace:      default
Labels:         <none>
Selector:       app=kubia
Type:          ClusterIP
IP:            10.111.249.153
Port:          <unset>  80/TCP
Endpoints:     10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080
Session Affinity: None
```

The service's pod selector is used to create the list of endpoints.

The list of pod IPs and ports that represent the endpoints of this service

Ingress Resource



Understanding Ingress

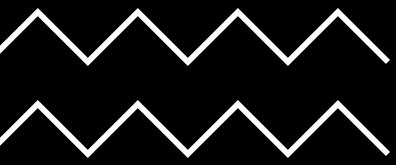
- Layer 7 load balancer
 - Provides cookie-based session affinity
 - Level 7 routing
- Requires ingress controller to be installed in the cluster

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
spec:
  rules:
    - host: kubia.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: kubia-nodeport
              servicePort: 80
```

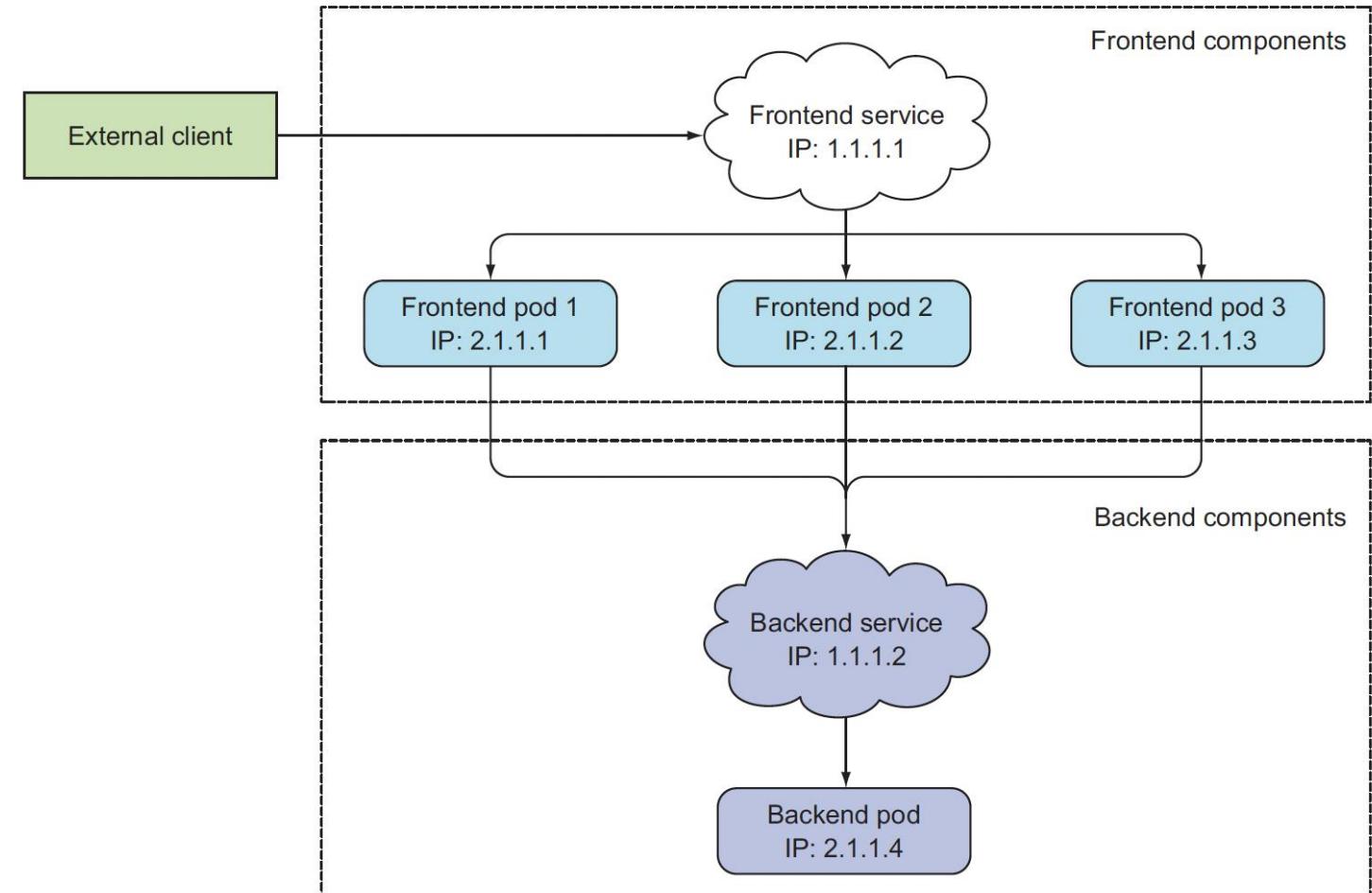
This Ingress maps the **kubia.example.com** domain name to your service.

All requests will be sent to port 80 of the **kubia-nodeport** service.

Workshop 3 – Using K8s Services



Example: 2-Tier Web application



Create the Namespace

```
# Copy all the deployment files to the EC2 with our kind cluster  
scp -i week6 week6Session2/* 100.27.36.251:/tmp/
```

```
# log into you EC2  
ssh -i week6 [EC2 public IP]
```

```
# Verify the cluster is running  
k get nodes
```

```
# Create a namespace for our Guestbook application  
k create ns guestbook
```

Deploying application components: MongoDB

```
# Deploy application component – MongoDB  
k apply -f /tmp/mongo-deployment.yaml -n guestbook
```

```
# Verify the successful deployment  
k rollout status deployment.apps/mongo -n guestbook  
k describe -n guestbook deployment.apps/mongo
```

```
# Create service to expose mongoDB to internal cluster users  
k apply -f /tmp/mongo-service.yaml -n guestbook
```

Explain: what type of service did we create? Why is this the right type of service to use with Mongo DB?

```
# Verify the service is created  
k get svc -n guestbook
```

```
# Examine Service Endpoints  
K describe service mongo -n guestbook
```

Deploy Application Components: Frontend

```
# Create frontend deployment  
k apply -f /tmp/frontend-deployment.yaml -n guestbook
```

```
# Verify the successful deployment  
k rollout status deployment.apps/frontend -n guestbook  
k describe -n guestbook deployment.apps/frontend
```

```
# Create service to expose mongoDB to internal cluster users  
k apply -f /tmp/frontend-service.yaml -n guestbook
```

```
# Verify the service is created  
k get svc -n guestbook
```

Connecting to GuestBook Application with ClusterIP

```
# Our ClusterIP service listens on port 80. We will forward local port 8080 to service port 80 with port-forward
```

```
k port-forward svc/frontend 8080:80 -n guestbook
```

```
# Verify access to the service from another terminal window
```

```
curl localhost:8080
```

```
# Open port 8080 in the Security Group and try to access the application via [PublicIP]:8080 in the browser.
```

Explain: Was it successful? Why?

Connecting to GuestBook Application with NodePort

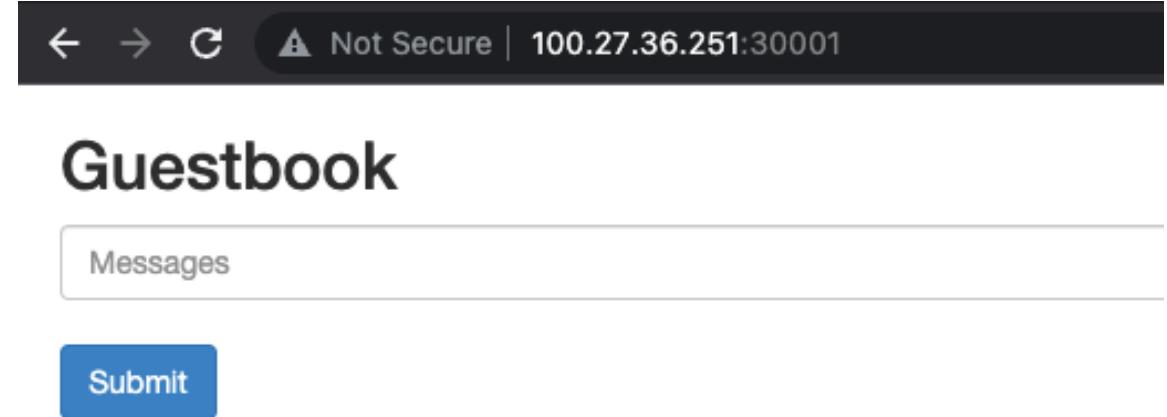
```
# Update the service type to NodePort
```

```
k apply -f /tmp/frontend-serviceNodePort.yaml -n  
guestbook
```

```
# Verify that the port forward is still running, and we  
can still connect via kubectl port forward
```

```
curl localhost:8080
```

```
# Open port 30001 in the Security Group and try  
connection from the browser with [EC2 Public  
IP]:30001
```



Connecting to GuestBook Application with NodePort – change the NodePort to 30100

```
# Try connecting with the different NodePort, e.g. 30100. Update the NodePort in the  
frontend-serviceNodePort.yaml and update the service
```

```
k apply -f /tmp/frontend-serviceNodePort.yaml -n guestbook
```

```
# Verify we can still connect with ClusterIP
```

```
curl localhost:8080
```

```
# Update EC2 Security Group and try connecting to the application from the browser  
using [EC2 Public IP]:30100
```

Explain: Was it successful? Why?

Frontend connection to MongoDB – *how does it work?*

- We will research Pod template
- Google paulczar/gb-frontend
- Find it on DockeHub
- Look at the digest
- Look for guestbook.php and mongodb
- <https://github.com/kevinraymond/php-guestbook/blob/main/app/guestbook.php>
- The answer

```
// this references the "mongo" service
$messages = (new MongoDB\Client('mongodb://mongo'))->guestbook->messages;
$method = $_SERVER['REQUEST_METHOD'];
```

- Explain: Would our application work if we give our MongoDB service a *different* name?



Changing MongoDB Service Name

```
# Delete the mongo service and frontend deployment and service
k delete svc frontend -n guestbook
k delete deployment frontend -n guestbook
k delete svc mongo -n guestbook

# Update the /tmp/mongo-service.yaml and rename the service
# Apply the changes
k apply -f /tmp/mongo-service.yaml -n guestbook
k apply /tmp/frontend-deployment.yaml
k apply -f /tmp/frontend-service.yaml -n guestbook

# Verify we can still connect with ClusterIP
curl localhost:8080
-bash-4.2$ curl localhost:8080
curl: (7) Failed to connect to localhost port 8080 after 0 ms: Connection refused
```



Workshop 3 – The End



Workshop 4 – DNS in K8s



Creating a deployment for our HTTP server

```
# Create a deployment for this very lightweight HTTP server:
```

```
k create deployment blue --image=jpetazzo/color
```

```
# Scale it to 10 replicas:
```

```
k scale deployment blue --replicas=10
```

```
# Pods are created
```

```
k get pods -l app=blue
```

Exposing our deployment – Create ClusterIP service

```
# Expose the HTTP port of our server  
k expose deployment blue --port=80
```

```
# Look up which IP address was allocated  
k get svc  
k get svc blue -o yaml
```

Test the Access to the Application

```
# Test the service from inside the K8s "node" that runs as a container in our environment  
docker exec -it [CONTAINER ID] /bin/bash
```

```
# Send curl request to ClusterIP from the bash prompt of the container that simulates our master  
node. Try it a few times! Our requests are load balanced across multiple pods.  
curl [ClusterIp of blue service]
```

```
# Result
```

```
root@kind-control-plane:/# curl 10.96.136.226
```

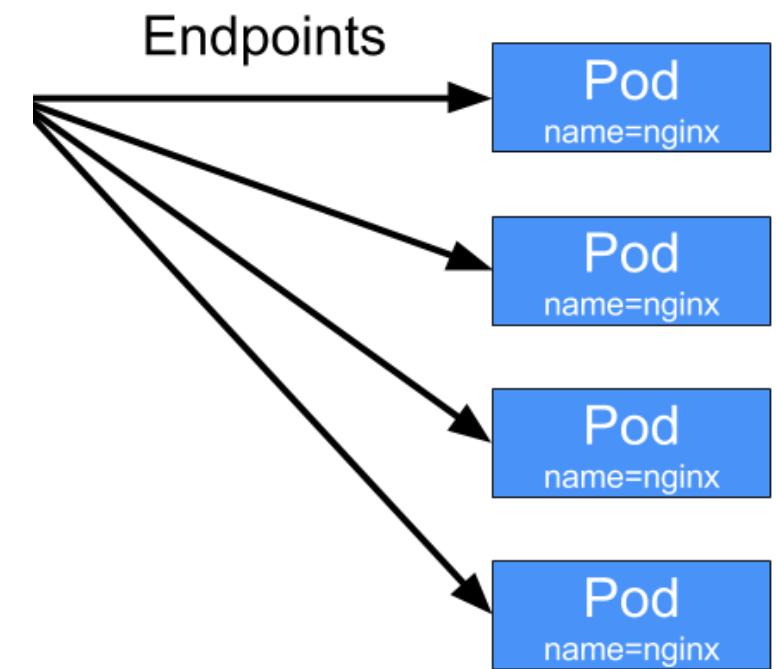
⌚ This is pod default/blue-796f87cc56-9dmrx on linux/amd64, serving / for 10.244.0.1:44398.

Services and endpoints

- A service has a number of "endpoints"
- Each endpoint is a host + port where the service is available
- The endpoints are maintained and updated automatically by Kubernetes

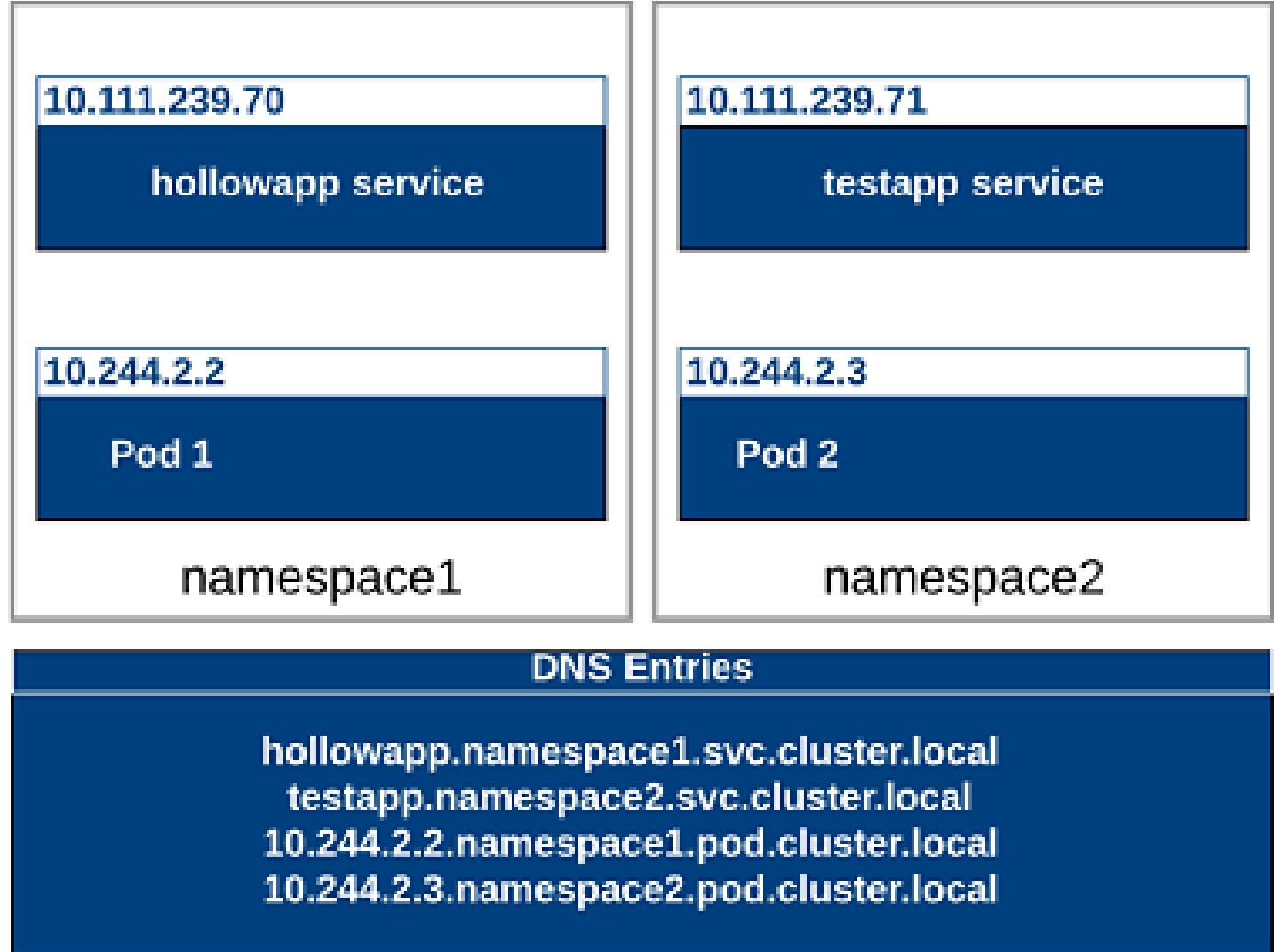
```
# Get service Endpoints  
k describe service blue  
  
k get endpoints  
k describe endpoints blue  
k get endpoints blue -o yaml
```

```
# Compare to the output below  
k get pods -l app=blue -o wide
```



DNS in K8s

- In the kube-system namespace, there should be a service named kube-dns
- This is the internal DNS server that can resolve service names
- The default domain name for the service we created is default.svc.cluster.local



Examine DNS in the pod

```
# log into blue pod
```

```
k exec -it blue-796f87cc56-7tbfc -- /bin/sh
```

```
# Examine the nameservers defined in the pod.
```

```
# What is running on IP is 10.96.0.10?
```

```
# What are the entries in the resolv.conf file?
```

```
cat /etc/resolv.conf
```

```
search default.svc.cluster.local svc.cluster.local cluster.local ec2.internal
```

```
nameserver 10.96.0.10
```

```
options ndots:5
```



Workshop 4 – The End





Additional Resources

- <https://kubernetes.io/docs/concepts/services-networking/service/>
- <https://medium.com/devops-mojo/kubernetes-service-types-overview-introduction-to-k8s-service-types-what-are-types-of-kubernetes-services-ea6db72c3f8c>
- <https://docs.vmware.com/en/VMware-Cloud-Foundation/services/vcf-developer-ready-infrastructure-v1/GUID-6F184EC5-AFC1-4D0A-A5D5-1E31EE938438.html>
- <https://container.training/kube-selfpaced.yml.html#273>
- <https://www.bookstack.cn/read/kubernetes-1.21-en/59a313e057e11c1b.md>
- <https://arthurchiao.art/blog/cracking-k8s-node-proxy/>
- <https://iximiuz.com/en/posts/service-discovery-in-kubernetes/>

Volumes: Attaching Disk Storage to Containers

Week 7

Agenda

- Volumes
 - emptyDir
 - hostPath
- PersistentVolumes (PV) and PersistentVolumeClaims (PVC)
- StorageClass

K8s Volumes

- Unlike Network, processes and CPU, filesystems are unique per container
- Used to persist data between container invocations
- Can be shared by containers in a pod
- Defined as part of the pod spec
- Share pod's lifecycle
 - Initiated when pod starts
 - Terminated when a pod is terminated

Volumes

- Volumes are special directories that are mounted in containers
- Volumes can have many different purposes:
 - share files and directories between containers running on the same machine
 - share files and directories between containers and their host
 - centralize configuration information in Kubernetes and expose it to containers
 - manage credentials and secrets and expose them securely to containers
 - store persistent data for stateful services

Volumes vs PersistentVolumes

- Volumes and Persistent Volumes are related, but very different!
- *Volumes*:
 - appear in Pod specifications (we'll see that in a few slides)
 - do not exist as API resources (**cannot** do kubectl get volumes)
- *Persistent Volumes*:
 - are API resources (**can** do kubectl get persistentvolumes)
 - correspond to concrete volumes (e.g. on a SAN, EBS, etc.)
 - cannot be associated with a Pod directly; but through a Persistent Volume Claim
 - won't be discussed further in this section

Volume Types

- `emptyDir`—A simple empty directory used for storing transient data.
- `hostPath`—Used for mounting directories from the worker node’s filesystem into the pod.
- `gitRepo`—A volume initialized by checking out the contents of a Git repository.
- `nfs`—An NFS share mounted into the pod.
 - `gcePersistentDisk` (Google Compute Engine Persistent Disk)
 - `awsElasticBlockStore` (Amazon Web Services Elastic Block Store Volume)
 - `azureDisk` (Microsoft Azure Disk Volume)—Used for mounting cloud provider-specific storage.

Volume Types, Cont

- cinder, cephfs, iscsi, flocker, glusterfs, quobyte, rbd, flexVolume, vsphereVolume, photonPersistentDisk, scaleIO—Used for mounting other types of network storage.
- configMap, secret, downward API—Special types of volumes used to expose certain Kubernetes resources and cluster information to the pod.
- persistentVolumeClaim—A way to use a pre- or dynamically provisioned persistent storage.

Volume Type: emptyDir

- We will start with the simplest Pod manifest we can find
- We will add a volume to that Pod manifest
- We will mount that volume in a container in the Pod
- By default, this volume will be an emptyDir (an empty directory)
- It will "shadow" the directory where it's mounted

Volume Lifecycle

- The lifecycle of a volume is linked to the pod's lifecycle
- This means that a volume is created when the pod is created
- This is mostly relevant for emptyDir volumes
- (other volumes, like remote storage, are not "created" but rather "attached")
- A volume survives across container restarts
- A volume is destroyed (or, for remote storage, detached) when the pod is destroyed



WORKSHOP 1 – USING EMPTYDIR VOLUME TO SHARE DATA BETWEEN CONTAINERS IN THE POD

GOALS

- Deployment of containerized application that uses emptyDir Volume to share data between containers

DESCRIPTION

- Application is running as two containers in the same pod
- One process is generating random fortune every 10 seconds and writes it to a shared volumes
- The same volume is mounted into nginx container as /usr/share/nginx/html. Nginx serves the fortunes from this folder

```
Executable File | 11 lines (9 sloc) | 181 Bytes

1  #!/bin/bash
2  trap "exit" SIGINT
3  mkdir /var/htdocs
4
5  while :
6  do
7      echo $(date) Writing fortune to /var/htdocs/index.html
8      /usr/games/fortune > /var/htdocs/index.html
9      sleep 10
10 done
11
```



Description

Application is running as two containers in the same pod

One process is generating random fortune every 10 seconds and writes it to a shared volumes

The same volume is mounted into nginx container as /usr/share/nginx/html. Nginx serves the fortunes from this folder

Pod Manifest

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune
spec:
  containers:
    - image: luksa/fortune
      name: html-generator
      volumeMounts:
        - name: html
          mountPath: /var/htdocs
    - image: nginx:alpine
      name: web-server
      volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
          readOnly: true
  ports:
    - containerPort: 80
      protocol: TCP
  volumes:
    - name: html
      emptyDir: {}
```

The first container is called **html-generator** and runs the **luksa/fortune** image.

The volume called **html** is mounted at **/var/htdocs** in the container.

The second container is called **web-server** and runs the **nginx:alpine** image.

The same volume as above is mounted at **/usr/share/nginx/html** as **read-only**.

A single **emptyDir** volume called **html** that's mounted in the two containers above

Deploy and Access the application

```
# Create a namespace  
$ k create ns week8  
$ k apply -f fortune_pod.yaml -n week8  
$ k get all -n week8  
  
# Deploy the pod  
# Access the app via port forwarding  
$ kubectl port-forward fortune 8080:80 -n week8  
  
$ curl http://localhost:8080
```

```
environment $ curl localhost:8081  
xpensive sports car, fine-tuned and well-built, Port  
and gorgeous, her red jumpsuit moulding her body, wh  
vers in July, her hair as dark as new tires, her eye  
ocaps, and her lips as dewy as the beads of fresh ra  
woman driven -- fueled by a single accelerant -- an  
n who wouldn't shift from his views, a man to steer  
d: a man like Alf Romeo.  
-- Rachel Sheeley, winner  
  
ball blocking the drain of the shower reminded Laura  
ittle dog Pritzi again.  
-- Claudia Fields, runner-up  
  
have been an organically based disturbance of the br  
a metabolic deficiency -- but after a thorough neuro  
mined that Byron was simply a jerk.  
-- Jeff Jahnke, runner-up  
  
n the 7th Annual Bulwer-Lytton Bad Writing Contest.  
er the author of the immortal lines: "It was a dark  
The object of the contest is to write the opening se  
sible novel.  
environment $ curl localhost:8081  
lles they make half the toilet soap we consume in Am  
illaise only have a vague theoretical idea of its us  
ined from books of travel.  
-- Mark Twain
```



Examine the mounted volume

```
# Attach shell to web-server container inside fortune Pod  
$ k exec fortune -it -n week8 -c web-server -- sh
```

```
/ # cat /usr/share/nginx/html/index.html
```

Try the Moo Shu Pork. It is especially good today.

```
# Wait 10 seconds
```

```
/ # cat /usr/share/nginx/html/index.html
```

You will be run over by a beer truck.

