Lecture 3: Containers I

AC215

Pavlos Protopapas
SEAS/Harvard

Outline

1. Recap & Motivation
2. What is a Container
3. Why use Containers
4. How to use Containers

Outline

1. **Recap & Motivation**
2. What is a Container
3. Why use Containers
4. How to use Containers

Recap Virtual Machines: Pros and Cons

Pros

- Full Autonomy:
  Complete control over the operating system and applications, similar to a physical server.
- Very Secure:
  Isolated environment helps in minimizing the risk of system intrusion.
- Lower Cost:
  Can be more cost-effective for applications that need full OS functionality.
- Cloud Adoption:
  Offered by all major cloud providers for on-demand server instances.

Cons

- Resource Intensive:
  Consumes hardware resources from the host machine.
- Portability Issues:
  VMs are large in size, making them harder to move between systems.
- Overhead:
  Requires additional resources to run the hypervisor and manage multiple operating systems.

Recap: Virtual Environments

Pros

- Reproducible Research:
  Easy to replicate experiments and share research outcomes due to consistent environments.
- Explicit Dependencies:
  Clear listing of all required packages and versions, reducing ambiguity.
- Improved Engineering Collaboration:
  Team members can quickly set up the same environment, streamlining development.

Cons

- Difficulty in Setup:
  Initial setup can be complex, especially for those new to the concept.
- No Isolation from Host:
  Virtual environments share the host's operating system, leading to potential conflicts.
- OS Limitations:
  May not be compatible across different operating systems, requiring additional configuration.

Wish List

Automated Setup:

Automatically set up (installs) OS and extra libraries and set up the python environment.

Isolation:

Complete separation from the host machine, ensuring a consistent run-time environment.

Resource Efficiency:

Minimal use of CPU, Memory, and Disk resources, optimized for performance.

Quick Startups:

Near-instantaneous initialization, reducing time to deployment.

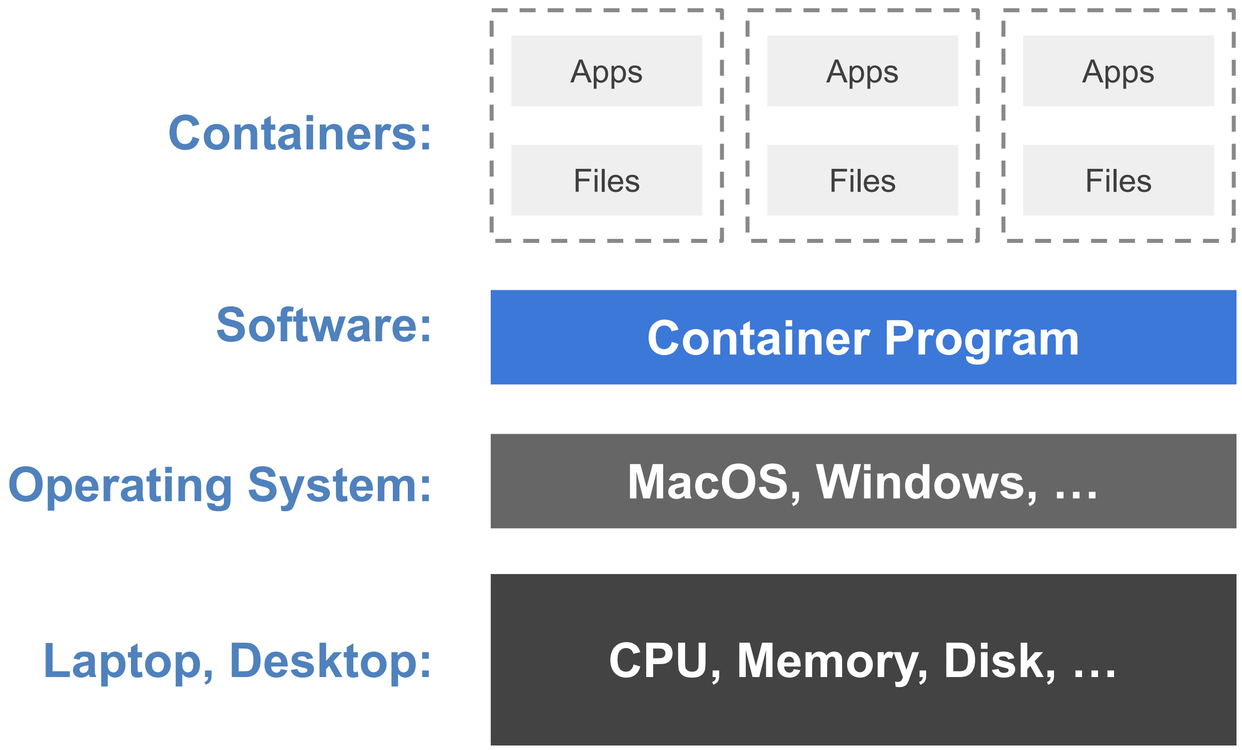
Containers

Outline

1. Recap & Motivation
2. **What is a Container**
3. Why use Containers
4. How to use Containers

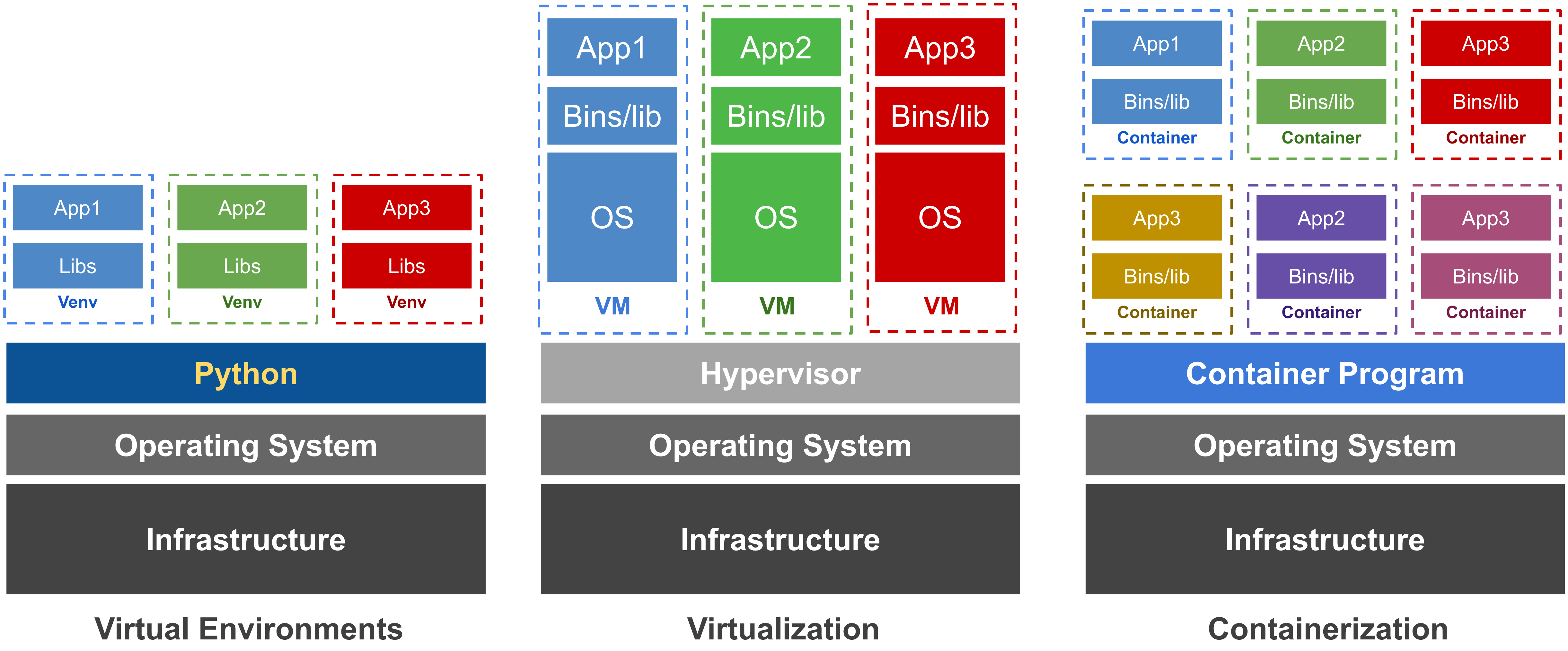
What is a CONTAINER

A container is a program that runs on your machine, essentially acting as a miniature computer within your main computer. It uses resources from the host machine (CPU, Memory, Disk, etc.) but behaves like its own operating system with an isolated file system and network.

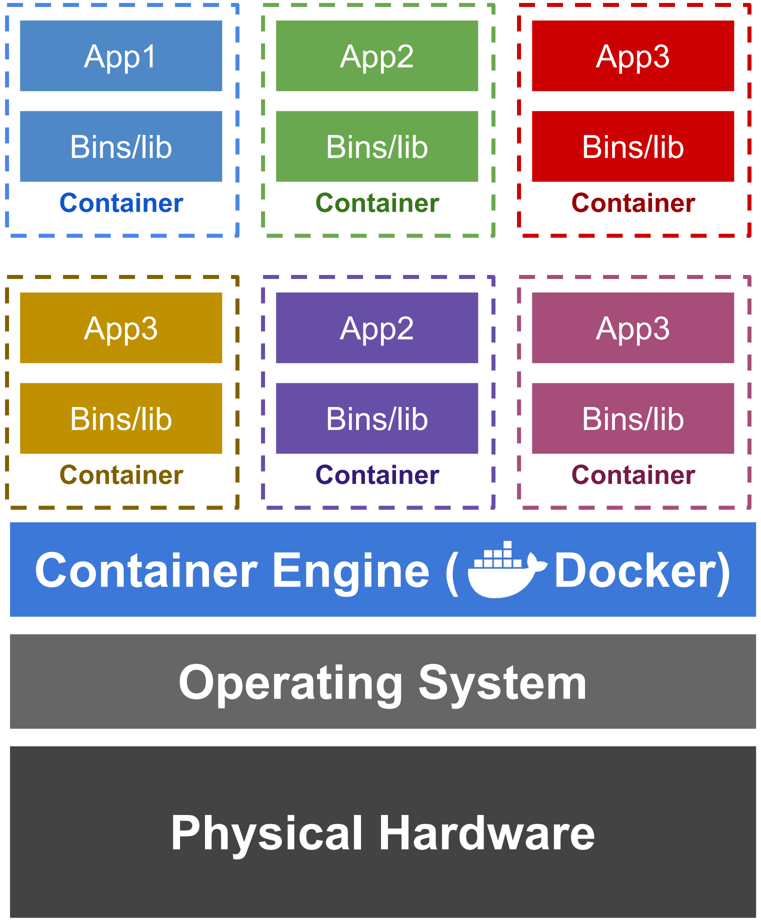


It packages code and all its dependencies to ensure that the application behaves the same way, regardless of where it's run.

