15-319 / 15-619 Cloud Computing

Course Overview 4

September 21st, 2021

Reflection of Last Week

- Conceptual content on OLI
  - Modules 3, 4 and Quiz 2
- Project theme - **Horizontal Scaling and Advanced Resource Scaling**
  - **AWS Horizontal Scaling**
    - Launch cloud resources via the AWS APIs (EC2)
    - Horizontally scale instances to reach a target RPS
  - **AWS Autoscaling**
    - Launch cloud resources via the AWS APIs (ALB / ASG…)
    - Design autoscaling policies to achieve RPS targets within instance hour limits
    - Handle instance failures
  - **AWS Autoscaling with Terraform**
    - Develop a Terraform template to launch cloud resources
    - Contrast infrastructure as code (IaC) and cloud APIs

This Week

- **Quiz 3 (OLI Modules 5, 6)**
  - Due on **Friday**, September 24th, 2021, 11:59PM ET
- **Project 1 Discussion**
  - Due on Sunday, September 26th, 2021, 11:59PM ET
- **Project 2**
  - Due on next **Sunday**, October 1st, 2021, 11:59PM ET
- **Primers released this week**
  - HBase Basics
  - MongoDB Primer
  - MySQL Primer
  - NoSQL Primer
  - Profiling a Cloud Service
  - Storage I/O benchmarking
  - Introduction to Apache Spark
  - Introduction to Scala
  - Zeppelin for Apache Spark
  - Online Programming Exercises
  - Introduction to consistency models[Optional]
  - Introduction to multithreaded programming in Java[Optional]

OLI Module 5 - Cloud Management

Cloud Software stack - enables provisioning, monitoring and metering of virtual user “resources” on top of the Cloud Service Provider’s (CSP) infrastructure.

- Cloud middleware
- Provisioning
- Metering
- Orchestration and automation
- Case Study: Openstack - Open-source cloud stack implementation

OLI Module 6 - Cloud Software Deployment Considerations

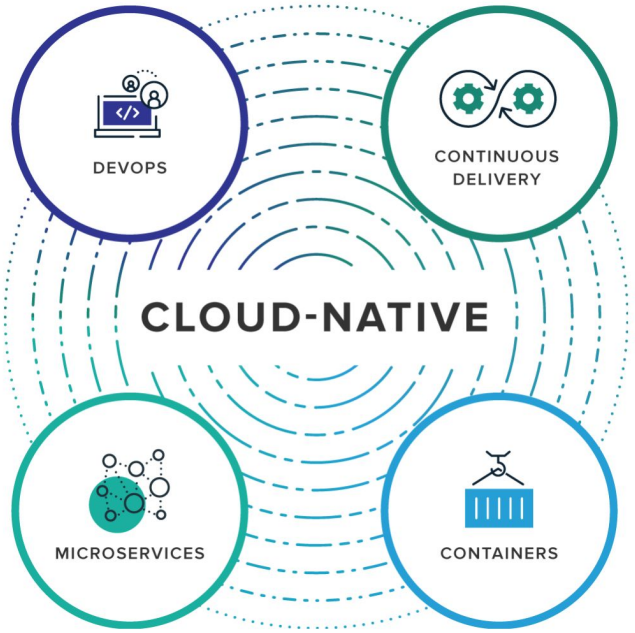
- Programming on the cloud
- Deploying applications on the cloud
  - Build fault-tolerant cloud services
  - Load balancing
  - Scaling resources
  - Dealing with tail latency
  - Economics for cloud applications

Cloud Native

- Cloud is about where we’re computing.
- Cloud-native is about how we’re computing.

Cloud & Cloud Native

Cloud-native technologies are used to describe applications built with services packaged in **containers**, deployed as **microservices** and managed on **elastic infrastructure** through agile **DevOps** processes and **continuous delivery** workflows.



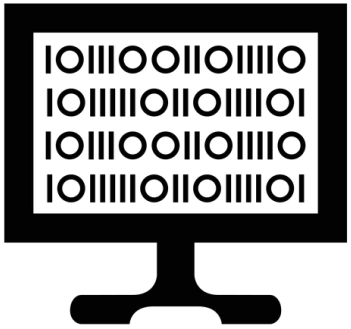
Cloud Native

- software is more stable than the infrastructure it runs on.
- software is designed to anticipate failure.
- software remains stable even when the infrastructure it is running on experiences outages or changes.
- software is scalable by design.
- software must operate in a constantly changing environment.
- Software is constantly changing.

Cloud–Native Applications: Platforms

- New platforms emerged, offering common services (features) that make it easier to develop cloud-native applications
  - Auto-scaling, replication, load balancing, health monitoring, service discovery, application-level routing, programmability
  - Commonly referred as “Platform as a Service” (PaaS)

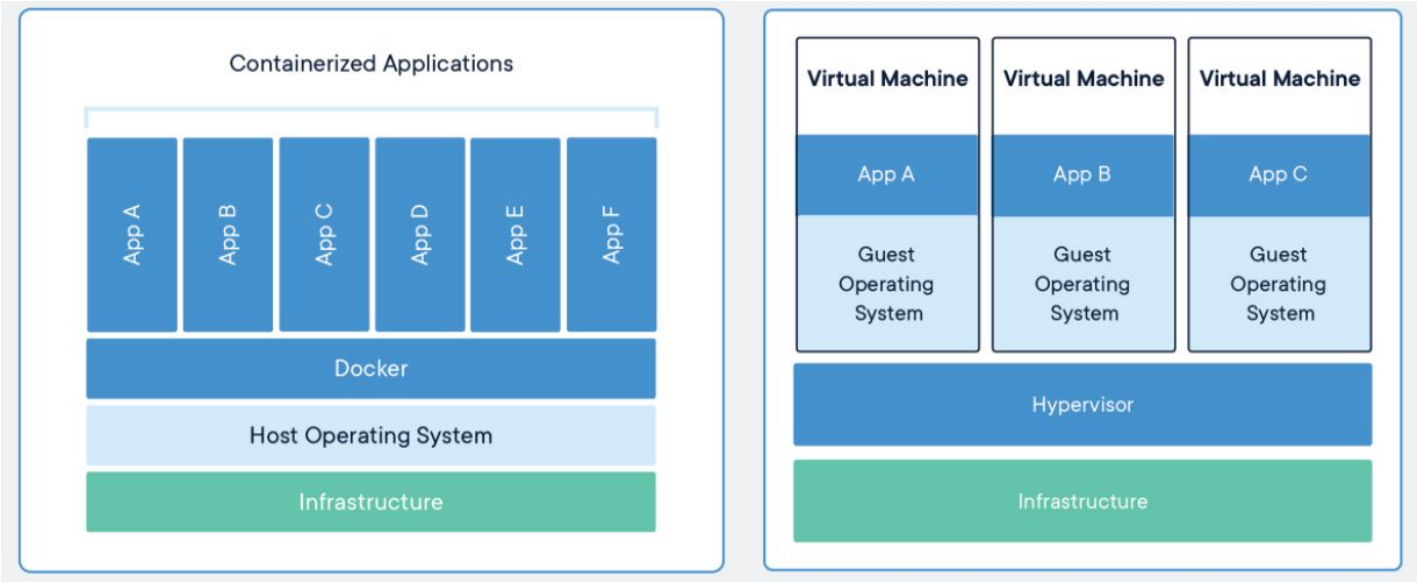
Project 2
Containers : Docker and Kubernetes



- Building your own container-based microservices
- Docker containers
- Manage multiple Kubernetes Cluster
- Multi Cloud deployments

Containers

- Provides OS-level virtualization.
- Provides private namespace, network interface and IP address, etc.
- A big difference with VMs is that containers share the host system's kernel

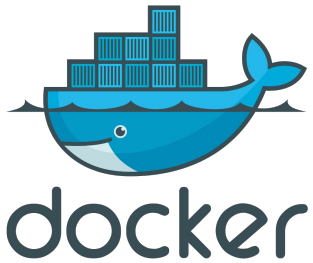


Why Containers?

- Faster deployment
- Portable
- Modularity
- Consistent Environment

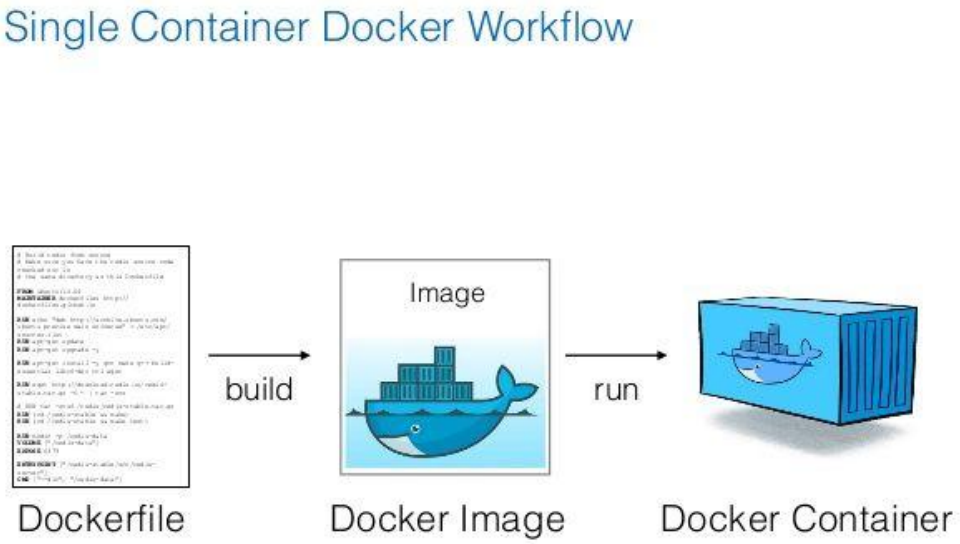
Build once, run anywhere

Docker

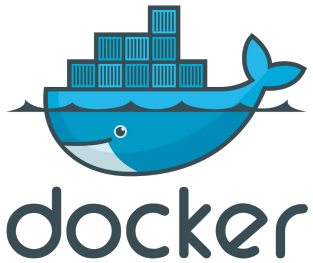


- Docker is an open platform for developing, shipping, and running applications.

