Lecture 4: Containers II

AC215

Pavlos Protopapas
SEAS/Harvard

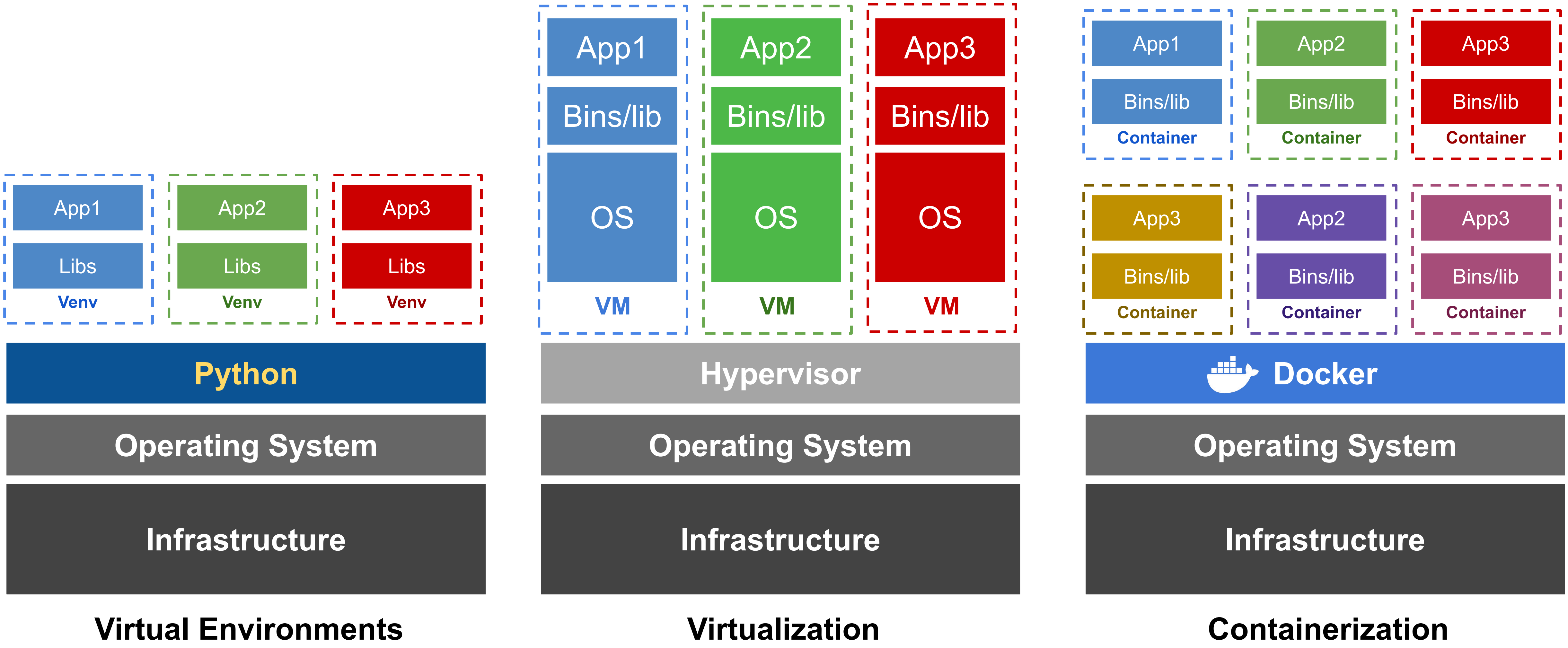
Outline

1. Recap: Review of Previous Material
2. Containers in Architecture: Microservices vs. Monolithic
3. Implementing Containers as Microservices
4. Working with Containers Workflow
5. Containers: Pro-tips

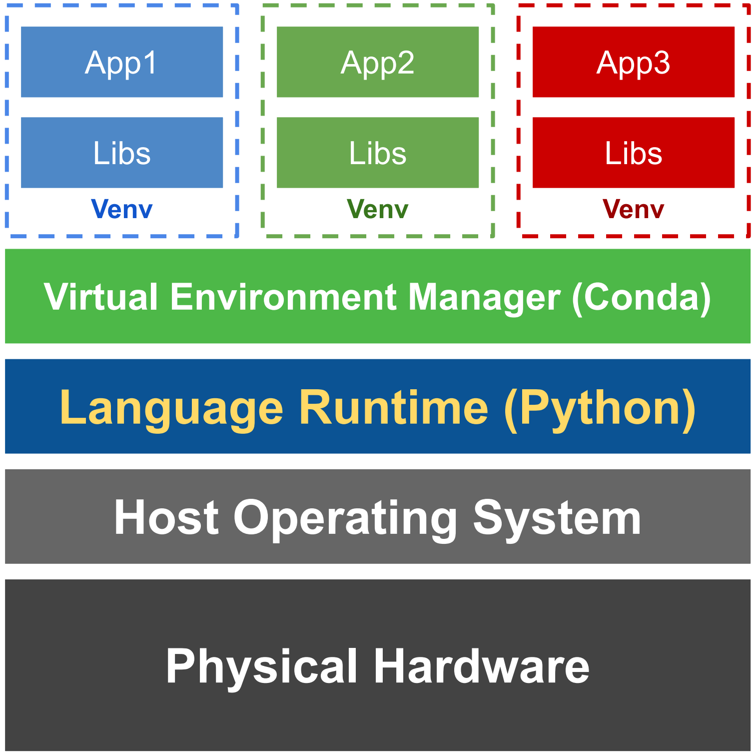
Outline

1. **Recap: Review of Previous Material**
2. Containers in Architecture: Microservices vs. Monolithic
3. Implementing Containers as Microservices
4. Working with Containers Workflow
5. Containers: Pro-tips

Recap: Environments vs Virtualization vs Containerization



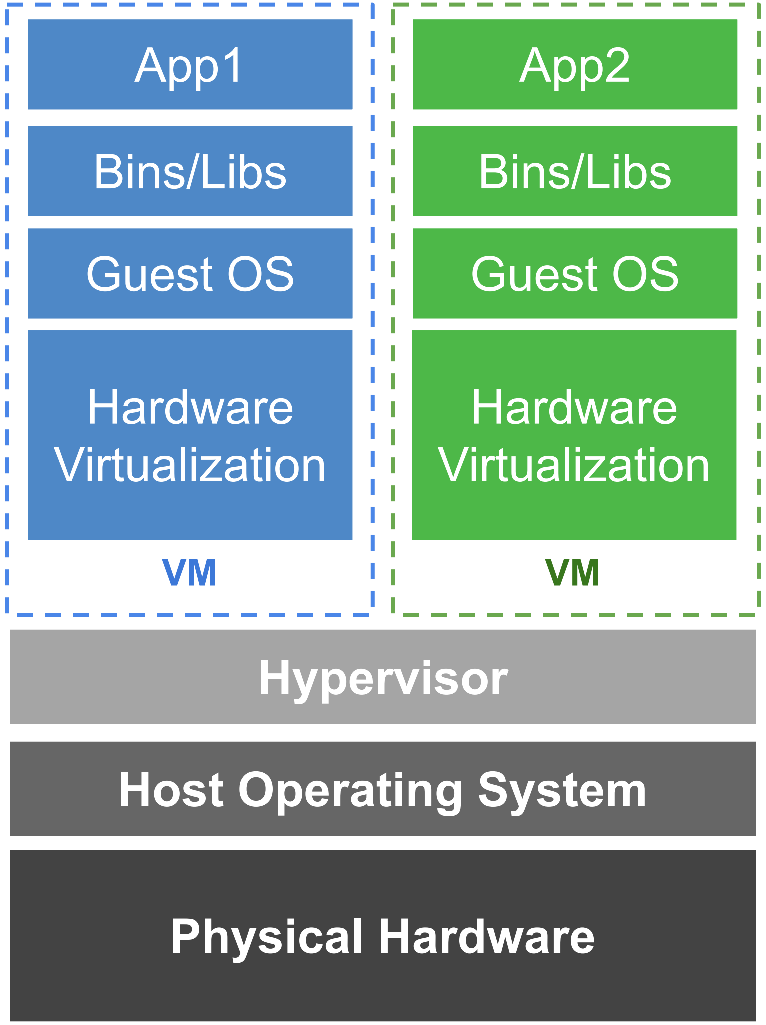
Environments



Virtual Environments

- **Dependency Isolation:** Virtual environments redirect dependencies to their own directories, avoiding system-wide installs.
- **No Kernel Isolation:** Unlike VMs or containers, they don't isolate the kernel.
- **Resource Efficiency:** Without an OS or kernel, virtual environments are lightweight and resource-efficient.
- **Filesystem Access:** Files written within a virtual environment can be accessed from other environments, as there's no filesystem isolation.