Environments vs Virtualization vs Containerization



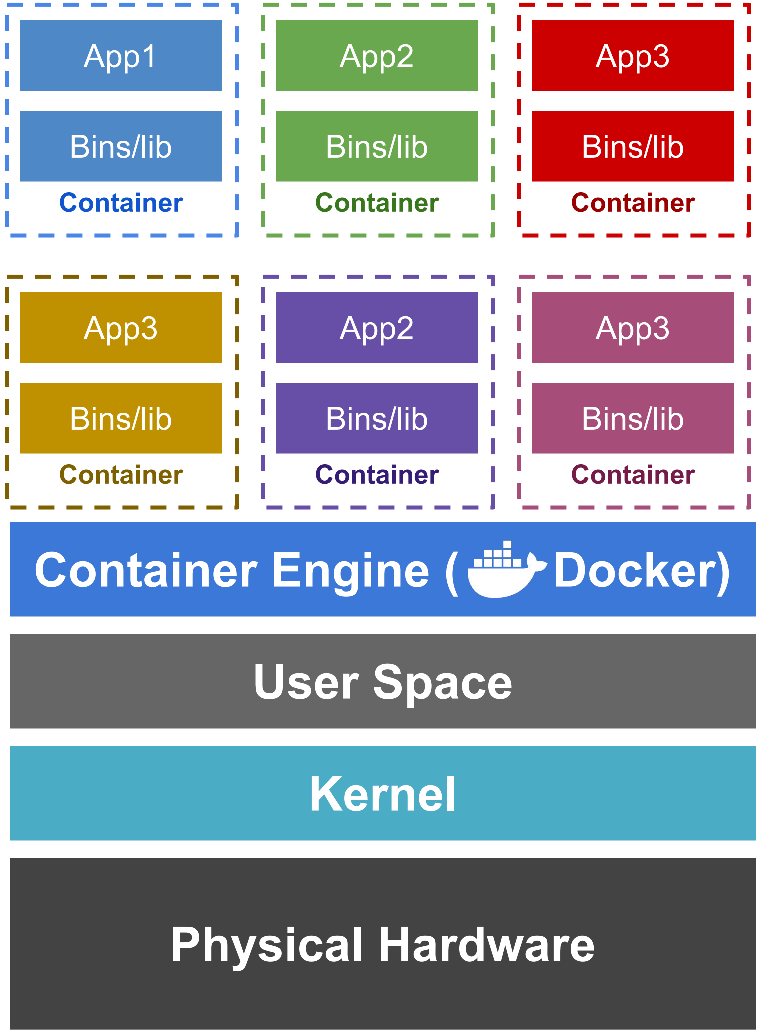
Containerization



Containerization

To understand how containers work, we need to first introduce two key Linux kernel features: namespaces and cgroups.

Containerization

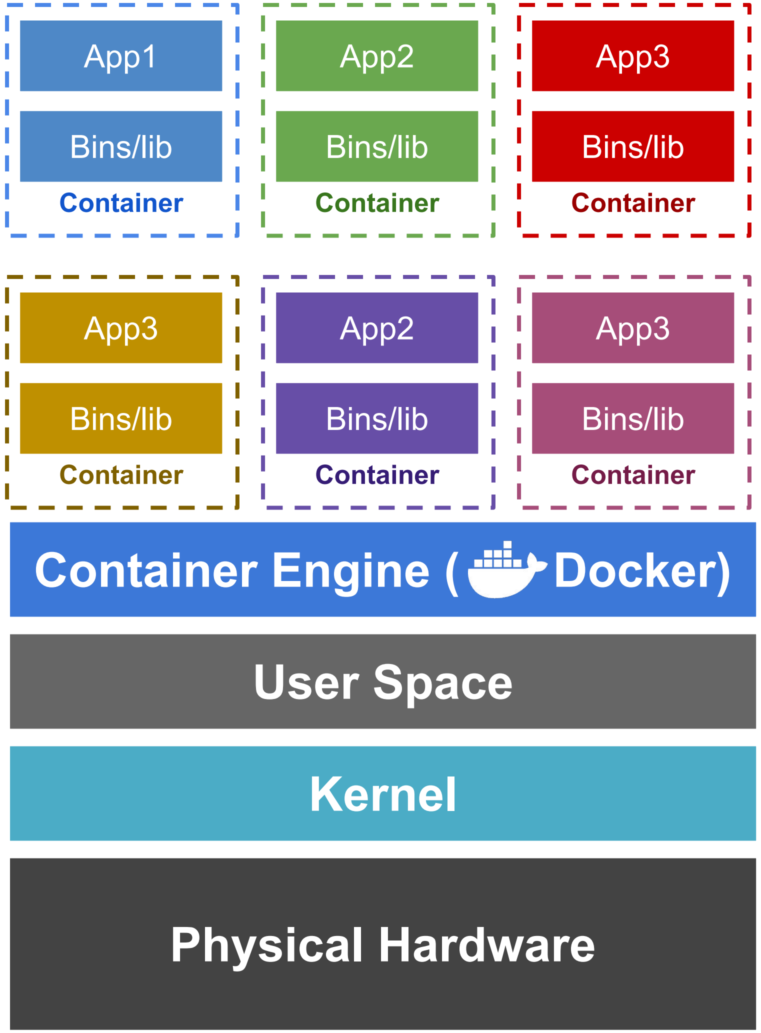


Containerization

To understand how containers work, we need to first introduce two key Linux kernel features: namespaces and cgroups.

The Operating System contains the **Kernel**, which has low level access to the hardware and the **User Space** which contains programs outside the Kernel.

Containerization



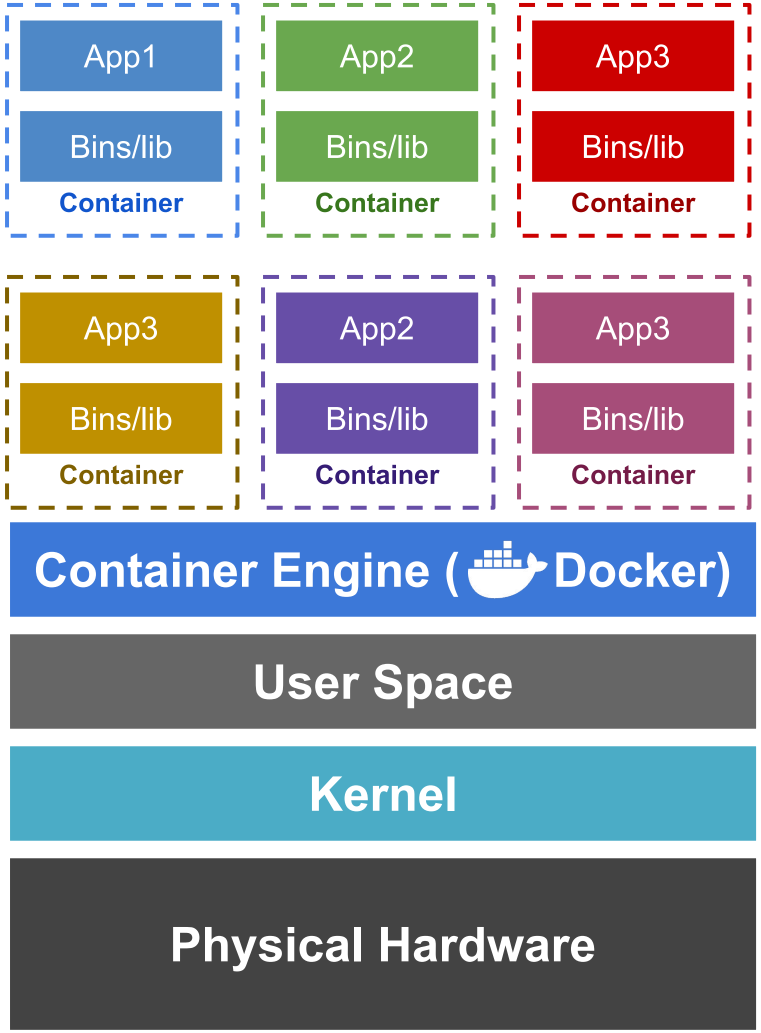
Containerization

Namespace is a feature provided by the Linux Kernel that creates an isolation between system components.

Namespaces allow different processes (or groups of processes) to have their own separate view of system resources, such as process IDs, file systems, network interfaces, and more.

When a process is placed into a namespace, it can only see and interact with the resources within that namespace.

Containerization: Namespaces

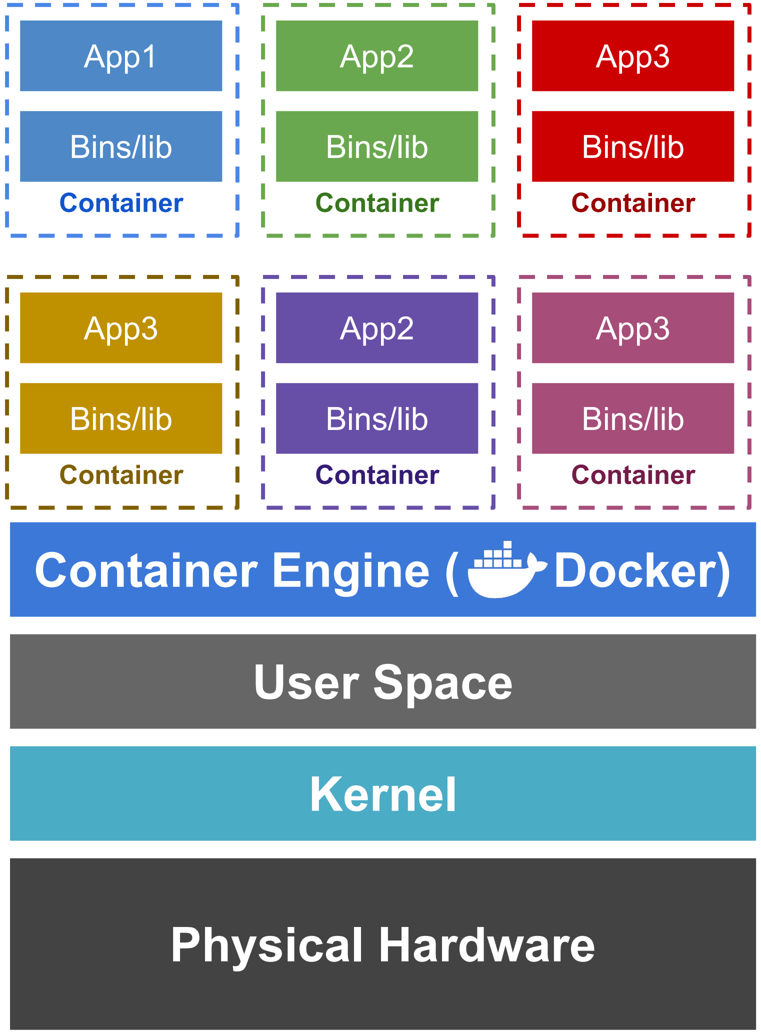


Containerization

- **PID Namespace:** processes inside different PID namespaces can have the same process ID (PID) without conflicts. The host will be able to see the different processes with a different PIDs.

  Example: two containers running on the same host. In one container, a web server process (e.g., Nginx) might have a PID of 1. In another container, a database process (e.g., MySQL) could also have a PID of 1. The host might assign 345 and 678, respectively.

Containerization: Namespaces

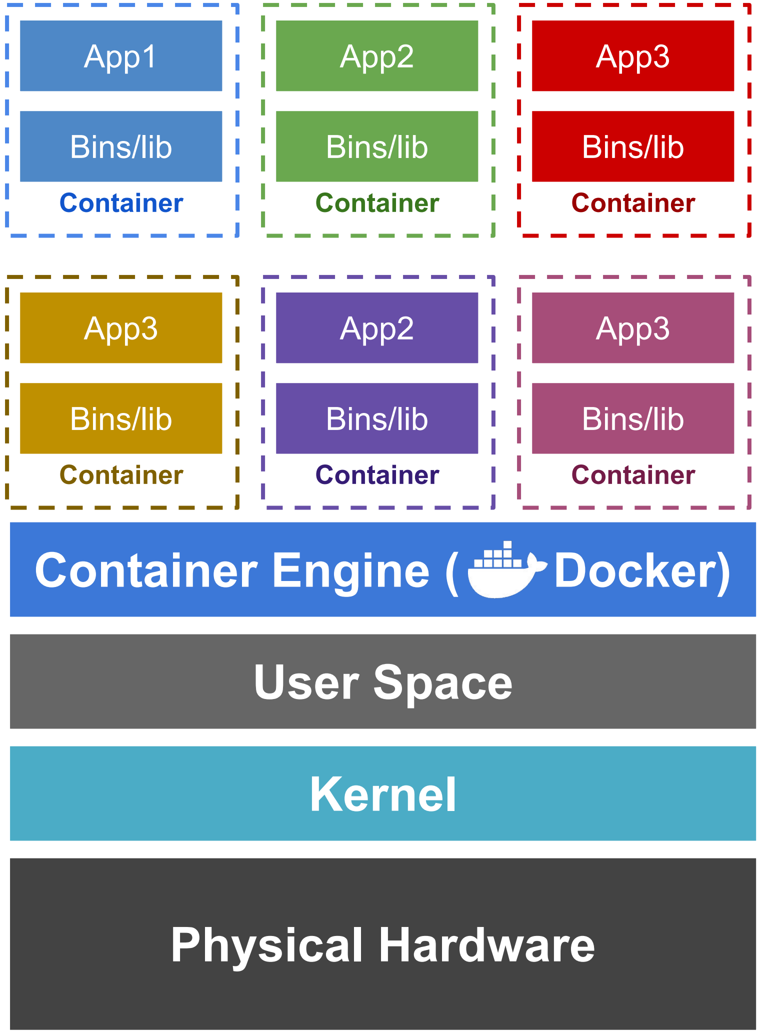


Containerization

- **Mount Namespace:** different containers will have their own view of the filesystem. This includes mounted disks, mount points or directories.

