

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 costs:**
 - 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, making it less efficient than containers.
- **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:**
 - Easier 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) all OS and extra libraries and set up the python environment.

Isolation:

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

Resource Efficiency:

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

Quick Startups:

Near-instantaneous container initialization, reducing time to deployment.

Containers

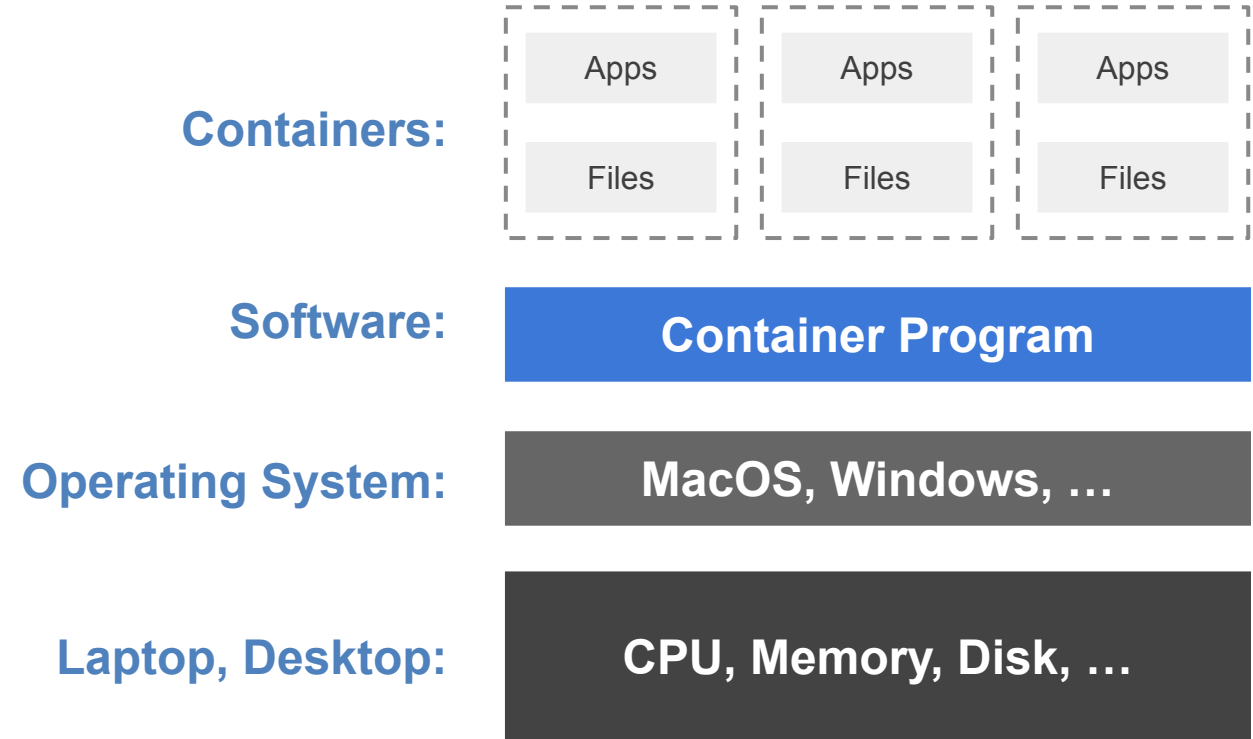
Outline

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

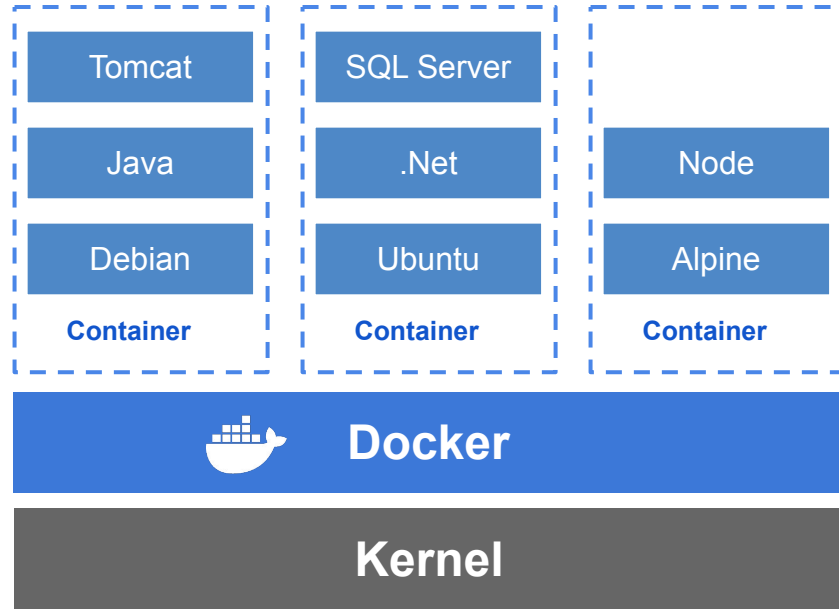
What is a CONTAINER

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 is **isolated** in terms of its operating system, 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.