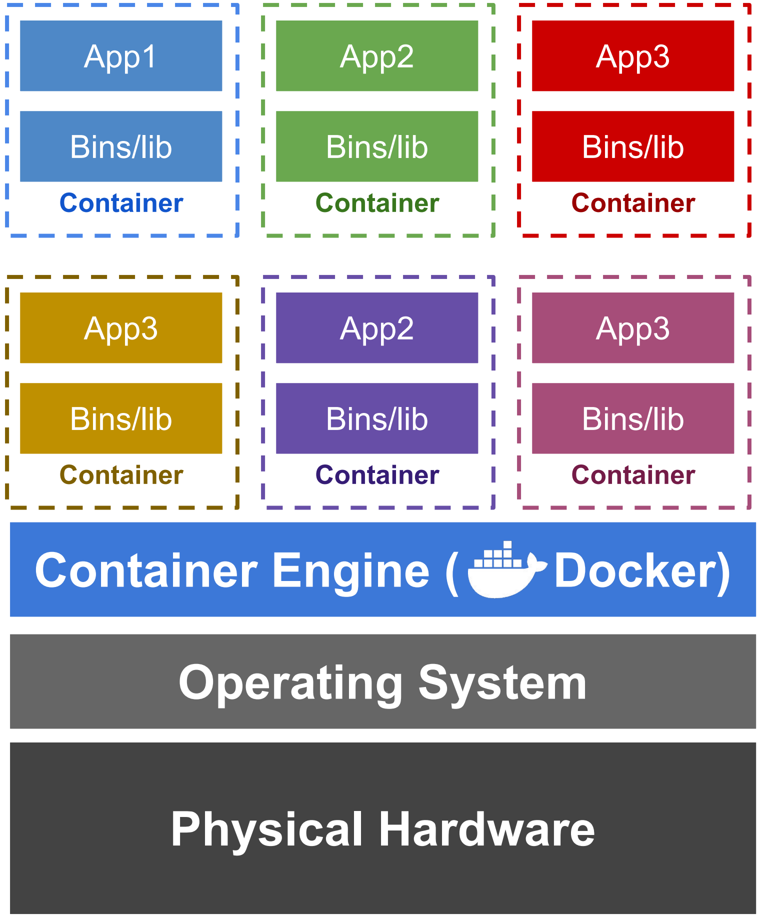
Virtualization (Virtual Machines)



Virtualization

- **CPU Virtualization:** Virtual CPUs are mapped to physical cores, but hypervisor management adds some overhead.
- **Emulated Devices:** VMs use virtual devices (CPUs, network adapters, disks) translated by the hypervisor to real hardware.
- **Full OS:** Each VM runs its own guest OS with independent kernel and user spaces, but this reduces efficiency.
- **Resource Allocation:** RAM, CPU, and disk space are often allocated in fixed blocks, limiting flexibility in resource usage.

Containerization



Containerization

- **Namespaces:** Containers isolate processes and resources, making them act like independent systems. For example, PID namespaces separate process IDs, and mount namespaces provide unique file systems.
- **Cgroups:** These limit CPU, memory, and IO usage for each container, ensuring efficient resource use.
- **Process Virtualization:** Namespaces and cgroups work together to isolate and control processes.
- **Shared Kernel:** Containers use the host's OS kernel but have their own files, making them lightweight and efficient.
- **Direct Access:** Containers interact with host resources directly, reducing overhead compared to VMs.

Outline

1. Recap: Review of Previous Material
2. **Containers in Architecture: Microservices vs. Monolithic**
3. Implementing Containers as Microservices
4. Working with Containers Workflow
5. Containers: Pro-tips

Why use Containers?

Conceptual Scenario

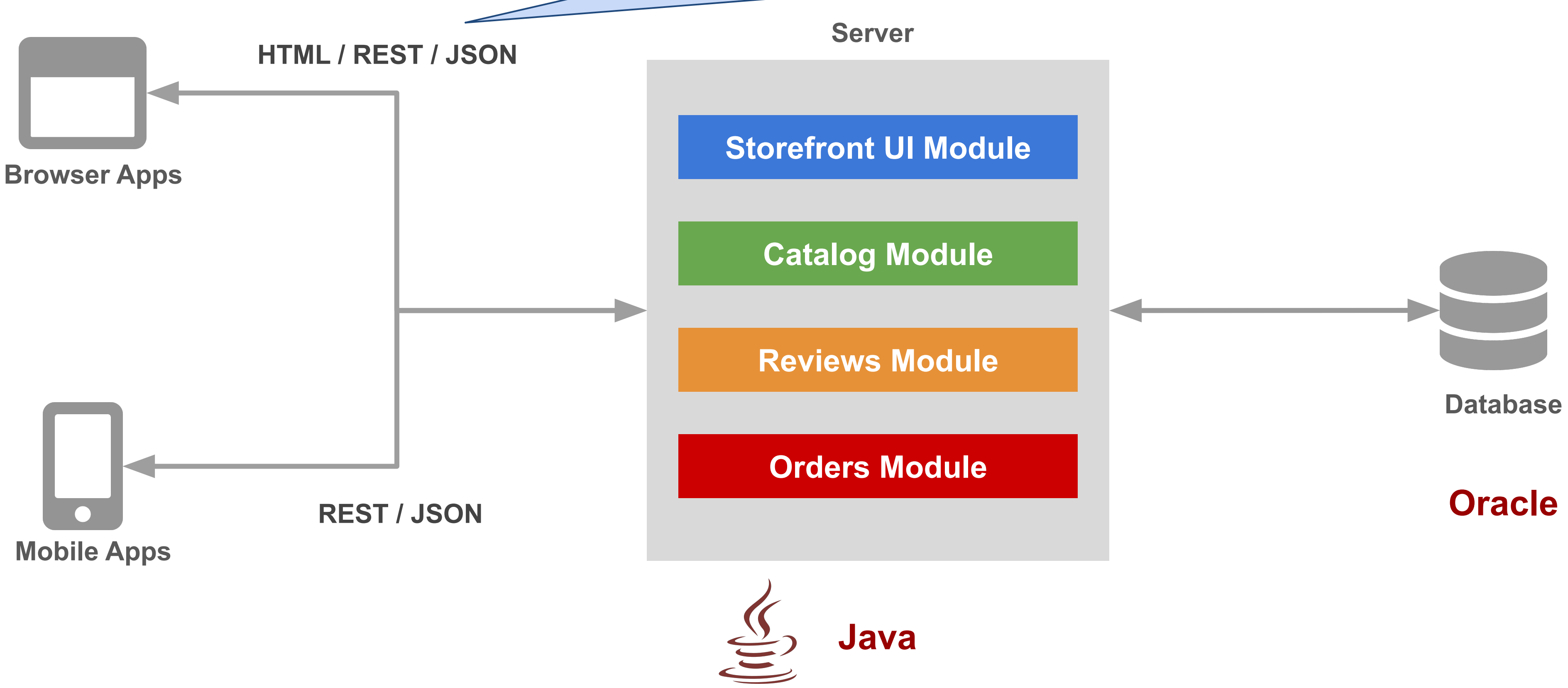
- Picture building an application, such as an online cheese store.

Traditional Approach

- Traditionality you would build this using a Monolithic Architecture.

Monolithic Architecture

HTML is HTML.
JSON is a data format for communication between clients and APIs.
REST is the protocol of communication with the API.



Monolithic Architecture - **Advantages**

Simplicity in Development:

Most tools and IDEs natively support monolithic applications.

Ease of Deployment:

All components bundled into a single, unified package.

Scalability:

Easier to scale by replicating the entire application as a whole (horizontal scaling).

