Lecture 4: Containers II

LIVE TUTORIAL BY YOU AND SHIVAS!

LOADS OF FUN!

AI-5

Productionizing AI (MLOps)

Pavlos Protopapas, Shivas Jayaram

WITH FUNKY SLIDES BY RAHUL DAVE

RECAP | MICROSERVICES | DOCKER BUILDS | LIVE PIPELINE!

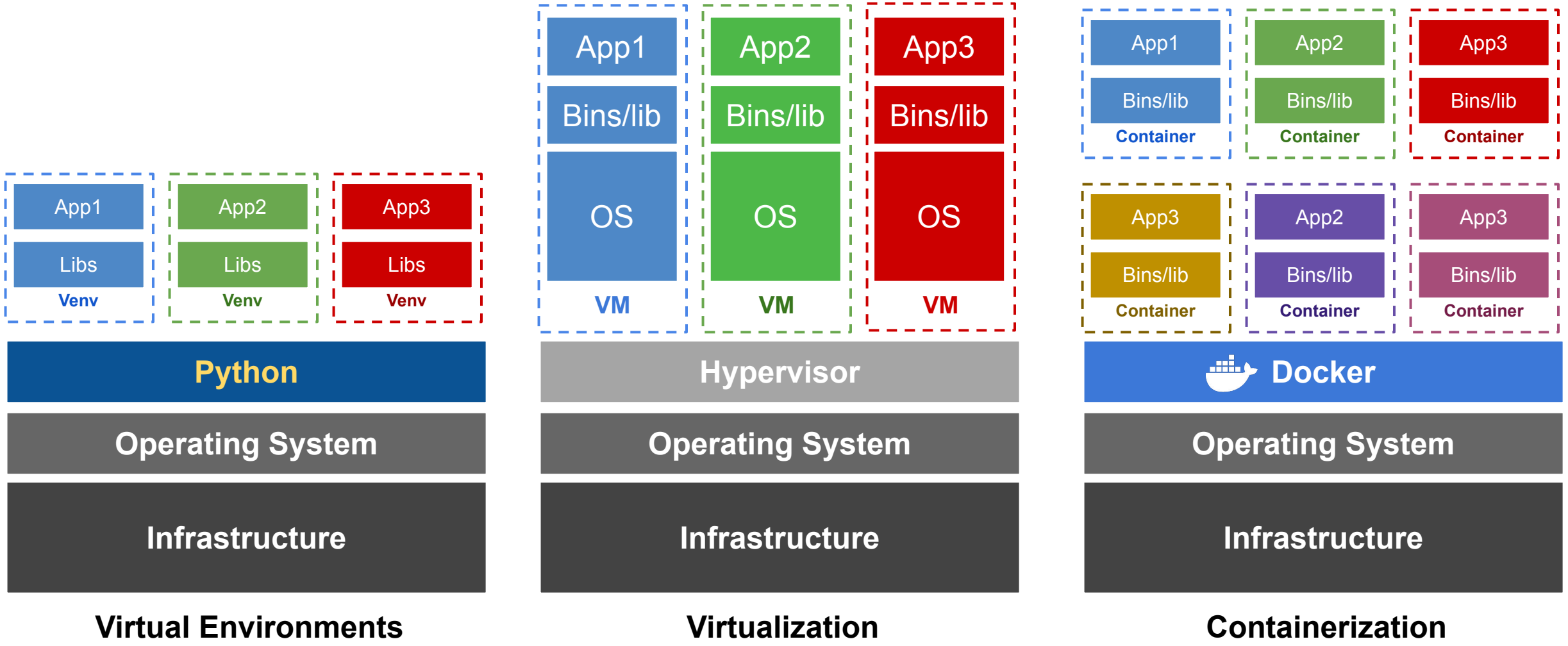
Outline

1. Recap: Review of Previous Material
2. Containers in Architecture: Microservices vs. Monolithic
3. Implementing Containers as Microservices

Outline

1. **Recap: Review of Previous Material**
2. Containers in Architecture: Microservices vs. Monolithic
3. Implementing Containers as Microservices

Recap: Environments vs Virtualization vs Containerization



Is This Really the right question?

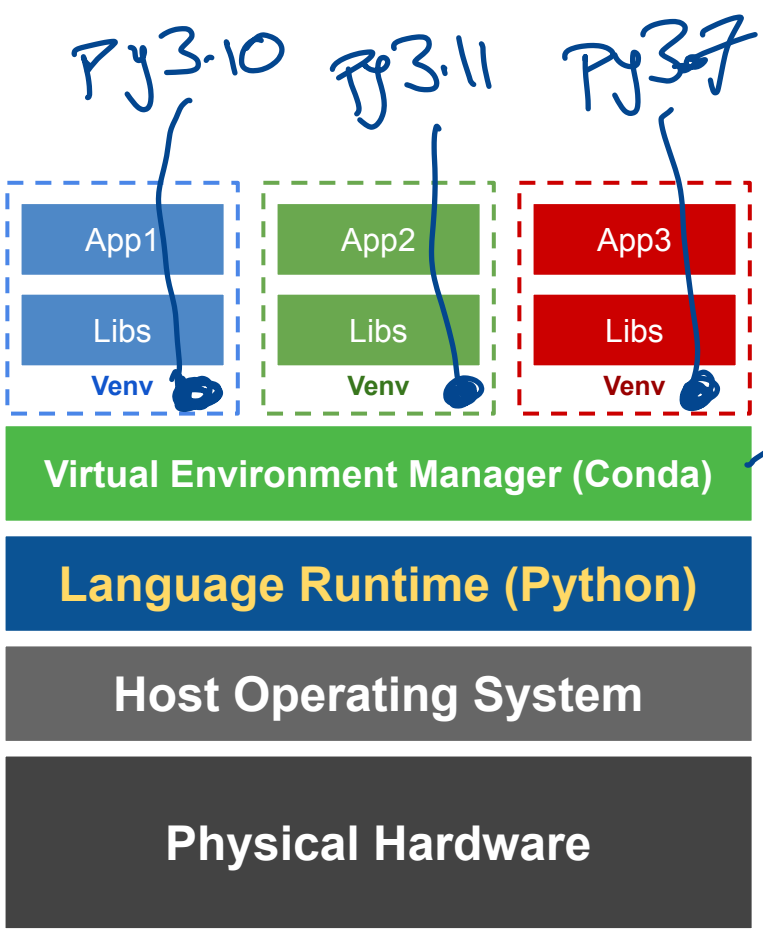
Real world scenarios will often combine.

→ VM's are great to isolate tenants from each other

→ But containers are best to isolate "jobs" from each other

→ Virtual Environments (venvs) are useful in either case. Indeed, they make building containers EASIER!

Environments



Virtual Environments

Also venv, pipenv, conda-lock, poetry, pixi, ...

- **Dependency Isolation**: Virtual environments modify the PATH and other environment variables so that the dependencies are loaded from the environment's directory, rather than system-wide directories
- **No Kernel Isolation**: Unlike VMs and containers, virtual environments don't provide any kernel level isolation.
- **Resource Utilization**: Since virtual environments don't have any additional OS or kernel, they are the most efficient in terms of resource utilization among the three.
- **Filesystem Boundaries**: Virtual environments usually don't provide isolation at the filesystem level; files written in one environment are accessible from others.