- Dockerfile
- Docker Image
- Docker Container



Dockerfile



- Dockerfile tells Docker how to build an **image**:
  - Base Image
  - Commands
  - Files
  - Ports
  - Startup Command
- In short, a Dockerfile is a recipe for Docker images

Let’s go through a sample Dockerfile!

Example Dockerfile

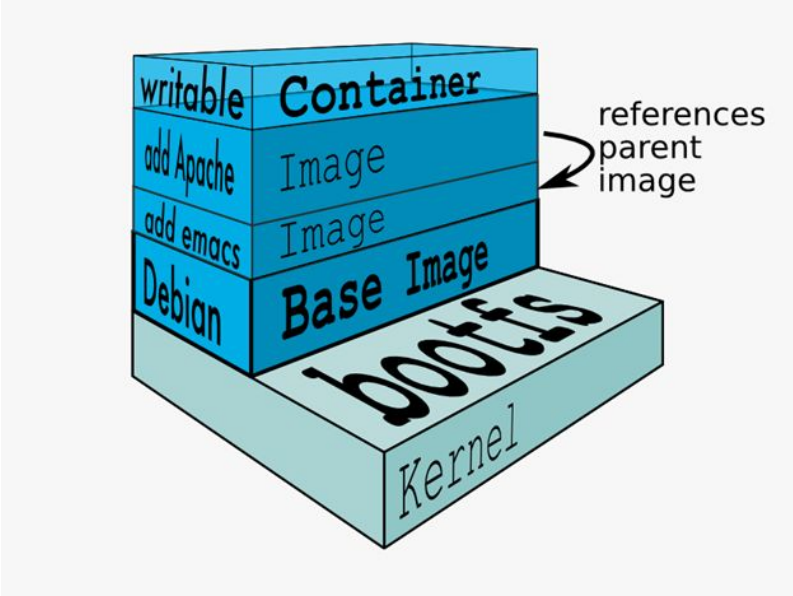
```
# Debian as the base image
FROM debian:latest

# Install additional packages
RUN apk add --update emacs
RUN apk add --update apache

# index.html must be in the current directory
ADD index.html /home/demo/

# Define the command which runs when the container starts
CMD ["cat /home/demo/index.html"]

# Use bash as the container's entry point. CMD is the argument to this entry
point
ENTRYPOINT ["/bin/bash", "-c"]
```



Example Dockerfile

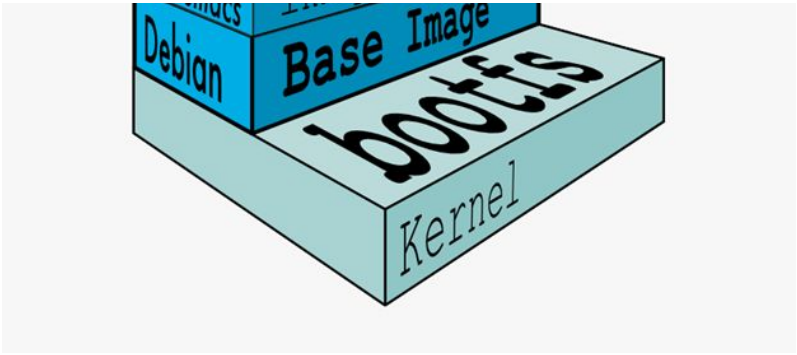
```
# Debian Linux as the base image
FROM debian:latest

# Install additional packages
RUN apk add --update emacs
RUN apk add --update apache

# index.html must be in the current directory
ADD index.html /home/demo/

# Define the command which runs when the container starts
CMD ["cat /home/demo/index.html"]

# Use bash as the container's entry point. CMD is the argument to this entry
point
ENTRYPOINT ["/bin/bash", "-c"]
```



Example Dockerfile

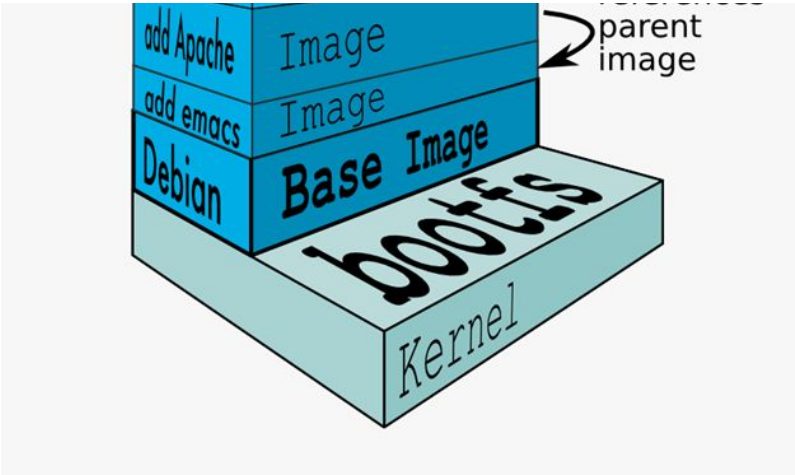
```
# Alpine Linux as the base image
FROM debian:latest

# Install additional packages
RUN apk add --update emacs
RUN apk add --update apache

# index.html must be in the current directory
ADD index.html /home/demo/

# Define the command which runs when the container starts
CMD ["cat /home/demo/index.html"]

# Use bash as the container's entry point. CMD is the argument to this entry
point
ENTRYPOINT ["/bin/bash", "-c"]
```



Example Dockerfile

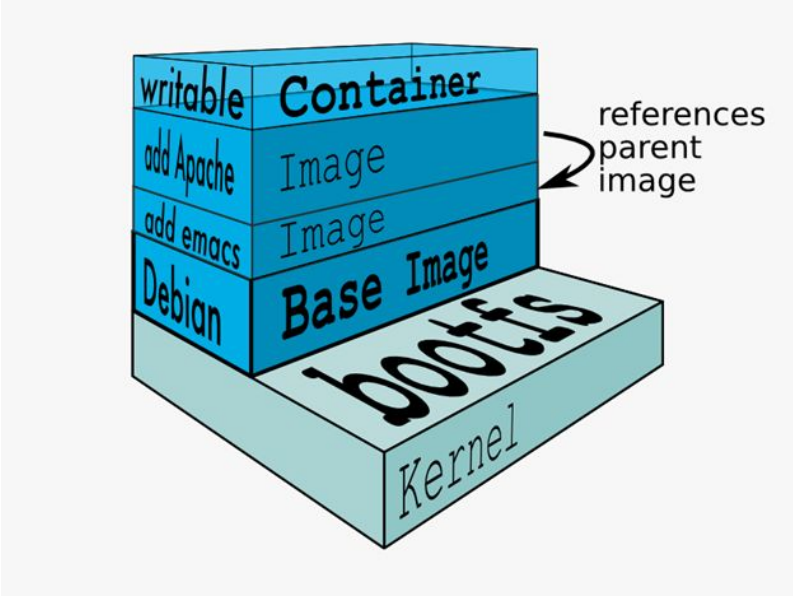
```
# Alpine Linux as the base image
FROM debian:latest

# Install additional packages
RUN apk add --update emacs
RUN apk add --update apache

# index.html must be in the current directory
ADD index.html /home/demo/

# Define the command which runs when the container starts
CMD ["cat /home/demo/index.html"]

# Use bash as the container's entry point. CMD is the argument to this entry point
ENTRYPOINT ["/bin/bash", "-c"]
```

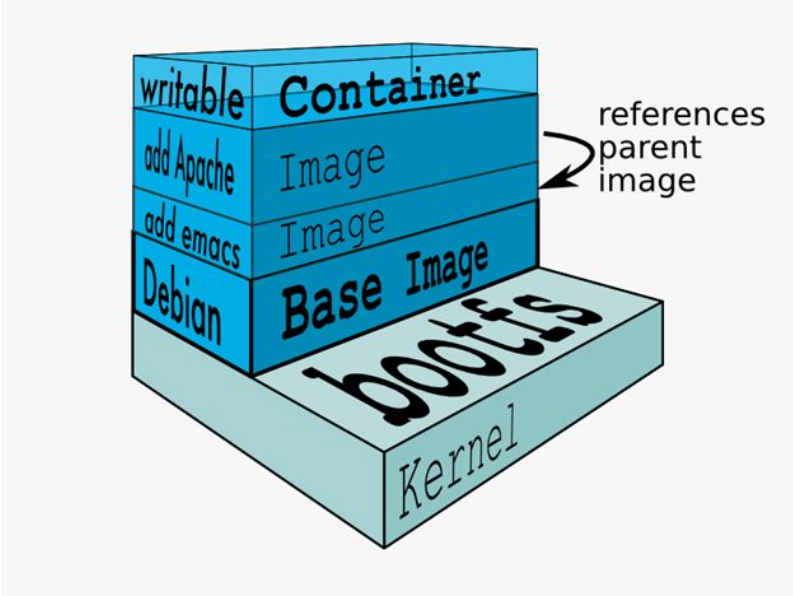


Example Dockerfile

```
# Alpine Linux as the base image
FROM debian:latest

# Install additional packages
RUN apk add --update emacs
RUN apk add --update apache

# index.html must be in the current directory
ADD index.html /home/demo/
```