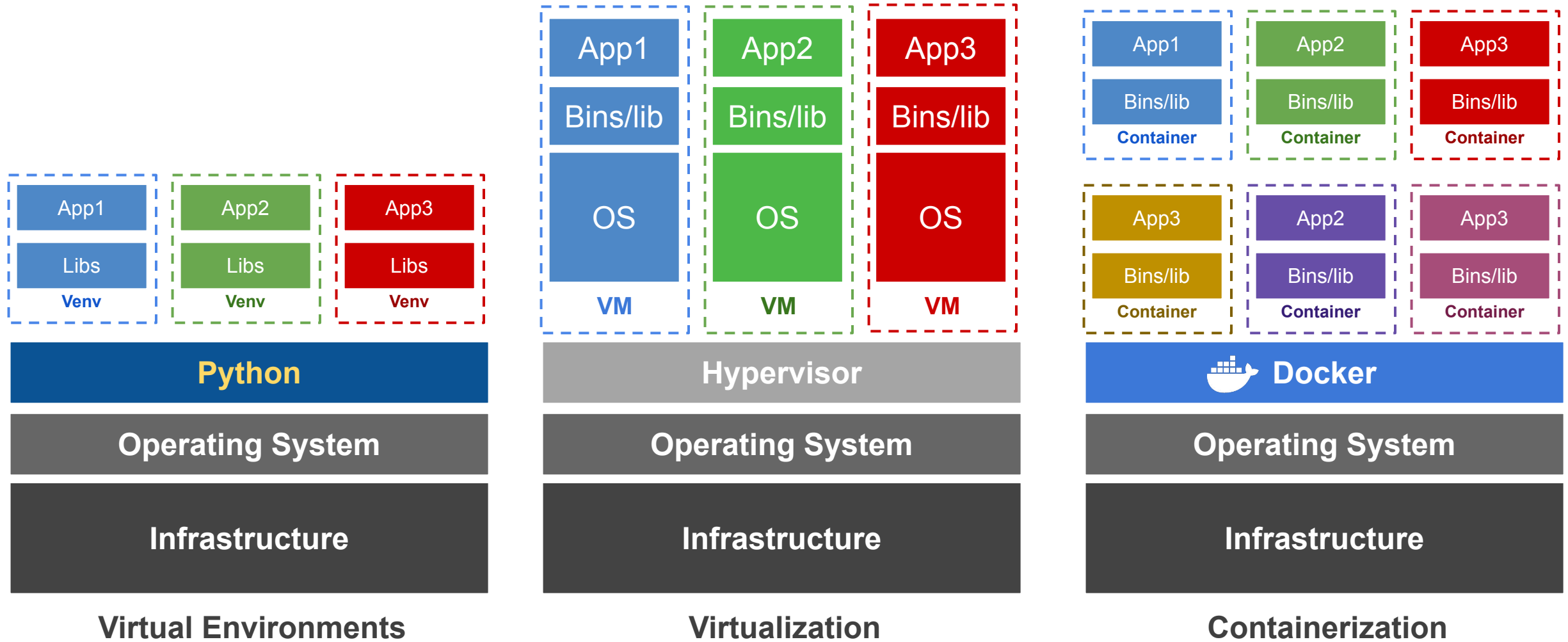


What is a Container



- **Standardized** packaging for software dependencies
- **Isolate** apps from each other
- **Works** for all major Linux distributions, MacOS, Windows

Environments vs Virtualization vs Containerization



Outline

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

Advantages of a CONTAINER

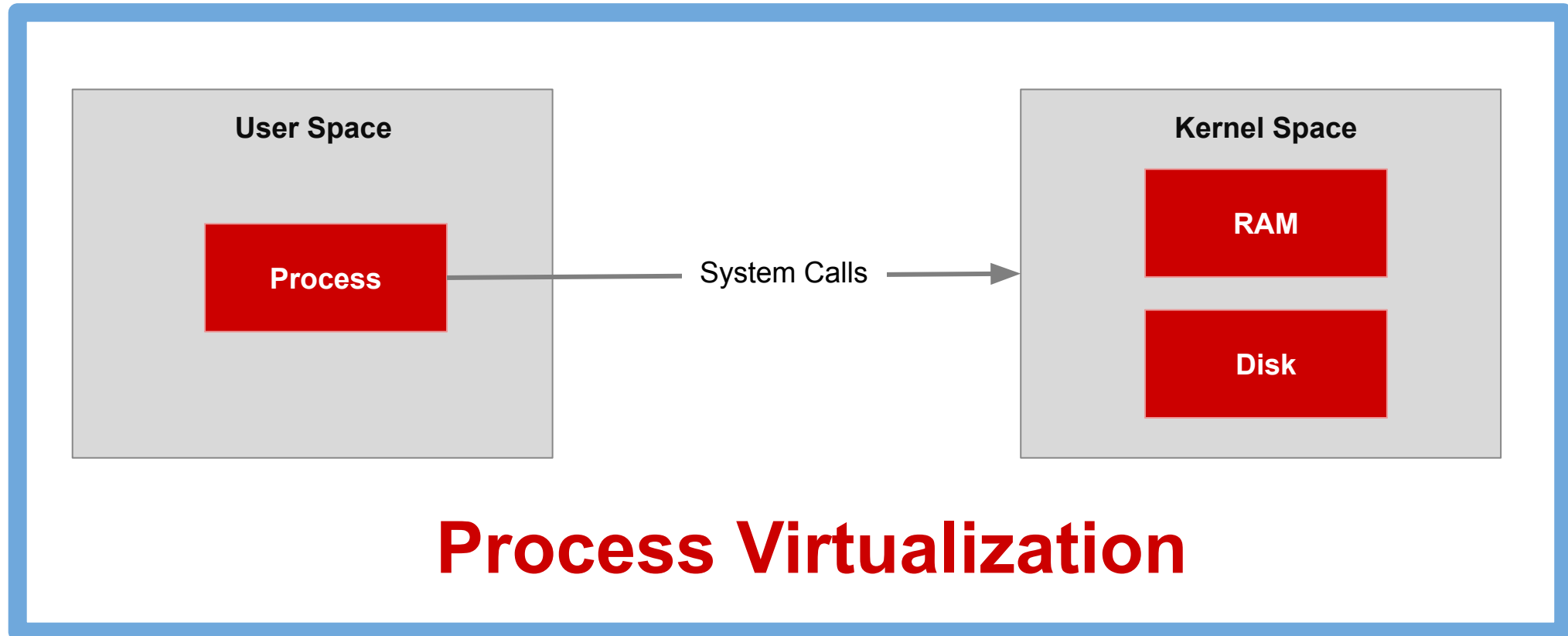
- **Portability & Lightweight:** Containers encapsulate everything needed to run an application, making them easy to move across different environments.
- **Fully Packaged:** Containers include the software and all its dependencies, ensuring a consistent environment throughout the development lifecycle.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

Docker is an open source platform for building, deploying, and managing containerized applications.

What Makes Containers so Small?

Container = User Space of OS

- User space refers to all of the code in an operating system that lives outside of the kernel



Outline

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

What is docker?

Open Source: Community-driven and compatible.

Platform: Develop, ship, and run applications containers.

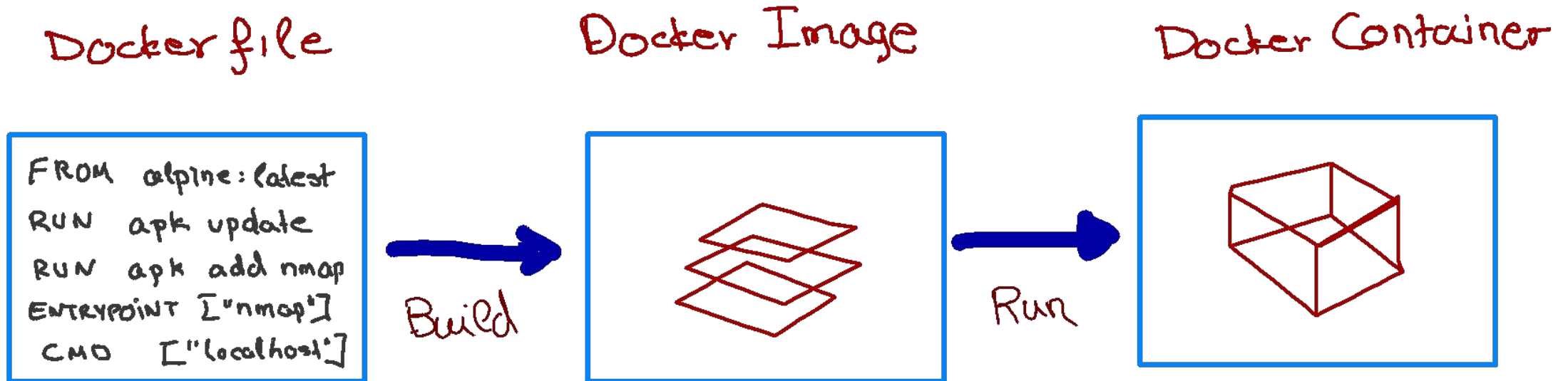
Portability: Consistent across various environments.

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, 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.

