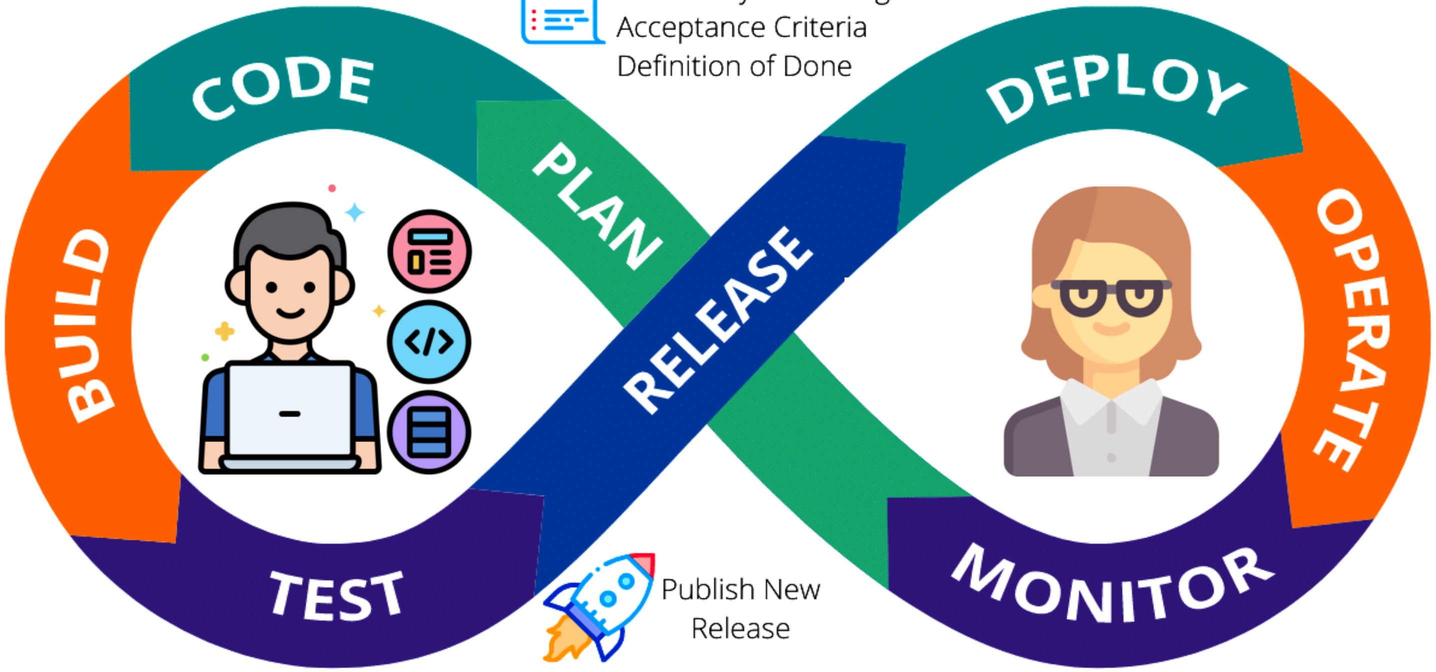
Containerization 1

By Majid Babaei



Architecture Design
Solutioning
Coding

Continuous Delivery
Deploy to Production

Requirements Gathering
User Story Grooming
Acceptance Criteria
Definition of Done

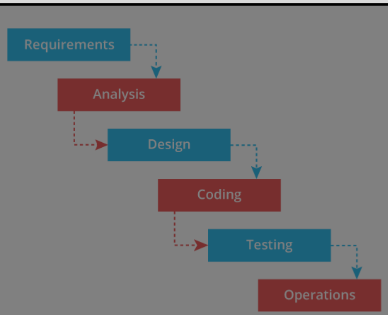
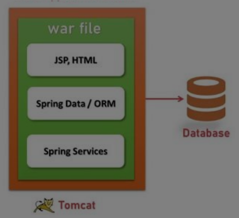
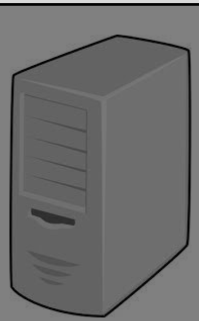
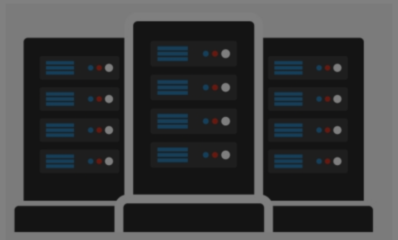
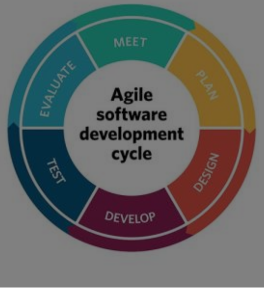
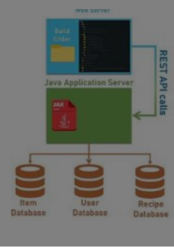
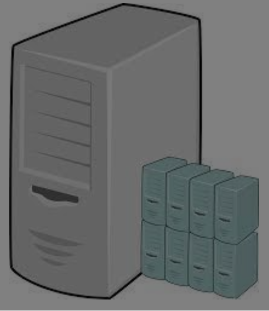
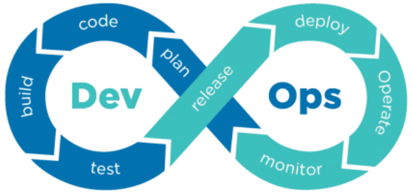
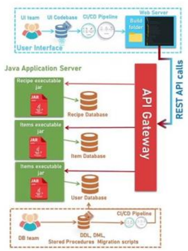
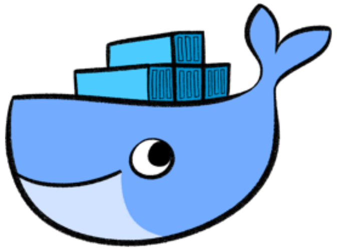
Continuous Integration
Unit Testing
Package Application

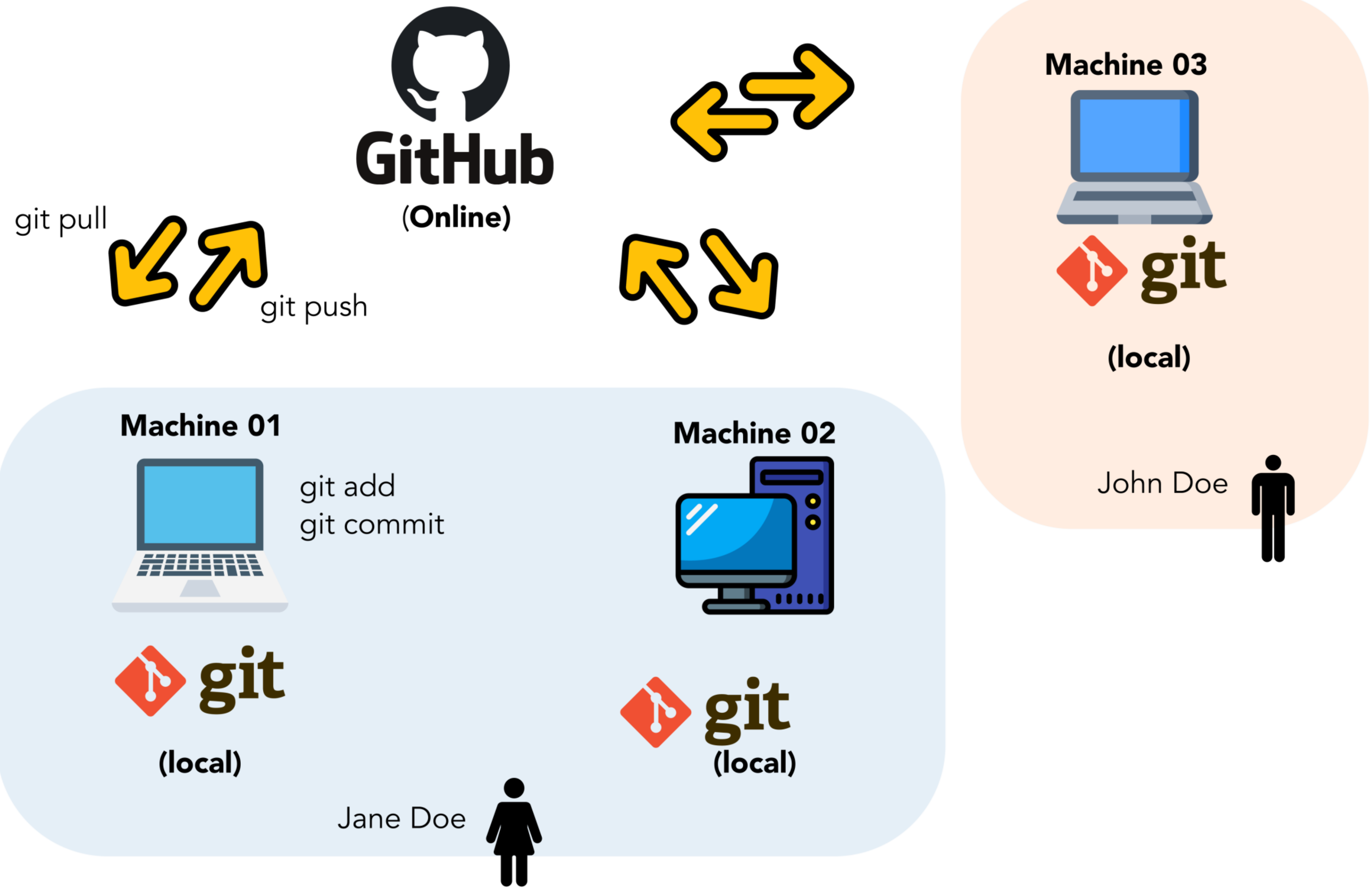
Gather Feedback from
Customers
Scale environments

Publish New
Release

Functionality Testing
User Acceptance Testing
Demonstrations

Observe system performance
Observe usability patterns
Trace Failures

Dates	Software Engineering Methodology	App Architecture	App Deployment	Data Storage
~1990 – 2000	 Requirements Analysis Design Coding Testing Operations Waterfall	 war file JSP, HTML Spring Data / ORM Spring Services Database Tomcat Monolithic	 Physical Server	 Local DCs
~2000 – 2010	 MEET PLAN DESIGN DEVELOP TEST EVALUATE Agile software development cycle Agile	 Java Application Server REST API calls Item Database User Database Recipe Database N-Tier	 Virtual Server	 Virtual DCs
~2010 – Now	 code plan release deploy Operate monitor test build Dev Ops	 User Interface Java Application Server API Gateway REST API calls Recipe Database Item Database User Database CI/CD Pipeline Microservices	 Containers	 Cloud DCs



GitHub

(Online)

git pull

git push

Machine 01

git add
git commit

git

(local)

Machine 02

git

(local)

Jane Doe

Machine 03

git

(local)

John Doe

What Are Containerized Applications?

- Containerized applications are applications that run in isolated runtime environments called *containers*.

- Containers encapsulate an application with all its dependencies, including system libraries, binaries, and configuration files.

- This all-in-one packaging makes a containerized application portable

- It to behave consistently across different hosts—allowing developers to write once and run almost anywhere.

Two main challenges with application hosting

- Developers often struggle to make applications run consistently across different hosting environments.

