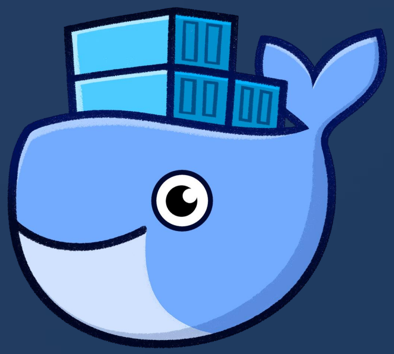
DOCKER DEEP-DIVE



Understanding how Docker leverage Linux features and its layer-based image mechanism to create the magic container world.

Lu Hoang Anh - DevOps Engineer
Application Architecture - Cloud Architecture Team

Containers

Images

Networking

Storage

Agenda

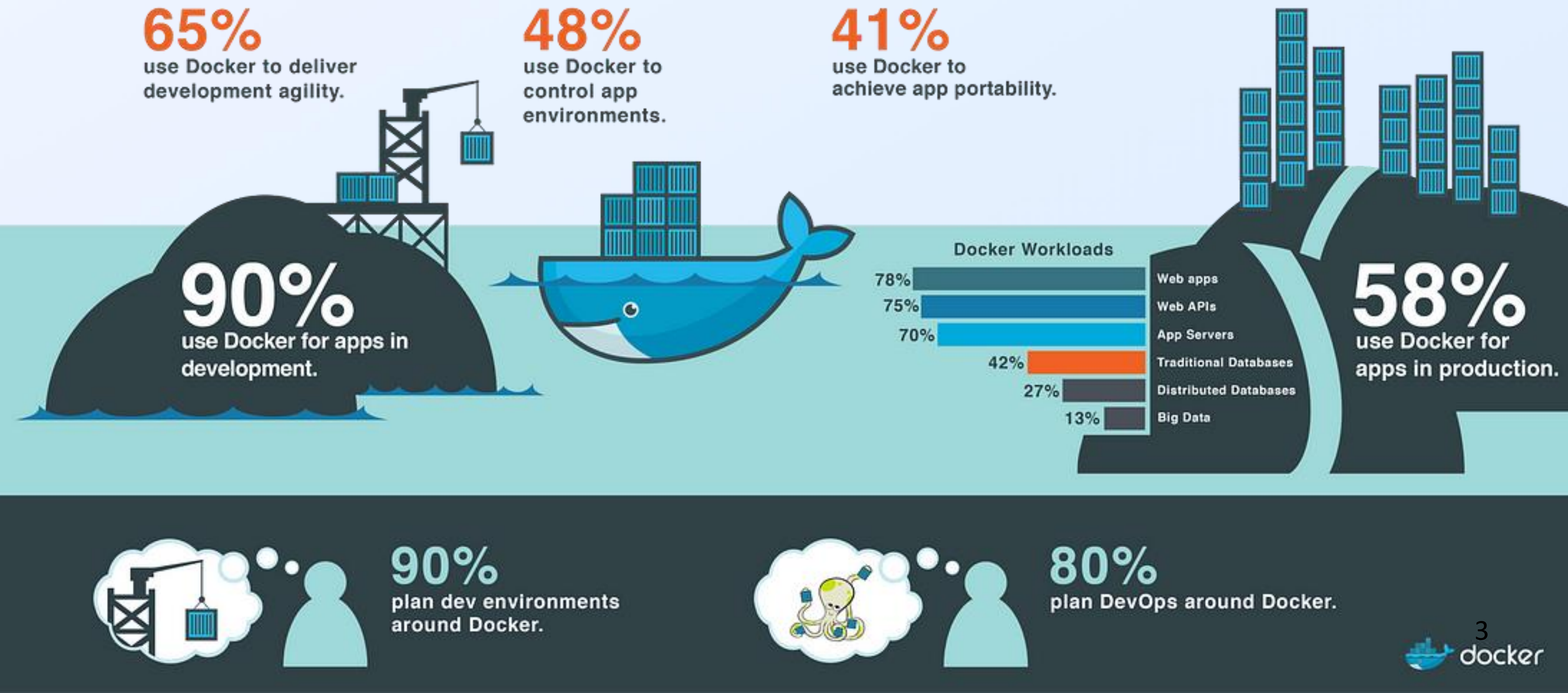
A. Docker - VM Killer

1. What is Docker?
2. Why we need Docker?
3. Why we call it “VM Killer”
4. Docker Architecture
5. Docker Terminology

B. Docker deep-dive

C. Lesson learned & Wrap-up

DOCKER MARKET OVERVIEW



65% use Docker to deliver development agility.

48% use Docker to control app environments.

41% use Docker to achieve app portability.

90% use Docker for apps in development.

58% use Docker for apps in production.

90% plan dev environments around Docker.

80% plan DevOps around Docker.

A. Docker - VM Killer

A. Docker - VM Killer

1. What is Docker?

Docker is an open platform for deploying applications in **lightweight, portable** software containers.

Each container includes:

- Application code
- Libraries & dependencies
- System tools

> Docker ensures applications run **the same way** on any machine.

A. Docker - VM Killer

1. What is Docker?

Docker is an open platform for deploying applications in **lightweight, portable** software containers.

Each container includes:

- Application code
- Libraries & dependencies
- System tools

Docker ensures applications run **the same way** on any machine.

2. Why do we need Docker?

Faster delivery

Standardized development, testing, and production environments accelerate software delivery.

Convenient packaging

Package applications and dependencies into a single unit for simplified deployment.

Cross-environment consistency

Solves "it works on my machine" problems with consistent runtime environments.

Easy and clear monitoring

Provides a unified way to read log files from all running containers. No need to remember all the specific paths

Scalability

Containerized applications can be easily scaled horizontally to handle growing loads.

A. Docker - VM Killer

3. Why we call it “VM Killer”

What is the difference between VMs and Docker Container?

A. Docker - VM Killer

3. Why we call it “VM Killer”

Virtual Machines

Complete OS

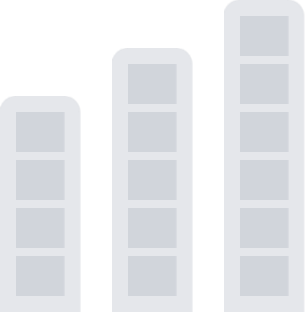
Each VM includes a complete operating system

Resource Heavy

High resource consumption due to multiple OS instances

Slow Startup

Time-consuming boot process



VS

Docker Containers

Shared OS

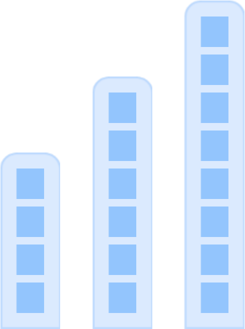
Containers share the host machine's operating system kernel