  Example: Suppose you have two containers that mount to the directory /data, where one container mounts *Drive_A* and the other *Drive_B*. Both containers will be unaware of other mounted drives.

Containerization: Namespaces

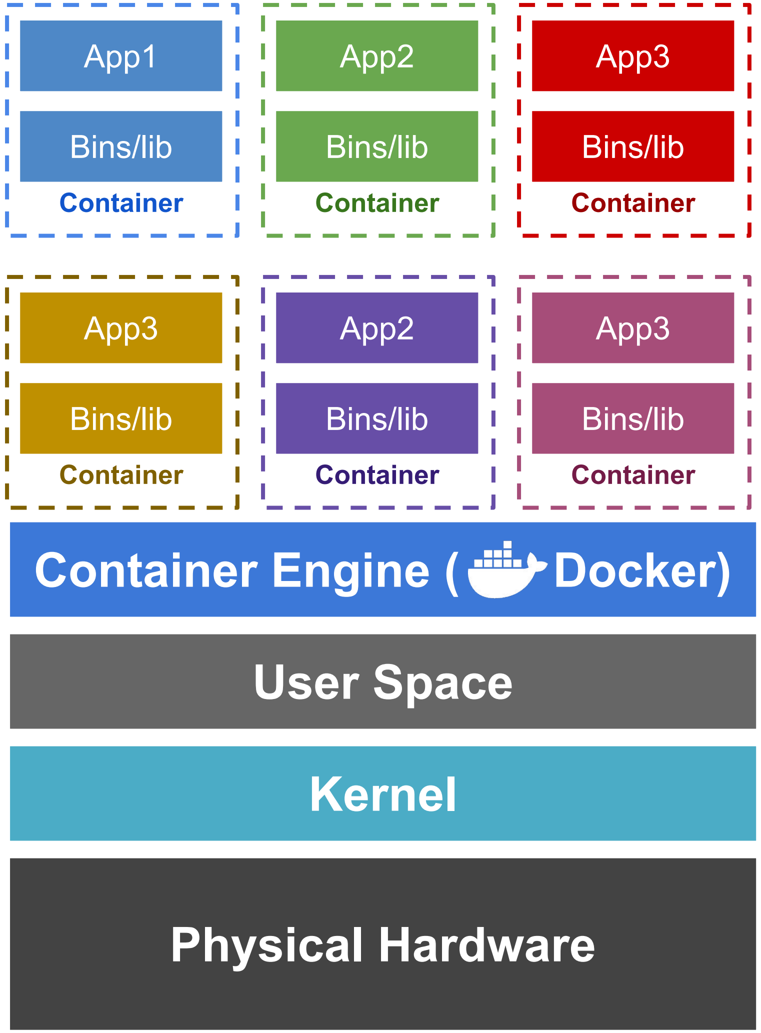


Containerization

- **Network Namespace:** different containers will have their own isolated sub-network to interact with. This include IP addresses, routing tables, firewall rules, etc.

  Example: multiple containers can use the same IP address to perform tasks, without worrying about interfering with each other or security concerns.

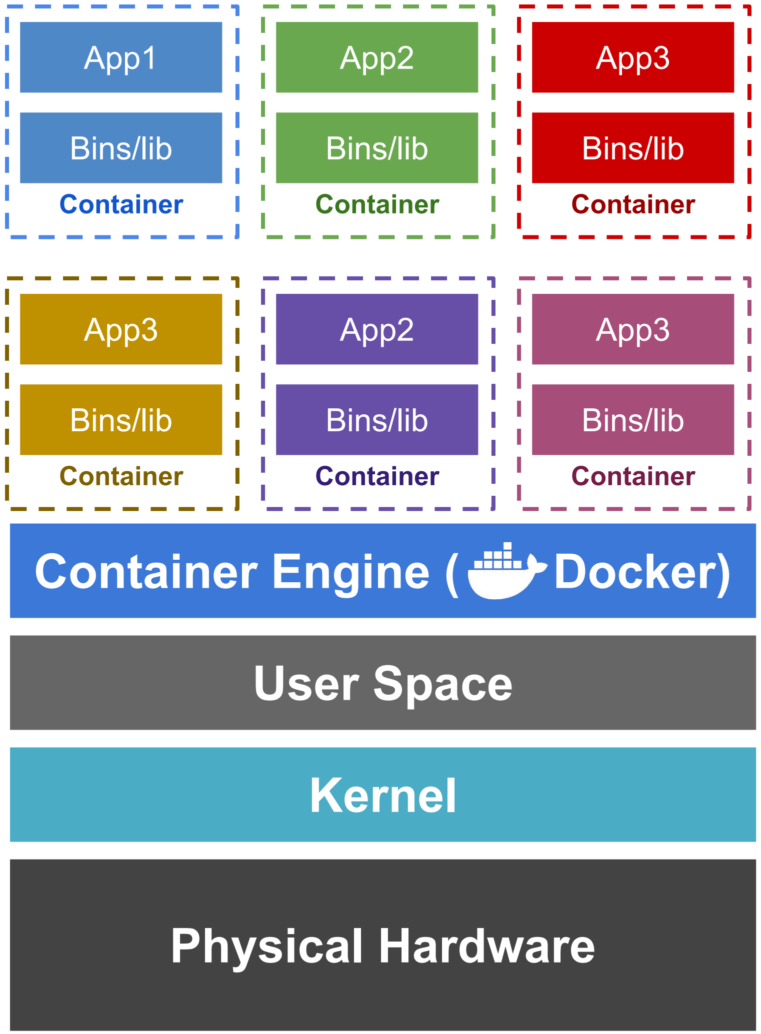
Containerization: Namespaces



Containerization

- **IPC Namespace:** Inter-Process Communication. It isolates the processes communication and shared memory within each namespace.
- **UTS Namespace:** UNIX Timesharing System. Allows each namespace to define its own localhost.
- **User Namespace:** Isolates user groups within each container. Even if a process is running as *root* inside the container, it will have *non-root* privileges in the host.

Containerization: Control Groups



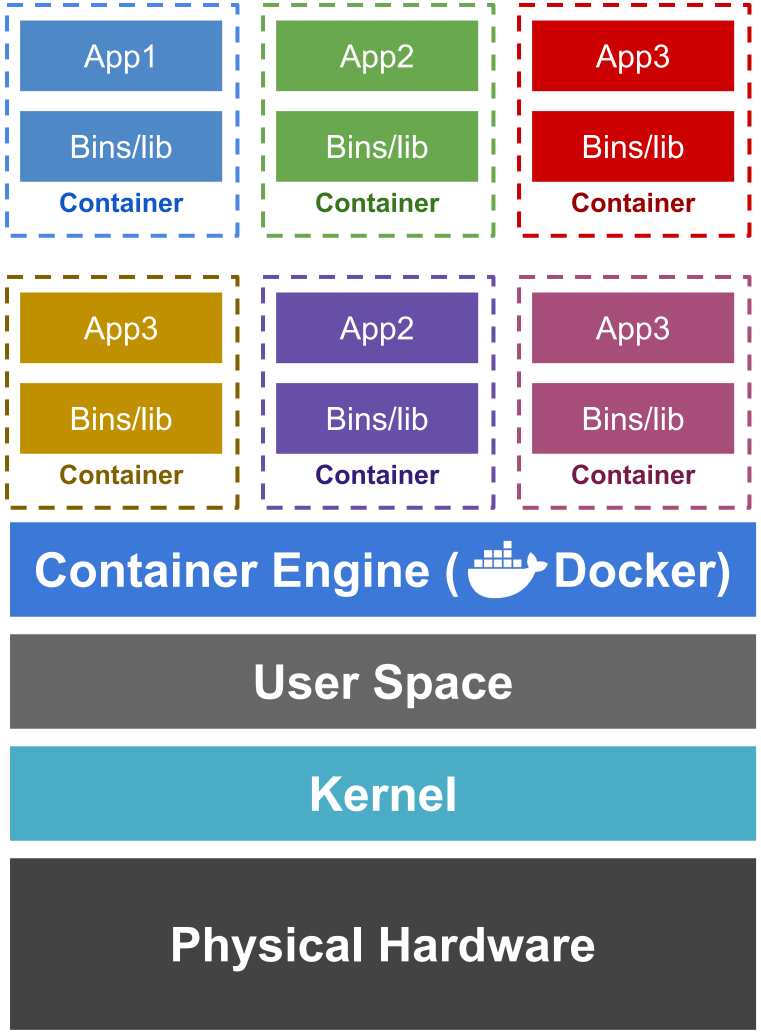
Containerization

Complementary to namespaces, cgroups (Control Groups) allow the limitation and management of system resources such as CPU, memory, disk I/O, and network bandwidth.

By controlling resource allocation, cgroups enable more efficient resource utilization and isolation within containers, making them more lightweight and flexible compared to virtual machines (VMs).

Also, they provide an additional layer of security ensuring that one container cannot bring the system down by exhausting one of those resources.

Containerization: Security Features



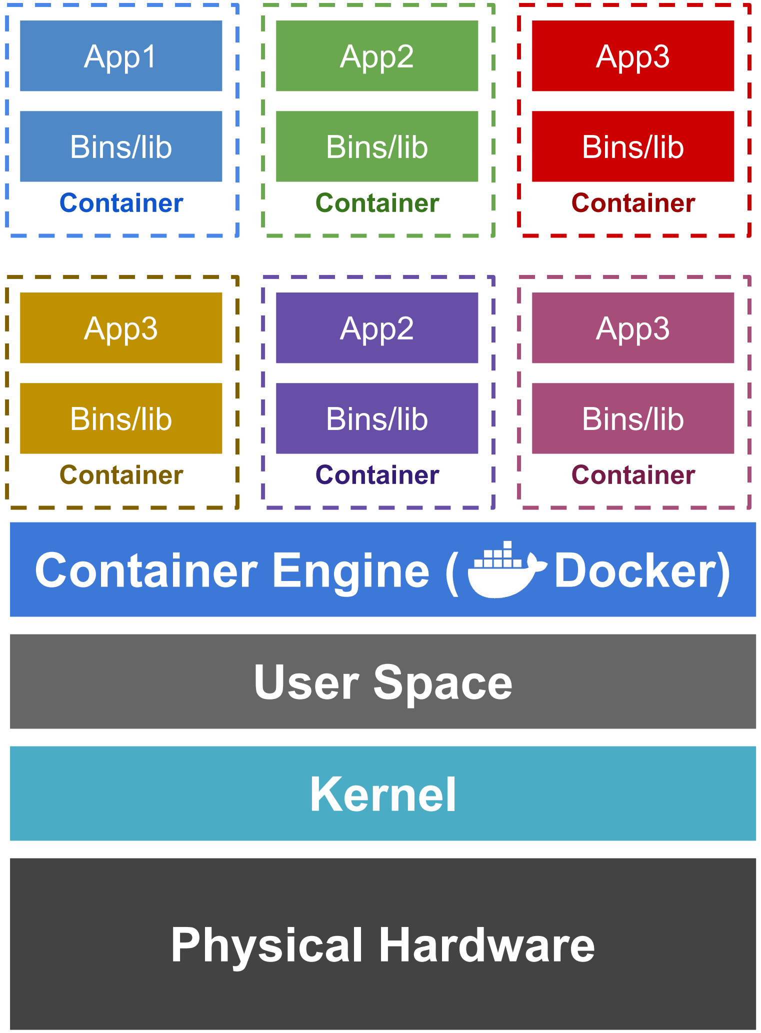
Containerization

Apart from namespaces and cgroups, the Docker engine utilizes additional kernel features to increase security.

By default, containers are given a reduced set of privileges (Secure computing mode, seccomp) reducing by 44 the available system calls (300+). This ensures that containers remain isolated and cannot control the host.