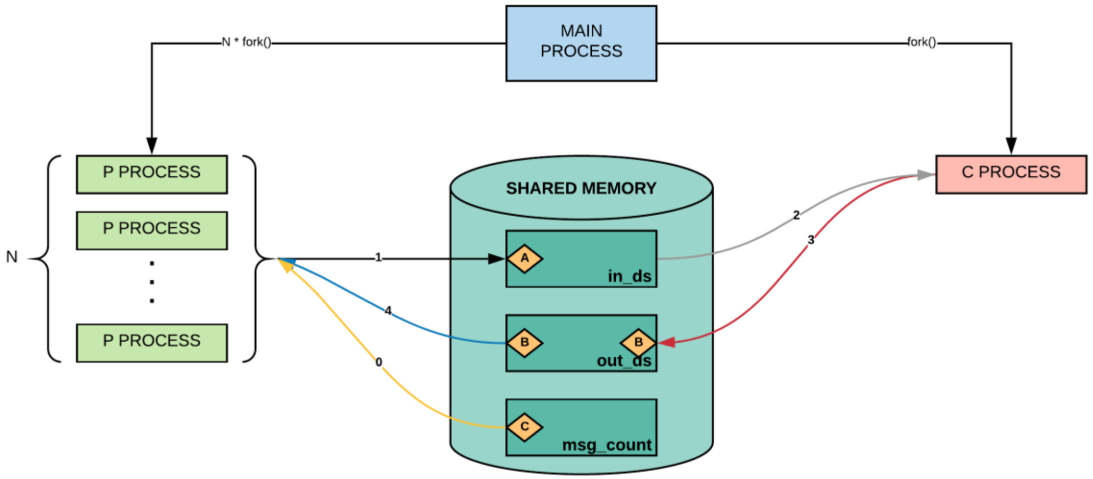
Containerizing an application avoids this problem by providing a consistent and standardized environment for that application to run in.

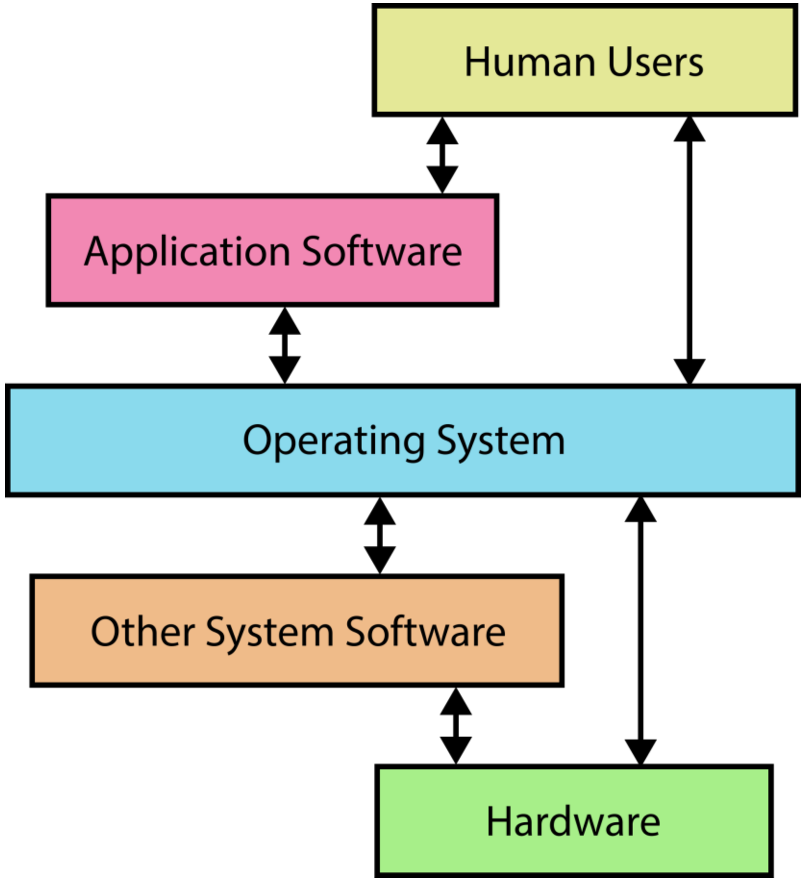
- Although any hosted application needs to be isolated from all others to run securely and reliably, achieving this isolation with physical servers is resource-intensive.

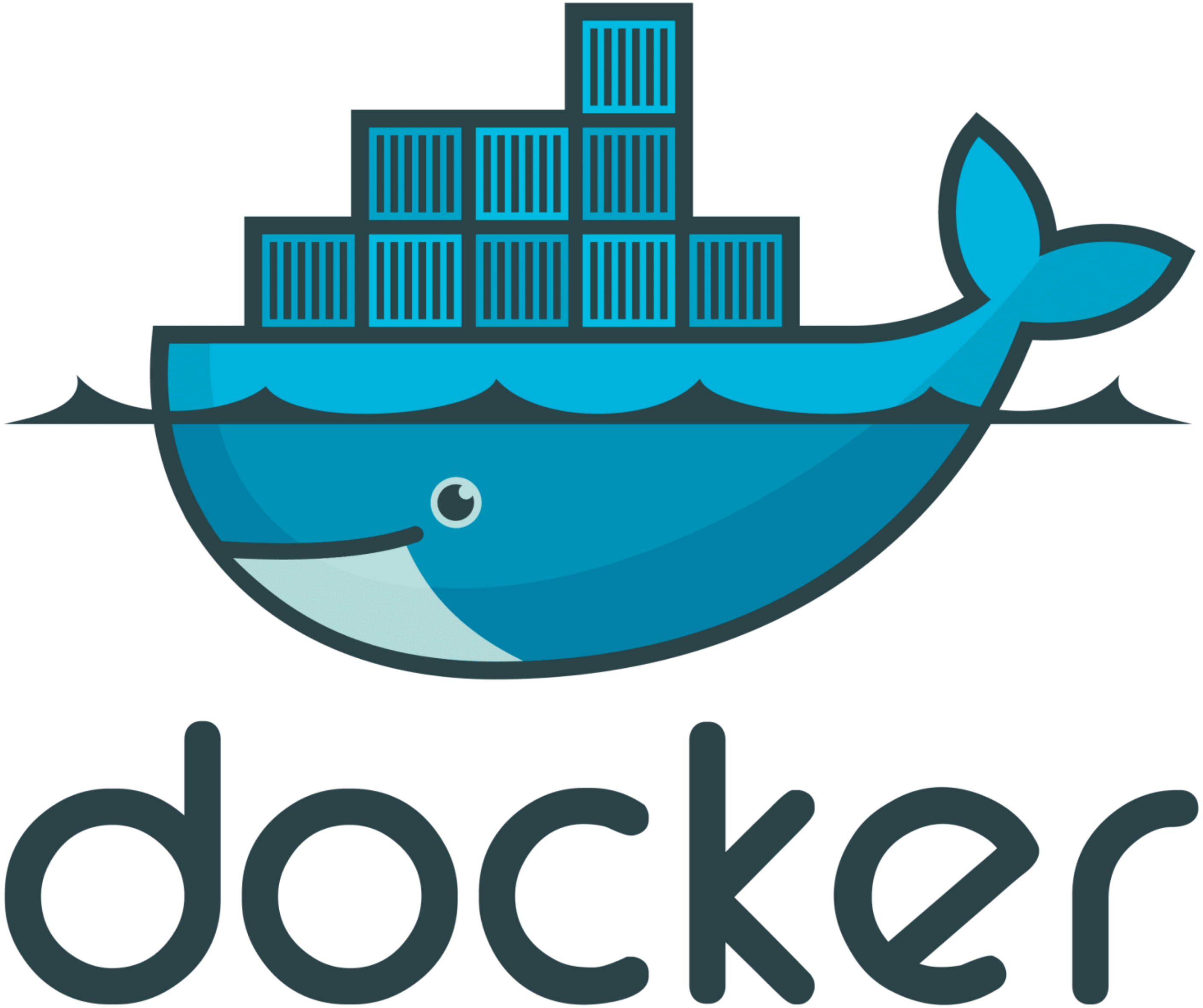
Using OS-native features, such as *Linux namespaces* *and* *cgroups*, to isolate each container from other processes running on the same host.

The process of containerizing applications makes application development faster, more efficient, and secure by separating different functionalities from hardware dependencies and other pieces of software. Containers can run on any host operating system and are isolated from other software and hardware objects, making them versatile tools to build applications that can be built once and run anywhere.

What are the potential effects of HWs and other SWs on the functionalities of our app?


MAIN PROCESS
N * fork()
fork()
P PROCESS
P PROCESS
P PROCESS
N
C PROCESS
SHARED MEMORY
A
in_ds
B
B
out_ds
C
msg_count
1
2
3
4
0

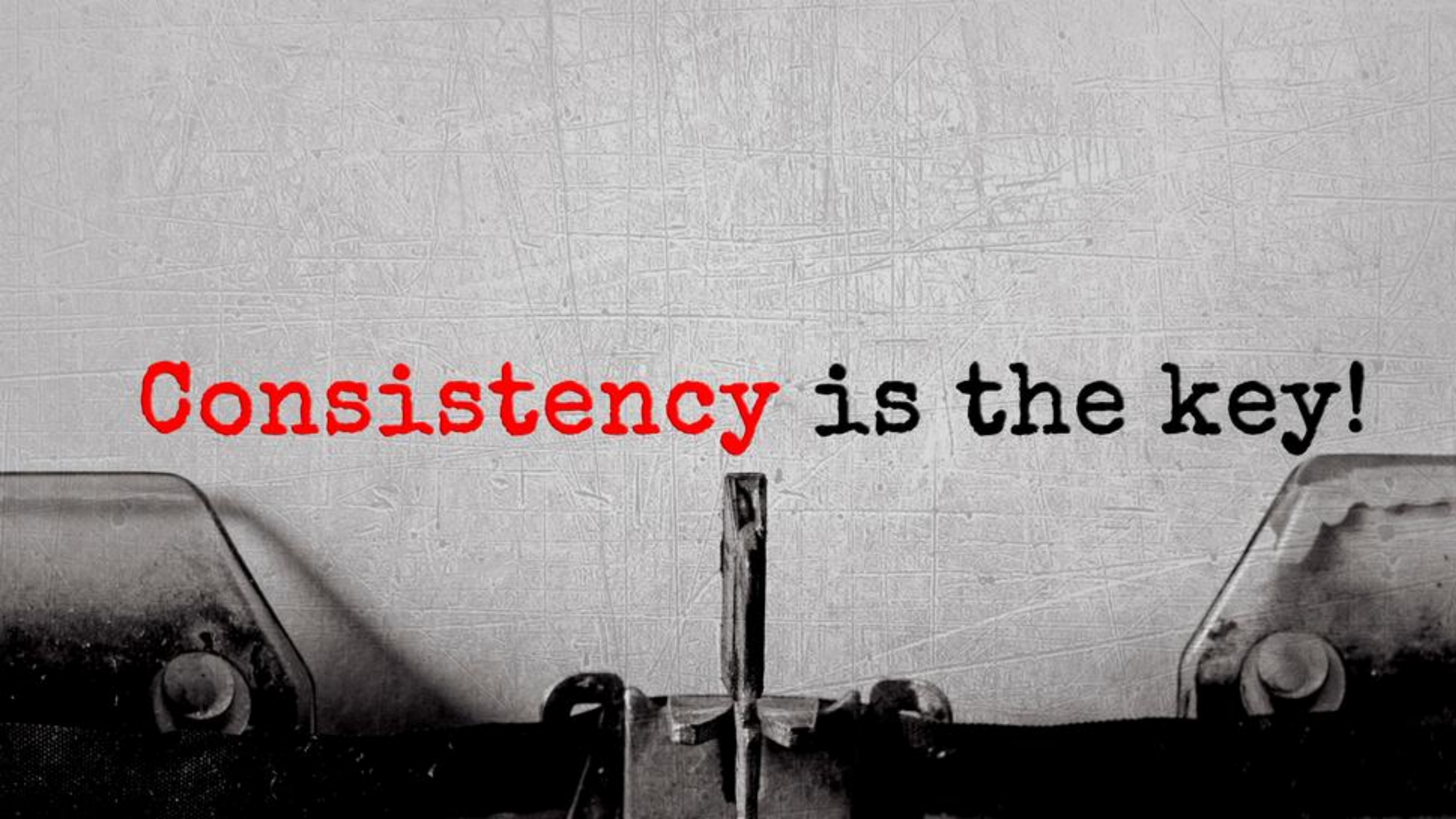

Human Users
Application Software
Operating System
Other System Software
Hardware


docker

A platform for building, running and shipping applications

But in a consistent manner!