Lightweight

Lower resource consumption, more efficient use of system resources

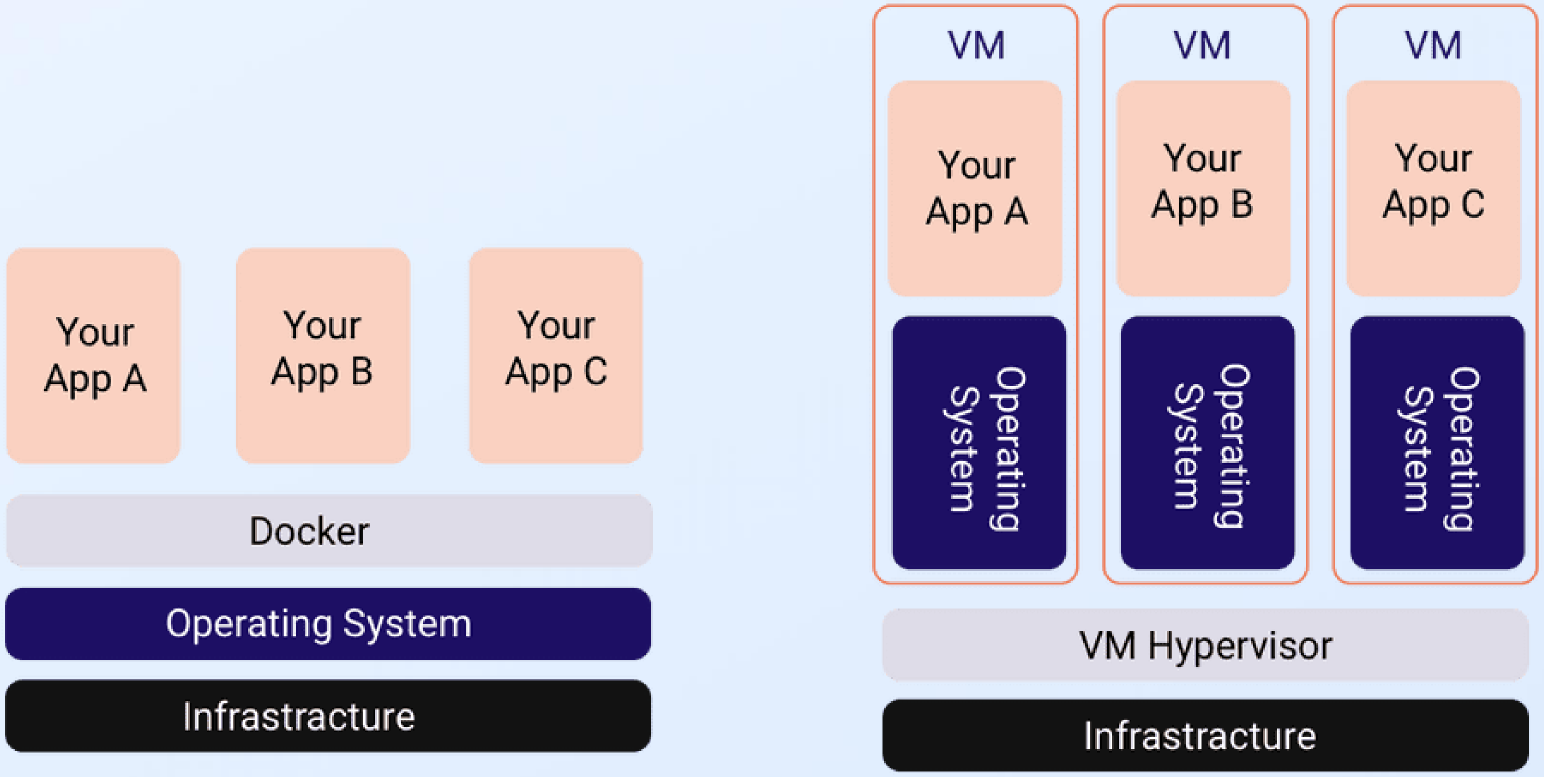
Fast Startup

Containers can be started in seconds



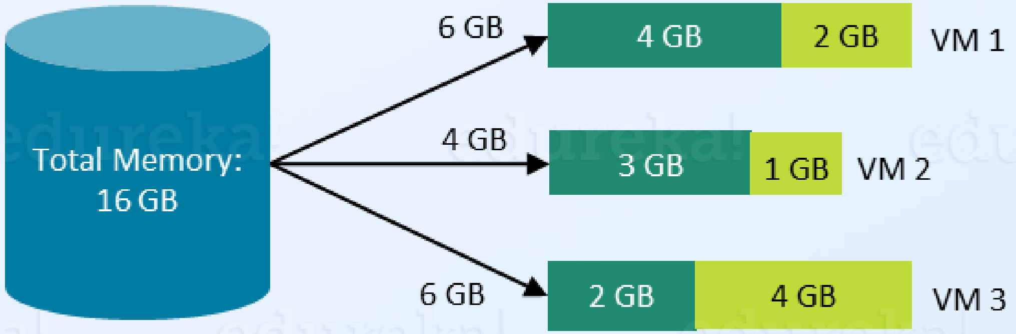
A. Docker - VM Killer

3. Why we call it “VM Killer”



A. Docker - VM Killer

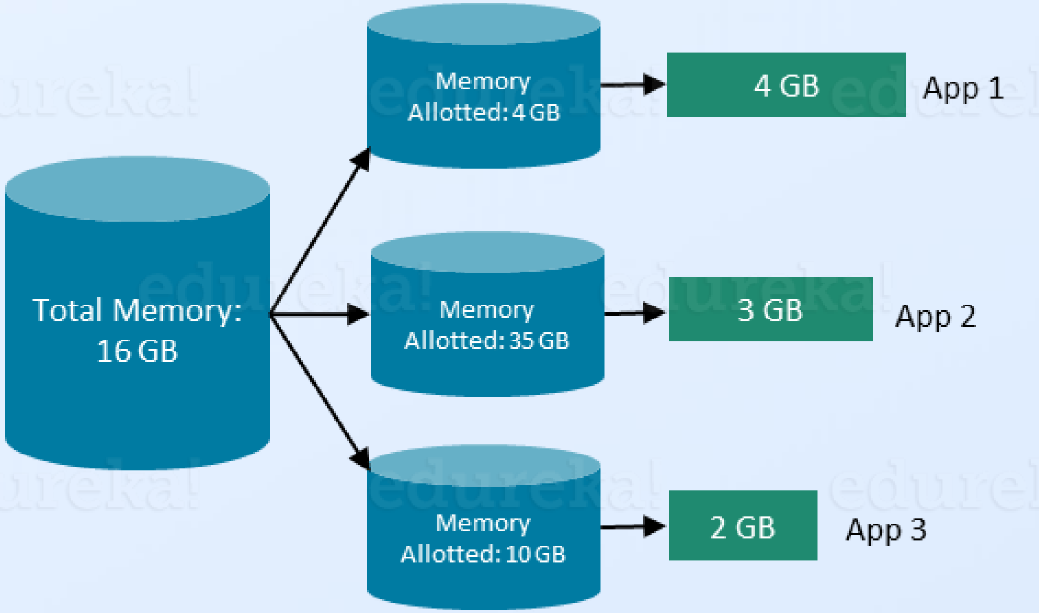
3. Why we call it “VM Killer”



🟩 → Memory Used: 9 GB

🟨 → Memory wasted: 7 GB

7 Gb of Memory is blocked and
cannot be allotted to a new VM



🟩 → Memory Used: 9 GB

Only 9 GB memory utilized;
7 GB can be allotted to a new Container

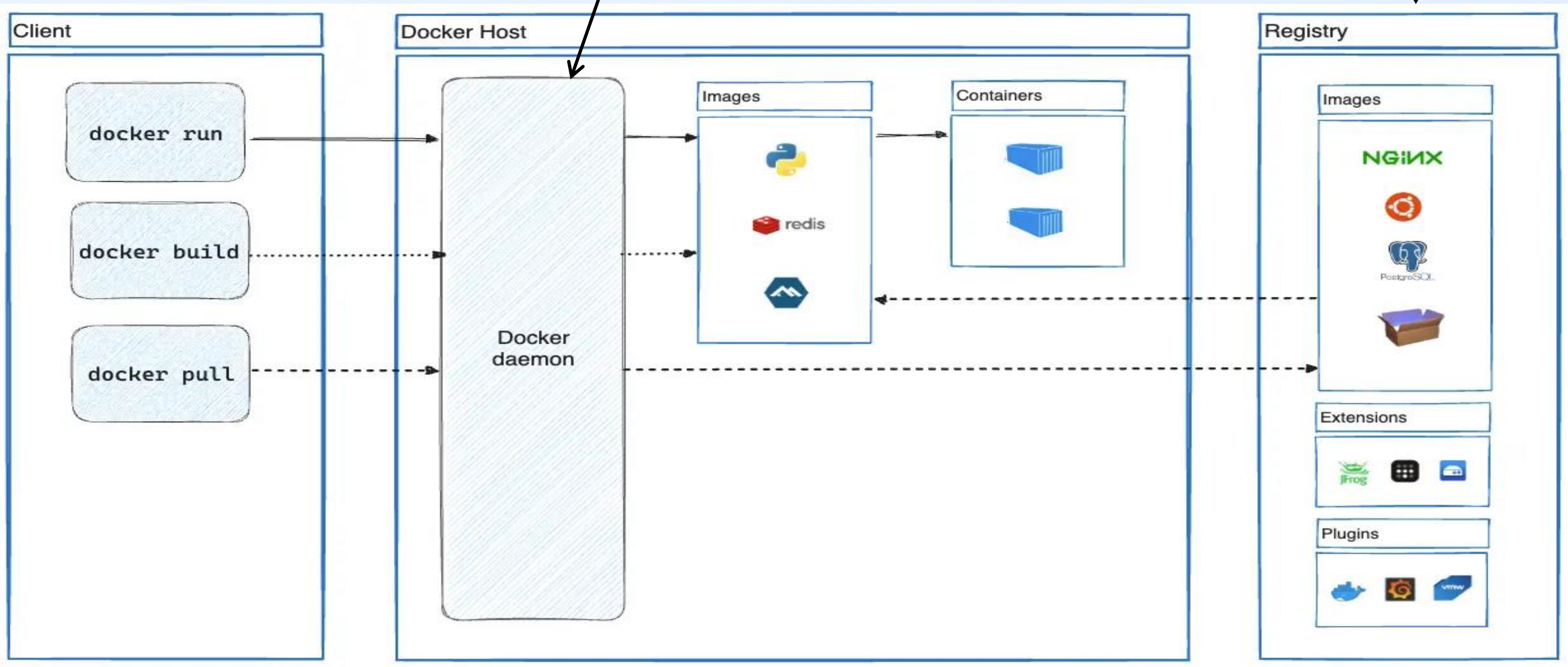
A. Docker - VM Killer

4. Docker Architecture

Docker API Server + Manage Docker Object

A Docker registry stores Docker images

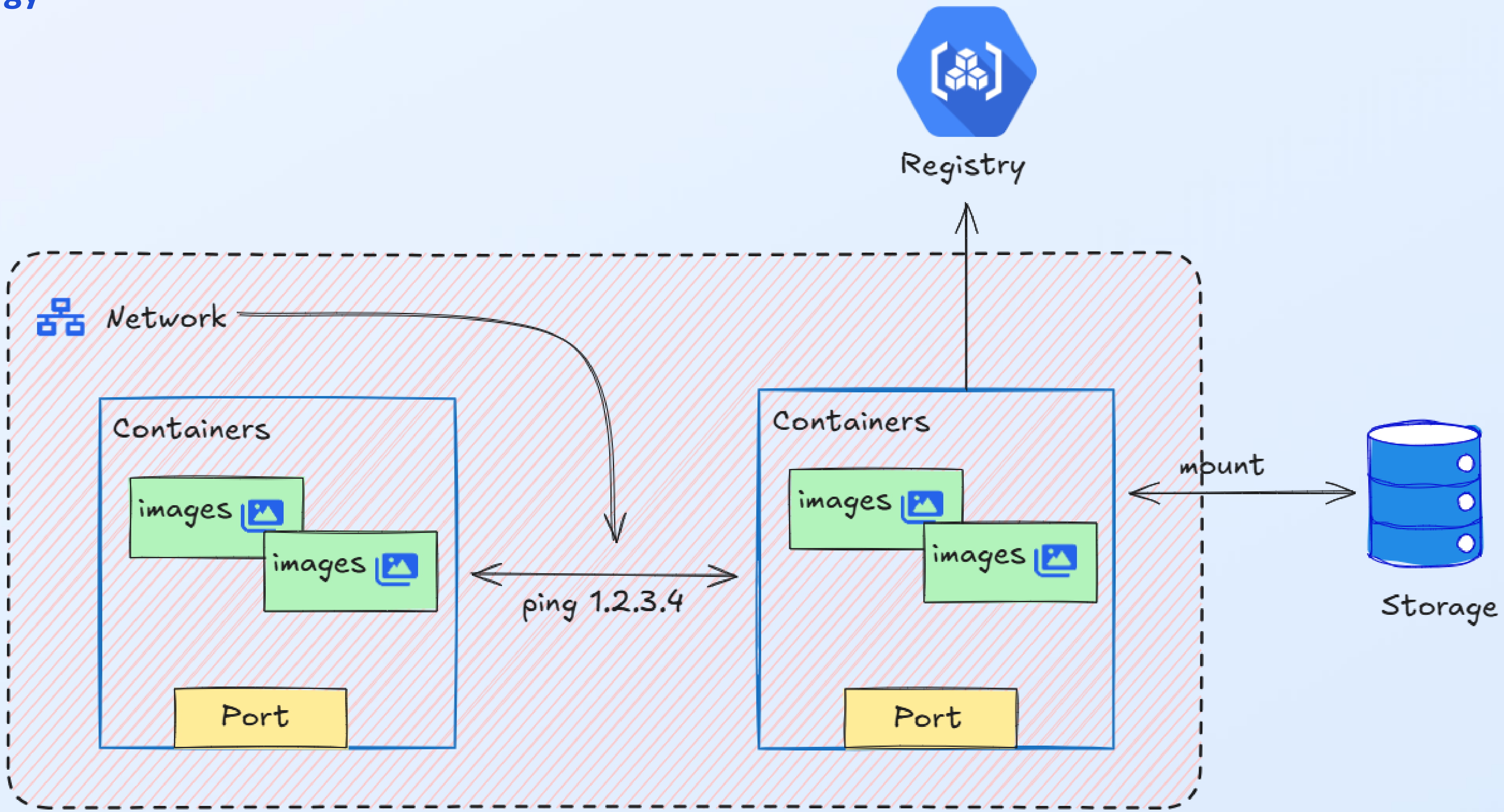
CLI to interact with Docker



Docker architecture - Source: https://docs.docker.com/get-started/docker-overview/

A. Docker - VM Killer

5. Docker Terminology



Docker Concept

A. Docker - VM Killer

5. Docker Terminology

Container

A running instance of an application, including the application and all its dependencies.