```
# Try upating the index.html file
```

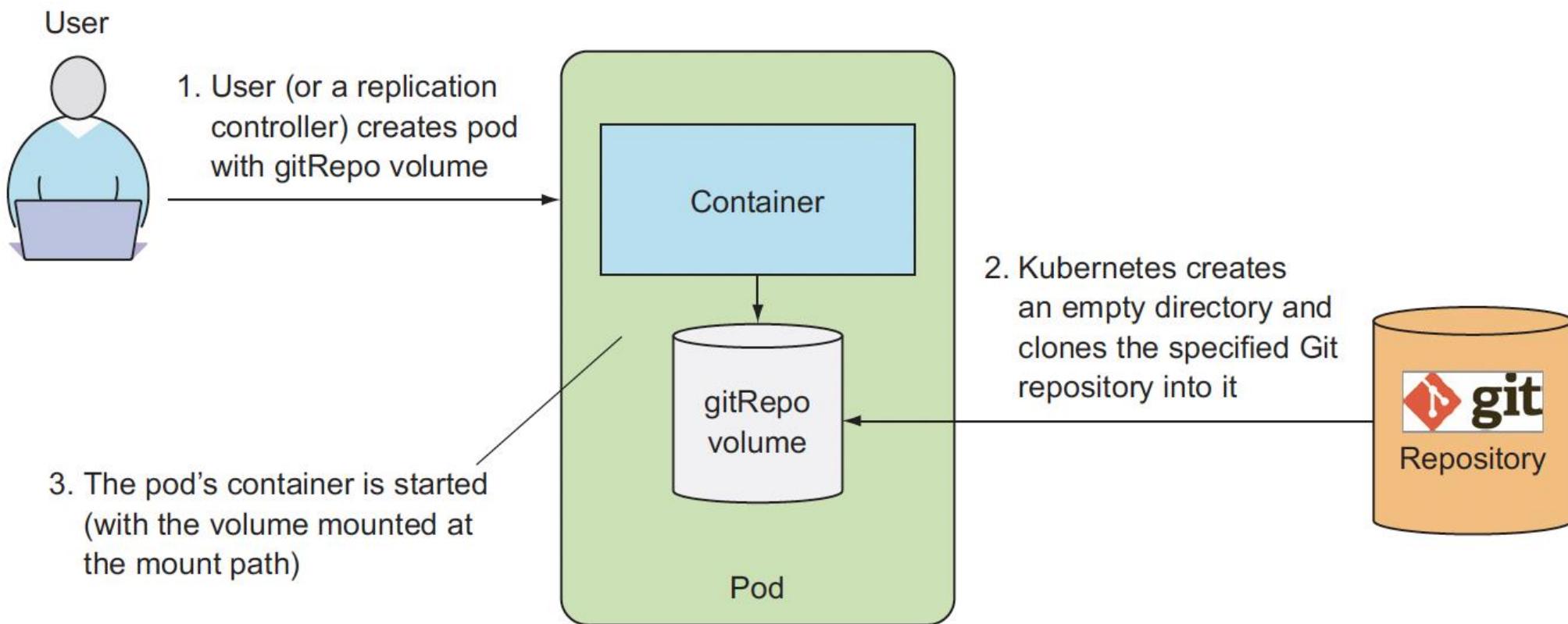
```
/ # vi /usr/share/nginx/html/index.html
```

Why does the editing attempt fail?



END OF WORKSHOP 1

Using GitRepo Volume



Pod Manifest with GitRepo

```
apiVersion: v1
kind: Pod
metadata:
  name: gitrepo-volume-pod
spec:
  containers:
    - image: nginx:alpine
      name: web-server
      volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
          readOnly: true
      ports:
        - containerPort: 80
          protocol: TCP
  volumes:
    - name: html
      gitRepo:
        repository: https://github.com/luksa/kubia-website-example.git
        revision: master
        directory: .
```

You're creating a gitRepo volume.

The volume will clone this Git repository.

You want the repo to be cloned into the root dir of the volume.

The master branch will be checked out.

Managing configuration

- Some applications need to be configured
- There are many ways for our code to pick up configuration:
 - command-line arguments
 - environment variables
 - configuration files
 - configuration servers (getting configuration from a database, an API...)
 - ... and more (because programmers can be very creative!)
- How can we do these things with containers and Kubernetes?

Persistent Volumes: Amazon EBS

- Using a persistent volume requires:
 - creating the volume out-of-band (outside of the Kubernetes API)
 - referencing the volume in the pod description, with all its parameters

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-my-ebs-volume
spec:
  containers:
    - image: ...
      name: container-using-my-ebs-volume
  volumeMounts:
    - mountPath: /my-ebs
      name: my-ebs-volume
  volumes:
    - name: my-ebs-volume
      awsElasticBlockStore:
        volumeID: vol-049df61146c4d7901
        fsType: ext4
```

Using NFS Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-my-nfs-volume
spec:
  containers:
    - image: ...
      name: container-using-my-nfs-volume
    volumeMounts:
      - mountPath: /my-nfs
        name: my-nfs-volume
  volumes:
    - name: my-nfs-volume
      nfs:
        server: 192.168.0.55
        path: "/exports/assets"
```



WORKSHOP 2 – INTORDUCING INITCONTAINER AND GITREPO VOLUME

Sharing a volume between two containers

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-git
spec:
  volumes:
    - name: www
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html/
    - name: git
      image: alpine
      command: [ "sh", "-c", "apk add git && git clone https://github.com/octo-deep/deep-learning-tutorial.git /www/" ]
      volumeMounts:
        - name: www
          mountPath: /www/
  restartPolicy: OnFailure
```

- \$ kubectl create -f nginx_with_git.yaml -n week8
- k port-forward pod/nginx-with-git 8080:80 -n week8
- \$ curl localhost:8080

The devil is in the details

- The default `restartPolicy` is `Always`
- This would cause our git container to run again ... and again ... and again (with an exponential back-off delay, as explained [in the documentation](#))
- That's why we specified `restartPolicy: OnFailure`

Using Init Containers

- The webserver should be started after the repository is cloned
- Starting simultaneously will cause a short downtime
- We can define containers that should execute *before* the main ones
- They will be executed in order
- (instead of in parallel)
- They must all succeed before the main containers are started

Defining Init Containers

```
Version: v1
kind: Pod
metadata:
  name: nginx-with-init
spec:
  volumes:
    - name: www
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html/
  initContainers:
    - name: git
      image: alpine
      command: [ "sh", "-c", "apk add git && git clone https://github.com/octo"
      volumeMounts:
        - name: www
          mountPath: /www/
```

```
$ kubectl create -f nginx_with_init.yaml -n week8
k port-forward pod/nginx-with-init 8080:80 -n week8
$ curl localhost:8080
```

Cleanup!

This command is your best friend from now on.

```
$eksctl delete cluster --name clo835 --  
region us-east-1
```



Workshop 2 – the End

WEEK 8

MANAGED K8S – AMAZON EKS



CLUSTER COST ESTIMATE – CONTROL PLANE ONLY!

Configure Amazon EKS [Info](#)

Description

Region [Info](#)

It is a physical location around the world where AWS clusters data centers. Changing the region for this service may require you to re-enter some or all of its configuration.

US East (N. Virginia) ▾

▼ EKS Cluster Pricing

Number of EKS Clusters

▼ Show calculations

1 Clusters x 0.10 USD per hour x 730 hours per month = 73.00 USD

EKS Total Cost (monthly): 73.00 USD

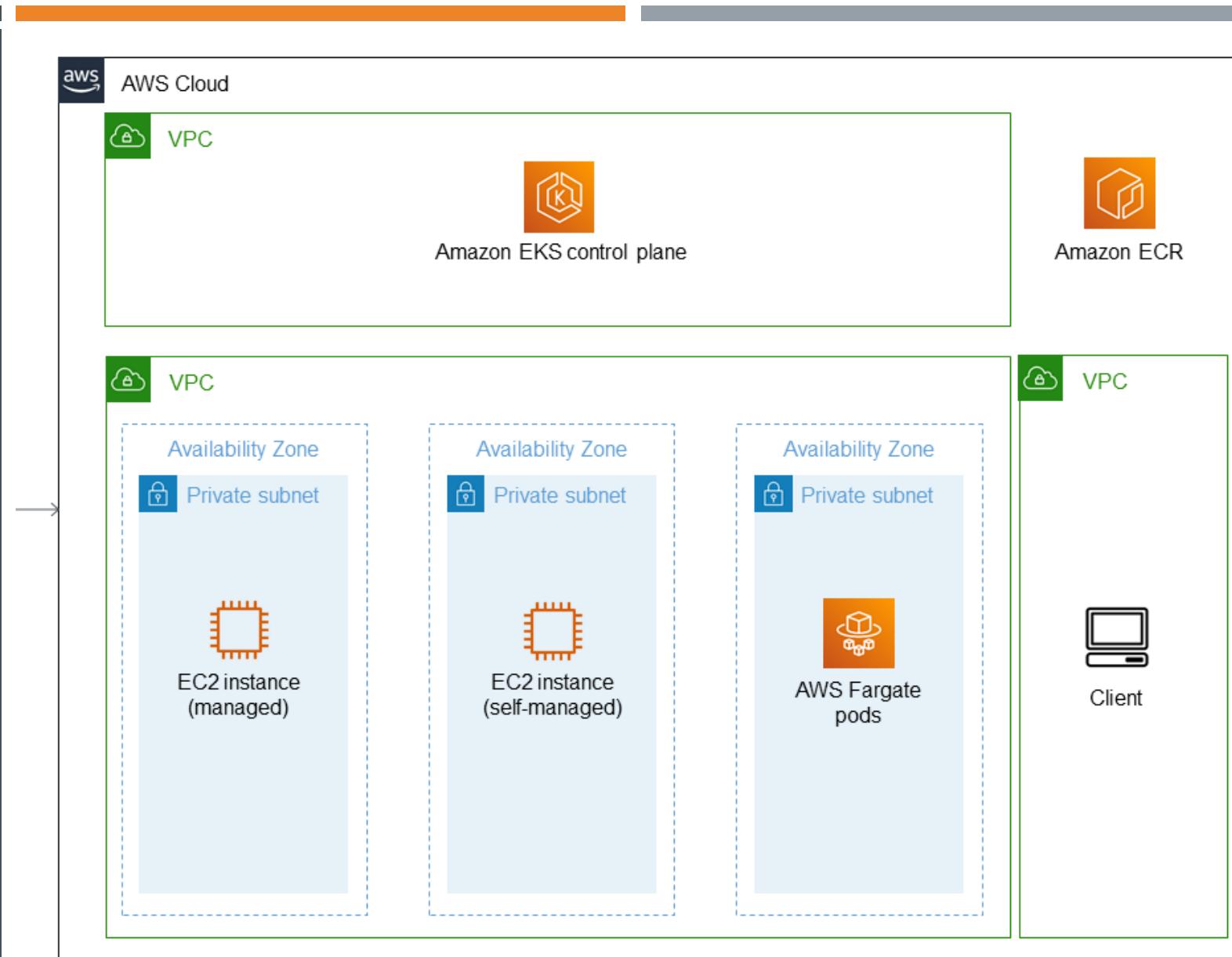
Amazon EKS estimate

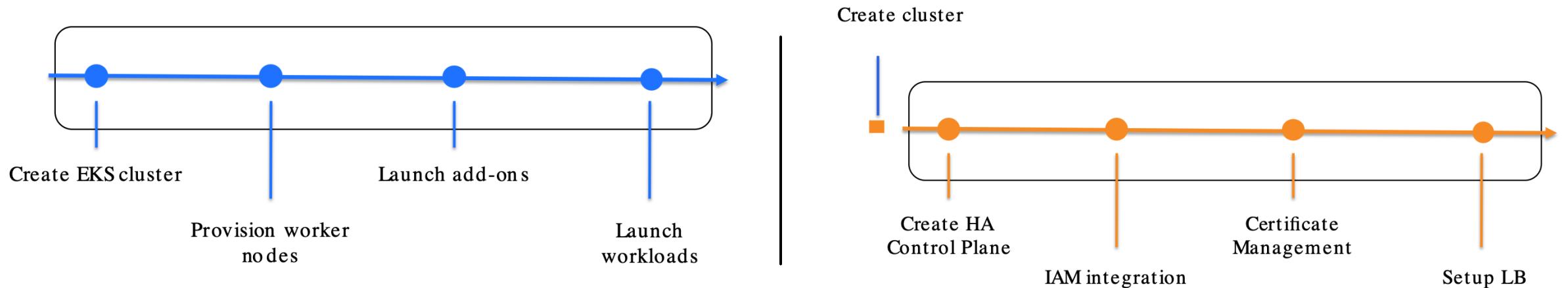
Total monthly cost: **73.00 USD**

[Cancel](#) [Add to my estimate](#)

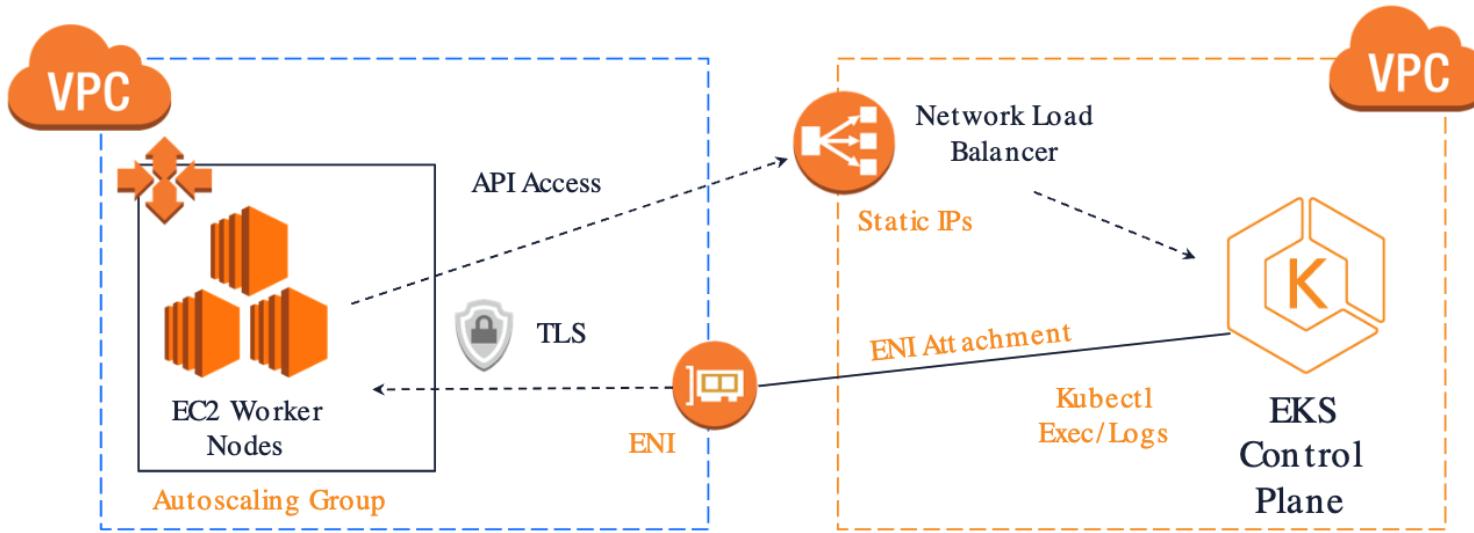
AMAZON EKS - HIGH LEVEL ARCHITECTURE

- Master nodes are running in AWS VPC, not in our VPC
- Amazon EKS API server can be either private or publicly accessible
- Worker nodes can be self-managed or AWS managed





AMAZON EKS CLUSTER CREATION



EKS ARCHITECTURE FOR CONTROL PLANE AND WORKER NODE COMMUNICATION

WORKSHOP 1 – CREATING AMAZON EKS CLUSTER WITH EKSCTL

GOALS

- Installing pre-requisites and configuring Cloud9 environment
- Configuring local environment
- Deployment K8s dashboard
- https://www.eksworkshop.com/010_introduction/



INCREASE DISK SPACE OF CLOUD9 ENVIRONMENT

https://www.eksworkshop.com/020_prerequisites/workspace/

The commands are available in Week8-Document.

Please note that we are not following all the workshop instructions.

DISABLE TEMPORARY CREDENTIALS

```
# Make sure you are using AWS CLI version 2. Use these instructions to update the AWS CLI version in your Cloud9 environment.
```

```
$ /usr/local/bin/aws --version
```

```
voclabs:~/environment $ /usr/local/bin/aws --version
aws-cli/2.11.0 Python/3.11.2 Linux/4.14.305-227.531.amzn2.x86_64 exe/x86_64.amzn.2 prompt/off
```

```
# Configure your permanent credentials and disable Cloud9 temporary credentials
```

```
$ /usr/local/bin/aws cloud9 update-environment --environment-id $C9_PID --managed-credentials-action DISABLE
```

```
voclabs:~/environment $ /usr/local/bin/aws cloud9 update-environment --environment-id $C9_PID --managed-credentials-action DISABLE
```

```
$ rm -vf ${HOME}/.aws/credentials
```

```
# Copy and paste the following credentials from AWS Academy into ~/.aws/credentials in Cloud9
```

AWS CLI:
Copy and paste the following into `~/.aws/credentials`

```
[default]
aws_access_key_id=ASIAODVIDYUC4TXM7A
aws_secret_access_key=s1FH4hGmI4r+mVgKRUAsGm98FSH8WFFpZLls9n
aws_session_token=FwGZXIVYXdzEkf//////////WeaDULPmrKilis6Xi6y
K9AaJFm7NAZuhYTkJBDfyamOI7Gx0uzsaafgiJgfTQecpGr/zRXAA4xm2LfptIeH
BnFXNzgnDphmBM+tIOPhMRxoj683PXFJS7vcqiEnqnxNfNdEdCD2S1mxcti
tgIZOyaJcjaxd-31ZFN6A198mBbc1Vhh0SMOfE1G7b0DklbKiXHVRvpztJI
2IkFtG06xmltik7ksR/H+UN02eV+vbuCJIVe03vvXozKHIBLNmeu1JvGauuWMtai
iNgaSg8jIthkJGqzzP79v0P5nqIj78U1T/Yqlhqc3T7q7aIKHkdgNFym9IqI/4xT
XRL570
```

INSTALLING PRE-REQUISITES

- Install kubectl - <https://docs.aws.amazon.com/eks/latest/userguide/install-kubectl.html>
- Install eksctl - <https://docs.aws.amazon.com/eks/latest/userguide/eksctl.html>
- Configure your permanent credentials and disable Cloud9 temporary credentials
- Update eks_config.yaml with your account id
- Create the cluster - this step will take a few minutes
- Update your Kube config
- Optional
 - Increase disk space of Cloud9
 - Install jq
 - Update AWS CLI
 - Install bash completion
 - Add kubectl alias
 - Set loadBalancer Controller version
 - Enable eksctl bash completion

EXPLORE THE CREATED RESOURCES WITH CLOUDFORMATION

CloudFormation > Stacks > eksctl-clo835-cluster

eksctl-clo835-cluster

Delete Update Stack actions ▾ Create stack ▾

Stack Info Events **Resources** Outputs Parameters Template Change sets

Resources (32)

Logical ID Physical ID Type Status Status reason Module

Logical ID	Physical ID	Type	Status	Status reason	Module
ClusterSharedNodeS ecurityGroup	sg- 0bb449f2ad586a08d	AWS::EC2::Secu rityGroup	CREATE_CO MPLETE	-	-
ControlPlane	clo835	AWS::EKS::Clust er	CREATE_IN_ PROGRESS	Resource creation Initiated	-
ControlPlaneSecuri tyGroup	sg- 0760e45257efdf8e	AWS::EC2::Secu rityGroup	CREATE_CO MPLETE	-	-
IngressInterNodeGro upSG	IngressInterNodeGro upSG	AWS::EC2::Secu rityGroupIngres s	CREATE_CO MPLETE	-	-
InternetGateway	igw- 01843414123f5c33a	AWS::EC2::Inter netGateway	CREATE_CO MPLETE	-	-
NATGateway	nat- 0a63d0c0c5923395c	AWS::EC2::NatG ateway	CREATE_CO MPLETE	-	-

EXPLORE THE CREATED CLUSTER WITH AMAZON EKS

The screenshot shows the AWS EKS Cluster details page for a cluster named 'clo835'. The top navigation bar includes 'EKS > Clusters > clo835'. The main header displays the cluster name 'clo835' and includes a refresh icon and a 'Delete cluster' button. Below the header, there's a 'Cluster info' section with tabs for 'Info' (selected), 'Overview', 'Resources', 'Compute', 'Networking' (highlighted in orange), 'Add-ons', 'Authentication', 'Logging', 'Update history', and 'Tags'. The 'Networking' section contains the following details:

VPC Info vpc-026fded885826a0c5	Subnets subnet-0e1fc50dac086b8c subnet-0e5617083bba4e00f subnet-0097305e94ab9ea84 subnet-0b80c5e8084333aaa subnet-04fc99240044a5049 subnet-067b85ea6d254cd09	Cluster security group Info Additional security groups sg-0760e45257efdf8e	API server endpoint access Info Public Public access source allowlist 0.0.0.0/0 (open to all traffic)
---	--	--	--

CLUSTER IS READY – WHAT RUNS IN THE CLUSTER?

- Examine the cluster in AWS Console

The screenshot shows the AWS EKS Cluster Overview page for cluster `clo835`. The top navigation bar includes links for EKS, Clusters, and the specific cluster name `clo835`. On the right side of the top bar are icons for refresh, delete, and a "Delete cluster" button. Below the navigation, there is a prominent message: "This cluster is running the **oldest** Kubernetes version currently supported by Amazon EKS. Ensure that your cluster is updated before the version end of support date. [Learn more](#)". To the right of this message is a "Update now" button.

The main content area is divided into sections: "Cluster info" (which is expanded), "Overview" (selected), "Resources", "Compute", "Networking", "Add-ons", "Authentication", "Logging", "Update history", and "Tags".

Cluster info:

Kubernetes version	Status	Provider
Info 1.19	Active	EKS

Details:

API server endpoint	https://E440836FD4DCA8B79AF41CEAEAB6A61E.gr7.us-east-1.eks.amazonaws.com	OpenID Connect provider URL	https://oidc.eks.us-east-1.amazonaws.com/id/E440836FD4DCA8B79AF41CEAEAB6A61E	Created	15 minutes ago
Certificate authority	LS0tLS1CRUdjTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUM1ekNDQWMrZ0F3SUJBZ0lCQURBTkJna3Fo a2lHOXcwQkFRc0ZBREFWTvJnd0VRWURWUfF	Cluster IAM role ARN	arn:aws:iam::047923129740:role/eksctl-clo835-cluster-ServiceRole-JBNQ3314TVDX	Cluster ARN	arn:aws:eks:us-east-1:047923129740:cluster/clo835
				Platform version	Info eks.10

Secrets encryption: [Info](#) Enable

Secrets encryption	Disabled	KMS key ID	-
--------------------	----------	------------	---

USE KUBECTL TO WORK WITH AMAZON EKS

```
# Can we use kubectl command?  
k get nodes  
# Update ~/.kube/config  
aws eks update-kubeconfig --name clo835 --region us-east-1
```

DEPLOY K8S DASHBOARD

```
export
```

```
DASHBOARD_VERSION="v2.0.0"
```

```
kubectl apply -f
```

```
https://raw.githubusercontent.com/kubernetes/dashboard//${DASHBOARD_VERSION}/aio/deploy/recommended.yaml
```

```
k get all -n kubernetes-dashboard
```

The screenshot shows the Kubernetes Dashboard interface. The top navigation bar includes tabs for Overview, Resources (which is currently selected), Compute, Networking, Add-ons, Authentication, Logging, Update history, and Tags. On the left, a sidebar titled 'Resource types' lists various Kubernetes objects: Workloads (PodTemplates, Pods, ReplicaSets, Deployments), StatefulSets, DaemonSets, Jobs, CronJobs, PriorityClasses, HorizontalPodAutoscalers, and Cluster. The 'Deployments' section under 'Workloads' is expanded, showing three entries: 'coredns' (managed by 'kube-system'), 'dashboard-metrics-scraper' (managed by 'kubernetes-dashboard'), and 'kubernetes-dashboard' (also managed by 'kubernetes-dashboard'). Each deployment entry includes columns for Name, Namespace, Type, Age, Pod count, and Status. All three deployments show a status of '2 Ready | 0 Failed | 2 Desired'. A 'View details' button is located in the top right corner of the main content area.

Name	Namespace	Type	Age	Pod count	Status
coredns	kube-system	deployments	Created 2 hours ago	2	2 Ready 0 Failed 2 Desired
dashboard-metrics-scraper	kubernetes-dashboard	deployments	Created a few seconds ago	1	1 Ready 0 Failed 1 Desired
kubernetes-dashboard	kubernetes-dashboard	deployments	Created a few seconds ago	1	1 Ready 0 Failed 1 Desired

VIEW THE DASHBOARD FROM CLOUD9

Since this is deployed to our private cluster, we need to access it via a proxy. kube-proxy is available to proxy our requests to the dashboard service. In your workspace, run the following command:

```
kubectl proxy --port=8080 --address=0.0.0.0 --disable-filter=true &
```

This will start the proxy, listen on port 8080, listen on all interfaces, and will disable the filtering of non-localhost requests.

This command will continue to run in the background of the current terminal's session.

EXPLORING KUBERNETES DASHBOARD

Now we can access the Kubernetes Dashboard

In your Cloud9 environment, click **Tools / Preview / Preview Running Application**

Scroll to **the end of the URL** and append:

```
/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-  
dashboard:/proxy/
```

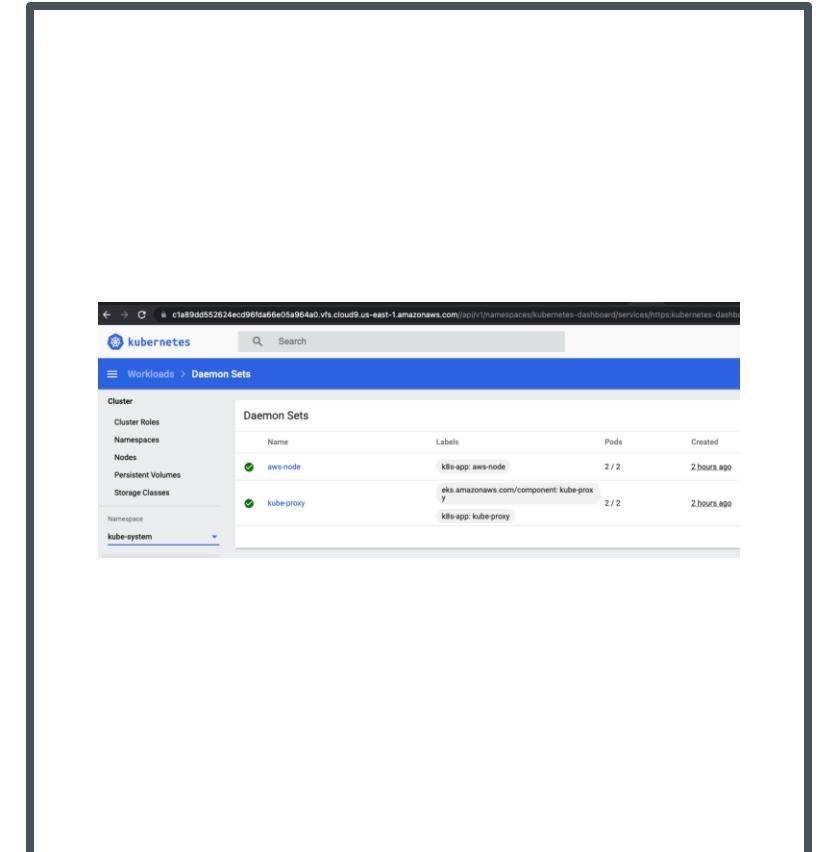
The Cloud9 Preview browser doesn't appear to support the token authentication, so once you have the login screen in the cloud9 preview browser tab, press the **Pop Out** button to open the login screen in a regular browser tab, like below:

Open a New Terminal Tab and enter

```
aws eks get-token --cluster-name clo835 | jq -r '.status.token'
```

Copy the output of this command and then click the radio button next to *Token* then in the text field below paste the output from the last command.

Then press *Sign In*.





