Docker

CMSC398W: Practical Tools For Efficient Development

Mohammad Durrani

June 22, 2026

It Works On My Machine

You clone a teammate's repo and run their code. Much to your surprise, it crashes.

- Their Python is 3.11 but yours is 3.9.
- They installed `libpq-dev` on Ubuntu and unfortunately you're on macOS.
- Their `.env` file sets a path that doesn't exist on your machine.

After an hour of debugging, it finally works but it doesn't have to be this hard

What Docker Solves

The Problem

Software depends on more than just its code:

- Runtime version (Python 3.11, Node 20, JDK 17)
- System libraries (`libssl`, `libpq`, `glibc` version)
- OS-level config (ports, file paths, env vars)
- Other services (database, cache, message queue)

What Docker Solves

The Problem

Software depends on more than just its code:

- Runtime version (Python 3.11, Node 20, JDK 17)
- System libraries (`libssl`, `libpq`, `glibc` version)
- OS-level config (ports, file paths, env vars)
- Other services (database, cache, message queue)

Docker's Answer

Package the code and its entire environment together.

What is Docker?

Docker

Docker is a platform that enables developers to build, deploy, run, and manage applications in a containerized fashion using images.

Where Docker Shows Up

Contexts

Context	What Docker Does
Local dev	Every dev gets identical environment
CI pipelines	Tests run in the same env as production
Deployment	Push the same image you tested
Microservices	Each service runs in its own isolated container

Where Docker Shows Up

Contexts

Context	What Docker Does
Local dev	Every dev gets identical environment
CI pipelines	Tests run in the same env as production
Deployment	Push the same image you tested
Microservices	Each service runs in its own isolated container

The Pattern

"Build once, run anywhere."

Think-Pair-Share: What's the Difference?

The Challenge

You want to run an isolated Linux environment on your laptop. You could use a virtual machine or a container.

The Question

What do you think is the key difference between running a VM and running a container? What might each one be better at?

Think-Pair-Share: What's the Difference?

The Challenge

You want to run an isolated Linux environment on your laptop. You could use a virtual machine or a container.

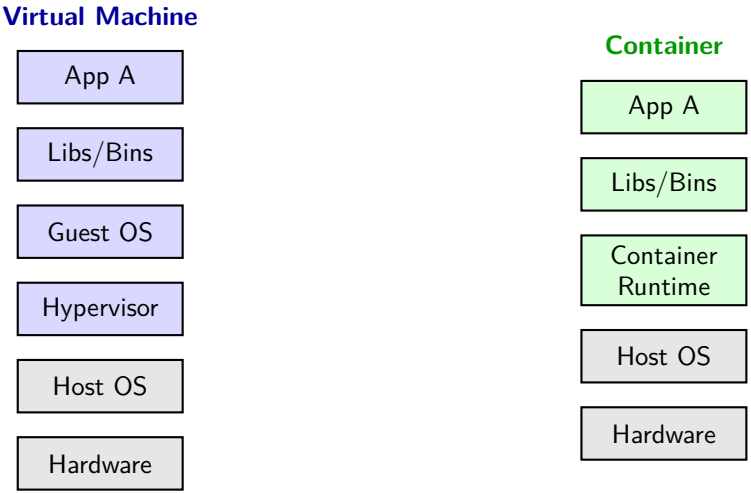
The Question

What do you think is the key difference between running a VM and running a container? What might each one be better at?

Answer

VMs virtualize hardware; each gets its own OS kernel. Containers share the host kernel and isolate only the filesystem and processes.

Containers vs VMs: Side by Side



Containers vs VMs: Tradeoffs

Comparison

Property	VM	Container
Startup time	Minutes	Milliseconds
Size	GBs (full OS)	MBs (just app + libs)
Isolation	Full OS boundary	Process + filesystem
Portability	Heavyweight	Lightweight
Use case	Full OS needed	App packaging

Containers vs VMs: Tradeoffs

Comparison

Property	VM	Container
Startup time	Minutes	Milliseconds
Size	GBs (full OS)	MBs (just app + libs)
Isolation	Full OS boundary	Process + filesystem
Portability	Heavyweight	Lightweight
Use case	Full OS needed	App packaging

Note on Security

Containers share the host kernel, so a kernel exploit can break out of container isolation. VMs provide a harder boundary since each has its own kernel.