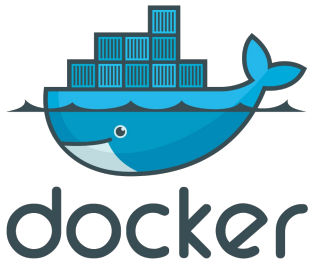

```
# Define the command which runs when the container starts
CMD ["cat /home/demo/index.html"]

# Use bash as the container's entry point. CMD is the argument to this entry point
ENTRYPOINT ["/bin/bash", "-c"]
```

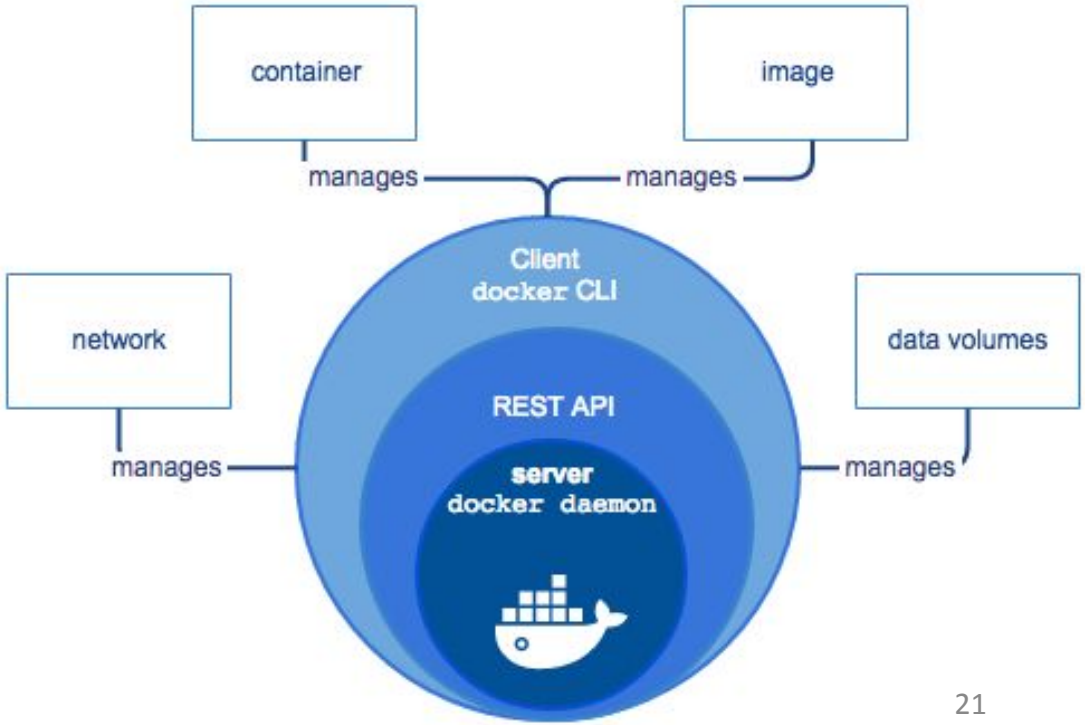
Images & Containers

- `docker build`
  - Builds an image
- `docker run`
  - Runs a container based on an image
- Images are the blueprints (Like a Class)
  - View these with `docker images`
- Containers are a ‘running instance of an Image’ (Like an Object)
  - View these with `docker ps`

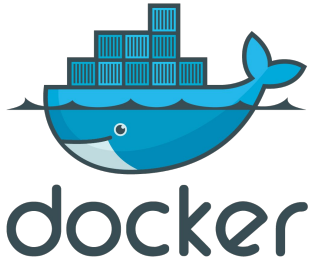
Docker Engine



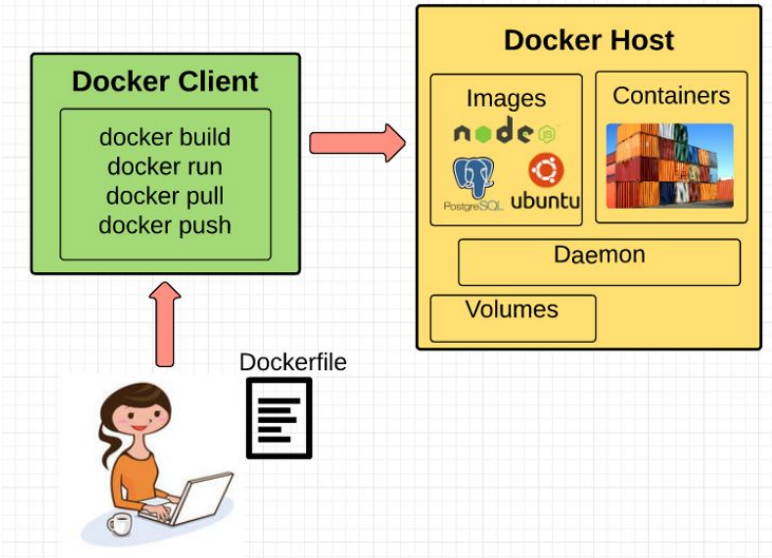
- A client-server application
  - Docker Daemon
  - Docker CLI
  - REST API



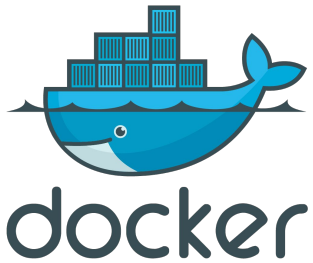
Docker Daemon



- Listens for Docker API requests
- Manages Docker objects
- The Daemon does not have to be on the same machine as the Client



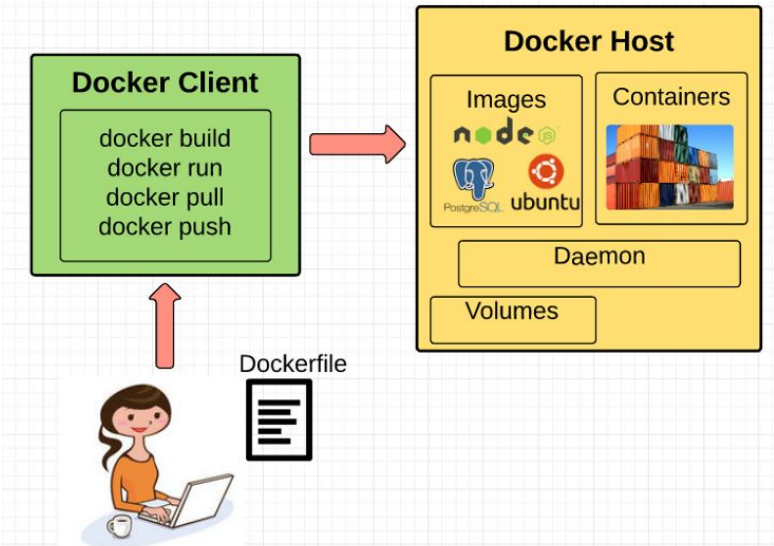
Docker CLI



- Communicates with Daemon using an API
- When you type:

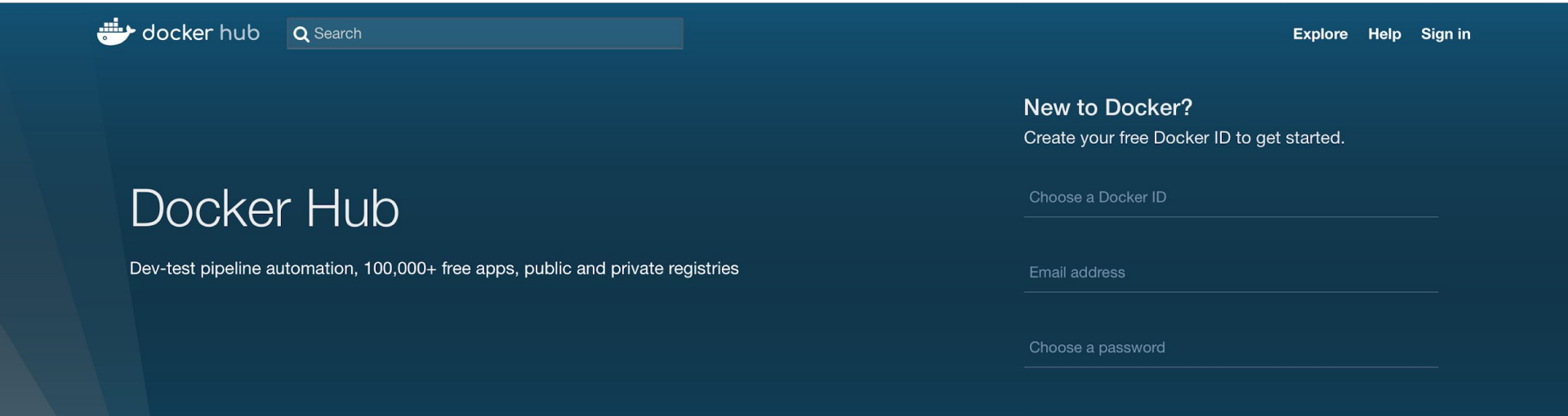
```
docker build nginx
```

You are telling the Docker client to forward the `build nginx` instruction to the Daemon



Docker Registries

- Store Docker images
- Examples
  - Docker Hub and Docker Cloud
  - GCP Container Registry
  - Azure Container Registry
- `docker pull`
- `docker push`



Containers are userful, but how to manage containers?

