

Lecture 4: Containers II

AC215

Pavlos Protopapas
SEAS/Harvard



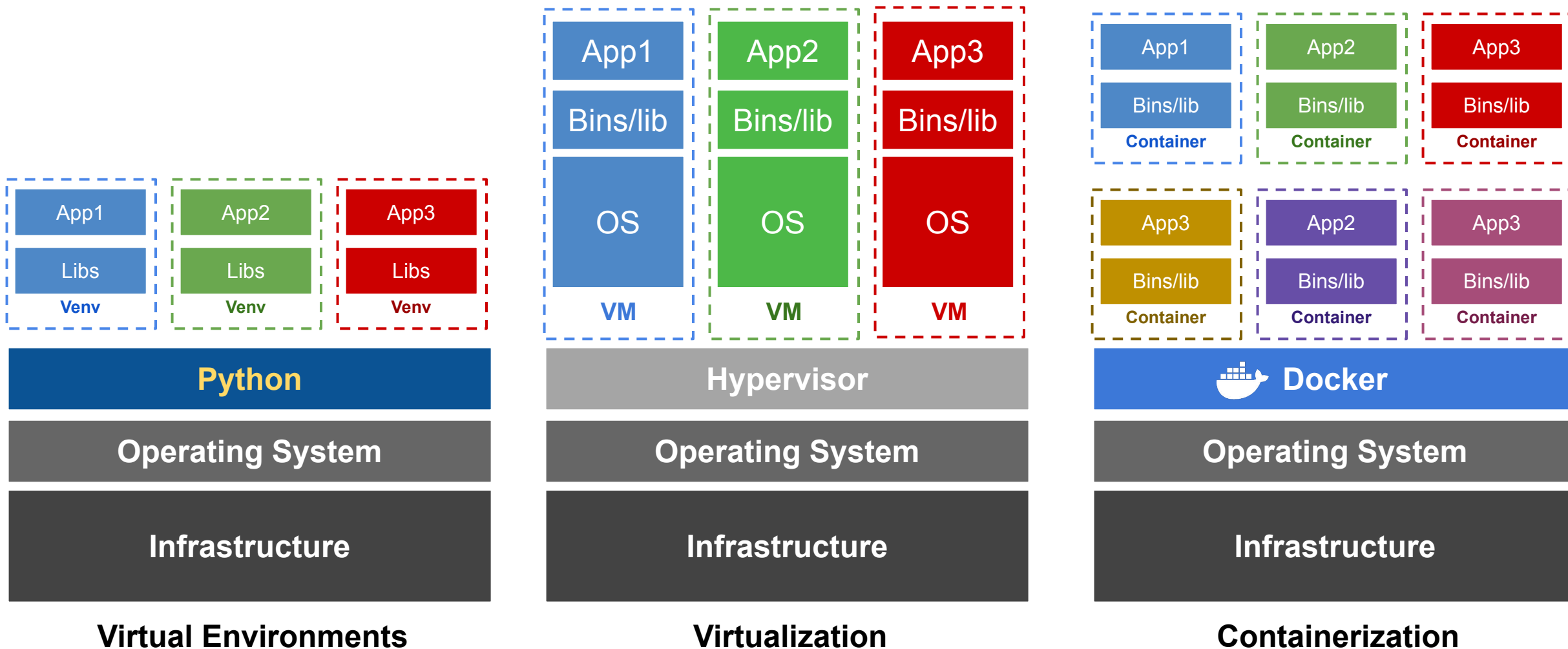
Outline

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

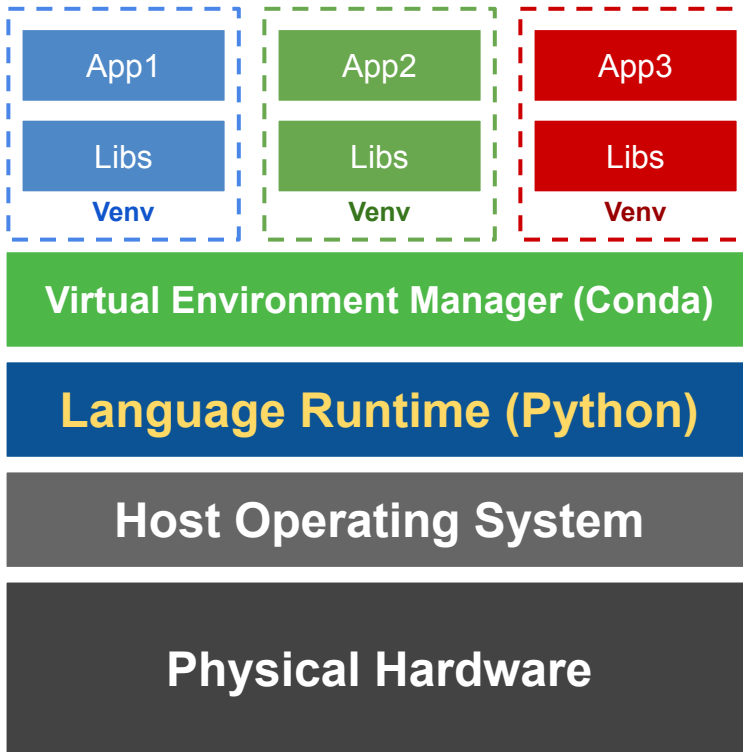
Outline

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

Recap: Environments vs Virtualization vs Containerization



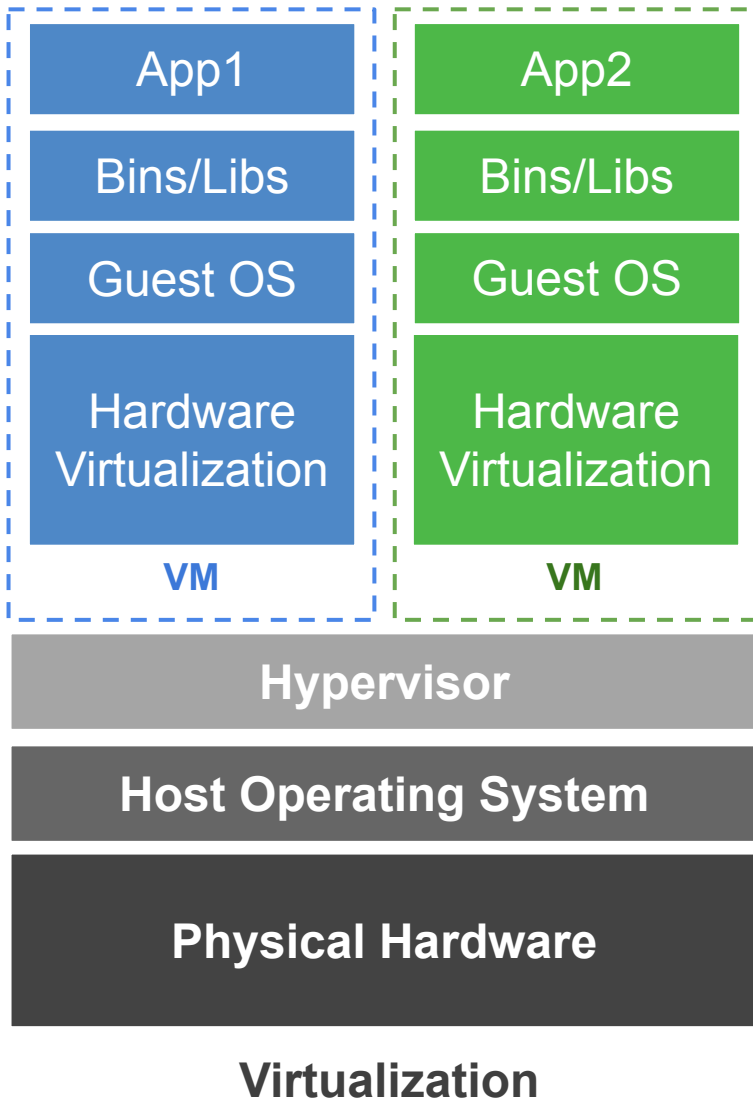
Environments