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

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
```

Docker Image as Layers

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

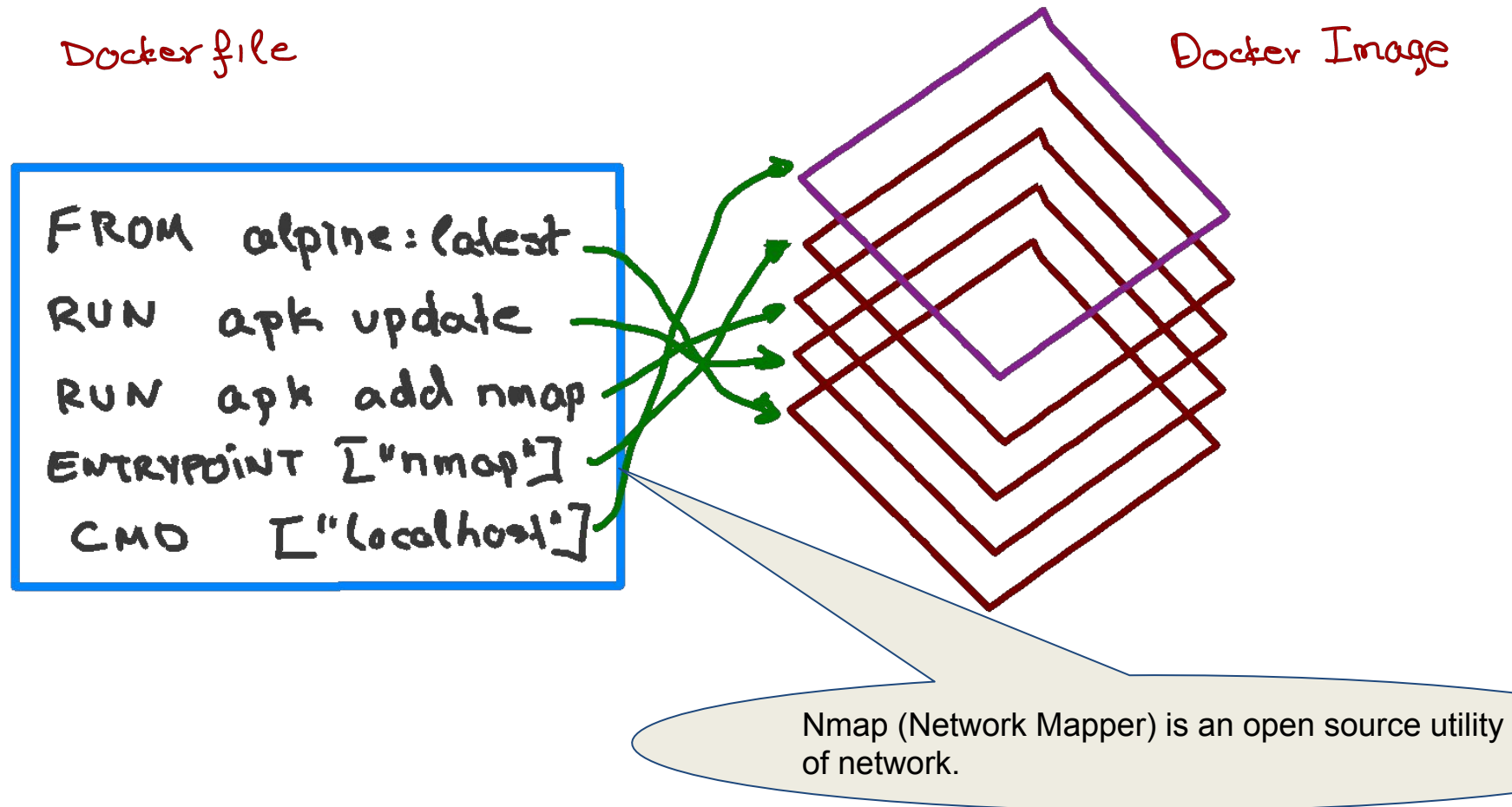
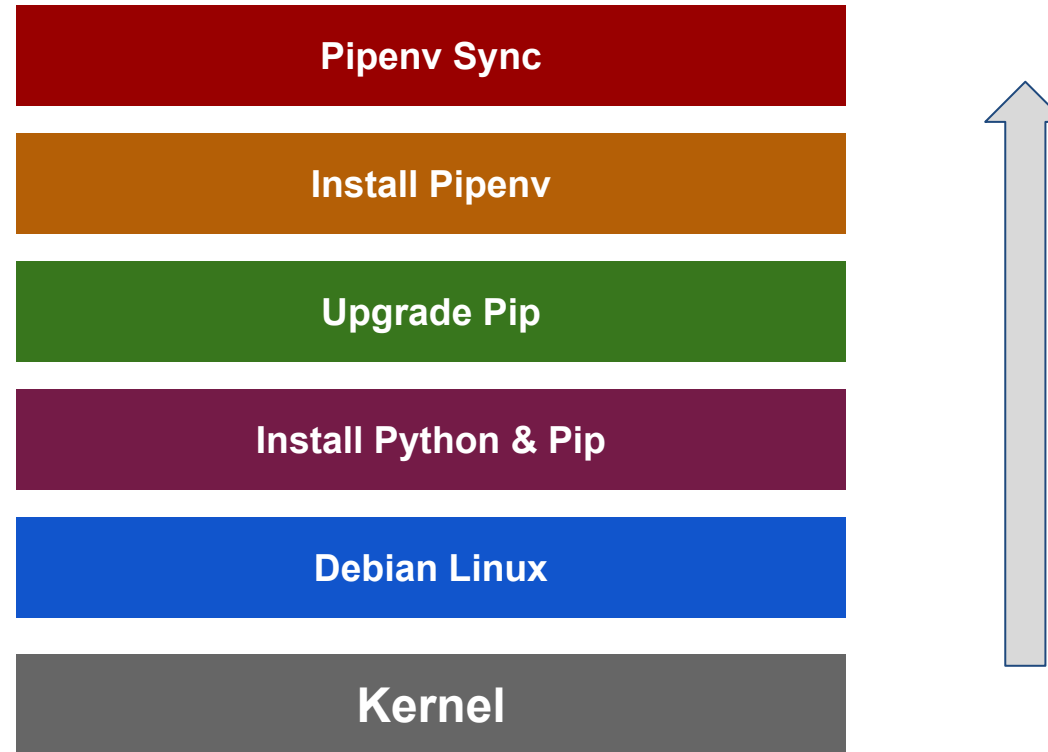


Image Layering - Example

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



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.

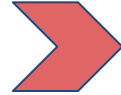
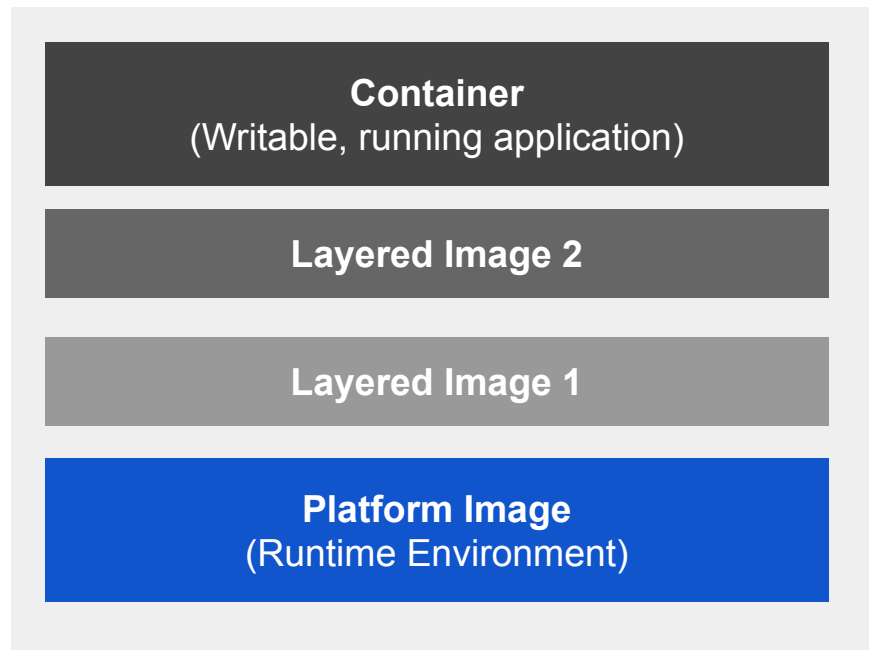
Modularity

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



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

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

Images

How you **store** your application



Docker Container

The standard unit in which the application service resides and executes

Containers

How you **run** your application



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)

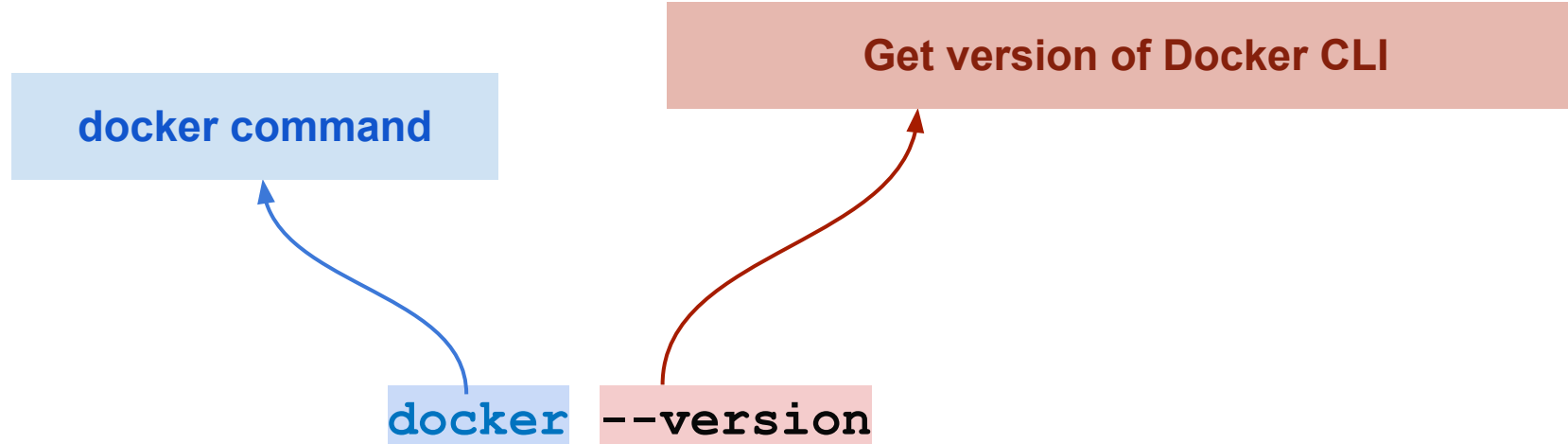
Cloud or server-based storage and distribution service for your images