Image

A read-only template used to create containers, containing application code, runtime, libraries, and configuration.

Port

Used to expose container-internal services to the external network, enabling communication.

Volume

Used to persist container data so it can be preserved beyond the container's lifecycle.

Network

Allows communication between containers and with the external world.

Registry

A service for storing and distributing Docker images, such as Docker Hub.

B. Docker Deep-dive

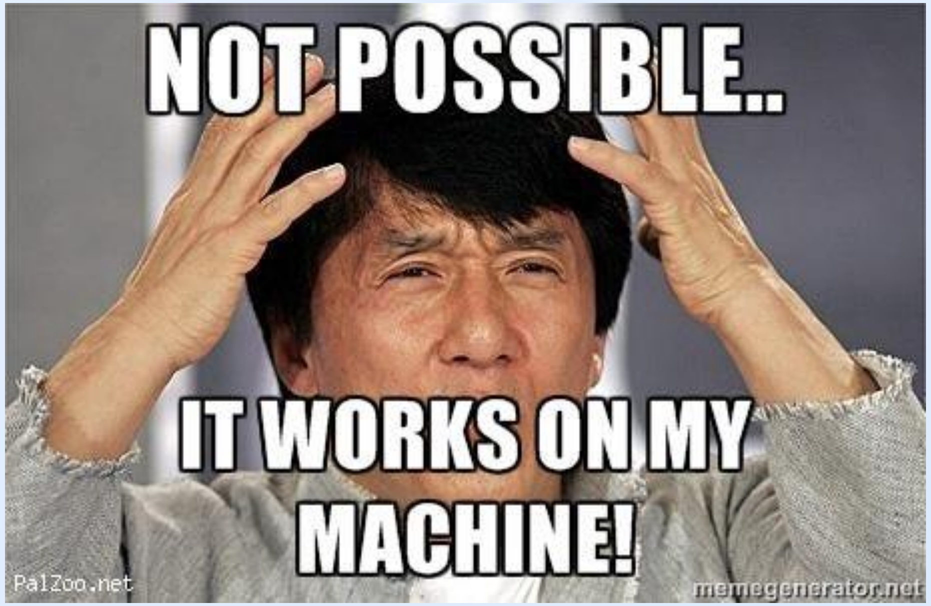
B. Storytelling: Docker Concepts in-action

The Project Story

A simple **todo-app** which will has 2 components: logic and database

What are the possible problems when we start developing this app?

- Developers are using different environments (Windows, Mac, Linux)
- Manually set up the environment is time-consuming and error-prone
- Famous problem: "It works on my machine"



B. Storytelling: Docker Concepts in-action

1. Understand Docker Images & Containers

Docker Containers

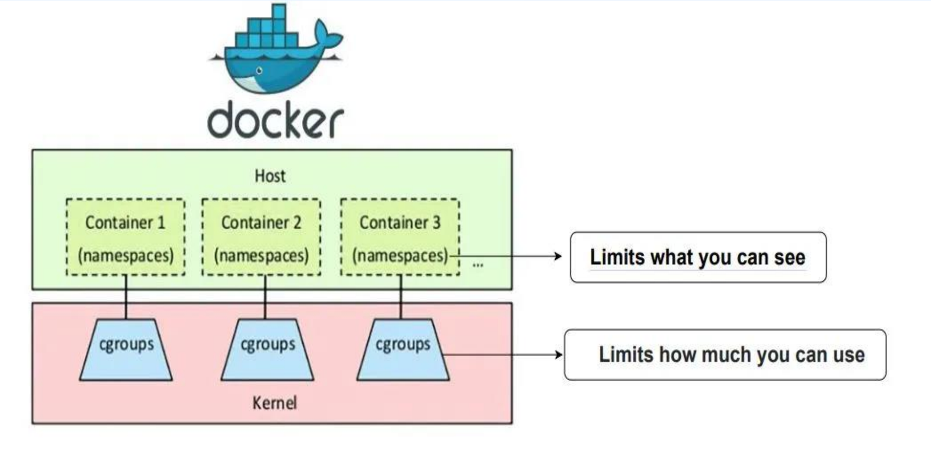
The smallest executable software unit, packaging all needed parts to run an application.

Key characteristics:

- Runs as a process on the host machine
- Uses Linux kernel features for isolation
- Ephemeral (no persistent state)

Container Isolation

cgroups	chroot	namespaces
Limit resource usage	Changes root directory	Isolates users/processes



Source: https://blog.devops.dev/linux-containers-deep-dive-c0668a4f347d

B. Storytelling: Docker Concepts in-action

1. Understand Docker Images & Containers

Docker Images

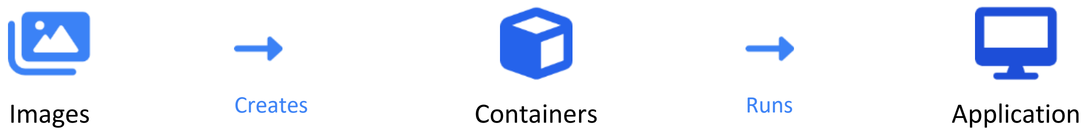
A package containing all files needed to create containers.

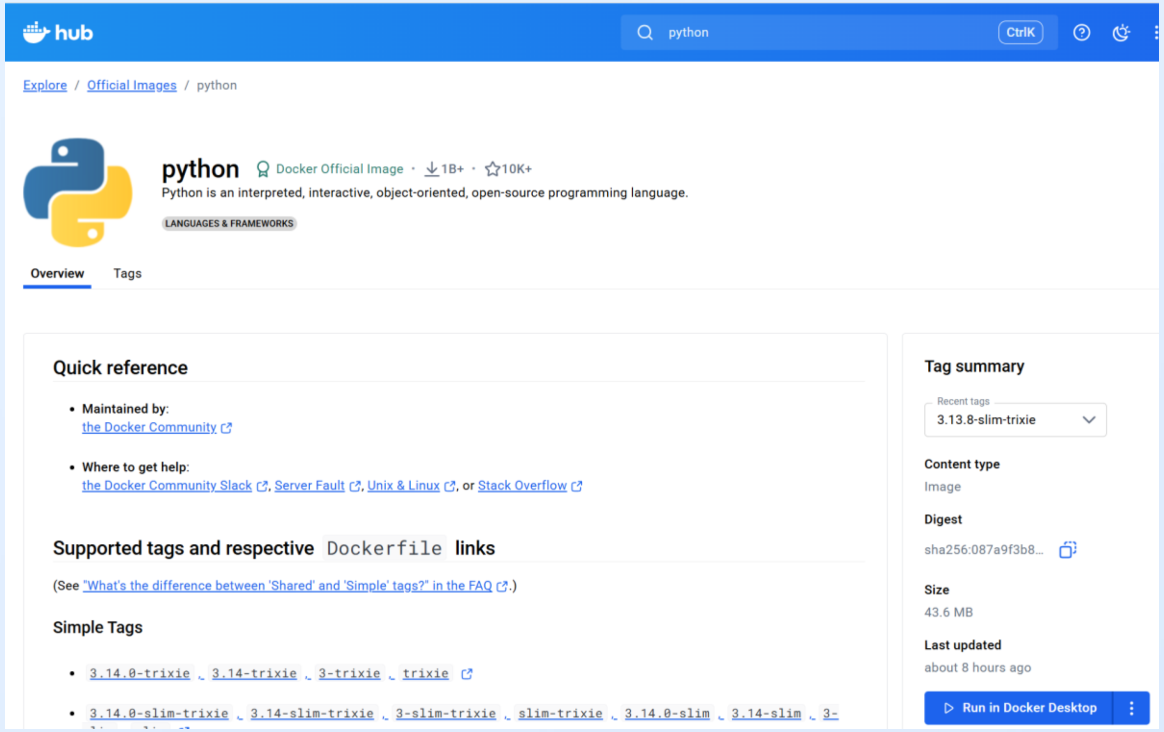
Copy-on-write modal: When create an image, every step is cached and can be reused in future builds.

Two important principles:

- Images are Immutable - changes require new image
- Images are composed of layers

Images & Containers Relationship





Explore / Official Images / python

python Docker Official Image · 1B+ · 10K+

Python is an interpreted, interactive, object-oriented, open-source programming language.

LANGUAGES & FRAMEWORKS

Overview Tags

Quick reference

- **Maintained by:**
  the Docker Community
- **Where to get help:**
  the Docker Community Slack, Server Fault, Unix & Linux, or Stack Overflow

Supported tags and respective `Dockerfile` links

(See "What's the difference between 'Shared' and 'Simple' tags?" in the FAQ.)

Simple Tags

- `3.14.0-trixie`, `3.14-trixie`, `3-trixie`, `trixie`
- `3.14.0-slim-trixie`, `3.14-slim-trixie`, `3-slim-trixie`, `slim-trixie`, `3.14.0-slim`, `3.14-slim`, `3-`

Tag summary

Recent tags
3.13.8-slim-trixie

Content type
Image

Digest
sha256:087a9f3b8...