- Containers provide many benefits
  - Fast and lightweight
  - Sandboxed and consistent

- However, using containers introduces its own complexity, e.g.,
  - Load Balancing
  - Fault Tolerance

- How should we deploy, scale and manage containers efficiently?

Kubernetes

- Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers.
  - Horizontally Scalable
  - Self-Healing
  - Service Discovery
  - Automated Rollbacks
  - Utilization

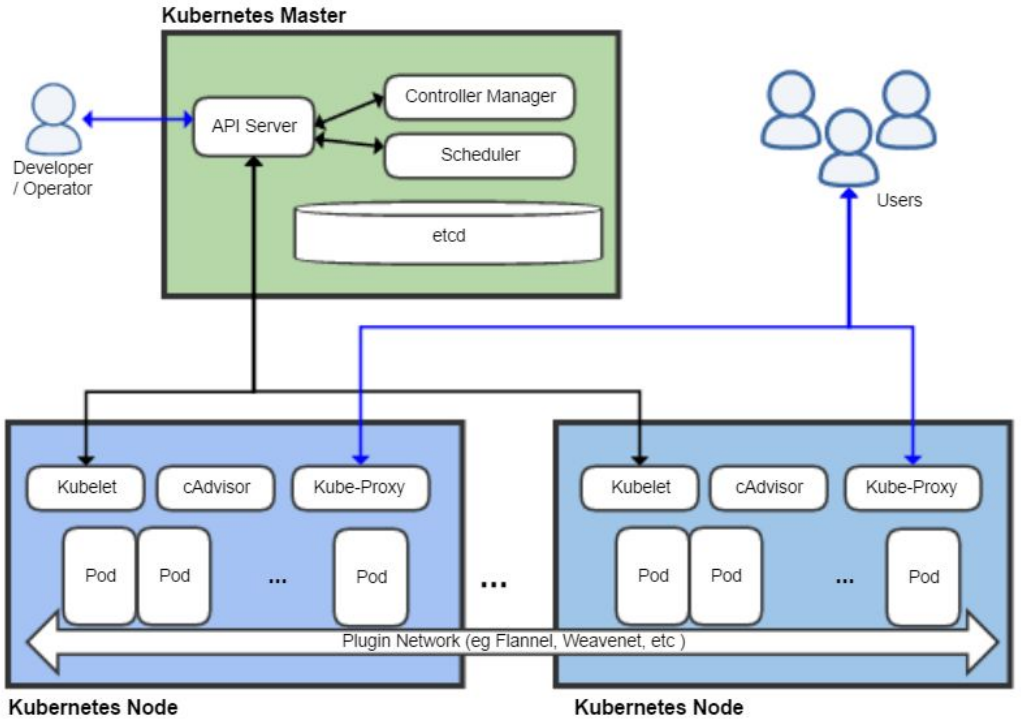
kubernetes

Kubernetes Overview

- API Objects
  - Pods - Collection of Containers
  - Deployment - Manages Pods
  - Service - Network Endpoint
- Desired State Management
  - YAML (YAML Ain’t a Markdown Language)
- Kubectl - CLI for Kubernetes
  - `kubectl create config.yaml`