A container is unlikely to require root privileges, since those tasks can be executed by the host. Only the absolutely necessary information is passed into the container.

Containerization: Security features

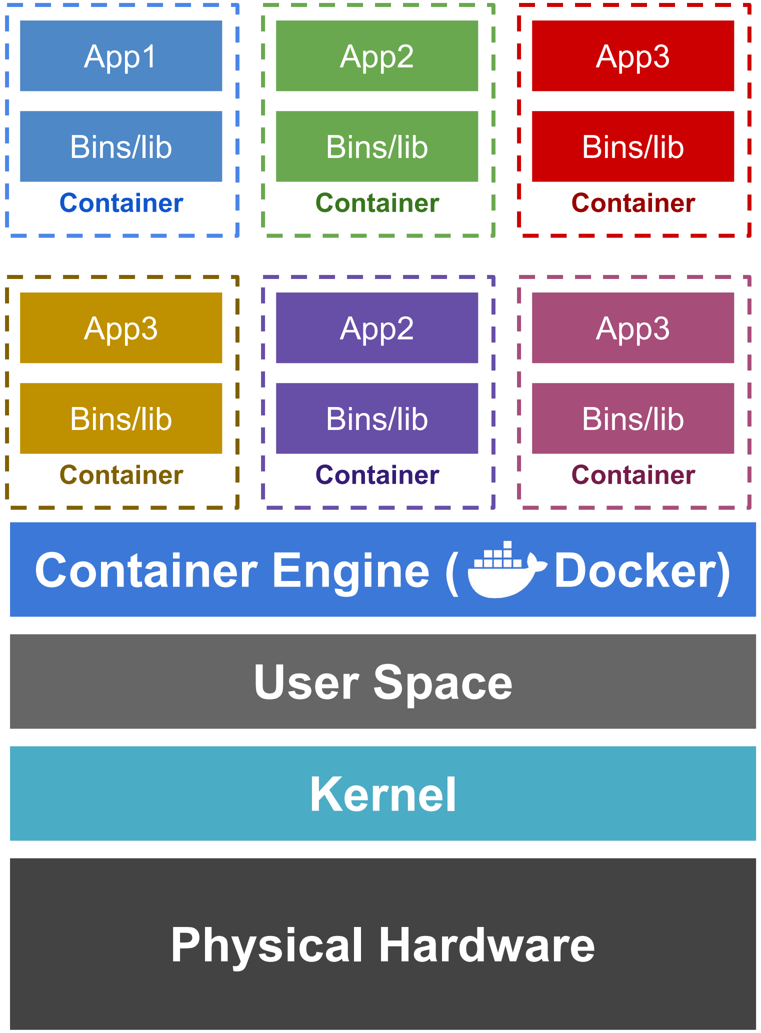


Containerization

Example:
Unless configured otherwise, the containers don't have access to the *syscall reboot*. Which would allow a container to reboot the system.

If a container has access to the syscall *quotactl*, it would have the ability to change the disk quotas, affecting the rest of the host and other containers or VMs.

Containerization: Security features



Containerization

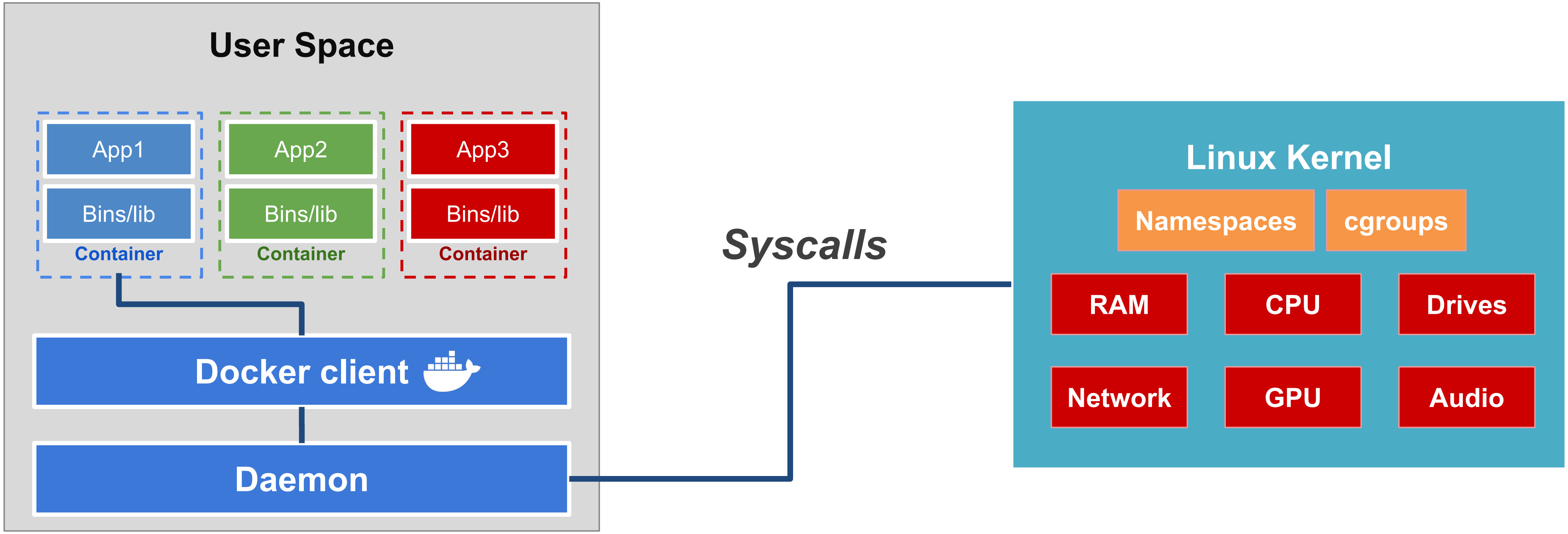
Additional security layer are applied, such as AppArmor (Application Armor) and SELinux (Security-Enhanced Linux).
These modules restrict the the usage of files, directories, sockets, and other processes. Providing an additional layer of security, preventing any container of accessing core system components.

Do not confuse with cgroups, which control resource management!

What Makes Containers so Small?

Container = User Space of OS

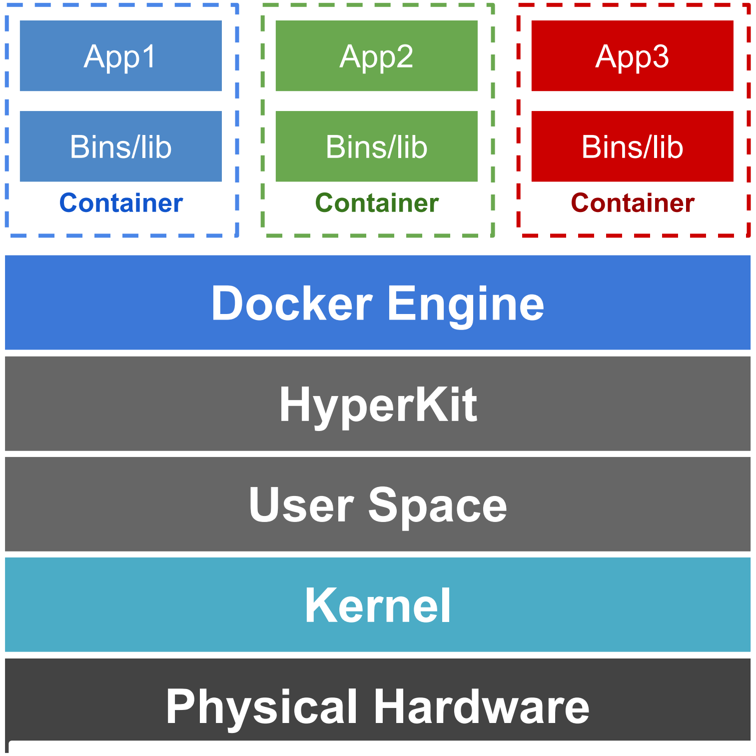
Each container has the minimum code required to run its program. It leverages the host Os (User Space and Kernel) to perform its task.



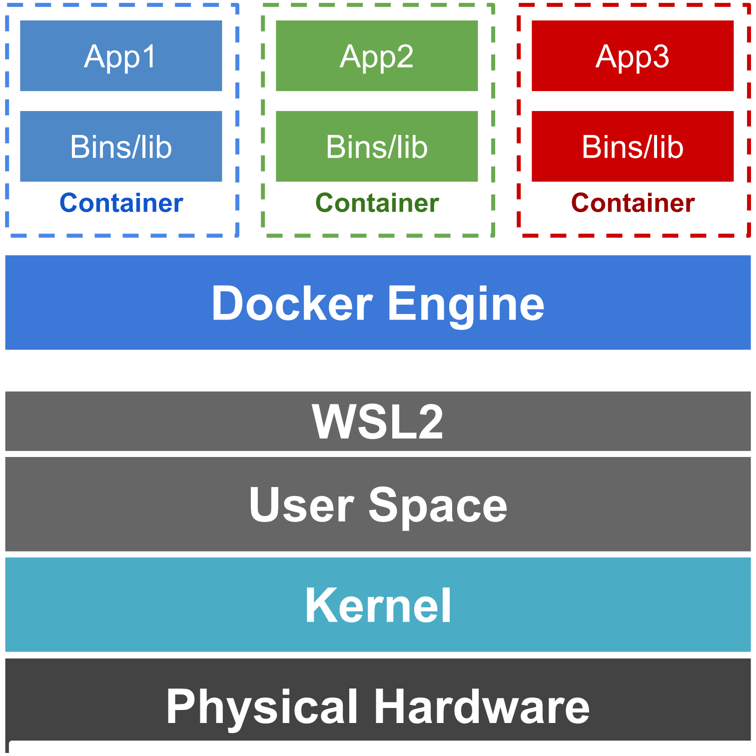
Containers compatibility

If Docker containers rely on Linux kernel features, how can we use them on MacOS and Windows?

Both OS's spin stripped down VMs that translate *syscalls* from the Daemon to the native Kernel.



MacOS



Windows

HyperKit is a virtualization technology.

WSL2 is a custom built Linux kernel, integrated with Windows.
It spins an even thinner Linux VM, which allows it to run native linux programs.

Outline

1. Recap & Motivation
2. What is a Container
3. **Why use Containers**
4. How to use Containers

Advantages of a CONTAINER

- **Portability & Lightweight:** Containers encapsulate everything needed to run an application, making them easy to move across different environments.
- **Fully Packaged:** Containers include the software and all its dependencies, ensuring a consistent environment throughout the development lifecycle.
- **Versatile Usage:** Containers can be used across various stages, from development and testing to training and production deployment

Outline

1. Recap & Motivation
2. What is a Container
3. Why use Containers
4. **How to use Containers**

Examples of Containerization Technologies:

LXC (Linux Containers): The original containerization technology on Linux, offering lightweight virtualization with less isolation than Docker.

Podman: A daemonless container engine that is compatible with Docker, providing more security features like running containers as non-root.

rkt (Rocket): A security-focused container runtime, designed as an alternative to Docker, with a strong emphasis on simplicity and composability.

Orbstack: A fast, lightweight container and VM platform optimized for seamless desktop development.

Docker: The most popular and widely used container platform, known for its ease of use, robust ecosystem, and extensive support.

What is docker?

Open Source: Community-driven and compatible.