Size
43.6 MB

Last updated
about 8 hours ago

Run in Docker Desktop

Source: https://hub.docker.com/_/python

B. Storytelling: Docker Concepts in-action

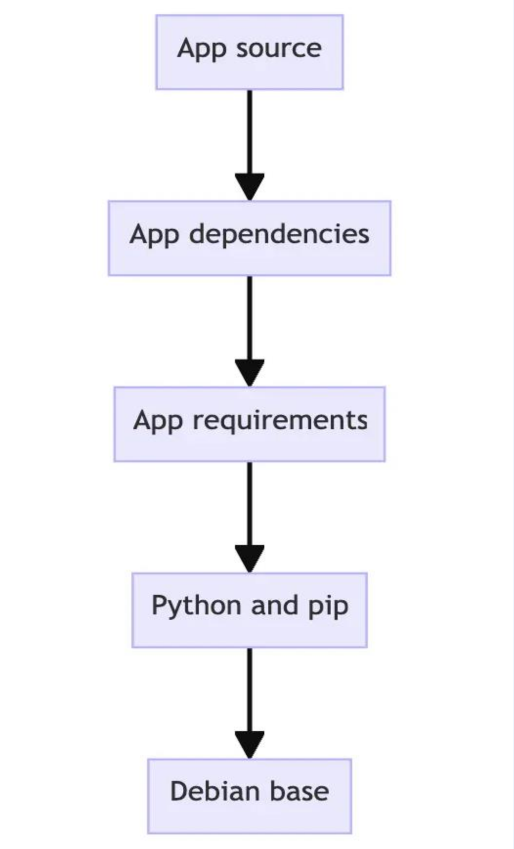
1. Understand Docker Images & Containers

Image Layers

- A layer is essentially **a snapshot / diff** of the filesystem captured at one point
- Layers are content-addressable
- Layers are stacked on top of each other to form the final image.
- Layers are cached and reused to optimize build times and reduce storage usage.

Example: a theoretical image

1. The first layer adds basic commands and a package manager, such as apt.
2. The second layer installs a Python runtime and pip for dependency management.
3. The third layer copies in an application's specific requirements.txt file.
4. The fourth layer installs that application's specific dependencies.
5. The fifth layer copies in the actual source code of the application.



Source: https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/

B. Storytelling: Docker Concepts in-action

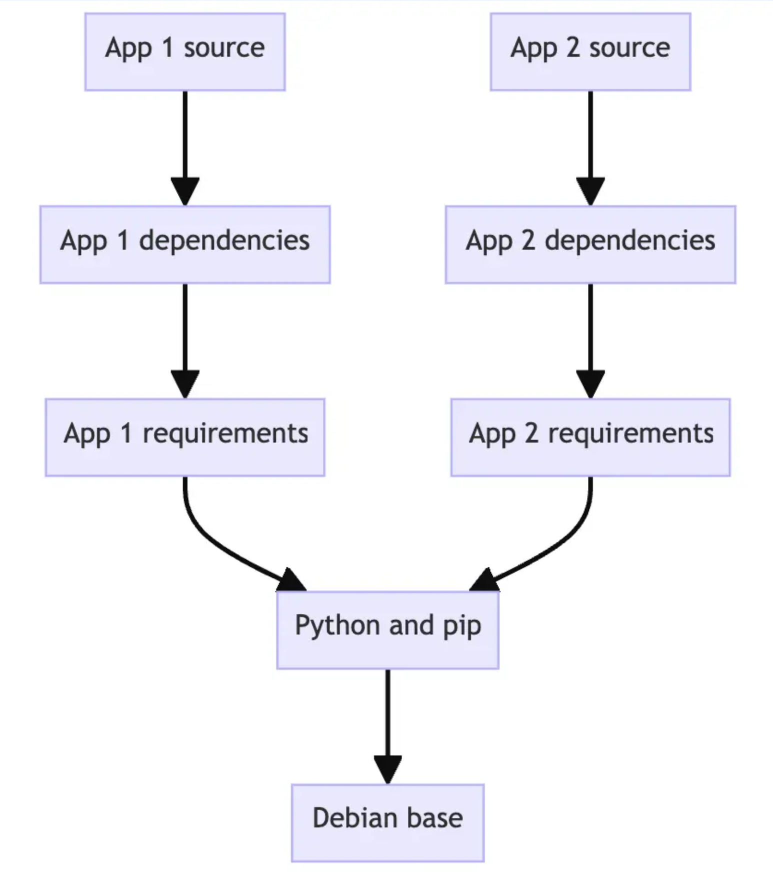
1. Understand Docker Images & Containers

Image Layers

- A layer is essentially **a snapshot / diff** of the filesystem captured at one point
- Layers are content-addressable
- Layers are stacked on top of each other to form the final image.
- Layers are cached and reused to optimize build times and reduce storage usage.

Example: create another Python application

1. Leverage the same Python base
2. Make builds faster
3. Reduce the amount of storage and bandwidth



Source: https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/

B. Storytelling: Docker Concepts in-action

1. Understand Docker Images & Containers

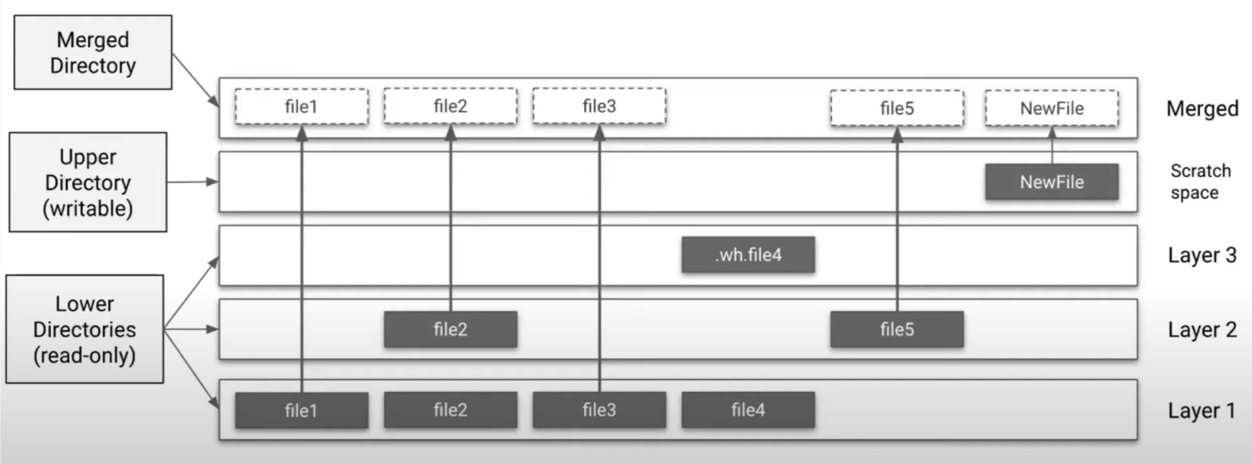
Image Layers

How layers are stacked to form an image

1. Image = layer digests + image manifest (metadata)
2. When run a container, an extra writable layer is added on the top → changes happened inside container are isolated in this writable layer
3. At runtime, Docker mounts these read-only layers using **a union filesystem** → single merged view

What is the benefit?

Union filesystem terminology



Source: https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/

B. Storytelling: Docker Concepts in-action

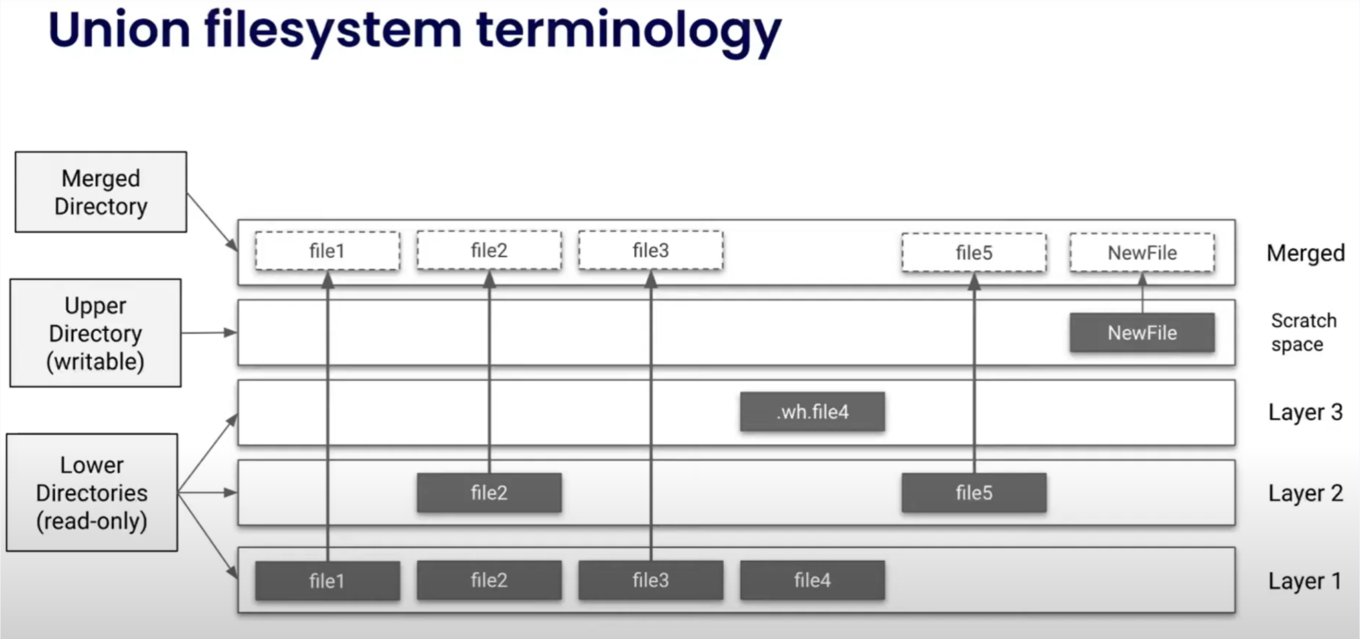
1. Understand Docker Images & Containers

Image Layers

How layers are stacked to form an image

1. Image = layer digests + image manifest (metadata)
2. When run a container, an extra writable layer is added on the top → changes happened inside container are isolated in this writable layer
3. At runtime, Docker mounts these read-only layers using **a union filesystem** → single merged view

What is the benefit?