CONDA VS PIPENV : THE LOCKFILE ISSUE

CONDA	PIPENV / POETRY / ETC.
• NON PYTHON SOFTWARE	• ONLY PYTHON SOFTWARE
• CUDA / HARDWARE	• CUDA IS HARDER

PEOPLE OFTEN COMBINE

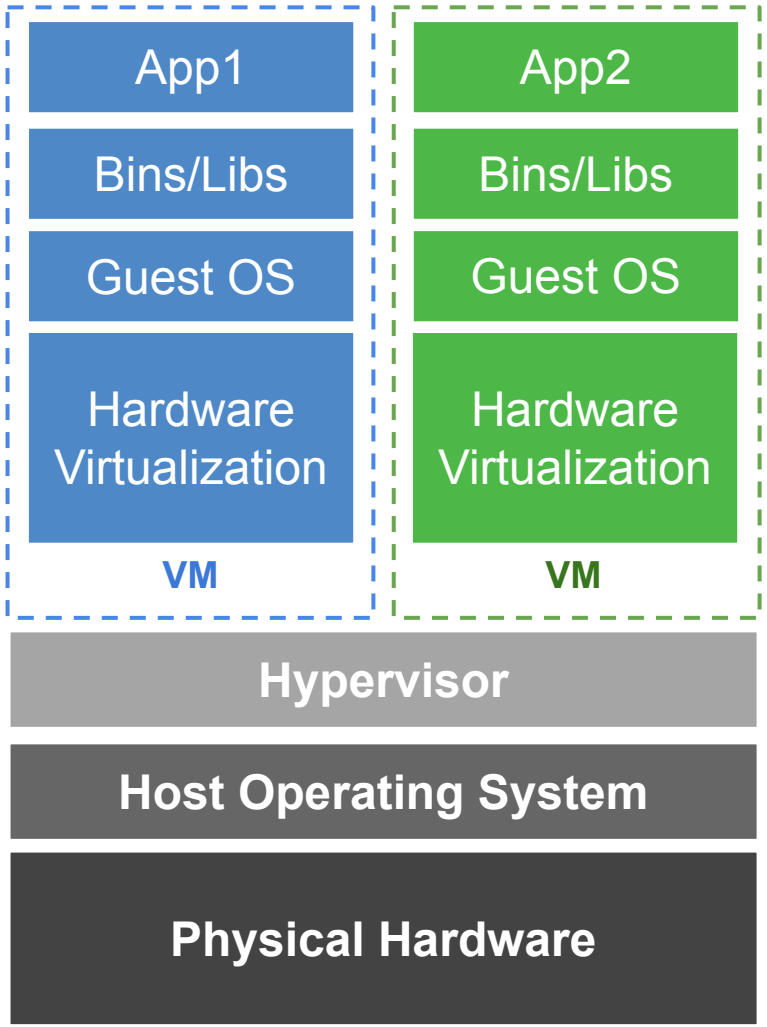
conda for cuda / pythonexec / rust / c++ / jcv a [conda-lock]
pip / poetry / pip-tools [latter 2 provide locks]

PROBLEM: lack of combined lockfiles.

SOLUTION: no general one yet, pixi combines lockfiles

EASIER TO STAY WITH ONE, BUT U DONT ALWAYS HAVE THE CHOICE

Virtualization (Virtual Machines)

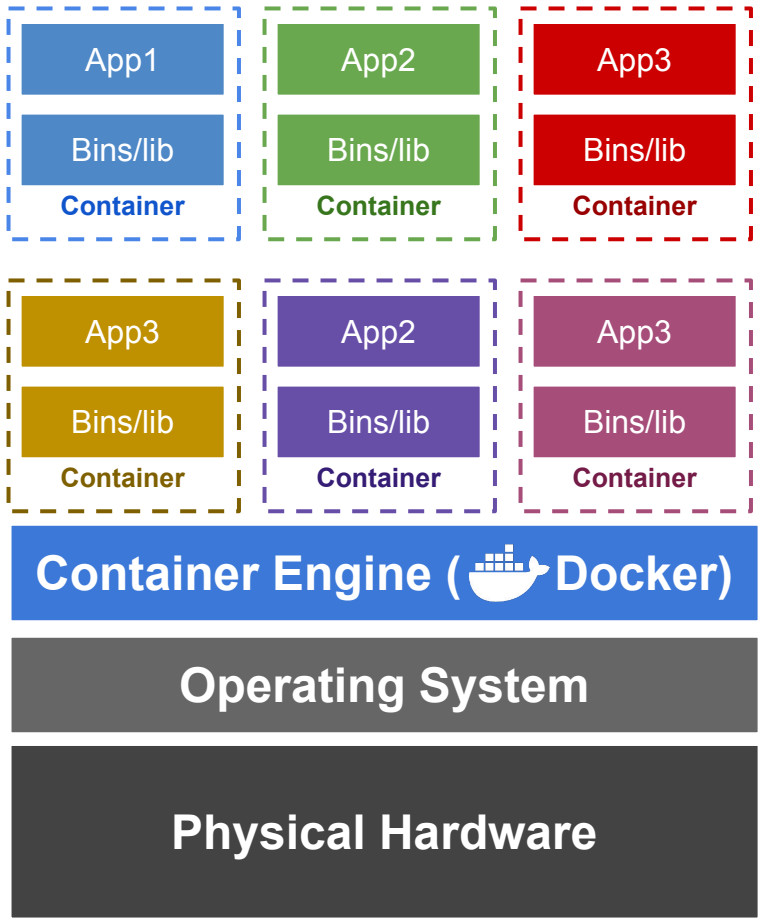


Virtualization

- **CPU Virtualization**: VMs usually have a set number of virtual CPU cores allocated by the hypervisor. These virtual CPUs map to physical CPU cores, but the hypervisor adds a layer of management and overhead, which can lead to inefficiencies.
- **Emulated Devices**: VMs have emulated hardware devices, meaning the VM sees virtual CPUs, virtual network adapters, and virtual disks that the hypervisor translates to real hardware resources.
- **Full OS**: Each VM runs its full guest OS. This means that each VM has its own separate kernel space and user space, making resource management fully independent but less efficient.
- **Resource Allocation:** RAM and CPU are often (not always) allocated in blocks, and disk space is generally pre-allocated, making VMs less flexible in terms of resource utilization.