Tutorial: 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: Docker commands

Check what version of Docker



Tutorial: 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) (<https://github.com/dlops-io/simple-translate>)

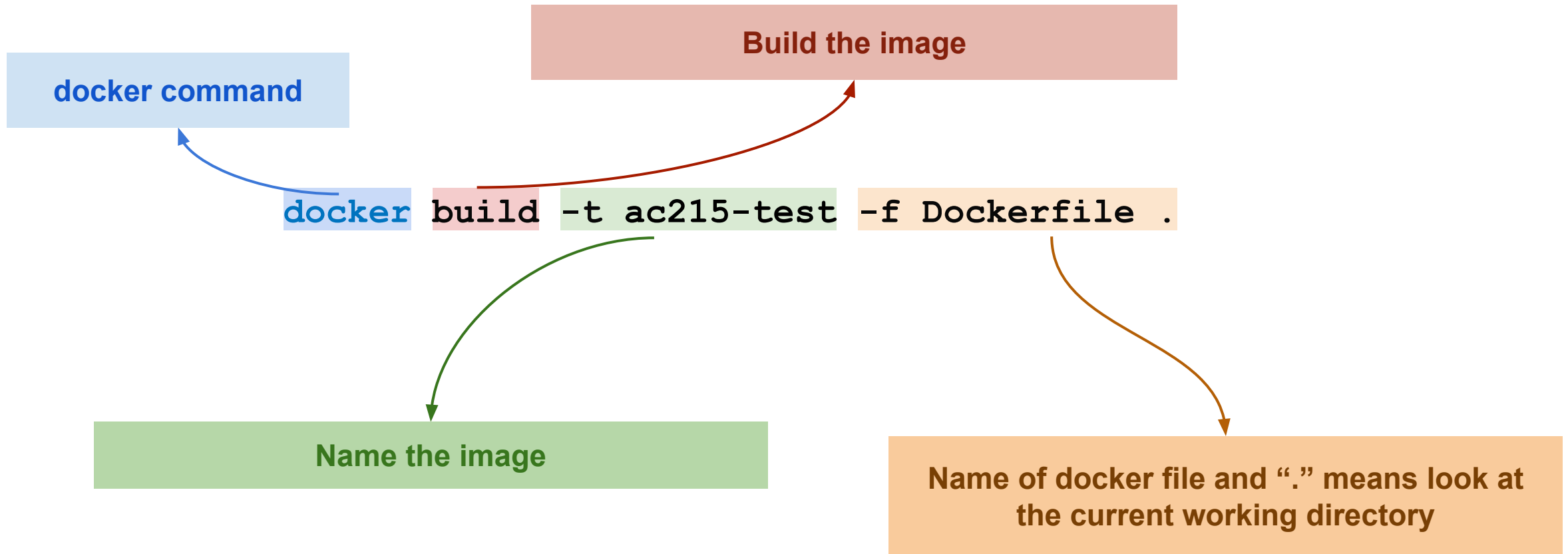
```
git clone https://github.com/dlops-io/simple-translate
```

Tutorial: 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) (https://github.com/dlops-io/simple-translate)
 - Build a container

Tutorial: Docker commands

Build an image based on a Dockerfile



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
RUN set -ex; \
    apt-get update && \
    apt-get upgrade -y && \
    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 . /

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

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

Step2: Instruction 2

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 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	

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
```