Source: https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/

"How do we create images for our app? And more importantly, how do we define what goes into the image?"

B. Storytelling: Docker Concepts in-action

2. Design Dockerfile

What is Dockerfile?

- A Dockerfile is a text-based document that's used to create a container image
- Provide instructions to the image builder, e.g. run cmd, copy file, set env variable,...

Common Instructions

FROM	Base image
WORKDIR	Set working directory
COPY	Copy files from host to image
ADD	Add local or remote files and directories
RUN	Execute commands in image
CMD	Default command to run
ENTRYPOINT	Specify default executable

Dockerfile Example

```
FROM python:3.13                  # Base image
WORKDIR  /usr/local/app           # Set working directory

# Install dependencies
COPY  requirements.txt ./         # Copy files from host to image
RUN pip install --no-cache-dir -r requirements.txt    # Run command inside image

# Copy in the source code
COPY src ./src
EXPOSE 8000                       # Document the port the app listens on

# Copy in the source code
RUN useradd app_user              # Set default user for all subsequent instructions
USER app

CMD uvicorn app.main:app \        # Default command to run when container starts
    --host 0.0.0.0 \
    --port 8080
```

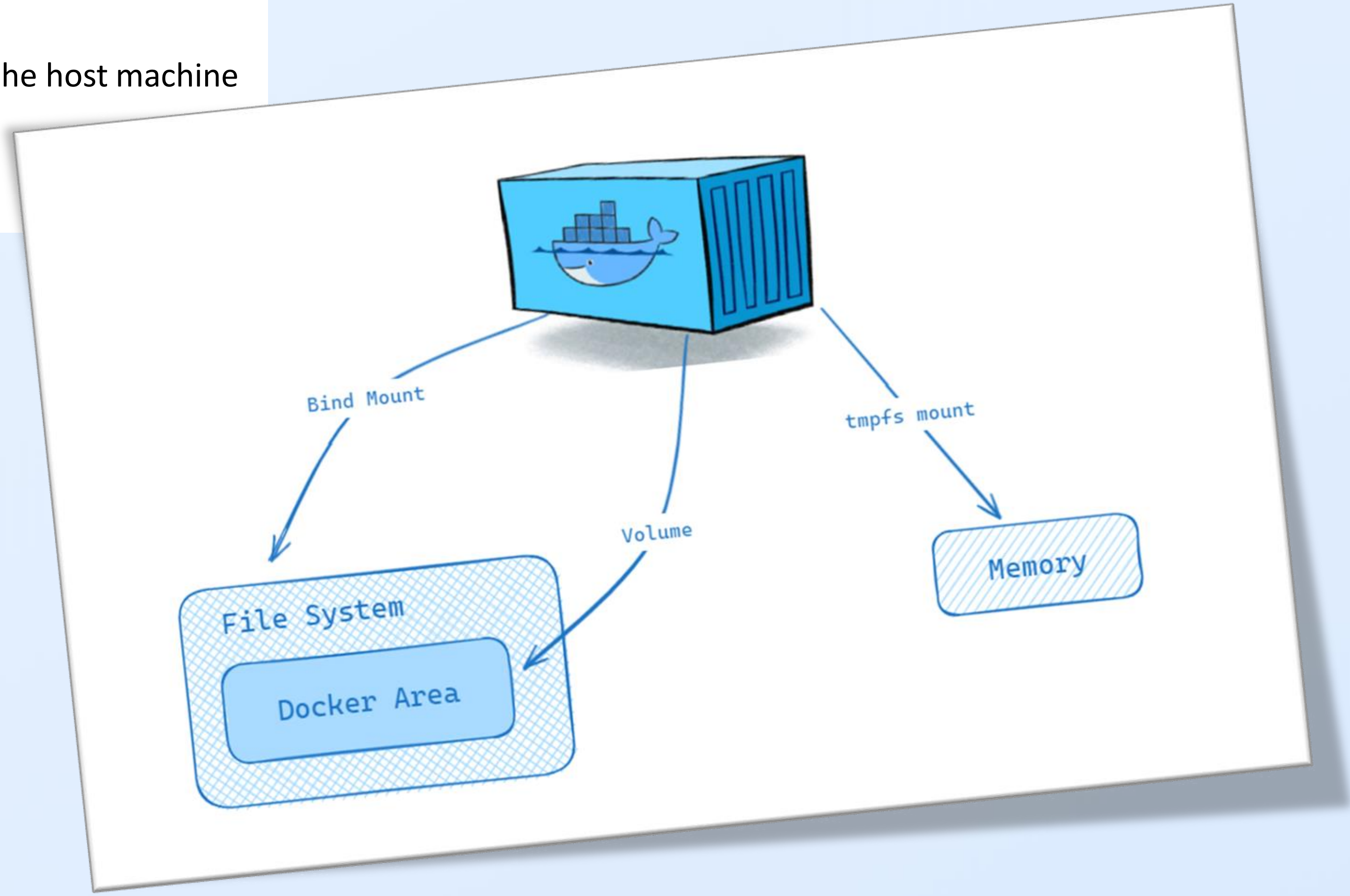
B. Storytelling: Docker Concepts in-action

3. Docker Storage

What is Docker Volume?

Docker Volume provide the ability to:

- Connect specific filesystem paths of the container back to the host machine
- Allow data to persist beyond the lifecycle of a container.



Source: https://therahulsarkar.medium.com/understanding-docker-volumes-a-comprehensive-guide-46339aa9ac53

B. Storytelling: Docker Concepts in-action

2. Docker Storage

Volume Mounts

Managed by Docker, persistent data storage.

- ✓ Managed by Docker
- ✓ Stored in Docker's storage area

```
>_ docker volume create mydata
```

Bind Mounts

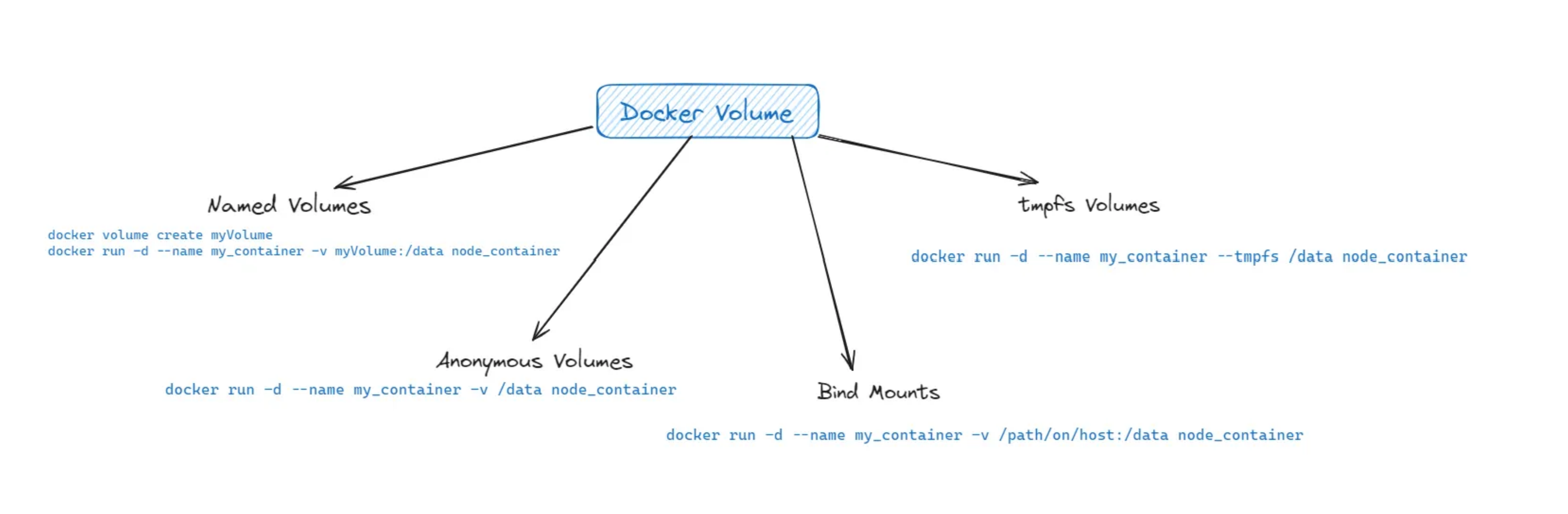
Direct connection between host and container.

- ✓ Direct mapping of host paths
- ✓ Good for development and sharing

```
>_ docker run -v /host/path:/container/path
```

B. Storytelling: Docker Concepts in-action

2. Docker Storage



Named Volumes

```
docker volume create myVolume
docker run -d --name my_container -v myVolume:/data node_container
```

tmpfs Volumes

```
docker run -d --name my_container --tmpfs /data node_container
```

Anonymous Volumes

```
docker run -d --name my_container -v /data node_container
```

Bind Mounts

```
docker run -d --name my_container -v /path/on/host:/data node_container
```

Source: https://therahulsarkar.medium.com/understanding-docker-volumes-a-comprehensive-guide-46339aa9ac53

