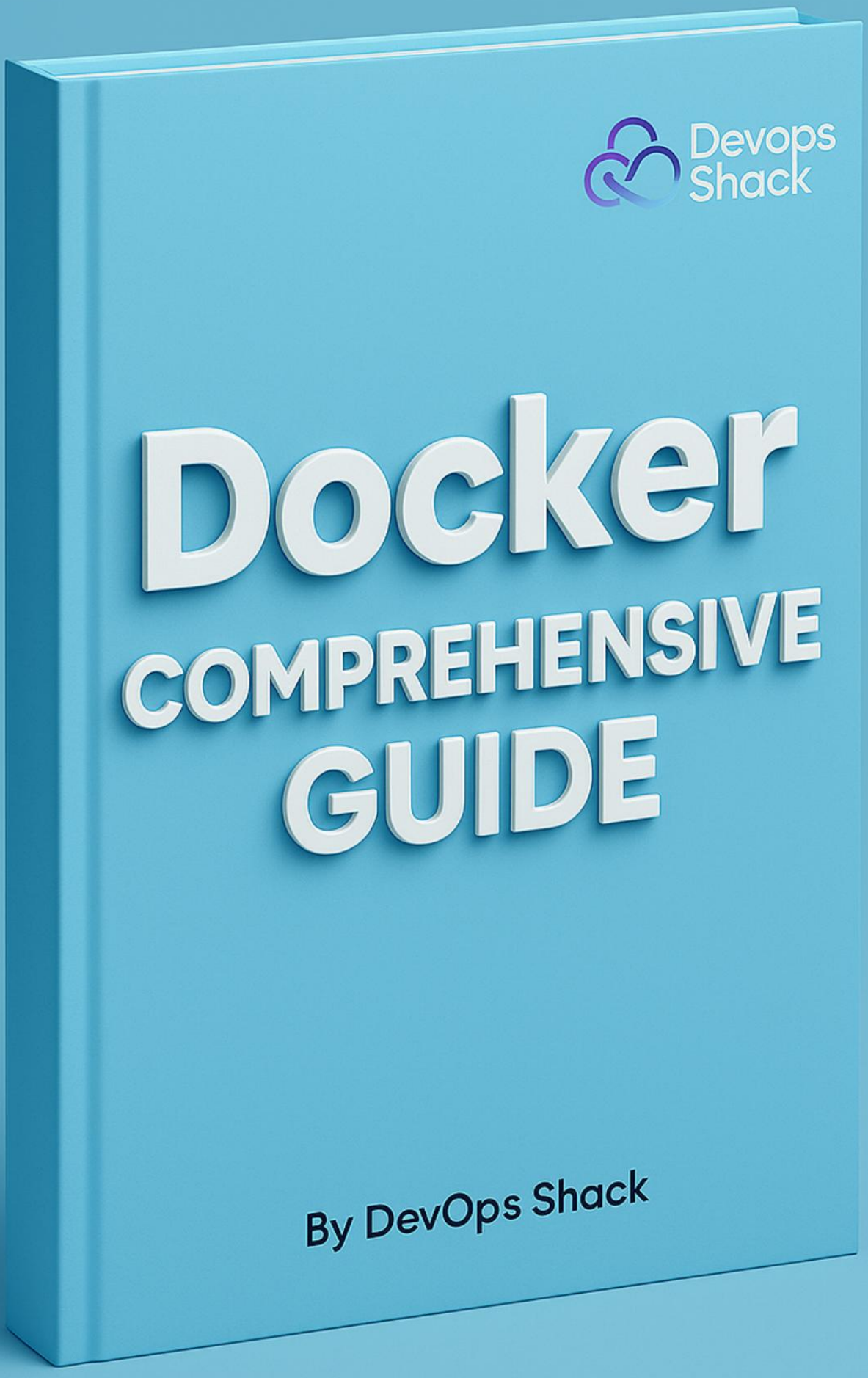

Devops Shack
Docker
COMPREHENSIVE GUIDE
By DevOps Shack

DOCKER COMPREHENSIVE GUIDE

Module 1: Introduction to Docker

1.1 What is Docker?

Docker is an open-source platform designed to simplify building, shipping, and running applications by using **containerization**. In essence, Docker allows developers to package applications with all the necessary dependencies (binaries, libraries, configurations) into a single container image that can run reliably across different computing environments.

A **Docker container** is a lightweight, stand-alone, executable package of software that includes everything needed to run an application. Containers share the same OS kernel, which makes them **much faster, smaller, and more efficient** than virtual machines.

1.2 The Evolution: From Bare Metal to Containers

Before we appreciate Docker, we must understand where it fits in the evolution of software deployment:

- **Bare Metal Era**
  - Applications ran directly on physical machines.
  - One app per server → Massive underutilization of resources.
  - Scaling meant buying more servers (CAPEX-intensive).
  - No isolation → Conflicts between software versions.
- **Virtual Machine Era**
  - Introduced hardware-level virtualization via **hypervisors** (e.g., VMware, VirtualBox, KVM).
  - Allowed multiple OS instances (VMs) on a single physical host.
  - Still heavyweight: Each VM carries its own full-blown OS.
  - Slow boot times, large images, high resource consumption.
- **Container Era (Docker Revolution)**
  - Containers provide **OS-level virtualization**.
  - Share the host's kernel while isolating the app in a user space.
  - Super lightweight → Instant startup, minimal overhead.
  - Repeatable builds, consistent environments, easier CI/CD.

1.3 Virtual Machines vs Docker Containers

Feature	Virtual Machines	Docker Containers
Boot Time	Minutes	Seconds or less
OS	Full OS per VM	Shared host OS kernel
Resource Usage	High (RAM, CPU, Disk)	Low
Image Size	GBs	MBs (multi-stage builds)
Portability	Low	High
Performance	Hypervisor overhead	Near-native performance
Use Case	Isolation, Legacy apps	Microservices, DevOps, CI/CD

1.4 Real-World Analogy: Shipping Containers

Imagine the logistics of transporting goods across the world.

- Before standardized **shipping containers**, transporting goods required handling individual items, repackaging, and often led to damage, delays, and confusion.
- After containers were standardized, **logistics became modular**. Goods could be transported by truck, ship, rail – regardless of what's inside – using a consistent packaging method.

Docker does this for software:

- Your code (goods) is packaged in a container image (shipping container).
- It can run on any Docker-enabled host (truck, ship, train).
- No need to worry about what's inside – it just works.

1.5 Core Concepts and Terminologies

✅ **Docker Image**

- A **read-only template** with the application and all its dependencies.
- Built using a **Dockerfile**.
- Multiple layers (cached, optimized).
- Can be versioned and shared (e.g., via Docker Hub).

✅ **Docker Container**

- A **runtime instance** of an image.
- Lightweight and isolated.
- Ephemeral by default (unless data is persisted via volumes).
- Executes the app defined in the image.