END OF WORKSHOP 1

Volume Lifecycle

- The lifecycle of a volume is linked to the pod's lifecycle
- This means that a volume is created when the pod is created
- This is mostly relevant for emptyDir volumes
- (other volumes, like remote storage, are not "created" but rather "attached")
- A volume survives across container restarts
- A volume is destroyed (or, for remote storage, detached) when the pod is destroyed



WORKSHOP 2 – DEPLOYMENT OF OUR FIRST APPLICATION TO AMAZON EKS

GOALS

- Deployment of containerized application that uses emptyDir Volume to share data between containers
- Serving the application to external users with LoadBalancer service

DESCRIPTION

- Application is running as two containers in the same pod
- One process is generating random fortune every 10 seconds and writes it to a shared volumes
- The same volume is mounted into nginx container as /usr/share/nginx/html. Nginx serves the fortunes from this folder

```
Executable File | 11 lines (9 sloc) | 181 Bytes

1  #!/bin/bash
2  trap "exit" SIGINT
3  mkdir /var/htdocs
4
5  while :
6  do
7      echo $(date) Writing fortune to /var/htdocs/index.html
8      /usr/games/fortune > /var/htdocs/index.html
9      sleep 10
10 done
11
```



Description

Application is running as two containers in the same pod

One process is generating random fortune every 10 seconds and writes it to a shared volumes

The same volume is mounted into nginx container as /usr/share/nginx/html. Nginx serves the fortunes from this folder

Pod Manifest

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune
spec:
  containers:
    - image: luksa/fortune
      name: html-generator
      volumeMounts:
        - name: html
          mountPath: /var/htdocs
    - image: nginx:alpine
      name: web-server
      volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
          readOnly: true
  ports:
    - containerPort: 80
      protocol: TCP
  volumes:
    - name: html
      emptyDir: {}
```

The first container is called **html-generator** and runs the **luksa/fortune** image.

The volume called **html** is mounted at **/var/htdocs** in the container.

The second container is called **web-server** and runs the **nginx:alpine** image.

The same volume as above is mounted at **/usr/share/nginx/html** as **read-only**.

A single **emptyDir** volume called **html** that's mounted in the two containers above

Deploy and Access the application

```
# Create a namespace  
$ k create ns week8  
$ k apply -f fortune_pod.yaml -n week8  
$ k get all -n week8  
  
# Deploy the pod  
# Access the app via port forwarding  
$ kubectl port-forward fortune 8080:80 -n week8  
  
$ curl http://localhost:8080
```

```
environment $ curl localhost:8081  
xpensive sports car, fine-tuned and well-built, Port  
and gorgeous, her red jumpsuit moulding her body, wh  
vers in July, her hair as dark as new tires, her eye  
ocaps, and her lips as dewy as the beads of fresh ra  
woman driven -- fueled by a single accelerant -- an  
n who wouldn't shift from his views, a man to steer  
d: a man like Alf Romeo.  
-- Rachel Sheeley, winner  
  
ball blocking the drain of the shower reminded Laura  
ittle dog Pritzi again.  
-- Claudia Fields, runner-up  
  
have been an organically based disturbance of the br  
a metabolic deficiency -- but after a thorough neuro  
mined that Byron was simply a jerk.  
-- Jeff Jahnke, runner-up  
  
n the 7th Annual Bulwer-Lytton Bad Writing Contest.  
er the author of the immortal lines: "It was a dark  
The object of the contest is to write the opening se  
sible novel.  
environment $ curl localhost:8081  
lles they make half the toilet soap we consume in Am  
illaise only have a vague theoretical idea of its us  
ined from books of travel.  
-- Mark Twain
```



Examine the mounted volume

```
# Attach shell to web-server container inside fortune Pod  
$ k exec fortune -it -n week8 -c web-server -- sh
```

```
/ # cat /usr/share/nginx/html/index.html
```

Try the Moo Shu Pork. It is especially good today.

```
# Wait 10 seconds
```

```
/ # cat /usr/share/nginx/html/index.html
```

You will be run over by a beer truck.

```
# Try updating the index.html file
```

```
/ # vi /usr/share/nginx/html/index.html
```

Why does the editing attempt fail?

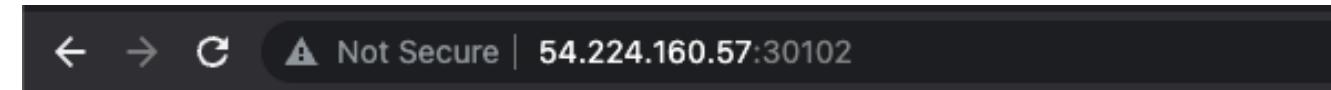
Expose Fortunes via NodePort

```
$ k expose pod fortune -n week8 --  
type NodePort -n week8
```

K8s assigned a NodePort randomly.
There only one pod and 2 nodes.

What IP should we use to access
the application?

The IPs of both nodes respond.
Why?



Expose Fortunes via LoadBalancer

```
$ k delete service fortune -n week8
```

```
$ k expose pod fortune -n week8 --type LoadBalancer -n week8
```

Load balancer: af646377940b0455f9edd8e1b505a9b8

Description Instances Health check Listeners Monitoring Tags Migration

Basic Configuration

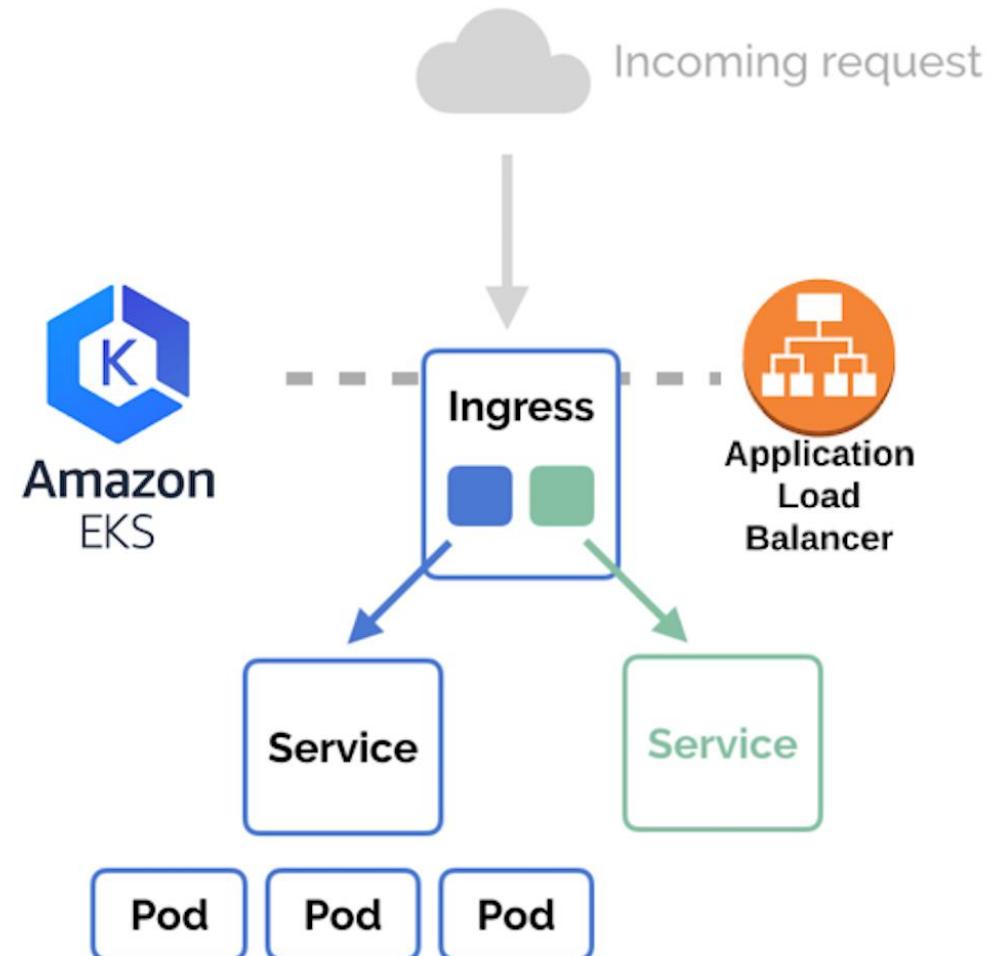
Name	af646377940b0455f9edd8e1b505a9b8	Creation time	July 2, 2022 at 11:00:33 PM UTC-4
* DNS name	af646377940b0455f9edd8e1b505a9b8-506539463.us-east-1.elb.amazonaws.com (A Record)	Hosted zone	Z355XDOTRQ7X7K
Type	Classic (Migrate Now)	Status	2 of 2 instances in service
Scheme	Internet-facing	VPC	vpc-026fded885826a0c5
Availability Zones	subnet-0097305e94ab9ea84 - us-east-1c, subnet-0e14fc50dac086b8c - us-east-1a, subnet-0e5617083bba4e00f - us-east-1b		

```
irinag:~/environment $ k get svc fortune -n week8
Name:           fortune
Namespace:      week8
Labels:         app=fortune
                week=week8
Annotations:   <none>
Selector:      app=fortune,week=week8
Type:          LoadBalancer
IP:            10.100.27.63
LoadBalancer Ingress: af646377940b0455f9edd8e1b505a9b8-506539463.us-east-1.elb.amazonaws.com
Port:          <unset>  80/TCP
TargetPort:    80/TCP
NodePort:      <unset>  32633/TCP
Endpoints:    192.168.38.52:80
Session Affinity: None
External Traffic Policy: Cluster
Events:
  Type  Reason     Age   From           Message
  ----  ----     --   --   --
  Normal  EnsuringLoadBalancer  2m20s  service-controller  Ensuring load balancer
  Normal  EnsuredLoadBalancer  2m16s  service-controller  Ensured load balancer
```

Expose Fortunes via LoadBalancer

← → ⌂ Not Secure | af646377940b0455f9edd8e1b505a9b8

What happened last night can happen again.





END OF WORKSHOP 2

Additional Reading

<https://kubernetes.io/docs/concepts/storage/volumes/>

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-volume-storage/>

<https://aws.amazon.com/premiumsupport/knowledge-center/eks-persistent-storage/>

<https://docs.giantswarm.io/getting-started/persistent-volumes/aws/>

<https://github.com/weaveworks/eksctl/tree/main/examples>

<https://eksctl.io/>

WEEK 9

PV, PVC, STORAGECLASS
AND STATEFULSETS



AGENDA

Kubernetes Volumes – hostPath

PersistentVolume

PersistentVolumeClaim

StorageClass

StatefulSet

VOLUMES: HOSTPATH

- First persistent storage – not deleted when the pod is deleted
- hostPath should not be used to store application's data
- Used by pods to get access to host data, e.g. kubeconfig or /var/log or OS devices through file system
- Used as a persistent volume in simulated clusters like "kind" and "minikube"
- Stored on a specific node – is it suitable for DB storage?
- What if we need to persist data in a cluster with multiple worker nodes?

hostPath Concerns

HostPaths can expose privileged system credentials

Pods with identical configuration (such as created from a PodTemplate) may behave differently on different nodes due to different files on the nodes

The files or directories created on the underlying hosts are only writable by root.

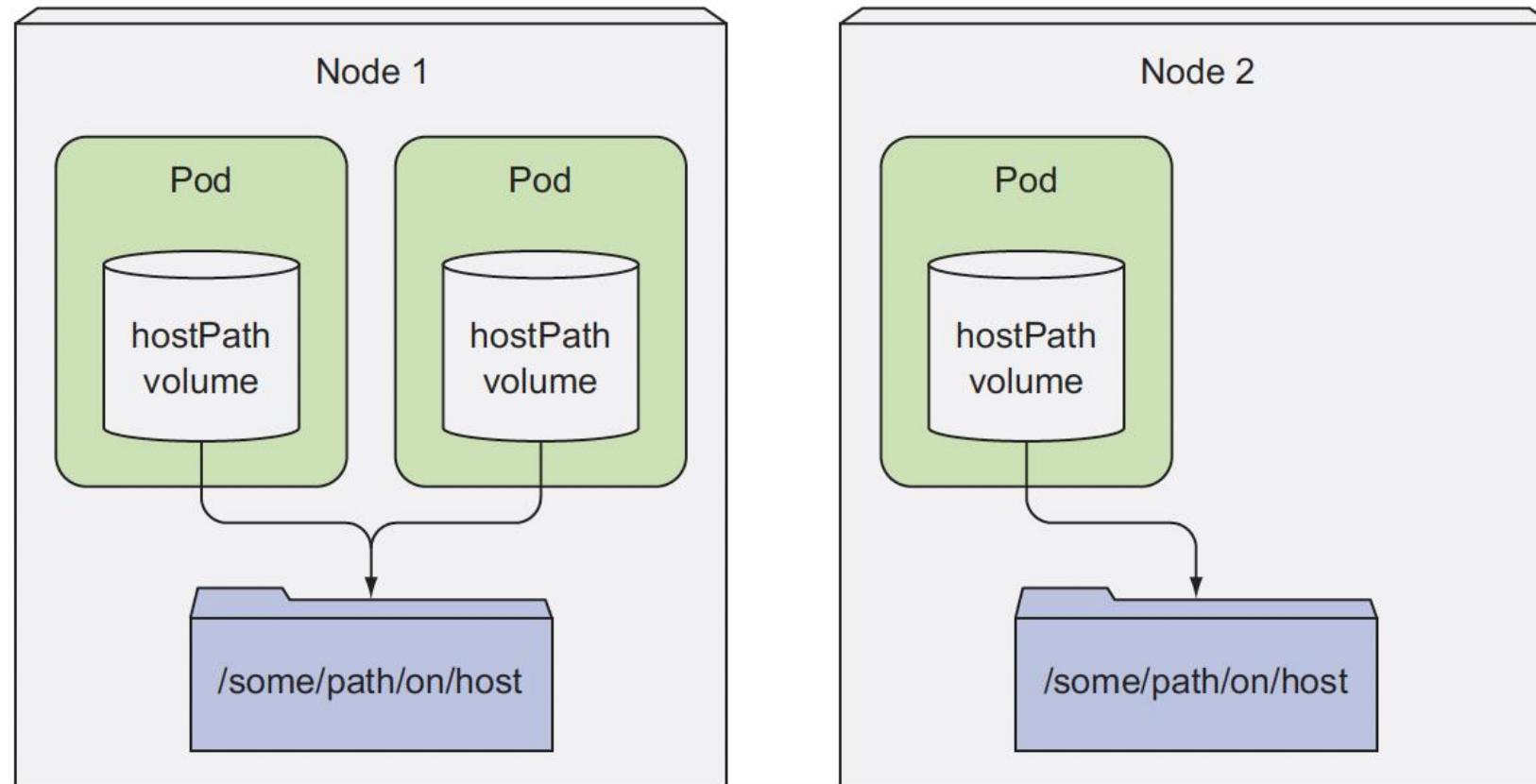
hostPath

Warning:

HostPath volumes present many security risks, and it is a best practice to avoid the use of HostPaths when possible. When a HostPath volume must be used, it should be scoped to only the required file or directory, and mounted as ReadOnly.

If restricting HostPath access to specific directories through AdmissionPolicy, `volumeMounts` MUST be required to use `readOnly` mounts for the policy to be effective.

Data Sharing between the node and the Pod's containers with hostPath volume



hostPath Configuration Example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
      # this field is optional
      type: Directory
```

Persistent Volumes: Amazon EBS

- Using a persistent volume requires:
 - creating the volume out-of-band (outside of the Kubernetes API)
 - referencing the volume in the pod description, with all its parameters

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-my-ebs-volume
spec:
  containers:
    - image: ...
      name: container-using-my-ebs-volume
  volumeMounts:
    - mountPath: /my-ebs
      name: my-ebs-volume
  volumes:
    - name: my-ebs-volume
      awsElasticBlockStore:
        volumeID: vol-049df61146c4d7901
        fsType: ext4
```

Using NFS Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-my-nfs-volume
spec:
  containers:
    - image: ...
      name: container-using-my-nfs-volume
    volumeMounts:
      - mountPath: /my-nfs
        name: my-nfs-volume
  volumes:
    - name: my-nfs-volume
      nfs:
        server: 192.168.0.55
        path: "/exports/assets"
```

Challenge of Volumes



Should be pre-deployed



Requires the app developer to
know the infrastructure details



Locks pod definition to a specific
cluster

Workshop 1 - Exploring hostPath

Deploy MongoDB to Amazon EKS

```
# Verify the cluster is running
```

```
$ k get nodes
```

```
# Create Amazon EKS cluster if there is no active cluster
```

```
$ eksctl create cluster clo835 -f eks_config.yaml
```

```
# Make sure the cluster is running
```

```
$ k get nodes
```

```
# Create the namespace
```

```
$ k create ns week9
```

```
# Deploy mongodb app backed by the hostpath
```

```
$ k apply -f mongodb_hostpath.yaml -n week9
```

Add Sample Data to MongoDB

```
# Start mongo shell in the mongo cotainer
$ k exec -it mongodb -n week9 -- mongosh # Please note that mongo client was replaced with mongosh
# Add sample json document to the DB
> use mystore
switched to db mystore
> db.foo.insert({name:'foo'})
WriteResult({ "nInserted" : 1 })
> db.foo.find()
{ "_id" : ObjectId("57a61eb9de0cf512374cc75"), "name" : "foo" }
# Exit the container and mongoDB shell
> exit
Bye
# Examine the node that the pod is scheduled to
$ k get pods -o wide -n week9
NAME    READY  STATUS   RESTARTS  AGE     IP          NODE           NOMINATED NODE  READINESS GATES
mongodb  1/1    Running  0         13m    192.168.53.113  ip-192-168-40-207.ec2.internal  <none>        <none>
```

Add Sample Data to MongoDB

```
# Delete the pod and create a new one
```

```
kubectl delete pod mongodb -n week9
```

```
# Recreate the mongoDB pod
```

```
$ k apply -f mongodb_hostpath.yaml -n week9
```

```
# Let's examine the data – it turns out to be hit and miss. Why?
```

```
$ k exec -it mongodb -n week9 -- mongosh
```

```
> use mystore
```

```
switched to db mystore
```

```
> db.foo.find() # data persisted. What would happen if the pod is scheduled to a different node?
```

```
{ "_id" : ObjectId("62c5b2586a2dbca9f32d3213"), "name" : "foo" }
```

```
# Examnine the node that the pod is scheduled to
```

```
$ k get pods -n week9 -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
mongodb	1/1	Running	0	5m42s	192.168.53.113	ip-192-168-40-207.ec2.internal	<none>	<none>

HostPath with Amazon EBS

```
# Provision Amazon EBS volume  
$ aws ec2 create-volume --volume-type gp2 --size 80 --availability-zone us-east-1a  
# Update the mongodb_aws.yaml with the value of the volume just created  
$ k apply -f mongodb_aws.yaml -n week9  
# Using it is hit and miss – what if the pod is not assigned to the node in the AZ of our Amazon EBS volume  
# There should be a better way!
```

Why to have hostPath at all – Examine kube-proxy

```
# System pods are using hostPath extensively
```

```
$ k get all -n kube-proxy
```

```
# Select any of the kube-proxy pods, the name of the pod will be kube-proxy-<random string>
```

```
$ k describe pod/kube-proxy-<random string> -n kube-system
```

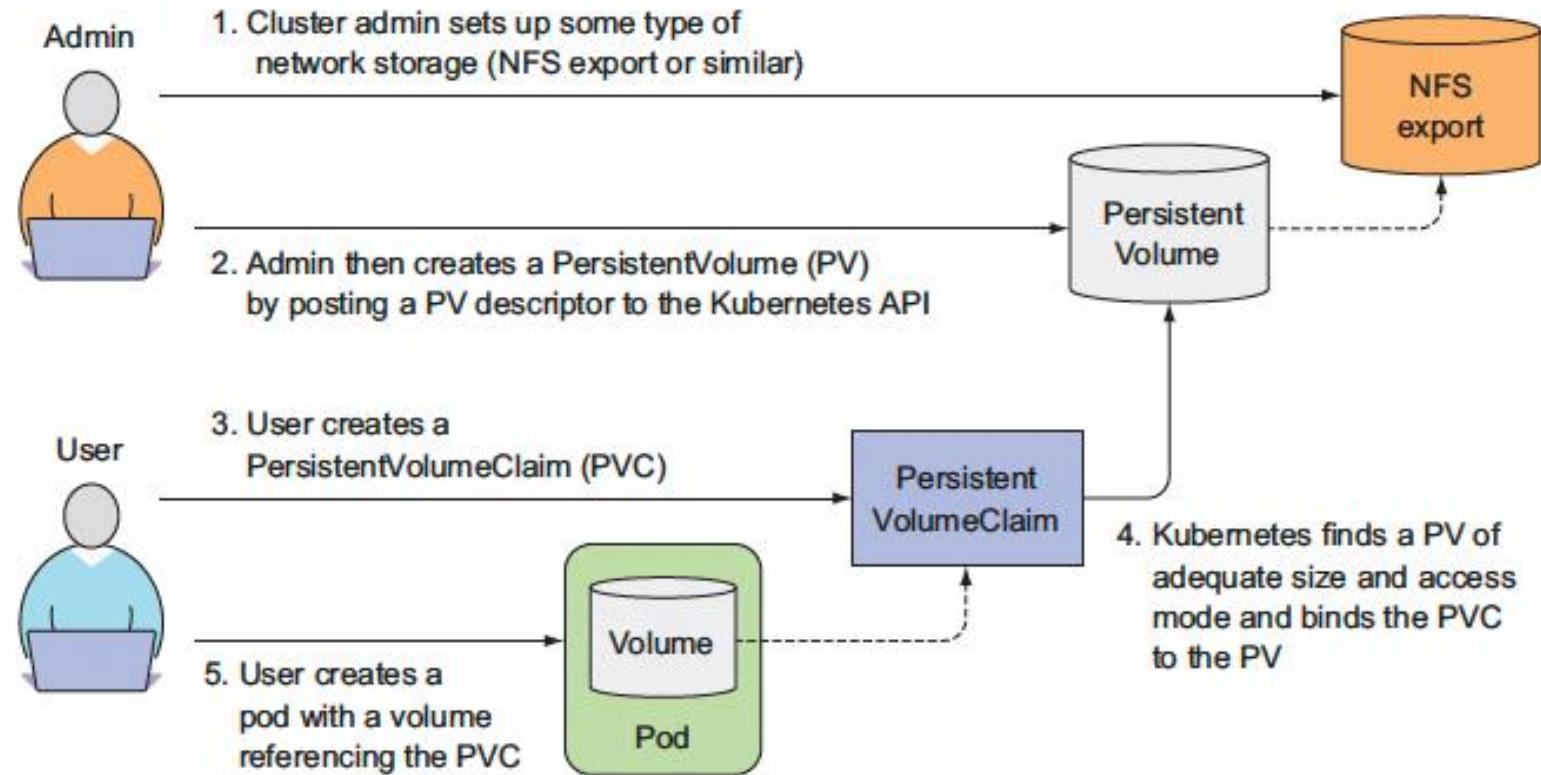
```
Volumes:
  varlog:
    Type:      HostPath (bare host directory volume)
    Path:      /var/log
    HostPathType: 
  xtables-lock:
    Type:      HostPath (bare host directory volume)
    Path:      /run/xtables.lock
    HostPathType: FileOrCreate
  lib-modules:
    Type:      HostPath (bare host directory volume)
    Path:      /lib/modules
    HostPathType:
```

Workshop 1 - The End



PV, PVC, and StorageClasses

Introducing PersistentVolumes and PersistentVolumeClaims



PersistentVolume Manifest

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
    pdName: mongodb
    fsType: ext4
```

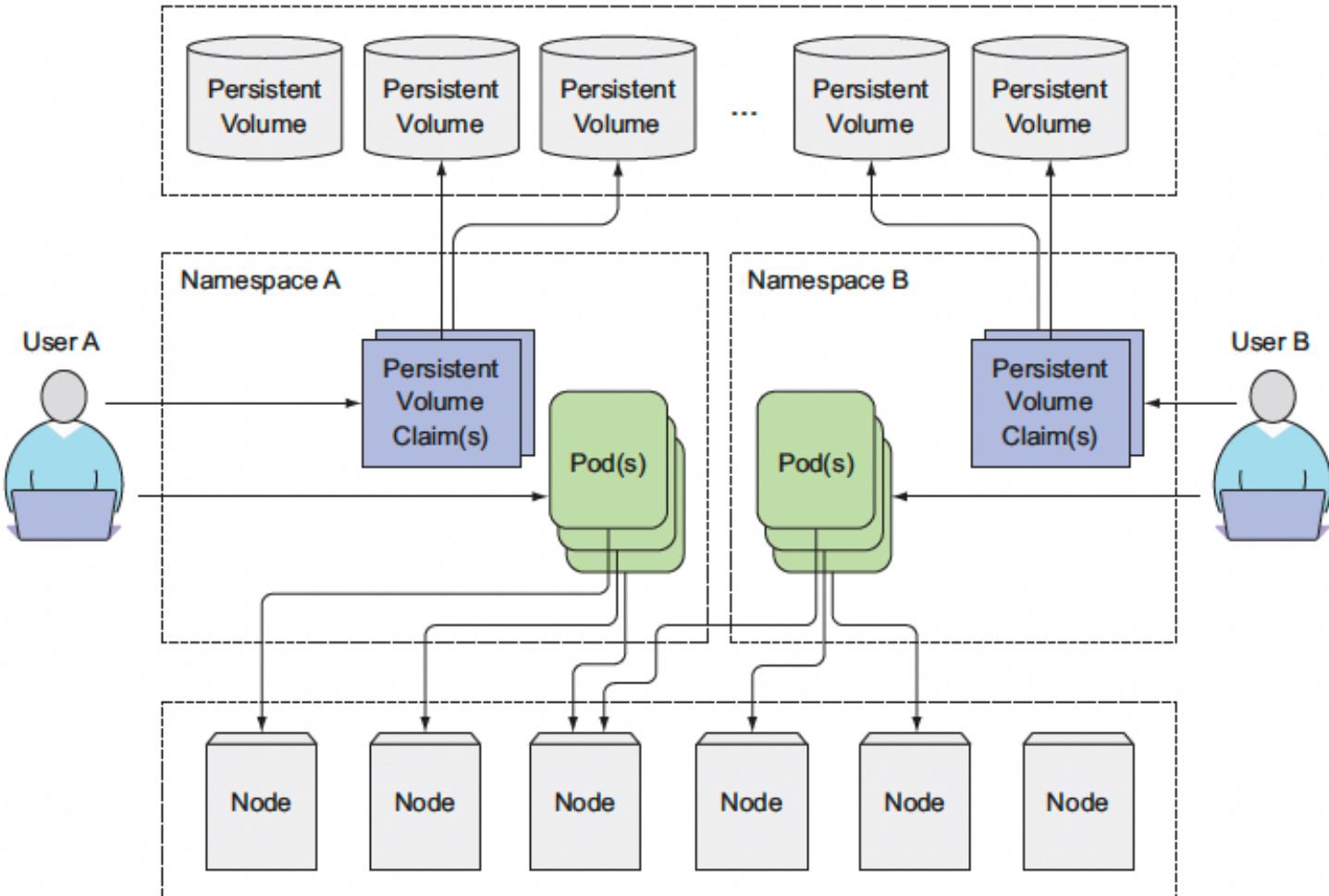
Defining the PersistentVolume's size

It can either be mounted by a single client for reading and writing or by multiple clients for reading only.