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.

Onboarding Challenges:

New users need to familiarize themselves with the entire codebase.

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

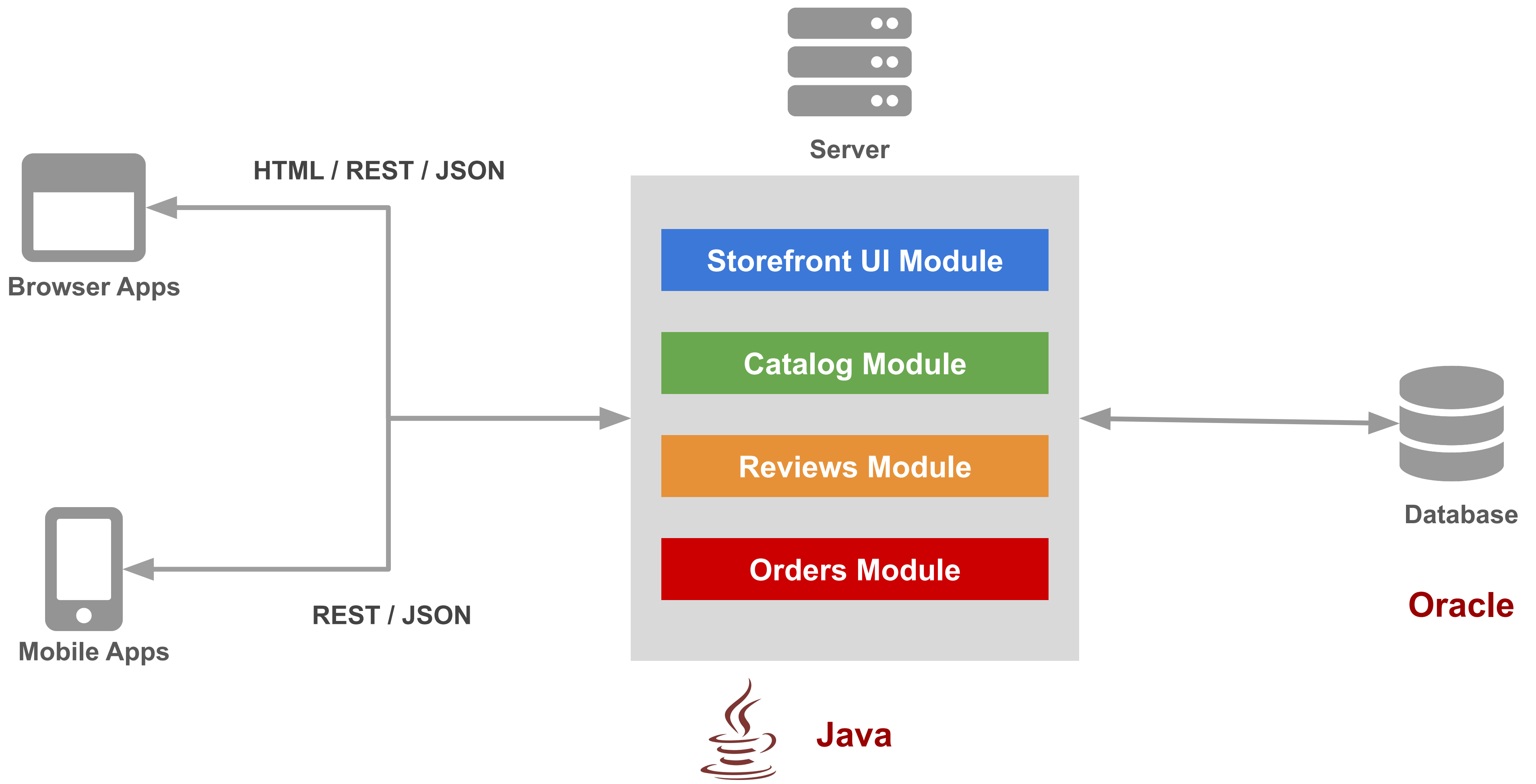
Today

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

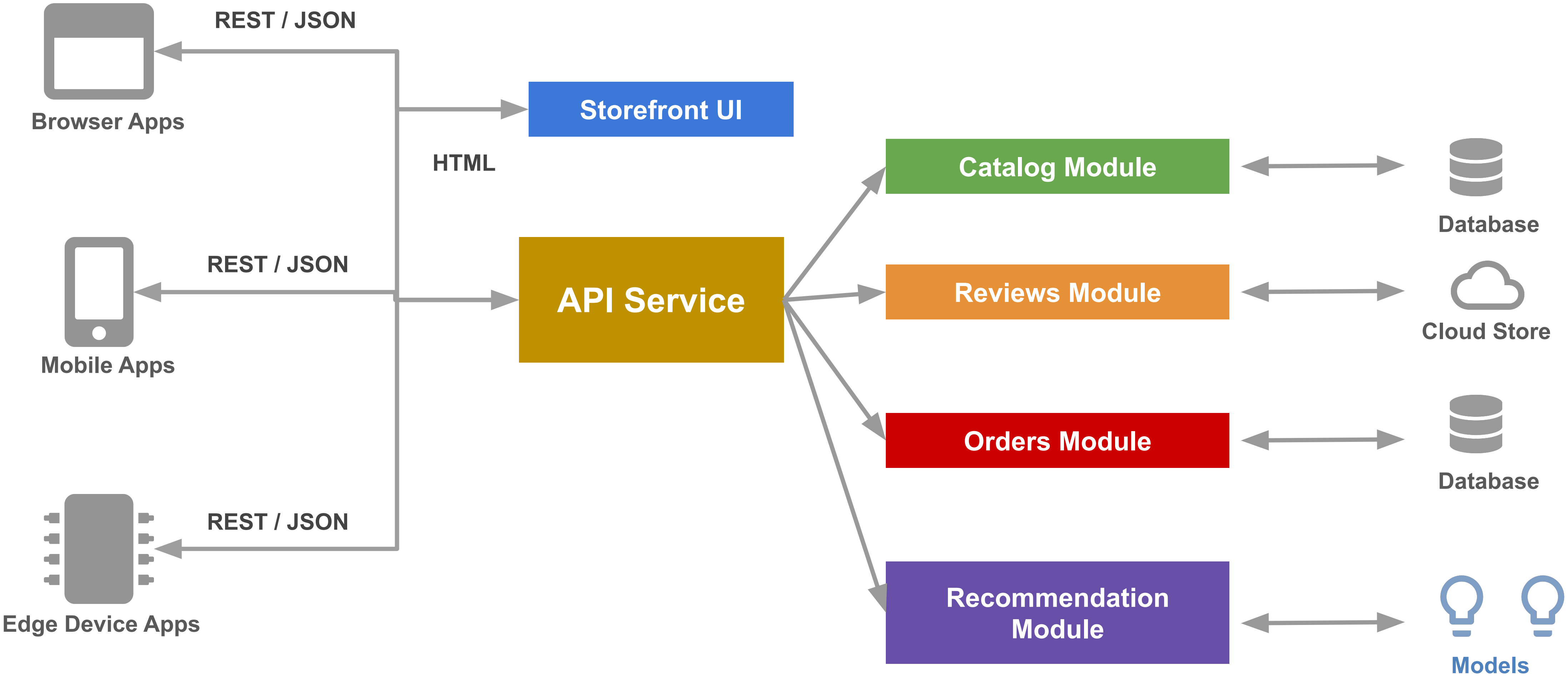
Data Science

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 and Deployment Complexity:

Using multiple technologies across components can complicate both development and deployment, as managing diverse dependencies requires a more intricate setup.