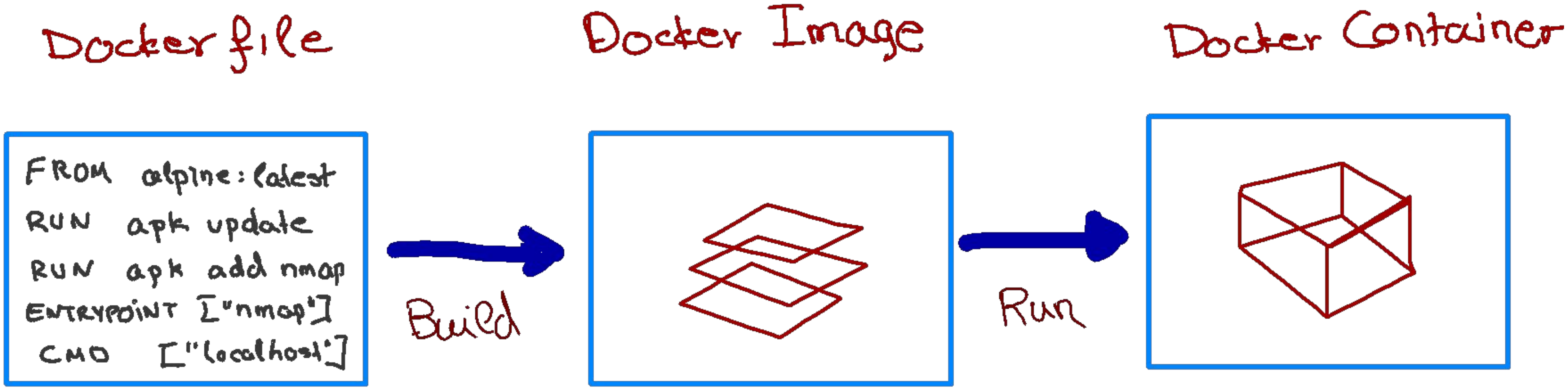
Platform: Develop, ship, and run applications containers.

Portability: Consistent across various environments.

Ecosystem: Docker Hub, Kubernetes, and more.

How to run a docker container

- We use a simple text file, the `Dockerfile`, to build the `Docker Image`, which consists of an iso file and other files.
- We run the `Docker Image` to get `Docker Container`.



What is the difference between an image and container

`Docker Image` is a template aka a blueprint to create a running `docker container`. Docker uses the information available in the `Image` to create (run) a `container`.

`Docker file` is the hand written description of a recipe, `Image` is like the formal recipe and ingredients, `container` is like a dish.

Alternatively, you can think of an `image` as a class and a `container` is an instance of that class.

Anatomy of a Dockerfile

Dockerfile

```
FROM alpine:latest
RUN apk update
RUN apk add nmap
ENTRYPOINT ["nmap"]
CMD ["localhost"]
```

FROM: Specifies the base OS image (e.g., alpine, Ubuntu) for building the Docker image.

RUN: Executes commands to build the image. Each **RUN** creates a new layer.

ENTRYPOINT: Sets the default executable for the container, making it behave like a standalone application.

CMD: Sets default commands or parameters for container startup, but can be overridden by the `docker run` command.

https://docs.docker.com/engine/reference/builder/

Anatomy of a Dockerfile

FROM: Specifies the base OS image (e.g., alpine, Ubuntu) for building the Docker image.

RUN: Executes commands to build the image. Each RUN creates a new layer.

ENTRYPOINT: Sets the default executable for the container, making it behave like a standalone application.

CMD: Sets default commands or parameters for container startup, but can be overridden by the `docker run` command.

ADD: Similar to **COPY**, but can also handle URLs and auto-extract compressed files.

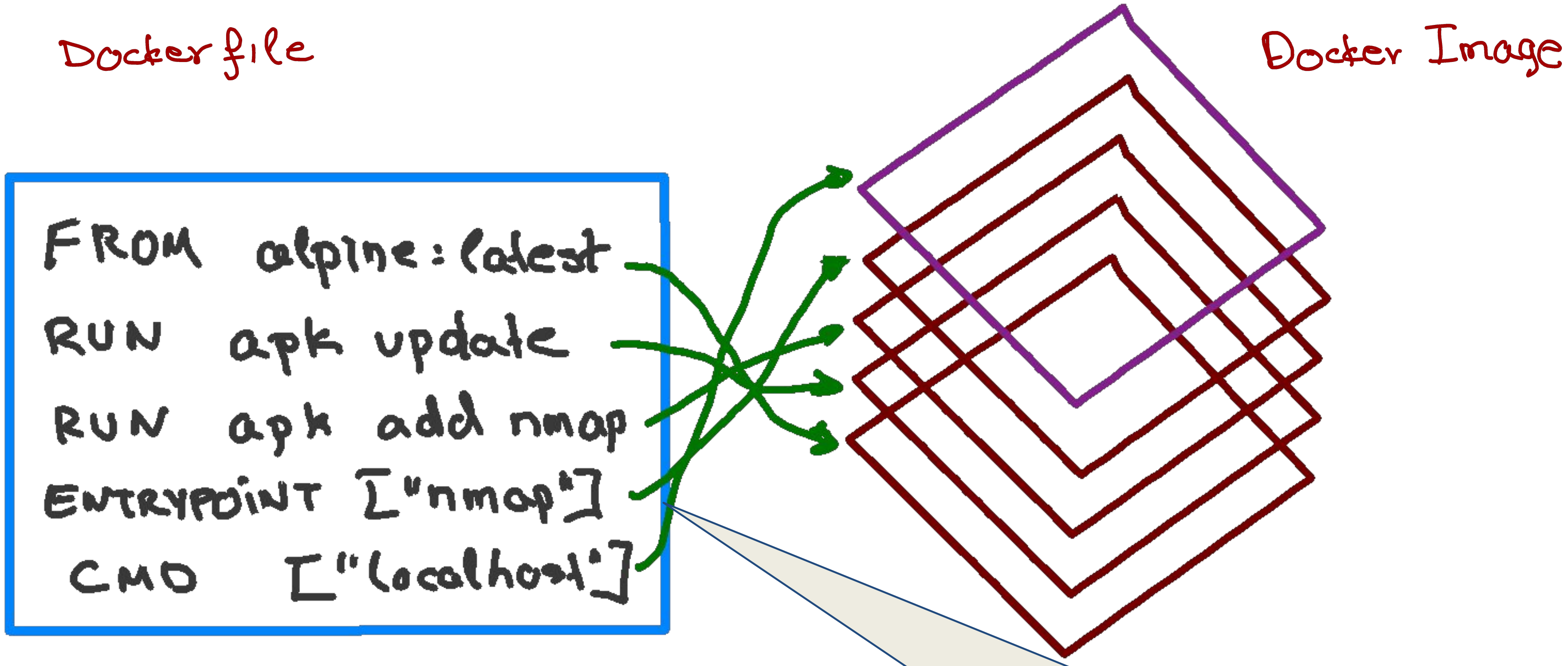
ENV: Sets environment variables within the Docker image. These variables can be used in subsequent commands or by applications within the container.

WORKDIR: Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD instructions that follow in the Dockerfile.

https://docs.docker.com/engine/reference/builder/

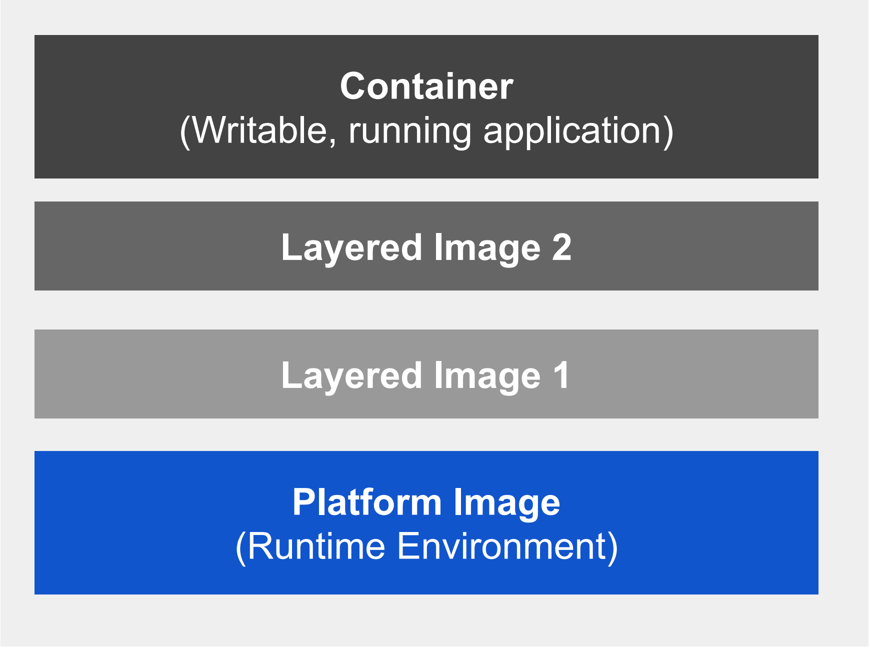
Docker Image as Layers

When we execute the `build` command, the daemon reads the `Dockerfile` and creates a layer for every command.



Nmap (Network Mapper) is an open source utility of network.

Image Layering



A application sandbox
- Each container is based on an image that holds necessary config data
- When you launch a container, a writable layer is added on top of the image

A static snapshot Images are read-only and capture the container's settings.
- Layer images are read-only
- Each image depends on one or more parent images

Platform images define the runtime environment, packages and utilities necessary for containerized application to run. It is an Image that has no parent

Why Layers

Why build an image with multiple layers when we can just build it in a single layer?

Efficiency

Reuse common layers across different images, saving storage and speeding up image creation.

Incremental Updates

Update only the changed layer, reducing the time and bandwidth needed for deployment.

Cache Utilization

Docker caches layers. If no changes are detected, subsequent builds are faster.

Modularity

Break down complex setup into manageable pieces, making debugging easier.

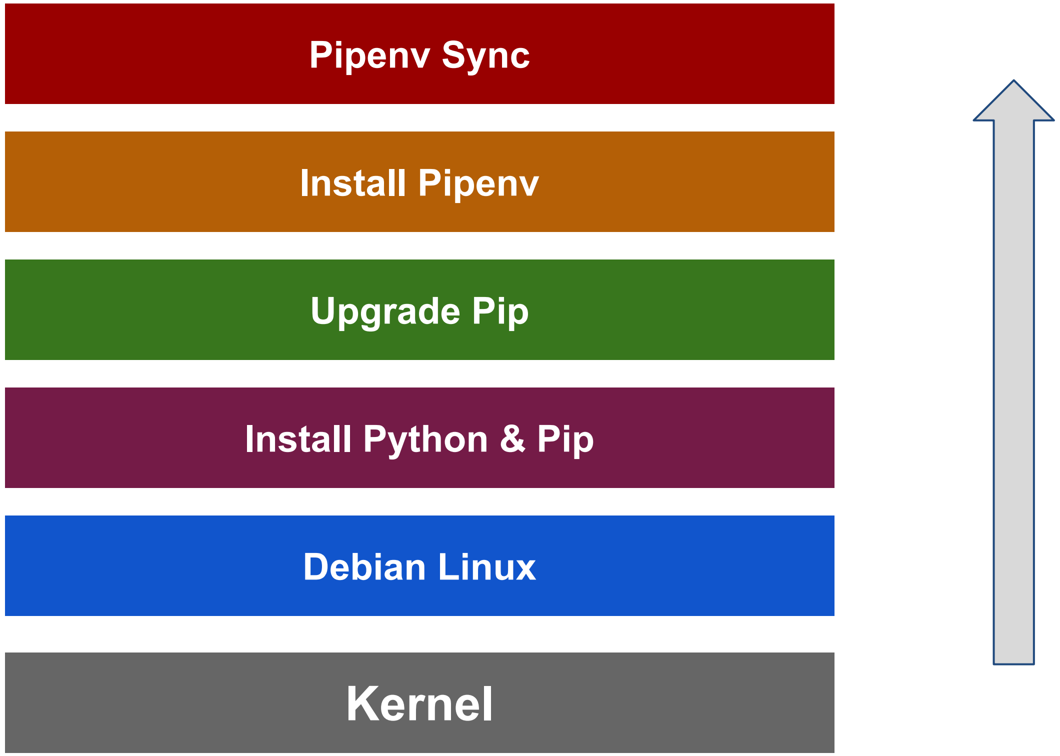
Security

Smaller attack surface per layer and easier to scan for vulnerabilities.

WE WILL SEE AN EXAMPLE LATER

Image Layering - Example

Docker layers for a container running debian and a python environment using Pipenv



Docker Vocabulary

Docker File
A text document with commands on how to create an Image

Docker Image
The basis of a Docker container. Represent a full application

Docker Container
The standard unit in which the application service resides and executes

Docker Engine
Creates, ships and runs Docker containers deployable on a physical or virtual, host locally, in a datacenter or cloud service provider

Registry Service (Docker Hub or Docker Trusted Registry)
Cloud or server-based storage and distribution service for your images

Images
How you **store** your application

Containers
How you **run** your application

FAQ: Running Multiple Containers from a Single Image

How can you run multiple containers from the same image?

