04834580 Software Engineering (Honor Track) 2025-26

Docker and Containerization

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

“It Works on My Machine”

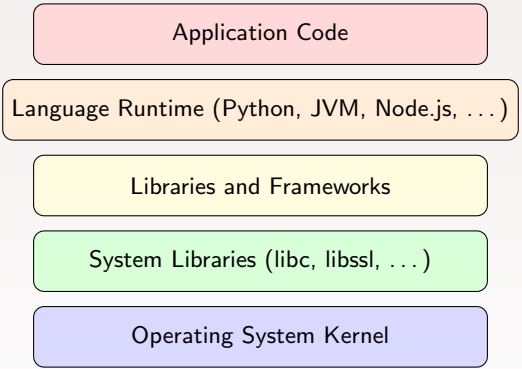
A universal problem in software engineering:

- Developer writes and tests code on their laptop — it works
- Code is deployed to a server — it breaks
- Another developer clones the repo — it doesn't build

Why does this happen?

- Different operating system or OS version
- Different library versions installed
- Different configuration files or environment variables
- Missing system-level dependencies

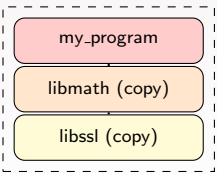
What Does Software Need to Run?



Each layer can differ between machines, causing failures.

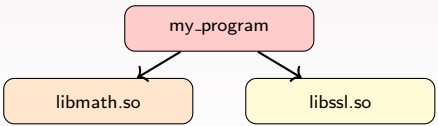
Static Linking

- All library code is copied into the executable at compile time
- Resulting binary is self-contained
- Larger file size
- No dependency on external `.so`/`.dll` files



Dynamic Linking

- Libraries are loaded at runtime from the system
- Smaller executables, shared memory
- Requires correct library version to be present
- Used by most Linux software (ELF + `.so`)



Inspecting Dynamic Dependencies

北京大学

On Linux, use `ldd` to list shared libraries required by a binary:

```
$ ldd /usr/bin/python3
    linux-vdso.so.1 (0x00007ffc...)
    libpython3.12.so.1.0 => /lib64/libpython3.12.so.1.0
    libm.so.6 => /lib64/libm.so.6
    libc.so.6 => /lib64/libc.so.6
    /lib64/ld-linux-x86-64.so.2
```

Key insight: If any of these libraries is missing or has an incompatible version on the target machine, the program will fail to start.

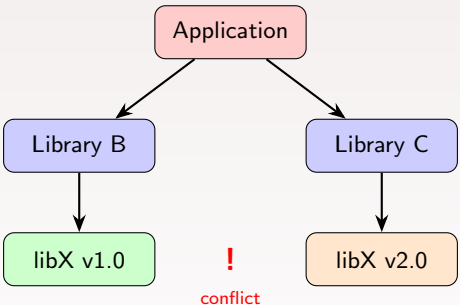
Definition (Dependency Hell [1])

A situation where software depends on specific versions of libraries, and these version requirements conflict with each other or with what is installed on the system.

Common manifestations:

- **Diamond dependency:** A depends on B and C; B requires libX v1, C requires libX v2
- **DLL Hell** (Windows): installing one program breaks another by overwriting shared DLLs
- **Version pinning conflicts:** package A pins library to 1.2, package B pins to 1.3

Diamond Dependency Problem



Only one version of libX can exist in a single filesystem. How do we resolve this?

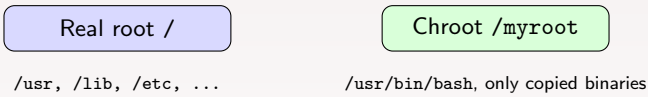
Approaches to Solving the Deployment Problem

北京大学

1. **Language-level package managers** (pip, npm, Maven)
   Manage application dependencies, but not system-level ones
2. **System package managers** (apt, yum, dnf)
   Manage system libraries, but only one version per system
3. **Virtual environments** (Python venv, nvm)
   Isolate dependencies per project, but only for one language
4. **Virtual machines**
   Full OS isolation, but heavyweight (GBs of disk, minutes to boot)
5. **Containers**
   Lightweight OS-level isolation — the best of both worlds?

Process Isolation: `chroot`

- `chroot` [2] — "change root" — has existed in UNIX since 1979
- Changes the apparent root directory for a process and its children
- The process sees a different filesystem, cannot access files outside



Limitation: `chroot` only isolates the filesystem. It does not isolate processes, network, users, or resource limits.

Linux **namespaces** provide isolation for different system resources:

Namespace	What it isolates
PID	Process IDs (process sees itself as PID 1)
NET	Network interfaces, routing tables
MNT	Mount points (filesystem)
UTS	Hostname and domain name
IPC	Inter-process communication
USER	User and group IDs
CGROUP	Cgroup root directory

A **container** is essentially a process (or group of processes) running with its own set of namespaces.