Scaling Concerns:

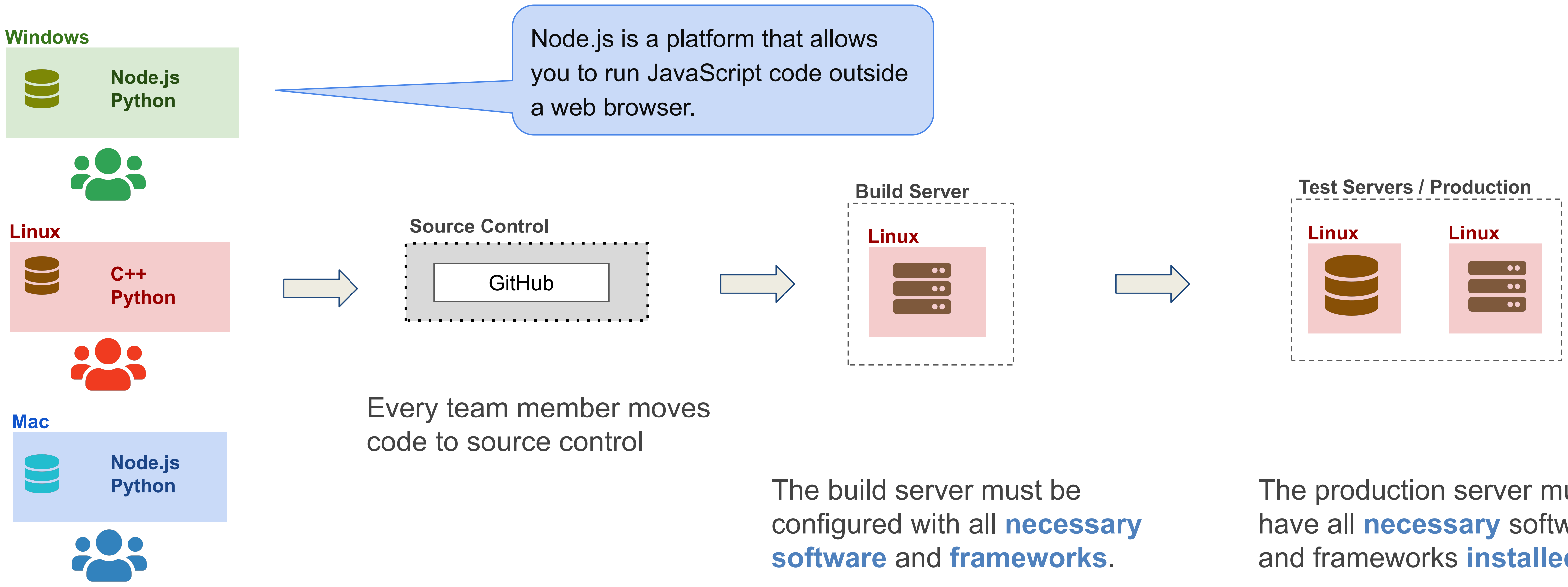
Scaling the entire application can be intricate due to disparate components.

Docker + Kubernetes

Why use Containers?

- 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

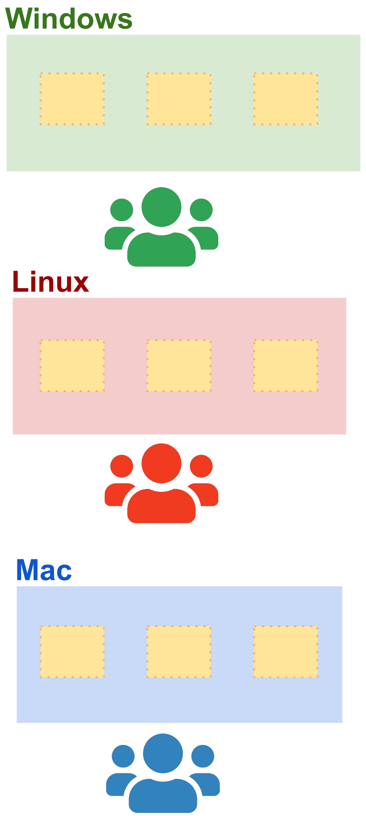
The build server must be configured with all **necessary software** and **frameworks**.

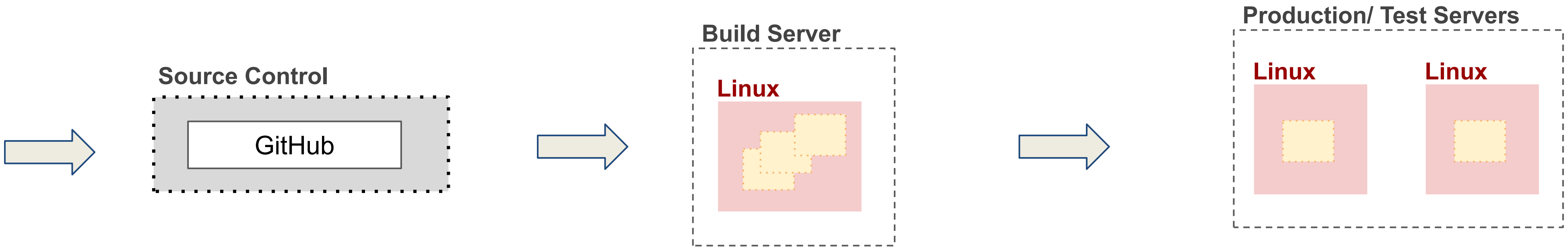
The production build process involves pulling the code from the source control system.

The production server must have all **necessary** software and frameworks **installed**.

It will also run on a different OS version compared to the 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 .

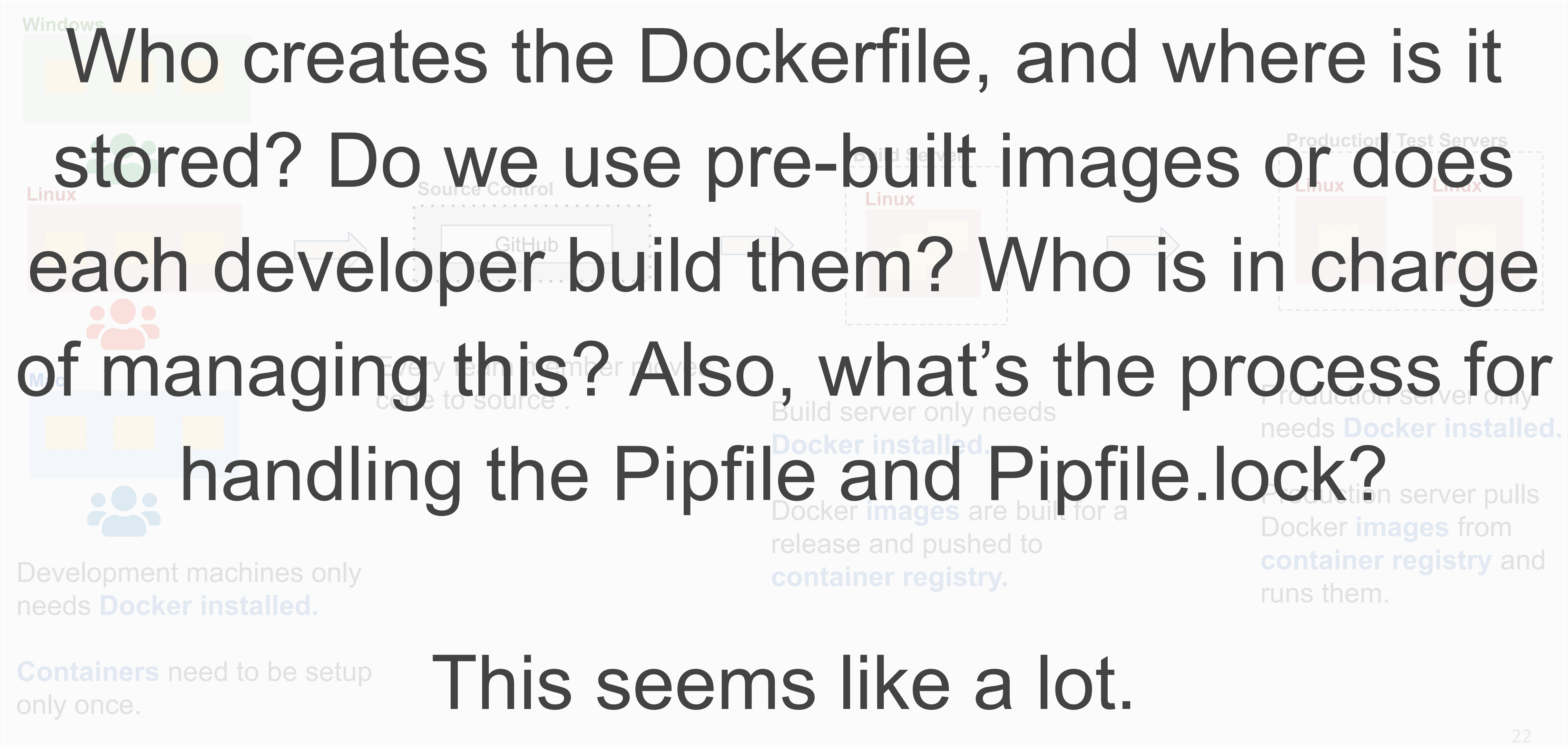
Build server only needs **Docker installed.**