Virtual Environments

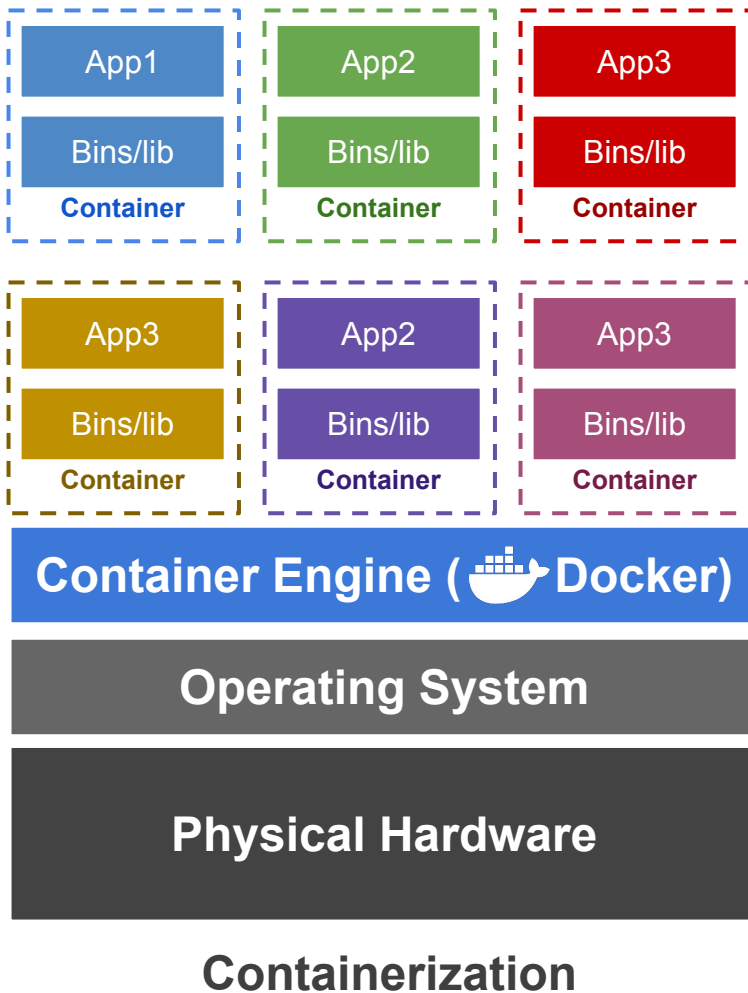
- **Dependency Isolation:** Virtual environments modify the PATH and other environment variables so that the dependencies are loaded from the environment's directory, rather than system-wide directories
- **No Kernel Isolation:** Unlike VMs and containers, virtual environments don't provide any kernel 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.