✅ **Dockerfile**

- A script containing instructions to **build an image**.
- Line-by-line instructions: FROM, COPY, RUN, CMD, EXPOSE, ENV, etc.

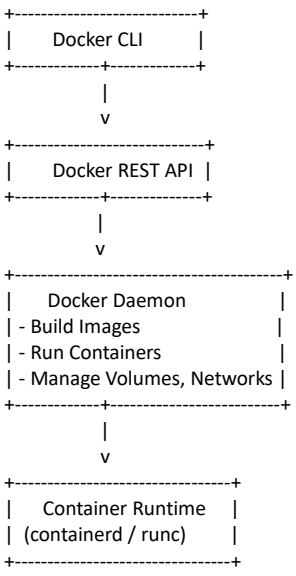
✅ **Docker Engine**

- The **core runtime** that executes containers.
- Manages lifecycle: build, run, stop, remove containers.
- Consists of:
  - **Docker Daemon (dockerd)**: Background service.
  - **Docker CLI**: Command-line interface.
  - **REST API**: For third-party tools or programmatic control.

✅ **Docker Hub**

- Docker's **official public registry**.
- Store, share, and distribute Docker images.
- Private repos and automated builds supported.

1.6 Docker Architecture (Visualized)



Key Points:

- The **CLI** talks to the **daemon** via REST API.
- The **daemon** delegates to lower-level runtimes like containerd and runc.
- All of this is transparent to the user, who only interacts with the CLI or Docker Desktop GUI.

1.7 The Rise of Docker in DevOps & Cloud-Native

🌎 **Docker Changed the Game by Enabling:**

1. **Consistency Across Environments**
    - No more "It works on my machine."
    - Same container runs in dev, staging, and prod.
2. **Rapid CI/CD Pipelines**
    - Spin up throwaway environments for each commit.
    - Run tests in isolated containers.
3. **Microservices Adoption**
    - Each service in its own container.
    - Easy scaling, independence, fault isolation.
4. **Cloud Portability**
    - Container images can run on AWS, GCP, Azure, on-prem.
5. **Infrastructure Simplification**
    - Reduced reliance on complex VM provisioning.

1.8 Docker Ecosystem

Docker is not just about containers. It's a full ecosystem:

Component	Description
Docker Engine	Core service that runs and manages containers
Docker CLI	Tool for interacting with the engine
Docker Compose	Tool for managing multi-container applications
Docker Swarm	Native clustering and orchestration
Docker Hub	Registry to host and distribute container images

Component	Description
Docker Desktop	GUI + local environment for Windows/macOS
Docker Registry	Private image registry (can be self-hosted)
BuildKit	Faster, advanced image builder with better caching
Docker Scout	Image analysis and security scanning
Docker Extensions	Plugin ecosystem for UI and developer tools

---

1.9 Real-World Docker Use Cases

🚀 **1. Application Containerization**

- Monolith or microservice – isolate and run securely in containers.
- Common stacks: Node.js + MongoDB, Django + PostgreSQL, Go APIs.

🧪 **2. Automated Testing**

- CI jobs can spin up containerized environments.
- Tools like Cypress, Selenium, JMeter run tests in containers.

🌎 **3. Cloud-Native Development**

- Docker + Kubernetes = cornerstone of cloud-native apps.
- Rapid prototyping and deployment.

📦 **4. Legacy Application Modernization**

- Containerize old apps for portability without rewriting.

📈 **5. Scaling and Horizontal Expansion**

- Run multiple instances of the same container behind a load balancer.

Module 2: Installing Docker

🚀 2.1 Overview: What Does Docker Installation Actually Mean?

When we say “install Docker,” we’re talking about setting up the entire runtime ecosystem that allows you to:

- Build Docker images
- Run Docker containers
- Manage volumes, networks, registries
- Interact via CLI or GUI (like Docker Desktop)

🔧 Installing Docker Means Installing:

Component	Purpose
dockerd (Docker Daemon)	Runs in the background, manages images/containers
docker (CLI)	User interface to interact with the daemon
containerd	Lightweight runtime used internally by Docker
runc	OCI-compliant runtime to run containers
docker-compose	For managing multi-container apps
buildkit	Optimized image build system

Different operating systems (Linux, Windows, macOS) come with different installation approaches. Let’s go platform-by-platform.

🐧 2.2 Docker Installation on Linux (Ubuntu/Debian)

✅ Step-by-Step Guide (Official Docker CE)

```
# Step 1: Remove old Docker versions (if any)
sudo apt remove docker docker-engine docker.io containerd runc

# Step 2: Update package index
sudo apt update

# Step 3: Install required packages for HTTPS over apt
sudo apt install -y ca-certificates curl gnupg lsb-release

# Step 4: Add Docker’s official GPG key
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
```

```
 sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

# Step 5: Set up the stable Docker repository
echo \
 "deb [arch=$(dpkg --print-architecture) \
 signed-by=/etc/apt/keyrings/docker.gpg] \
 https://download.docker.com/linux/ubuntu \
 $(lsb_release -cs) stable" | \
 sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Step 6: Install Docker packages
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

# Step 7: Post-installation check
sudo docker version
sudo docker run hello-world
```

🔐 **Optional: Run Docker Without sudo**

```
sudo groupadd docker   # May show 'already exists'

sudo usermod -aG docker $USER

newgrp docker

docker ps
```

⚠️ **Security Note**: Adding your user to the Docker group gives root-equivalent access.

2.3 Docker Installation on macOS (Intel & Apple Silicon)

Docker runs on macOS through **Docker Desktop**, a packaged application that bundles the Docker Engine, CLI, Docker Compose, and a lightweight Linux VM using Apple's Hypervisor or QEMU.

💻 **Installation Steps:**

1. Go to: https://www.docker.com/products/docker-desktop/
2. Download the correct installer:
    - Apple Silicon (.dmg for M1/M2)
    - Intel-based Mac (.dmg for x86_64)
3. Install by dragging the Docker icon to Applications
4. Launch Docker Desktop from Spotlight
5. Allow system permissions (if prompted)
6. Open Terminal and verify:

docker version

docker run hello-world

🛑 **Common Issues on macOS**

- **macOS file permissions**: Docker volumes may have permission denied errors; bind mount to /Users/... paths only.
- **VPN conflicts**: Some VPNs interfere with Docker Desktop's internal network.
- **Rosetta emulation**: M1 chips need compatible images; use multi-arch images.

🪟 **2.4 Docker Installation on Windows (Windows 10/11 Pro)**

Docker on Windows comes in two flavors:

- **Docker Desktop** – GUI interface + CLI, preferred for most users.
- **Docker on WSL2 backend** – Lightweight CLI-only installation, ideal for advanced users.