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

Production server only needs **Docker installed.**

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

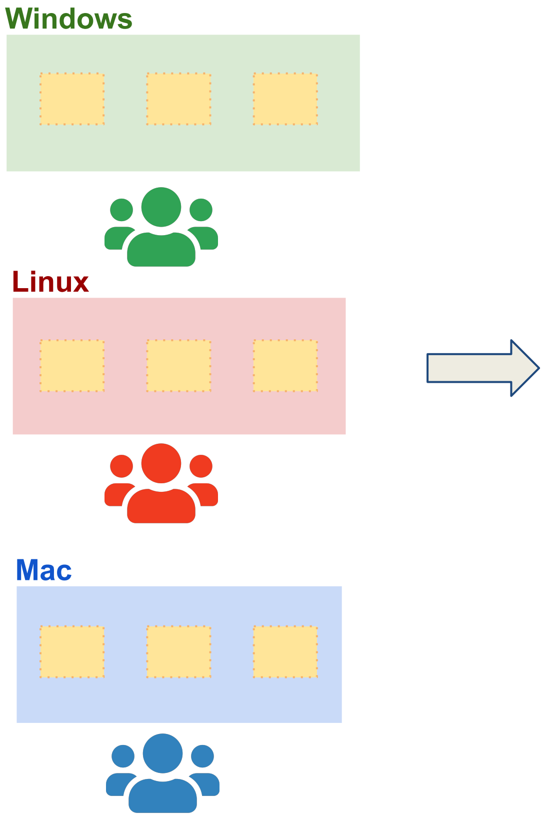
Software Development Workflow (with Docker)



Who creates the Dockerfile, and where is it stored? Do we use pre-built images or does each developer build them? Who is in charge of managing this? Also, what’s the process for handling the Pipfile and Pipfile.lock?

This seems like a lot.

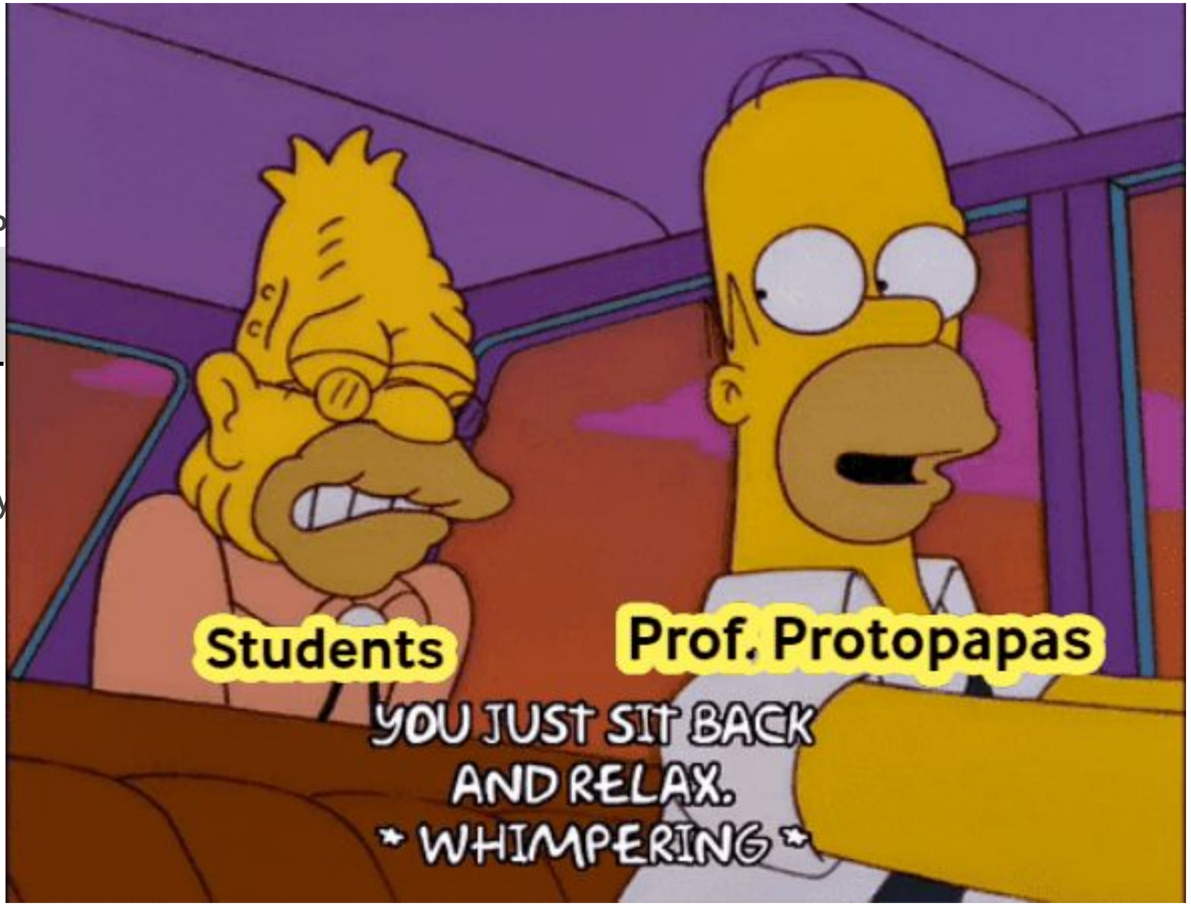
Software Development Workflow (with Docker)

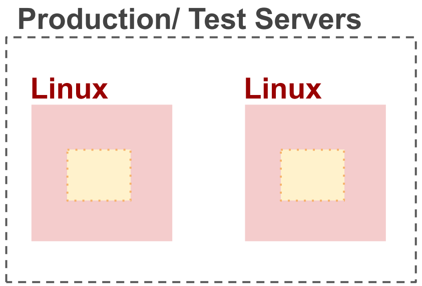


Development machines only needs **Docker installed.**

Containers need to be setup only once.

Every
code





Production server only needs **Docker installed.**

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

Outline

1. Recap: Review of Previous Material
2. Containers in Architecture: Microservices vs. Monolithic
3. **Implementing Containers as Microservices**
4. Working with Containers Workflow
5. Containers: Pro-tips