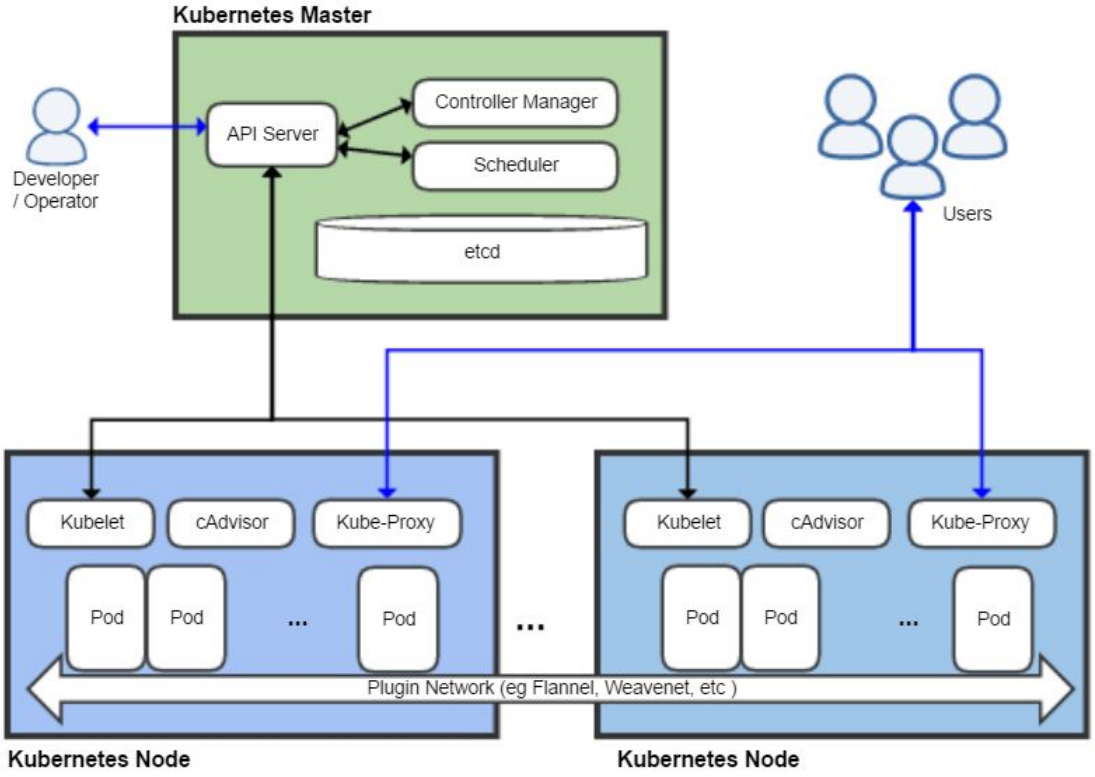
Kubernetes Cluster - Master

- Master Node
  - API Server
  - Controller Manager
  - Scheduler



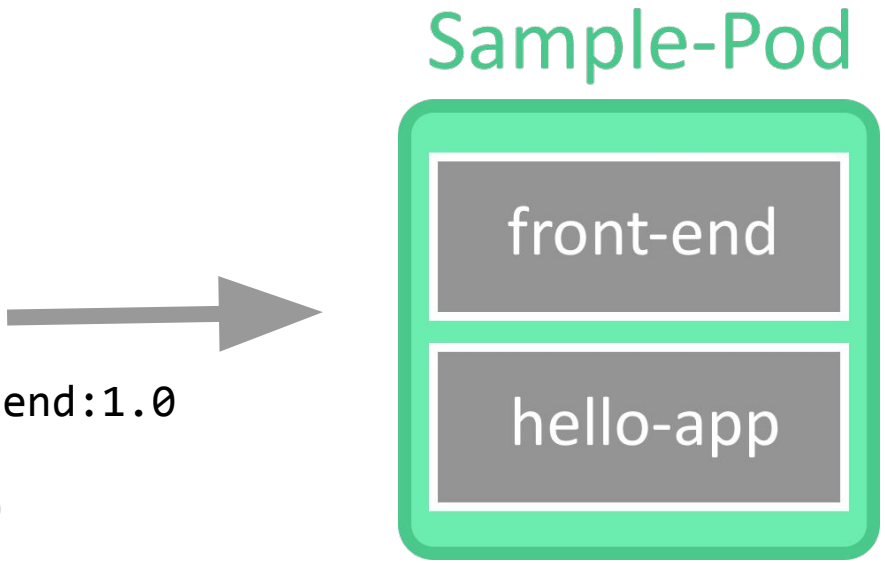
Kubernetes Cluster - Worker

- Worker Nodes
  - Kubelet Daemon
  - Kube-Proxy



Sample Kubernetes Config YAML

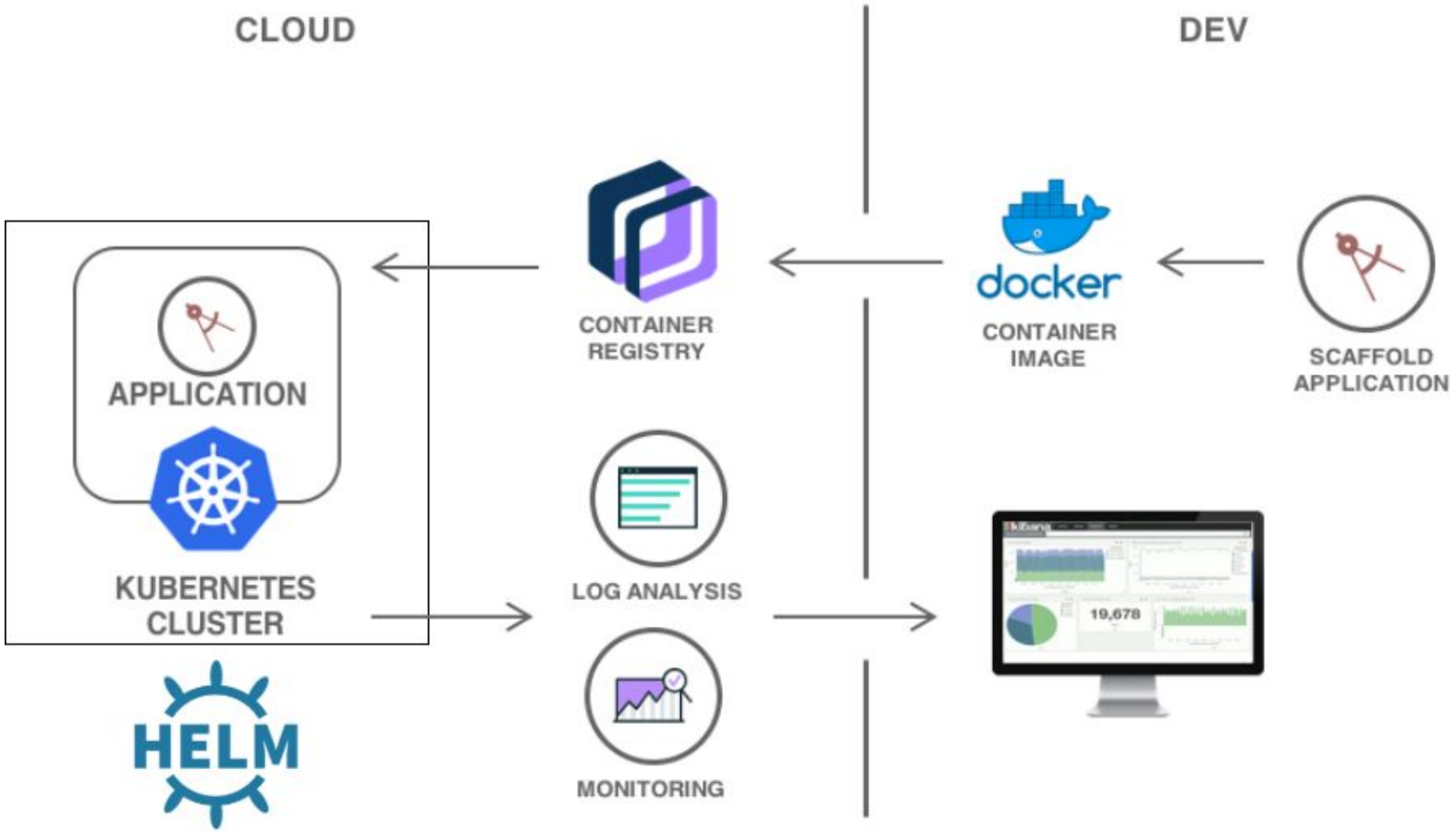
```
apiVersion: apps/v1beta1
kind: Pod
metadata:
 name: Sample-Pod
 labels:
   app: web
spec:
 containers:
   - name: front-end
     image:
gcr.io/samples/hello-frontend:1.0
     ports:
       - containerPort: 80
   - name: hello-app
     image:
gcr.io/samples/hello-app:1.0
     ports:
       - containerPort: 8080
```



Helm

- A tool for managing Kubernetes applications
- Helm Charts help you define, install, and upgrade complex Kubernetes application
- Chart structure:
  - **Chart.yaml**
    - A YAML file that contains chart information (name, version, description, etc.)
  - **Values.yaml**
    - The default configuration of this chart. The values listed in this file will be substituted in the files under the templates/ directory.
  - **templates/**
    - A directory of template files that will be combined with the values defined in Values.yaml. The files under this directory will be used to define all of the Kubernetes objects required to deploy the application.