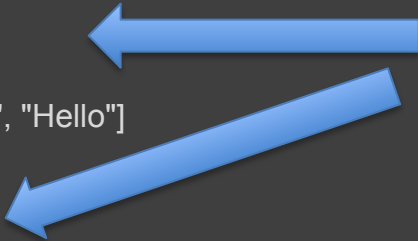
```
Step 4/4 : CMD ["world"]
```

```
---> Using cache
```

```
---> a89172ee2876
```

```
Successfully built a89172ee2876
```

```
Successfully tagged hello_world_cmd:latest
```

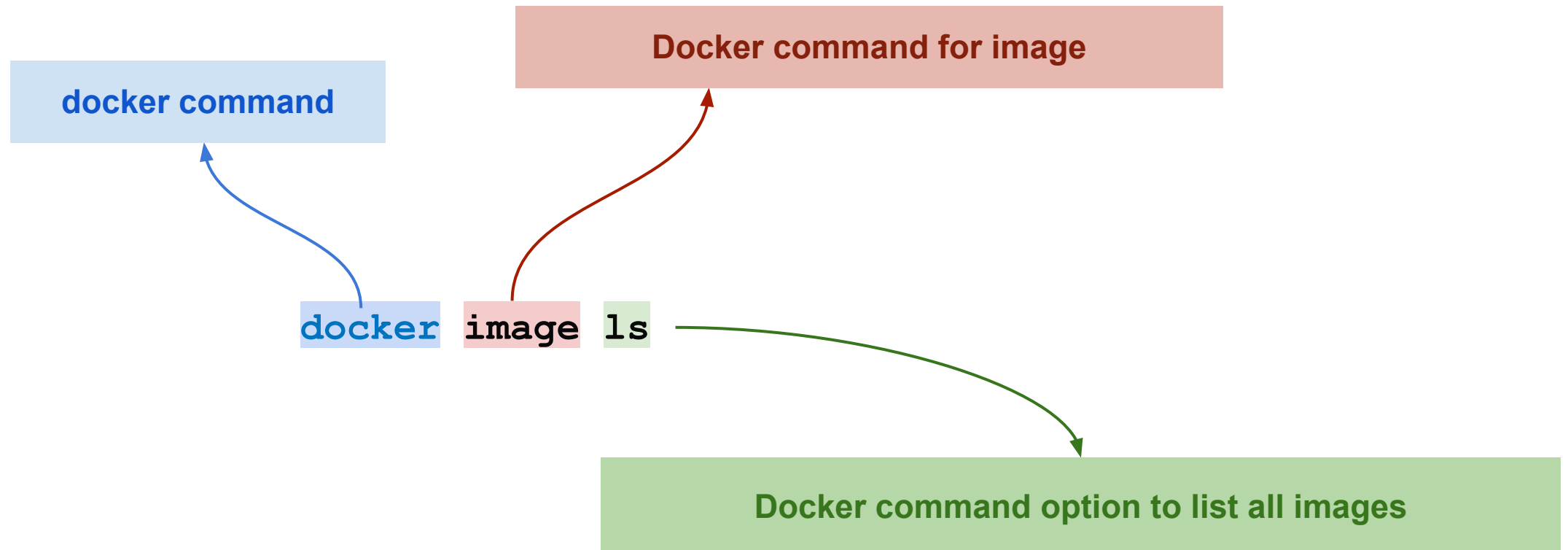


Have seen this before. Use
cache

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.