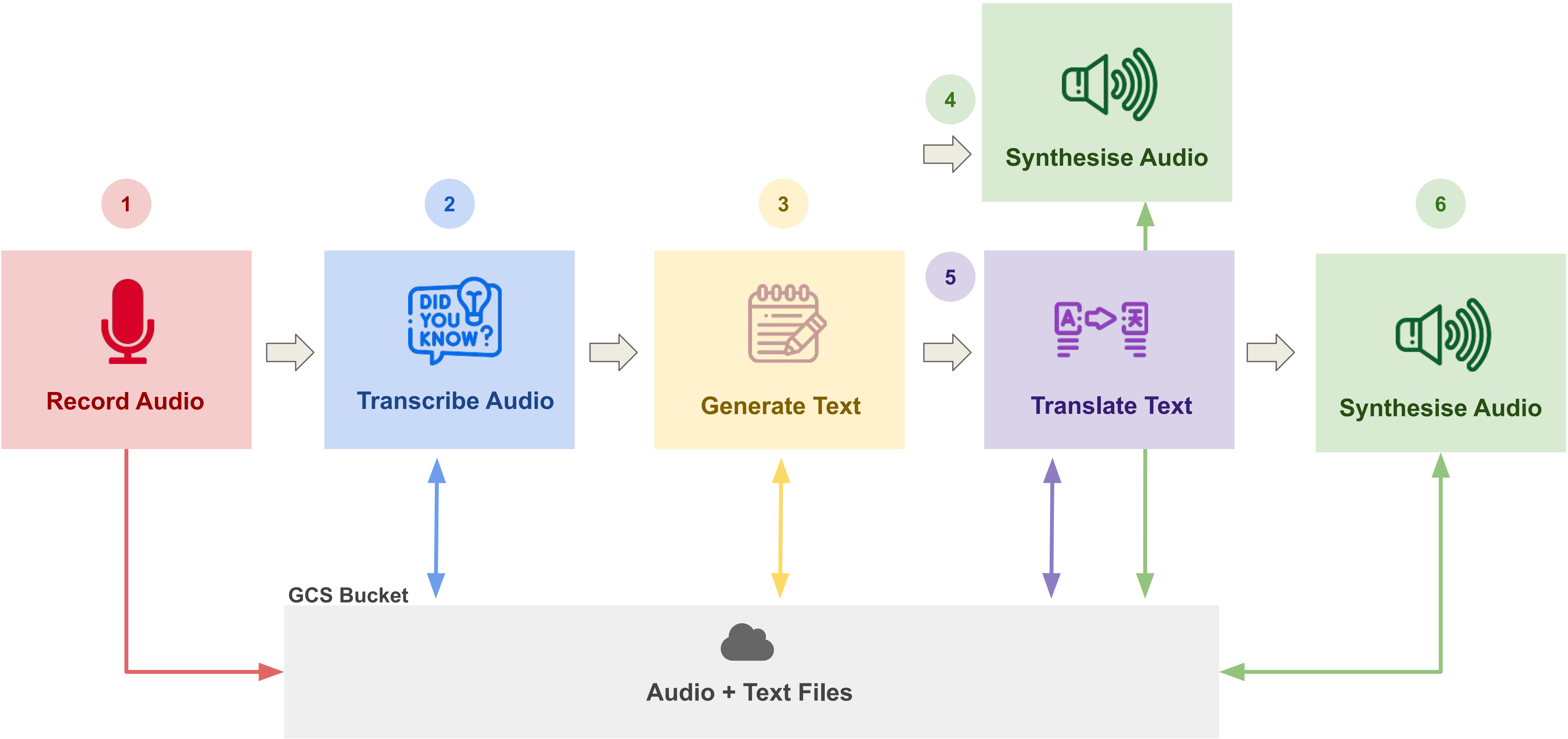
Tutorial (T5) - Building the Mega Pipeline App

One of *Formaggio.me*'s goals is to create a podcast on various cheese-related topics. After recording the podcast, we plan to transcribe the audio, use a language model to correct grammar and enhance the text, and then generate audio that will be made available to our users. Remember, we aim to reach an audience all over the world, so the podcast will be translated into various languages and synthesized into audio.

The goal here is to simulate a realistic development scenario where each component will be developed by different teams and containerized.

BONUS: You can use elevenlabs API to generate text with Pavlos' voice or your own voice.

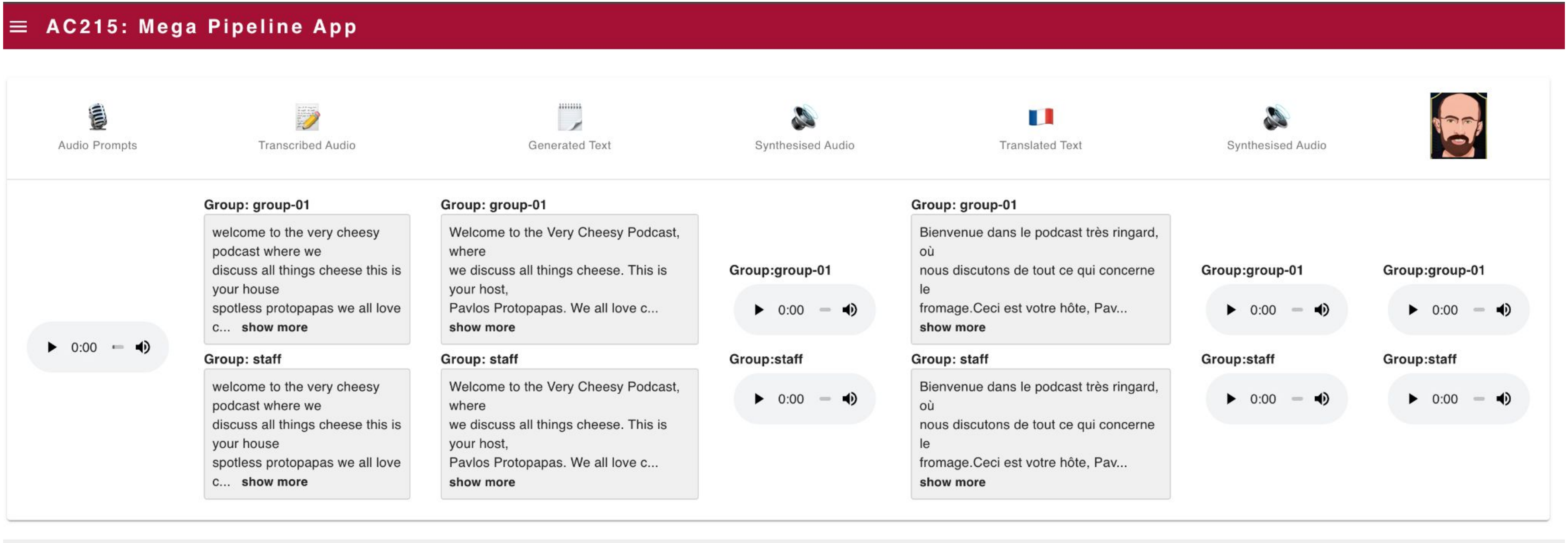
Tutorial (T5) - Building the Mega Pipeline App



Tutorial (T5) - Team Challenge!

- We'll form teams of 5, and each team must complete 5 tasks.
- The first team to finish all tasks will win a special prize!
- Don't worry, every team that completes the tasks will also get a reward!
- The rewards are a surprise—so give it your best effort and have fun!

Tutorial (T5) - Building the Mega Pipeline App



GCP Authentication Methods

Before we start let us review how do we authenticate to different services/accounts and APIs

OAuth 2.0:

For user-driven authentication and access.

Service Account (part of IAM - Identity and Access Management- in GCP):

For server-to-server interactions requiring automation and high control without user intervention.

API Key:

For lightweight, less secure access to APIs, use cautiously.

Default Service Accounts (part of IAM in GCP):

For Compute Engine, Kubernetes Engine, and App Engine with predefined permissions.

Workload Identity Federation:

For external identities to access GCP securely.

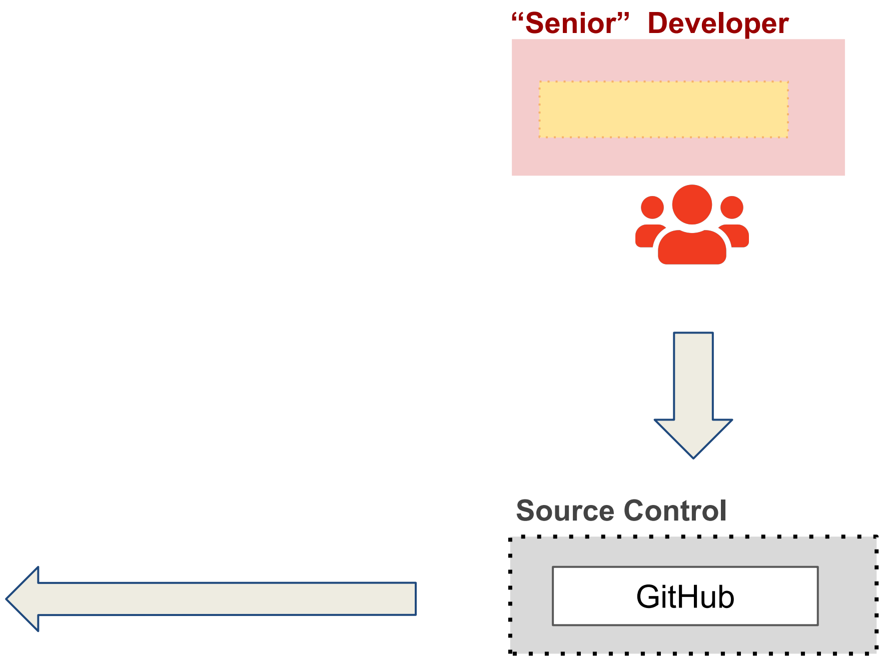
Tutorial (T5) - Building the Mega Pipeline App