Control Groups (cgroups)

While namespaces provide **isolation** (what a process can see), **cgroups** provide **resource limits** (what a process can use):

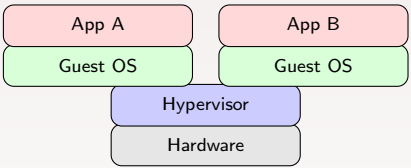
- **CPU:** Limit CPU time or pin to specific cores
- **Memory:** Set maximum memory usage; OOM kill if exceeded
- **I/O:** Throttle disk read/write bandwidth
- **PIDs:** Limit number of processes (prevent fork bombs)

Together: namespaces + cgroups + a layered filesystem = a container.

Containers vs Virtual Machines

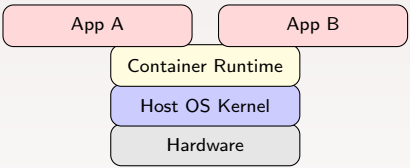
北京大学

Virtual Machines



- Each VM runs a full OS
- Heavyweight: GBs of disk
- Minutes to start
- Strong isolation

Containers



- Share the host kernel
- Lightweight: MBs of disk
- Seconds to start
- Process-level isolation

What is Docker?

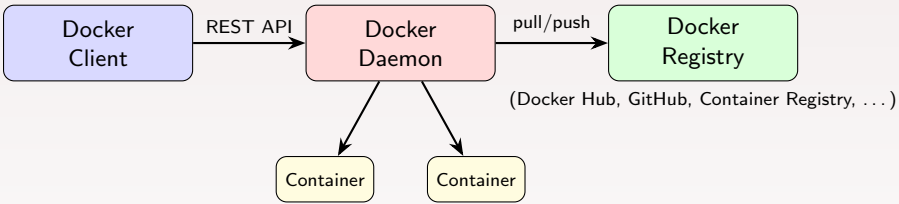
北京大学

- Docker [3] is a platform for building, shipping, and running applications in containers.
- Released in 2013 by Solomon Hykes (originally dotCloud).
- Docker did not invent containers — Linux had LXC since 2008 — but made them **accessible** and **practical**.

Key contributions:

- Simple CLI and Dockerfile format
- Layered image filesystem
- Docker Hub — a public registry for sharing images
- Standardized via OCI (Open Container Initiative) [4]

Docker Architecture



- **Client:** `docker` CLI sends commands to the daemon
- **Daemon:** Manages images, containers, networks, volumes
- **Registry:** Stores and distributes images

Definition (Docker Image)

A read-only template containing the application code, runtime, libraries, and configuration needed to run the software.

Definition (Docker Container)

A running instance of an image — a lightweight, isolated process with its own filesystem, network, and process space.

Analogy: an image is like a **class**, a container is like an **object** (instance).

You can run multiple containers from the same image.

Basic Docker Commands

北京大学

```
# Pull an image from Docker Hub
$ docker pull ubuntu:24.04

# Run a container interactively
$ docker run -it ubuntu:24.04 /bin/bash

# List running containers
$ docker ps

# List all containers (including stopped)
$ docker ps -a

# Stop a running container
$ docker stop <container_id>

# Remove a container
$ docker rm <container_id>

# List downloaded images
$ docker images
```

What is a Dockerfile?

A **Dockerfile** [5] is a text file containing instructions to build a Docker image.

- Each instruction creates a new **layer** in the image
- Layers are cached — unchanged layers are reused
- Dockerfiles make builds **reproducible** and **self-documenting**

Core philosophy: **Infrastructure as Code**

- The environment specification lives alongside the source code
- Versioned, reviewed, and tested like any other code

Dockerfile: A Simple Example

北京大学

```
# Start from a base image
FROM python:3.12-slim

# Set working directory
WORKDIR /app

# Copy dependency specification
COPY requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Define the command to run
CMD ["python", "app.py"]
```

```
$ docker build -t myapp .
$ docker run myapp
```

Key Dockerfile Instructions

北京大学

Instruction	Purpose
`FROM`	Base image to build upon
`RUN`	Execute a command during build (creates a layer)
`COPY`	Copy files from host to image
`ADD`	Like `COPY`, but also handles URLs and tar extraction
`WORKDIR`	Set the working directory for subsequent instructions
`ENV`	Set environment variables
`EXPOSE`	Document which ports the container listens on
`CMD`	Default command when container starts
`ENTRYPOINT`	Fixed command; `CMD` becomes its arguments

CMD vs ENTRYPOINT

CMD

- Provides the default command
- Can be **overridden** by the user at `docker run`