Containerization

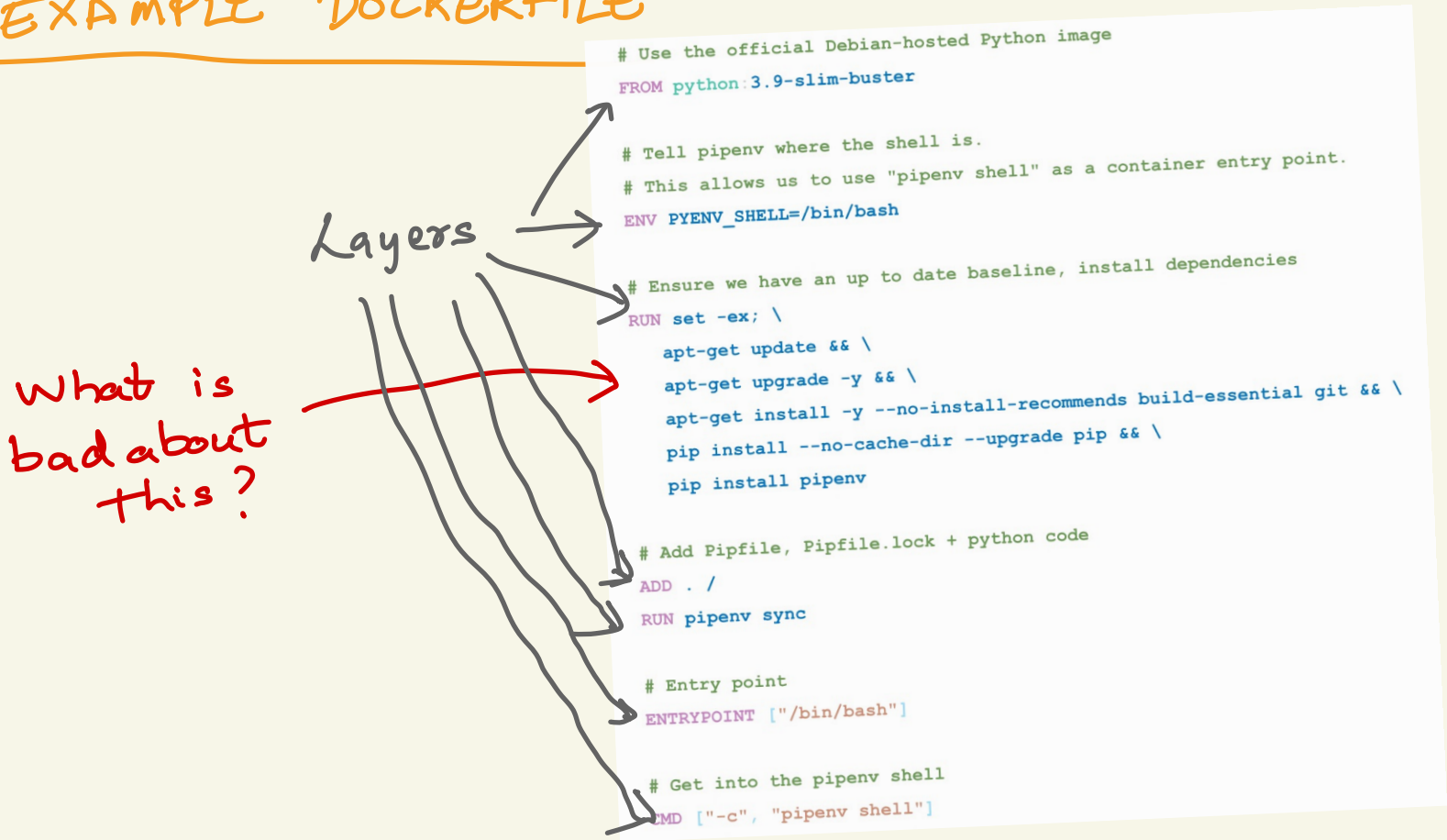
REPRODUCIBILITY TO THE OS LEVEL



Containerization

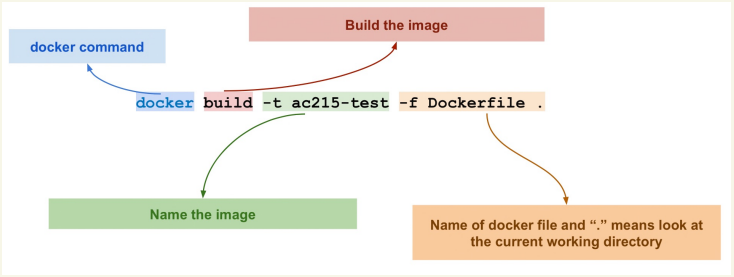
- **Namespaces**: Containers use kernel features like namespaces to provide isolation of processes and resources. This allows each container to operate as if it is the only application running on the system. Example namespaces include:
  - PID Namespace: Isolates the process ID number space. In other words, processes in different PID namespaces can have the same PID.
  - Mount Names: Isolates the file system tree so that each namespace can have its own file system layout.
- **Control Groups (cgroups)**: Complementary to namespaces, cgroups limit resource usage, like CPU, memory, and IO, allowing for better resource utilization compared to VMs.
- **Process Virtualization:** Namespaces and cgroups together enable process virtualization by allowing processes to run in isolated environments with controlled access to system resources.
- **Shared Kernel**: Containers share the host's OS kernel but have their own filesystem, libraries, and bins, making them lightweight yet isolated.
- **Direct Access**: Containers can access host resources more directly, avoiding much of the overhead introduced by hypervisors in VMs.

EXAMPLE DOCKERFILE

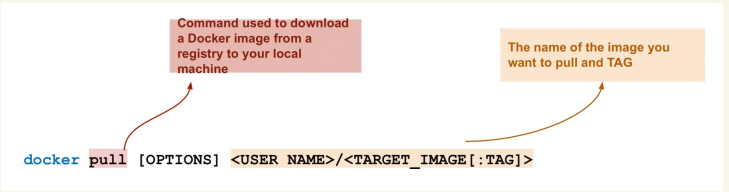


Some Docker Commands

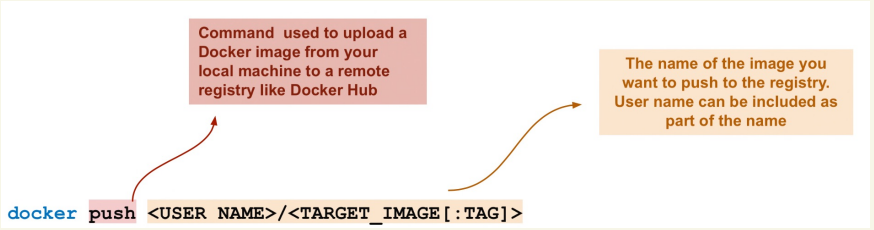
Build



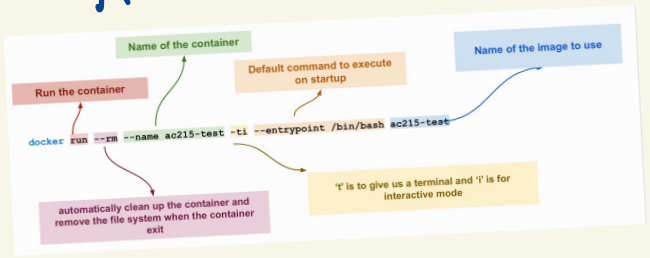
Pull



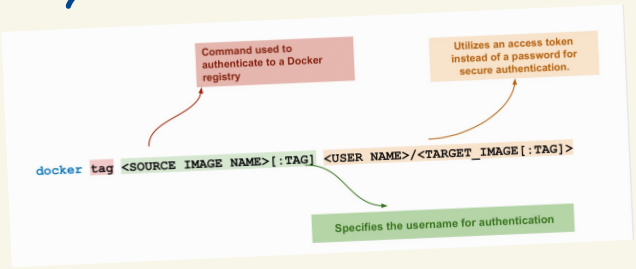
Push



Run



Tag



DOCKERFILE vs IMAGE vs CONTAINER

```
FROM python:3.7-slim

RUN apt-get update
RUN apt-get install -y --no-i
nstall-recommends gcc

COPY myapp/ .
COPY setup.py .

RUN python setup.py install
```

Just a file!

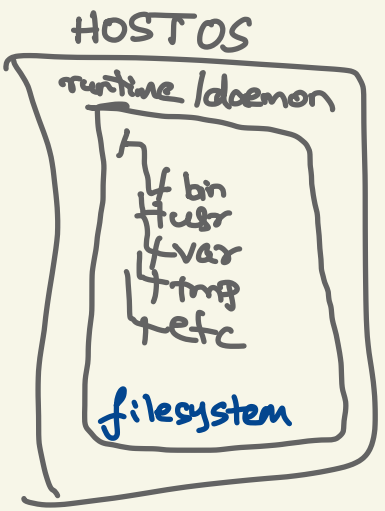
The image built
243 MB

A large image

```
FROM python:3.7-slim

RUN apt-get update
RUN apt-get install -y --no-i
nstall-recommends gcc

COPY myapp/ .
COPY setup.py .
RUN python setup.py install

RUN apt-get remove -y gcc
RUN apt-get -y autoremove
```

Layers are cached on host

This Image
245 MB



A not so large container

IMAGES from PYTHONSPEED

PROBLEM DOCKER IMAGES ARE APPEND-ONLY

- The image has layers with gcc in them to build our project
- It also has a remove layer
  This does NOT REDUCE image size
- The container has the files removed

  BUT

  YOU HAVE A LARGE IMAGE AND YOU HAVE TO DOWNLOAD COMPILER TO BUILD.

SOLUTION 1 : SINGE LAYER

```
FROM python:3.7-slim

COPY myapp/ .
COPY setup.py .

RUN apt-get update && \
    apt-get install -y --no-i
nstall-recommends gcc && \
    python setup.py install
&& \
    apt-get remove -y gcc &&
apt-get -y autoremove
```

161 MB IMAGE

Put everything into one layer so that compiler files are gone

PROBLEM
- we destroyed the build cache
- builds are slower

(see middle dockerfile 2 slides back)

or call "docker build --squash"

Pro Tips 1: **multi-stage builds**

SOLUTION 2

Running commands can take up a lot of disk space. For example, when **installing** and **building** packages, we **download** and **produce** many files. This can make our image size **very large**.

Question: Does our app need all of these files?