Yes, you could think of an image as instating a class. You can create multiple instances (containers) from a single image.

Wouldn't all these containers be identical?

Not necessarily. Containers can be instantiated with different parameters using the CMD command, making them unique in behavior.

Dockerfile

```
FROM ubuntu:latest
RUN apt-get update
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["world"]
```

```
> docker build -t hello_world_cmd -f Dockerfile .

> docker run -it hello_world_cmd
> Hello world
> docker run -it hello_world_cmd Pavlos
> Hello Pavlos
```

Tutorial (T3): Installing Docker Desktop

- Install Docker Desktop. Use one of the links below to download the proper Docker application depending on your operating system.
  - For Mac users, follow this link- https://docs.docker.com/docker-for-mac/install/.
  - For Windows users, follow this link- https://docs.docker.com/docker-for-windows/install/ Note: You will need to install Hyper-V to get Docker to work.
  - For Linux users, follow this link- https://docs.docker.com/install/linux/docker-ce/ubuntu/
- Once installed run the docker desktop.
- Open a Terminal window and type `docker run hello-world` to make sure Docker is installed properly.

Tutorial (T3): Docker commands

Check what version of Docker

Get version of Docker CLI

docker command

```
docker --version
```

Tutorial (T3): Developing App using Containers

- Let us build the simple-translate app using Docker
- For this we will do the following:
    - Clone or download code (https://github.com/dlops-io/simple-translate)

```
git clone https://github.com/dlops-io/simple-translate
```

Tutorial (T3): Developing App using Containers

- Let us build the simple-translate app using Docker
- For this we will do the following:
  - Clone or download code (https://github.com/dlops-io/simple-translate)
  - Build a container

Tutorial (T3): Docker commands

Build an image based on a Dockerfile

Build the image

docker command

```
docker build -t simple-translate  -f Dockerfile .
```

Name the image

Name of docker file and "." means look at the current working directory

Dockerfile

```
# Use the official Debian-hosted Python image
FROM python:3.9-slim-buster

# Tell pipenv where the shell is.
# This allows us to use "pipenv shell" as a container entry point.
# ENV PYENV_SHELL=/bin/bash

# Ensure we have an up to date baseline, install dependencies
# apt-get is a command-line tool used to manage packages
RUN set -ex; \    # -e build process will stop if any command following set -ex fails. -x prints the output
   apt-get update && \   # updates the local package index
   apt-get upgrade -y && \  # upgrade all the installed packages
   apt-get install -y --no-install-recommends build-essential git && \
   pip install --no-cache-dir --upgrade pip && \
   pip install pipenv

# Add Pipfile, Pipfile.lock + python code
ADD . /  # adds the content of the current directory "." into the root directory of the container
RUN pipenv sync

# Entry point
ENTRYPOINT ["/bin/bash"]

# Get into the pipenv shell
CMD ["-c", "pipenv shell"]
```

Docker Image as Layers

```
>docker build -t hello_world_cmd -f Dockerfile .

Sending build context to Docker daemon   34.3kB
Step 1/4 : FROM ubuntu:latest
latest: Pulling from library/ubuntu
54ee1f796a1e: Already exists
f7bfea53ad12: Already exists
46d371e02073: Already exists
b66c17bbf772: Already exists
Digest: sha256:31dfb10d52ce76c5ca0aa19d10b3e6424b830729e32a89a7c6eee2cda2be67
Status: Downloaded newer image for ubuntu:latest
 ---> 4e2eef94cd6b
Step 2/4 : RUN apt-get update
 ---> Running in e3e1a87e8d6e
Get:1 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
Get:2 http://security.ubuntu.com/ubuntu focal-security InRelease [107 kB]
Get:3 http://security.ubuntu.com/ubuntu focal-security/universe amd64 Packages [67.5 kB]
Get:4 http://archive.ubuntu.com/ubuntu focal-updates InRelease [111 kB]
Get:5 http://archive.ubuntu.com/ubuntu focal-backports InRelease [98.3 kB]
Get:6 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [231 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal/restricted amd64 Packages [33.4 kB]
Get:8 http://archive.ubuntu.com/ubuntu focal/main amd64 Packages [1275 kB]
Get:9 http://security.ubuntu.com/ubuntu focal-security/multiverse amd64 Packages [1078 B]
```

Step1: Instruction 1

Step2: Instruction 2

```
FROM ubuntu:latest
RUN apt-get update
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["world"]
```

. . .

Docker Image as Layers

```
FROM ubuntu:latest
RUN apt-get update
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["world"]
```

```
>docker build -t hello_world_cmd -f Dockerfile .

….
Step 3/4 : ENTRYPOINT ["/bin/echo", "Hello"]
 ---> Running in 52c7a98397ad
Removing intermediate container 52c7a98397ad
 ---> 7e4f8b0774de
Step 4/4 : CMD ["world"]
 ---> Running in 353adb968c2b
Removing intermediate container 353adb968c2b
 ---> a89172ee2876
Successfully built a89172ee2876
Successfully tagged hello_world_cmd:latest
```

Step3: Instruction 3

Step4: Instruction 4

Docker Image as Layers

```
>docker build -t simple-translate  -f Dockerfile .

 => [internal] load build definition from Dockerfile                                                        0.0s
 => => transferring dockerfile: 756B                                                                        0.0s
 => [internal] load metadata for docker.io/library/python:3.11-slim-buster                                  1.0s
 => [internal] load .dockerignore                                                                           0.0s
 => => transferring context: 2B                                                                             0.0s
 => [1/4] FROM docker.io/library/python:3.11-slim-buster@sha256:c46b0ae5728c2247b99903098ade3176a58e274d9c7d2efeaaab3e0621a53935    2.8s
 => => resolve docker.io/library/python:3.11-slim-buster@sha256:c46b0ae5728c2247b99903098ade3176a58e274d9c7d2efeaaab3e0621a53935    0.0s
 => => sha256:81b2c804d9ba5014835bedffff61fb42e5d78be661b781654cda06b3d95237f0 1.37kB / 1.37kB              0.0s
 => => sha256:12cacc23b6dec78ca7b056d56e3de48252669ed49fffd95ed36adbf9dfe3cec0 6.85kB / 6.85kB              0.0s
 => => sha256:d191be7a3c9fa95847a482db8211b6f85b45096c7817fdad4d7661ee7ff1a421 25.92MB / 25.92MB            1.4s
 => => sha256:14aea17807c4c653827ca820a0526d96552bda685bf29293e8be90d1b05662f6 2.65MB / 2.65MB              0.6s
 => => sha256:67cefd826e1d4a3bce3c47a040ab445ba7ba6586dea8b8380bdad6ee3462f9c1 12.10MB / 12.10MB            1.3s
 => => sha256:c46b0ae5728c2247b99903098ade3176a58e274d9c7d2efeaaab3e0621a53935 988B / 988B                  0.0s
 => => sha256:195c388ea91b233c774667795edf5a47d3b02b3db8da49447d964dbafee7a786 244B / 244B                  0.7s
 => => sha256:db8899040fb5395274edb3f6930ed67e7c7a4cd70adc8f6f21cfa2ab92dce912 3.38MB / 3.38MB              1.2s
 => => extracting sha256:d191be7a3c9fa95847a482db8211b6f85b45096c7817fdad4d7661ee7ff1a421                   0.8s
 => => extracting sha256:14aea17807c4c653827ca820a0526d96552bda685bf29293e8be90d1b05662f6                   0.1s
 => => extracting sha256:67cefd826e1d4a3bce3c47a040ab445ba7ba6586dea8b8380bdad6ee3462f9c1                   0.3s
 => => extracting sha256:195c388ea91b233c774667795edf5a47d3b02b3db8da49447d964dbafee7a786                   0.0s
 => => extracting sha256:db8899040fb5395274edb3f6930ed67e7c7a4cd70adc8f6f21cfa2ab92dce912                   0.2s
 => [internal] load build context                                                                           0.0s
 => => transferring context: 161.76kB                                                                       0.0s
 => [2/4] RUN set -ex;         apt-get update &&     apt-get upgrade -y &&         apt-get install -y --no-install-recommends build-essential git &&  p    24.0s
 => [3/4] ADD . /                                                                                           0.0s
 => [4/4] RUN pipenv sync                                                                                   7.8s
 => exporting to image                                                                                      0.7s
 => => exporting layers                                                                                     0.7s
 => => writing image sha256:e473d8916478a1f09ecfeba01dde5113133490541018fb34dba99947ff140ba0                0.0s
 => => naming to docker.io/library/simple-translate                                                         0.0s
```

Step1: Instruction 1

Step2: Instruction 2

Step3: Instruction 3

Step4: Instruction 4

Docker Image as Layers

```
> docker images
REPOSITORY        TAG         IMAGE ID        CREATED          SIZE
hello_world_cmd   latest      a89172ee2876    7 minutes ago    96.7MB
ubuntu            latest      4e2eef94cd6b    3 weeks ago      73.9MB
```

```
> docker image history hello_world_cmd
IMAGE          CREATED          CREATED BY                                      SIZE       COMMENT
a89172ee2876   8 minutes ago    /bin/sh -c #(nop)  CMD ["world"]                0B
7e4f8b0774de   8 minutes ago    /bin/sh -c #(nop)  ENTRYPOINT ["/bin/echo" "…   0B
cfc0c414a914   8 minutes ago    /bin/sh -c apt-get update                       22.8MB
4e2eef94cd6b   3 weeks ago      /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>      3 weeks ago      /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
<missing>      3 weeks ago      /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   811B
<missing>      3 weeks ago      /bin/sh -c [ -z "$(apt-get indextargets)" ]     1.01MB
<missing>      3 weeks ago      /bin/sh -c #(nop) ADD file:9f937f4889e7bf646…   72.9MB
```

Docker Image as Layers

```
> docker images
REPOSITORY        TAG        IMAGE ID        CREATED          SIZE
hello_world_cmd   latest     e473d8916478    47 hours ago     8.83MB
simple-translate  latest     e473d8916478    22 minutes ago   483MB
```

```
> docker image history simple-translate
IMAGE          CREATED          CREATED BY                                        SIZE      COMMENT
e473d8916478   22 minutes ago   CMD ["-c" "pipenv shell"]                         0B        buildkit.dockerfile.v0
<missing>      22 minutes ago   ENTRYPOINT ["/bin/bash"]                          0B        buildkit.dockerfile.v0
<missing>      22 minutes ago   RUN /bin/sh -c pipenv sync # buildkit             40.5MB    buildkit.dockerfile.v0
<missing>      22 minutes ago   ADD . / # buildkit                                155kB     buildkit.dockerfile.v0
<missing>      22 minutes ago   RUN /bin/sh -c set -ex;       apt-get update &…   329MB     buildkit.dockerfile.v0
<missing>      22 minutes ago   ENV PYENV_SHELL=/bin/bash                         0B        buildkit.dockerfile.v0
<missing>      15 months ago    CMD ["python3"]                                   0B        buildkit.dockerfile.v0
<missing>      15 months ago    RUN /bin/sh -c set -eux;   savedAptMark="$(a…     12.2MB    buildkit.dockerfile.v0
<missing>      15 months ago    ENV PYTHON_GET_PIP_SHA256=96461deced5c2a487d…     0B        buildkit.dockerfile.v0
<missing>      15 months ago    ENV PYTHON_GET_PIP_URL=https://github.com/py…     0B        buildkit.dockerfile.v0
<missing>      15 months ago    ENV PYTHON_SETUPTOOLS_VERSION=65.5.1              0B        buildkit.dockerfile.v0
<missing>      15 months ago    ENV PYTHON_PIP_VERSION=23.1.2                     0B        buildkit.dockerfile.v0
<missing>      15 months ago    RUN /bin/sh -c set -eux;  for src in idle3 p…     32B       buildkit.dockerfile.v0
<missing>      15 months ago    RUN /bin/sh -c set -eux;   savedAptMark="$(a…     31.4MB    buildkit.dockerfile.v0
<missing>      15 months ago    ENV PYTHON_VERSION=3.11.4                         0B        buildkit.dockerfile.v0
<missing>      15 months ago    ENV GPG_KEY=A035C8C19219BA821ECEA86B64E628F8…     0B        buildkit.dockerfile.v0
<missing>      15 months ago    RUN /bin/sh -c set -eux;  apt-get update;  a…     6.66MB    buildkit.dockerfile.v0
<missing>      15 months ago    ENV LANG=C.UTF-8                                  0B        buildkit.dockerfile.v0
<missing>      15 months ago    ENV PATH=/usr/local/bin:/usr/local/sbin:/usr…     0B        buildkit.dockerfile.v0
<missing>      15 months ago    /bin/sh -c #(nop)  CMD ["bash"]                   0B
<missing>      15 months ago    /bin/sh -c #(nop) ADD file:d4a87f28032264e15…     63.5MB
```

Why Layers

Why build an image with multiple layers when we can just build it in a single layer?
Let's take an example to explain this concept better, let us try to change the Dockerfile_cmd we created and rebuild a new Docker image.

```
> docker build -t hello_world_cmd -f Dockerfile_cmd .
Sending build context to Docker daemon   34.3kB
Step 1/4 : FROM ubuntu:latest
 ---> 4e2eef94cd6b
Step 2/4 : RUN apt-get update
 ---> Using cache
 ---> cfc0c414a914
Step 3/4 : ENTRYPOINT ["/bin/echo", "Hello"]
 ---> Using cache
 ---> 7e4f8b0774de
Step 4/4 : CMD ["world"]
 ---> Using cache
 ---> a89172ee2876
Successfully built a89172ee2876
Successfully tagged hello_world_cmd:latest
```

Have seen this before. Use cache

As you can see that the image was built using the existing layers from our previous docker image builds. If some of these layers are being used in other containers, they can just use the existing layer instead of recreating it from scratch.