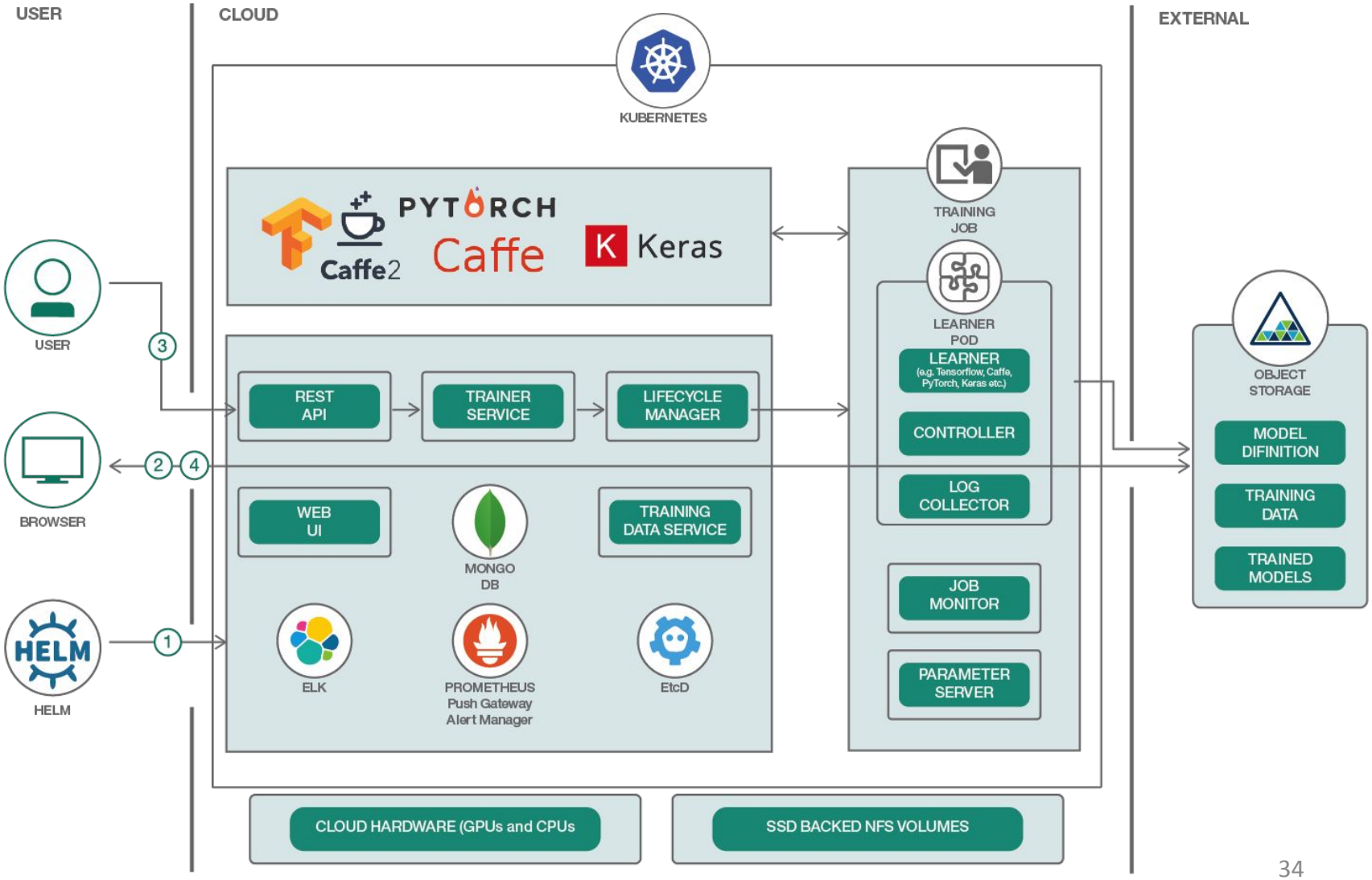
Docker, Kubernetes Workflow



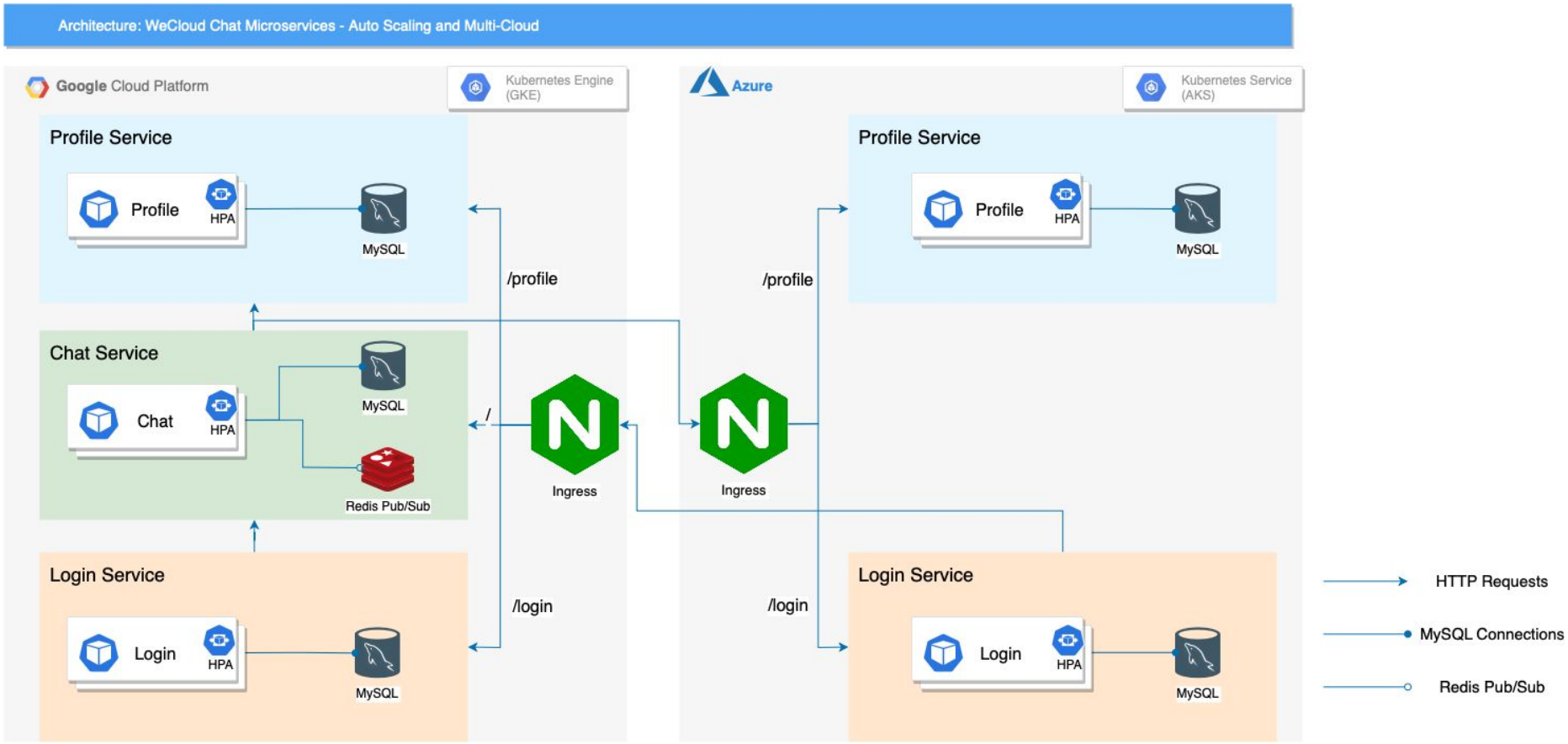
Microservice Architecture

- Loosely coupled applications, that generally communicate over a network and exist independently of each other.
- Why adopt a Microservice architecture?
  - Application Size
  - Scalability
  - Modifiability
  - Fault-tolerance

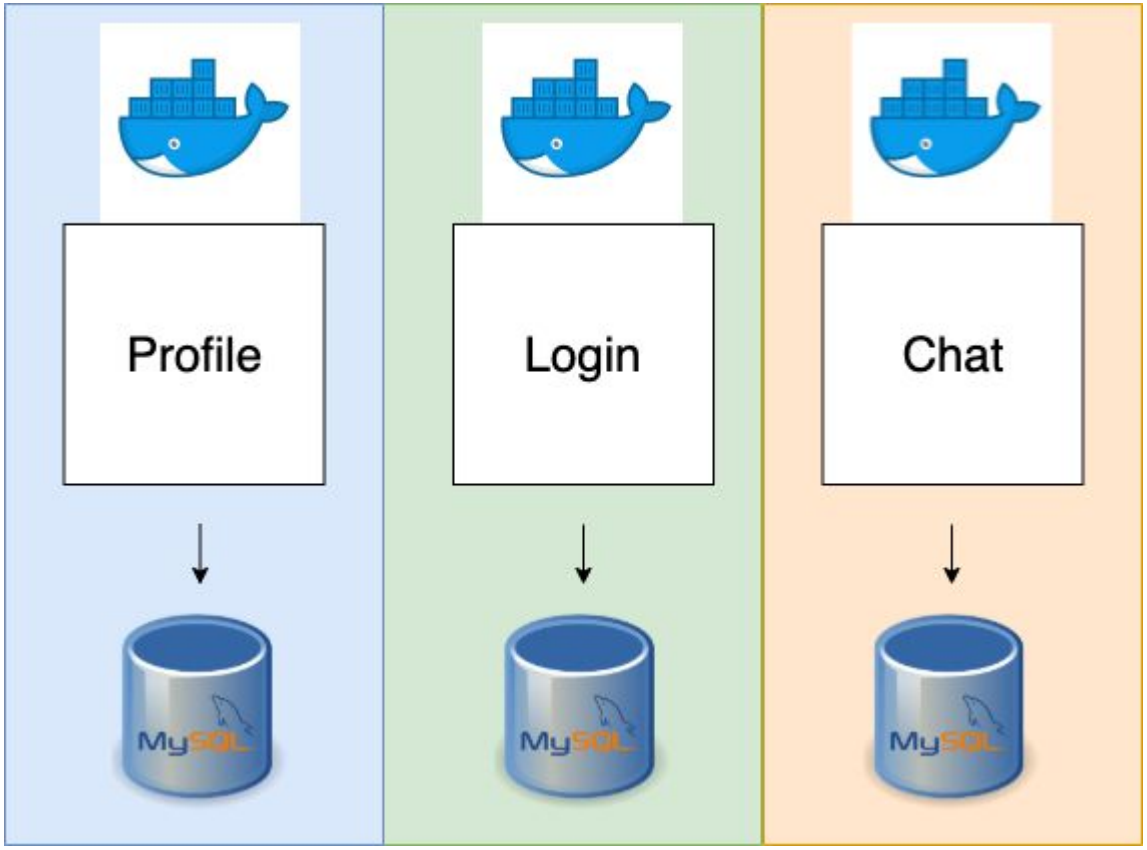
An Industrial Example



Project 2 - Containers: Docker & Kubernetes



Project 2 - Containers: Docker & Kubernetes



Project 2 - Containers & Kubernetes

- Build a chat room application using the microservice pattern
- Project overview:
  - Task 1: Containerize the profile service and run it locally
  - Task 2: Deploy the profile service to GKE
  - Task 3: Migrate the profile service's database from H2 to MySQL. Use Helm to manage the Kubernetes application.
  - Task 4: Install the chat service and login service using Helm charts. Connect the microservices to build an application.
  - Task 5: Replicate the profile and login services to AKS. Implement autoscaling rules to horizontally scale pods.
  - Task 6: DNS using Azure front door service

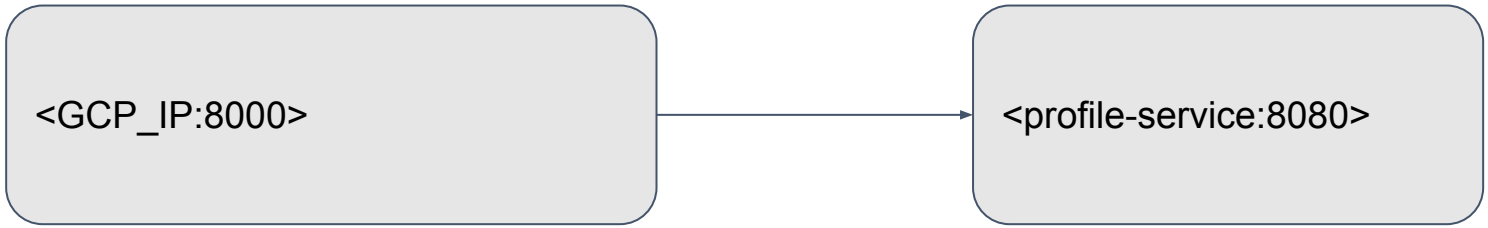
Task 1 - Containerize Profile Service

- Introduction to Dockerfiles
- Become familiar with the Docker CLI
  - `docker build`
  - `docker images`
  - `docker run`
  - `docker ps`
- Containerizing Java applications (a REST service)
- Consider the interactions between the host machine and the container
  - See the next slide

Task 1 - Containerize Profile Service

- Run a Docker container to host the profile service
  - The Profile service exposes port 8080 on the container
  - Port 8000 of VM is mapped to the container port

- How do we achieve this port mapping?