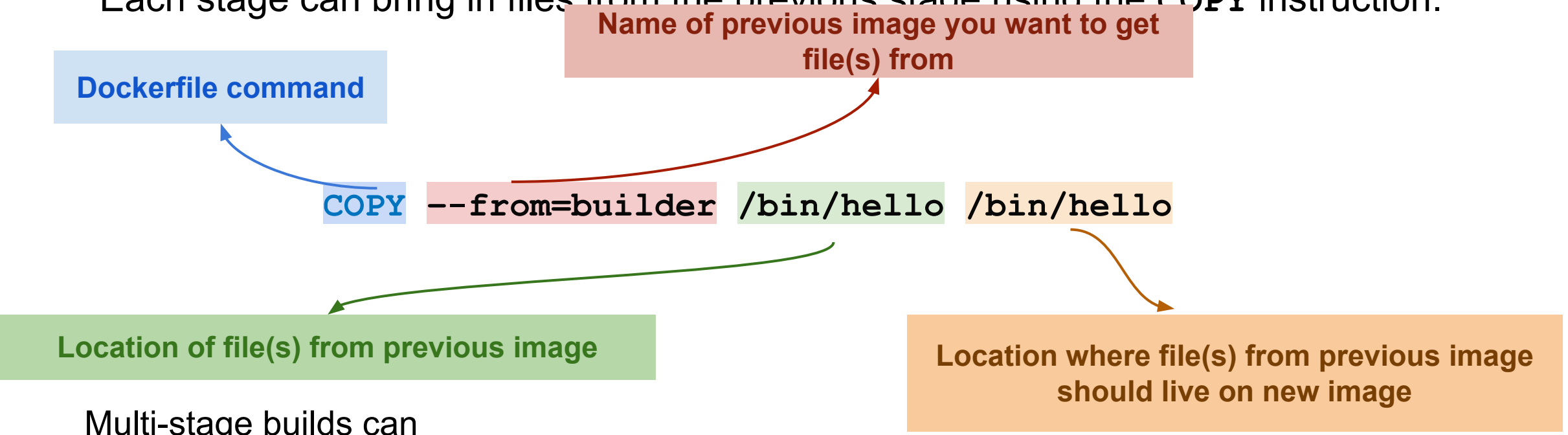
Answer: Usually no! We just need the executable and its runtime dependencies.

Multi-stage builds allow us to bring **only what we need** into the final Docker image.

Pro Tips 1: **multi-stage builds**

Multi-stage builds in Docker allow the use of multiple **`FROM`** statements in a single **`Dockerfile`**.

Each stage can bring in files from the previous stage using the **`COPY`** instruction.



Multi-stage builds can

- decrease container size
- improve security (e.g., avoid sharing private keys from GitHub)
- leverage different base images for each stage while preserving only the final image.

multi-stage builds

Pro Tips 1: **multi-stage builds: example**

```
# First Stage: Building the Python application
FROM python:3.8 AS build-env
WORKDIR /app
COPY BLAH BLAH
RUN  BLAH BLAH

# Second Stage: Copy the dependencies and run the application
FROM python:3.8-slim
COPY --from=build-env /usr/local/lib/python3.8/site-packages
/usr/local/lib/python3.8/site-packages
WORKDIR /app
COPY BLAH BLAH
CMD BLAH BLAH
```

MULTI STAGE BUILD EXAMPLE

```
# This is the first image:
FROM ubuntu:18.04 AS compile-
image
RUN apt-get update
RUN apt-get install -y --no-i
nstall-recommends gcc build-e
ssential

WORKDIR /root
COPY hello.c .
RUN gcc -o helloworld hello.c

# This is the second and fina
l image; it copies the compil
ed
# binary over but starts from
the base ubuntu:18.04 image.
FROM ubuntu:18.04 AS runtime-
image

COPY --from=compile-image /ro
ot/helloworld .
CMD ["./helloworld"]
```

88.9 MB

- this is great: small size and no compiler
- problem 1: standard python has a problem in that it is splatted all over the filesystem
- One way "pip install --user" and that puts everything into ".local"
- BETTER VENVS

VENVS : all of python in one spot

FIRST STAGE

```
FROM python:3.7-slim AS compile-image
RUN apt-get update
RUN apt-get install -y --no-install-recommends build-essential gcc

RUN python -m venv /opt/venv
# Make sure we use the virtualenv:
ENV PATH="/opt/venv/bin:$PATH"

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY setup.py .
COPY myapp/ .
RUN pip install .
```

← or install from .pipenv lockfile

```
FROM python:3.7-slim AS build-image
COPY --from=compile-image /opt/venv /opt/venv

# Make sure we use the virtualenv:
ENV PATH="/opt/venv/bin:$PATH"
CMD ['myapp']
```

← SECOND STAGE

Virtual Environments and Containers come together to make multi-stage builds nice.

DOES THIS FIX ALL PROBLEMS? MOSTLY

BUT

1. No. Multi-Stage Builds can be slow.
   Why? Note that gcc layer in first build may not be in cache if you pull from registry. SOLUTION: push/pull all stages!

2. CAVEAT: if you are using a venv system like poetry or anything based on "pyproject.toml", be careful. Adding a new task into the file can invalidate the cache.

STARTING WITH (POSSIBLY BUILT) CONTAINER

The -u treats unset variables as an error

GOOD SHELL HYGEINE

error if any command in a pipe fails, and all errors return a failure code

The ||true handles first time

```
#!/bin/bash
set -euo pipefail
# Pull the latest version of
the image, in order to
# populate the build cache:
docker pull itamarst/hellowor
ld || true
# Build the new version:
docker build -t itamarst/hell
oworld .
# Push the new version:
docker push itamarst/hellowor
ld
```

https://sipb.mit.edu/doc/safe-shell/

what about for 2 stage pipelines?

BUILDING IN 2 STAGES

STAGE 1

```
#!/bin/bash
set -euo pipefail
# Pull the latest version of
the image, in order to
# populate the build cache. T
his isn't necessary if
# you're using BuildKit.
docker pull itamarst/hellowor
ld:compile-stage || true
docker pull itamarst/hellowor
ld:latest        || true

# Build the compile stage:
docker build --target compile
-image \
       --build-arg BUILDKIT_I
NLINE_CACHE=1 \
       --cache-from=itamarst/
helloworld:compile-stage \
       --tag itamarst/hellowo
rld:compile-stage .
```

```
# Push the new versions:
docker push itamarst/hellowor
ld:compile-stage
docker push itamarst/hellowor
ld:latest
```

STAGE 2

```
# Build the runtime stage, us
ing cached compile stage:
docker build --target runtime
-image \
       --build-arg BUILDKIT_I
NLINE_CACHE=1 \
       --cache-from=itamarst/
helloworld:compile-stage \
       --cache-from=itamarst/
helloworld:latest \
       --tag itamarst/hellowo
rld:latest .

# Push the new versions:
docker push itamarst/hellowor
ld:compile-stage
docker push itamarst/hellowor
ld:latest
```

match docker classical behavior.

TAG STAGES EXPLICITLY

You can build from your local Dockerfile but this make sure to pull down all images to cache.

FROM PYTHONSPEED.

DOCKER vs PYTHON

```
FROM python:3.8-slim-buster
COPY requirements.txt /tmp
RUN pip install -r requirements.txt
COPY . /tmp/myapp
RUN pip install /tmp/myapp
```

(1) COPY AND INSTALL FIRST (could use lockfile)

(2) Then put in your app, so any changes in it dont affect cached layers.

PROBLEM

say something here changes.

```
[tool.poetry]
name = "myexample"
version = "0.1.0"
description = ""
authors = ["Itamar Turner-Trauring"]

[tool.poetry.dependencies]
python = "^3.6"
Flask = "^1.1.2"

# ...
```

then

```
FROM python:3.8-slim-buster

WORKDIR /app

# Install poetry:
RUN pip install poetry

# Copy in the config files:
COPY pyproject.toml poetry.lock ./
# Install only dependencies:
RUN poetry install --no-root --no-dev

# Copy in everything else and install:
COPY . .
RUN poetry install --no-dev
```

INVALIDATION

WORKAROUNDS

- see poetry issue 1301 on github [try to use only lockfile]
- no probs on pipenv.
- on pixi: "pixi --frozen"

GENERAL PROBLEM WITH PYPROJECT.TOML.

Expect venv managers to fix.

Pro Tips 2: **multi-platform images**

When building a more complicated Docker image, there is a small chance the specific platform (OS and CPU architecture) on your machine causes issues when sharing the Docker image with someone on a different machine

Example: Building a complex image on M1 Mac (linux/arm64) and trying to run the image on an older Macbook (linux/amd64)

Error message to look out for

```
The requested image's platform (linux/arm64) does not match
the detected host platform (linux/amd64) and nospecific
platform was requested
```

Solution:

- Use the `--platform` flag within the FROM command in your Dockerfile to specify the target OS and CPU architecture for the build output

multi-platform images

Outline

1. Recap: Review of Previous Material
2. **Containers in Architecture: Microservices vs. Monolithic**
3. Implementing Containers as Microservices

Why use Containers? MONOLITHIC VS SEPARATED APPS.