Virtualization (Virtual Machines)



- **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



- **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.

Pro Tips 1: multi-stage builds

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?

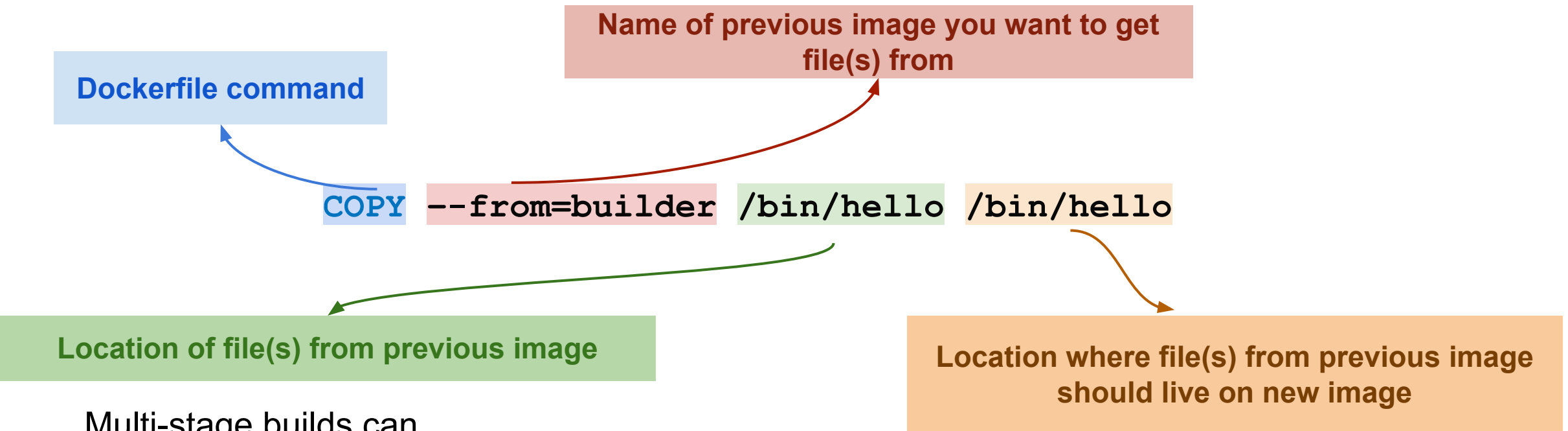
Answer: Usually no! We just need the executable and its runtime dependencies.