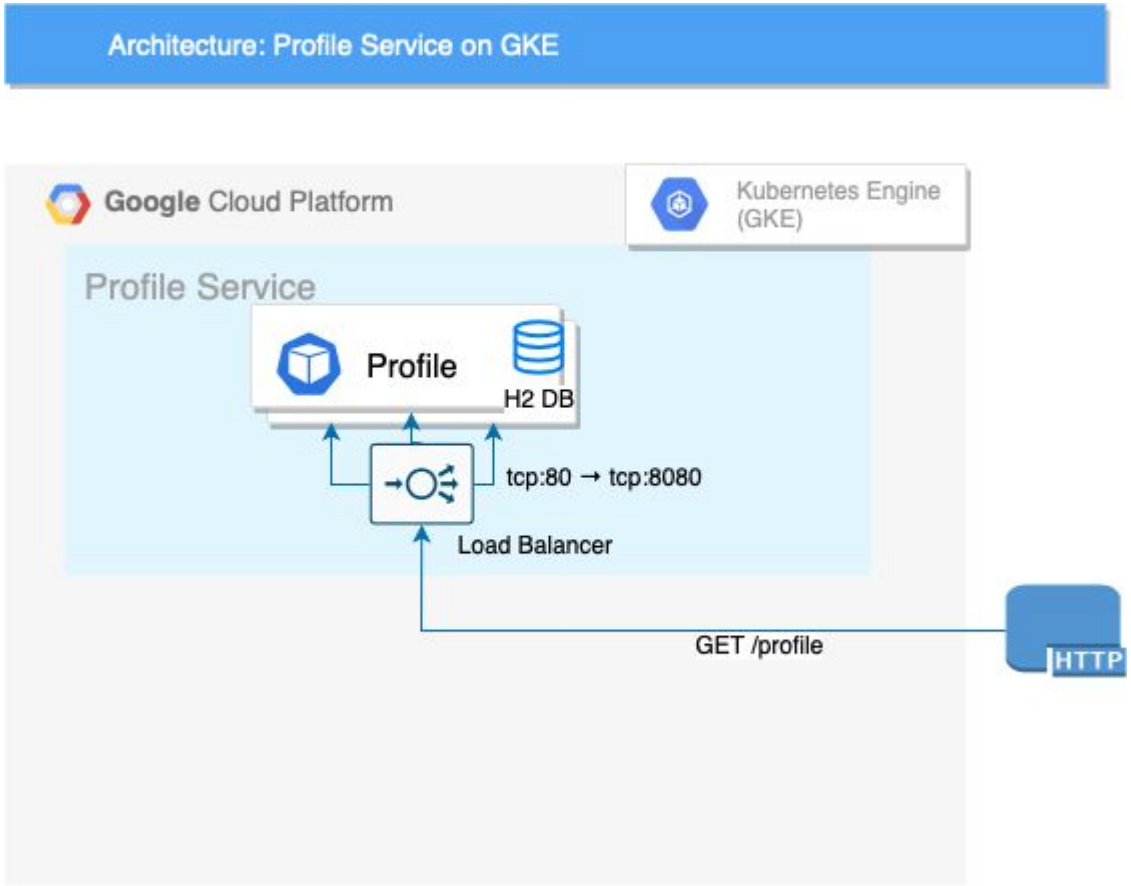


Task 2 - Using GCR and GKE to Deploy the Profile Service

- Push your image to a private registry
  - Push the profile service Docker image to Google Container Registry (GCR)
- Define a Kubernetes YAML configuration to
  - Create a deployment based on the image pushed to GCR
  - Expose the profile service via a (GCP) load balancer

Task 2 - Using GCR and GKE to Deploy the Profile Service

- Profile service architecture
- The backend application accepts **GET** requests at `/profile`
- The load balancer will map port 80 to port 8080



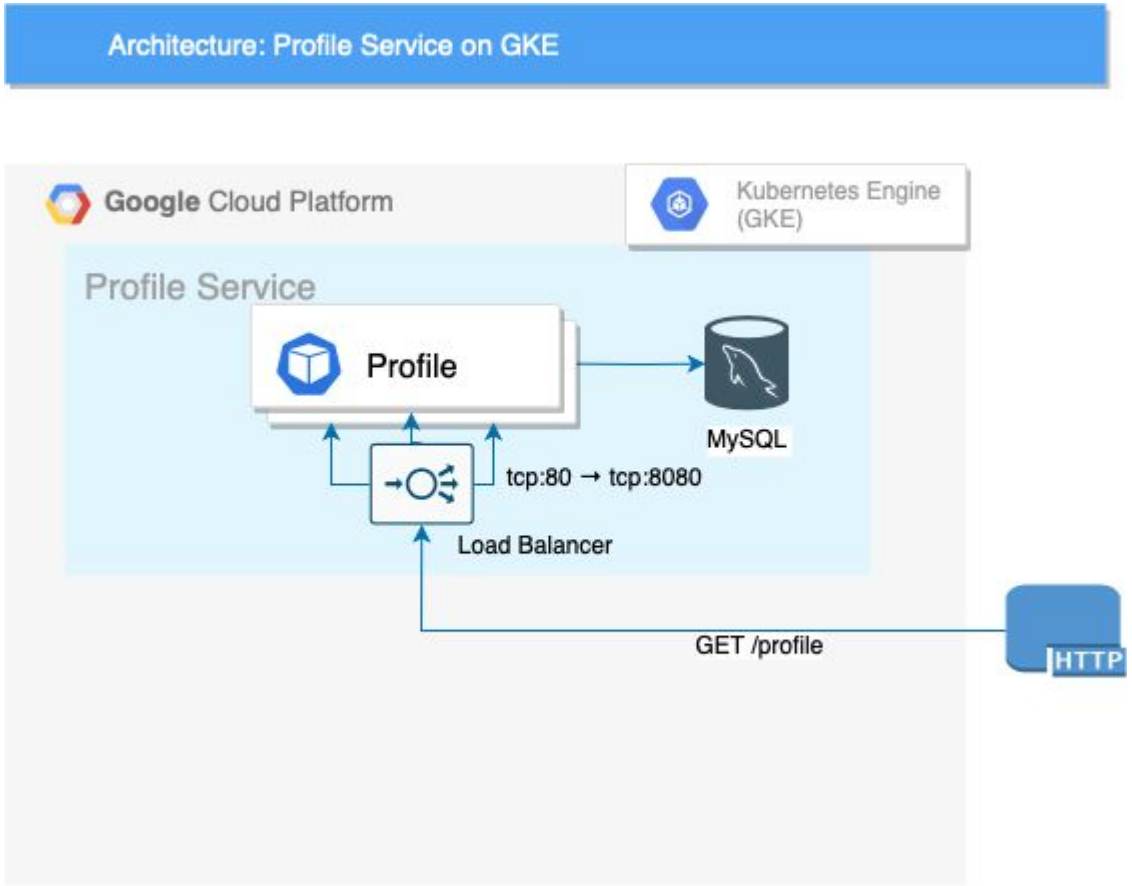
Task 3 - Introduction to Helm Charts

- Deploy a MySQL database using Helm
  - Update the profile service to use MySQL instead of the embedded H2 database
  - Remember to push your updated image to GCR!

- Develop a Helm chart for the profile service
  - Release the profile service via helm

Task 3 - Use Helm Charts and Migrate to MySQL

- Profile service architecture (MySQL)
- The backend application accepts **GET** requests at `/profile`
- The load balancer should map 80 to 8080



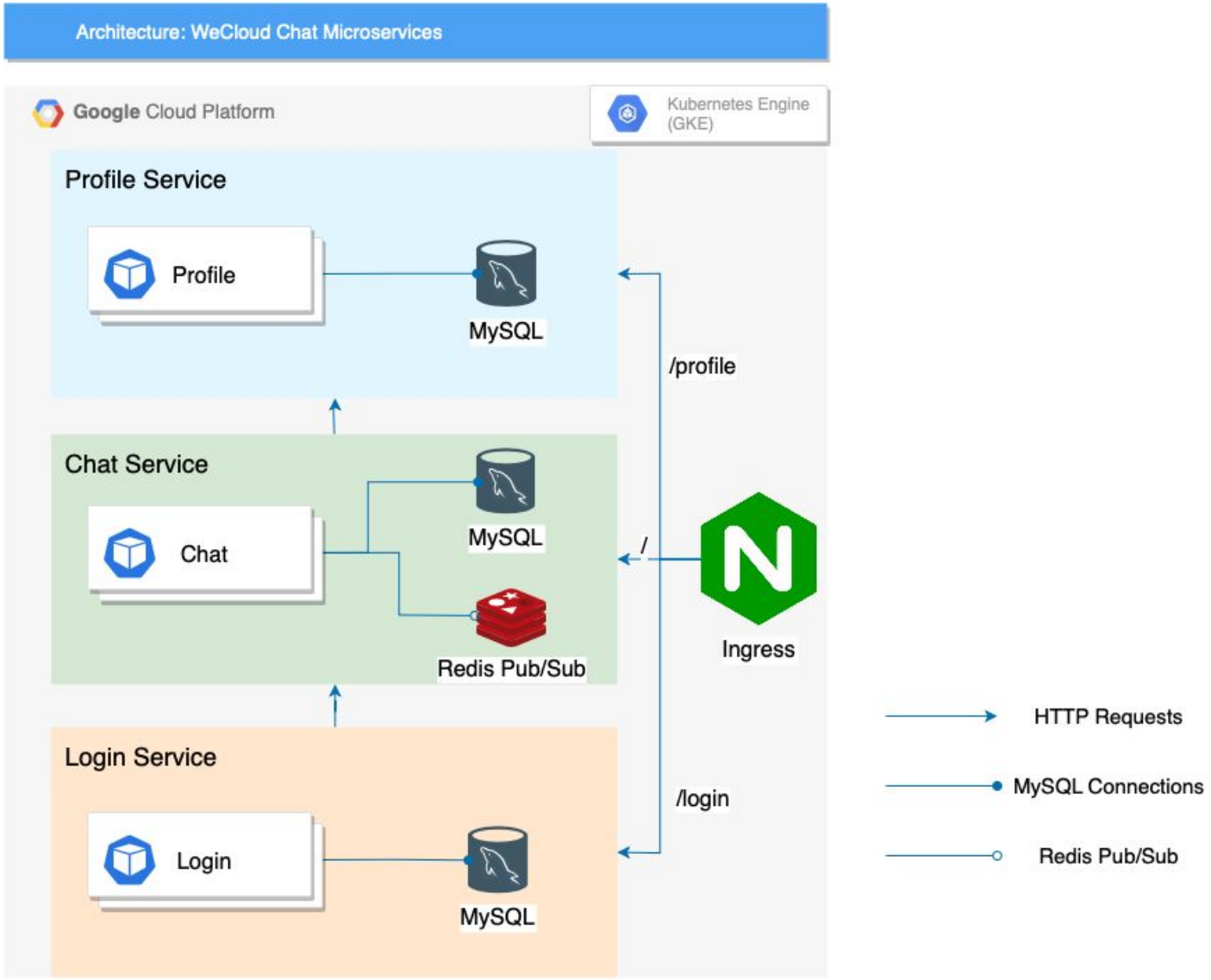
Task 4 - Cloud Chat Microservices

- Builds on Task 3
  - Additional login and group chat services

- Login service
  - Requires a **separate** MySQL database to store user login information