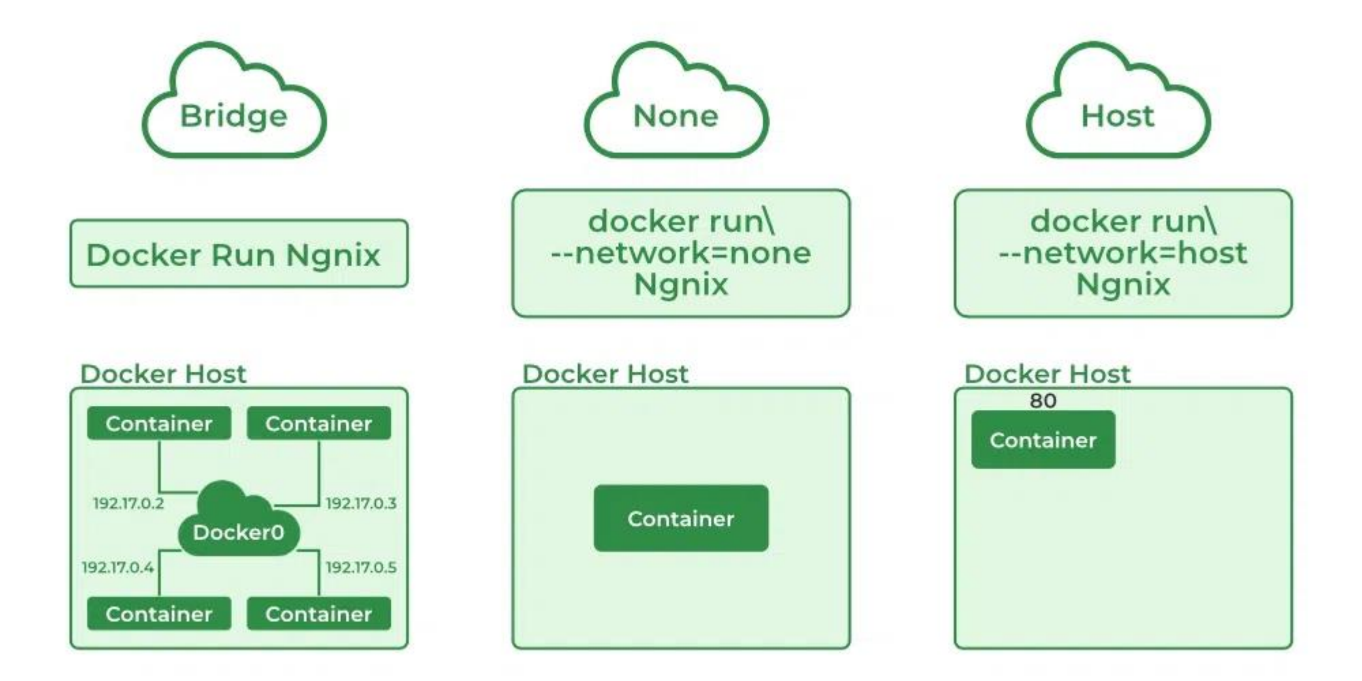
B. Storytelling: Docker Concepts in-action

4. Docker Networking

What is Docker Network?

A Docker Network is a **virtual network** that allows isolated containers to communicate with each other and with the outside world.

Docker Network Types



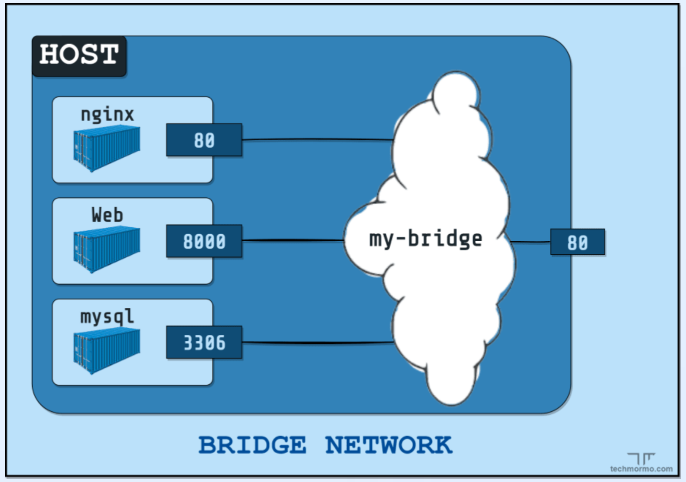
Source: https://www.geeksforgeeks.org/devops/basics-of-docker-networking/

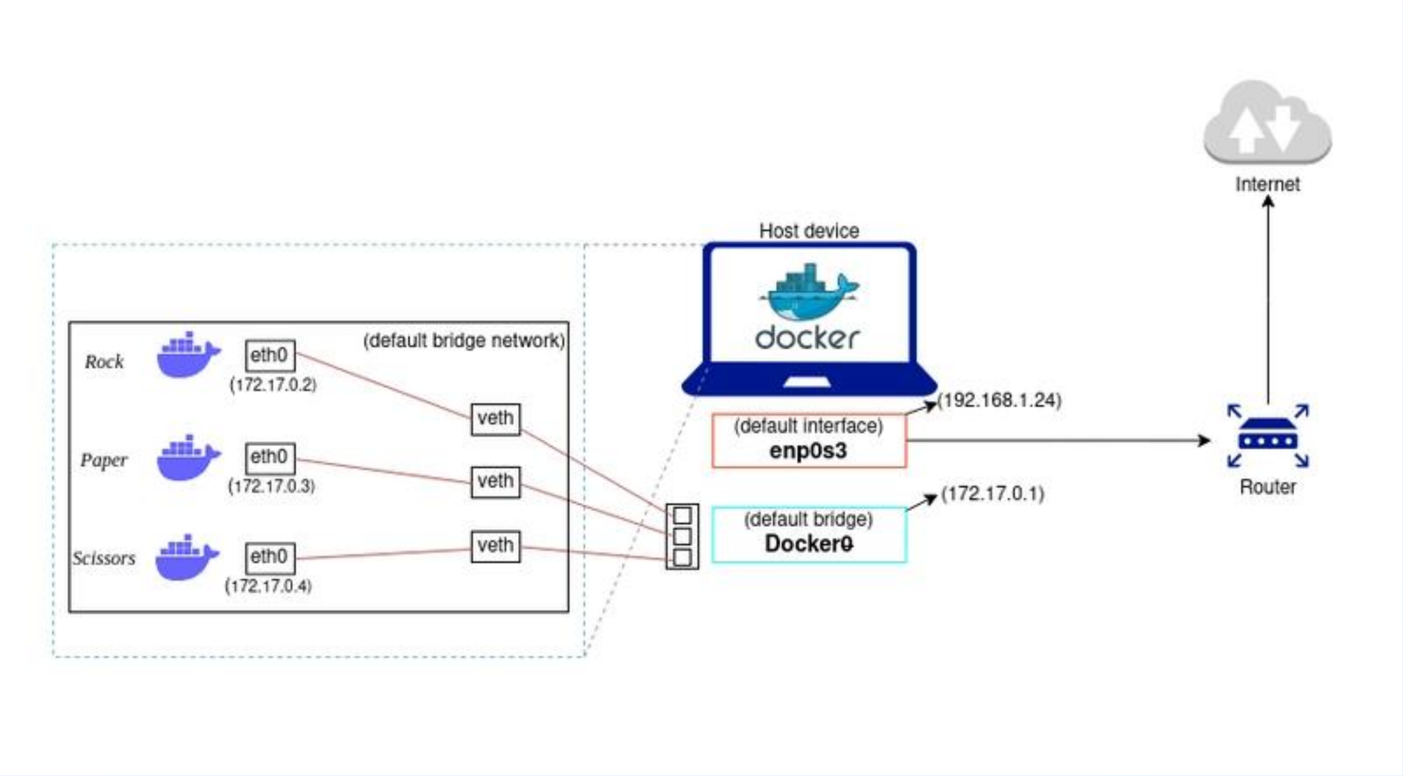
B. Storytelling: Docker Concepts in-action

⇄ Bridge (Default)

Single host container default network.

- ✓ Containers can communicate
- ✓ Isolated from other networks





Bridge Type

Source: https://www.geeksforgeeks.org/devops/basics-of-docker-networking/

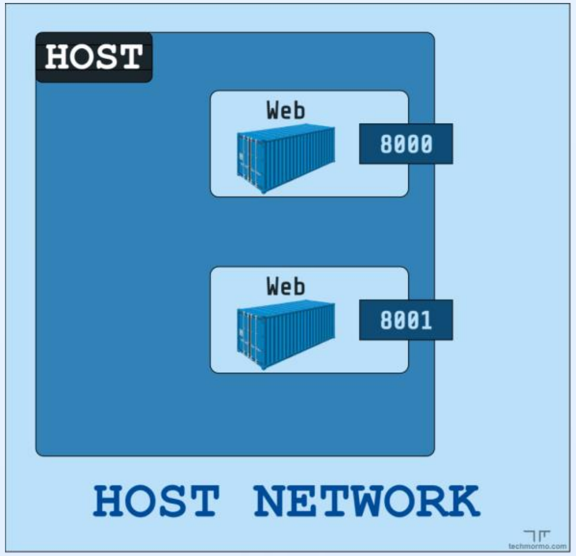
Docker Networking Fundamentals

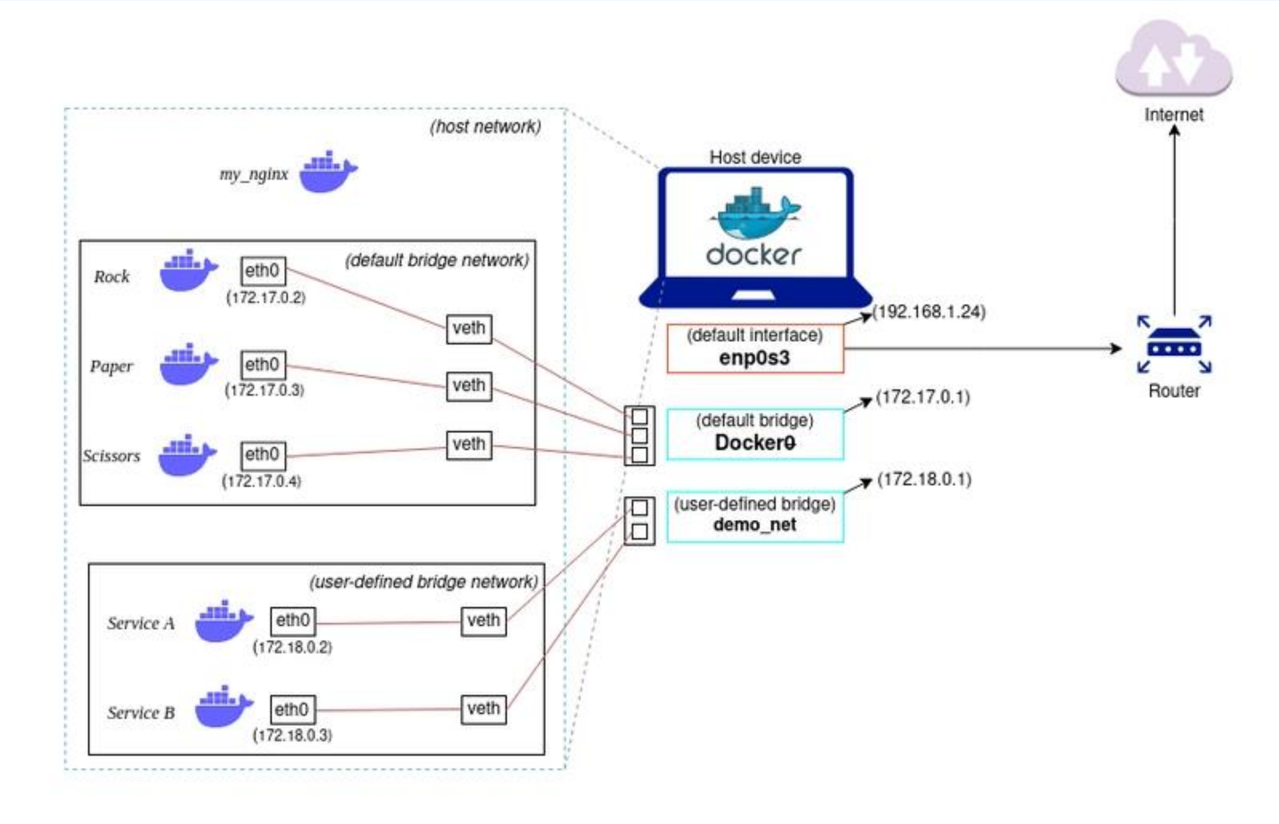
4. Docker Networking

Host

Shares host machine network stack.