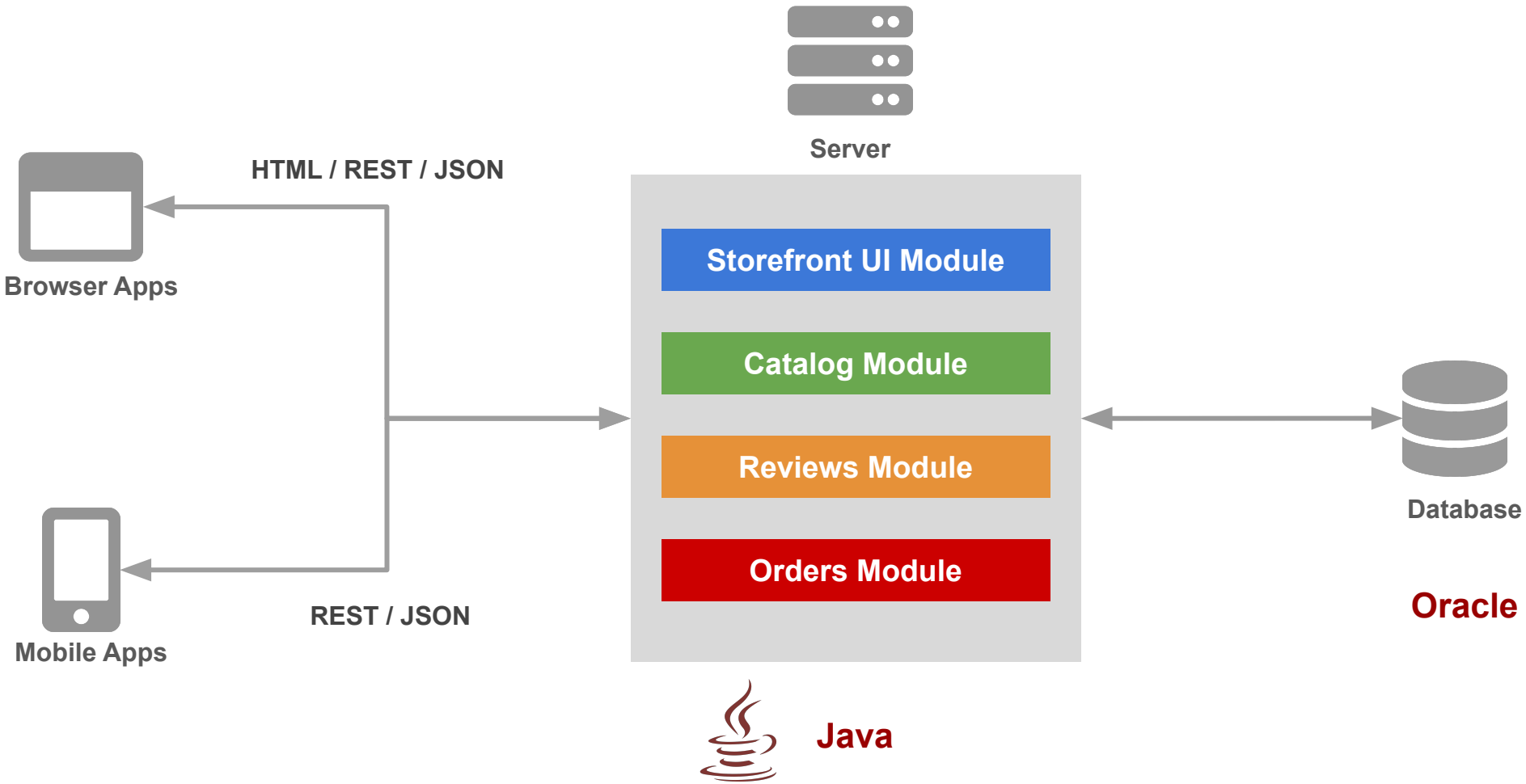
Conceptual Scenario

- Picture building a comprehensive application, such as an online store.

Traditional Approach

- Traditionality you would build this using a Monolithic Architecture

Monolithic Architecture



Monolithic Architecture - **Advantages**

Simplicity in Development:

Streamlined development process as most tools and IDEs natively support monolithic applications.

Ease of Deployment:

Hassle-free deployment with all components bundled into a single, unified package.

Scalability:

Easier to scale horizontally by replicating the entire application as a whole.

Monolithic Architecture - **Disadvantages**

Maintenance Challenges:

Complexity increases over time, making it harder to implement changes or find issues.

System Vulnerability:

A failure in a single component can lead to the collapse of the entire system.

Patching Difficulties:

Patching or updating specific modules can be cumbersome due to tightly-coupled components.

Monolithic Architecture - **Disadvantages**

Technology Lock-in:

Adopting new technologies or updating existing ones can be problematic due to interdependencies.

Slow Startup:

Increased startup time as all components must be initialized simultaneously.

Applications have changed dramatically

A decade ago

Apps were monolithic
Built on a single stack (e.g. .Net or Java)
Long lived
Deployed to a single server

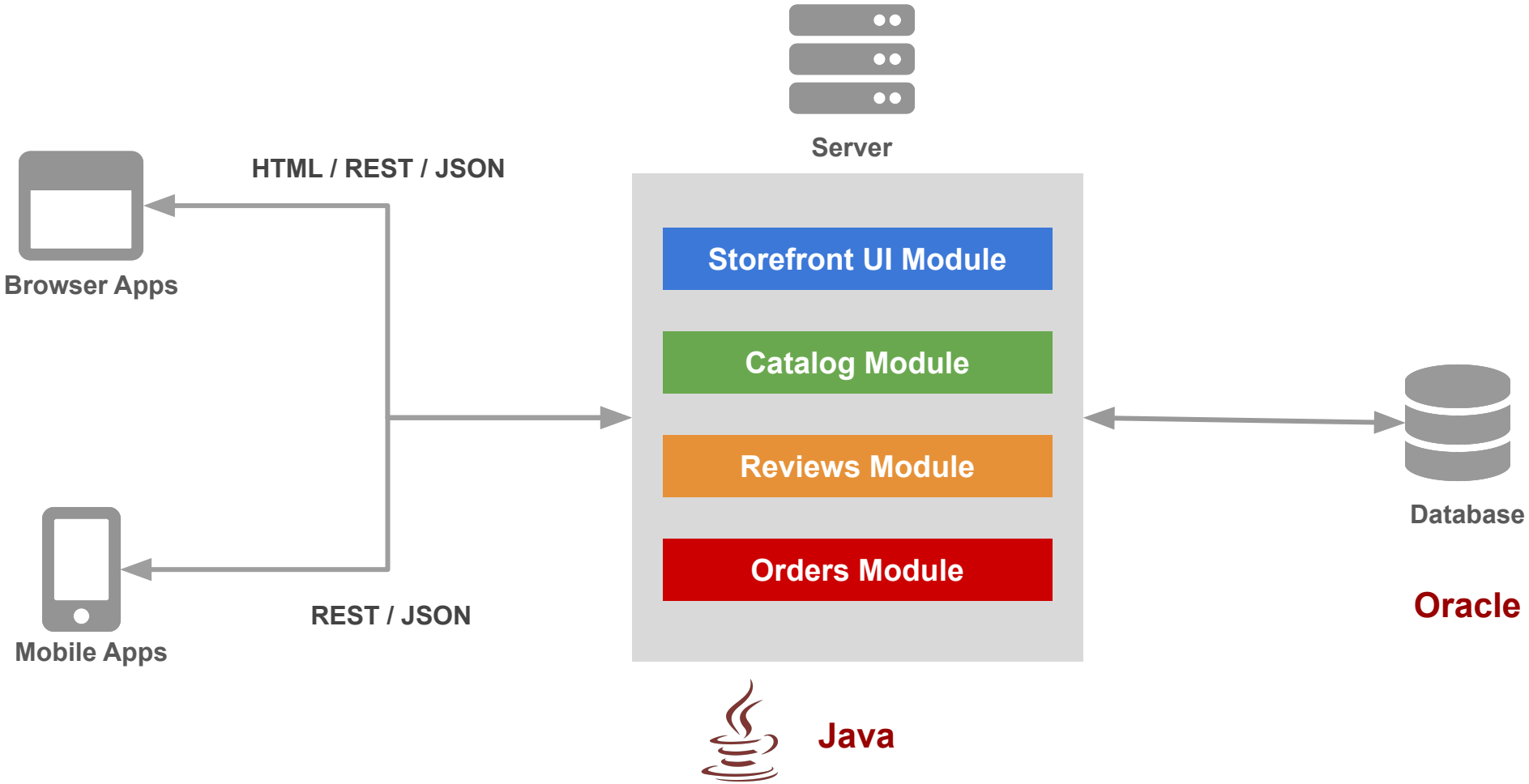
Today

Apps are constantly being developed
Build from loosely coupled components
Newer version are deployed often
Deployed to a multitude of servers