If your app works on your development machine it can run and function the same way on other machines

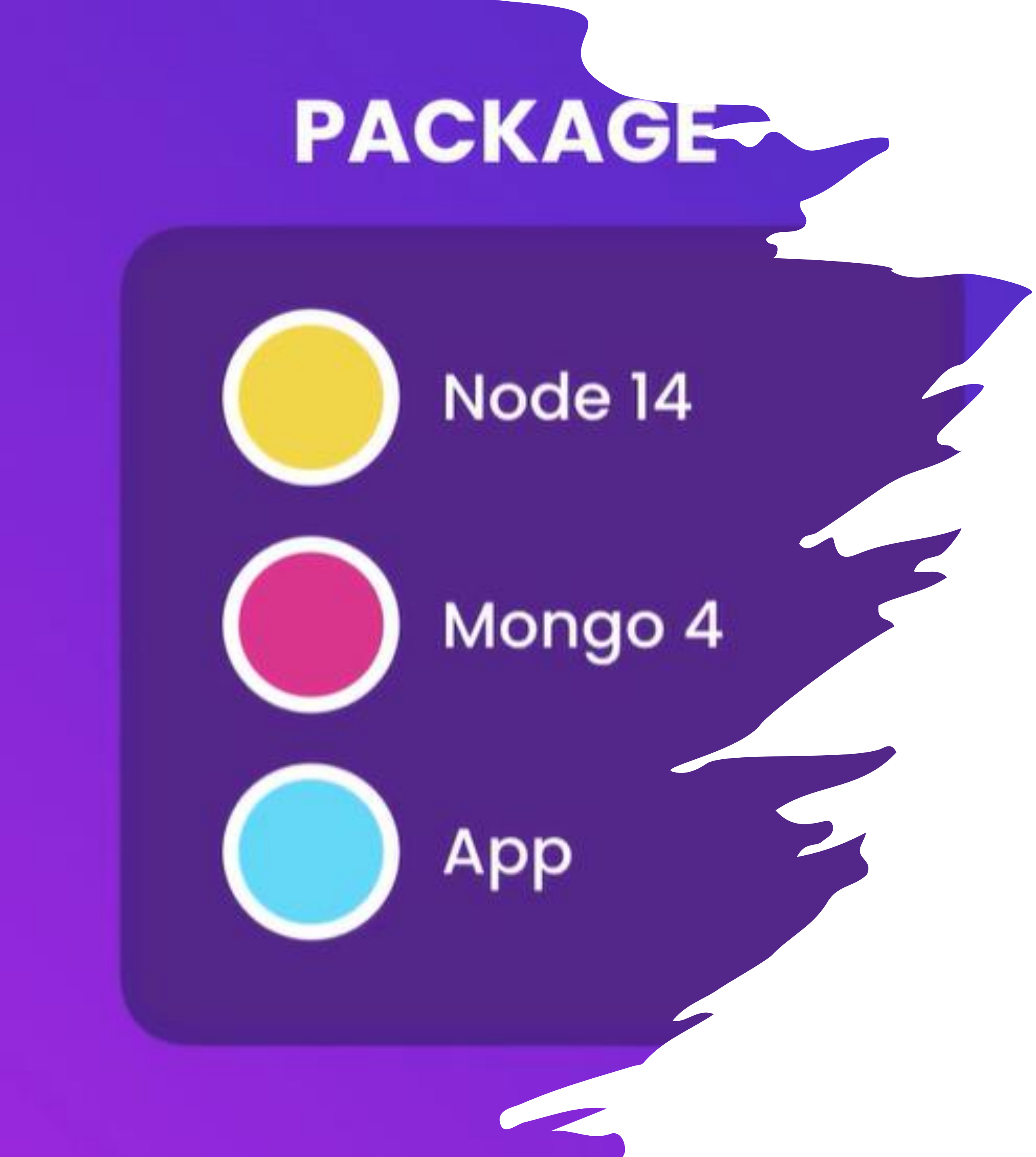

Consistency is the key!

Can you think of the reasons for app inconsistency?

- ***One or more file not included as part of your deployment!***

- ***If your target machine running a different version of a lib that your app needs (e.g., different node versions)***

- ***Configuration settings (e.g., environmental variables) are different***



With Docker we can easily package our application and everything it needs and run it anywhere!

If someone joins your team, they don’t need to spend all day to setup a new machine to run your application!

How can we do that?

They simply tell Docker to bring up your app.
Docker will take care of everything else!

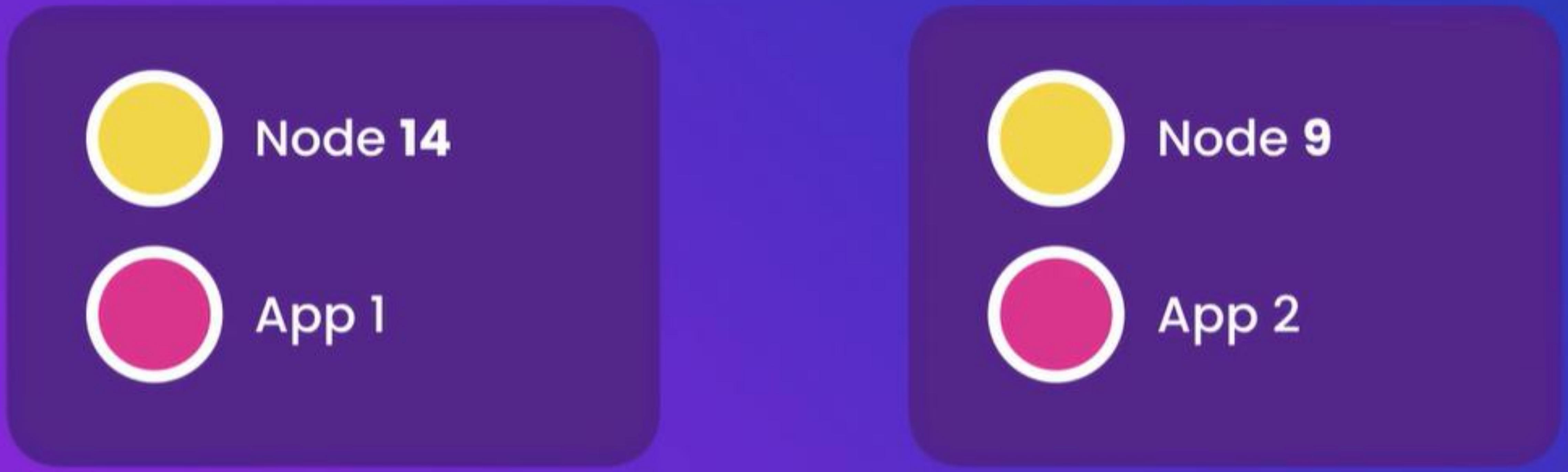
- *Download your app*

- *Download all the dependencies (right version!)*

- *Setup the environment*

- *Run your app in an isolated space*

This isolated environment allows one or more applications use different versions of some software side by side on the same machine!



Node **14**

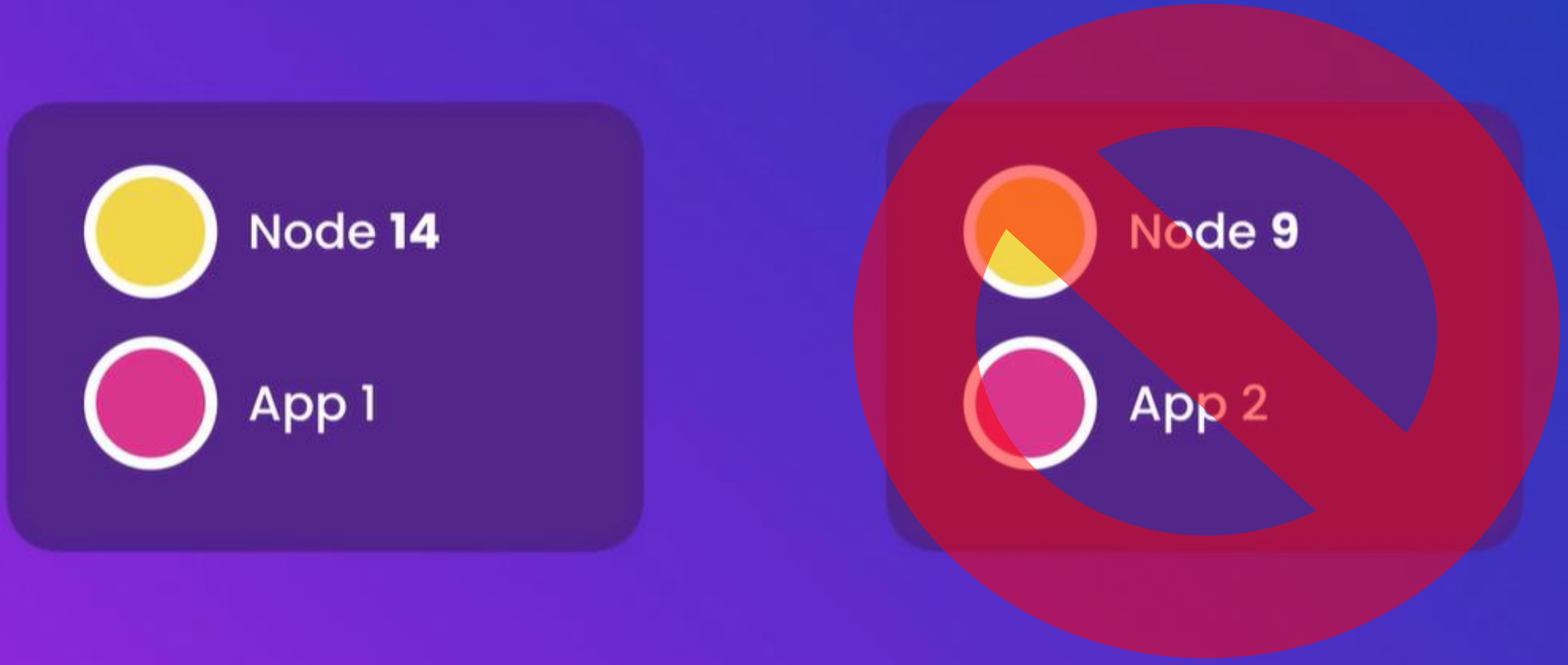
App 1

Node **9**

App 2

Both apps can work on the same machine without messing with each other!

What happens if you want to remove one app?



Node 14

App 1

Node 9

App 2

With Docker you can easily remove an app and all its dependencies in one go!

Without Docker as we are working on different projects, our development machine has become cluttered with so many libs and tools used by different apps

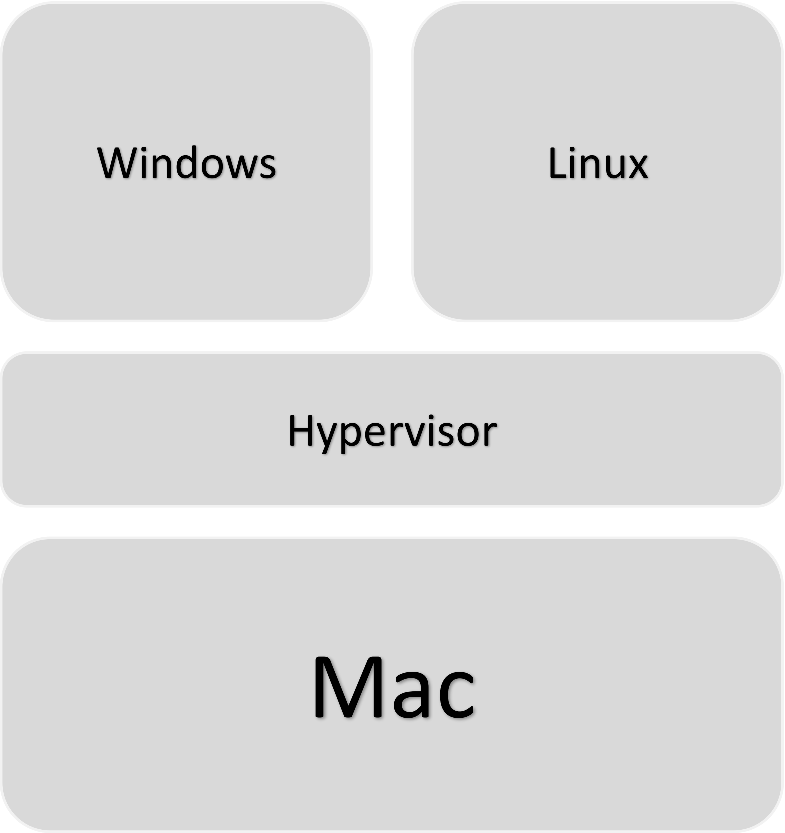
After a while we don't know if we really can remove one or more of these libs or tools!