- App: https://ac215-mega-pipeline.dlops.io/
- Teams
  - 📝Task A transcribe_audio:
  - 🗒️Task B generate_text:
  - 🔊Task C synthesis_audio_en:
  - 🇫🇷Task D translate_text:
  - 🔊Task E synthesis_audio:
- Instructions: https://github.com/dlops-io/mega-pipeline

Outline

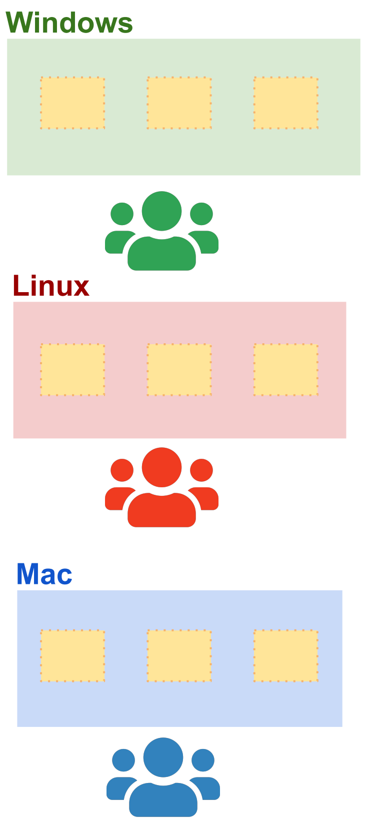
1. Recap: Review of Previous Material
2. Containers in Architecture: Microservices vs. Monolithic
3. Implementing Containers as Microservices
4. **Working with Containers Workflow**
5. Containers: Pro-tips

Workflow with Docker: Scenario 1 (early stages of development)



A senior developer is responsible for creating the Dockerfile, along with the initial Pipfile and Pipfile.lock.

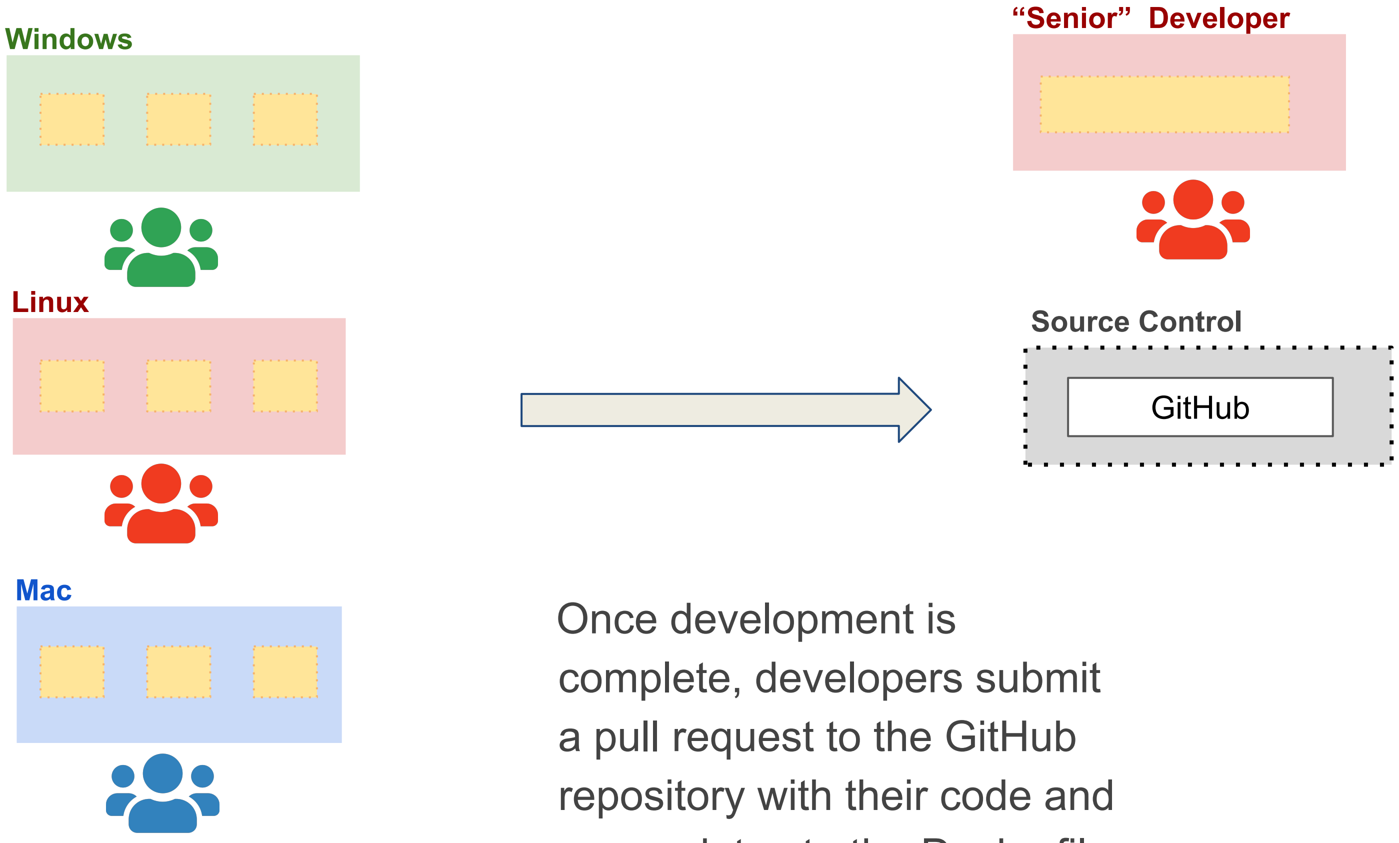
Push Dockerfile and Pipfiles to the GitHub repository



Developers pull the Dockerfile and Pipfiles from the GitHub repository.

- Build the Docker image and run containers.
- Develop within the containers, with the option to install packages (Pipfiles will be changed) and modify the Dockerfile if needed, then rebuild.

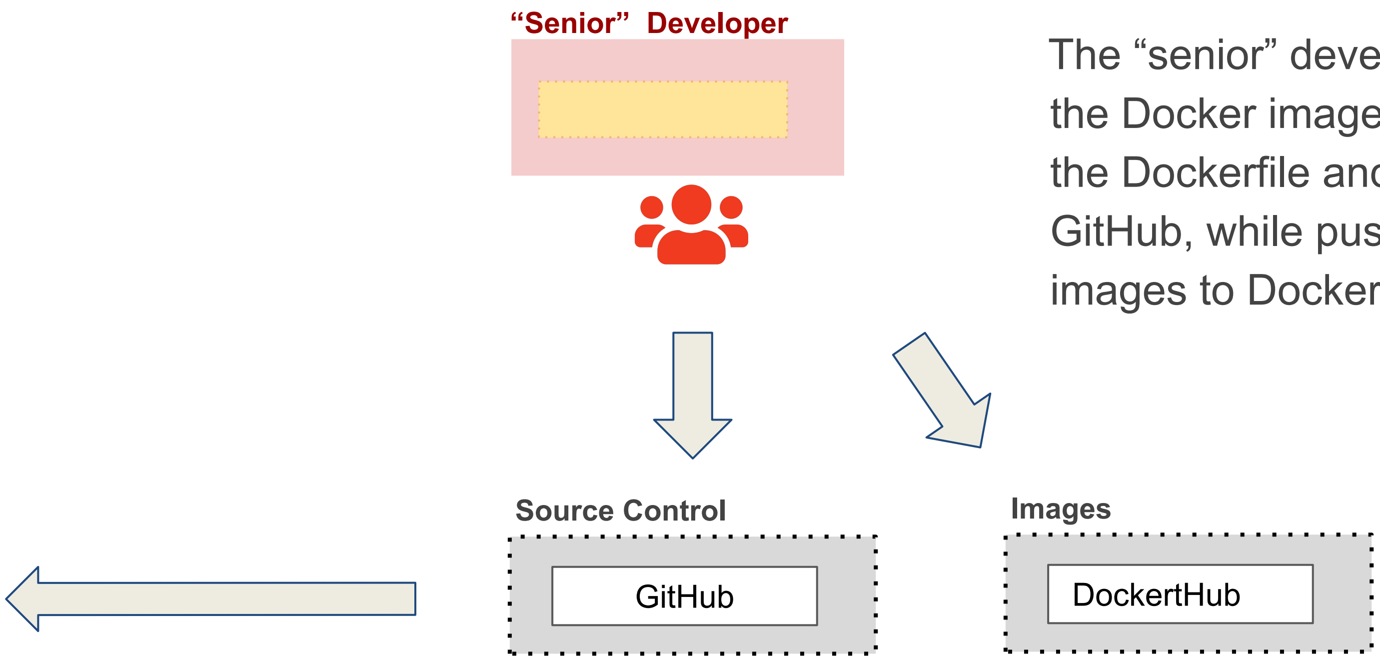
Workflow with Docker: Scenario 1 (early stages of development)