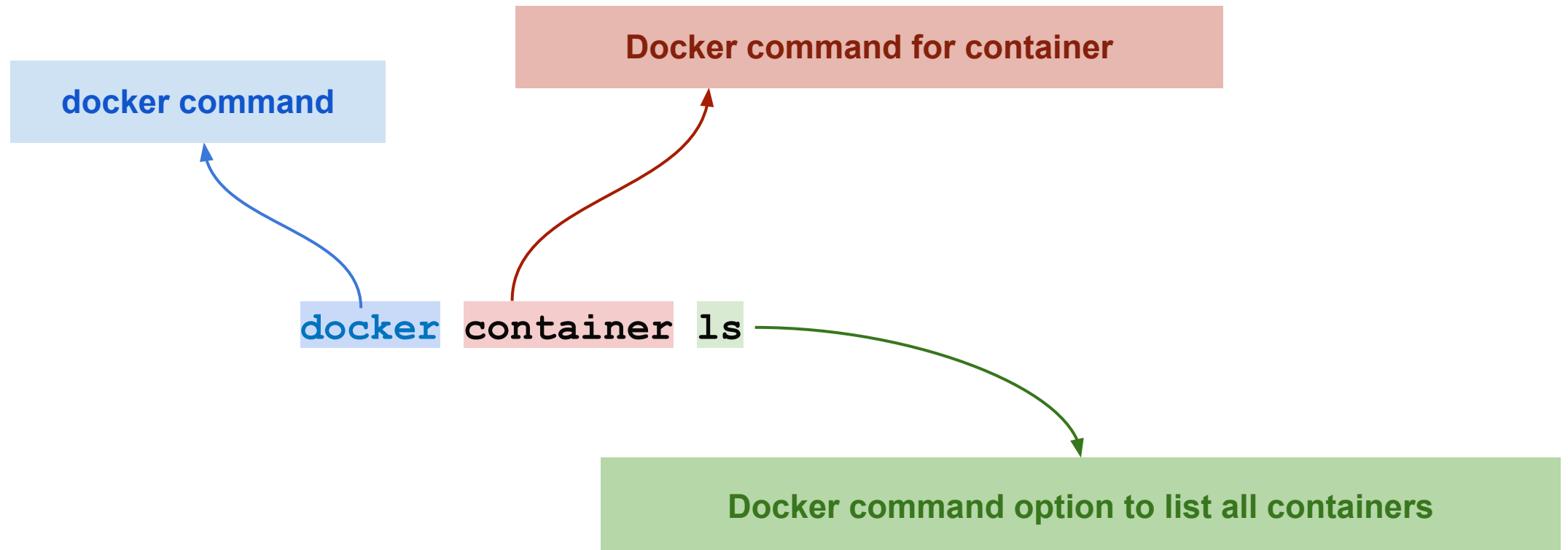
Tutorial: Docker commands

List all docker images



Tutorial: Docker commands

List all running docker containers

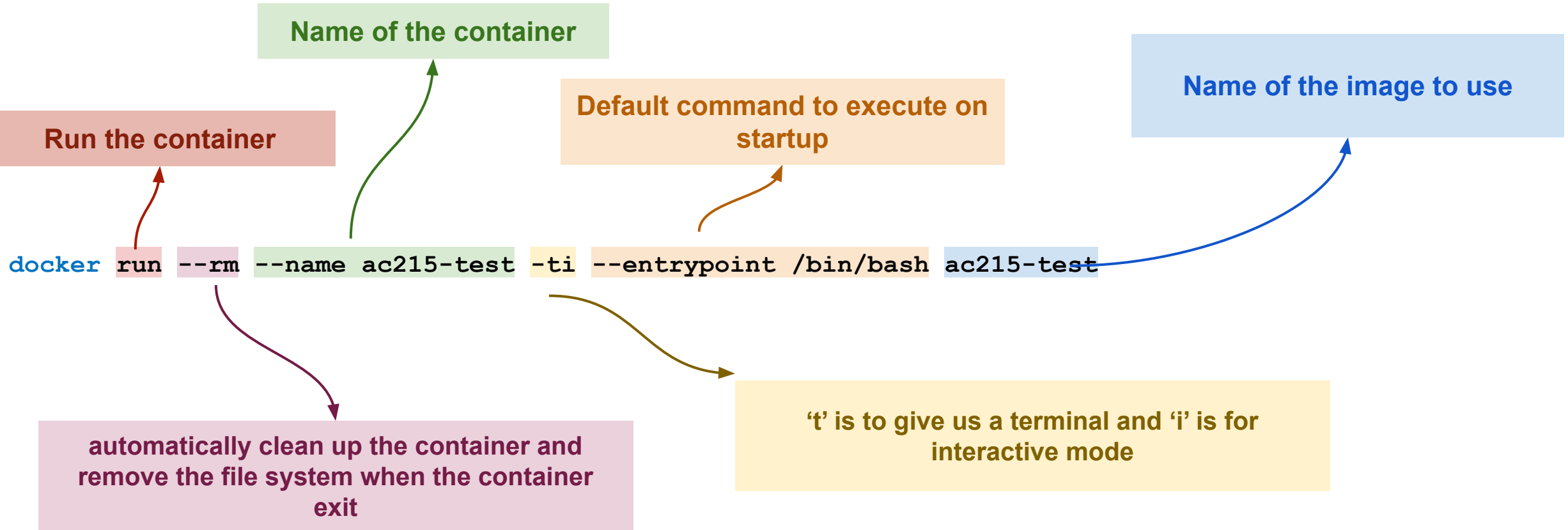


Tutorial: Developing App using Containers

- 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) (https://github.com/dlops-io/simple-translate)
 - Build a container
 - Run a container

Tutorial: Docker commands

Run a docker container using an image from Docker Hub



Tutorial: Docker commands

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

```
docker container ls
```

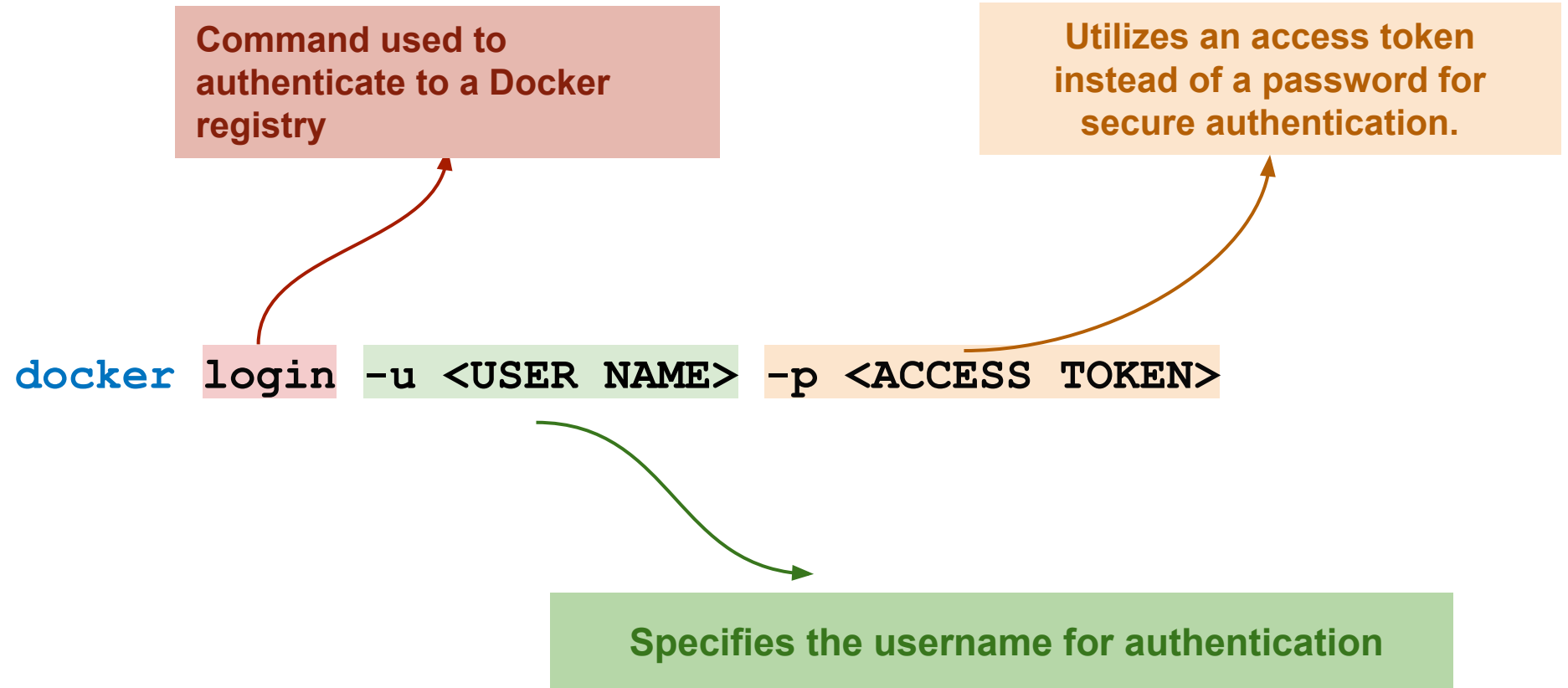
```
docker image ls
```

Tutorial: 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) (https://github.com/dlops-io/simple-translate)
 - Build a container
 - Run a container
 - Push container on Docker Hub