- ✓ High performance
- ⚠ No isolation





Host Type

Source: https://www.geeksforgeeks.org/devops/basics-of-docker-networking/

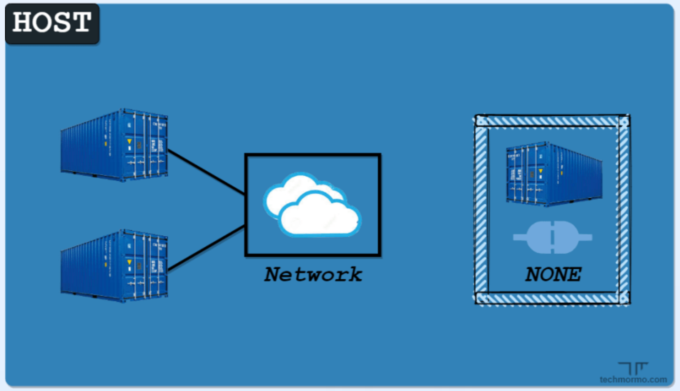
B. Storytelling: Docker Concepts in-action

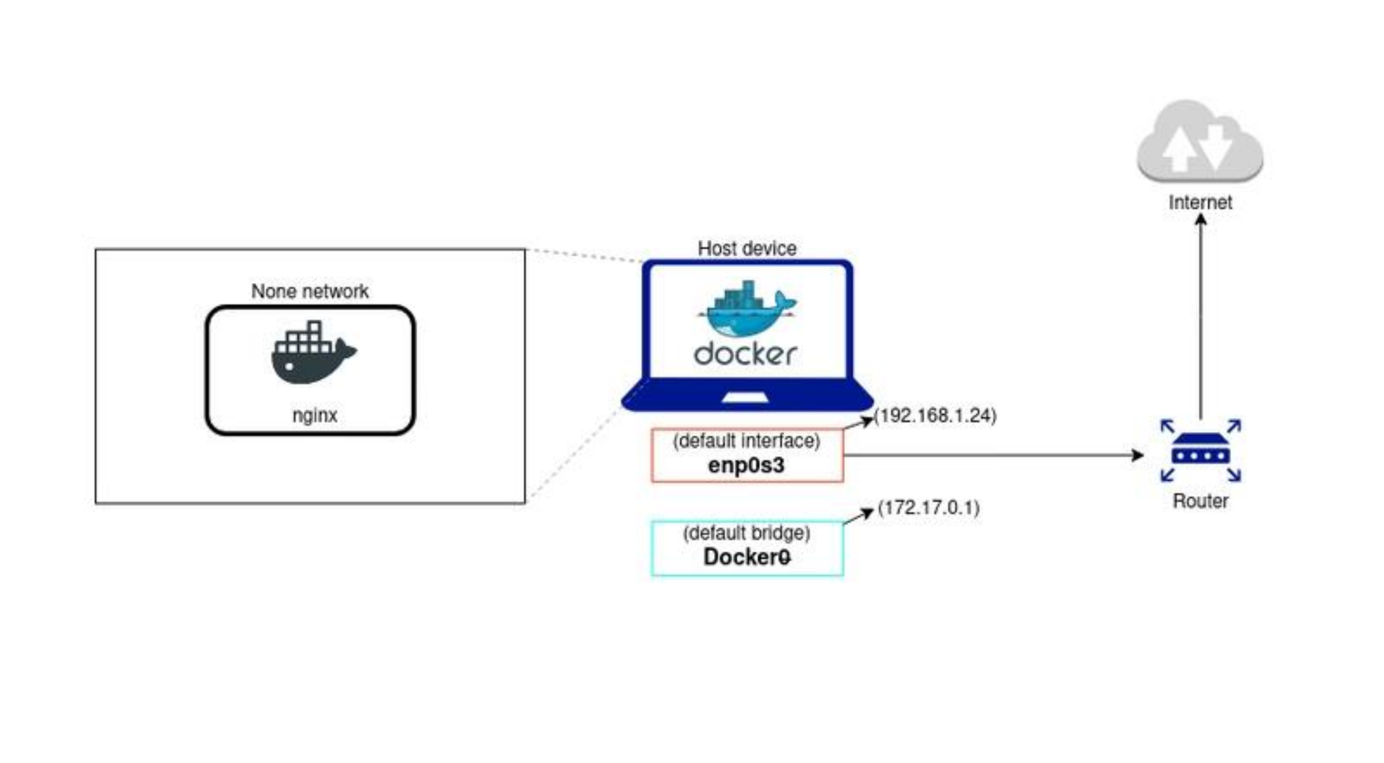
4. Docker Networking

🚫 None

No network connection.

✓ Complete isolation





None Type

Source: https://www.geeksforgeeks.org/devops/basics-of-docker-networking/

B. Storytelling: Docker Concepts in-action

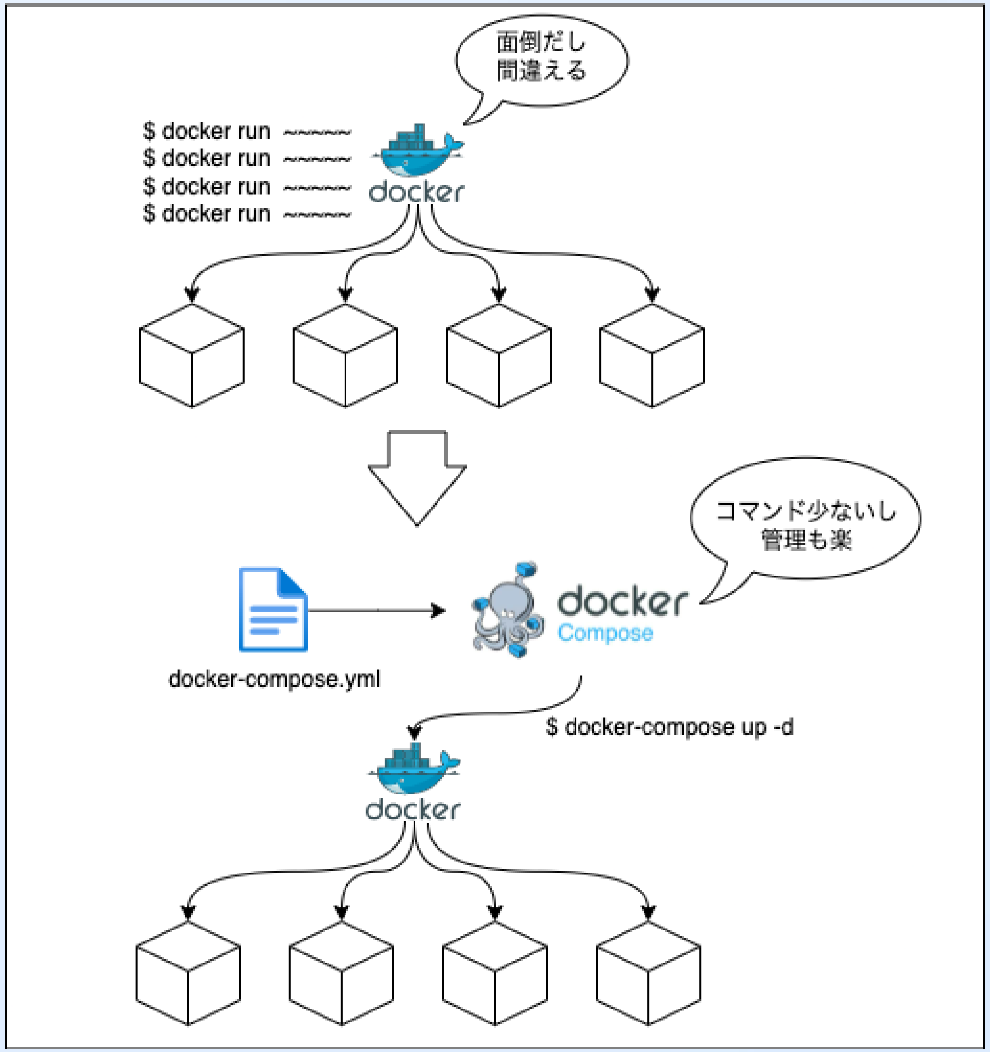
5. Docker Compose

What is Docker Compose?

- Tool for defining and running multi-container Docker applications
- Configuration via YAML file
- CLI commands to start/stop all services

Why Use Docker Compose?