Why Layers

Why build an image with multiple layers when we can just build it in a single layer?
Let's take an example to explain this concept better, let us try to change the Dockerfile_cmd we created and rebuild a new Docker image.

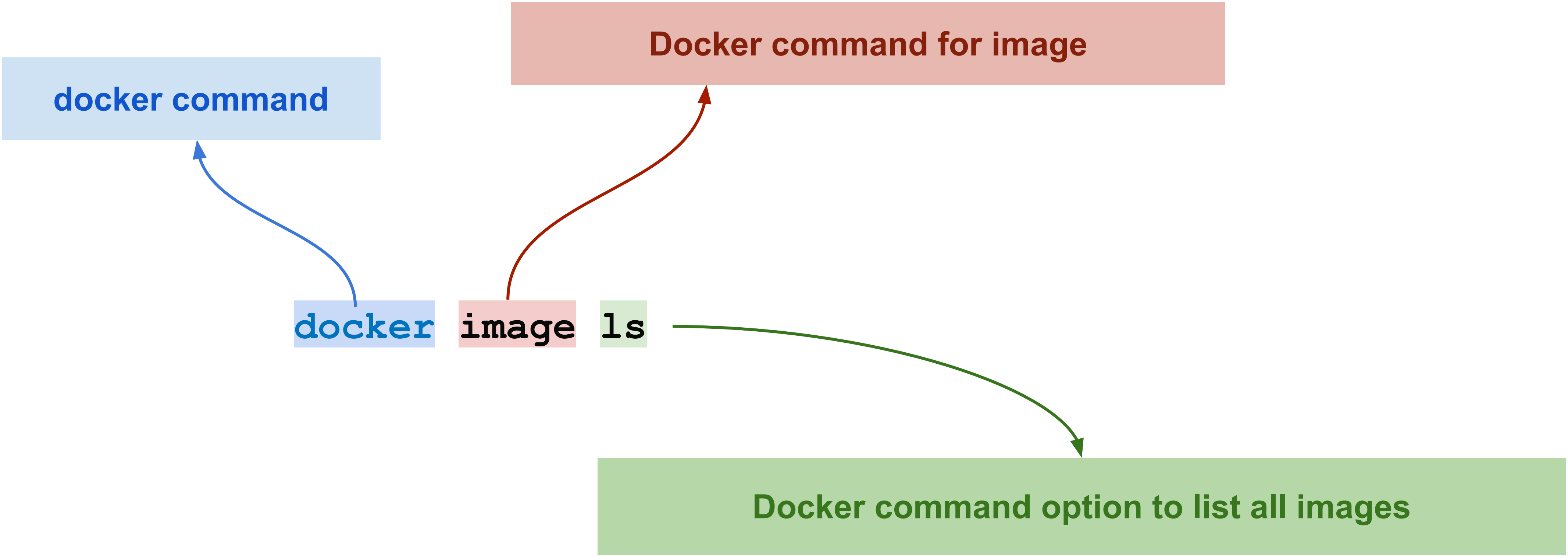
```
> docker build -t hello_world_cmd -f Dockerfile_cmd .
[+] Building 0.6s (9/9) FINISHED                                                          docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                          0.0s
 => => transferring dockerfile: 756B                                                                          0.0s
 => [internal] load metadata for docker.io/library/python:3.11-slim-buster                                    0.5s
 => [internal] load .dockerignore                                                                             0.0s
 => => transferring context: 2B                                                                               0.0s
 => [1/4] FROM docker.io/library/python:3.11-slim-buster@sha256:c46b0ae5728c2247b99903098ade3176a58e274d9c7d2efeaaab3e0621a53935   0.0s
 => [internal] load build context                                                                             0.0s
 => => transferring context: 5.91kB                                                                           0.0s
 => CACHED [2/4] RUN set -ex;   apt-get update &&   apt-get upgrade -y &&     apt-get install -y --no-install-recommends build-essential git &&   0.0s
 => CACHED [3/4] ADD . /                                                                                      0.0s
 => CACHED [4/4] RUN pipenv sync                                                                              0.0s
 => exporting to image                                                                                        0.0s
 => => exporting layers                                                                                       0.0s
 => => writing image sha256:e473d8916478a1f09ecfeba01dde5113133490541018fb34dba99947ff140ba0                  0.0s
 => => naming to docker.io/library/simple-translate                                                           0.0s
```

Have seen this before. Use cache

As you can see that the image was built using the existing layers from our previous docker image builds. If some of these layers are being used in other containers, they can just use the existing layer instead of recreating it from scratch.

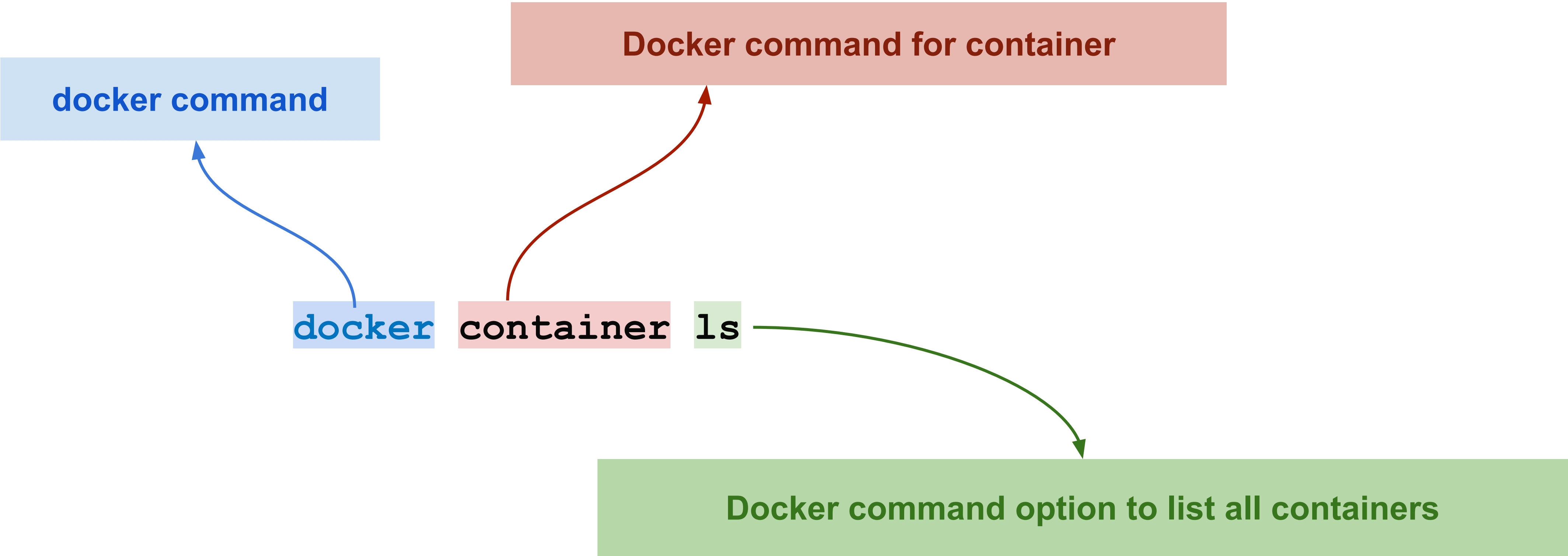
Tutorial (T3): Docker commands

List all docker images



Tutorial (T3): Docker commands

List all running docker containers



Tutorial (T3): Developing App using Containers

- Let us build the simple-translate app using Docker
- For this we will do the following:
    - Clone or download code (https://github.com/dlops-io/simple-translate)
    - Build a container
    - Run a container

Tutorial (T3): Docker commands

Run a docker container using an image

```
docker run --rm --name simple-translate -ti --entrypoint /bin/bash simple-translate
```

- **run**: Run the container
- **--rm**: automatically clean up the container and remove the file system when the container exit
- **--name simple-translate**: Name of the container
- **-ti**: 't' is to give us a terminal and 'i' is for interactive mode
- **--entrypoint /bin/bash**: Default command to execute on startup. Overwrites the ENTRYPOINT from Dockerfile
- **simple-translate**: Name of the image to use

Tutorial (T3): Docker commands

Open another command prompt and check how many container and images we have

```
docker image ls
```

```
docker container ls
```

Tutorial (T3): Developing App using Containers

- Let us build the simple-translate app using Docker
- For this we will do the following:
  - Clone or download code (https://github.com/dlops-io/simple-translate)
  - Build a container
  - Run a container
  - **Push container on Docker Hub**

Tutorial (T3): Docker commands

Sign up in Docker Hub and create an Access Token. Use that token to authenticate with the command below

Command used to authenticate to a Docker registry

Utilizes an access token instead of a password for secure authentication.

```
docker login -u <USER NAME> -p <ACCESS TOKEN>
```

Specifies the username for authentication

Tutorial (T3): Docker commands

Tag the Docker Image

Assigns a new name or tag to an image

Docker Hub username or repository namespace. The new name for the tagged image.

```
docker tag <SOURCE IMAGE NAME>[:TAG] <USER NAME>/<TARGET_IMAGE[:TAG]>
```

The existing image name to be tagged. Optional tag for the source image (defaults to latest if omitted).