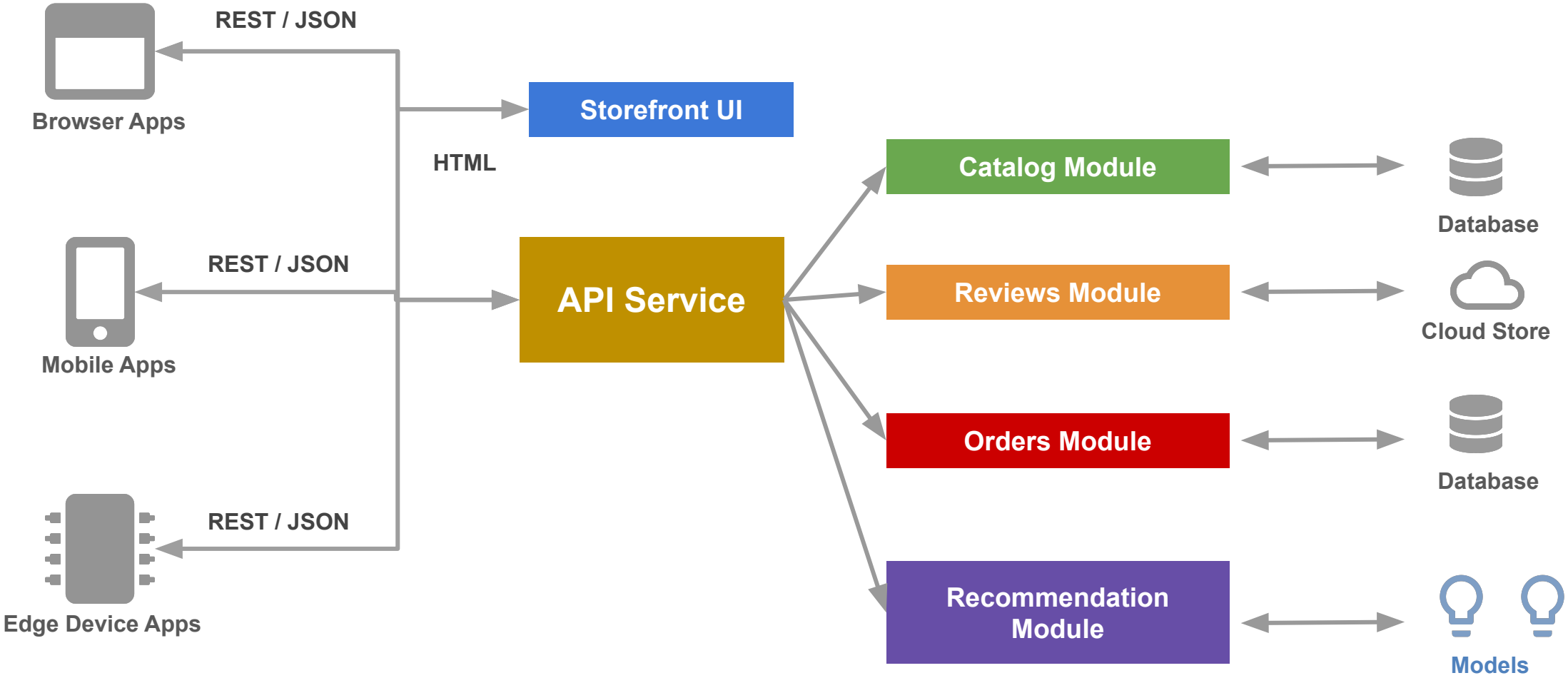
Data Science

Apps are being integrated with various data types/sources and models

Monolithic Architecture



Today: Microservice Architecture



Microservice Architecture - **Advantages**

Simplified Maintenance:

Modular design makes it easier to manage, update, and debug individual services.

Fault Isolation:

Independent components ensure that failure in one service doesn't bring down the entire application.

Streamlined Patching:

Easier to patch or update specific services without affecting the entire system.

Technological Flexibility:

Adapting to or adopting new technologies becomes seamless due to service independence.

Quick Startup:

Reduced startup time as all components can be initialized in parallel.

Microservice Architecture - **Disadvantages**

Development Complexity:

Varied technologies across components can complicate the development process.

Deployment Hurdles:

Multiple technologies and dependencies require a complex setup for deployment.

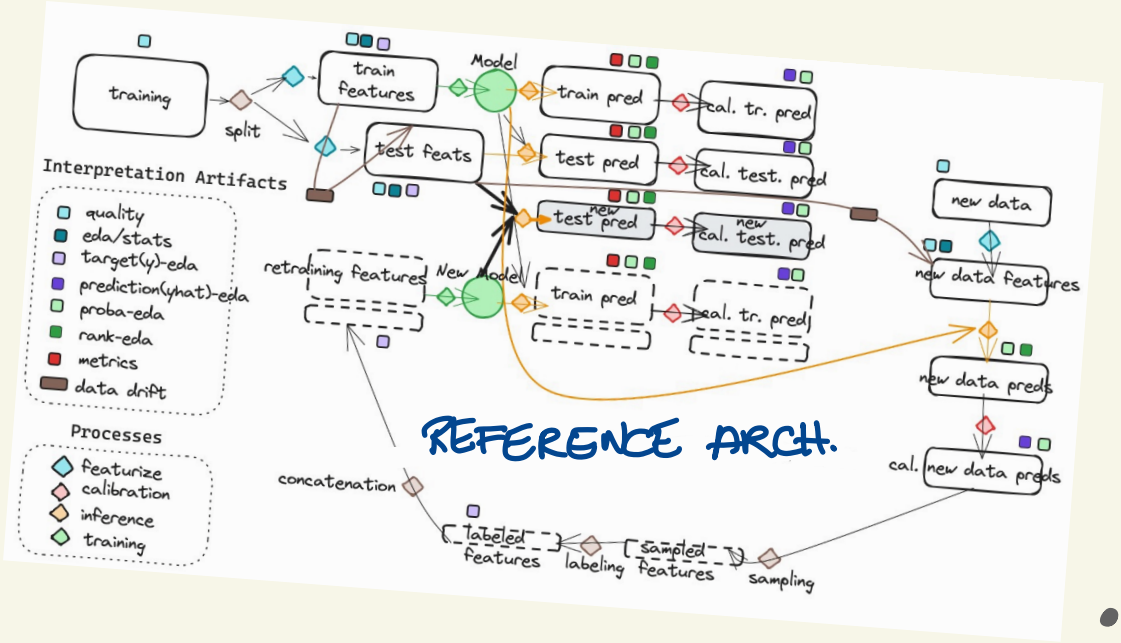
Scaling Concerns:

Scaling the entire application can be intricate due to disparate components.

BUT : Opportunity to Scale Individual Components

Docker + Kubernetes

MICROSERVICES AND DATA SCIENCE



- many moving parts
- data oriented
- processes (eg featurize) or inference repeated in many contexts
- intricate dependencies

- repeated processes can be dockerized as microservices, kept completely general, with filenames, etc passed in as environment. DOCKER COMPOSE / KUBERNETES

DATA MEANS STORAGE MEANS MOUNTS

- If you are going to create and store
- form data you must store it
- BIND mounts are stored anywhere on host system
- VOLUMES are stored in part of the host filesystem managed by docker

Create a volume named vol-name –

```
docker volume create vol-name
```

View detailed information about the volume, like creation date, mount-point, storage-driver etc. –

```
docker volume inspect vol-name
```

Remove the volume –

```
docker volume rm vol-name
```

Remove all unused local volumes (excluding the ones currently being using by containers) –

```
docker volume prune
```