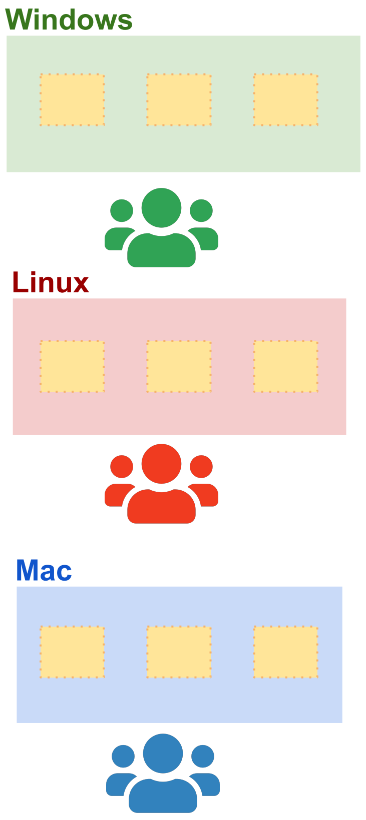
The senior developer reviews and merges all pull requests, then consolidates the updates into a new Dockerfile and Pipfiles.

Once development is complete, developers submit a pull request to the GitHub repository with their code and any updates to the Dockerfile or Pipfiles.

Workflow with Docker: Scenario 2 (later stages of development)

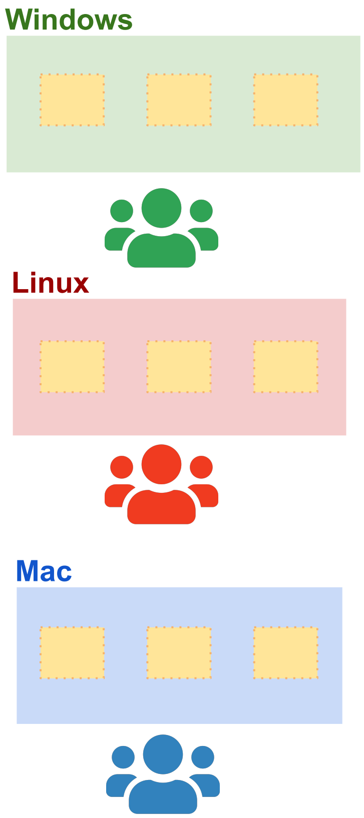


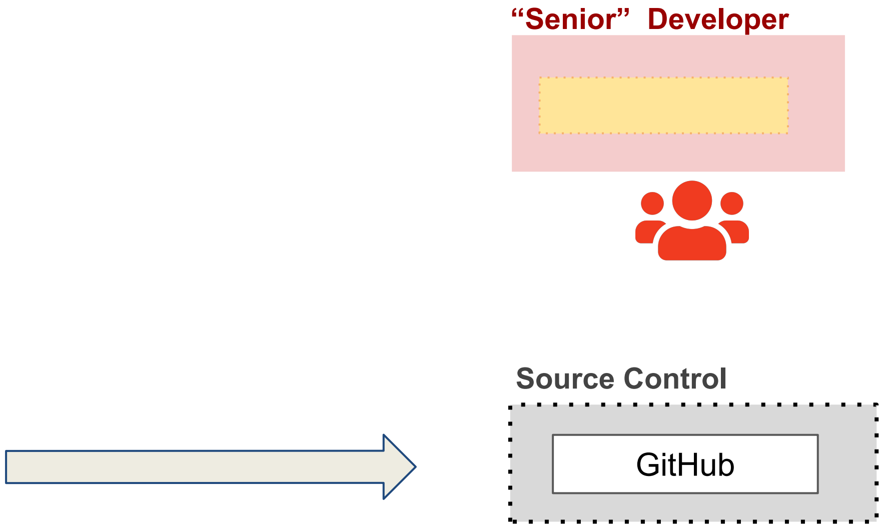
The "senior" developer creates the Docker images and pushes the Dockerfile and Pipfiles to GitHub, while pushing the images to DockerHub.



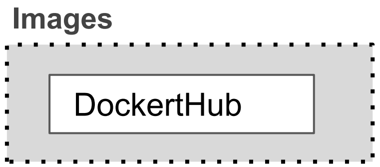
- Developers pull images from DockerHub and the Dockerfile/Pipfiles from GitHub.
- They run containers from the images and develop inside them.
- Rebuilding the images is only needed on rare occasions.

Workflow with Docker: Scenario 2 (later stages of development)





The senior developer reviews and merges all pull requests, then consolidates the updates into a new Dockerfile and Pipfiles and builds new images which are pushed to DockerHub.



- Developers submit a pull request to GitHub for their code.
- In some cases, they may also include changes to the Dockerfile or Pipfiles in the pull request.

Workflow with Docker: A Flexible Approach

Is there a “prefect” workflow?

No

So, how do we decide what to do?

Clear communication and rules are essential. Each team can have its own workflow, based on the project and team needs.

Outline

1. Recap: Review of Previous Material
2. Containers in Architecture: Microservices vs. Monolithic
3. Implementing Containers as Microservices
4. Working with Containers Workflow
5. **Containers: Pro-tips**

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

Running commands during the build process can take significant disk space. For example, when **installing** and **building** packages, we **download** and generate, numerous files, which can substantially **increase** the **size** of the Docker image.

Question: Do we really need all of these files for our app?
Answer: Usually no! Usually not! We typically only need the executable and its runtime dependencies.

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

Pro Tip 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.

Name of previous image you want to get file(s) from

Dockerfile command

```
COPY --from=builder /bin/hello .
```

Location of file(s) from previous image

Location where file(s) from previous image should live on new image

Multi-stage builds can

- decrease container size
- improve security
- leverage different base images for each stage while preserving only the final image.

[multi-stage builds]

Pro Tip 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
```

Pro Tip 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

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

Another way is to use buildx

```
# Check if buildx is available
docker buildx version

# Create a new builder instance named multi_platform.
docker buildx create --use --name multi_platform

# Create and load image for linux/arm64
docker buildx build --platform linux/arm64 -t print-platform-arm64 --load .
# Create and load image for linux/amd64
docker buildx build --platform linux/amd64 -t print-platform-amd64 --load .
# Run image container for linux/arm64
docker run --rm print-platform-arm64
# Run image container for linux/amd64
docker run --rm print-platform-amd64
```

multi-platform images

Pro Tip 3: **Docker Compose**

Docker Compose is a tool for defining and running multi-container Docker applications.

With Docker Compose, you can use a simple YAML file to configure your application's services, networks, and volumes, and then start everything with a single command.

We go through docker compose in more details next lecture.

Logistics/Reminders

- If you have formed groups - please update group info sheet

- Please fill out survey - https://canvas.harvard.edu/courses/136127/assignments/866239

(Survey responses have been updated)

- Office Hours details here - https://edstem.org/us/courses/58478/discussion/5229430


PRETTY SHOWS
Delete! Delete!

Now check: https://formaggio.me/