CONTAINER

An isolated environment for running an application

VIRTUAL MACHINE

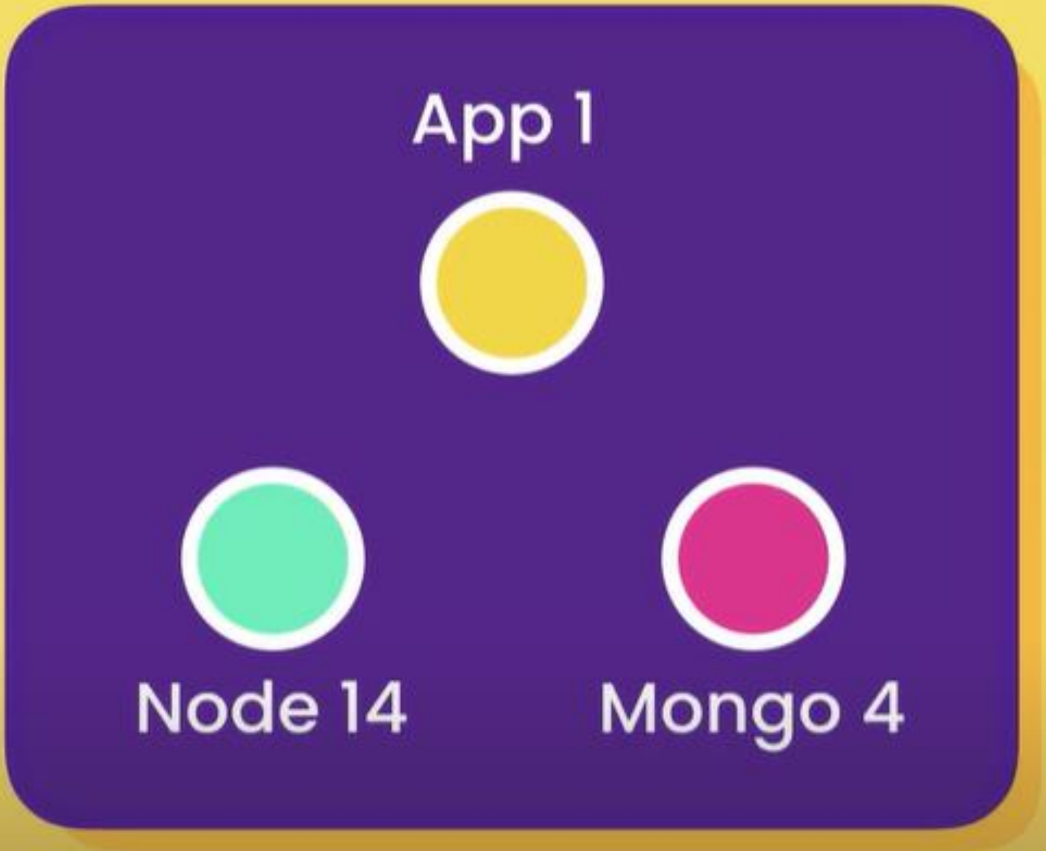
An abstraction of a machine (physical hardware)



We use hypervisor to create and manage VMs and run apps in an isolated environment:

- *VirtualBox*
- *VMware*
- *Hyper-V (only for windows)*

VIRTUAL MACHINE 1

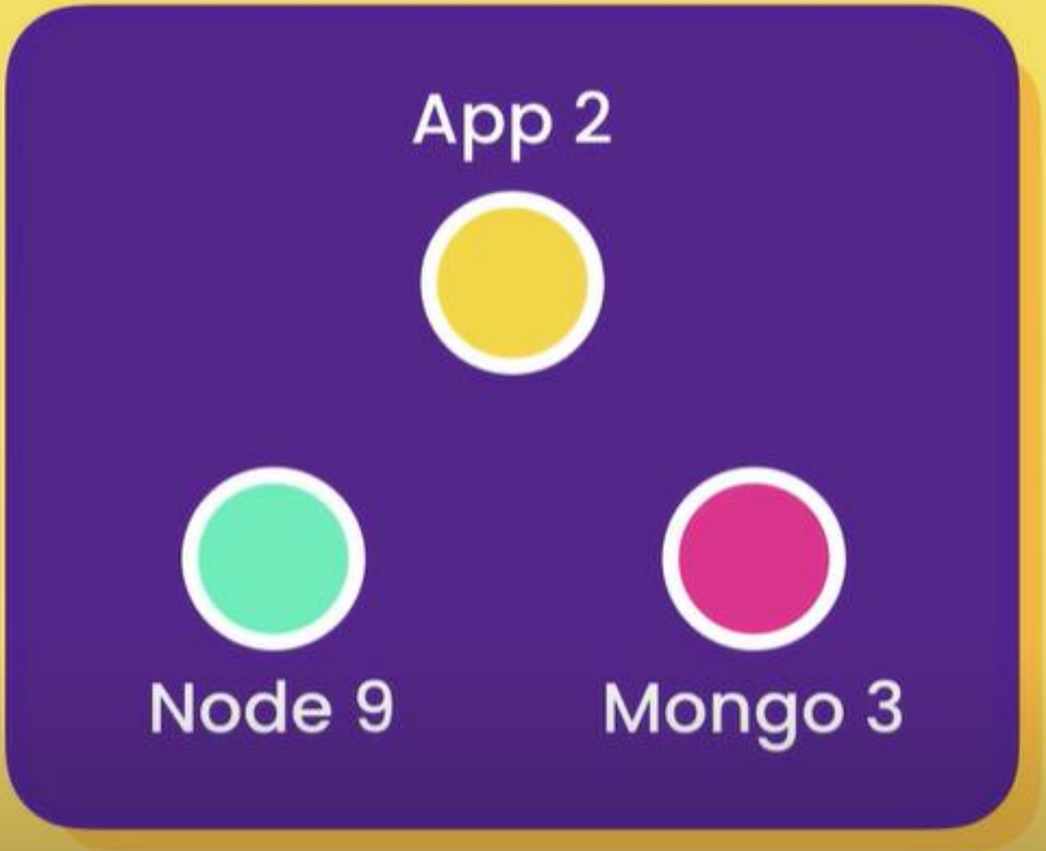


App 1

Node 14

Mongo 4

VIRTUAL MACHINE 2



App 2

Node 9

Mongo 3

Problems associated with using virtual machines:

- *Each VM needs a full copy of the OS (licensed, patched)*

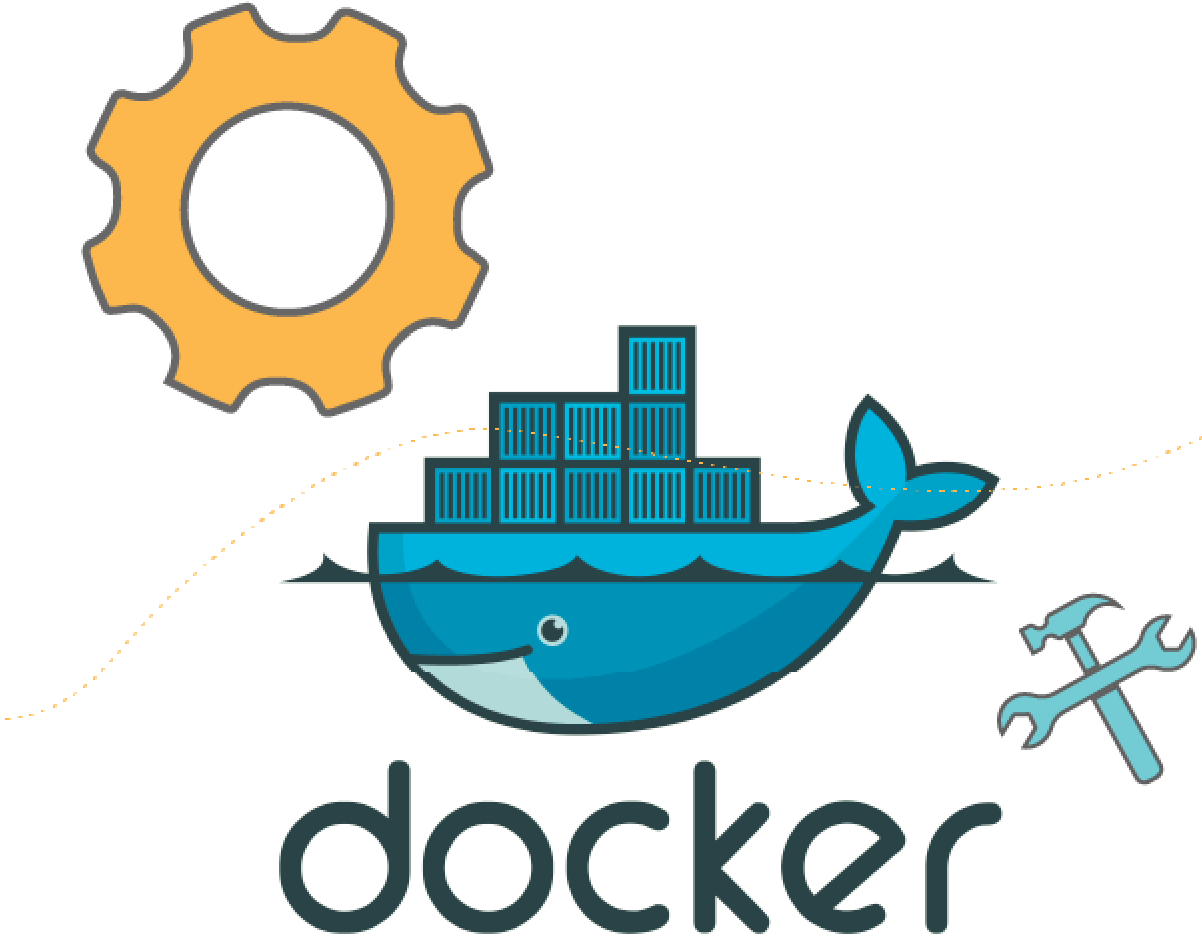
- *Slow to start: the entire OS needs to be loaded!*

- *Resource intensive: each VM takes part of actual resources (e.g., CPU, Memory, Disk)*

If you have 8GB of memory you need to decide how much of that is for VM1, VM2, and so on!
We should think how many VMs we are going to run on our machine!