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
```

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 no specific 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?

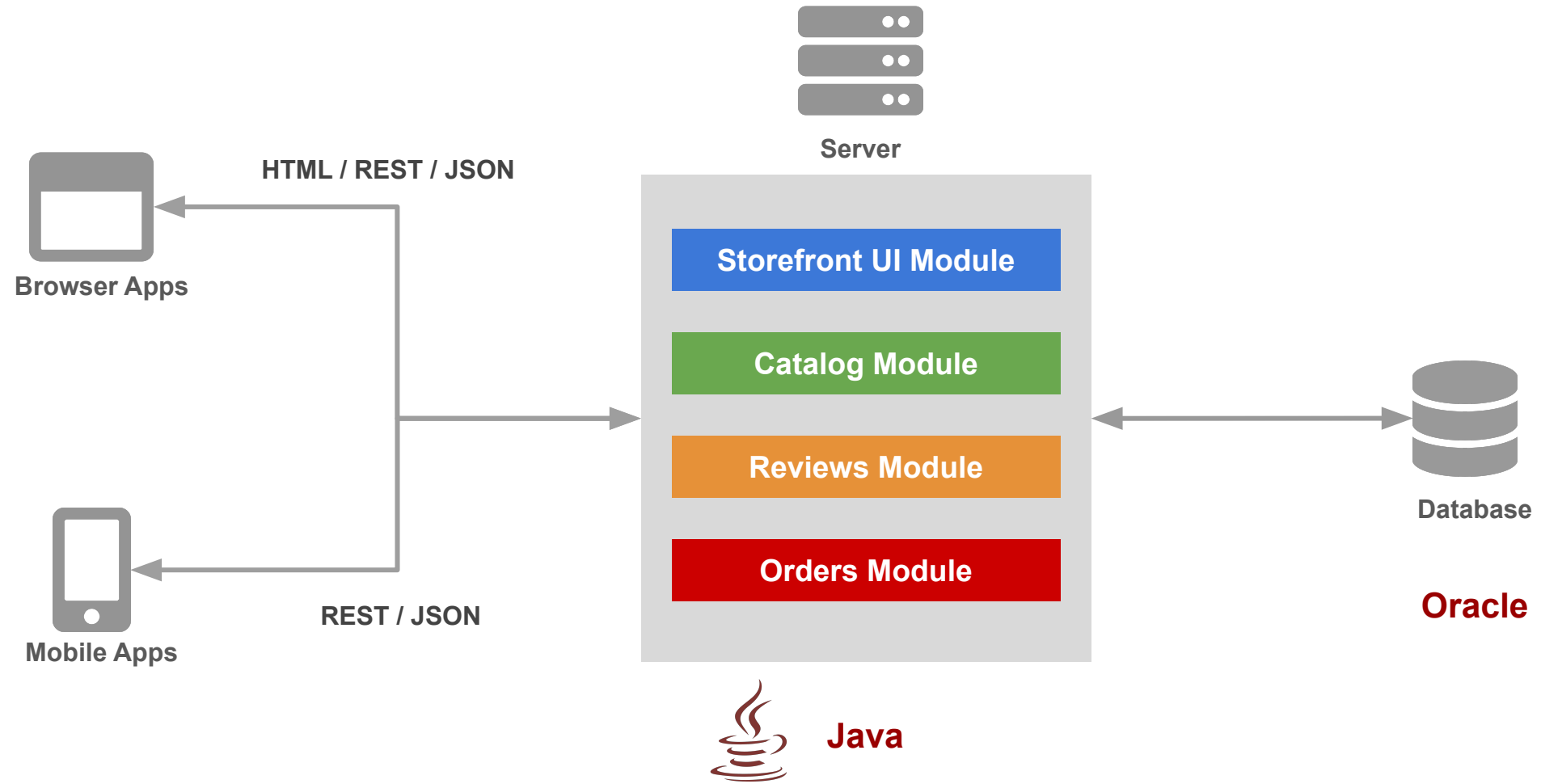
Conceptual Scenario

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

Traditional Approach

- Traditionally 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

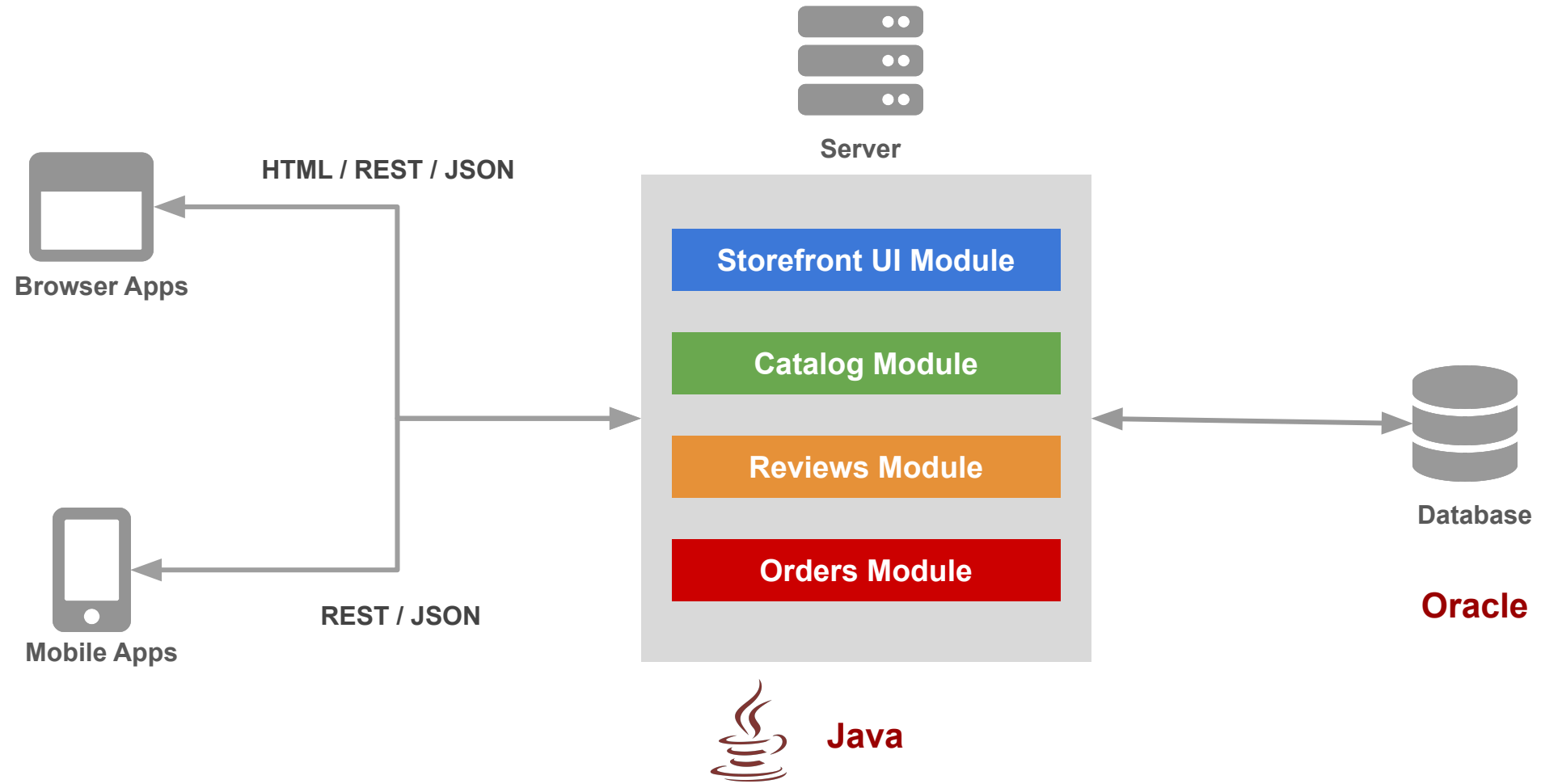
Today

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

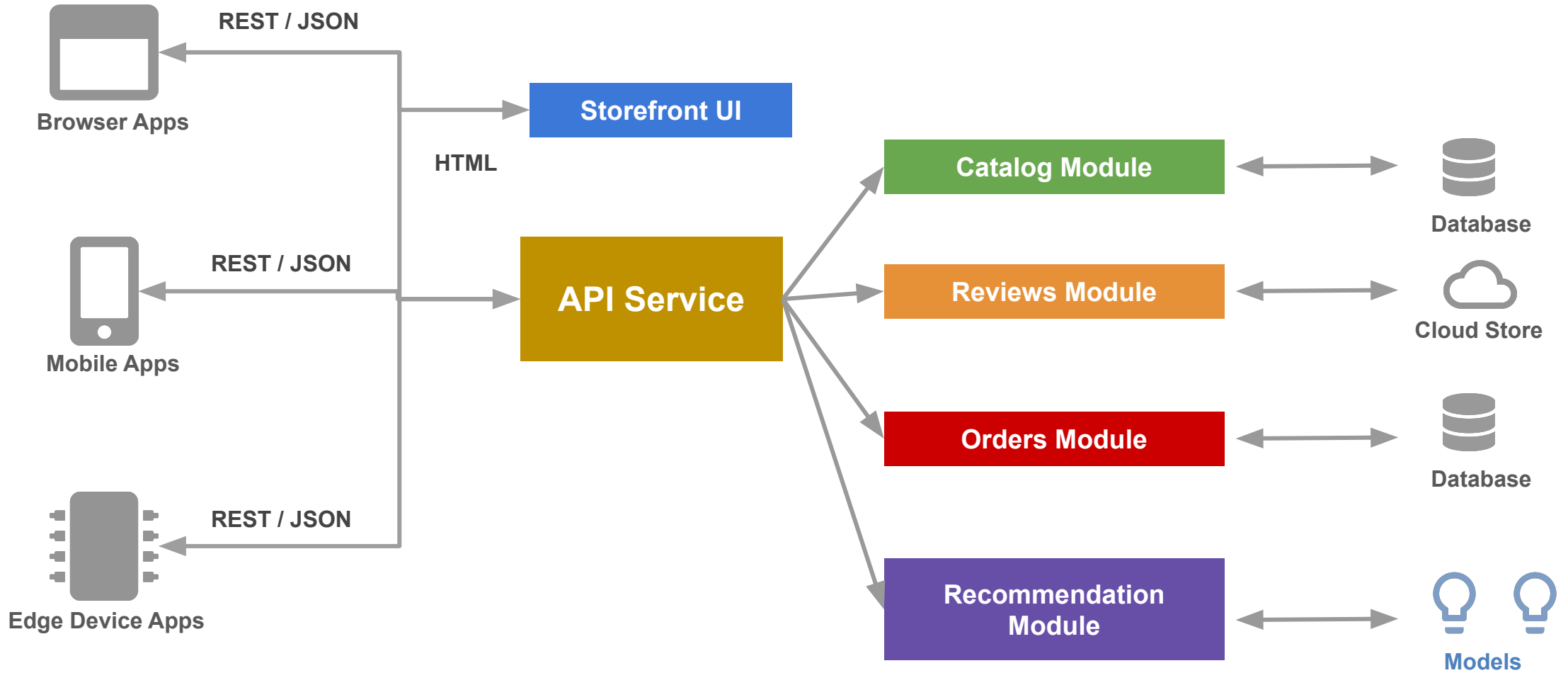
Data Science

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.

Docker + Kubernetes

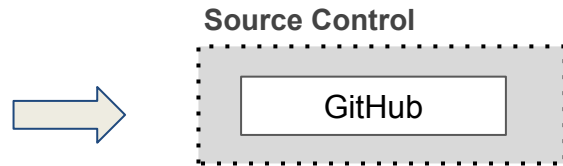
Why use Containers?