Two Key Concepts

Image

A Docker image is a read-only blueprint. It contains:

- A filesystem snapshot (OS files, libraries, your code)
- Metadata (what command to run, what port to expose)
- Layers: each build step adds a layer on top of the previous

Two Key Concepts

Image

A Docker image is a read-only blueprint. It contains:

- A filesystem snapshot (OS files, libraries, your code)
- Metadata (what command to run, what port to expose)
- Layers: each build step adds a layer on top of the previous

Container

A container is a running instance of an image. When you start a container, Docker adds a thin writable layer on top of the image. The image itself never changes.

Docker Concepts

Class vs Object

Concept	Docker
Class definition	Image
Object instance	Container
new MyClass()	docker run myimage

One image $\rightarrow$ many containers running simultaneously, each isolated from the others.

Core Objects

Primitives

Primitive	What It Is
Image	Read-only filesystem snapshot + metadata
Container	Running instance of an image
Volume	Persistent storage that outlives containers
Network	Virtual network connecting containers

Core Objects

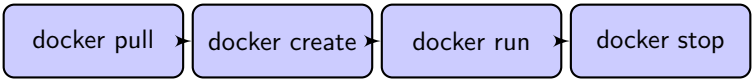
Primitives

Primitive	What It Is
Image	Read-only filesystem snapshot + metadata
Container	Running instance of an image
Volume	Persistent storage that outlives containers
Network	Virtual network connecting containers

Registry

A registry stores and distributes images. Docker Hub is the public default. Companies run private registries (AWS ECR, GitHub Container Registry). `docker pull nginx` fetches from Docker Hub.

Lifecycle of a Container



`docker rm` removes the container. `docker rmi` removes the image.

Dockerfile Basics

A `Dockerfile` is a script of instructions that builds an image layer by layer.

Common Instructions

Instruction	Purpose
`FROM`	Base image to start from
`RUN`	Execute a shell command during build
`COPY`	Copy files from host into the image
`WORKDIR`	Set working directory for subsequent commands
`ENV`	Set environment variable in the image
`EXPOSE`	Document which port the app listens on
`CMD`	Default command when container starts

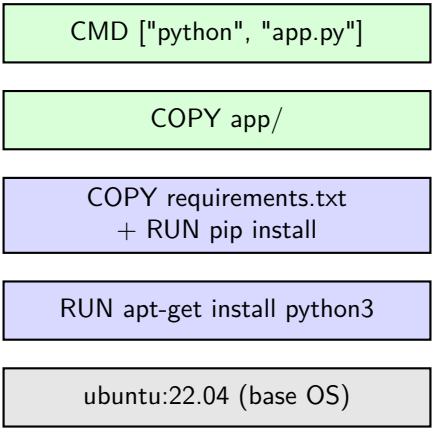
A Flask App Dockerfile

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir \
    -r requirements.txt

COPY . .

EXPOSE 5000

CMD ["python", "app.py"]
```

- `python:3.11-slim` small official base
- `WORKDIR` before any copies
- Copy `requirements.txt` first, then `pip install`
- Copy source code last

Image Layers

Each instruction in a Dockerfile adds a layer which are then cached and shared.



Change `app.py` → only the top two layers rebuild. Dependencies are cached.

Think-Pair-Share: Dockerfile Order

The Challenge

A teammate writes this Dockerfile:

```
FROM node:20-slim
WORKDIR /app
COPY . .
RUN npm install
CMD ["node", "server.js"]
```

The Question

What's the problem with this layer order? How would you fix it?

Think-Pair-Share: Dockerfile Order

The Challenge

A teammate writes this Dockerfile:

```
FROM node:20-slim
WORKDIR /app
COPY . .
RUN npm install
CMD ["node", "server.js"]
```

The Question

What's the problem with this layer order? How would you fix it?

Answer

`COPY . .` copies all source code before `npm install`. Any code change will invalidate the install cache. How do you think we could fix this?

**Commands

Managing Images

Command	Description
`docker pull nginx`	Download image from registry
`docker images`	List local images
`docker rmi nginx`	Remove a local image
`docker build -t myapp .`	Build image from Dockerfile

Running Containers: Flags

Flag	Meaning
`-d`	Detach
`-p 8080:80`	Map host p 8080 to container 80
`-v ./data:/data`	Mount a volume
`-e KEY=val`	Set environment variable
`--name web`	Give the container a name
`--rm`	Auto-remove when it exits

```
# Run nginx, port 8080 on host
# maps to port 80 in container
docker run -d \
  -p 8080:80 \
  --name web \
  nginx

# Open a shell in a running container
docker exec -it web bash

# See logs
docker logs web
```

The Problem With Container Storage

Containers Are Ephemeral

When a container stops and is removed, everything written inside it is gone.

The Problem With Container Storage

Containers Are Ephemeral

When a container stops and is removed, everything written inside it is gone.

Persistence

Volumes decouple storage from the container lifecycle. Data lives on the host and the containers just mount it.

Volumes in Practice

Named Volume for a Database

```
# Create named volume
docker volume create pgdata

# Mount it into postgres container
docker run -d \
  -e POSTGRES_PASSWORD=secret \
  -v pgdata:/var/lib/postgresql/data \
  postgres:16

# Volume persists across container restarts/rm
```

Volumes in Practice

Named Volume for a Database

```
# Create named volume
docker volume create pgdata

# Mount it into postgres container
docker run -d \
  -e POSTGRES_PASSWORD=secret \
  -v pgdata:/var/lib/postgresql/data \
  postgres:16

# Volume persists across container restarts/rm
```

Key Point

Docker stores named volumes on the host, outside any container. This means that if you remove the container, the data stays.

Why Compose?

The Problem

Most apps need multiple services running together:

- Flask backend on port 5000
- PostgreSQL database
- Redis cache
- Nginx reverse proxy

Running each with a long `docker run` command and wiring them together by hand is annoying.

Why Compose?

The Problem

Most apps need multiple services running together:

- Flask backend on port 5000
- PostgreSQL database
- Redis cache
- Nginx reverse proxy

Running each with a long `docker run` command and wiring them together by hand is annoying.

Docker Compose

Define the whole app in one `docker-compose.yml` file and bring everything up or down with one command.

A Compose File

```
services:
  web:
    build: .
    ports:
      - "5000:5000"
    environment:
      DATABASE_URL: XX
    depends_on:
      - db

  db:
    image: postgres:16
    environment:
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: appdb
    volumes:
```

- `build: .` build from local Dockerfile
- `image:` pull from registry
- `depends_on` start order (not readiness)
- `volumes:` named volume declared once, used anywhere
- Services talk to each other by service name (`db`)

Compose Commands

Everyday Workflow

Command	Description
`docker compose up`	Start all services (foreground)
`docker compose up -d`	Start all services (detached)
`docker compose down`	Stop and remove containers
`docker compose down -v`	Also remove volumes
`docker compose logs web`	Stream logs from `web` service
`docker compose exec web bash`	Shell into running service
`docker compose build`	Rebuild images
`docker compose ps`	Status of all services

Think-Pair-Share: Service Startup Order

The Challenge

The Flask app starts immediately and crashes because PostgreSQL isn't ready yet, even though `depends_on: db` is set in the Compose file.

The Question

Why doesn't `depends_on` solve this? What approaches would actually fix it?

Think-Pair-Share: Service Startup Order

The Challenge

The Flask app starts immediately and crashes because PostgreSQL isn't ready yet, even though `depends_on: db` is set in the Compose file.

The Question

Why doesn't `depends_on` solve this? What approaches would actually fix it?

Answer

`depends_on` only waits for the container to start, not for the DB to be ready to accept connections. Fix: use `depends_on: condition: service_healthy` with a healthcheck, or add retry logic in the app.

Healthchecks in Compose

Making depends_on Wait for Readiness

```
services:
  db:
    image: postgres:16
    healthcheck:
      test: ["CMD-SHELL",
             "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 5

  web:
    build: .
    depends_on:
      db:
        condition: service_healthy
```