Tutorial (T3): Docker commands

- Push to Docker Hub

Command used to upload a Docker image from your local machine to a remote registry like Docker Hub

The name of the image you want to push to the registry. User name can be included as part of the name

```
docker push <USER NAME>/<TARGET_IMAGE[:TAG]>
```

Tutorial (T3): Developing App using Containers

- Let us build the simple-translate app using Docker
- For this we will do the following:
  - Clone or download code (https://github.com/dlops-io/simple-translate)
  - Build a container
  - Run a container
  - Push container on Docker Hub
  - **Pull the new container and run it**

Tutorial (T3): Docker commands

- Pull from Docker Hub

Command used to download a Docker image from a registry to your local machine

The name of the image you want to pull and TAG

```
docker pull [OPTIONS] <USER NAME>/<TARGET_IMAGE[:TAG]>
```

Tutorial (T3): Developing App using Containers

- Let us build the simple-translate app using Docker
- For this we will do the following:
  - Clone or download code (https://github.com/dlops-io/simple-translate)
  - Build a container
  - Run a container
  - Push container on Docker Hub
  - Pull the new container and run it
- For detail instruction go here (https://github.com/dlops-io/simple-translate#developing-app-using-containers-t3)

Tutorial (T3): Docker commands

Exit from all containers and let us clear of all images

Docker command for system

docker command

```
docker system prune -a
```

Docker command option to remove all images not referenced by any containers

Tutorial (T3): Docker commands

Check how many containers and images we have currently

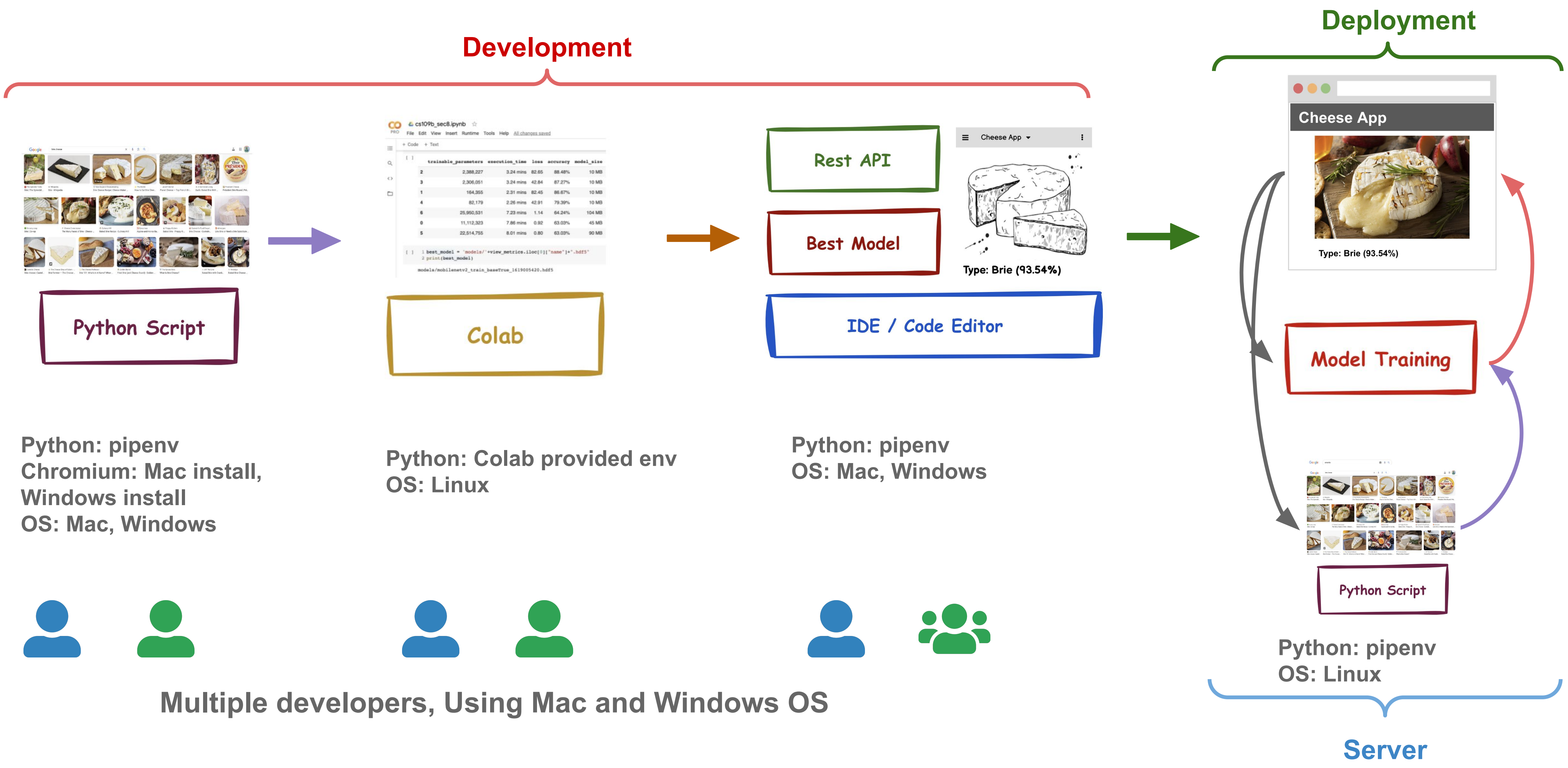
```
docker container ls

docker image ls
```

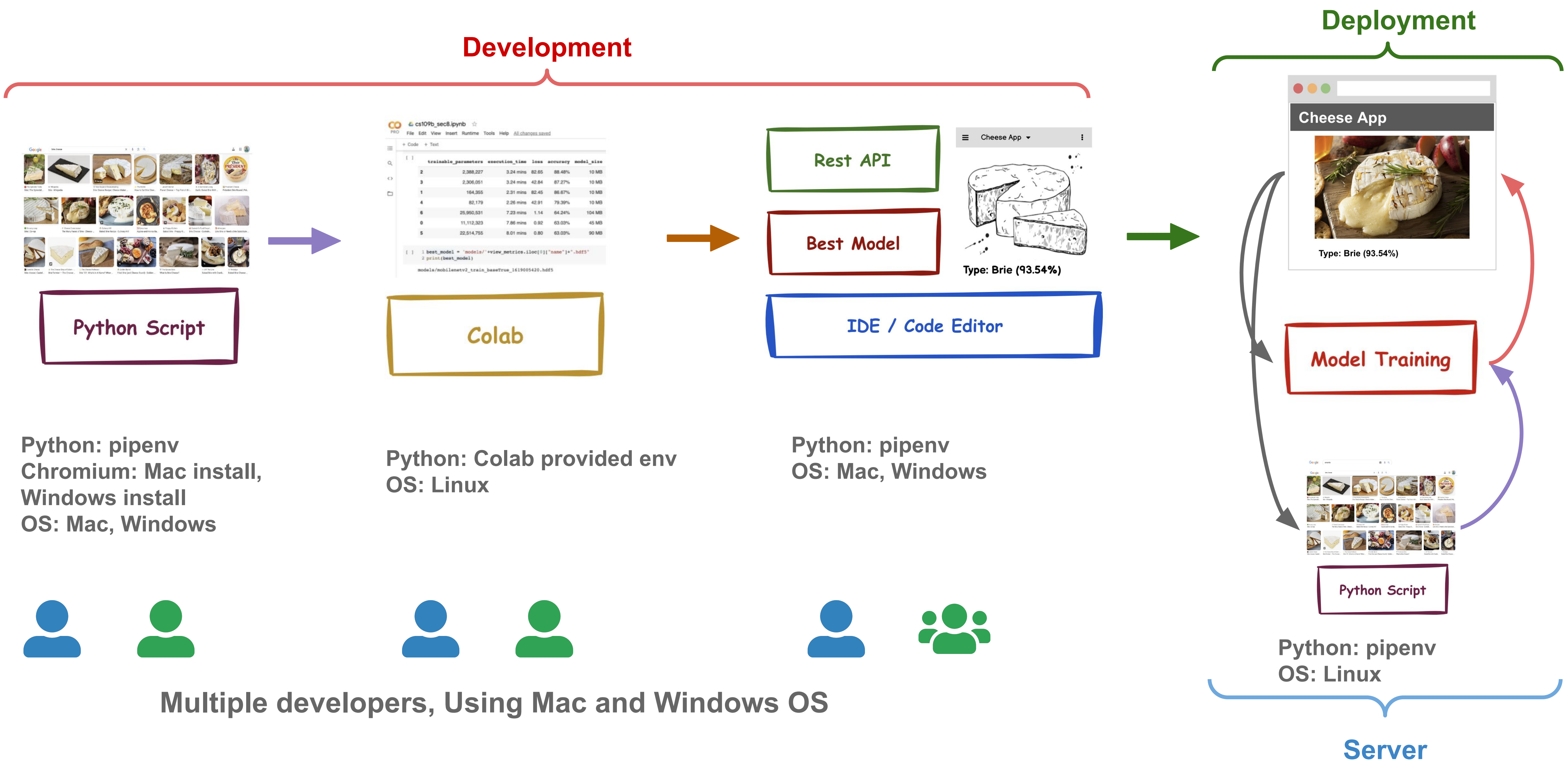
Tutorial (T4): Running App on **VM** using Docker

- Let us run the simple-translate app using Docker
- For this we will do the following:
  - Create a VM Instance
  - SSH into the VM
  - Install Docker inside the VM
  - Run the **containerized simple-translate** app
- Full instructions can be found here (https://github.com/dlops-io/simple-translate#running-app-on-vm-using-docker-t4)

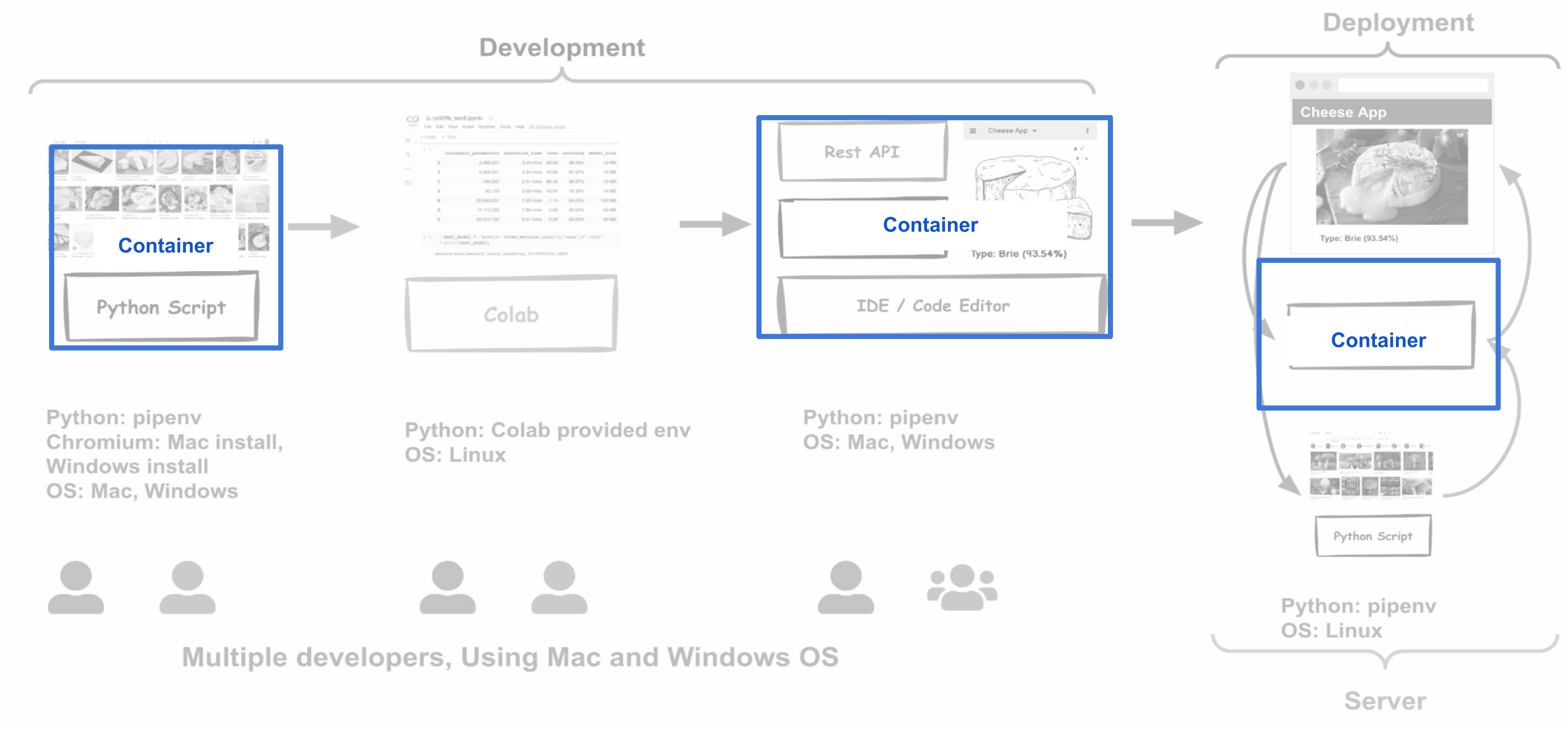
Recap: How do we build an App?



Recap: How do we build an App?



Isolate work into containers



Logistics/Reminders

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

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

Thank you