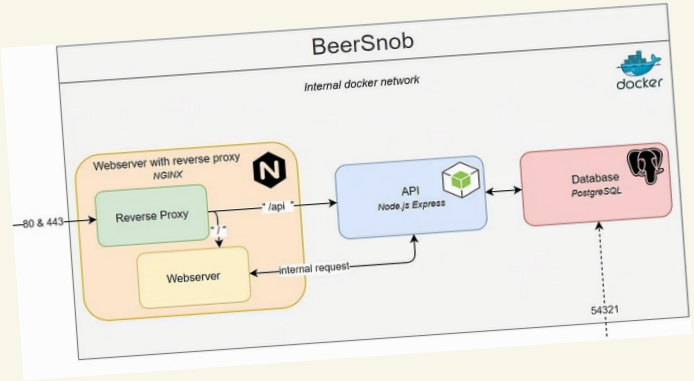
VOLUME docker run BIND

```
source=$HOME/docker/volumes/postgr
```

```
--mount type=bind,source="$pwd",ta
```

from techmoreMO.com

Docker Compose




```
  beersnob_api:
    container_name: beersnob_api
    hostname: beersnob_api
    build:
      context: ./beersnob_api
      dockerfile: Dockerfile
    ports:
     - 54322:5000
    volumes:
      - ./beersnob_api/src/:/usr/src/app/
      - /usr/src/app/node_modules
    restart: unless-stopped
    environment:
      NODE_ENV: ${BEERSNOB_ENVIRONMENT}
    depends_on:
      - beersnob_database
```

docker-compose.yml

```
version: '3'

services:

  beersnob_database:
    container_name: beersnob_database
    hostname: beersnob_database
    image: postgres
    volumes:
      - ./volume:/var/lib/postgresql
    environment:
      - POSTGRES_DB=beersnobdb, beersnobdb
      - POSTGRES_USER=mhuls
      - POSTGRES_PASSWORD=aStrongPassword
    ports:
      - 54321:5432
    restart: unless-stopped
```

```
  beersnob_webserver:
      container_name: beersnob_webserver
      hostname: beersnob_webserver
      build:
          context: ./beersnob_webserver
          dockerfile: Dockerfile
      ports:
        - 80:80
        - 443:443
      volumes:
        - ./beersnob_webserver/src/test:/u
      restart: always
      depends_on:
        - beersnob_database
```

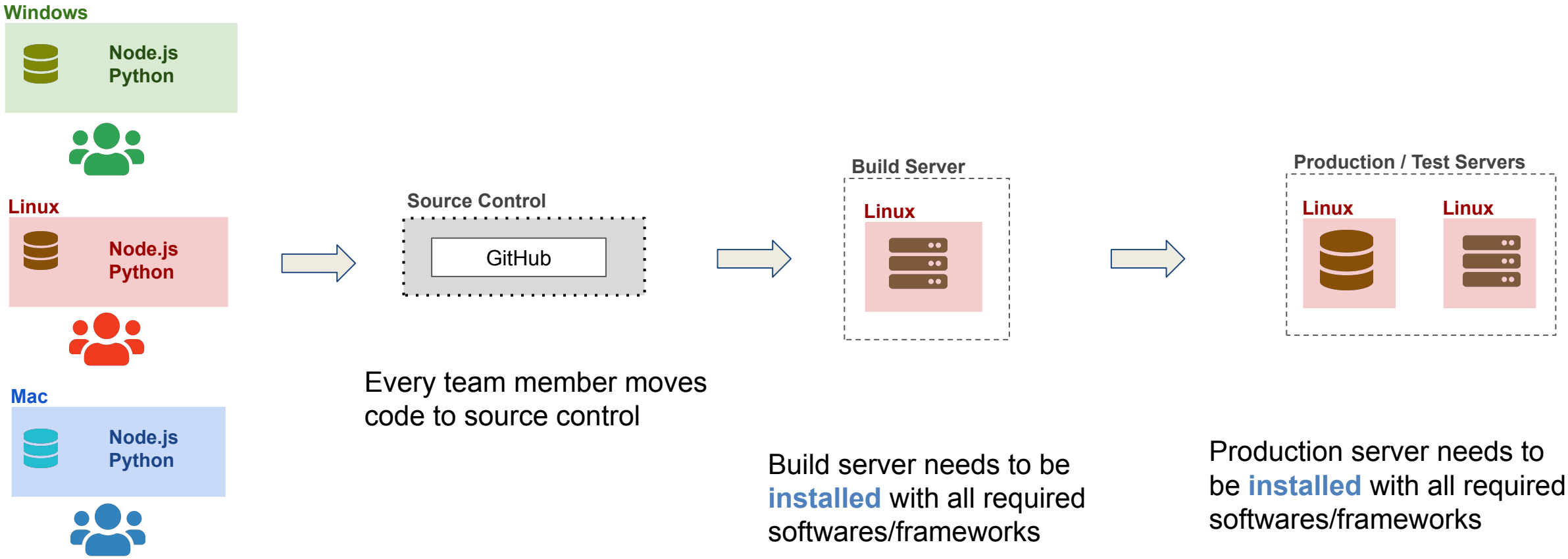
https://towardsdatascience.com/docker-compose-for-absolute-beginners-how-does-it-work-and-how-to-use-it-examples-733ca24c5e6c

Why use Containers?

DEVELOPER EXPERIENCE

- Consider a software development team workflow for developing an App
- Traditionality you would develop/build this independently in various machines (dev, test, qa, prod)

Software Development Workflow (no Docker)



Every team member moves code to source control

OS Specific **installation** in every developer machine

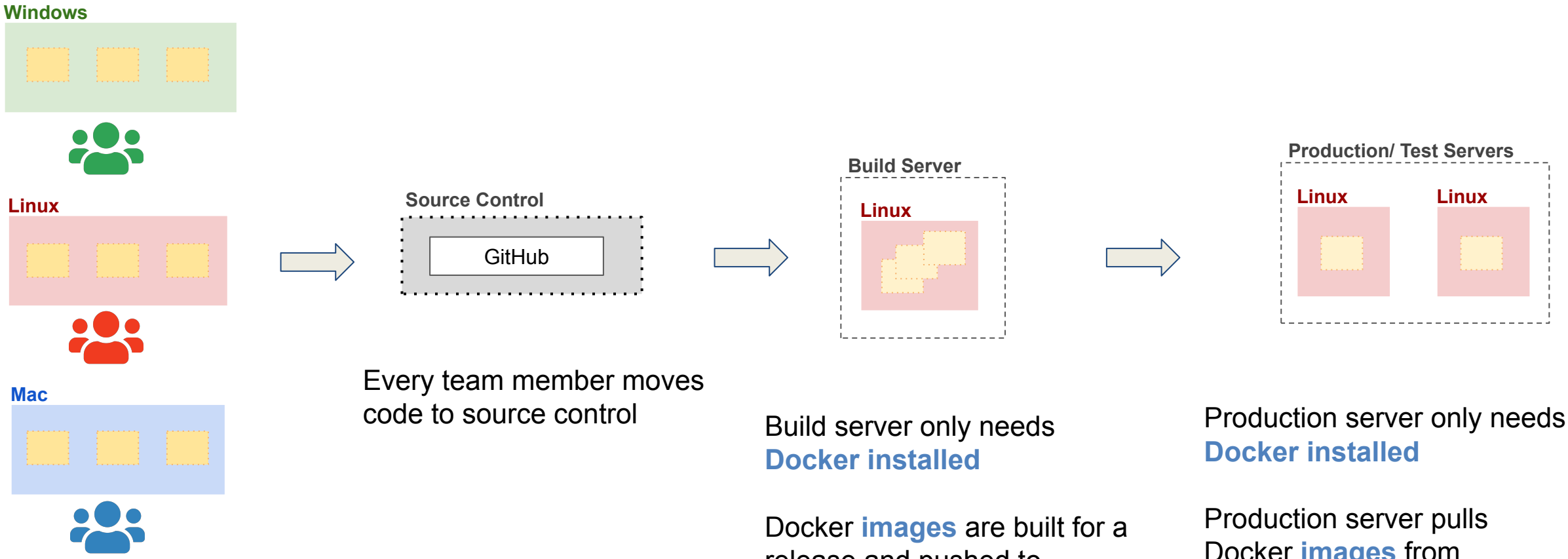
Build server needs to be **installed** with all required softwares/frameworks

Production build is performed by pulling code from source control

Production server needs to be **installed** with all required softwares/frameworks

Production server will be different OS version than development machines

Software Development Workflow (with Docker)



Development machines only needs **Docker installed**

Containers need to be setup only once

Every team member moves code to source control

Build server only needs **Docker installed**

Docker **images** are built for a release and pushed to **container registry**

ghcr.io for example.