The PersistentVolume is backed by the GCE Persistent Disk you created earlier.

After the claim is released, the PersistentVolume should be retained (not erased or deleted).

Decoupling pods from the underlying storage technology



Using PersistentVolumes

When an application needs storage, it creates a `PersistentVolumeClaim` (either directly, or through a volume claim template in a Stateful Set)

The `PersistentVolumeClaim` is initially Pending

Kubernetes then looks for a suitable `PersistentVolume` (maybe one is immediately available; maybe we need to wait for provisioning)

Once a suitable `PersistentVolume` is found, the PVC becomes Bound

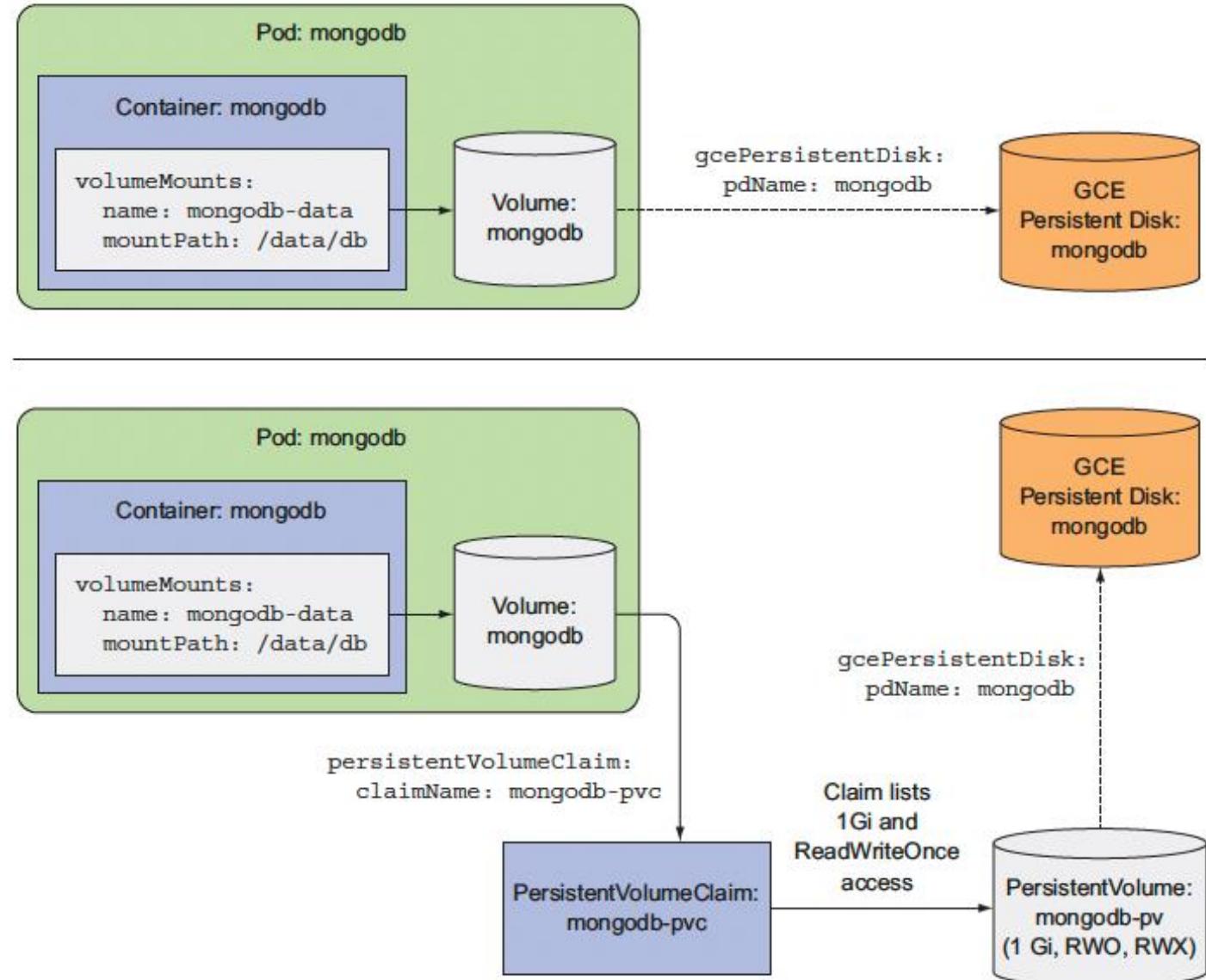
The PVC can then be used by a Pod (as long as the PVC is Pending, the Pod cannot run)

Using a PersistentVolumeClaim in a pod

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
    - image: mongo
      name: mongodb
      volumeMounts:
        - name: mongodb-data
          mountPath: /data/db
      ports:
        - containerPort: 27017
          protocol: TCP
      volumes:
        - name: mongodb-data
          persistentVolumeClaim:
            claimName: mongodb-pvc
```

Referencing the PersistentVolumeClaim by name in the pod volume

Understanding the benefits of using PersistentVolumes and claims



Provisioning Automation: Storage Class

- What if we have multiple storage systems available? (e.g. NFS and iSCSI; or AzureFile and AzureDisk; or Cinder and Ceph...)
- What if we have a storage system with multiple tiers? (e.g. SAN with RAID1 and RAID5; general purpose vs. io optimized EBS...)
- Kubernetes lets us define *storage classes* to represent these(see if you have any available at the moment with `kubectl get storageclasses`)

Using storage classes

- Optionally, each PV and each PVC can reference a StorageClass (field spec.storageClassName)
- When creating a PVC, specifying a StorageClass means “use that particular storage system to provision the volume!”
- Storage classes are necessary for dynamic provisioning (but we can also ignore them and perform manual provisioning)

Storage Class Manifest

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  zone: europe-west1-b
```

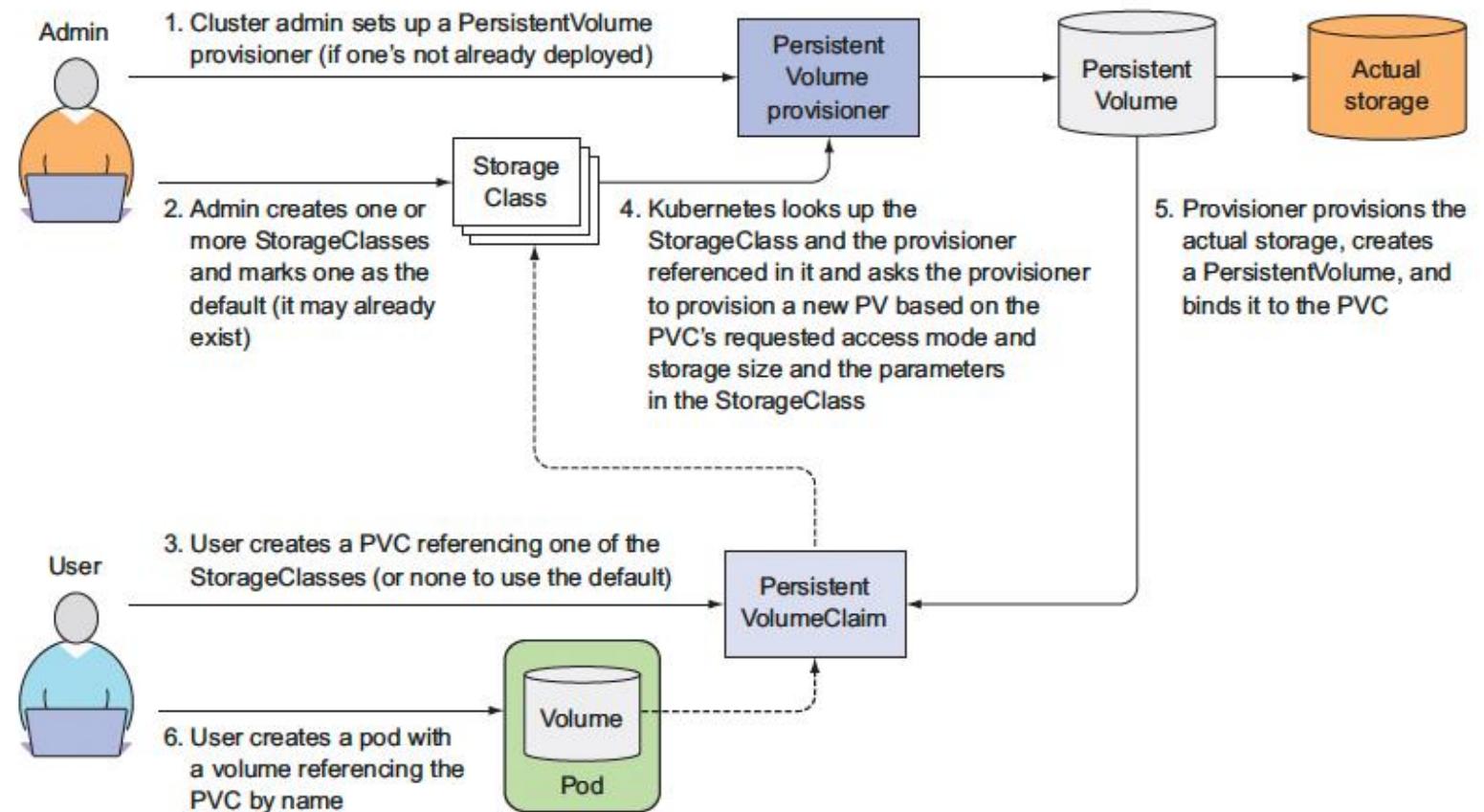
The volume plugin to use for provisioning the PersistentVolume

The parameters passed to the provisioner

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
spec:
  storageClassName: fast
  resources:
    requests:
      storage: 100Mi
  accessModes:
    - ReadWriteOnce
```

This PVC requests the custom storage class.

Dynamic PersistentVolume Provisioning





WORKSHOP 2 – PV, PVC, STORAGECLASSES

AMAZON EBS BACKED VOLUME – DATA PERSISTS INDEPENDENT OF THE HOSTING NODE

```
# Create redis deployment, service and PVC
$ k get sc
$ k create ns week9
$ k apply -f redis_service.yaml -n week9
$ k apply -f redis_pvc.yaml -n week9
# Review the created PVC and PV, describe the created PV and PVC
$ k get pvc -n week9
$ k apply -f redis_deployment.yaml -n week9
# Add PVC mount to Redis deployment manifest and redeploy Redis
$ k delete deployment redis -n week9
$ k apply -f redis_deployment.yaml -n week9
```

STORE DATA IN REDIS AND RECREATE THE POD

```
# Run redis cli client

$ kubectl run redis-cli --rm -ti --image=redis:3.2.5 --restart=Never -- /bin/sh

# redis-cli -h redis.week9.svc.cluster.local

> set foo bar

> get foo

"bar"

> quit

# Delete the pod

$ k delete pod-xxx redis -n week9

# Notice if the pod got rescheduled to another node

$ k get pods -n week9

# run redis cli and verify that data is still available when the pod gets
# rescheduled to another node

$ kubectl run redis-cli --rm -ti --image=redis:3.2.5 --restart=Never -- /bin/bash

# redis-cli -h redis.week9.svc.cluster.local

> get foo
```



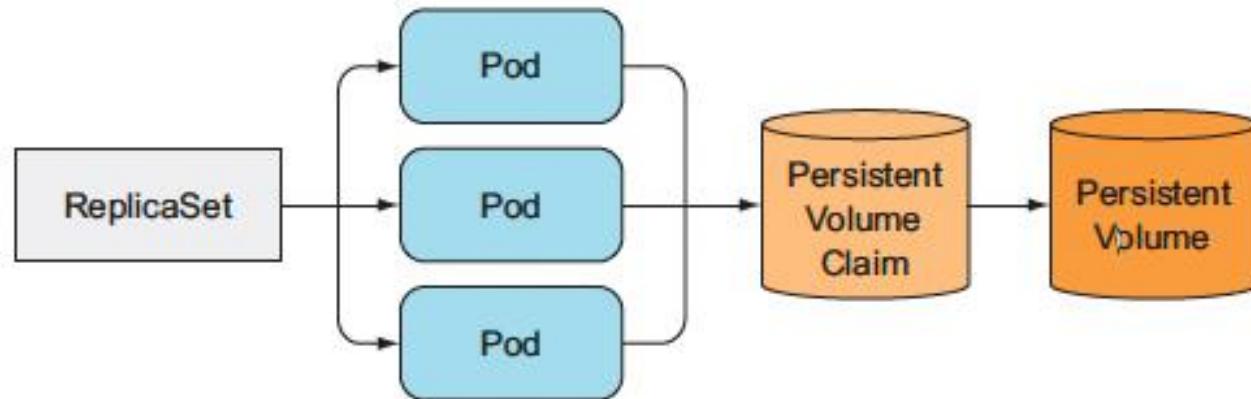
WORKSHOP 2 – THE END

DEPLOYING CLUSTERED APPLICATIONS



SCALING PODS THAT REFERENCE PERSISTENT VOLUME

- All the replicas in the replicaset are identical
- All pods in the replicaset will reference the same volume since volume are defined at a pod level
- What we are deploying a stateful application (DB) and each replica needs to have an **independent** storage



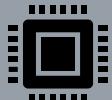
STABLE NETWORK IDENTITY FOR PODS



Clustered applications also require that each instance has a long-lived stable identity (hostname and IP)



Each member of the stateful application's cluster need to know network identities of all member of the cluster



This would not work with applications created with K8s deployment – each time the pods is re-created it's IP and hostname changes

STATEFULSETS



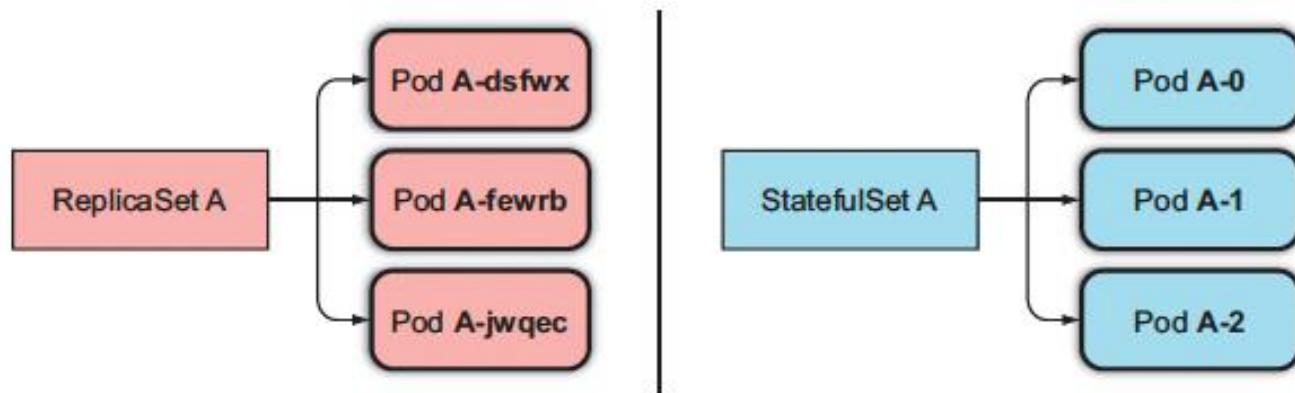
StatefulSet manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods, suitable for applications that require one or more of the following.

Applications are treated as non-fungible individuals, with each one having a stable name and state. (pets vs cattle analogy)

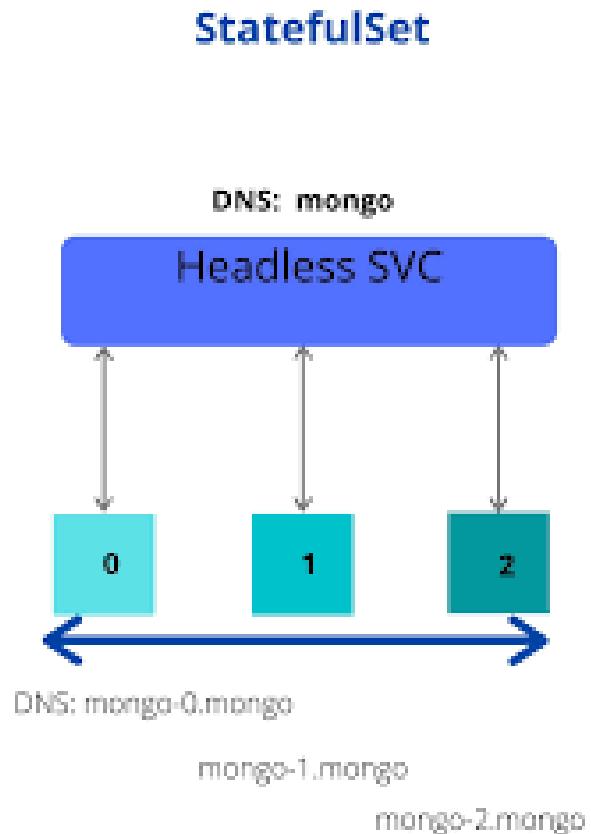
- Stable, unique network identifiers
- Stable, persistent storage
- Ordered, graceful deployment and scaling
- Ordered, automated rolling updates

STATEFULSETS VS REPLICASETS

- Pods in ReplicaSet are like cattle. They are indistinguishable and replaceable. If the pod dies a new pod with a new IP/hostname and data is created.
- Pods in Stateful sets are like pets. When a "pet" dies, it needs to be resurrected with the same name, IP and data. Pods in Statefulsets are not replicas of each other. They can have individual volumes (data).



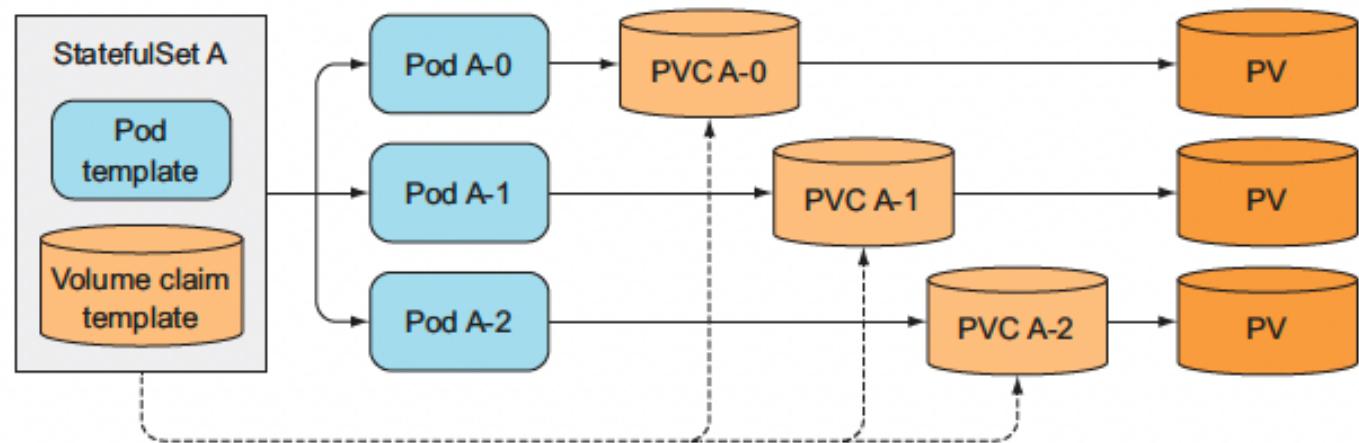
EXPOSING STATEFULSETS



- Headless service, ClusterIP: None
- Each pod has its own DNS entry, e.g. a-0.foo.default.svc.cluster.local

ADDING VOLUMES TO STATEFULSET'S PODS

- Storage needs to be persistent and decoupled from pods
- Each pod needs to reference a different PVC
- PVCs need to be created by Statefulset and not separately as we did for the stateless pods





WORKSHOP 3

STATEFULSETS

APPLICATION FUNCTIONALITY

- On "POST", store the data in a local file
- On GET, return the hostname and the content of the file
- <https://github.com/luksa/kubernetes-in-action/blob/master/Chapter10/kubia-pet-image/app.js>

```
...
const dataFile = "/var/data/kubia.txt";
...
var handler = function(request, response) {
  if (request.method == 'POST') {
    var file = fs.createWriteStream(dataFile);
    file.on('open', function (fd) {
      request.pipe(file);
      console.log("New data has been received and stored.");
      response.writeHead(200);
      response.end("Data stored on pod " + os.hostname() + "\n");
    });
  } else {
    var data = fileExists(dataFile)
      ? fs.readFileSync(dataFile, 'utf8')
      : "No data posted yet";
    response.writeHead(200);
    response.write("You've hit " + os.hostname() + "\n");
    response.end("Data stored on this pod: " + data + "\n");
  }
};
```

On POST requests, store the request's body into a data file.

On GET (and all other types of) requests, return your hostname and the contents of the data file.

WHAT ARE WE CREATING?

- StatefulSet
- Governing service
- PersistentVolumeClaims as part of StatefulSet manifest. Volume provisioning will be handled by AWS EBS provisioner if we use default StorageClass gp2.
- <https://github.com/luksa/kubernetes-in-action/blob/master/Chapter10/kubia-statefulset.yaml>

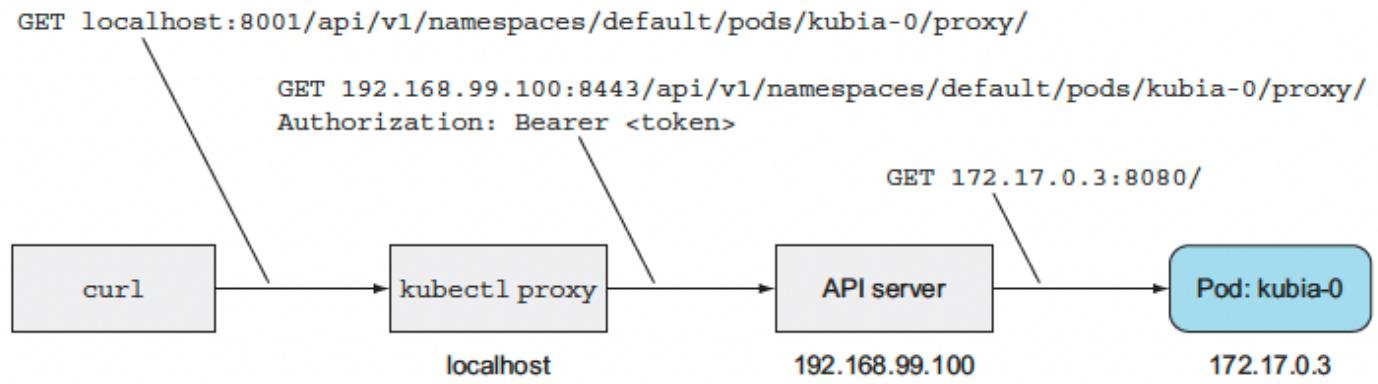
DEPLOYMENT - STATEFULSET

```
irinag:~/environment/week9/kubia-statefulset $ k create -f kubia-statefulset.yaml
statefulset.apps/kubia created
irinag:~/environment/week9/kubia-statefulset $ k get po
NAME      READY  STATUS    RESTARTS   AGE
kubia-0   0/1    Pending   0          5s
irinag:~/environment/week9/kubia-statefulset $ k get po
NAME      READY  STATUS    RESTARTS   AGE
kubia-0   0/1    ContainerCreating   0          13s
irinag:~/environment/week9/kubia-statefulset $ k get po
NAME      READY  STATUS    RESTARTS   AGE
kubia-0   0/1    ContainerCreating   0          20s
irinag:~/environment/week9/kubia-statefulset $ k get po
NAME      READY  STATUS    RESTARTS   AGE
kubia-0   1/1    Running   0          32s
kubia-1   0/1    Pending   0          6s
irinag:~/environment/week9/kubia-statefulset $
```

- # Pods are created one by one, unlike replicaset where all pods are created at once
- \$ k create -f kubia-statefulset
- \$ k get pvc
- \$ k get pv

COMMUNICATING WITH THE APPLICATION VIA PROXY

- \$ kubectl proxy
- \$ curl localhost:8001/api/v1/namespaces/default/pods/kubia-0/proxy/
- curl -X POST -d "Hey there! This greeting was submitted to kubia-0."
- curl -X POST -d "Hey there! This greeting was submitted to kubia-0." localhost:8001/api/v1/namespaces/default/pods/kubia-0/proxy/
- curl localhost:8001/api/v1/namespaces/default/pods/kubia-0/proxy/
- curl localhost:8001/api/v1/namespaces/default/pods/kubia-1/proxy/



DELETING A STATEFUL POD TO SEE IF THE RESCHEDULED POD IS REATTACHED TO THE SAME STORAGE

- \$ kubectl delete po kubia-0
- \$ kubectl get po # Execute a few times
- \$ curl localhost:8001/api/v1/namespaces/default/pods/kubia-0/proxy/
- # persistent volume gets re-attached to a "new" kibia-0 pod

```
irinag:~/environment $ k delete po kubia-0
pod "kubia-0" deleted
irinag:~/environment $ curl localhost:8001/api/v1/namespaces/default/pods/kubia-0/proxy/
You've hit kubia-0
Data stored on this pod: Hey there! This greeting was submitted to kubia-0.
irinag:~/environment $ █
```

EXPOSING STATEFUL PODS THROUGH A REGULAR CLUSTERIP SERVICE

- \$ k create -f kubia-clusterIP.yaml
- \$ k proxy
- \$ curl localhost:8001/api/v1/namespaces/default/services/kubia-public/proxy/

```
irinag:~/environment/week9/kubia-statefulset $ curl localhost:8001/api/v1/namespaces/default/pods/kubia-0/proxy/
You've hit kubia-0
Data stored on this pod: Hey there! This greeting was submitted to kubia-0.
irinag:~/environment/week9/kubia-statefulset $ curl localhost:8001/api/v1/namespaces/default/services/kubia-public/proxy/
You've hit kubia-0
Data stored on this pod: Hey there! This greeting was submitted to kubia-0.
```

DISCOVERING PEERS IN A STATEFULSET

Application should be K8s agnostic so querying K8s API to find all the pods in the cluster is undesirable

K8s uses DNS SRV records to facilitate the discovery. SRV records are used to point to hostnames and ports of servers providing a specific service. Kubernetes creates SRV records to point to the hostnames of the pods backing a headless service.

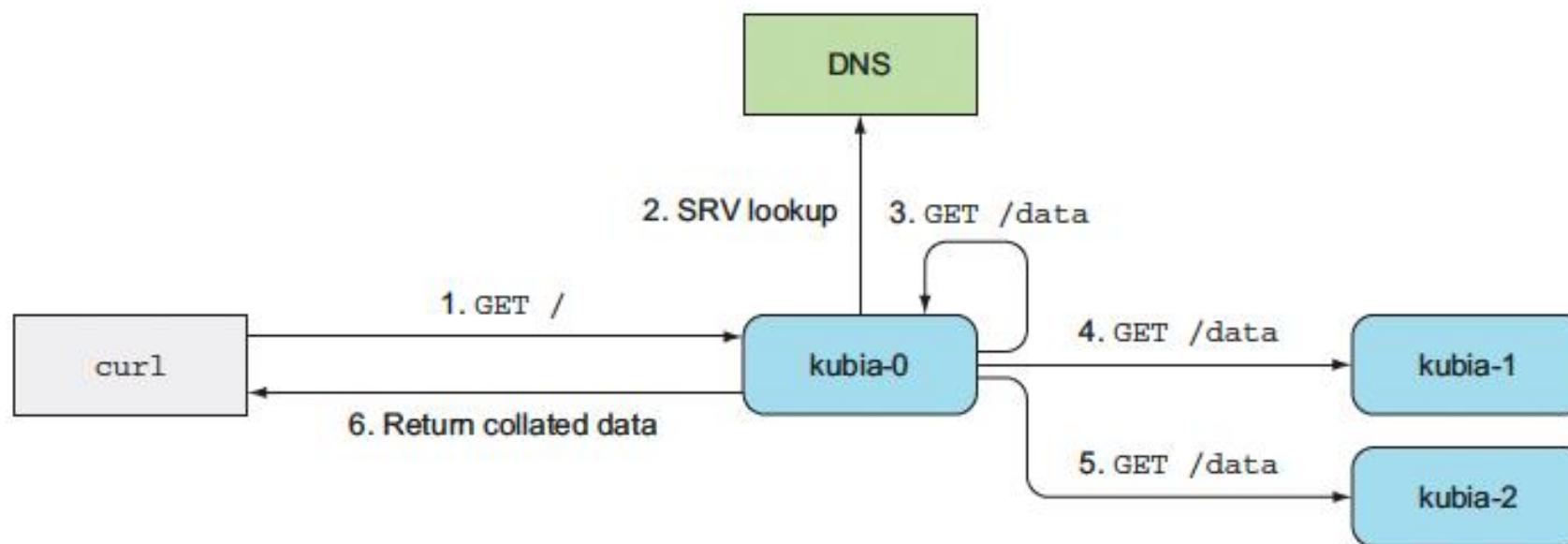
- # create headless service
- k apply -f kubia-headless.yaml
- # discovering the SRV records for the kubia cluster
- \$ k run -it srvlookup --image=tutum/dnsutils --rm --restart=Never -- /bin/bash
- # dig SRV kubia.default.svc.cluster.local

For a pod to get a list of all the other pods of a StatefulSet, all you need to do is perform an SRV DNS lookup.