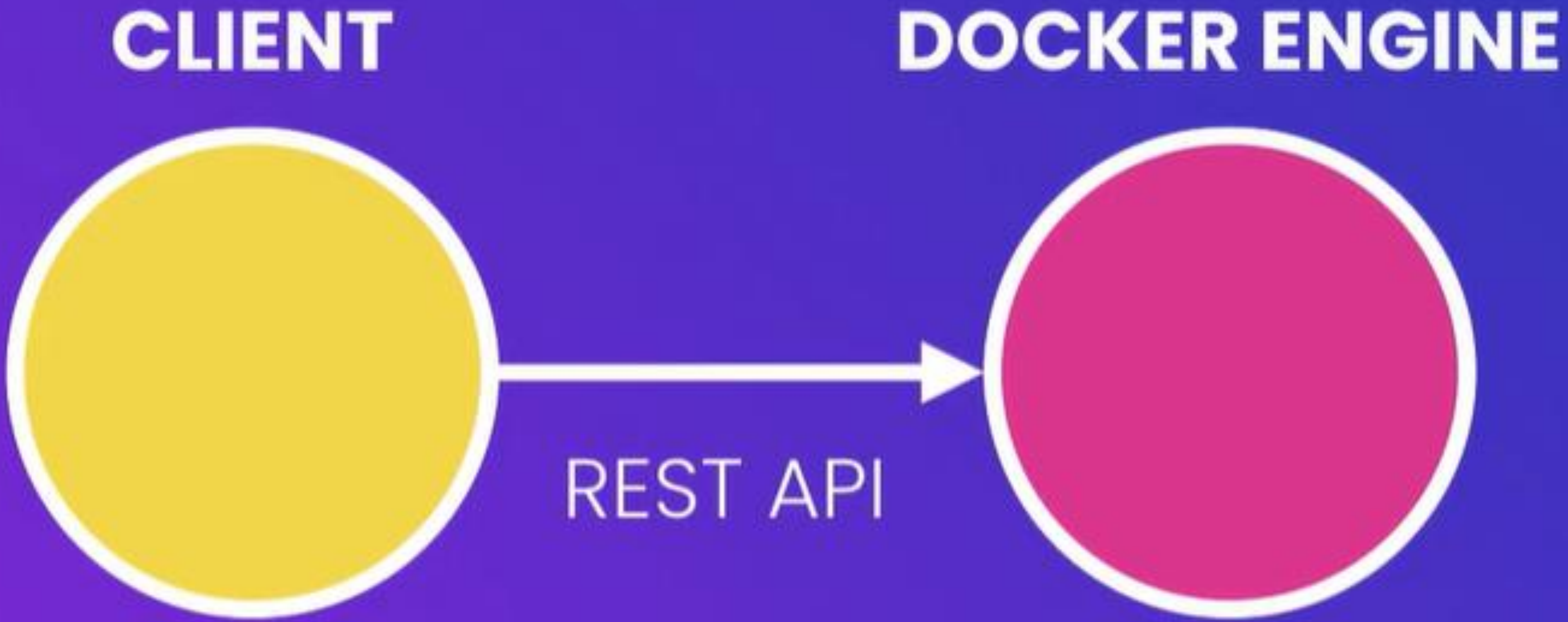
With Containers:

- *We can get (almost) the same level of isolation*
- *They are more lightweight! They don’t need a full OS!*
- *They share the OS of the host machine*
- *They start quickly: OS already started on the host!*
- *They don’t need a slice of hardware resources*



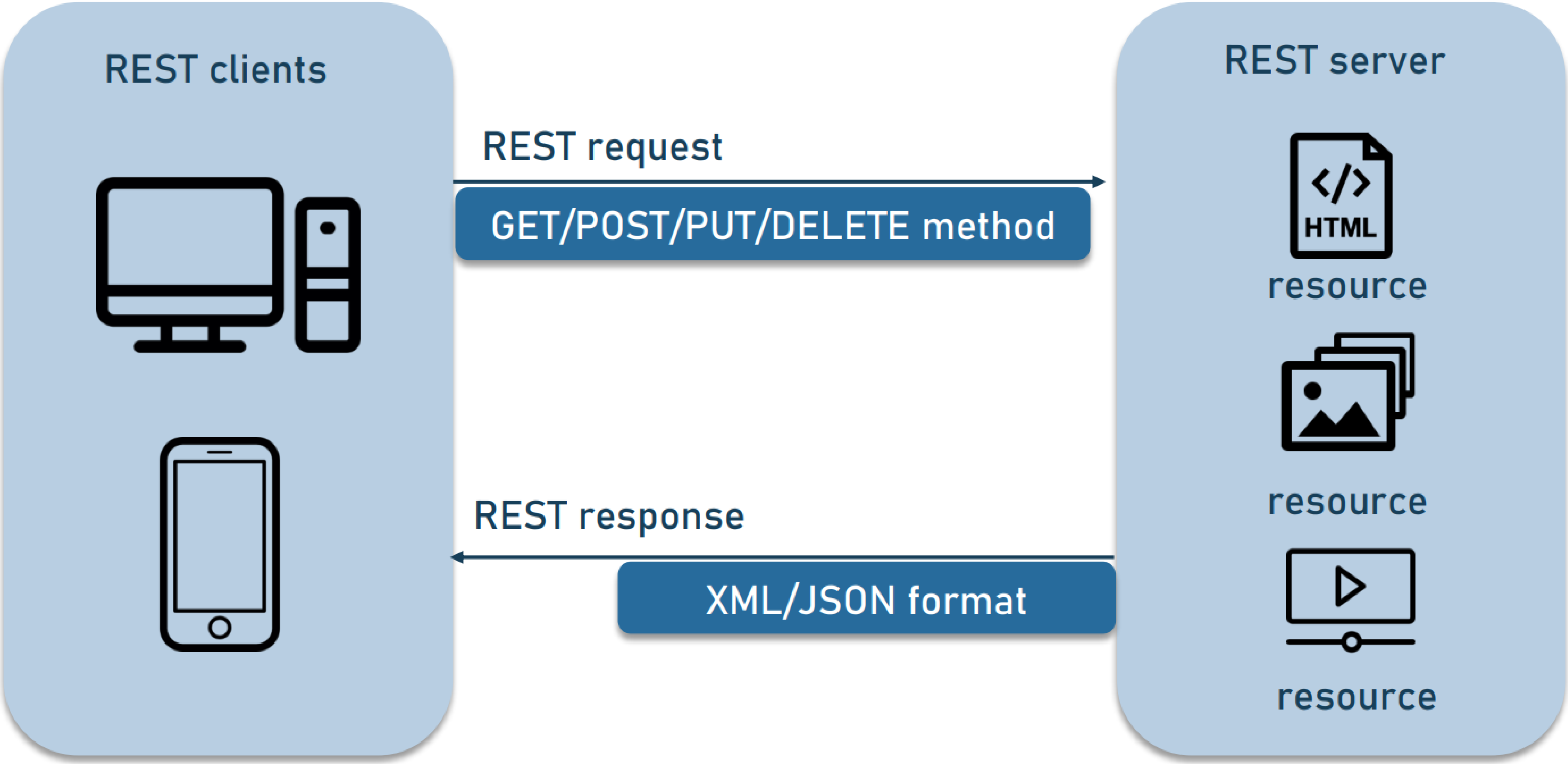
Docker Architecture

Docker uses a client-server architecture



The server takes care of building and running docker containers

REST API IN ACTION



REST clients

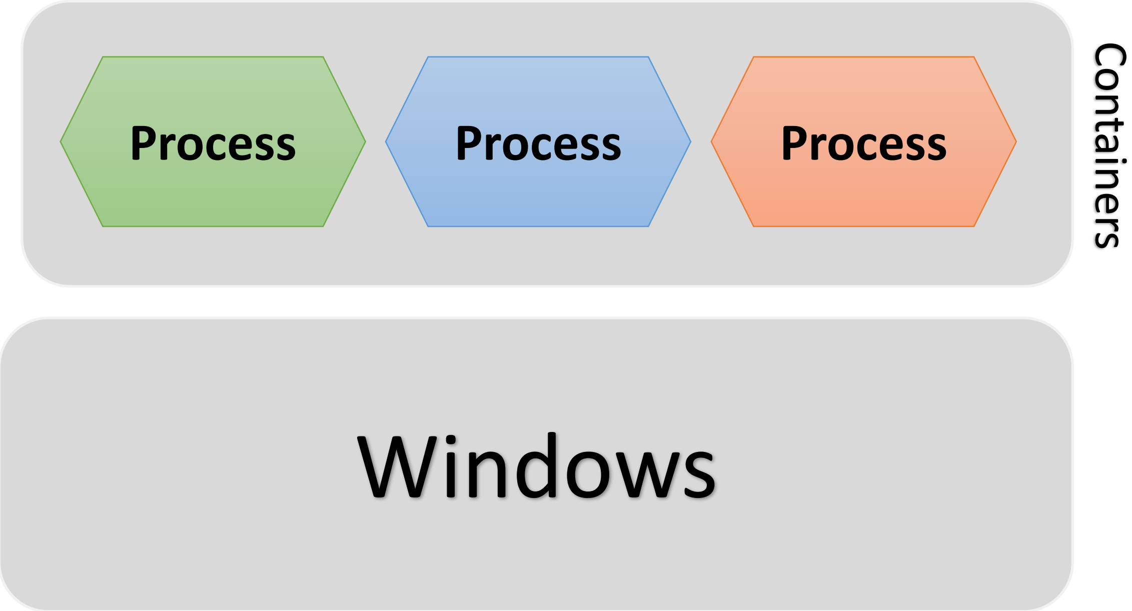
REST request

GET/POST/PUT/DELETE method

REST response

XML/JSON format

REST server



All container share the OS of the host machine! But only the kernel of the host machine!

**every OS has its own Kernel!*

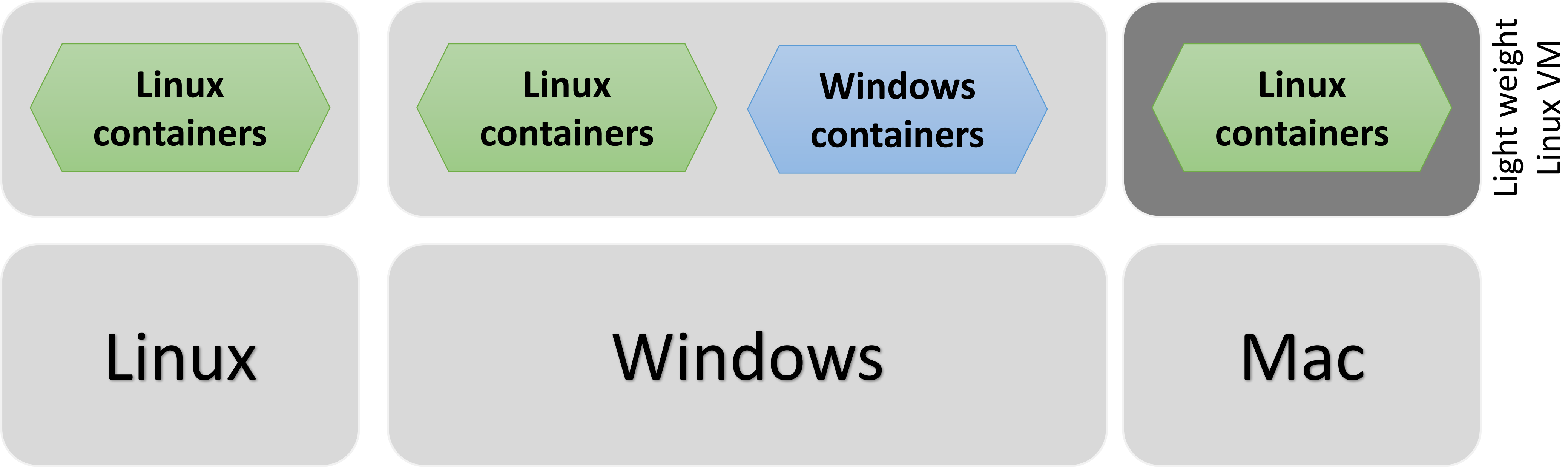
A kernel is the core of an OS, like the engine of a car! It manages all applications as well as hardware resources like memory and CPU.

Every kernel has different APIs.