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

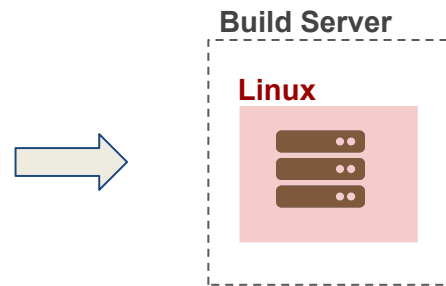
Software Development Workflow (no Docker)



OS Specific **installation** in every developer machine

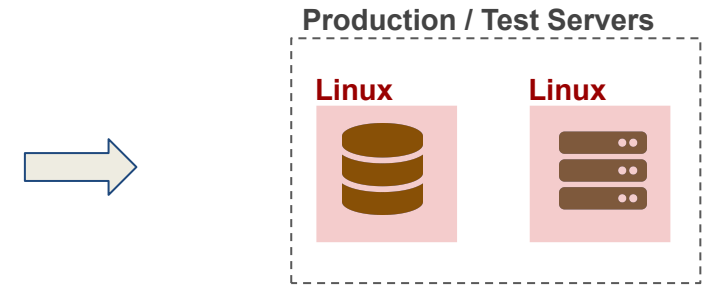


Every team member moves code to source control



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

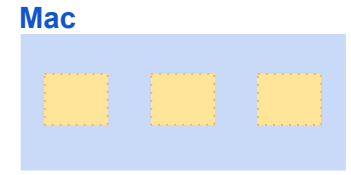
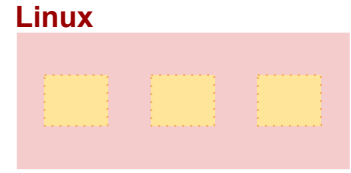
Production build is performed by pulling code from source control



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

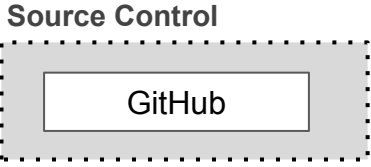
Production server will be different OS version than development machines

Software Development Workflow (with Docker)