```
;; ANSWER SECTION:  
kubia.default.svc.cluster.local. 5 IN SRV      0 50 80 kubia-0.kubia.default.svc.cluster.local.  
kubia.default.svc.cluster.local. 5 IN SRV      0 50 80 kubia-1.kubia.default.svc.cluster.local.  
  
;; ADDITIONAL SECTION:  
kubia-0.kubia.default.svc.cluster.local. 5 IN A 192.168.12.90  
kubia-1.kubia.default.svc.cluster.local. 5 IN A 192.168.86.61
```

IMPLEMENTING PEER DISCOVERY THROUGH DNS

- Data stored by individual pods is independent – this is not a cluster yet
- We can improve the functionality by discovering all the pods in the StatefulSet and serving data from all the pods in the cluster
- # Adding DNS lookup to obtain SRV record
- <https://github.com/luksa/kubernetes-in-action/blob/d15844af7b4153eb8d10ba56c5925a21205f490e/Chapter10/kubia-peers-image/app.js#L46>



UPDATING A STATEFULSET

- \$ k edit statefulset kubia #
Update container image
to luksa/kubia-pet-peers
- \$ k get po

```
        app: kubia
spec:
  containers:
    - image: luksa/kubia-pet-peers
      imagePullPolicy: Always
      name: kubia
    ports:
      - containerPort: 8080
        name: http
        protocol: TCP
      resources: {}
```

TRYING OUT OUR CLUSTERED DATA STORE

- # Send POST to a random pod through ClusterIP service
- \$ curl -X POST -d "The sun is shining"
localhost:8001/api/v1/namespaces/default/services/kubia-public/proxy/
- \$ curl -X POST -d "The rain is raining"
localhost:8001/api/v1/namespaces/default/services/kubia-public/proxy/
- # Reading data from a clustered data store
- \$ curl localhost:8001/api/v1/namespaces/default/services/kubia-public/proxy/

Nicely Done

```
irinag:~/environment/week9/kubia-statefulset $ curl localhost:80
You've hit kubia-0
Data stored in the cluster:
- kubia-0.kubia.default.svc.cluster.local: The rain is raining
- kubia-1.kubia.default.svc.cluster.local: The sun is shining
```

WORKSHOP 3

THE END

CLEANUP



```
# kill proxy pkill -f 'kubectl proxy --port=8080'  
  
# delete dashboard kubectl delete -f  
https://raw.githubusercontent.com/kubernetes/dashboard/${DASHBOARD_VERSION}/aio/deploy/recommended.yaml  
unset DASHBOARD_VERSION  
  
# Delete the week8 namespace  
  
$ k delete ns week8  
  
# Delete the cluster  
  
$ eksctl delete cluster --name clo835 --region us-east-1  
  
# Stop cloud9 instance
```

WEEK 10 FEELS LIKE 100

CONFIGMAPS AND SECRETS

I found my spirit animal



Glonk

(Glonk)

Does absolutely nothing and dies.

AGENDA

Configuring containerized applications

Passing command line options to the application

Setting environment variables exposed to the app

Configuring apps through ConfigMaps

Passing sensitive information to apps through Secrets

ImagePullSecrets

CONFIGURING CONTAINERIZED APPLICATIONS

- The application can be configured through command line arguments, environment variables or through config files
- Application images are immutable. Should we bake the configuration data in? (not a great idea)
- Ways to pass configuration data to containerized applications by:
 - Passing command-line arguments to containers
 - Setting custom environment variables for each container
 - Mounting configuration files into containers through a special type of volume

nginx.conf

```
user www; ## Default: nobody
worker_processes 5; ## Default: 1
error_log logs/error.log;
pid logs/nginx.pid;
worker_rlimit_nofile 8192;

events {
    worker_connections 4096; ## Default: 1024
}

http {
    include conf/mime.types;
    include /etc/nginx/proxy.conf;
    include /etc/nginx/fastcgi.conf;
    index index.html index.htm index.php;

    default_type application/octet-stream;
    log_format main '$remote_addr - $remote_user [$time_local] $status '
                  '$request' '$body_bytes_sent "$http_referer"'
                  '"$http_user_agent" "$http_x_forwarded_for"';
    access_log logs/access.log main;
    sendfile on;
    tcp_nopush on;
    server_names_hash_bucket_size 128; # this seems to be required for some vhosts

    server { # php/fastcgi
        listen 80;
        server_name domain1.com www.domain1.com;
        access_log logs/domain1.access.log main;
        root html;

        location ~ \.php$ {
            fastcgi_pass 127.0.0.1:1025;
        }
    }
}
```

REMINDER: PASSING COMMAND- LINE ARGUMENTS TO DOCKER CONTAINERS

- Docker reminder
 - ENTRYPOINT defines the executable invoked when the container is started.
 - CMD specifies the arguments that get passed to the ENTRYPOINT.
 - We can overwrite CMD arguments by providing command line parameters in docker cli
 - \$ docker run -it docker.io/luksa/fortune:args 15

```
FROM ubuntu:latest
RUN apt-get update ; apt-get -y install fortune
ADD fortuneloop.sh /bin/fortuneloop.sh
ENTRYPOINT ["/bin/fortuneloop.sh"]
CMD ["10"]
```

The exec form of the
ENTRYPOINT instruction

The default argument
for the executable

PASSING COMMAND LINE ARGUMENTS IN KUBERNETES

- We will usually override the CMD arguments only and not the process started in the container
- We will provide the process as well in generic images like busybox

```
kind: Pod
spec:
  containers:
    - image: some/image
      command: ["/bin/command"]
      args: ["arg1", "arg2", "arg3"]
```

Docker	Kubernetes	Description
ENTRYPOINT	command	The executable that's executed inside the container
CMD	args	The arguments passed to the executable

INJECTING CONFIGURATION WITH ENVIRONMENT VARIABLES

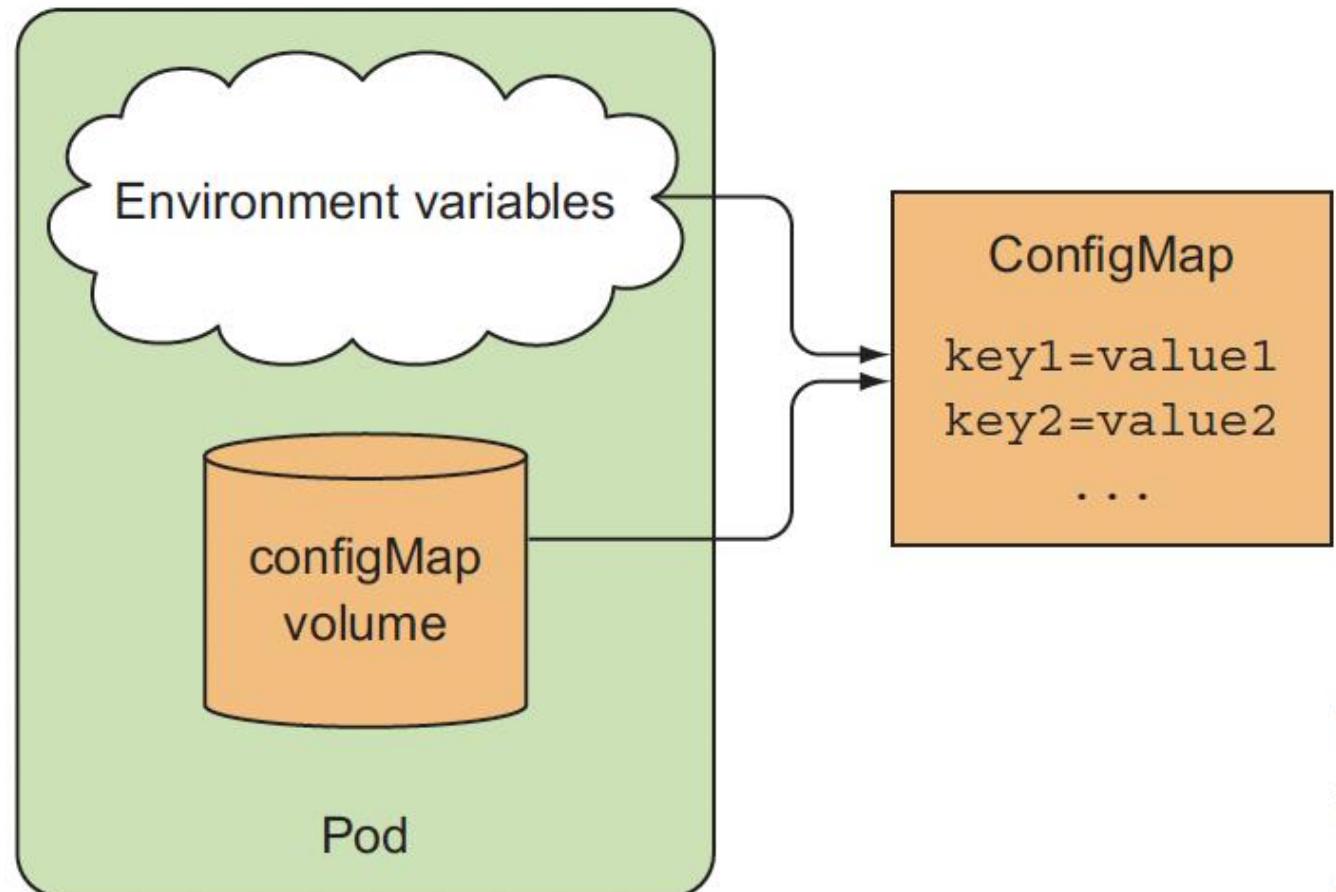
- Variables are set at the container level, not a pod level
- Kubernetes automatically injects environment variables corresponding to all services in the namespace
- Drawbacks:
 - Environment variables defined in the Pod manifest are hard coded. It makes it difficult to maintain different values in prod and nonprod environments
 - Configuration should not be part of the Pod description

```
kind: Pod
spec:
  containers:
    - image: luksa/fortune:env
      env:
        - name: INTERVAL
          value: "30"
      name: html-generator
```

Adding a single variable to the environment variable list

CONFIGMAPS

- ConfigMap is a resource in K8s
- Contains key/value pairs
- Configuration parameters made accessible to containers as environment variables or files in a mounted folder



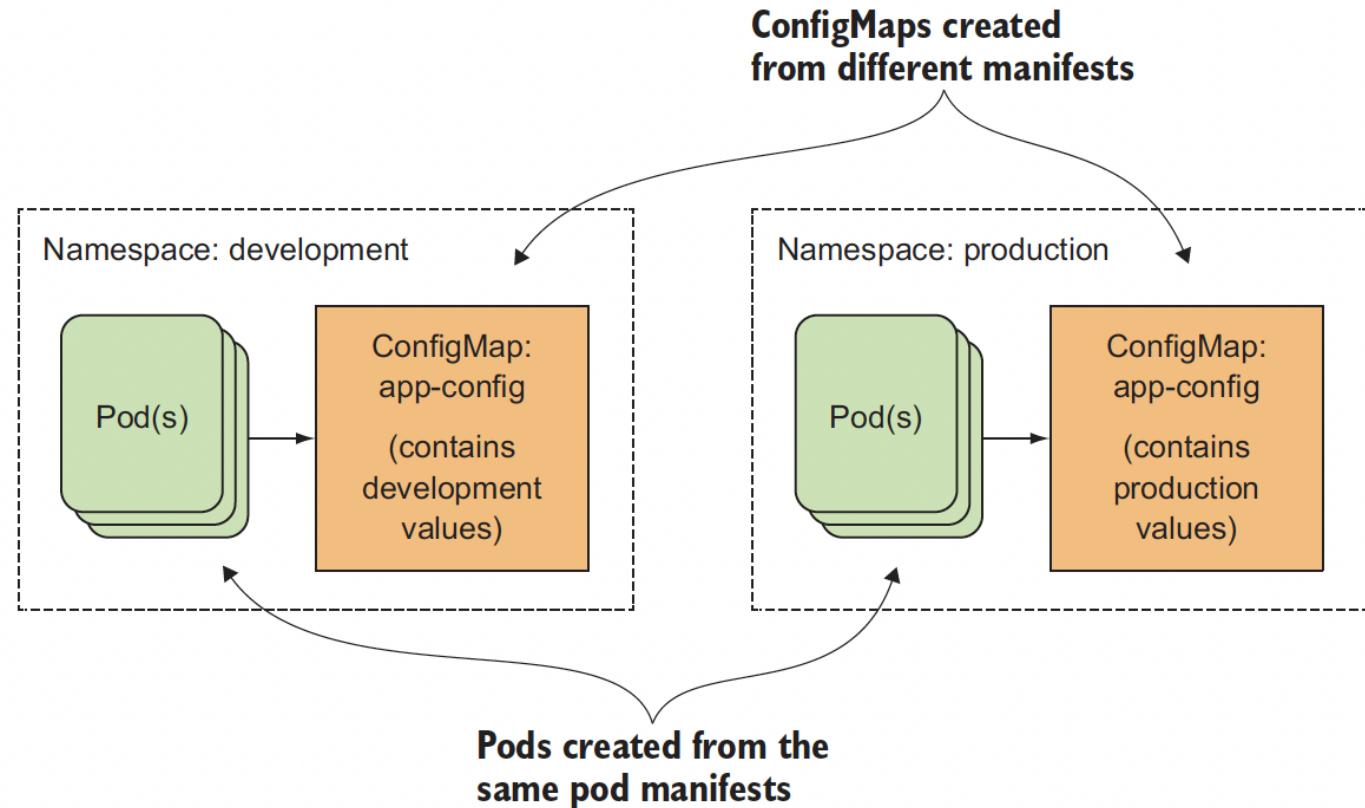
MULTI-ENVIRONMENT SUPPORT USING CONFIGMAPS



Pods share the same manifest referencing configuration values from configMap



ConfigMaps are created from different manifests and have environment-specific values



WORKSHOP 1: CONFIGURE APPLICATIONS WITH ENVIRONMENT VARIABLES AND CONFIGMAPS

CONFIGURE INTERVAL IN FORTUNE TELLING APPLICATION USING ENVIRONMENT VARIABLES VALUE IN THE POD MANIFEST

- # Deploy fortune application, configure the interval with environment variables populated from the Pod manifest
- \$ k create ns week10
- \$ k create -f fortune-pod-env.yaml -n week10
- \$ k describe pod fortune-env -n week10
- # Is the interval value reflected in the log?
- \$ k logs fortune-env -c html-generator -n week10

```
Namespace:      week10
Priority:      0
Node:          ip-192-168-82-156.ec2.internal/192.168.82.156
Start Time:    Mon, 11 Jul 2022 17:58:45 +0000
Labels:        <none>
Annotations:   kubernetes.io/psp: eks.privileged
Status:        Running
IP:            192.168.67.125
IPs:
  IP: 192.168.67.125
Containers:
  html-generator:
    Container ID:  docker://cce73c2148e7705d26119452fd6624
    Image:         luksa/fortune:env
    Image ID:     docker-pullable://luksa/fortune@sha256:8
    Port:          <none>
    Host Port:    <none>
    State:        Running
    Started:     Mon, 11 Jul 2022 17:58:50 +0000
    Ready:        True
    Restart Count: 0
Environment:
  INTERVAL: 30
```

CONFIGURE INTERVAL IN FORTUNE TELLING APPLICATION USING ENVIRONMENT VARIABLES VALUE FROM THE CONFIGMAP

- # Create configMap from the manifest
- \$ k create -f fortune-config.yaml -n week10
- # use fortune-pod-env-configmap.yaml to deploy the application
- \$ k create -f fortune-pod-env-configmap.yaml -n week10
- \$ k describe pod fortune-env-from-configmap -n week10
- # Is the interval value reflected in the log?
- \$ k logs fortune-env-from-configmap -c html-generator -n week10
- # Log into container and verify the environment variable was set
- \$ k exec -it fortune-env-from-configmap -n week10 -c html-generator -- /bin/bash
- # printenv | grep -i interval

```
Containers:
  html-generator:
    Container ID:  docker://12eac7a1572eb1593e14f7521e9b91583130b79c8f37e6d3f558f52617c159d9
    Image:         luksa/fortune:env
    Image ID:      docker-pullable://luksa/fortune@sha256:8af10b8eb1b1dcc6512e0061c0722db43f4795ae
    Port:          <none>
    Host Port:    <none>
    State:        Running
    Started:      Mon, 11 Jul 2022 18:33:03 +0000
    Ready:        True
    Restart Count: 0
    Environment:
      INTERVAL: <set to the key 'sleep-interval' of config map 'fortune-config'> Optional: false
    Mounts:
```

WHAT IF APPLICATION IS ACCEPTING CONFIGURATION THROUGH COMMAND LINE ARGUMENTS ONLY?

```
kind: Pod
metadata:
  name: fortune-args-from-configmap
spec:
  containers:
    - image: luksa/fortune:args
      env:
        - name: INTERVAL
          valueFrom:
            configMapKeyRef:
              name: fortune-config
              key: sleep-interval
      args: ["$(INTERVAL)"]
```

- # Provide arguments that act as CMD in DOCKERFILE.
The value is hard coded in the pod manifest
- k apply -f fortune-pod-args.yaml -n week10
- # Use configMap to provide the value in args and
decouple pod manifest from the configuration data

DIFFERENT WAYS TO CREATE CONFIGMAPS

- # From the content of a file
- \$ kubectl create configmap my-config --from-file=config-file.conf
- # From all the files in the folder
- \$ kubectl create configmap my-config --from-file=/path/to/dir
- # From literal values provided in command line
- \$ kubectl create configmap fortune-config --from-literal=sleep-interval=25 --from-literal=sky-color=blue

USING A CONFIGMAP VOLUME TO EXPOSE CONFIGMAP ENTRIES AS FILES

```
irinag:~/environment/week10 $ k get configmap/fortune-config -n week10 -o yaml
apiVersion: v1
data:
  my-nginx-config.conf: |-
    server {
      listen 80;
      server_name www.kubia-example.com;
      gzip on;
      gzip_types text/plain application/xml;
      location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
      }
    }
    sleep-interval: "22"
kind: ConfigMap
```

- # Delete the previous configMap
- \$ k delete configmap fortune-config -n week10
- # Create a new configMap from the files in the configmap-files folder. Each file becomes a key entry in the configMap
- \$ kubectl create configmap fortune-config --from-file=configmap-files -n week10
- # Examine the data in the configMap
- \$ k get configmap/fortune-config -n week10 -o yaml
- # The nginx configuration will get loaded from the configMap
- \$ k apply -f fortune-pod-configmap-volume.yaml -n week10

USING A CONFIGMAP VOLUME TO EXPOSE CONFIGMAP ENTRIES AS FILES, CONT

- # Verifying the settings of nginx
- \$ k port-forward fortune-configmap-volume 8080:80 -n week10
- \$ kubectl exec fortune-configmap-volume -c web-server ls /etc/nginx/conf.d
- \$ curl -H "Accept-Encoding: gzip" -I localhost:8080
- # Examine the content of the mounted /etc/nginx/conf.d directory :
- \$ kubectl exec fortune-configmap-volume -c web-server -n week10 -- ls /etc/nginx/conf.d

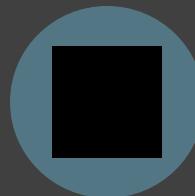
```
irinag:~/environment $ curl -H "Accept-Encoding: gzip" -I localhost:8080
HTTP/1.1 200 OK
Server: nginx/1.23.0
Date: Mon, 11 Jul 2022 23:09:34 GMT
Content-Type: text/html
Last-Modified: Mon, 11 Jul 2022 23:09:14 GMT
Connection: keep-alive
ETag: W/"62ccad9a-4a"
Content-Encoding: gzip
```

WORKSHOP 1: THE END

STORING SENSITIVE DATA IN K8S: INTRODUCING SECRETS



Key value pairs that are stored as base64 encoded strings



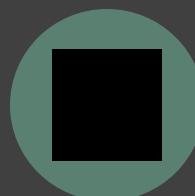
Like configMaps, available to containers as environment variables or mounted files



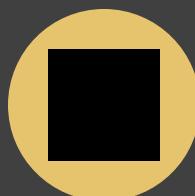
Never stored on the disk of the underlying host



Stored in etcd and are encrypted



Size is limited to 1MB



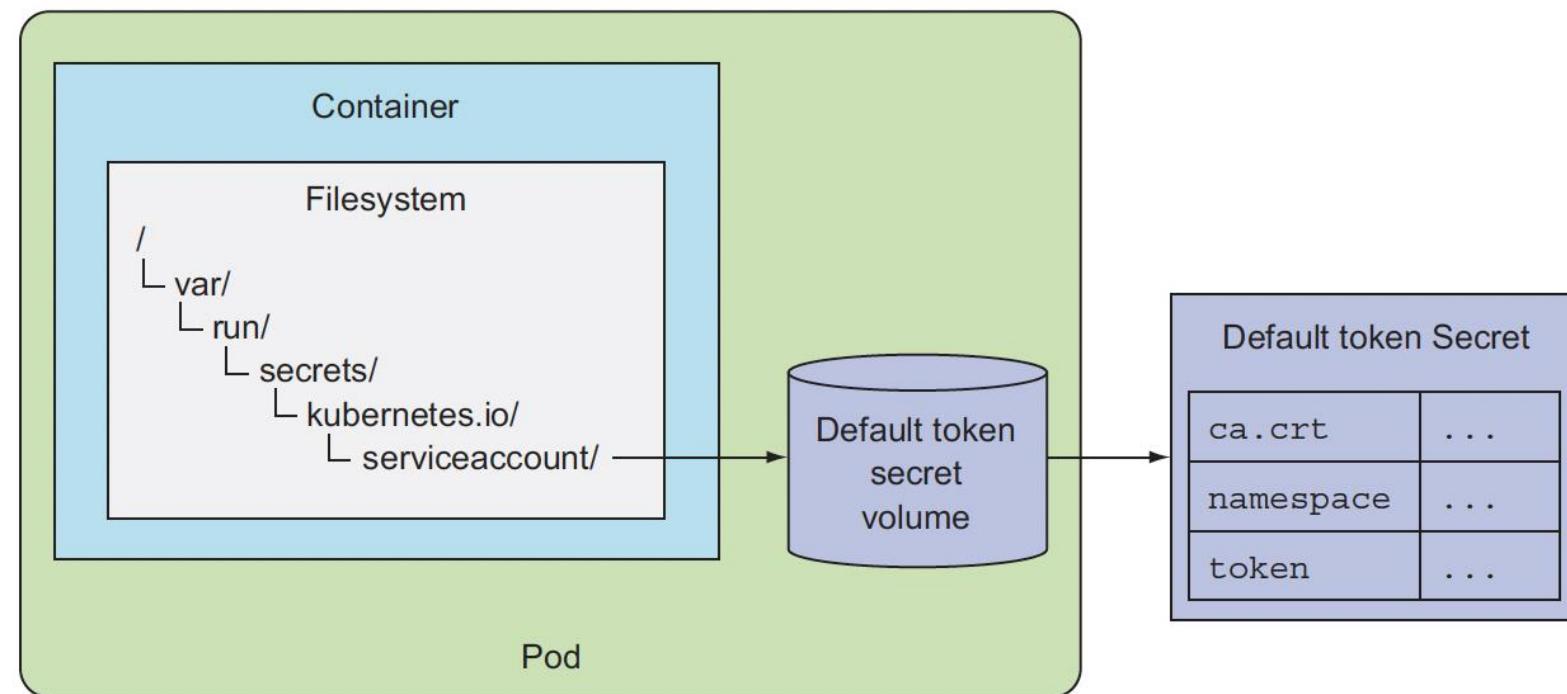
Available to application in its original form, application does not need to decode it

DEFAULT SECRETS