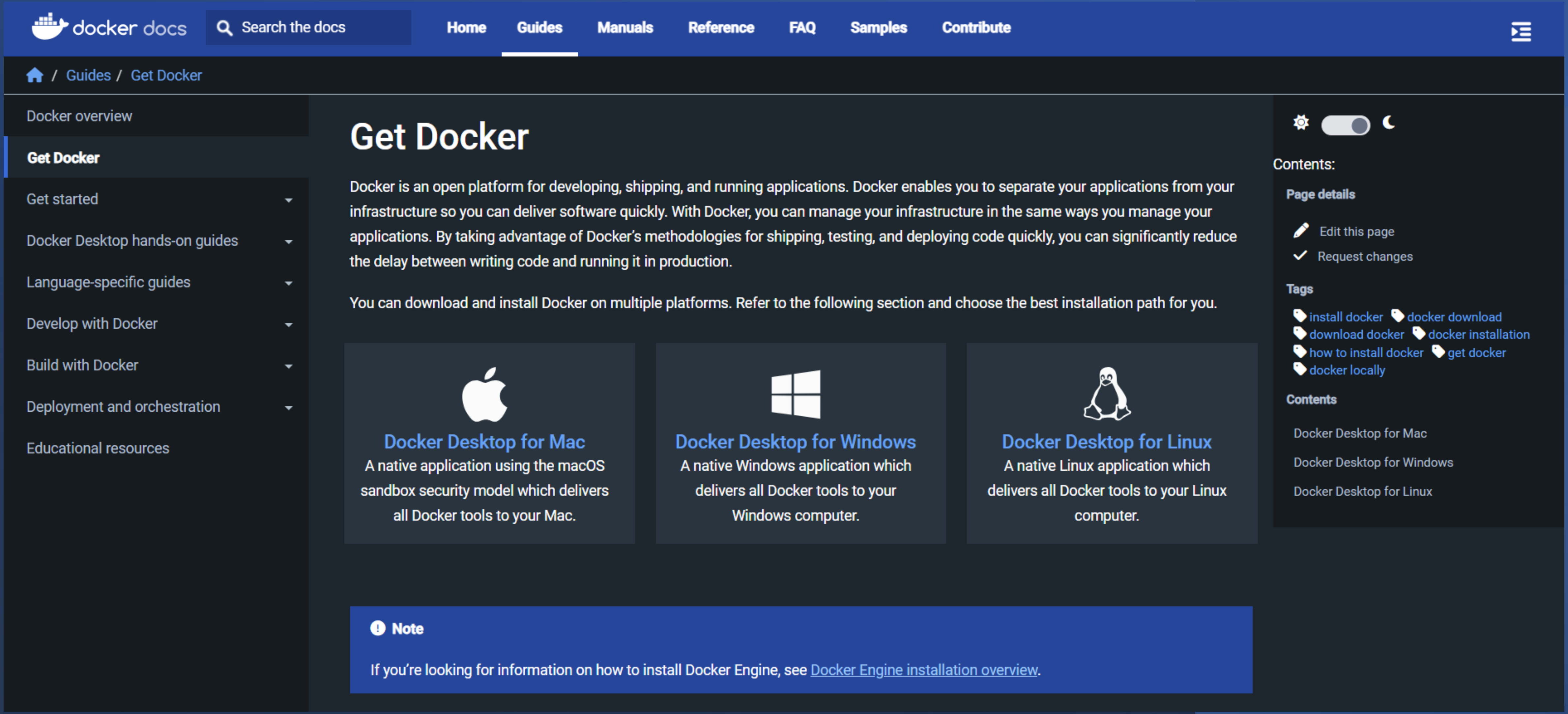
(that is why we cannot run windows app on Linux!)

On a Linux machine we can only run Linux containers!

- *Windows (from 10) shipped with custom built Linux kernel, in addition to the windows kernel!*
- *We can run Linux apps natively on Windows machines*
- *Mac does not have native support for containerized apps. Docker on Mac use light weight Linux VM to run Linux containers*



https://docs.docker.com/get-docker



Get Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

You can download and install Docker on multiple platforms. Refer to the following section and choose the best installation path for you.

Docker Desktop for Mac
A native application using the macOS sandbox security model which delivers all Docker tools to your Mac.

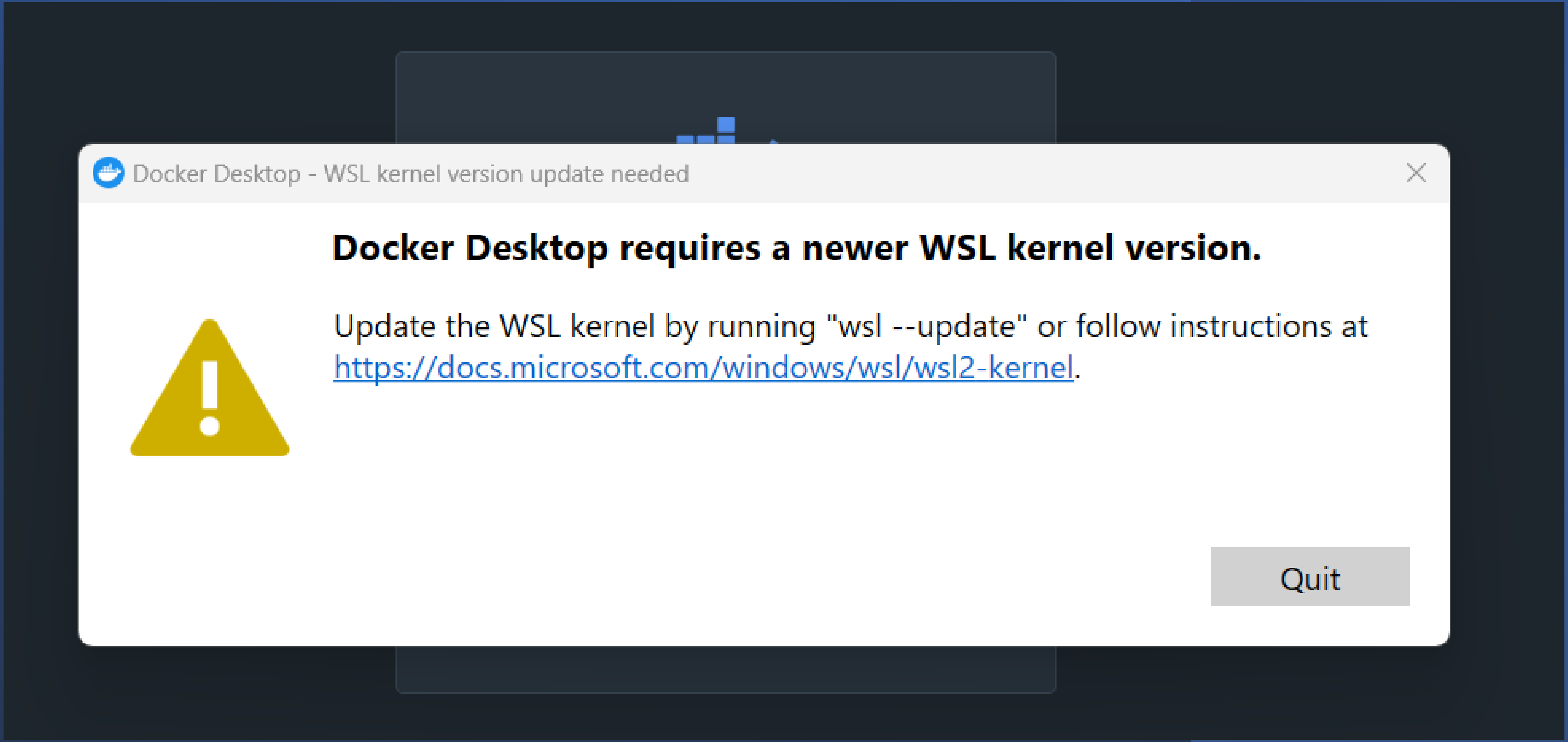
Docker Desktop for Windows
A native Windows application which delivers all Docker tools to your Windows computer.

Docker Desktop for Linux
A native Linux application which delivers all Docker tools to your Linux computer.

Note

If you're looking for information on how to install Docker Engine, see Docker Engine installation overview.

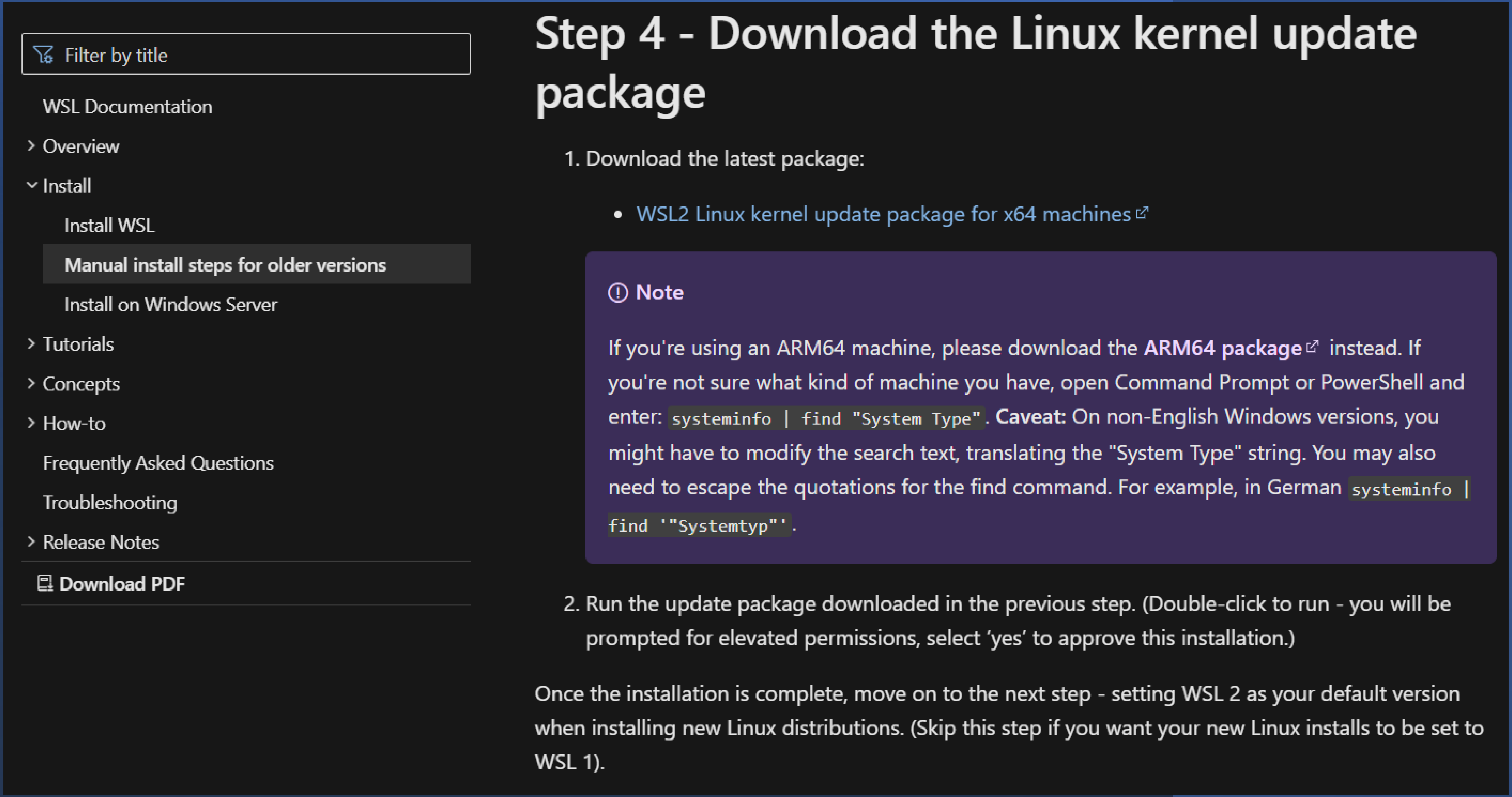
You need to upgrade the Linux kernel shipped with your Windows



Docker Desktop - WSL kernel version update needed

Docker Desktop requires a newer WSL kernel version.

Update the WSL kernel by running "wsl --update" or follow instructions at https://docs.microsoft.com/windows/wsl/wsl2-kernel.