- Group chat service
  - Redis Pub/Sub messaging channel for scalability and real time communication
  - A **separate** MySQL database to persist messages

Task 4 - Cloud Chat Microservices



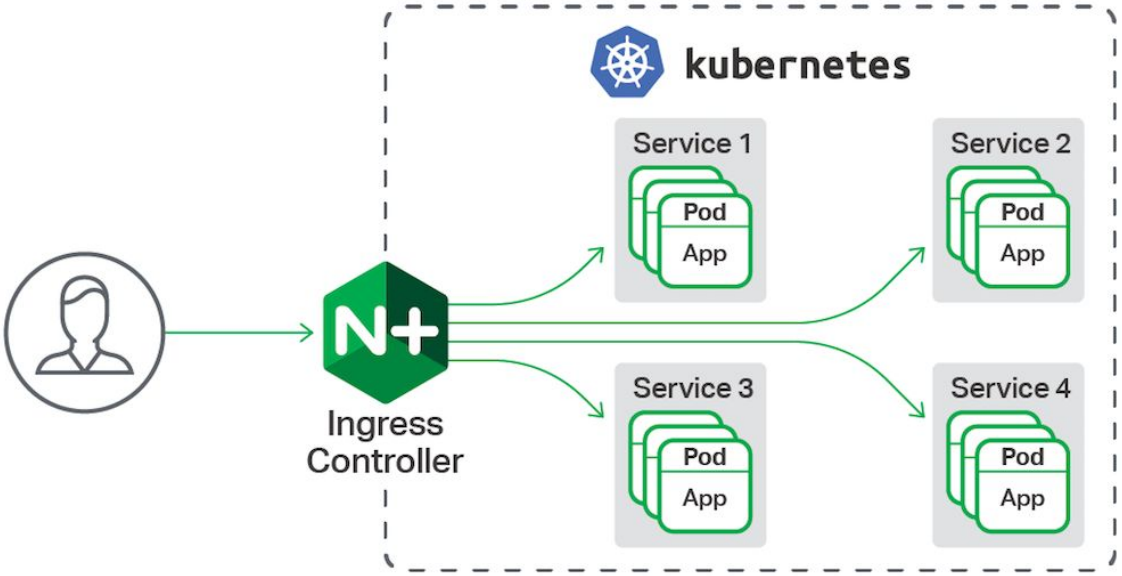
Task 4 - Cloud Chat Microservices

- Ingress: An API object that manages external access to the services in a cluster, typically HTTP.
- Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.
- In our case for Task 4, we have the following port mapping:

path	serviceName	servicePort
/chat	spring-chat-service	80
/login	spring-login-service	80
/profile	spring-profile-service	80

Task 4 - Cloud Chat Microservices

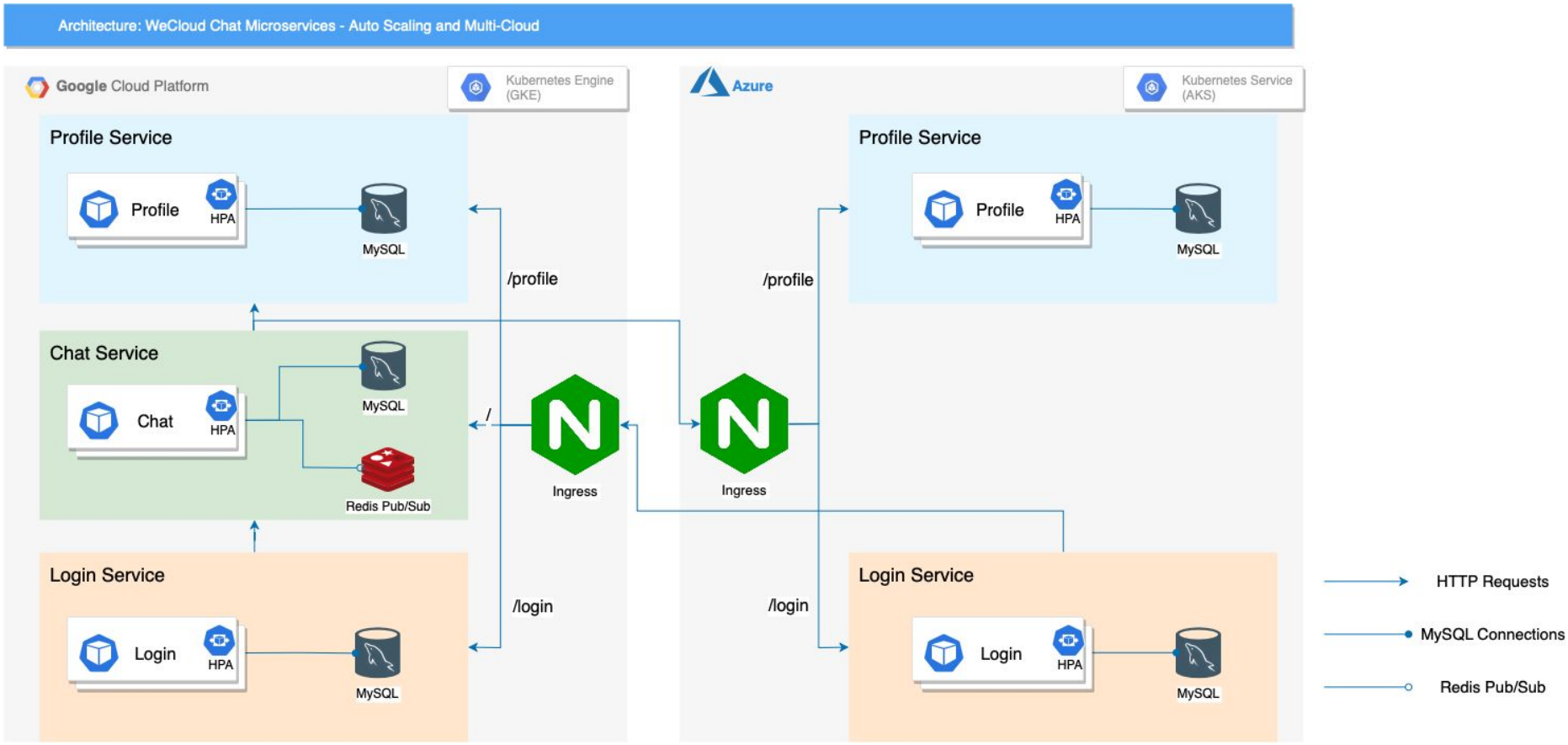
- You must have an ingress controller to satisfy an Ingress. Only creating an Ingress resource has no effect. An Ingress controller is responsible for fulfilling the Ingress, usually with a load balancer. You may need to deploy an Ingress controller such as ingress-nginx.



Task 5 - Autoscaling, Multiple Cloud Deployment and Fault Tolerance

- Build upon Task 4
  - Consider how to handle downstream service failures
- Achieve high availability
  - Multi cloud deployments!
  - Autoscaling Kubernetes deployments to accommodated increased traffic
  - Use the `HorizontalPodAutoscaler` Kubernetes object to scale the pods

Task 5 - Auto-scaling, Multiple Cloud Deployment and Fault-tolerance



Task 6 - Domain Name and Azure Front Door Service

- In this task, you will use Azure Front Door Service to achieve a path-based routing to the web application deployed on Azure and GCP.

- We will define Domain Name System (DNS) to map two IP address from previous tasks, to a single domain name.

Tips, Trips, and Tricks

- Debug, debug, debug
  - This project has many moving pieces!
  - Where is the issue occurring?
  - What is the expected behavior of the system?
- Pods and Logs
  - Did my pod start?
    - **(kubectl get pods , kubectl describe pods)**
  - Is my pod generating any logs?
    - **(kubectl logs …)**

Project 2 Penalties

Project Grading Penalties

The following table outlines the violations of the project rules and their corresponding grade penalties for this project.

Note that a penalty is the absolute value as per the table, not calculated by a percentage of your total score.

Violation	Penalty of the project grade
Incomplete submission of required files	-10%
Submitting your credentials, other secrets, or Andrew Id in your code for grading	-100%
Submitting only executables (`.jar` , `.pyc` , etc.) without human-readable code (`.py` , `.java` , `.sh` , etc.)	-100%
Attempting to hack/tamper the grader	-100%
Cheating, plagiarism or unauthorized assistance (please refer to the university policy on academic integrity and our syllabus)	-200% or R in the course

Upcoming Deadlines

- **Quiz 3 (OLI Modules 5, 6)**
  - Due on **Friday**, September 24th, 2021, 11:59PM ET
- **Project 1 Discussion**
  - Due on **Sunday**, September 26th, 2021, 11:59PM ET
- **Project 1 Project discussion (graded, 3 points)**
  - Due on **Sunday**, September 26th, 2021, 11:59PM ET
- **Project 2**
  - Due on next **Sunday**, October 1st, 2021, 11:59PM ET