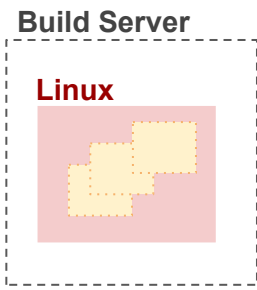


Development machines only needs **Docker installed**

Containers need to be setup only once

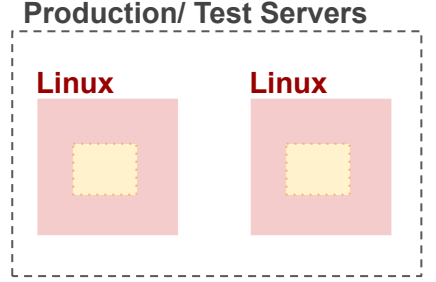


Every team member moves code to source control



Build server only needs **Docker installed**

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



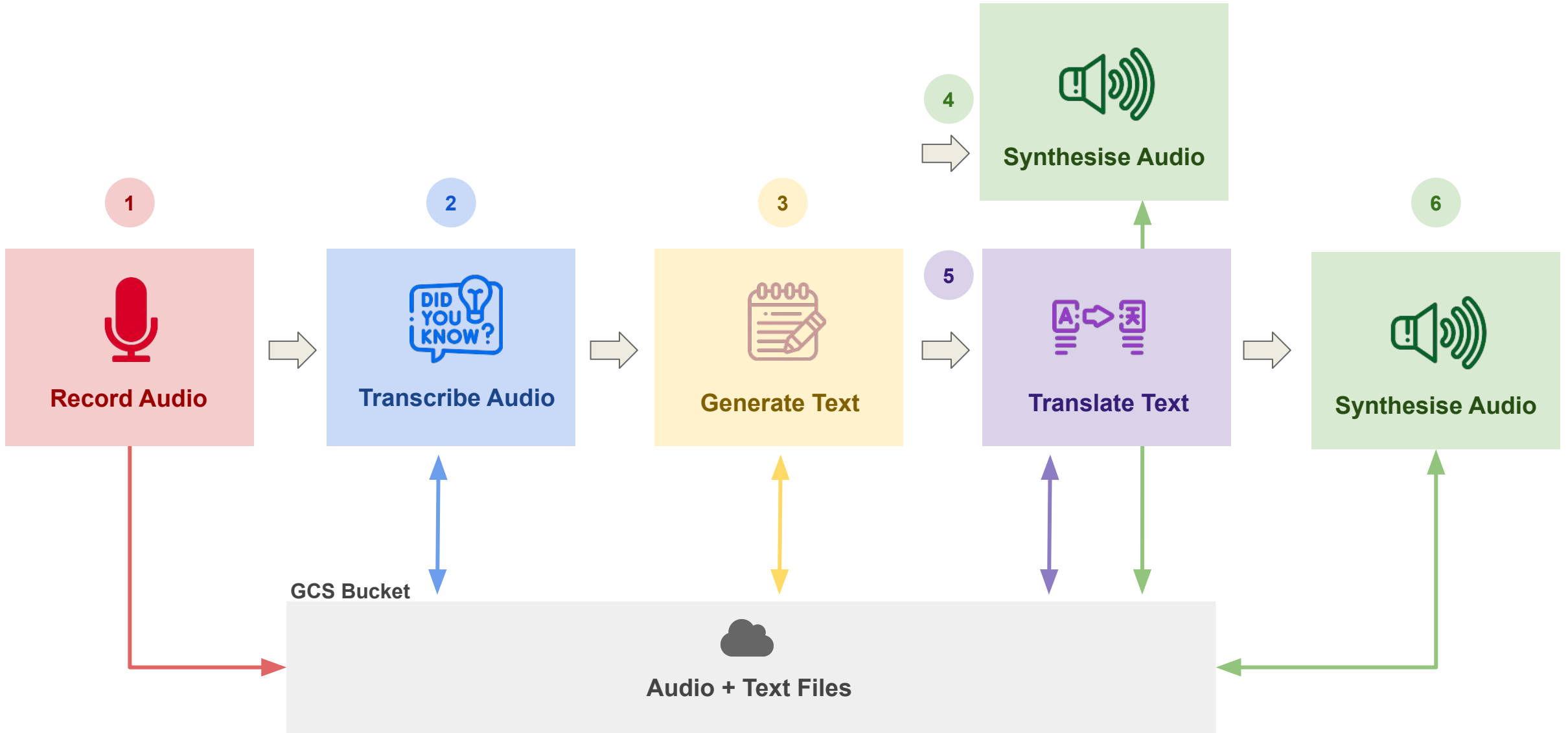
Production server only needs **Docker installed**

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**

Tutorial - Building the Mega Pipeline App



Tutorial - Building the Mega Pipeline App

≡ AC215: Mega Pipeline App

Click mic to record a Prompt:



Audio Prompts



Transcribed Audio



Generated Text



Synthesised Audio



Translated Text



Synthesised Audio

▶ 0:00 / 0:14

we will assume that the response variable wide relates to the product of X through some unknown function FX which expresses an underlying






we will assume that the response variable wide relates to the product of X through some unknown function FX which expresses an underlying function, the one with zero sign, that defines what the product-length is, what is the amount of time given to the product, whether the sum is larger or smaller in proportion to the product length, and what is the order in which the product is to be given, and so forth. However, what is a meaningful measure of the product length? It is not always obvious

▶ 0:00 / 0:26

Nous supposons que la variable de réponse large concerne le produit de X à travers une fonction de fonction inconnue qui exprime une fonction sous-jacente, celle avec un signe zéro, qui définit la longueur de la longueur du produit, quelle est la quantité de temps donnée au produit, si la somme est plus grande ou plus petite proportionnelle à la longueur du produit et quelle est l'ordre dans lequel le produit doit être donné, etc.Cependant, quelle est une mesure significative de la longueur du produit?Ce n'est pas toujours évident

▶ 0:00 / 0:31

Tutorial - Building the Mega Pipeline App

- App: <https://ac215-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