```
/environment $ k get secrets -n kube-system
NAME                           TYPE
etach-controller-token-mfkmj   kubernetes.io/secrets
d-provider-token-mgxds        kubernetes.io/secrets
-token-xnxl8                  kubernetes.io/secrets
ate-controller-token-7svx5    kubernetes.io/secrets
ole-aggregation-controller-token-tj29p kubernetes.io/secrets
token-cpqph                   kubernetes.io/secrets
controller-token-24wlk        kubernetes.io/secrets
et-controller-token-mqb7f    kubernetes.io/secrets
token-xzltz                   kubernetes.io/secrets
nt-controller-token-jdn4q    kubernetes.io/secrets
on-controller-token-s6j4t     kubernetes.io/secrets
resource-controller-token-9fjcj kubernetes.io/secrets
-controller-token-66v5c       kubernetes.io/secrets
slice-controller-token-pnth5  kubernetes.io/secrets
slicemirroring-controller-token-bf76l kubernetes.io/secrets
ontroller-token-q66tg        kubernetes.io/secrets
```

- Default secrets are mounted into every container we run unless specified otherwise
- Default secrets are mounted as files into "/var/run/secrets/kubernetes.io/serviceaccount" folder in the container
- The folder has 3 entries.
 - ca.crt
 - namespace
 - token
- Default secrets allow containers to communicate with K8s API using read-only permissions

THE DEFAULT-TOKEN SECRET IS CREATED AUTOMATICALLY AND A CORRESPONDING VOLUME IS MOUNTED IN EACH POD AUTOMATICALLY.



WORKSHOP 2: WORKING WITH K8S SECRETS



GOALS



Learn how to
create the secrets



Compare Secrets
and ConfigMaps
in K8s



Learn how to use
K8s secrets in the
application

EXPLORING DEFAULT SECRETS

- \$ k create ns week10
- \$ k get secrets -n week10 -o yaml
- \$ k describe secrets -n week10
- # Create mypod pod in the week10 namespace using nginx image
- \$ k run mypod --image nginx -n week10
- \$ k describe pod mypod -n week10
- # Explore the secrets mounted
at /var/run/secrets/kubernetes.io/serviceaccount/
- \$ k exec mypod -n week10 -- ls /var/run/secrets/kubernetes.io/serviceaccount/
- \$ kubectl exec mypod -n week10 --
cat /var/run/secrets/kubernetes.io/serviceaccount/namespace
- \$ kubectl exec mypod -n week10 --
cat /var/run/secrets/kubernetes.io/serviceaccount/token
- \$ kubectl exec mypod -n week10 --
cat /var/run/secrets/kubernetes.io/serviceaccount/ca.crt

WORKING WITH SECRETS – CREATE A SECRET, COMPARE K8S SECRETS AND CONFIGMAPS

- # Create the private key and the certificate
- \$ openssl genrsa -out https.key 2048
- \$ openssl req -new -x509 -key https.key -out https.cert -days 3650 -subj /CN=www.kubia-example.com
- # Create a secret with the key and the certificate
- \$ k create secret generic fortune-https --from-file=https.key --from-file=https.cert -n week10
- # Comparing Secrets and ConfigMaps
- \$ k create -f configMaps/fortune-config.yaml -n week10
- \$ k get secret fortune-https -o yaml -n week10
- \$ k get configmap fortune-config -n week10 -o yaml

MODIFYING G CONFIGMAP TO WORK WITH CERTIFICATES

```
data:  
my-nginx-config.conf: |  
  server {  
    listen 80;  
    listen 443 ssl;  
    server_name www.kubia-example.com;  
    ssl_certificate certs/https.cert;  
    ssl_certificate_key certs/https.key;  
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;  
    ssl_ciphers HIGH:!aNULL:!MD5;  
    location / {  
      root /usr/share/nginx/html;  
      index index.html index.htm;  
    }  
  }
```

- # Create fortune-config from the files in configmap-files folder
- \$ k create configmap fortune-config --from-file=week10/secrets/configmap-files/ -n week10
- # This nginx config configures the server to read the certificate and key files from /etc/nginx/certs

DEPLOY THE POD, VERIFY HTTPS CONNECTION

- # Deploy the fortune application
- \$ k create -f week10/secrets/fortune-pod-https.yaml -n week10
- # Verify HTTPS connection using port-forwarding to the pod and curl command
- \$ k port-forward fortune-https -n week10 8443:443
- \$ curl https://localhost:8443 -k -v

```
* ALPN, server accepted to use http/1.1
* Server certificate:
*   subject: CN=www.kubia-example.com
*   start date: Jul 13 15:36:42 2022 GMT
*   expire date: Jul 10 15:36:42 2032 GMT
*   issuer: CN=www.kubia-example.com
*   SSL certificate verify result: self signed certificate (18), continuing anyway.
> GET / HTTP/1.1
> Host: localhost:8443
> User-Agent: curl/7.79.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Server: nginx/1.23.0
```

UNDERSTANDING THE WAY SECRET VOLUMES ARE STORED

- # Secret volumes look like files on the disk at /etc/nginx/certs folder.
- # It contradicts the statement that secrets are not stored on host's filesystem.
- # K8s secrets are stored on "in-memory filesystem" /tmpfs
- \$ k exec fortune-https -c web-server --mount | grep certs
- # Sensitive data stored in a secret is never written to the disk where it could be compromised

WORKSHOP 2: THE END

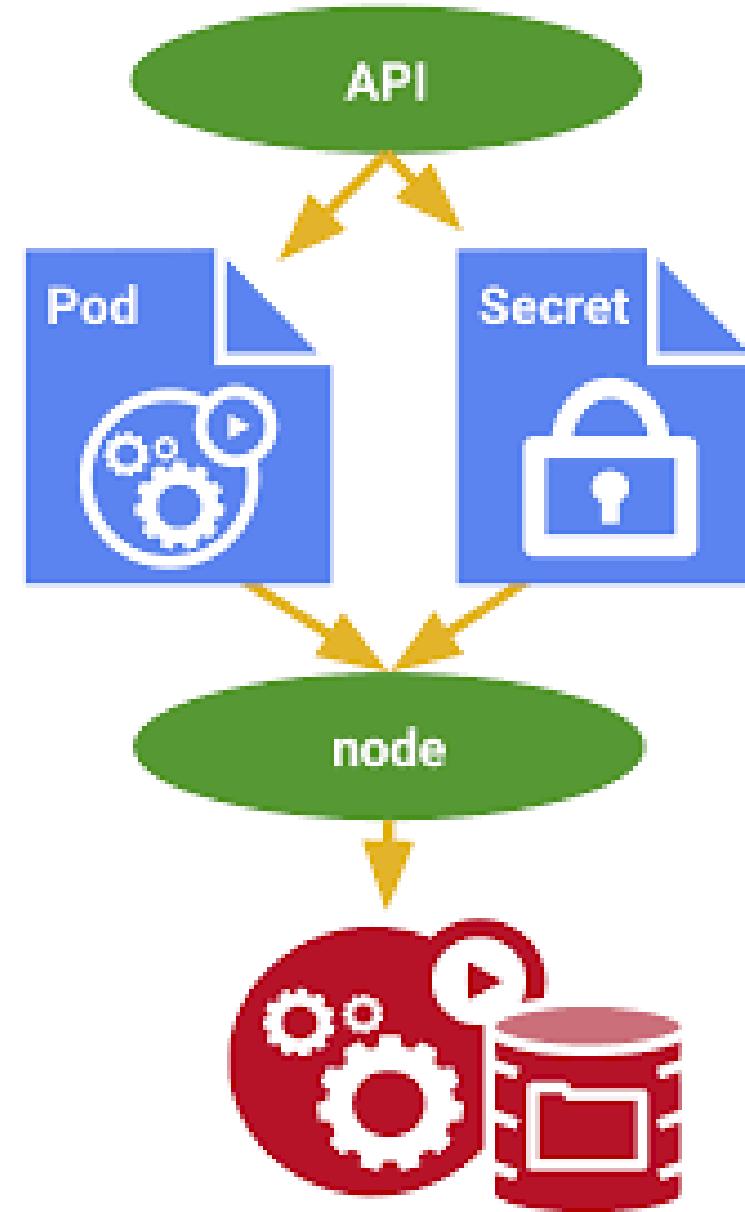


IMAGE PULL SECRET

- Container run time needs to be authenticated with the private registry to pull the image
- Special "imagePullSecret" can be used to provide the credentials
- Local credentials can be provided as a secret that container run time is used to pull the image
- The secret should be created in the same namespace as the pod that uses the secret
- kubelet checks that it can access that Secret before attempting to pull the image.

```
apiVersion: v1
kind: Pod
metadata:
  name: private-reg
spec:
  containers:
  - name: private-reg-container
    image: <your-private-image>
  imagePullSecrets:
  - name: regcred
```

```
kubectl create secret generic regcred \
--from-file=.dockerconfigjson=<path/to/.docker/config.json> \
--type=kubernetes.io/dockerconfigjson
```



WORKSHOP 3: USING IMAGEPULLSECRET WITH KIND RUNNING IN CLOUD9 ENVIRONMENT

GOALS



Learn how to create secret from docker config file



Use imagePullSecret in pod manifest



Deploy application from private Amazon ECR registry to KIND cluster hosted in Cloud9 environment

BUILDING AND SHARING APPLICATION IMAGE

- # Build an image and test it locally
- \$ docker build -t 498343093558.dkr.ecr.us-east-1.amazonaws.com/cowsay:v1.0 .
- \$ docker run -it 498343093558.dkr.ecr.us-east-1.amazonaws.com/cowsay:v1.0
- # Create a private registry named cowsay
- \$ aws ecr create-repository --repository-name cowsay
- # Examine local docker credentials
- \$ cat ~/.docker/config.json
- \$ Login into AWS ECR
- \$ aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin [YOUR ACCOUNT ID].dkr.ecr.us-east-1.amazonaws.com
- # Examine local docker credentials
- \$ cat ~/.docker/config.json
- # Push the image to Amazon ECR repository
- \$ docker push [YOUR ACCOUNT ID].dkr.ecr.us-east-1.amazonaws.com/cowsay:v1.0

DEPLOY THE APPLICATION USING IMAGE FROM THE PRIVATE REPOSITORY

```
# Create kind cluster in Cloud9 environment  
$ k create cluster --config kind.yaml  
  
# Make sure kubectl reports matching versions  
$ k version  
  
# Create a namespace  
$ k create ns cowsay  
  
# Update the pod deployment manifests with the name of your Amzon ECR repo  
  
# Attempt to deploy the pod with the image hosted in private registry without imagePullSecrets specified in the manifest  
$ k apply -f pod-without-imagepullsecrets.yaml  
  
Was the pod created successfully? Examine the error using "k describe cowsay-pod -n cowsay". Explain the error.
```

DEPLOY THE APPLICATION USING IMAGE FROM THE PRIVATE REPOSITORY

```
# Create a secret from using authentication data in  
the ~/.docker/config.json file  
  
$ kubectl create secret generic ecr-secret --  
from-file=.dockerconfigjson=/home/ec2-  
user/.docker/config.json --  
type=kubernetes.io/dockerconfigjson -n cowsay  
  
# Deploy the manifest with imagePullSecrets  
included  
  
$ k apply -f pod-without-imagepullsecrets.yaml  
  
# Verify the application worked as expected  
  
$ k logs pod/cowsay-pod-private -n cowsay
```

WORKSHOP 3

THE END

```
voclabs:~/environment/imagePullSecrets $ k logs pod/cowsay-pod-private -n cowsay
```

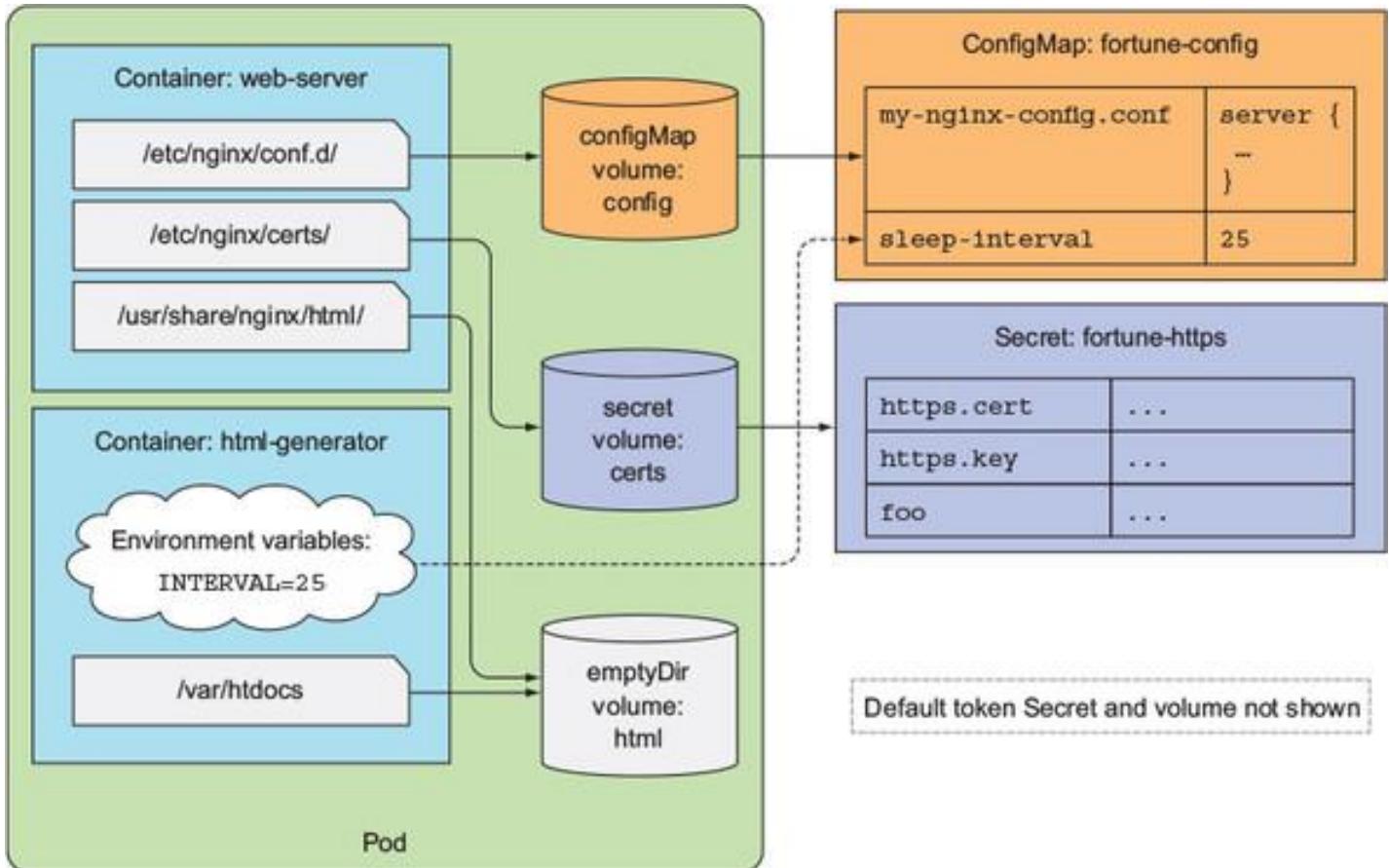
```
/ Compassion -- that's the one things no \
| machine ever had. Maybe it's the one
| thing that keeps men ahead of them.
|
| -- McCoy, "The Ultimate Computer",
\ stardate 4731.3
```



NOTES ABOUT SECRETS AS ENVIRONMENT VARIABLES

- Secrets can be provided to the containers via environment variables
- This is not the recommended way to use secrets
 - Environment variables are often dumped into log files those exposing the secrets
 - Child processes will inherit environment variables exposing application to the supply chain attack if the third-party libraries are compromised

SUMMARY: INJECTING DATA INTO CONTAINERS IN KUBERNETES



REFERENCES

- <https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/>
- <https://kubernetes.io/docs/tutorials/configuration/>
- <https://github.com/mmumshad/simple-webapp-mysql>
- <https://kubernetes.io/docs/concepts/configuration/secret/>
- <https://kubernetes.io/docs/tasks/inject-data-application/distribute-credentials-secure/>
- <https://github.com/luksa/kubernetes-in-action/tree/master/Chapter07>

CLEANUP



Delete the cluster

```
$ eksctl delete cluster --name clo835
```

Stop cloud9 instance



I SEE KUBERNETES

EVERYWHERE

WEEK 11

INTRODUCTION TO K8S RBAC

IRSA (IAM ROLES FOR SERVICE ACCOUNTS)



Okay...I'm Ready

Lets Go!

AGENDA

Understanding authentication

Enable IAM users access to Amazon EKS cluster

What ServiceAccounts are and why they're used

Understanding the role-based access control (RBAC) plugin

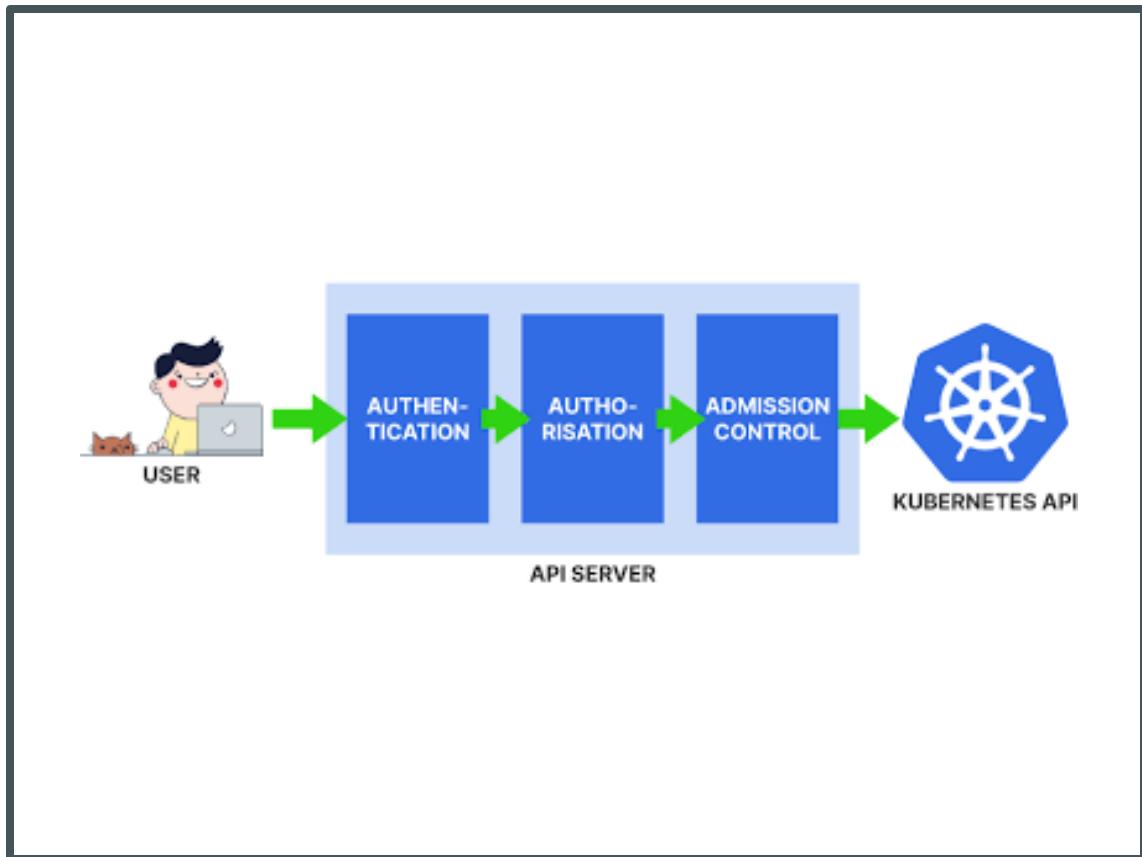
Using Roles and RoleBindings

Using ClusterRoles and ClusterRoleBindings

Understanding the default roles and bindings

IRSA (IAM Roles for Serviceaccounts) in AWS

AUTHENTICATION AND AUTHORIZATION PROCESS IN K8S



- Kubernetes API can be configured with one or more **authentication** plugins
 - X509 client certificates
 - Static token file
 - Bootstrap tokens
 - Static password file
 - Service account tokens
 - OpenID Connect tokens
 - Webhook token authentication
 - Authenticating proxy
- Auth plugin retrieves username, user ID and client's K8s groups from client's request
- K8s API proceeds to authorization phase

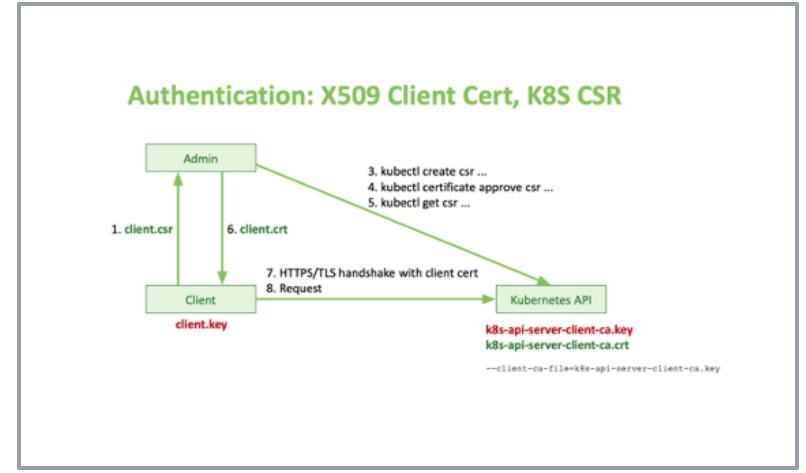
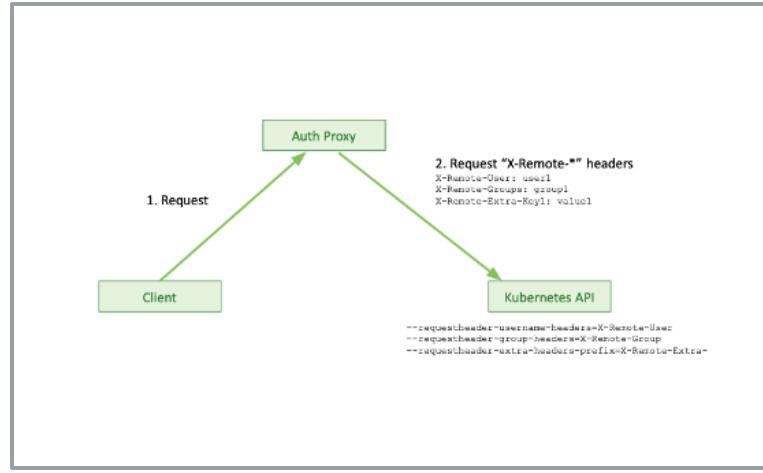
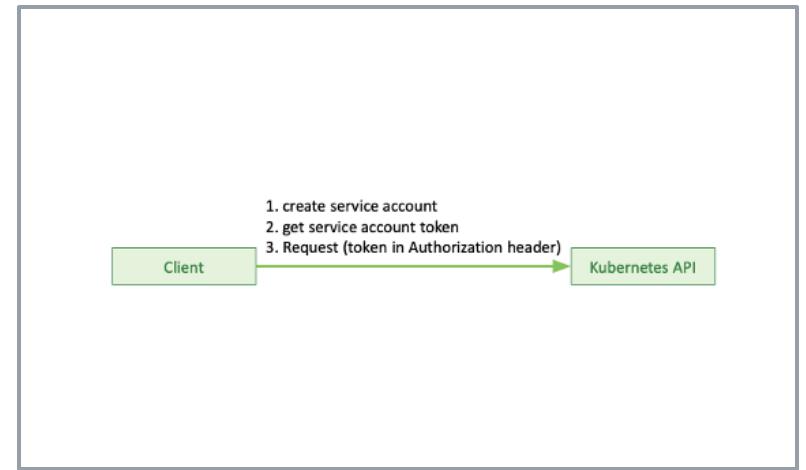
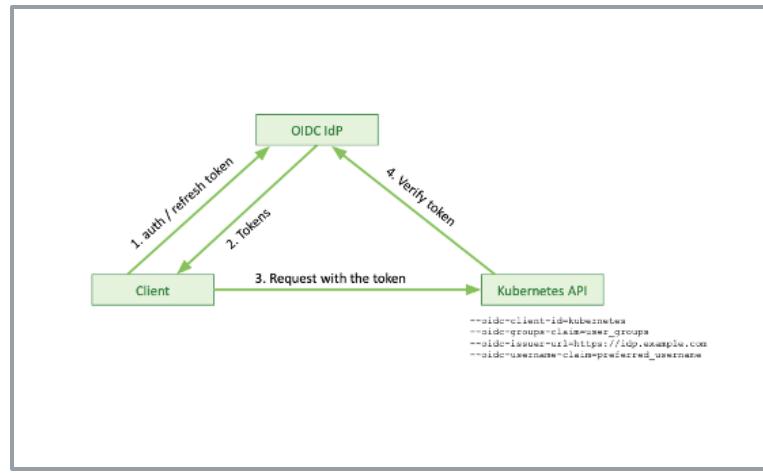
K8S API CLIENT TYPES

K8s API distinguishes between 2 types of clients

- Users – human clients
 - Admins deploying K8s add-ons and extensions
 - Developers deploying applications
 - **K8s does not manage users. User management is provided by a third party such as**
 - SSO
 - AWS IAM
 - Google accounts
- Processes
 - Application interacting with K8s API
 - Observability tools collecting metrics from K8s API
 - Security tools querying K8s API
 - **K8s manages serviceaccounts**

USERS AUTHENTICATION METHODS

- K8s users can be authenticated using one of the following methods:
 - Client certificates
 - Bearer tokens
 - Authenticating proxy
- Authentication plugin returns all the **K8s groups** the user belongs to



GROUPS IN KUBERNATES

To avoid assigning roles to individual users, K8s support the use of groups.

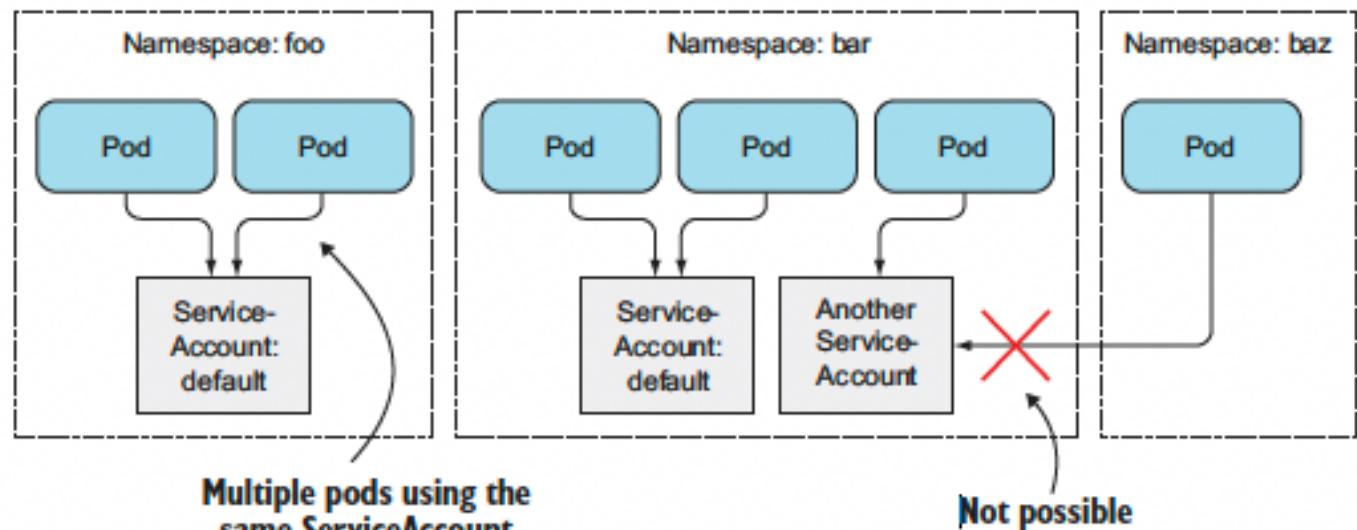
Kubernetes includes two types of groups:

- **System assigned:** These groups start with the system: prefix and are assigned by the API server.
 - system:authenticated
 - system:serviceaccounts:namespace
 - System:unauthenticated
- **User-asserted groups:** These groups are asserted by the authentication system either in the token provided to the API server or via the authentications webhook.
 - Groups don't exist as objects in the API server.
 - Groups are asserted at authentication time by external users and tracked locally for system-generated groups

Users can be assigned to arbitrary groups

SERVICEACCOUNTS

- Default serviceaccount per namespace
- First class resource in K8s
- Multiple pods can use the same serviceaccount
- Pod can only use one serviceaccount located in its namespace
- Serviceaccount's token used to authenticate the pod and get pod's username
- Authorization plugin configured by Admin is used to identify pod's permissions. RBAC is a default plugin



WORKSHOP 1 – CREATE AND EXPLORE SERVICEACCOUNT