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

`docker run myapp` → runs `python app.py`
`docker run myapp bash` → runs `bash`

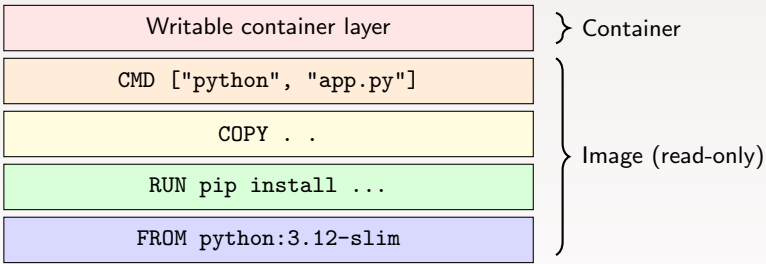
ENTRYPOINT

- Fixed executable
- User arguments are **appended**

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

`docker run myapp` → runs `python app.py`
`docker run myapp test.py` → runs `python test.py`

Each Dockerfile instruction creates a **read-only layer**. A container adds a thin **writable layer** on top.



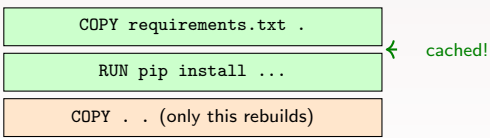
- Layers are shared between images — saves disk space
- If a layer hasn't changed, it is reused from cache

Build Cache: Why Order Matters

Docker caches each layer. If a layer's input changes, all subsequent layers are **invalidated**.

Bad: Copy all files first, then install dependencies.
Changing *any* source file invalidates the dependency install layer.

Good: Copy dependency file first, install, then copy source.
Changing source files does not re-trigger dependency installation.



The Problem: Large Images

A typical build process needs tools that are **not needed at runtime**:

- Compilers (gcc, javac, go build)
- Build tools (make, gradle, npm)
- Development headers and debug symbols

Including build tools in the final image:

- Wastes disk space and bandwidth (100s of MBs)
- Increases the attack surface for security
- Slows down deployment

Multi-stage Builds [6]

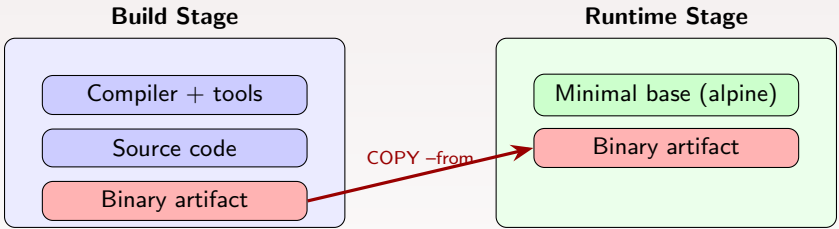
Use multiple `FROM` statements. Copy only the artifacts you need from earlier stages.

```
# Stage 1: Build
FROM golang:1.22 AS builder
WORKDIR /app
COPY . .
RUN go build -o server .

# Stage 2: Runtime
FROM alpine:3.19
COPY --from=builder /app/server /usr/local/bin/server
CMD ["server"]
```

- `golang:1.22` image: ~800 MB (includes compiler, tools)
- Final `alpine` image: ~15 MB (just the binary + minimal OS)

Multi-stage Build: Conceptual View



The build stage is discarded. Only the runtime stage becomes the final image.

Dockerfile Best Practices

1. **Use specific base image tags** — `python:3.12-slim`, not `python:latest`
2. **Order instructions by change frequency** — copy dependency files before source code for better caching
3. **Use multi-stage builds** — separate build and runtime environments
4. **Use** `.dockerignore` — exclude `.git`, `node_modules`, build artifacts from the build context
5. **Don't run as root** — use `USER` instruction to switch to a non-root user
6. **Minimize the number of layers** — combine related `RUN` commands with `&&`
7. **Use small base images** — `alpine` or `-slim` variants

Summary

- The deployment problem stems from software depending on specific environments
- Dynamic linking creates version conflicts (dependency hell)
- Linux namespaces + cgroups enable lightweight OS-level isolation (containers)
- Docker made containers practical: simple CLI, layered images, registries
- Dockerfiles = infrastructure as code = reproducible environments
- Multi-stage builds keep images small and secure
- Docker Compose orchestrates multi-container applications

Further reading: Docker documentation [7], Merkel 2014 [3]

References I

[1] Wikipedia contributors.
DLL Hell.
https://en.wikipedia.org/wiki/DLL_Hell, 2025.

[2] Wikipedia contributors.
chroot.
https://en.wikipedia.org/wiki/Chroot, 2025.

[3] Dirk Merkel.
Docker: lightweight Linux containers for consistent development and deployment.

Linux journal, 2014(239):2, 2014.

[4] Open Container Initiative.
Open container initiative runtime specification.
https://opencontainers.org/, 2024.

References II

[5] Docker, Inc.
Dockerfile reference.
https://docs.docker.com/engine/reference/builder/, 2025.

[6] Docker, Inc.
Multi-stage builds.
https://docs.docker.com/build/building/multi-stage/, 2025.

[7] Docker, Inc.
Docker documentation.
https://docs.docker.com/, 2025.