✅ **Installing Docker Desktop on Windows (GUI)**

1. Install **Windows Subsystem for Linux (WSL2)** if not already present:

wsl --install

2. Enable virtualization in BIOS and **Hyper-V** in Windows Features
3. Download Docker Desktop from:
   https://www.docker.com/products/docker-desktop/
4. Run the installer
5. After installation:
   - It will set up WSL backend with a Linux distribution (usually Ubuntu)
   - Start Docker Desktop from the Start Menu
6. Validate with PowerShell or CMD:

docker version

docker run hello-world

⚙️ **Advanced: Docker CLI Without Docker Desktop (Using WSL2)**

1. Install WSL2 and Ubuntu from Microsoft Store
2. Inside Ubuntu (WSL), follow the Linux install instructions (same as 2.2)
3. Add Docker CLI path to Windows Environment Variable (optional)

Docker Inc. recently made Docker Desktop license-commercial. CLI + WSL method avoids that for enterprise users.

2.5 Testing Docker Installation (Cross-Platform)

Regardless of OS, once Docker is installed, validate it using:

```
docker info          # Shows runtime, drivers, cgroup version
docker version       # Show CLI and daemon versions
docker run hello-world
```

Expected Output:

Hello from Docker!

This message shows that your installation appears to be working correctly.

🔐 2.6 Rootless Docker (Advanced)

Docker usually needs root privileges, but **rootless mode** is designed for security-constrained environments.

📦 **Use Cases:**

- Multi-user systems (e.g., university servers)
- CI/CD agents
- Running containers without exposing the host kernel directly

Setup on Linux:

```
# Step 1: Install Docker binaries
curl -fsSL https://get.docker.com/rootless | sh

# Step 2: Add binary path
export PATH=$HOME/bin:$PATH

# Step 3: Start the rootless daemon
systemctl --user start docker

# Step 4: Test
docker context use rootless
docker info | grep Rootless
```

✅ Note: Some features (e.g., bind mounts to /) are restricted in rootless mode.

🛡️ 2.7 Post-Installation Security Recommendations

Security Task	Command or Notes
Audit Docker group	getent group docker
Disable TCP socket	Avoid -H tcp://0.0.0.0:2375
Protect Docker socket (/var/run/docker.sock)	Avoid mounting it into containers
Use secrets manager	Use Docker Secrets / HashiCorp Vault
Enable AppArmor or SELinux	Enforce policies via profile plugins
Disable legacy --privileged containers	Reduces kernel attack surface

---

🗄️ 2.8 Docker Versions: CE vs EE vs Docker Engine

Type	Description
Docker CE	Community Edition – free, open-source, used by most developers
Docker EE	Enterprise Edition – includes extra security, support, UCP/DTR (deprecated)
Moby	Upstream open-source project for Docker Engine

💡 Today, Docker CE is still the go-to edition for almost everyone, with **Kubernetes** handling orchestration instead of EE's UCP.

🧰 2.9 Docker Desktop Alternatives (Advanced Use Cases)

Tool	Use Case	Comment
Podman	Rootless container runtime	Compatible CLI (alias docker=podman)
Rancher Desktop	Docker Desktop alternative w/ Kubernetes	Lightweight, GUI-based
Colima	Containers on macOS (w/ Lima VM)	Open-source, faster on M1
Minikube + Docker	Use Docker as K8s container runtime	Good for testing K8s

---

2.10 Behind-the-Scenes: What Happens on docker run?

1. CLI sends request to Docker Daemon over UNIX socket
2. Daemon pulls image (if not cached)
3. Daemon creates container filesystem (layer union)
4. Allocates network namespace, PID namespace
5. Starts container with given entrypoint
6. Returns container ID to user

docker inspect <container_id> shows detailed JSON of configuration, mounts, IPs, etc.

2.11 Common Installation Problems & Fixes

Problem	**Fix**
Docker daemon not running	sudo systemctl start docker
“Cannot connect to Docker daemon”	Check socket permission or sudo
WSL2 networking issues (Windows/macOS)	Restart Docker Desktop, reconfigure
Proxy issues	Set HTTP_PROXY and HTTPS_PROXY
"docker: command not found"	Add Docker path to $PATH or reinstall

Module 3: Docker CLI

🔧 3.1 The Role of Docker CLI

The **Docker CLI** (docker) is the primary interface to communicate with the **Docker Daemon** (dockerd). Every time you run a command like docker run, you're instructing the daemon to perform an action like starting a container, pulling an image, or inspecting container metadata.

You'll use the Docker CLI for:

- Creating and managing containers
- Building and tagging images
- Managing networks and volumes
- Logging, debugging, and monitoring
- Communicating with registries
- Orchestrating multi-container applications

🏁 3.2 Core Docker CLI Commands (Real-World Breakdown)

📦 docker run – Run a New Container

Syntax:

docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Example:

docker run -d -p 8080:80 nginx

🔍 Deep Dive:

Option	Meaning
-d	Detached mode (run in background)
-p	Publish container port to host
--name	Assign name to the container
--rm	Automatically remove after exit
-e	Set environment variables
-v	Mount volumes

Option	Meaning
--network	Join container to a network
--entrypoint	Override default ENTRYPOINT
--restart	Restart policy (e.g., always, on-failure)
--cpu-shares, --memory	Resource limits

Real-world:

```
docker run --name webapp \
  -e NODE_ENV=production \
  -v "$(pwd)"/app:/usr/src/app \
  -p 3000:3000 \
  node:20-alpine npm start
```

🛠️ docker exec – Run Commands Inside a Running Container

Syntax:

```
docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

Examples:

```
docker exec -it myapp bash
docker exec mydb pg_isready
```

Real-world:

- Debug an app live.
- Check logs or configuration files inside.
- Run health checks manually.

-i, -t:

- -i: Interactive (keep STDIN open)
- -t: Allocate pseudo-TTY (terminal-like behavior)

🔍 docker ps – List Running Containers

Syntax:

```
docker ps [OPTIONS]
```

Examples:

docker ps

docker ps -a # All containers (including stopped)

docker ps -q # Only container IDs

docker ps --format '{{.Names}}: {{.Status}}'

Common fields:

- Container ID
- Image
- Command
- Created time
- Status
- Ports
- Names

❌ docker stop / docker kill – Terminate Containers

Syntax:

docker stop [OPTIONS] CONTAINER

docker kill [OPTIONS] CONTAINER

Difference:

stop	**Sends SIGTERM → waits (default: 10s) → then SIGKILL**
kill	Immediately sends SIGKILL

---

docker rm – Remove Stopped Containers

docker rm container-name

docker rm $(docker ps -aq) # Remove all stopped

📂 docker images – List Local Images

docker images

docker images -f dangling=true # Unused intermediary layers

🗑️ docker rmi – Remove Images

docker rmi nginx

docker rmi $(docker images -q)

📦 docker build – Build Docker Image from Dockerfile

Syntax:

docker build [OPTIONS] PATH | URL | -

Examples:

docker build -t devopsshack/webapp:1.0 .

docker build --no-cache --build-arg NODE_ENV=production .

Flags:

Option	Meaning
-t	Tag (name:tag)
--build-arg	Pass arguments to Dockerfile
--no-cache	Skip cache during build
--platform	Target architecture (linux/arm64, linux/amd64)

🔍 docker inspect – Low-Level Container/Image Metadata

docker inspect container_id

docker inspect --format '{{.Config.Env}}' container_id

Returns massive JSON including:

- Mounts
- Networks
- Config
- Volumes
- PID namespace
- Runtime stats

📜 docker logs – Container Logs

docker logs container-name

docker logs -f --tail 100 myapp

Flag	Purpose
-f	Follow logs (live tail)
--tail	Limit output to last N lines

📈 docker stats – Live Resource Usage

docker stats

Monitors:

- CPU usage %
- Memory consumption
- Net I/O
- Block I/O
- PIDs

🧠 docker top – Show Running Processes in Container

docker top myapp

Like ps inside container – shows PID, CMD, CPU, memory per process.

🧼 docker system – Cleanups

docker system prune # Remove unused containers/images/networks

docker system df # Disk usage report

🌐 3.3 Docker Networking CLI

🔌 List networks:

docker network ls

🔍 Inspect network:

docker network inspect bridge

Create custom network:

docker network create mynet

🧩 **Run containers on same network:**

docker run --network=mynet --name app1 nginx

docker run --network=mynet --name app2 busybox ping app1

💾 **3.4 Docker Volumes & Bind Mounts via CLI**

📋 **List volumes:**

docker volume ls

🧠 **Inspect volume:**

docker volume inspect myvol

📦 **Create named volume:**

docker volume create pgdata

📌 **Real-World Bind Mount:**

docker run -v $(pwd)/code:/app myapp

📦 **Real-World Volume Use:**

docker run -v pgdata:/var/lib/postgresql/data postgres

⚙️ **3.5 Docker Environments and Variables**

Set Environment Variables in Container:

docker run -e ENV=production myapp

Use .env File:

docker --env-file ./my.env run nginx

Access inside container:

printenv ENV

🔁 **3.6 Restart Policies**

Control how containers behave on host/system restart:

docker run --restart=always myapp

Policy	Meaning
no	Default – don't restart
on-failure	Restart only on non-zero exit
always	Always restart
unless-stopped	Restart unless manually stopped

🎯 3.7 EntryPoint vs CMD – What Actually Runs?

In Dockerfile:

ENTRYPOINT ["node"]

CMD ["app.js"]

🧠 **Behavior:**

- ENTRYPOINT = fixed executable
- CMD = arguments passed to ENTRYPOINT

You can override CMD with CLI:

docker run myapp index.js

You can override ENTRYPOINT:

docker run --entrypoint "bash" myapp

🔐 3.8 Access and Permissions

Docker socket:

ls -l /var/run/docker.sock

Add user to Docker group:

sudo usermod -aG docker $USER

Security warning: Docker group = root access.

📥 3.9 Docker Pull, Push, and Tag

Pull image:

docker pull nginx:1.25

Tag local image:

docker tag myapp devopsshack/myapp:v1

Push to Docker Hub:

docker push devopsshack/myapp:v1

Must be logged in:

docker login

🔧 3.10 Advanced CLI: Debugging & Lifecycle

🔍 docker container inspect

Get mounted volumes, ports, health check info.

💉 Health Checks:

In Dockerfile:

HEALTHCHECK CMD curl --fail http://localhost || exit 1

🔄 Lifecycle states:

- Created
- Running
- Paused
- Restarting
- Exited
- Dead

Monitor with:

docker ps -a --format '{{.Names}} {{.Status}}'

🧰 3.11 Pro Tips & Patterns

- Use --rm for temporary debugging containers:

docker run --rm -it busybox sh

- Use --log-driver=none for silence:

docker run --log-driver=none myapp

- Restart container quickly:

docker restart container_name

- Use named containers:

docker run --name webserver nginx

- Combine docker ps with grep:

docker ps | grep nginx

Module 4: Dockerfile Mastery – Ultra Deep Dive

🚀 4.1 What Is a Dockerfile?

A **Dockerfile** is a text-based blueprint that defines how a **Docker image** should be built. It contains a series of **layered instructions** that:

- Start from a **base image**
- Install dependencies
- Copy application code
- Define runtime configuration (e.g., ENV, ENTRYPOINT)
- Package it all into a **repeatable, portable image**

Think of it as **infrastructure as code for containers.**

🧱 4.2 Dockerfile Syntax: The Basic Building Blocks

✅ Fundamental Instructions

Instruction	Description
FROM	Base image (must be first non-comment line)
RUN	Execute shell commands while building image
COPY	Copy files from host into image
ADD	Similar to COPY but can also unpack archives or fetch remote URLs
CMD	Default command when container starts
ENTRYPOINT	Main executable of the container
ENV	Set environment variables
ARG	Build-time variables
WORKDIR	Set working directory
EXPOSE	Document ports the container listens on

Instruction	Description
USER	Change default user
VOLUME	Mark mount points for external volumes
LABEL	Metadata (author, version, etc.)
HEALTHCHECK	Define health probe for container
SHELL	Customize shell used in RUN (bash, sh, etc.)

🔍 4.3 Understanding Docker Image Layers

Every Dockerfile instruction (except some like ENV, LABEL, etc.) creates a **new image layer**.

Layers are:

- **Cached** to speed up future builds
- **Immutable** and stacked
- **Shared** across images (saves disk/network)

Example:

FROM ubuntu:20.04 # Layer 1

RUN apt update # Layer 2

RUN apt install -y curl # Layer 3

COPY . /app # Layer 4

🔁 Modify COPY . /app? You invalidate Layer 4 **only**, rest remain cached.

🛠️ 4.4 Real-World Optimized Dockerfile Example

Let’s say you're deploying a Node.js app:

Stage 1: Builder

FROM node:20-alpine AS builder

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

RUN npm run build

```
# Stage 2: Runtime

FROM node:20-alpine

WORKDIR /app

COPY --from=builder /app/dist ./dist

COPY --from=builder /app/node_modules ./node_modules

CMD ["node", "dist/server.js"]
```

Why this is excellent:

✅ Multi-stage → final image is lightweight
✅ Avoids installing dev dependencies
✅ Isolates build and run environments
✅ Uses alpine base for smaller size
✅ Explicit CMD entry point

🏎️ 4.5 Optimizing Build Performance

🔁 Layer Caching Tips

- Place least-changing layers first
- Always separate COPY package*.json from rest of code
- Use .dockerignore to skip unnecessary files

📦 Use .dockerignore properly:

```
.git

node_modules

Dockerfile

*.log

coverage/
```

🧬 4.6 CMD vs ENTRYPOINT – Deep Dive

Feature	CMD	ENTRYPOINT
Default usage	Define default arguments	Define main executable
Overridable	Yes (easily overridden)	Yes, but requires --entrypoint
Combination	Can be combined together	ENTRYPOINT runs, CMD = args

Example:

ENTRYPOINT ["python3", "main.py"]

CMD ["--help"]

You can run:

docker run myscript --version

Which runs:

python3 main.py --version

📐 4.7 Multi-Stage Builds (Advanced Pattern)

Why Multi-Stage?

- Keep final image minimal
- Strip out unnecessary dependencies/tools
- Build once, reuse everywhere

Real-World Example: Go Binary

Stage 1: Build

FROM golang:1.21 AS builder

WORKDIR /src

COPY . .

RUN go build -o app

Stage 2: Runtime

FROM gcr.io/distroless/static

COPY --from=builder /src/app /app

ENTRYPOINT ["/app"]

✅ Final image: Secure, non-root, ~15MB
✅ No shell, no extra binaries, only the app

🧪 4.8 Advanced Dockerfile Patterns

🧹 Cleaning Up After RUN

```
RUN apt update && apt install -y \
  curl \
&& rm -rf /var/lib/apt/lists/*
```

Keeps image size small by:

- Deleting cache
- Reducing unnecessary files

🔁 Build Arguments (ARG)

```
ARG NODE_VERSION=20
FROM node:${NODE_VERSION}-alpine
```

Set during build:

```
docker build --build-arg NODE_VERSION=18 .
```

🔐 Environment Variables (ENV)

```
ENV PORT=3000 NODE_ENV=production
```

Used inside container:

```
process.env.PORT
```

📋 Health Checks

```
HEALTHCHECK --interval=30s --timeout=5s \
 CMD curl -f http://localhost:3000/health || exit 1
```

Adds status in docker inspect, helps orchestrators like Docker Swarm or Kubernetes.

👷 Use Non-Root Users

```
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser
```

Better security than running as root.

🔒 4.9 Distroless & Minimal Base Images

🔎 What is Distroless?

- No package manager, no shell, no glibc
- Only your app and runtime libraries
- Based on Google's secure containers

Example:

```
FROM golang:1.21 AS builder

WORKDIR /app

COPY . .

RUN go build -o server


FROM gcr.io/distroless/base

COPY --from=builder /app/server /

ENTRYPOINT ["/server"]
```

✅ Super secure
✅ Cannot be SSH-ed into
✅ Reduces attack surface

🧨 4.10 Common Mistakes & Anti-Patterns

Mistake	Fix
Using latest tag	Always pin exact version: node:20-alpine
Layer cache busting	Avoid COPY . . too early
Keeping dev tools in image	Use multi-stage to isolate
Missing .dockerignore	Always include it
No cleanup in RUN	Clean APT/YUM cache
Running as root	Use USER for app execution

4.11 Debugging Builds

Build with verbose output:

DOCKER_BUILDKIT=1 docker build .

View intermediate images:

docker history image-id

Run image with shell:

docker run -it --entrypoint sh myimage

📋 4.12 Real-World Dockerfile Templates

1. 🟢 Node.js

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
CMD ["node", "server.js"]
```

---

2. 🐍 Python Flask

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt ./
RUN pip install -r requirements.txt
COPY . .
ENV FLASK_APP=main.py
CMD ["flask", "run", "--host=0.0.0.0"]
```

---

3. ☕ Spring Boot

```
FROM eclipse-temurin:17-jdk
COPY target/app.jar /app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

---

4. 🔵 .NET Core

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0
COPY ./publish /app
WORKDIR /app
ENTRYPOINT ["dotnet", "myapp.dll"]
```

---

5. 🌐 NGINX Static Site

```
FROM nginx:1.25-alpine
COPY ./static /usr/share/nginx/html
```

🧠 4.13 Dockerfile Best Practices

- Pin versions (python:3.11-slim, node:20-alpine)
- Multi-stage builds for clean separation
- Reduce number of layers (combine RUN)
- Use .dockerignore aggressively
- Never hardcode secrets
- Avoid ADD unless needed (use COPY)
- Always validate with docker build . && docker run

Module 5: Docker Volumes & Storage

🧠 5.1 Why Persistent Storage Matters in Docker

By default, when a container stops or is deleted:

- All data stored **inside the container's writable layer** is lost.
- Containers are **ephemeral** — but your **data shouldn't be**.

Persistent storage is critical when:

- Running databases like PostgreSQL, MySQL, MongoDB
- Saving logs, user uploads, configuration files
- Sharing data between multiple containers
- Retaining build artifacts (CI/CD pipelines)

Solution: Volumes and Bind Mounts.

⚙️ 5.2 Types of Docker Storage Mechanisms

Type	Description	Use Case
Bind Mounts	Maps host directory to container	Full control, local dev
Named Volumes	Managed by Docker itself	Production-grade, portable
tmpfs Mounts	In-memory storage, non-persistent	Speed-critical temp storage
Volume Plugins	External storage backends	NFS, Amazon EFS, Portworx

---

📁 5.3 Bind Mounts – Full Control, High Flexibility

Syntax:

```
docker run -v /host/path:/container/path image
```

Example:

```
docker run -v $(pwd)/code:/app myapp
```

✅ **Pros:**

- Easy to edit files from host
- Useful in development
- No Docker volume needed

❌ **Cons:**

- Not portable across machines
- Security risk if host directory is compromised
- Performance varies depending on OS

📦 **5.4 Named Volumes – Portable, Managed by Docker**

Syntax:

docker volume create myvol

docker run -v myvol:/app/data myapp

Inspect Volume:

docker volume inspect myvol

✅ **Pros:**

- Managed entirely by Docker
- More secure (path is isolated)
- Easy to back up and restore
- Preferred for databases and production

❌ **Cons:**

- Opaque to the user (can't directly edit without Docker)
- Limited to local storage unless using a plugin

⚠️ **5.5 Differences: Bind Mount vs Named Volume**

Feature	Bind Mount	Named Volume
Managed by	You (host filesystem)	Docker
Path	Explicit /path on host	Automatically in /var/lib/docker/volumes
Portability	Low	High
Use in Dev	High	Moderate
Use in Prod	Risky	Preferred
Performance	Variable (esp. on macOS)	Optimized

🔐 5.6 Example: PostgreSQL with Named Volume

```
docker volume create pgdata


docker run -d \
  --name pg \
  -e POSTGRES_PASSWORD=mysecret \
  -v pgdata:/var/lib/postgresql/data \
  postgres:15
```

🔍 Inspect Volume:

```
docker volume inspect pgdata
```

🔁 Backup Volume:

```
docker run --rm -v pgdata:/data -v $(pwd):/backup alpine \
  tar czf /backup/pg-backup.tar.gz -C /data .
```

🔁 Restore Volume:

```
docker run --rm -v pgdata:/data -v $(pwd):/backup alpine \
  tar xzf /backup/pg-backup.tar.gz -C /data
```

🧬 5.7 Data Sharing Between Containers

📁 Shared Volume:

```
docker volume create sharedvol


docker run -d --name app1 -v sharedvol:/shared busybox tail -f /dev/null
docker run -d --name app2 -v sharedvol:/shared busybox tail -f /dev/null
```

Both app1 and app2 can access /shared and see the same files.

⚙️ 5.8 tmpfs Mounts – Temporary RAM Storage

```
docker run --tmpfs /cache:rw,size=100m alpine
```

Used when:

- You want data only during container runtime
- You need ultra-fast access (RAM-backed)

- You are dealing with sensitive data you want gone after shutdown

---

🔌 5.9 Docker Volume Plugins (Advanced)

Want to store Docker volumes **remotely or on specialized infrastructure**?

Use **volume plugins**:

Plugin	Use Case
local	Default driver
nfs	Mount from NFS
rexray/ebs	AWS EBS
glusterfs	Scalable file system
azurefile	Azure File Shares

Example: Using NFS Plugin

```
docker volume create \
  --driver local \
  --opt type=nfs \
  --opt o=addr=192.168.1.100,rw \
  --opt device=:/nfs/data \
  nfsvolume
```

Then mount:

```
docker run -v nfsvolume:/mnt alpine
```

🔍 5.10 Inspecting and Managing Volumes

```
docker volume ls              # List all volumes
docker volume inspect myvol   # View metadata
docker volume rm myvol        # Delete volume
docker system df              # Show disk usage
```

🚨 5.11 Common Mistakes with Volumes

Mistake	Problem
Using bind mount to host-sensitive path	Could overwrite system files
Forgetting to use volume in DB container	Data lost when container deleted
Assuming docker rm deletes volume	It doesn't unless --volumes used
Mounting read-only volumes as writable	App will crash on write
Not using .dockerignore with bind mount	Can lead to large unwanted context copies

---

🧰 5.12 Practical Volume Management Patterns

🧪 Development Workflow:

- Use bind mounts to code:

```
docker run -v $(pwd):/app node npm run dev
```

🧱 Production Workflow:

- Use named volumes:

```
docker volume create logs

docker run -v logs:/var/log/nginx nginx
```

🔁 Volume Lifecycle:

1. docker volume create
2. Attach to container
3. Optional backup/restore
4. docker volume rm to clean up

🔒 5.13 Security Implications of Volumes

- Never mount / or host root paths
- Protect host paths via AppArmor/SELinux
- Avoid exposing the Docker socket (/var/run/docker.sock) as volume
- Don't mount secrets unencrypted

🧪 5.14 Real-World Use Cases

✅ **Dev Environment with Live Code Reloading**

```
docker run -v $(pwd)/src:/app -v /app/node_modules myapp nodemon app.js
```

✅ **CI/CD Caching Volumes**

```
docker volume create cache
docker run -v cache:/root/.npm myapp npm install
```

✅ **Database Persistence**

```
docker volume create mongo_data
docker run -v mongo_data:/data/db mongo
```

✅ **Backups and Migration**

```
docker run --rm -v myvol:/vol -v $(pwd):/backup alpine \
 tar czf /backup/vol.tar.gz -C /vol .
```

🔁 5.15 Docker Compose & Volumes

```
version: "3.9"
services:
 db:
  image: postgres
  volumes:
   - pgdata:/var/lib/postgresql/data
 app:
  build: .
  volumes:
   - .:/app
volumes:
 pgdata:
```

Module 6: Docker Networking

🔧 6.1 Why Networking in Docker Matters

Every modern application is **network-dependent**. Whether you're:

- Connecting a backend to a database
- Exposing an API to the public
- Orchestrating microservices

Docker provides a **powerful virtual networking layer** that lets containers:

- Talk to each other securely
- Map internal ports to host ports
- Isolate environments
- Emulate production-like topologies locally

🌐 6.2 Types of Docker Networks

Network Type	Description	Use Case
bridge	Default, per-host container network	Simple container-to-container
host	Bypass Docker's virtual network, share host network stack	High-performance, single-node
none	Disable networking entirely	Isolated, sandboxed containers
overlay	Multi-host networking (Swarm only)	Cross-node communication
macvlan	Containers get direct IPs on LAN	Legacy systems, custom network setups

---

🐳 6.3 Bridge Network (Default Mode)

When you run a container without specifying a network, Docker connects it to the **default bridge network**.

🔍 View default networks:

docker network ls

NETWORK ID NAME DRIVER SCOPE

a1b2c3d4e5f6 bridge bridge local

🔍 **Inspect:**

docker network inspect bridge

- Containers are assigned a **private IP** (e.g., 172.17.0.X)
- DNS is available inside network
- You **cannot resolve containers by name** on default bridge — use custom bridge for that

---

👫 **6.4 Custom Bridge Networks (Named Networking)**

✅ **Why use them?**

- Enables DNS-based container discovery
- Easier service-to-service communication
- Clean separation between environments

🔧 **Create custom network:**

docker network create mynet

🧪 **Test example:**

docker run -d --name app1 --network=mynet nginx

docker run -it --rm --network=mynet busybox ping app1

PING app1 (172.18.0.2): 56 data bytes

🌐 **6.5 Host Networking**

In **host networking**, Docker container uses the **host's network namespace** — no isolation.

docker run --network host nginx

⚠️ **Important:**

- Only available on Linux
- Containers share host IP and port space
- Cannot run multiple containers on same port

✅ **Use case:**

- Ultra-low-latency services

- Performance testing
- Avoiding NAT

🛑 6.6 None Network – Full Isolation

docker run --network none alpine

- No network access (not even loopback)
- Ideal for security isolation, sandboxed workloads

🌍 6.7 Overlay Network (Multi-Host)

Overlay networks enable **containers running on different hosts** to talk to each other. Built-in with **Docker Swarm**.

```
docker network create \
 --driver overlay \
 --attachable \
 myoverlay
```

Requirements:

- Docker Swarm initialized
- Hosts can communicate via ports: 2377, 7946, 4789

📞 6.8 Port Mapping and Exposure

🔧 **Syntax:**

docker run -p <host_port>:<container_port>

Example:

docker run -d -p 8080:80 nginx

- Exposes container's port 80 on host's port 8080
- Accessible at http://localhost:8080

Advanced:

```
docker run -p 127.0.0.1:9000:80 nginx    # Bind to localhost only
docker run -p 192.168.1.10:3000:3000     # Bind to specific host IP
```

6.9 Docker DNS and Service Discovery

Containers connected to the same **user-defined bridge** or overlay network can resolve each other **by container name**.

Example:

docker run --name redis --network appnet redis

docker run --rm --network appnet busybox ping redis

DNS resolves:

plaintext

CopyEdit

PING redis (172.18.0.3)

🔍 6.10 Inspecting Container Networks

Inspect container:

docker inspect container-name

Look under "Networks" section for:

- IP Address
- MAC
- Network name
- Gateway

🧷 6.11 Connecting Containers to Multiple Networks

docker network create frontend

docker network create backend

docker run -d --name app1 --network frontend nginx

docker network connect backend app1

Now, app1 is connected to both frontend and backend.

🛠️ 6.12 Docker Compose Networking

Docker Compose automatically:

- Creates a default network

- Adds services to that network
- Enables name-based resolution

Example:

```
version: "3"
services:
  web:
    image: nginx
  app:
    build: .
    depends_on:
      - web
```

Here, app can connect to web using just:

curl http://web

🔒 6.13 Security and Isolation in Docker Networks

Feature	Available in Docker?
Network segmentation	✅ via user-defined networks
Internal-only ports	✅ use --internal flag
Encrypted overlay	✅ in Swarm only
Firewalling between containers	❌ (not native, needs external tools like Cilium)

📋 6.14 Real-World Microservices Example

docker network create appnet

docker run -d --name api --network appnet myapi

docker run -d --name redis --network appnet redis

docker run -d --name nginx --network appnet -p 80:80 nginx

- myapi talks to redis at hostname redis
- nginx talks to api at hostname api

🧠 6.15 Network Troubleshooting

❌ **“Cannot reach container”:**

- Not on same network?
- Wrong container name (DNS)
- Firewall on host?

🔧 **Tools:**

docker exec -it container ping another

docker exec -it container nslookup other

docker exec -it container curl http://service:port

🔄 **6.16 Clean Up and Manage Networks**

List:

docker network ls

Inspect:

docker network inspect mynet

Remove:

docker network rm mynet

Prune:

docker network prune

Module 7: Docker Compose

🔧 7.1 What is Docker Compose?

Docker Compose is a tool used to **define and manage multi-container Docker applications**.

Rather than running multiple docker run commands, Compose allows you to:

- Define services, networks, volumes in **a single YAML file**
- Start all services with docker-compose up
- Stop, rebuild, scale, or update them with simple commands

It acts as **orchestration for development and testing**, perfect for:

- Microservices stacks
- Full-stack development
- CI/CD environments
- Integration testing

🧠 7.2 Key Concepts in Compose

🔹 **services**

Defines each container (e.g., app, database, cache).

🔹 **volumes**

Data persistence across containers.

🔹 **networks**

Enables inter-container communication.

🔹 **build**

Defines how to build images directly from Dockerfiles.

🔹 **depends_on**

Sets startup ordering (not health-based).

📄 7.3 Compose File Format (Version 3.x)

```
version: "3.9"
services:
  app:
    build: .
    ports:
      - "8080:80"
    environment:
      - NODE_ENV=production
  db:
    image: postgres:15
    volumes:
      - pgdata:/var/lib/postgresql/data
volumes:
  pgdata:
```

🛠️ 7.4 Real-World Full Stack Example (Node.js + PostgreSQL)

```
version: "3.8"
services:
  web:
    build: ./web
    ports:
      - "3000:3000"
    depends_on:
      - db
  db:
    image: postgres:15
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
      POSTGRES_DB: mydb
```

```
  volumes:
    - dbdata:/var/lib/postgresql/data
volumes:
  dbdata:
```

Web connects to the DB using db as hostname.

🚀 7.5 Common docker-compose Commands

Command	Description
docker-compose up	Build and run all services
docker-compose up -d	Run in background
docker-compose down	Stop and remove containers, networks
docker-compose build	Build/rebuild services
docker-compose logs	View aggregated logs
docker-compose exec <svc>	Run commands inside container
docker-compose ps	List containers
docker-compose restart	Restart services

🧱 7.6 Service Configuration Options

Key	Purpose
image	Pull from Docker Hub
build	Build from Dockerfile
ports	Port mapping
volumes	Volume binding
environment	Set environment variables
networks	Attach to Docker networks
restart	Restart policies
command	Override default command

Key	Purpose
entrypoint	Override entrypoint
healthcheck	Define service health

📦 7.7 Build Contexts

```
web:
  build:
    context: ./app
    dockerfile: Dockerfile.prod
```

Supports:

- Custom Dockerfile paths
- Build args

```
args:
  NODE_ENV: production
```

📁 7.8 Using .env Files in Compose

```
POSTGRES_USER=admin
POSTGRES_PASSWORD=secret
```

Compose automatically loads .env in the same directory.

Or use:

```
docker-compose --env-file my.env up
```

📌 7.9 Advanced Volumes and Bind Mounts

Bind Mount:

```
volumes:
  - ./code:/app
```

Named Volume:

```
volumes:
  - pgdata:/var/lib/postgresql/data
```

External Volume:

volumes:

pgdata:

external: true

🌐 7.10 Custom Networks

networks:

appnet:

driver: bridge

Attach services:

services:

backend:

networks:

- appnet

Compose gives **service-name-based DNS** by default.

🍼 7.11 depends_on vs Health Check

depends_on:

- db

⛔ Does **not wait** for db to be healthy, only checks container start.

✅ To wait for healthy state, use:

healthcheck:

test: ["CMD", "pg_isready", "-U", "postgres"]

interval: 10s

timeout: 5s

retries: 5

And use wait-for-it or similar in app container entrypoint.

🧩 7.12 Override Compose for Dev/Prod

Default file:

docker-compose.yml

Override:

docker-compose.override.yml

Or custom:

docker-compose -f docker-compose.yml -f docker-compose.prod.yml up

Use Case:

- dev: bind mounts, live reload
- prod: use prebuilt images, healthchecks

🧠 7.13 Compose in CI/CD Pipelines

In GitHub Actions:

- uses: docker/setup-buildx-action@v2

- run: docker-compose -f docker-compose.ci.yml up --abort-on-container-exit

In GitLab CI:

services:

- docker:dind

script:

- docker-compose up -d

- docker-compose exec app npm test

🔐 7.14 Secrets Management with Compose

secrets:

my_secret:

file: ./secrets/db_password.txt

Attach to service:

yaml

CopyEdit

services:

db:

secrets:

- my_secret

Inside container: /run/secrets/my_secret

✅ Requires Docker Swarm mode.

🔁 7.15 Rebuild and Restart Patterns

Rebuild with cache:

docker-compose build

Rebuild fresh:

docker-compose build --no-cache

Force recreate:

docker-compose up --force-recreate

📈 7.16 Logging and Debugging

docker-compose logs app

docker-compose logs -f db

Troubleshooting tip:

docker-compose config

This shows fully merged config (default + override).

🧠 7.17 Real-World Microservice Example

Stack:

- NGINX as reverse proxy
- Node.js app
- MongoDB
- Redis

version: "3.8"

services:

api:

build: ./api

ports: ["3000:3000"]