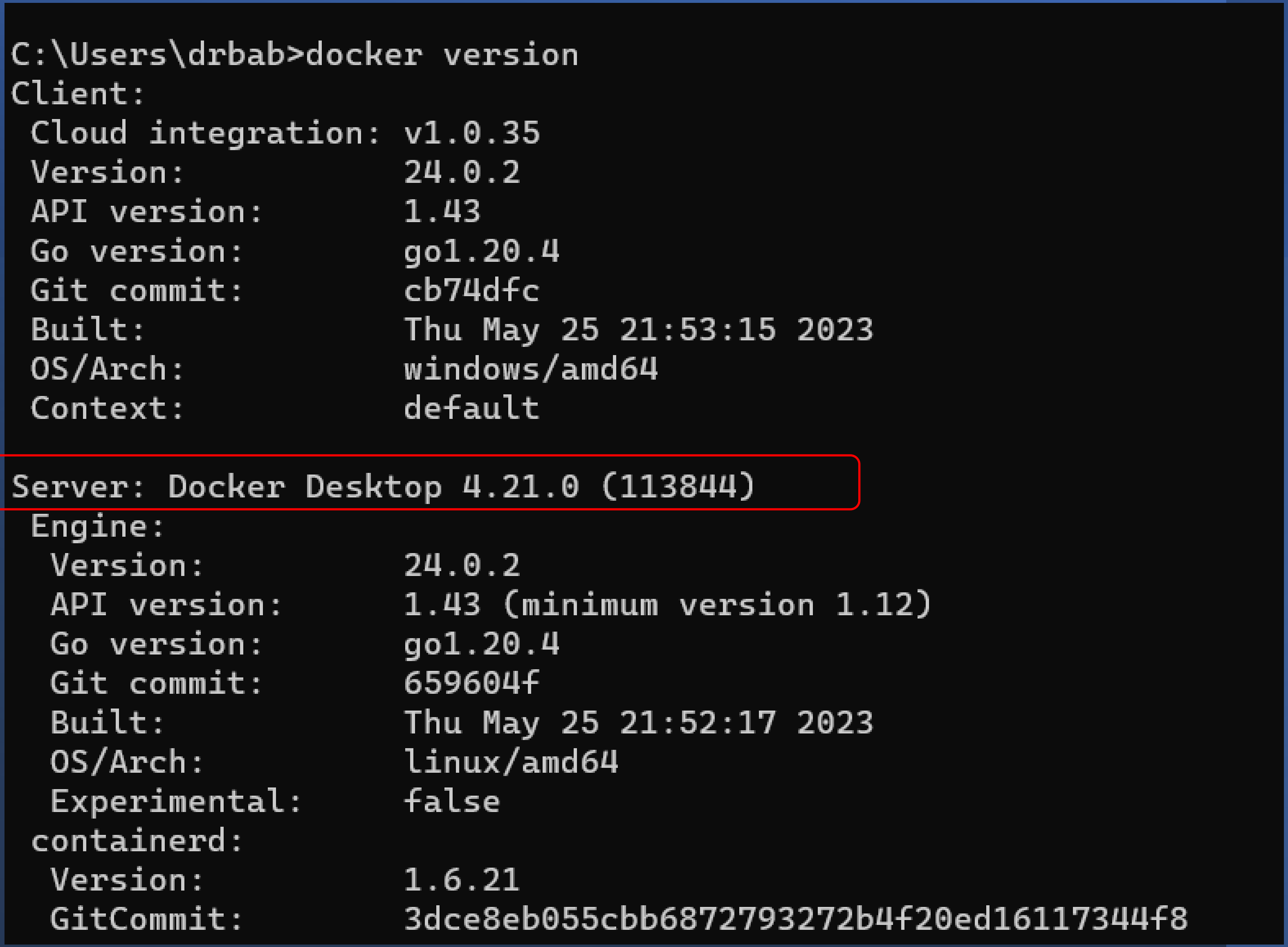
Step 4 - Download the Linux kernel update package

1. Download the latest package:
   - WSL2 Linux kernel update package for x64 machines

> **Note**
>
> If you're using an ARM64 machine, please download the **ARM64 package** instead. If you're not sure what kind of machine you have, open Command Prompt or PowerShell and enter: `systeminfo | find "System Type"`. **Caveat:** On non-English Windows versions, you might have to modify the search text, translating the "System Type" string. You may also need to escape the quotations for the find command. For example, in German `systeminfo | find '"Systemtyp"'`.

2. Run the update package downloaded in the previous step. (Double-click to run - you will be prompted for elevated permissions, select ‘yes’ to approve this installation.)

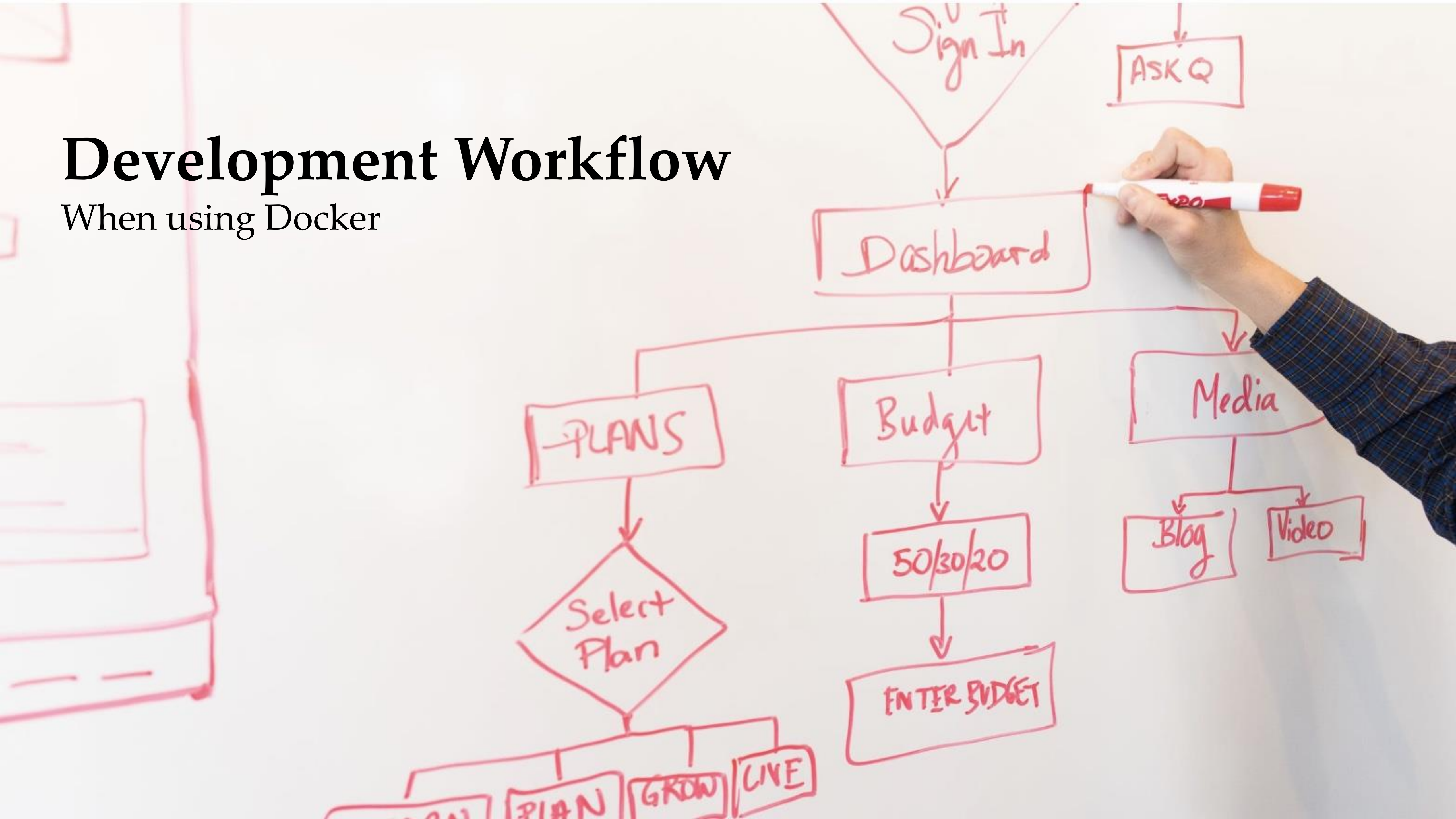
Once the installation is complete, move on to the next step - setting WSL 2 as your default version when installing new Linux distributions. (Skip this step if you want your new Linux installs to be set to WSL 1).



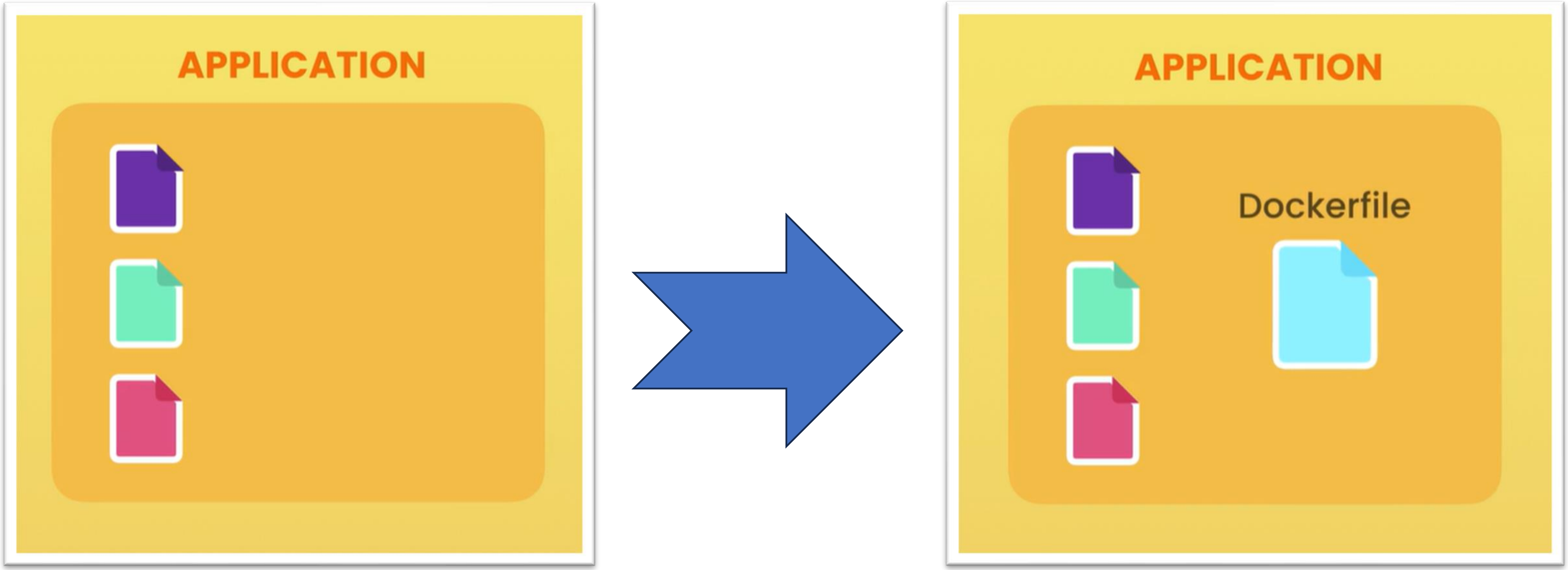
```
C:\Users\drbab>docker version
Client:
 Cloud integration: v1.0.35
 Version:           24.0.2
 API version:       1.43
 Go version:        go1.20.4
 Git commit:        cb74dfc
 Built:             Thu May 25 21:53:15 2023
 OS/Arch:           windows/amd64
 Context:           default

Server: Docker Desktop 4.21.0 (113844)
 Engine:
  Version:          24.0.2
  API version:      1.43 (minimum version 1.12)
  Go version:       go1.20.4
  Git commit:       659604f
  Built:            Thu May 25 21:52:17 2023
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.6.21
  GitCommit:        3dce8eb055cbb6872793272b4f20ed16117344f8
```