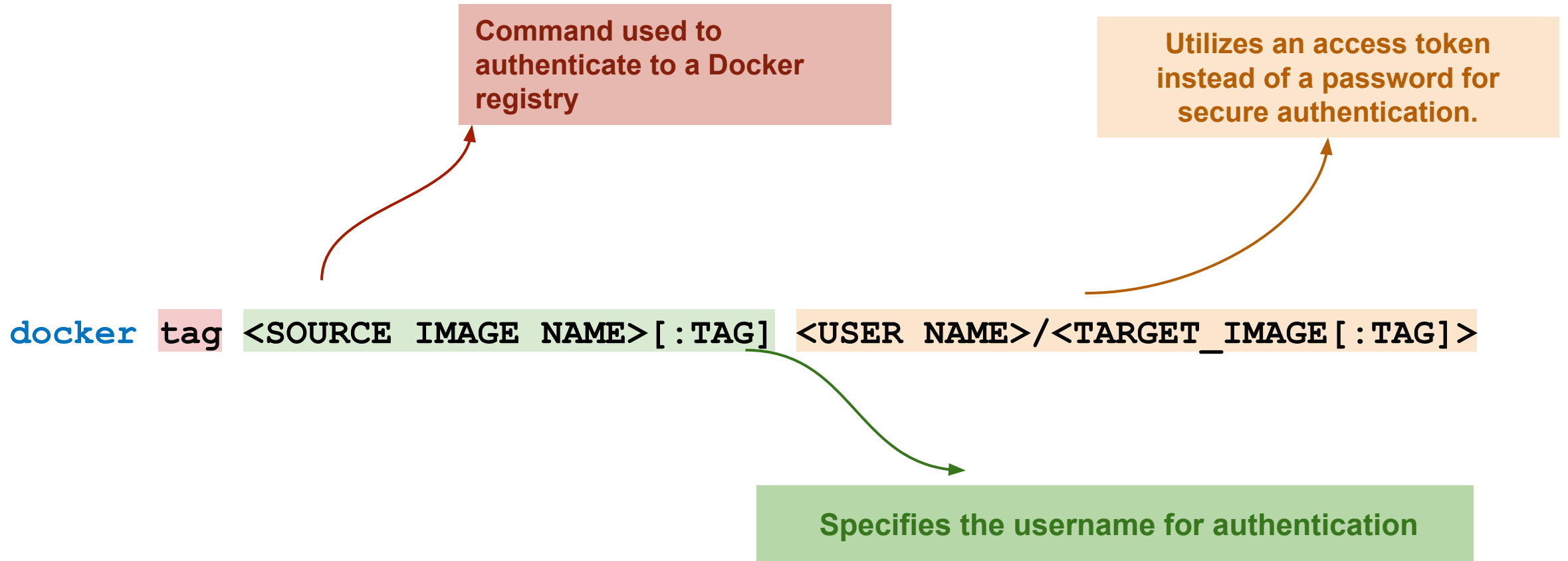
Tutorial: Docker commands

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



Tutorial: Docker commands

Tag the Docker Image



Tutorial: 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

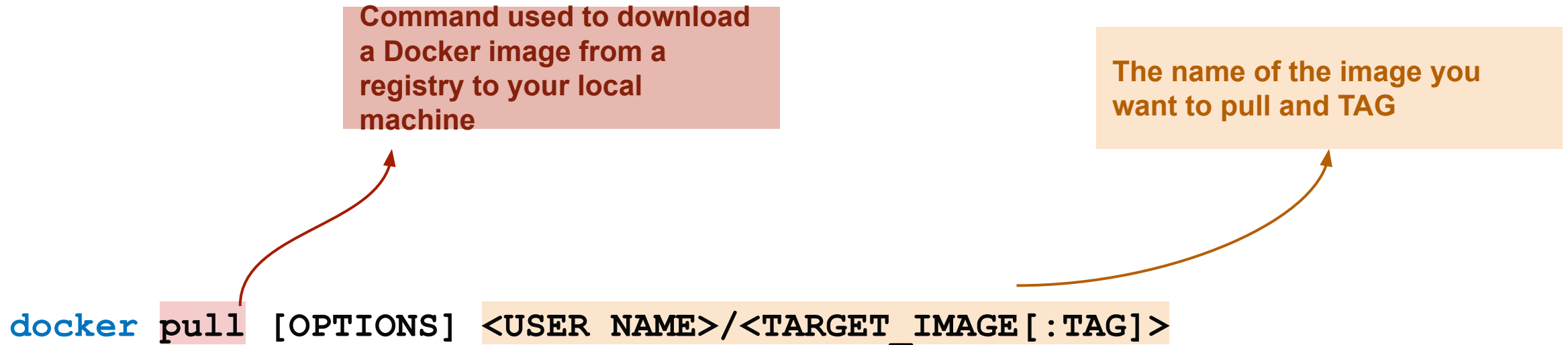
`docker push <USER NAME>/<TARGET_IMAGE[:TAG]>`

Tutorial: 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) (<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: Docker commands

- Pull from Docker Hub

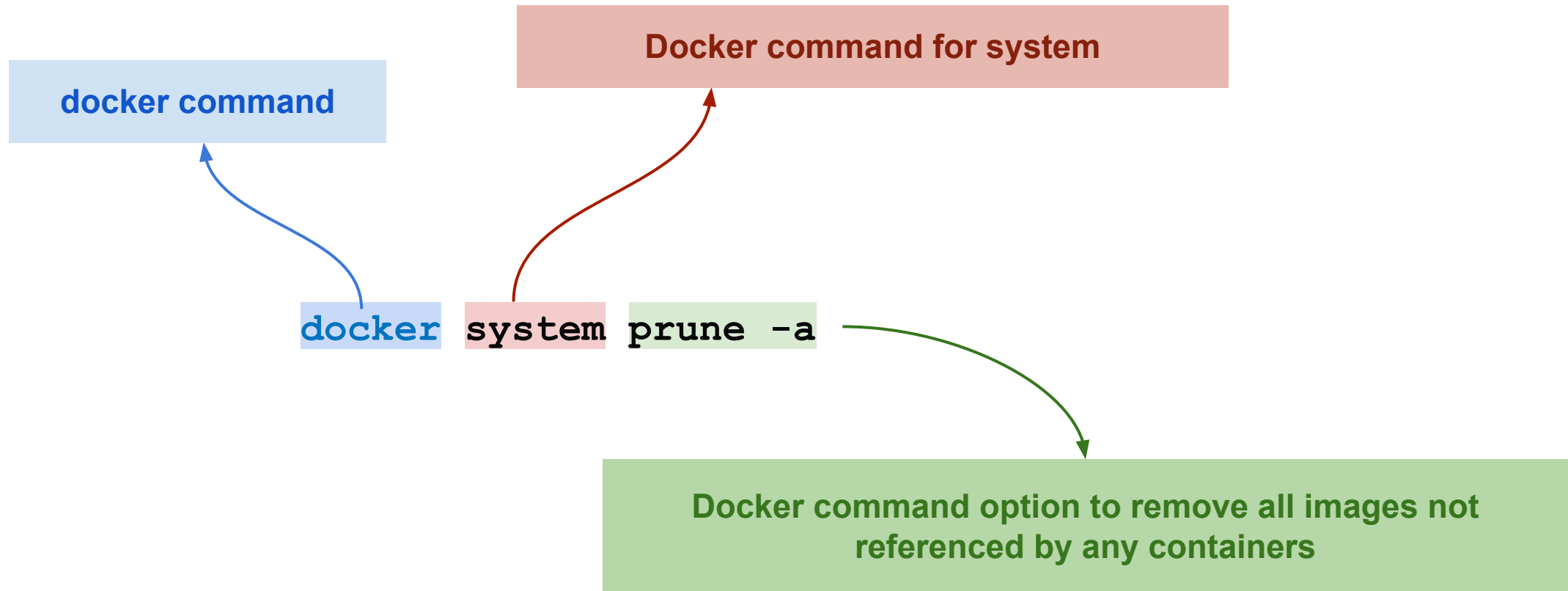


Tutorial: Developing App using Containers

- 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) (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)
(https://github.com/dlops-io/simple-translate#developing-app-using-containers)