- **Easy management:** Simplifies deployment and management of complex applications
- **Enhanced collaboration:** Team members can work in the same environment through shared Compose files
- **Time savings:** Reuse existing container configurations
- **Cross-environment compatibility:** Support for different settings (configuration profiles, override mechanisms)



Source: https://hitolog.blog/2022/02/19/docker_tutorial_docker-compose/

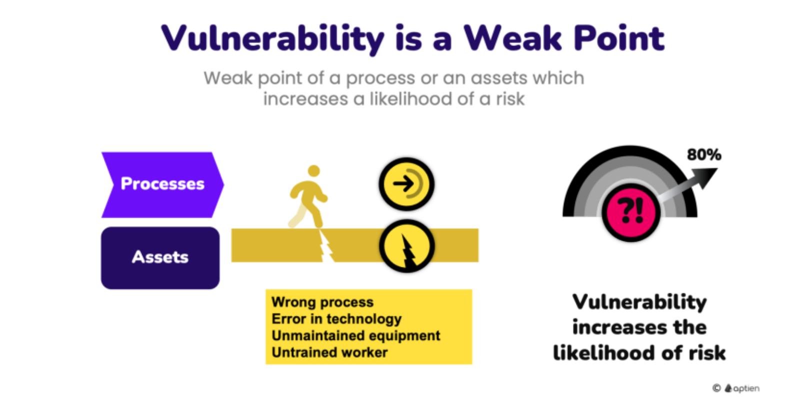
B. Storytelling: Docker Concepts in-action

6. Docker Security

B. Storytelling: Docker Concepts in-action

6. Docker Security

Common threats



Vulnerability is a Weak Point

Weak point of a process or an assets which increases a likelihood of a risk

Processes

Assets

Wrong process
Error in technology
Unmaintained equipment
Untrained worker

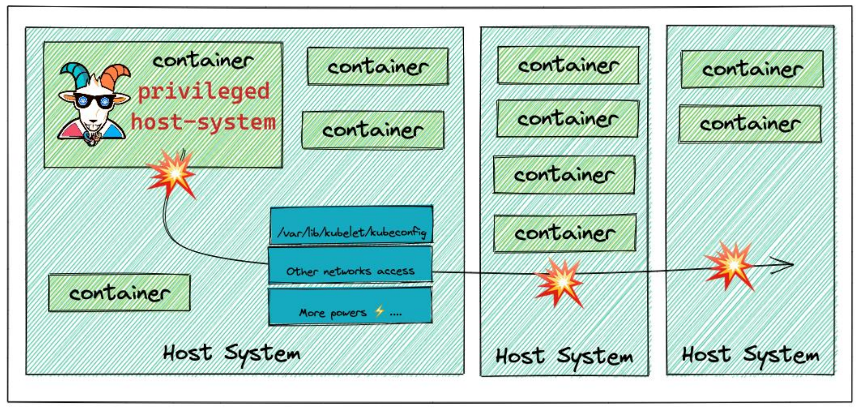
80%

?!

Vulnerability increases the likelihood of risk

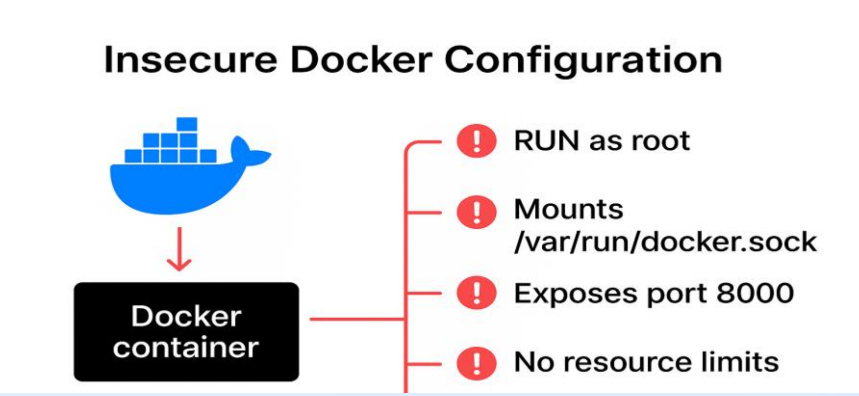
© aptien

Image vulnerabilities



Container escape

Data breaches



Insecure Docker Configuration

Docker container

- RUN as root
- Mounts /var/run/docker.sock
- Exposes port 8000
- No resource limits

Insecure configuration

B. Storytelling: Docker Concepts in-action

6. Docker Security

Security practices

Principle of Least Privilege
Containers and users should have only the minimum privileges needed.

Official Images
Use official images that are typically more secure and regularly updated.

Image Vulnerability Scanning
Use tools to detect known vulnerabilities in images.

Non-root Users
Run containers as non-root users to reduce potential damage.

Limit Container Capabilities
Restrict system calls that containers can execute.

Regular Updates
Keep Docker and images up to date with security patches.

Production Readiness

Performance Optimization

- ✓ **Multi-stage builds** to reduce image size
- ✓ **Optimized layer caching** for build times
- ✓ **Resource limits** for CPU and memory

Troubleshooting & Debugging

- ✓ **Docker logs** for container output
- ✓ **Docker exec** to run commands in containers
- ✓ **Health checks** for container status

C. Lesson Learned & Wrap-up

C. Lesson Learned & Wrap-up

Recap key points

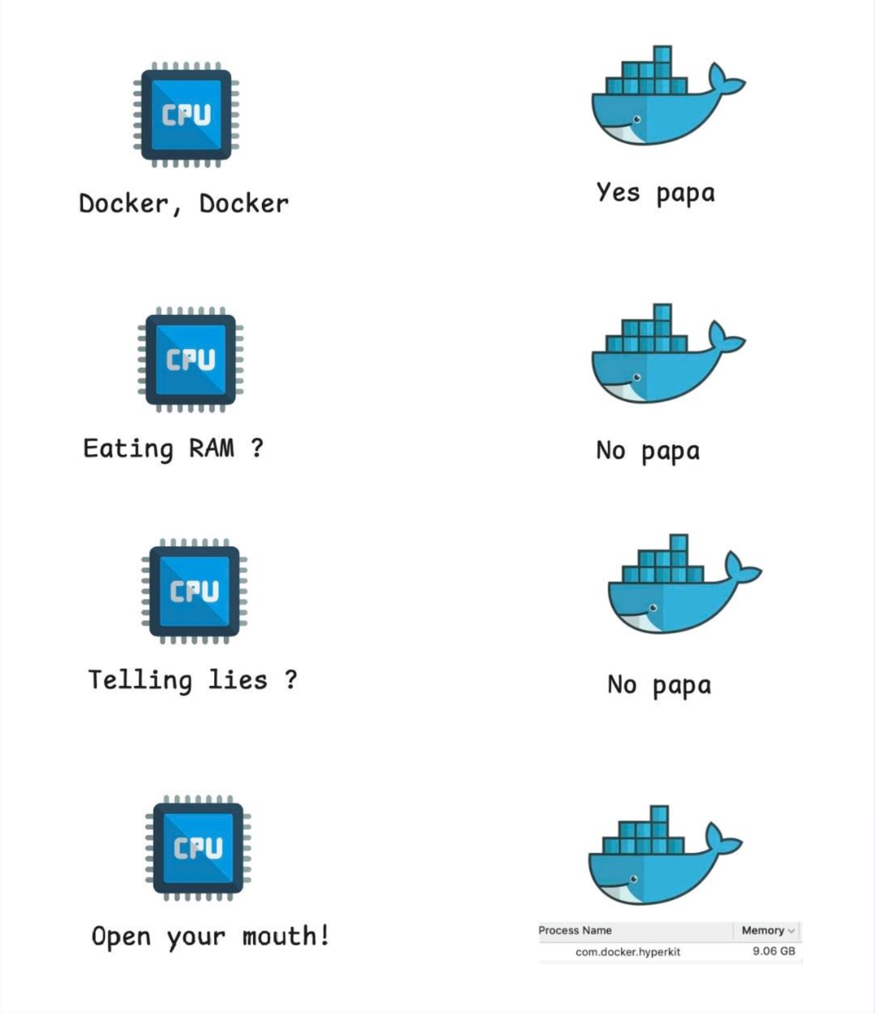
- ✓ Docker basics: container, image, Dockerfile, networking, volume
- ✓ Docker Compose for multi-container apps
- ✓ Docker security best practices
- ✓ Production readiness: performance optimization, troubleshooting

Lesson learned

- ✓ Docker ensures consistent environments → no “works on my machine”.
- ✓ Images, containers, networks, and volumes are the building blocks.
- ✓ Compose helps manage multi-container apps easily.
- ✓ Security and optimization are essential in production.

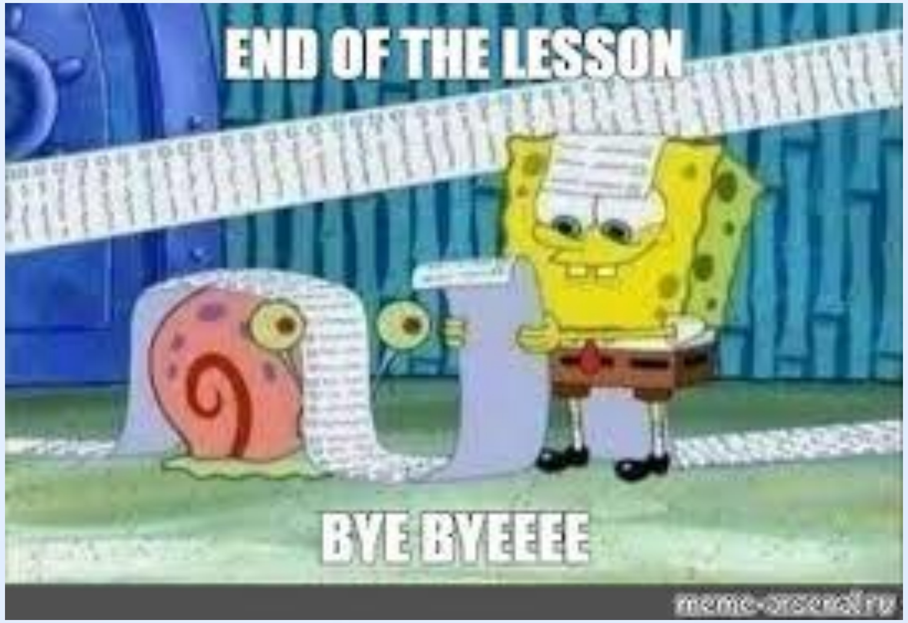
Advanced topics

- ✓ Orchestration with Kubernetes
- ✓ CI/CD integration
- ✓ Docker Swarm
- ✓ Docker in cloud environments (AWS, GCP, Azure)



Reference

- https://docs.docker.com/get-started/workshop/
- https://github.com/alexryabtsev/docker-workshop
- https://docs.docker.com/
- https://dev.to/nobleman97/docker-networking-101-a-blueprint-for-seamless-container-connectivity-3i5b
- https://blog.devops.dev/linux-containers-deep-dive-c0668a4f347d



Thank you !