```
HEADER: ALGORITHM & TOKEN TYPE
{
  "alg": "RS256",
  "kid": "f9Xha2QgbLFC7rtY0RQYgxJRuj4MnhULitXMgn3CadU"
}

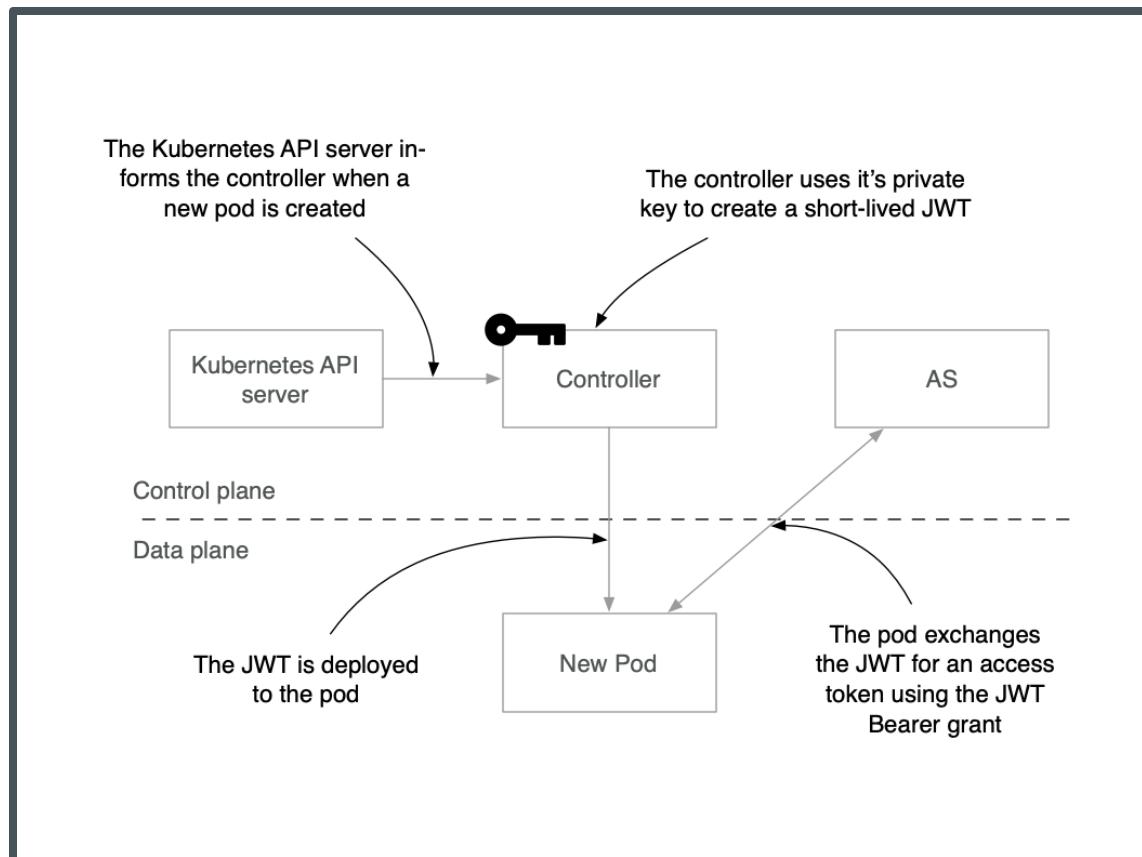
PAYLOAD: DATA
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "week10",
  "kubernetes.io/serviceaccount/secret.name": "clo835-
token-7zwf5",
  "kubernetes.io/serviceaccount/service-account.name": "clo835",
  "kubernetes.io/serviceaccount/service-account.uid": "93ba91df-4e0b-4a1e-9eed-ccdd6cabfee3",
  "sub": "system:serviceaccount:week10:clo835"
}
```

- \$ kubectl create ns week11
- \$ kubectl create serviceaccount clo835 –n week11
- \$ kubectl describe sa clo835 –n week11
- \$ kubectl describe secret clo835-token-xxx –n week11
- # The authentication tokens used in ServiceAccounts are JWT tokens.
- # Copy the token of the default serviceaccount in week11 namespace and decode it using <https://jwt.io/>

- Image Pull secret used to authenticate against private registries
- Only mountable secrets can be mounted into containers

```
irinag:~/environment $ k describe sa clo835 -n week10
Name:           clo835
Namespace:      week10
Labels:          <none>
Annotations:    <none>
Image pull secrets: <none>
Mountable secrets: clo835-token-7zwf5
Tokens:          clo835-token-7zwf5
Events:          <none>
```

UNDERSTANDING THE JWT TOKEN



- The SA name is visible in "sub": "sub": "system:serviceaccount:week10:clo835"
- The token is signed by the master key of the Kubernetes cluster; in Amazon EKS this key is stored in Control plane
- Pods are using JWT token for **authentication** when accessing K8s API
- The API server validates the token by using its public key:
 - K8s API verifies that the token was not tampered with
 - K8s API verifies that the token was issued by this Kubernetes cluster

Note: If SA is not specified, the "default" SA is used and the default secrets are mounted into the containers

DEPLOY A POD WITH NON-DEFAULT SERVICE ACCOUNT

- # From week11 folder
- \$ k create -f curl-custom-sa-token.yaml
- # The SA specific token is mounted into the pod
- \$ k exec -it curl-custom-sa-token -c main -n week11 -- /bin/sh
- # Send curl request from inside the pod using environment variables
- /\$ printenv | grep KUBERNETES
- / \$
curl https://\$KUBERNETES_SERVICE_HOST:\$KUBERNETES_SERVICE_PORT/api/v1/namespaces/default/pods/ -k
- # Why are we getting the "Forbidden" response?

```
/ $ curl https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT -k
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

},
  "status": "Failure",
  "message": "forbidden: User \\"system:anonymous\\" cannot get path \"/\"",
  "reason": "Forbidden",
  "details": {

},
  "code": 403
```

ADDING AUTHENTICATION TO THE CURL REQUEST

- # Create environment variables for certificate and token
- / \$ CERT=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
- / \$ TOKEN=\$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
- # Send the curl request with the certificate and token for authentication. This time K8s API server authenticates our pod as expected
- curl --cacert \$CERT -H "Authorization: Bearer \$TOKEN"
"https://\$KUBERNETES_SERVICE_HOST:\$KUBERNETES_SERVICE_PORT/api/v1/namespaces/default/pods/"

```
/ $ curl --cacert $CERT -H "Authorization: Bearer $TOKEN" "https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT/api/v1/namespaces/default/pods/"  
{  
    "kind": "Status",  
    "apiVersion": "v1",  
    "metadata": {  
  
    },  
    "status": "Failure",  
    "message": "pods \"\\u200b\" is forbidden: User \"system:serviceaccount:week11:clo835\" cannot get resource \"pods\" in API group \"\" in the namespace \"default\"",  
    "reason": "Forbidden",  
    "details": {  
        "name": "",  
        "kind": "pods"  
    },  
    "code": 403  
}
```

DEPLOY A POD WITH NON-DEFAULT SERVICE ACCOUNT AND AMBASSADOR CONTAINER

- # From week11 folder
- \$ k create -f curl-custom-sa.yaml
- # The SA specific token is mounted into the pod
- \$ k exec -it curl-custom-sa -c main -n week11 -- /bin/sh
- # Send curl request from inside the pod, the request is denied. We got a proper response, but the SA is not allowed to list the pods. **Why?**
- / \$ curl localhost:8001/api/v1/pods

```
/ $ curl localhost:8001/api/v1/pods
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

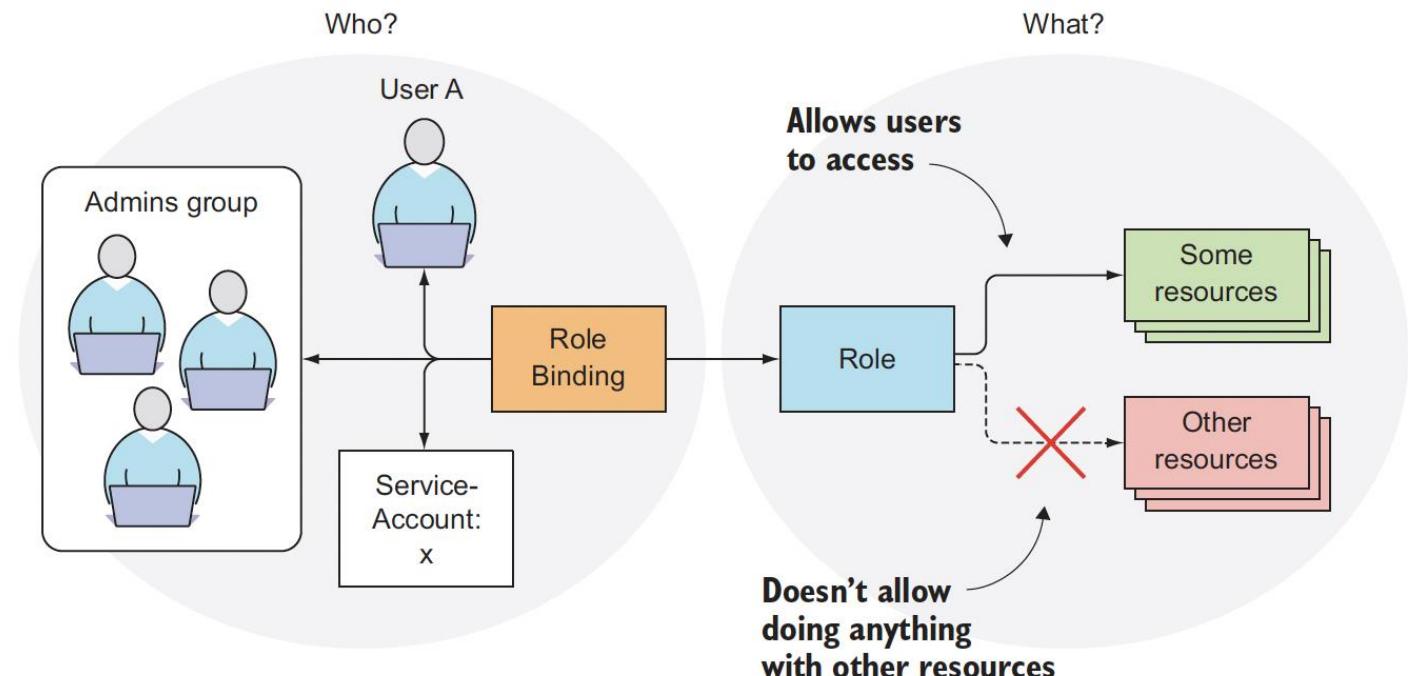
},
  "status": "Failure",
  "message": "pods is forbidden: User \\"system:serviceaccount:week11:clo835\\" cannot list resource \\\"pods\\\" in API group \\"\\\" at the cluster scope",
  "reason": "Forbidden",
  "details": {
    "kind": "pods"
},
  "code": 403
}/ $ █
```

AUTHORIZATION IN K8S

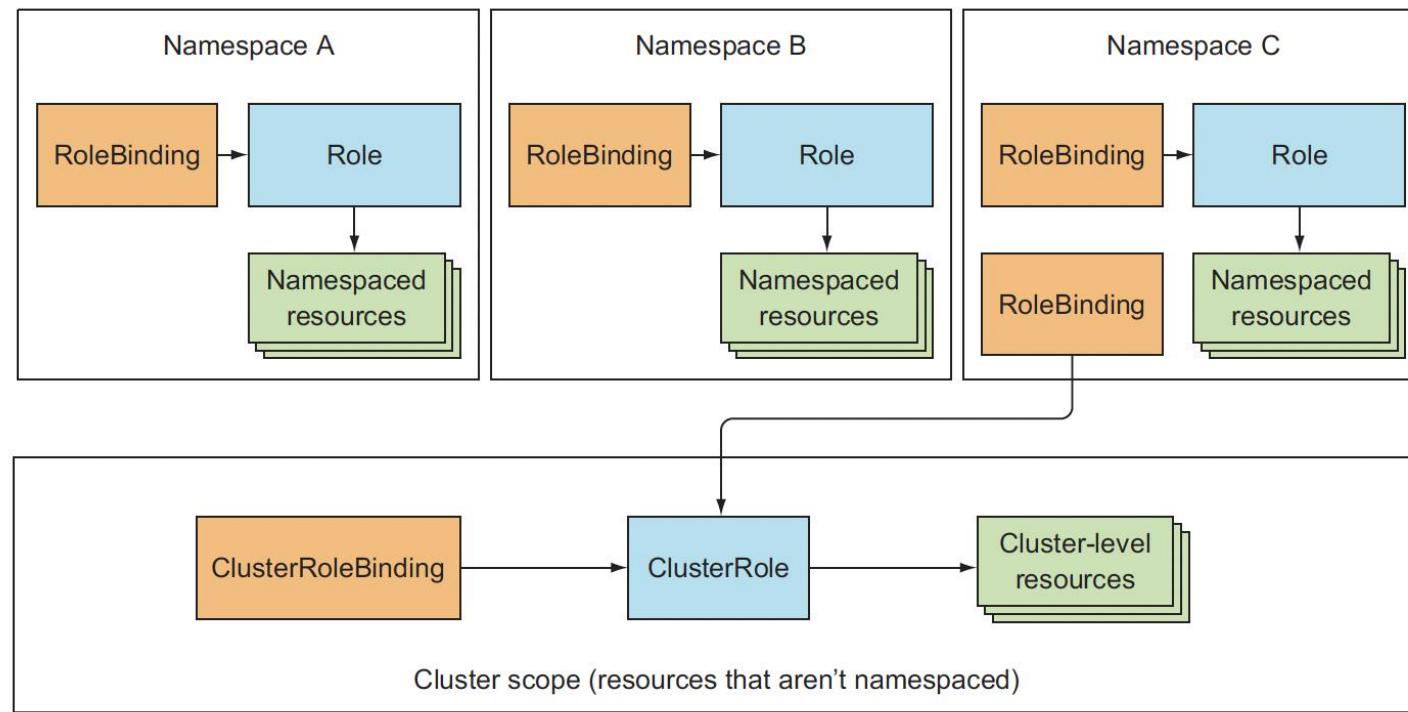
- Up until version 1.6.0, authentication token was enough to get authorized access to K8s API
- In 1.8.0, Role Based Access Control (RBAC) authorization plugin was released as GA
- RBAC authorization is the default in Amazon EKS
- A few properties of RBAC authorization plugin:
 - RBAC prevents unauthorized users from viewing or modifying the cluster state.
 - The default ServiceAccount or newly created ServiceAccount and CRUD operations unless you grant it privileges
- Other authorization plugins supported by K8s:
 - Attribute-based access control (ABAC) plugin
 - Web-Hook plugin
 - Custom plugin implementations.

INTRODUCING RBAC RESOURCES

- Roles and ClusterRoles - specify which verbs can be performed on which resources.
- Verbs: [create delete deletecollection get list patch update watch]
- RoleBindings and ClusterRoleBindings, which bind the above roles to specific users, groups, or ServiceAccounts.
- Roles define what can be done, while bindings define who can do it



WORKSHOP 2 – ROLES AND ROLEBINDINGS



CONNECT TO K8S API USING DEFAULT SA

- # Create two namespaces and run proxy containers in the namespaces
- # kube-proxy containers intercept the curl requests and add authentication token mounted to the container
- \$ k create ns week11-1
- \$ k create ns week11-2
- \$ k create sa clo835 -n week11-1
- \$ k create sa clo835 -n week11-1
- \$ k create sa clo835 -n week11-2
- \$ k create -f week11/workshop1/kubectl-proxy-pod.yaml -n week11-1
- \$ k create -f week11/workshop1/kubectl-proxy-pod.yaml -n week11-2
- # Open shell prompts in 2 different terminals
- \$ kubectl exec -it test -n week11-1 – sh
- \$ kubectl exec -it test -n week11-2 – sh

```
#!/bin/sh

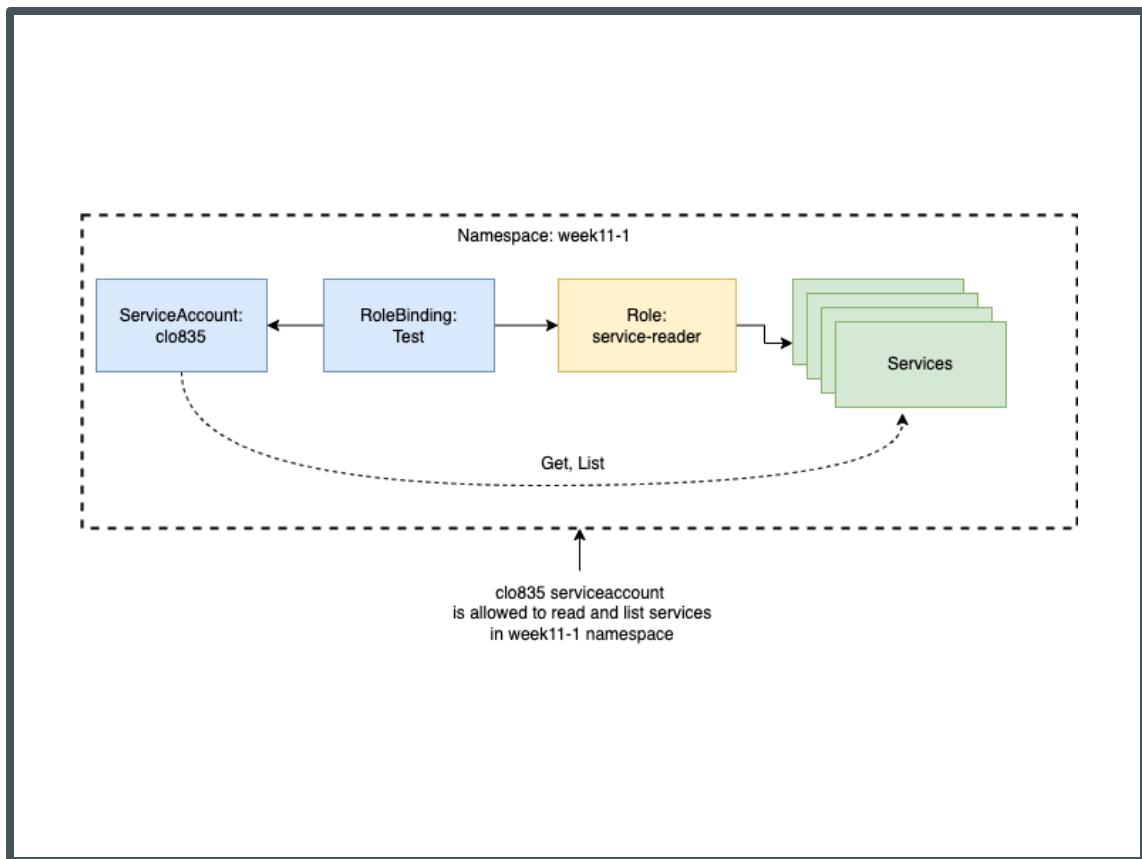
API_SERVER="https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT"
CA_CRT="/var/run/secrets/kubernetes.io/serviceaccount/ca.crt"
TOKEN="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)"

/kubectl proxy --server="$API_SERVER" --certificate-authority="$CA_CRT" --token="$TOKEN" --accept-paths='^.*'
```

CONNECT TO K8S API USING DEFAULT SA

- \$ kubectl exec -it test -n week11-2 – sh
- # Send a request to K8s API server to list all the services in the namespace from the pod
- # The pod is authenticated as \"system:serviceaccount:week11-1:default" but the request is denied
- / # curl localhost:8001/api/v1/namespaces/week11-1/services
- # List all the verbs available for services resource
- \$ k api-resources --sort-by name -o wide | grep services
- # We will create a service-reader role that authorizes clo835 read access to services
- \$ k create -f service-reader.yaml -n week11-1
- \$ k create -f service-reader.yaml -n week11-2
- # Creating the role is not enough, we need to create RoleBinding to associate our SA with the Role
- \$ k create rolebinding test --role=service-reader --serviceaccount=week11-1:clo835 -n week11-1

VERIFYING THE FUNCTIONALITY



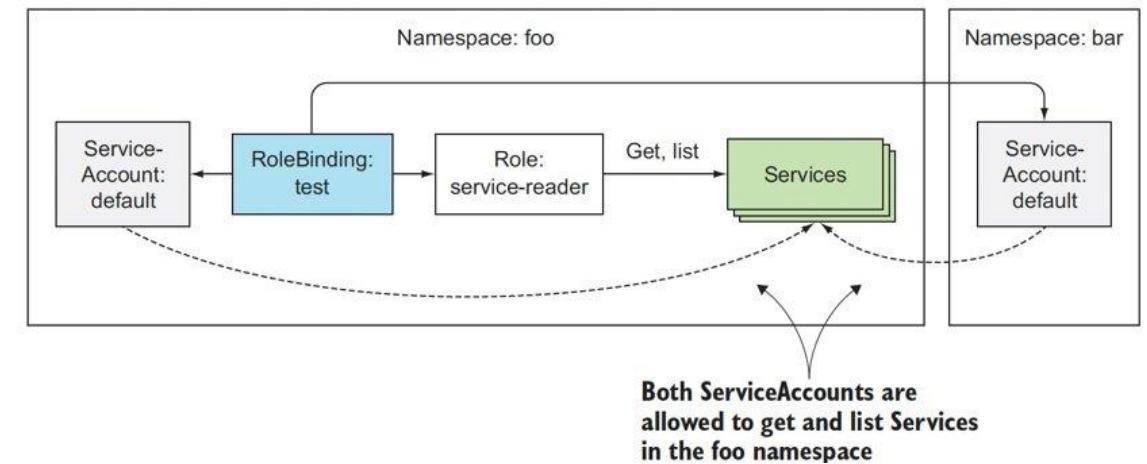
- `$ k get rolebinding test -n week11-1 -o yaml`
- `$ kubectl exec -it test -n week11-1 – sh`
- `/ # curl localhost:8001/api/v1/namespaces/week11-1/services`
- `/ # curl localhost:8001/api/v1/namespaces/week11-2/services`

GRANT SA IN ANOTHER NAMESPACE ACCESS TO THE SERVICES IN THE WEEK11-1 NAMESPACE

- # Clo835 in week11-2 namespace has no access to services in either of the namespaces.
- # We will grant it access to services in week11-1 namespace
- \$ k create sa clo835 -n week11-2
- # Add clo835 serviceaccount defined in week11-2 namespace to the "subjects" section
- \$ k edit rolebinding test -n week11-1
- # Deploy the pod to week11-2 namespace
- k create -f kubectl-proxy-pod.yaml -n week11-2

subjects:

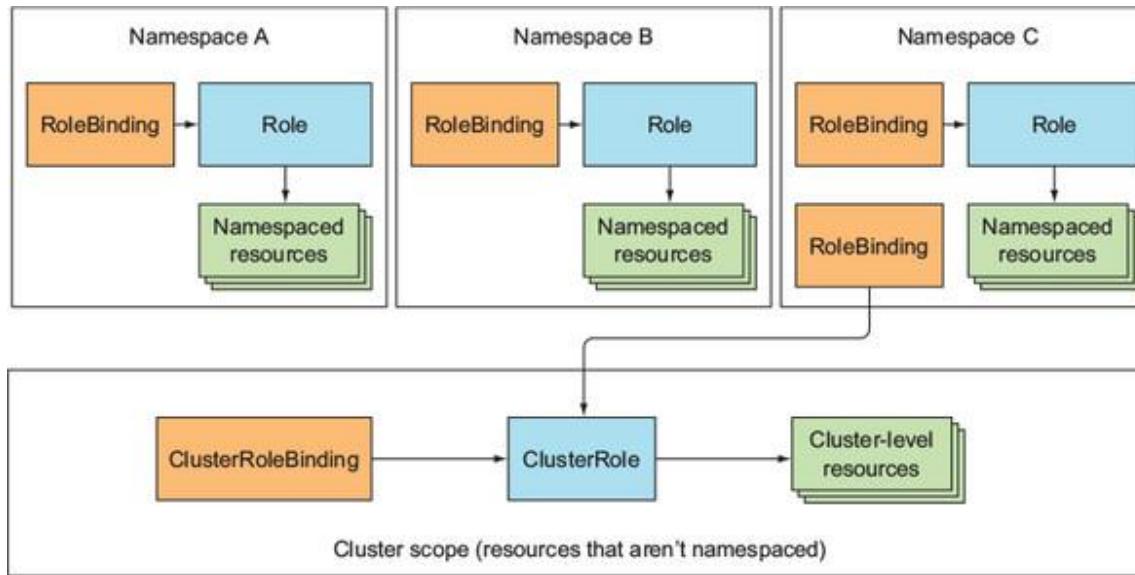
```
- kind: ServiceAccount
  name: clo835
  namespace: week11-1
- kind: ServiceAccount
  name: clo835
  namespace: week11-2
```



VERIFYING THE FUNCTIONALITY

- # Attache to the test pod in week11-2 namespace
- \$ kubectl exec -it test -n week11-2 – sh
- # SA clo835 can list services in **week11-1 namespace**
- / # curl localhost:8001/api/v1/namespaces/week11-1/services
- # SA clo835 cannot list services in **week11-2 namespace**
- / # curl localhost:8001/api/v1/namespaces/week11-2/services

CLUSTERROLES AND CLUSTERROLEBINDINGS



- # Some K8s resources are not namespace-specific.
- # Some APIs such as /healthz are not related to K8s resources
- # We might need cluster-level access, for example, for cluster admins
- # The above cases are covered by ClusterRoles and ClusterRoleBinding
- # A ClusterRole is a cluster-level resource for allowing access to non-namespaced resources or non-resource URLs or used as a common role to be bound inside individual namespaces, saving you from having to redefine the same role in each of them.
- \$ kubectl api-resources --sort-by name -o wide | grep nodes

USE CLUSTERROLES TO ACCESS PERSISTENTVOLUMES

- # Grant our test pod access to PersistentVolumes
- \$ k create clusterrole pv-reader --verb=get,list --resource=persistentvolumes
- \$ k get clusterrole pv-reader -o yaml
- \$ kubectl exec -it test -n week11-2 – sh
- # By default, our pod does not have permissions to query PersistentVolumes
- / # curl localhost:8001/api/v1/persistentvolumes
- # Create RoleBinding to the reader-pv ClusterRole
- \$ kubectl create rolebinding pv-test --clusterrole=pv-reader --serviceaccount=week11-1:clo835 -n week11-1