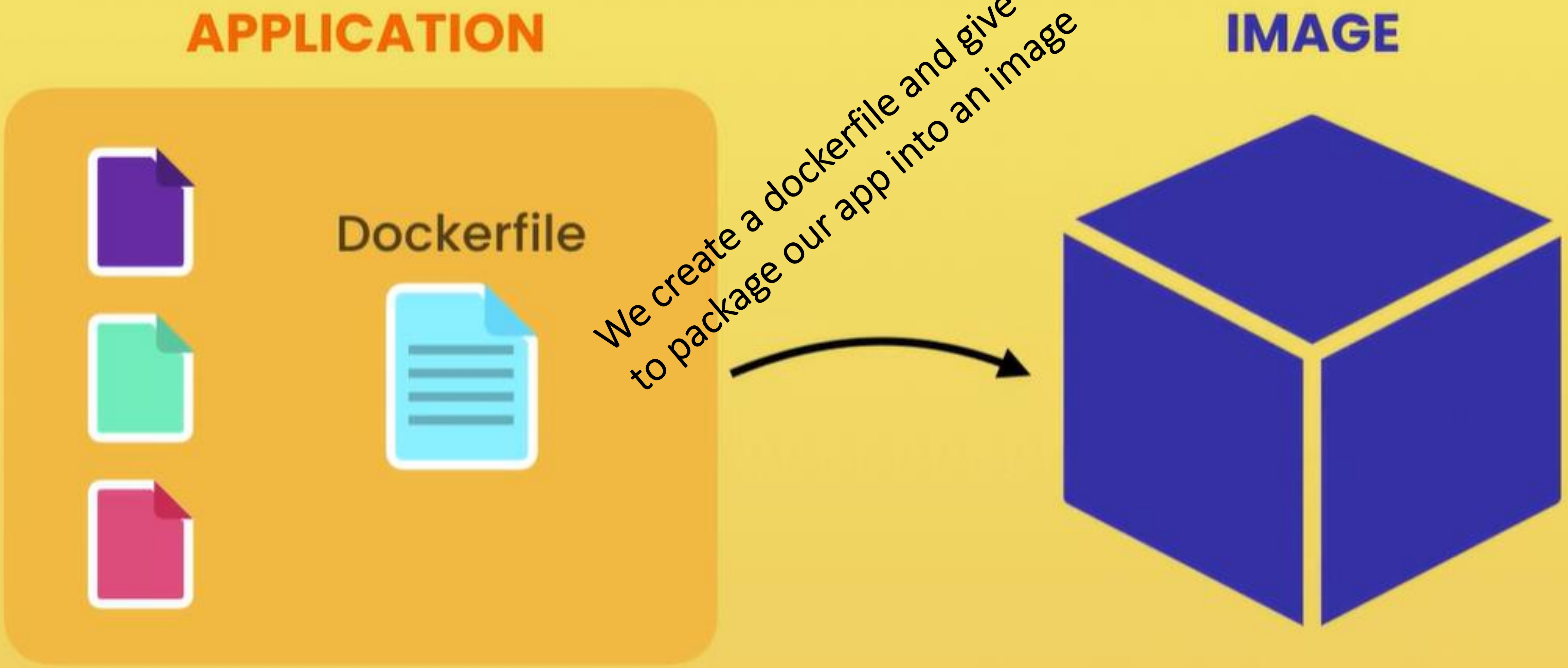
Development Workflow

When using Docker



We take an application and Dockerize it!



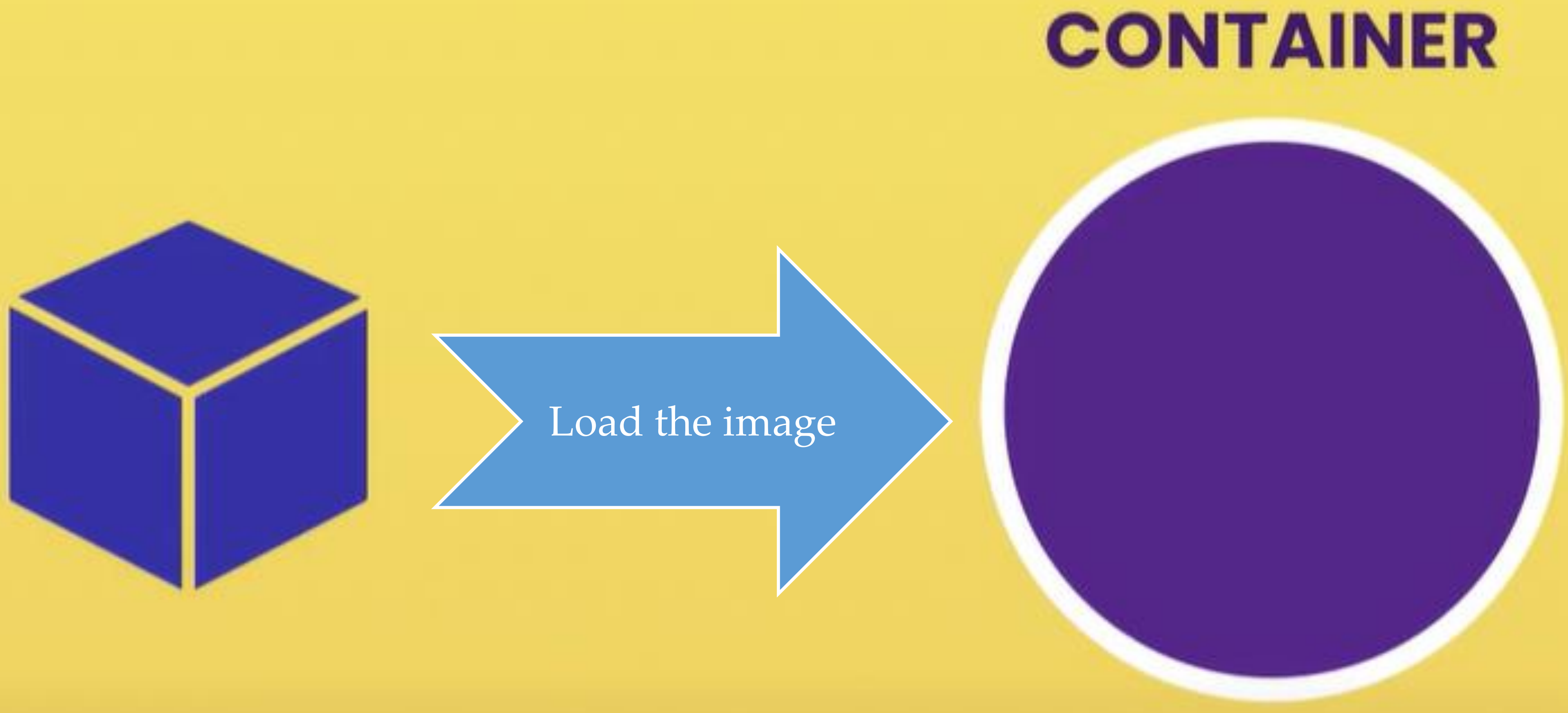


We create a dockerfile and give it to Docker to package our app into an image

An image contains:

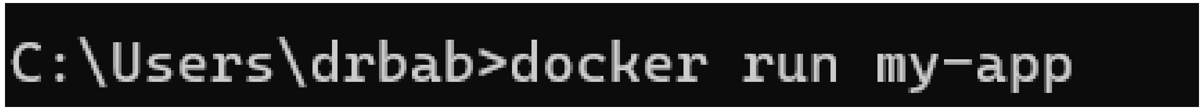
- *A cut down Operating System*
- *A runtime environment such as Node*
- *Application files*
- *Third party libraries*
- *Environment variables*

Once we have an image, we can instruct docker to start a container using that image!



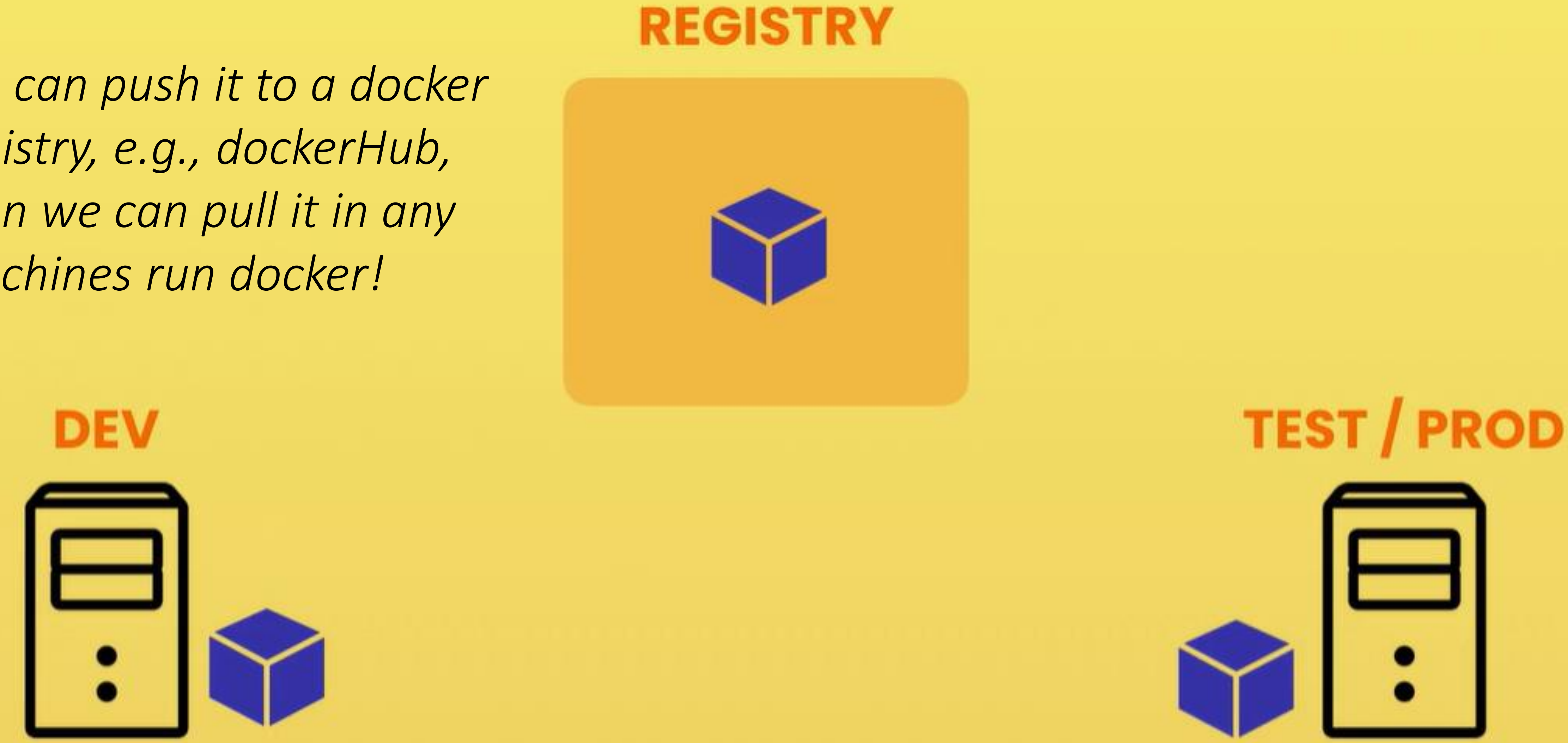
The container is just a process, but it uses the file system provide by the image

Instead of directly launching the App we tell Docker to run it inside a container.



```
C:\Users\drbab>docker run my-app
```

We can push it to a docker registry, e.g., dockerHub, then we can pull it in any machines run docker!



With docker we no longer need to maintain long complex released documents that have to be precisely followed!

All the information to build and run the application are written in the docker file!