Tutorial: Docker commands

Exit from all containers and let us clear of all images



Tutorial: Docker commands

Check how many containers and images we have currently

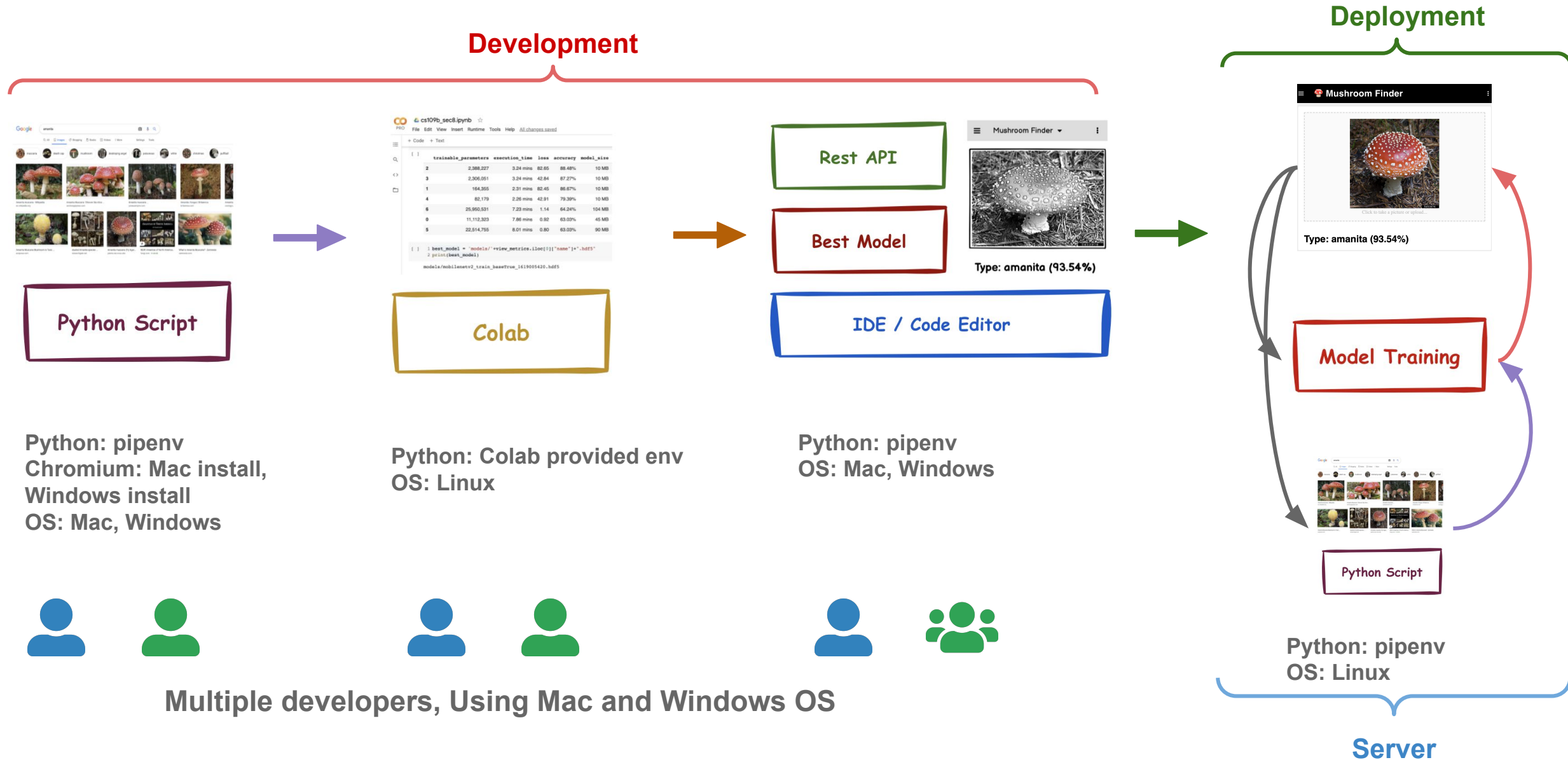
```
docker container ls
```

```
docker image ls
```

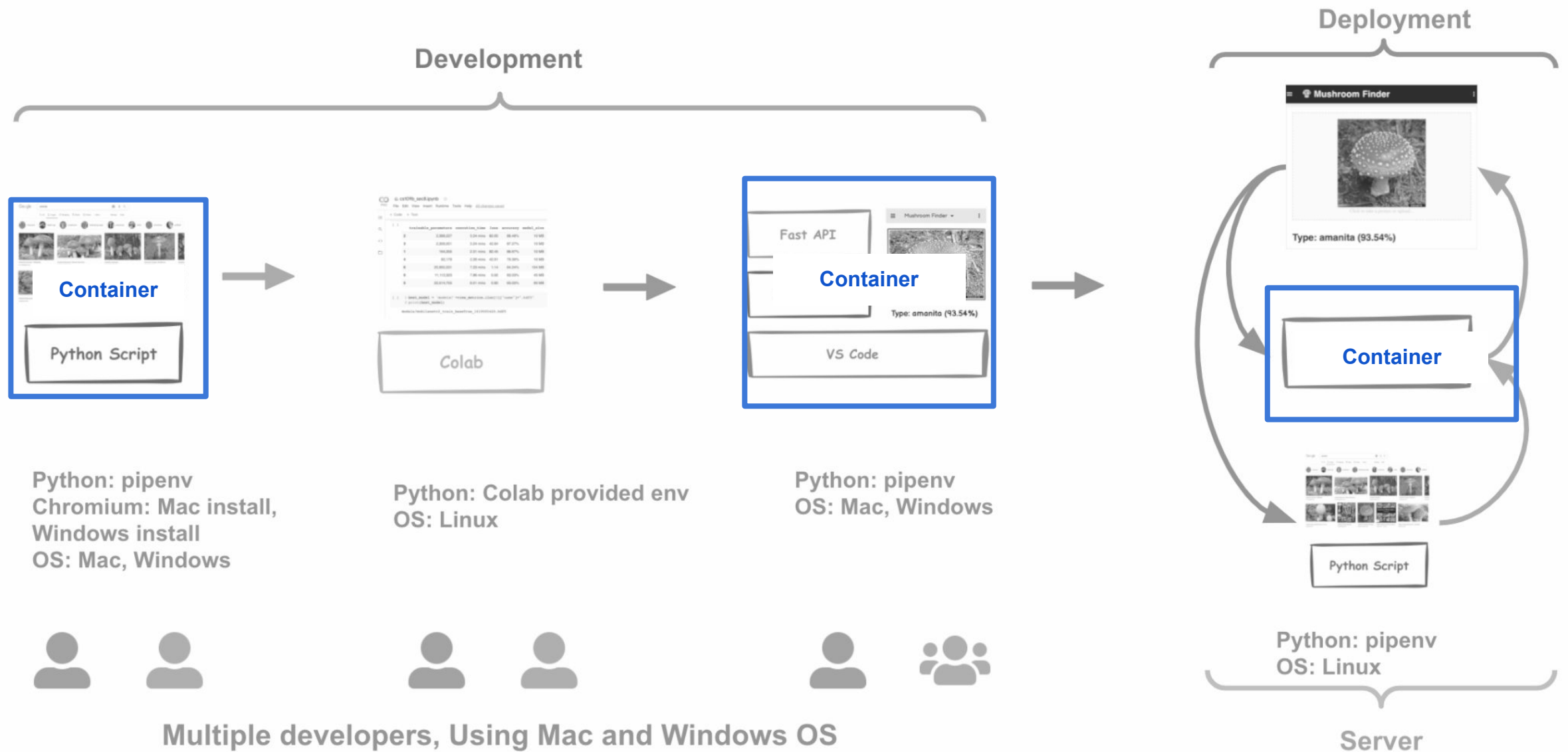

Tutorial: Running App on **VM** using Docker

- 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)
(<https://github.com/dlops-io/simple-translate#running-app-on-vm-using-docker>)

Recap: How do we build an App?



Isolate work into containers



THANK YOU