Production server only needs **Docker installed**

Production server pulls Docker **images** from **container registry** and runs them

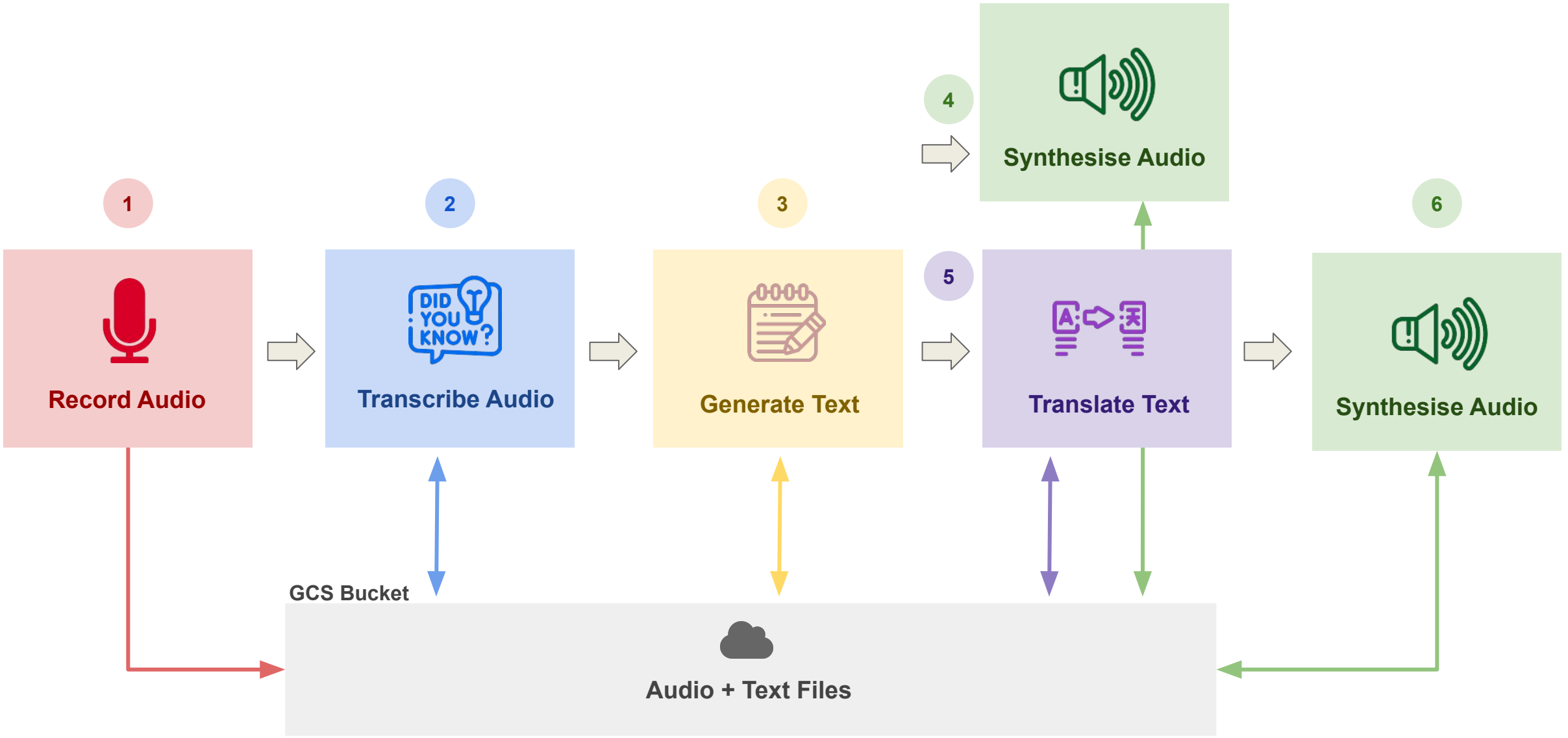
DEVCONTAINERS

- Create docker containers for development : use vscode or jupyterlab.
- Can be enabled on ANY repository on github
- Customize dev experience using "devcontainer.json" standard that builds on docker.
- Use devpod.sh on local machines. Switch between local dev / gpu dev etc based on docker containers locally or on any cloud.

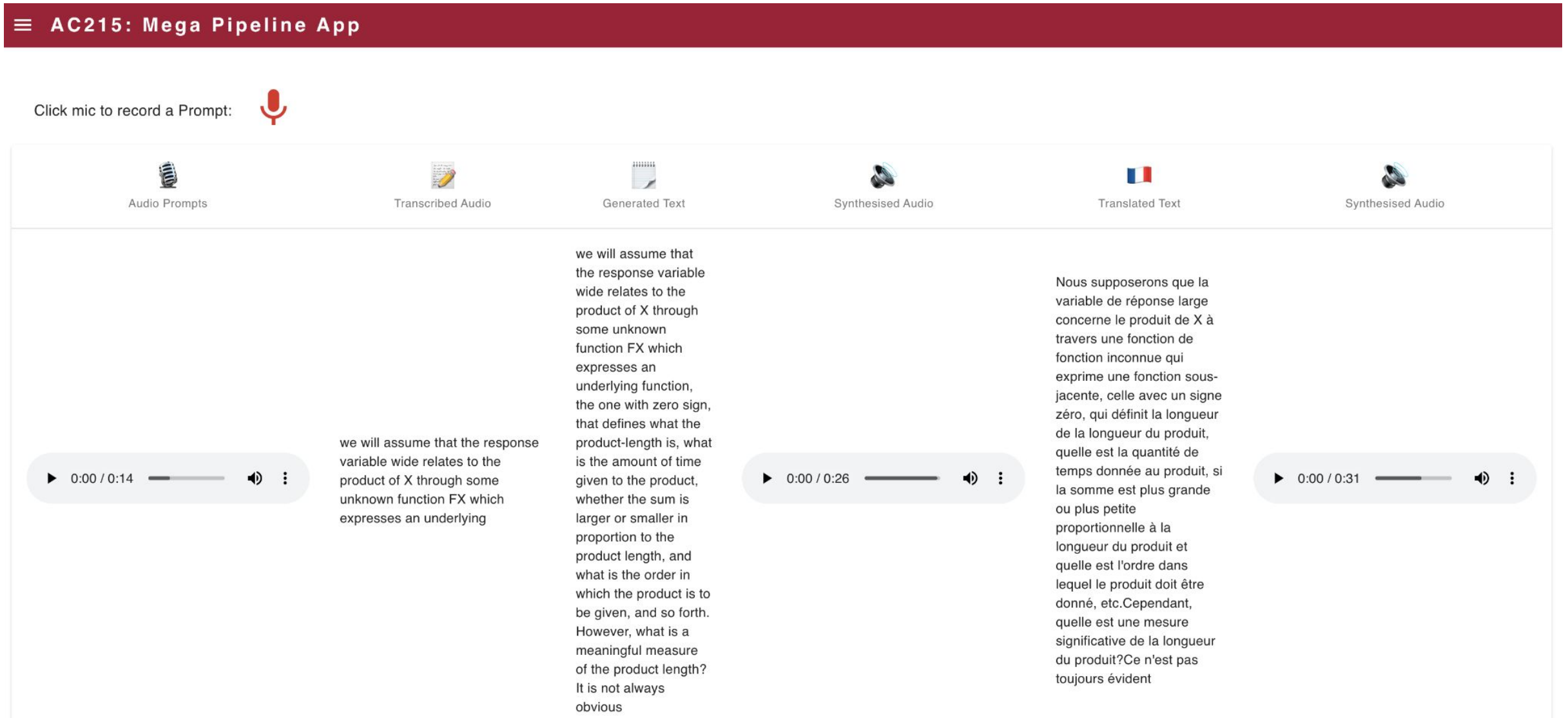
Outline

1. Recap: Review of Previous Material
2. Containers in Architecture: Microservices vs. Monolithic
3. **Implementing Containers as Microservices**

Tutorial - Building the Mega Pipeline App



Tutorial - Building the Mega Pipeline App



Tutorial - Building the Mega Pipeline App

- App: https://ai5-mega-pipeline.dlops.io/
- Teams
  - 📝Team A transcribe_audio:
  - 🗒️Team B generate_text:
  - 🔊Team C synthesis_audio_en:
  - 🇫🇷Team D translate_text:
  - 🔊Team E synthesis_audio:
- Instructions: https://github.com/dlops-io/mega-pipeline

THANK YOU