```
/ # curl localhost:8001/api/v1/persistentvolumes
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "persistentvolumes is forbidden: User \\"system:serviceaccount:week11-2:clo835\\" is not authorized to list resource \"persistentvolumes\" in API group \\"\\\" at the cluster scope",
  "reason": "Forbidden",
  "details": {
    "kind": "persistentvolumes"
  },
  "code": 403
}/ # █
```

USE CLUSTERROLES TO ACCESS PERSISTENTVOLUMES

- # Can we query PersistentVolumes now?
- \$ kubectl exec -it test -n week11-1 -- sh
- / # curl localhost:8001/api/v1/persistentvolumes
- # We still do not have access to persistent volumes!
- # To grant access to cluster-level resources, we must always use a ClusterRoleBinding.
- \$ k delete rolebindings pv-test -n week11-1
- \$ k create clusterrolebinding pv-test --clusterrole=pv-reader --serviceaccount=week11-1:clo835
- \$ kubectl exec -it test -n week11-1 – sh
- / # curl localhost:8001/api/v1/persistentvolumes
- # Now we can query PersistentVolumes!

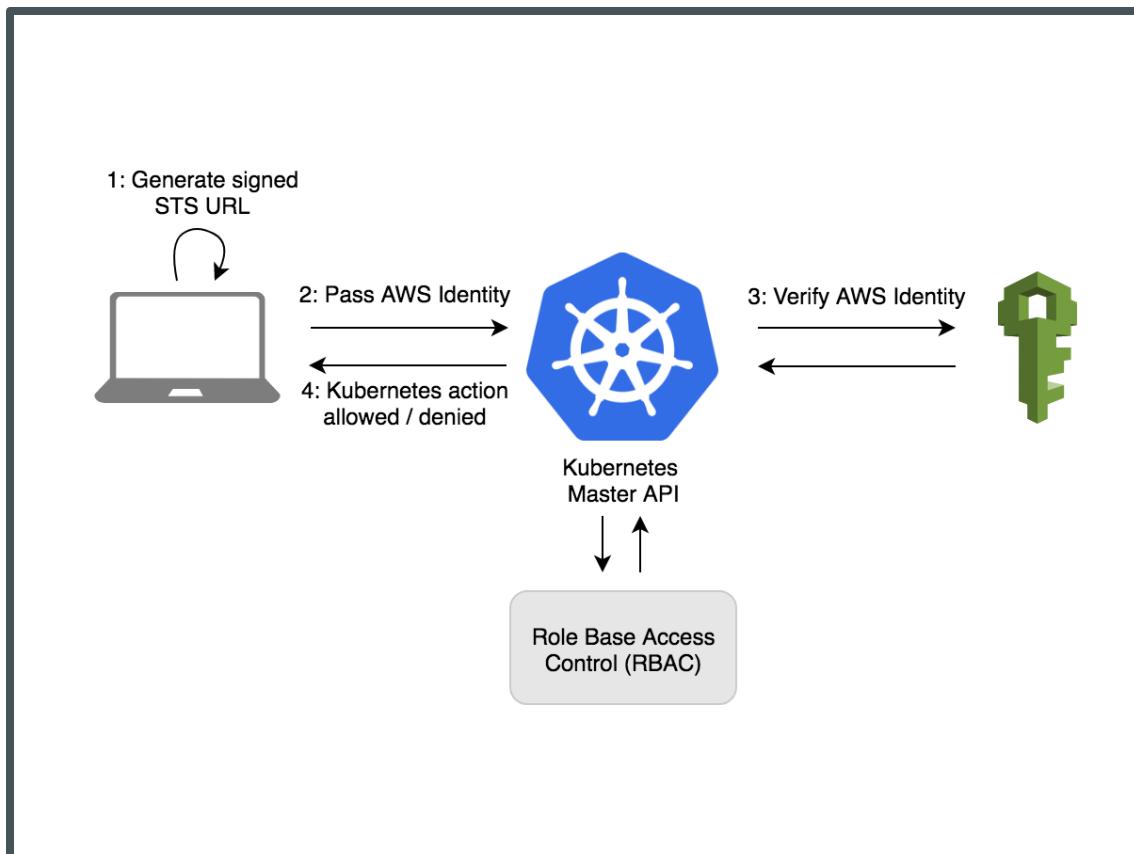
```
irinag:~/environment/week11 $ kubectl exec -it test -n week11-1 -- sh
/ # curl localhost:8001/api/v1/persistentvolumes
{
  "kind": "PersistentVolumeList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/persistentvolumes",
    "resourceVersion": "284079"
  },
  "items": []
}/ # █
```

EXPLORE PRE-CREATED CLUSTERROLES AND CLUSTERROLEBINDINGS

- # Allowing access to non-resource URLs
- \$ k get clusterrole system:discovery -o yaml
- # Examine the subjects of the RoleBinding
- \$ k get clusterrolebinding system:discovery -o yaml
- # List all the SAs, Roles and RoleBindings
- kubectl get rolebindings,clusterrolebindings \
--all-namespaces \
-o custom-
columns='KIND:kind,NAMESPACE:metadata.namespace,NAME:metadata.name,
SERVICE_ACCOUNTS:subjects[?(@.kind=="ServiceAccount")].name'

ClusterRoleBinding	<none>	system:controller:pv-protection-controller	pv-protection-controller
ClusterRoleBinding	<none>	system:controller:pvc-protection-controller	pvc-protection-controller
ClusterRoleBinding	<none>	system:controller:replicaset-controller	replicaset-controller
ClusterRoleBinding	<none>	system:controller:replication-controller	replication-controller
ClusterRoleBinding	<none>	system:controller:resourcequota-controller	resourcequota-controller
ClusterRoleBinding	<none>	system:controller:route-controller	route-controller
ClusterRoleBinding	<none>	system:controller:service-account-controller	service-account-controller
ClusterRoleBinding	<none>	system:controller:service-controller	service-controller
ClusterRoleBinding	<none>	system:controller:statefulset-controller	statefulset-controller
ClusterRoleBinding	<none>	system:controller:ttl-controller	ttl-controller

AMAZON EKS AUTHENTICATION FOR IAM USERS, GROUPS AND ROLES

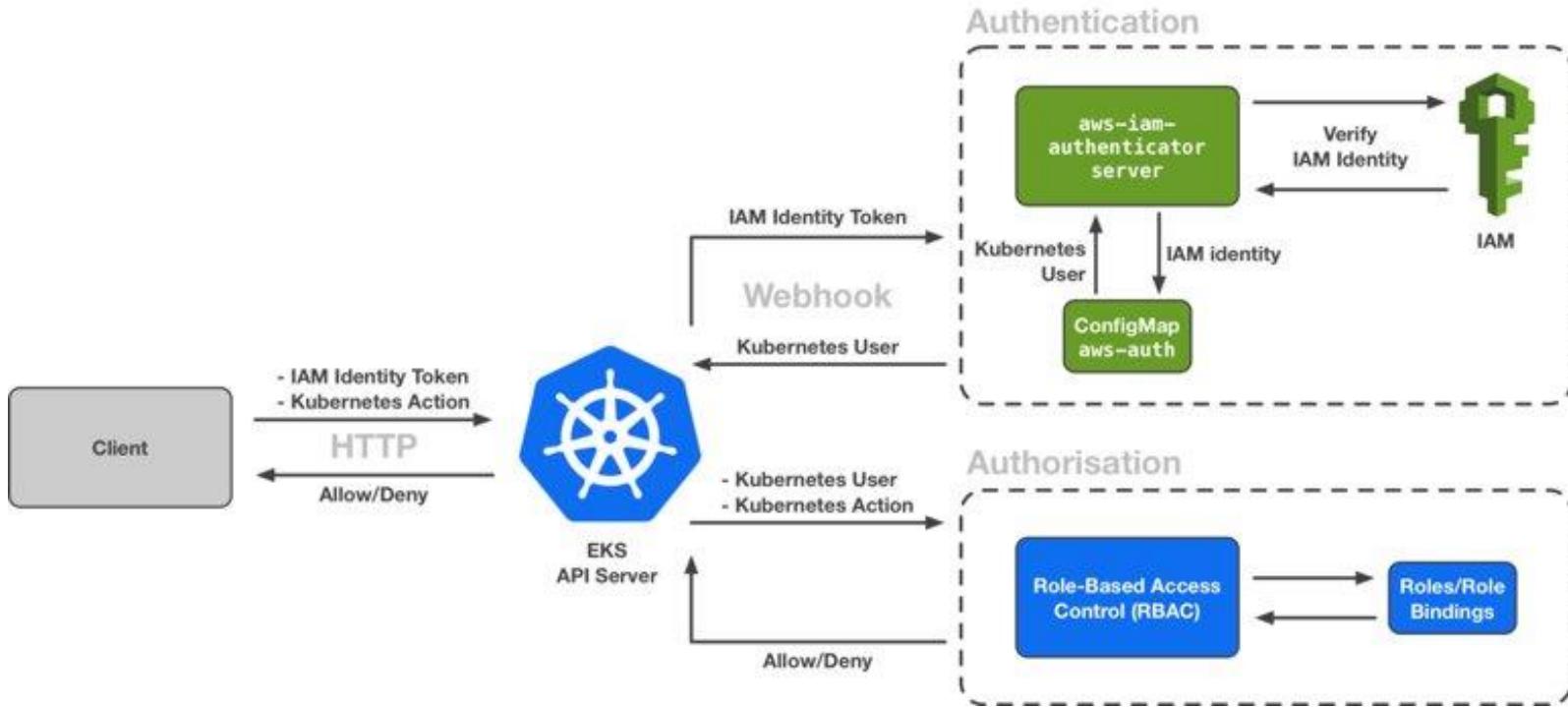


- Reminder – K8s does not support resource of type "user"
- User identities should be stored and verified externally by Identity providers (IdP, e.g., AD, OIDC provider or Amazon IAM)
- Externally stored identities are mapped to K8s users and groups
- Amazon EKS Control Plane is configured with:
 - **Authentication:** [webhook token authentication](#)
 - **Authorization:** [role-based access control \(RBAC\)](#)
- **Amazon IAM does not control user's data plane permissions in Amazon EKS;** permissions are controlled by K8s RBAC

MAPPING IAM IDENTITIES TO K8S USERS AND GROUPS

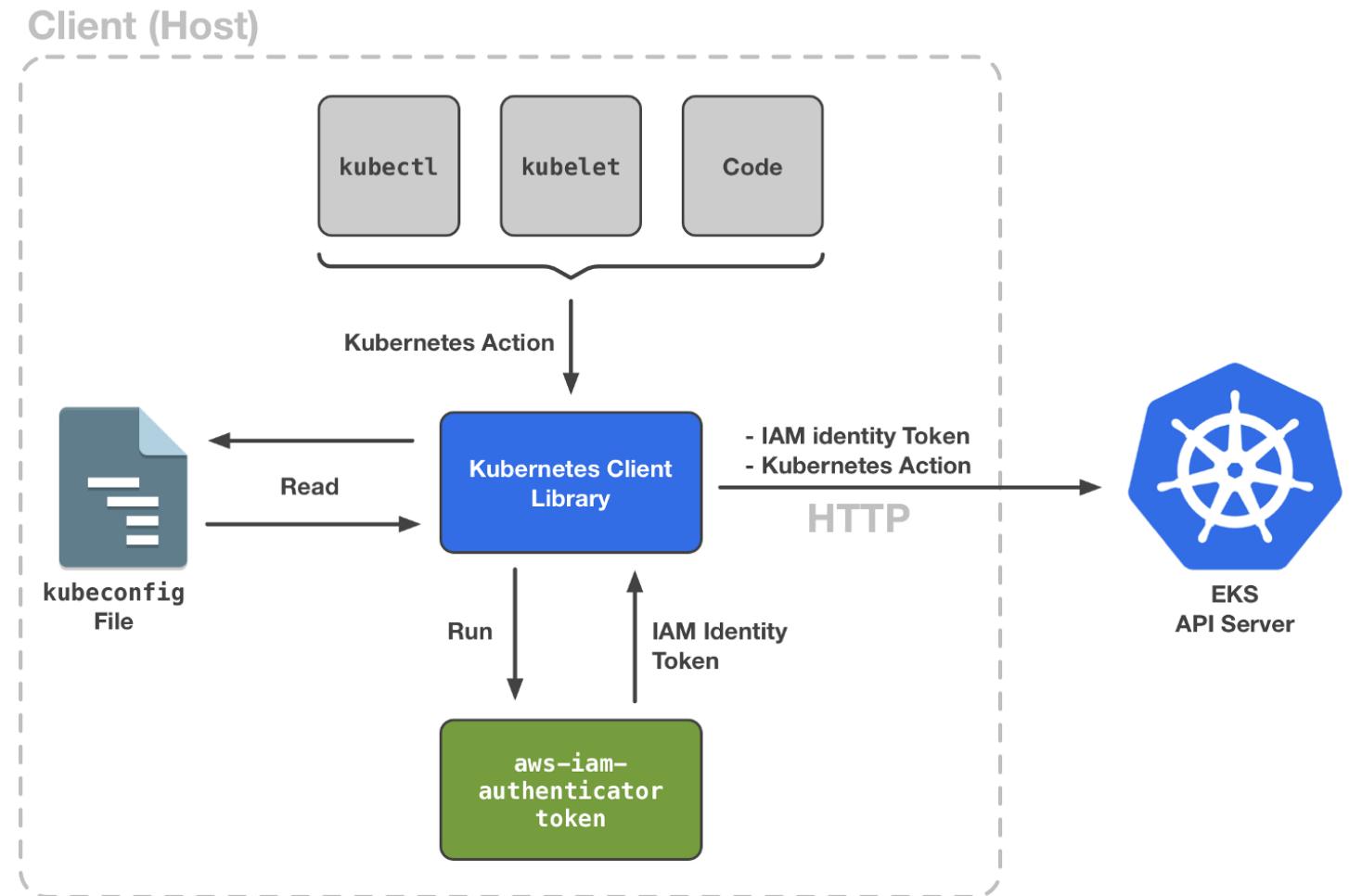
- Amazon EKS uses "aws-auth" ConfigMap in kube-system namespace to map IAM Users and Roles to K8s Users with the sections below
 - mapRoles**
 - mapUsers**
 - mapAccounts**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
data:
  mapRoles: |
    - rolearn: arn:aws:iam::1111111111:role/worker-node-instance-role
      username: system:node:{{EC2PrivateDNSName}}
      groups:
        - system:bootstrappers
        - system:nodes
```



ENABLE IAM USERS' ACCESS TO AMAZON EKS CLUSTER – SERVER SIDE

ENABLE IAM USERS ACCESS TO AMAZON EKS CLUSTER – CLIENT SIDE



WORKSHOP 2 – GRANT IAM USER K8S API PERMISSIONS

OUR GOAL

Create an IAM user called clo835-user who is authenticated to access the EKS cluster but is only authorized (via RBAC) to list, get, and watch pods and deployments in the rbac-test' namespace.

To achieve this, we'll create an IAM user, map that user to a kubernetes role, then perform kubernetes actions under that user's context.

CREATE THE AMAZON EKS CLUSTER AND CHECK AWS-AUTH CONFIGMAP SETTINGS

- # Create the cluster with the CloudWatch logs enabled for authenticator actions to be recorded
- \$ eksctl create cluster -f ..eks_config_week11.yaml
- # Check the aws-auth ConfigMap
- \$ k get configmap aws-auth -o yaml -n kube-system
- # Check the permissions of "cluster-admin" ClusterRole
- \$ k get clusterrolebinding cluster-admin -o yaml
- # Create rbac-test namespace and nginx deployment
- \$ k create ns rbac-test
- \$ k create deploy nginx --image=nginx -n rbac-test
- \$ k get all -n rbac-test

CREATE IAM USER AND EXPORT ITS CREDENTIALS

- # Create IAM User using AWS CLI
- \$ aws iam create-user --user-name clo835-user
- \$ aws iam create-access-key --user-name clo835-user | tee /tmp/create_output.json
- # Run a script that stores the credentials as a local file. It will help switching between our Admin user and clo835 user
- \$ cd workshop2 && ./creds.sh
- # Verify the content of clo835_creds.sh you created
- \$ cat clo835_creds.sh
- # Switch to clo835 IAM user permissions and verify that you are using clo835 IAM user permissions
- \$./clo835_creds.sh && aws sts get-caller-identity # mind the dot at the beginning of the command
- # Check clo835 access to K8s API. The user is unauthorized. **Why?**
- \$ k get nodes
- # Switch back to the Admin user and verify caller identity
- \$./switch_to_admin.sh && aws sts get-caller-identity # mind the dot at the beginning of the command

MAP IAM USER TO K8S USER IN AWS- AUTH CONFIGMAP. CHECK ACCESS TO K8S API

- # Add mapUsers block to auth-auth ConfigMap
- \$ k edit configmap aws-auth -n kube-system
- mapUsers: |
- - userarn: arn:aws:iam::<your account number>:user/clo835-user
- username: clo835-user-k8s
- # Still no access to the K8s resources, we are missing authorization component.
- # Let's add a role with deployment permissions and bind this role to our clo835-user-k8s. Please note – role binding is using K8s user, **not** AWS IAM user.

```
apiVersion: v1
data:
  mapRoles: |
    - groups:
        - system:bootstrappers
        - system:nodes
      rolearn: arn:aws:iam::047923129740:role/eksctl-clo835-nodegroup-nodegroup-NodeInstanceRole-1E0W8SH1FZ7FF
      username: system:node:{EC2PrivateDNSName}
  mapUsers: |
    - userarn: arn:aws:iam::047923129740:user/clo835-user
      username: clo835-user-k8s
kind: ConfigMap
```

GRANT DEPLOYMENT PERMISSIONS TO CL0835-USER-K8S WITH ROLE AND ROLEBINDING

```
1 kind: RoleBinding
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: read-pods
5   namespace: rbac-test
6 subjects:
7 - kind: User
8   name: cl0835-user-k8s
9   apiGroup: rbac.authorization.k8s.io
10 roleRef:
11   kind: Role
12   name: pod-reader
13   apiGroup: rbac.authorization.k8s.io
```

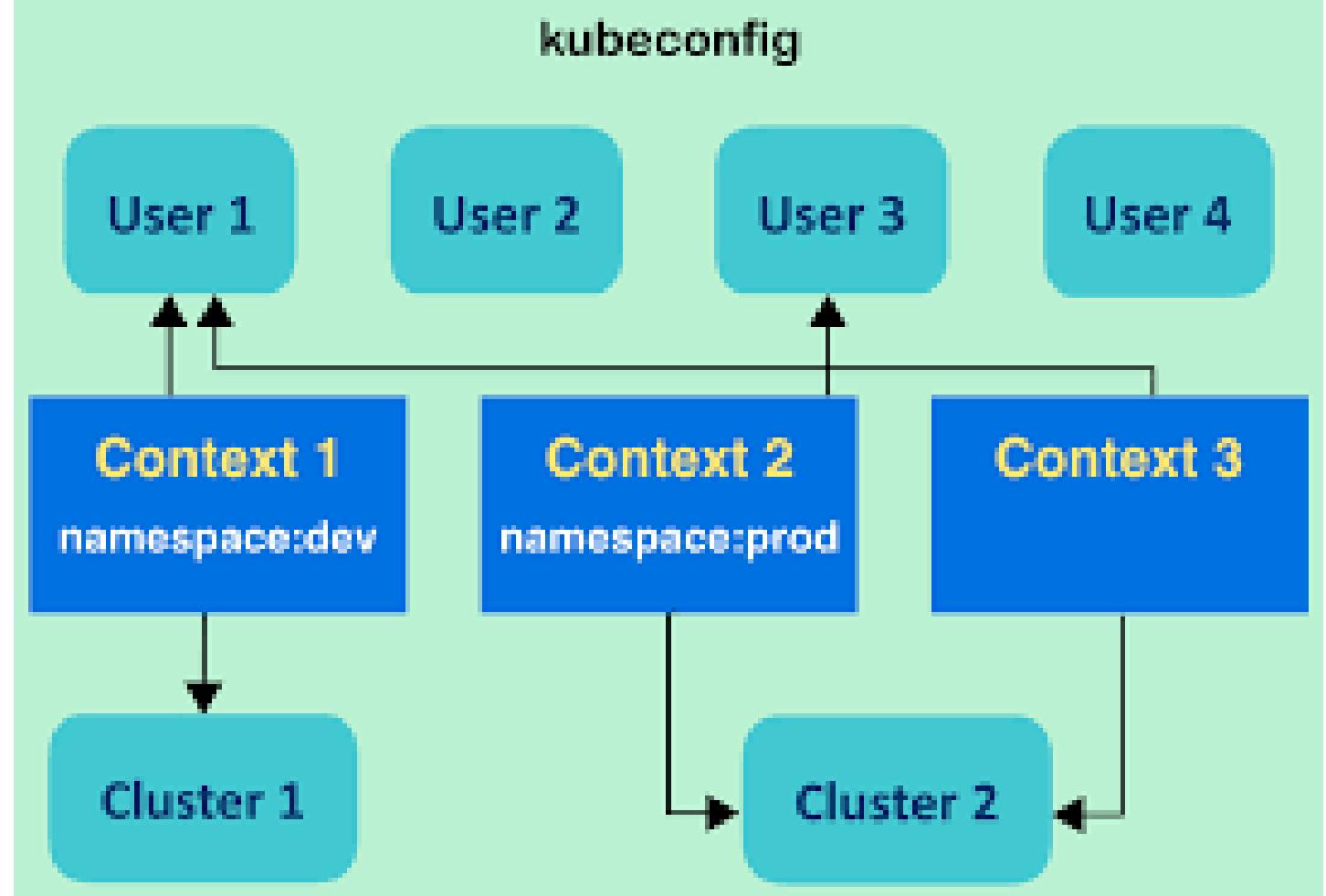
- # Switch to Admin role
- \$./switch_to_admin.sh && aws sts get-caller-identity
- Create a role that provides list, get, and watch access for pods and deployments in the rbac-test namespace only.
- \$ k create -f clo835-role.yaml
- # Bind the user and the Role together with the RoleBinding resource
- \$ k create -f clo835-role-binding.yaml
- # Switch to cl0835-user IAM user and check user's permissions in Amazon EKS
- \$. clo835_creds.sh; aws sts get-caller-identity
- # IAM user cl0835-user has permissions to list pods in rbac-test namespace
- \$ kubectl get pods -n rbac-test
- # IAM user cl0835-user does not have permissions to list pods in kube-system namespace
- \$ kubectl get pods -n kube-system

ADD IAM USER TO AWS-AUTH WITH EKSCTL

- # Editing aws-auth ConfigMap manually is not ideal. We can achieve the same result with eksctl tool
- # We will map our clo835-user to cluster-admin role through system:masters group
- # Delete the mapUser section from aws-auth ConfigMap
- \$ k edit configmap aws-auth -n kube-system
- eksctl create iamidentitymapping --cluster clo835 --region=us-east-1 --arn arn:aws:iam::<AWS_ACCOUNT_ID>:user/clo835-user --group system:masters --username clo835admins
- # Now clo835-user is part of the system:masters group which is Admins group in K8s. Let's check user's permissions.
- \$. clo835_creds.sh; aws sts get-caller-identity
- \$ k get pods -n kube-system

```
ion: v1
  roles: | 
    groups:
      - system:bootstrappers
      - system:nodes
    olearn: arn:aws:iam::047923129740:role/eksctl
    sername: system:node:{EC2PrivateDNSName}
    ers: | 
      groups:
        - system:masters
    serarn: arn:aws:iam::047923129740:user/clo835user
    sername: clo835admins
    onfigMap
```

UNDERSTANDING KUBECONFIG – WORKING WITH MULTIPLE CLUSTERS, USERS AND NAMESPACES



CONFIGURING LOCAL ENVIRONMENT – WORKING WITH KUBECONFIG

- # Examine `~/.kube/config`
- # Let's check if our clo835-user can update the `~/.kube/config`
- `$. clo835_creds.sh; aws sts get-caller-identity`
- `$ aws eks update-kubeconfig --name clo835`
- # IAM user does not have any permissions to work with the EKS control plane. Let's grant the permissions
- # Create the policy with access to your EKS cluster; **Update** your account id in the JSON document and in the commands below
- `$. ./switch_to_admin.sh && aws sts get-caller-identity`
- `$ aws iam create-policy --policy-name EKSCL0835 --policy-document file://iam_policy.json`
- `$ aws iam attach-user-policy --policy-arn arn:aws:iam::<ACCOUNT-ID>:policy/EKSCL0835 --user-name clo835-user`
- `$. clo835_creds.sh; aws sts get-caller-identity && aws eks update-kubeconfig --name clo835`
- # Examine `~/.kube/config`

REFERENCES

- <https://dev.to/aws-builders/eks-auth-deep-dive-4fib>
- https://www.eksworkshop.com/beginner/090_rbac/
- <https://docs.aws.amazon.com/eks/latest/userguide/ci-luster-auth.html>
- <https://itnext.io/how-does-client-authentication-work-on-amazon-eks-c4f2b90d943b>
- <https://www.cncf.io/blog/2020/07/31/kubernetes-rbac-101-authentication/>
- https://www.eksworkshop.com/beginner/110_irsa/
- <https://aws.amazon.com/blogsopensource/introducing-fine-grained-iam-roles-service-accounts/>
- <https://faun.pub/how-to-access-aws-services-from-eks-ab5fa003a1b6>

CLEANUP



Delete the cluster

```
$ eksctl delete cluster --name clo835
```

Stop cloud9 instance