```
  depends_on: [ mongo, redis ]
 mongo:
  image: mongo
  volumes: [ "mongodata:/data/db" ]
 redis:
  image: redis
 nginx:
  image: nginx
  ports: [ "80:80" ]
  volumes:
   - ./nginx.conf:/etc/nginx/nginx.conf
volumes:
 mongodata:
```

📋 7.18 Compose vs Kubernetes (K8s)

Feature	Docker Compose	Kubernetes
Use Case	Dev/Test/CI	Production orchestration
Portability	Local + CI/CD	Multi-cloud
Networking	Built-in DNS	Advanced + Service Mesh
Scaling	Manual (via CLI)	Auto-scaling
Health Checks	Basic	Full readiness/liveness
Secrets	Swarm-only	Native support

Module 8: Docker in CI/CD Pipelines (Ultra Deep Dive)

🚀 8.1 Why Use Docker in CI/CD?

Docker revolutionizes CI/CD by enabling:

- **Isolated, reproducible builds** across environments
- **Fast spin-up and teardown** of testing environments

- **Immutable artifacts** for production
- Seamless packaging + deployment in the same workflow

Docker ensures **“works on my machine” == “works in CI == works in production.”**

⚙️ 8.2 Typical Docker-Enabled Pipeline Phases

1. **Checkout Code**
2. **Build Image (docker build)**
3. **Run Tests inside Container**
4. **Scan Image for Vulnerabilities**
5. **Sign & Push to Registry**
6. **Deploy to Staging/Prod**

🧪 8.3 Example CI/CD Pipeline Workflow with Docker

Git Push →

CI Pipeline →

[docker build]

[docker run tests]

[docker scan]

[docker tag & push]

[trigger deploy]

🔧 8.4 Docker in Jenkins Pipelines (Declarative)

Jenkinsfile Example:

```
pipeline {
  agent any
  environment {
    IMAGE = "myapp:${env.BUILD_ID}"
  }
  stages {
    stage('Build') {
      steps {
```

```
        sh 'docker build -t $IMAGE .'
      }
    }
    stage('Test') {
      steps {
        sh 'docker run --rm $IMAGE npm test'
      }
    }
    stage('Scan') {
      steps {
        sh 'trivy image $IMAGE'
      }
    }
    stage('Push') {
      steps {
        withCredentials([usernamePassword(credentialsId: 'dockerhub', passwordVariable: 'PASS',
usernameVariable: 'USER')]) {
          sh '''
            echo "$PASS" | docker login -u "$USER" --password-stdin
            docker tag $IMAGE myrepo/myapp:$BUILD_ID
            docker push myrepo/myapp:$BUILD_ID
          '''
        }
      }
    }
  }
}
```

⚙️ 8.5 Docker in GitHub Actions

.github/workflows/docker-ci.yml

```
name: Build & Push Docker

on:
  push:
    branches: [ main ]

jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Login to DockerHub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USER }}
          password: ${{ secrets.DOCKERHUB_PASS }}

      - name: Build and Push Image
        uses: docker/build-push-action@v4
        with:
          push: true
          tags: devopsshack/app:latest
```

🔧 8.6 Docker in GitLab CI

```
image: docker:20.10
services:
  - docker:dind

stages:
  - build
  - test
  - push

variables:
  IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA

build:
  stage: build
  script:
    - docker build -t $IMAGE_TAG .

test:
  stage: test
  script:
```

```
  - docker run --rm $IMAGE_TAG npm test

push:
  stage: push
  script:
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER --password-stdin
$CI_REGISTRY
    - docker push $IMAGE_TAG
```

📦 8.7 Test Strategies Inside Docker

Strategy	Example
Unit tests	docker run --rm myimage npm test
Integration tests	Use docker-compose to spin up DB + app
Parallel tests	Run multiple containers concurrently
End-to-End (E2E)	Cypress/Selenium with headless browser inside container

---

🛡️ 8.8 Security: Scan Images in CI

Use tools like:

- **Trivy**
- **Grype**
- **Docker Scout**
- **Snyk**

Example:

trivy image devopsshack/app:latest

Add into CI job for vulnerability scanning. Fail build on CVEs above threshold.

🔐 8.9 Secrets in Docker CI

Never hardcode secrets in Dockerfile or CI YAML.

Use:

- GitHub Secrets
- Jenkins Credentials plugin
- GitLab CI/CD Variables

In Dockerfile, use build args instead of ENV for temporary secrets:

ARG SECRET

RUN echo $SECRET | some_command

Build with:

bash

CopyEdit

docker build --build-arg SECRET=value .

But prefer to **inject secrets at runtime**, not bake into image.

🧠 8.10 Reproducible Docker Builds in CI

Key practices:

- Pin exact base image version (python:3.11.3-slim)
- Use multi-stage builds
- Use --platform for architecture-specific builds
- Leverage BuildKit for caching and cross-platform support

DOCKER_BUILDKIT=1 docker build --platform linux/amd64 .

⏱️ 8.11 Accelerate CI with Docker Caching

Enable build cache:

- **GitHub Actions**: build-push-action with caching
- **Jenkins**: use persistent workspace layers
- Use --cache-from if previous images are available in registry

docker build --cache-from=devopsshack/app:latest -t app:v2 .

🧩 8.12 SBOM Generation in CI

Software Bill of Materials:

- List of packages in your image
- Required for compliance (e.g., supply chain audits)

Use:

syft devopsshack/app:latest -o json > sbom.json

Validate:

cosign attest --predicate sbom.json ...

🔄 8.13 Containerized CI Runners

Run CI runners **inside containers** for isolated job execution.

Example with GitHub self-hosted runners in Docker:

```
docker run -d \
  -e RUNNER_TOKEN=... \
  -e REPO_URL=https://github.com/myorg/myrepo \
  myrunner-image
```

🔁 8.14 Container-Based Deployment

CI can:

- Tag and push image
- Trigger deployment to:
    - Kubernetes (kubectl, helm, ArgoCD)
    - ECS (Fargate)
    - Nomad
    - Docker Swarm

Tagging example:

```
docker tag myapp myrepo/myapp:1.0.${BUILD_ID}
docker push myrepo/myapp:1.0.${BUILD_ID}
```

📋 8.15 End-to-End Example: Dockerized Build + Deploy

- Checkout
- Build Docker image
- Run tests inside container
- Scan with Trivy
- Sign with Cosign
- Push to DockerHub
- Trigger ArgoCD sync

Module 9: Docker Security in Depth

⚠️ 9.1 Why Docker Security Matters

Containers may feel isolated, but they:

- Share the same OS kernel
- Can be misconfigured to escalate privileges
- May carry vulnerable libraries or secrets

Without proper security, Docker becomes an attack vector — not just an abstraction.

You must secure:

- The **Docker daemon**
- **Container images**
- **Container runtime**
- **Secrets**
- **Build pipelines**
- The **host system**

🔒 9.2 Security Boundaries: VM vs Container

Factor	Virtual Machines	Containers
Kernel isolation	Strong (hypervisor)	Weak (shared kernel)
Privilege Escalation	Difficult (hardware level)	Easier if container breaks out
Attack Surface	Full OS per VM	Shared kernel, thin abstraction

⚠️ One bad container could compromise the host if improperly configured.

🧱 9.3 Docker Daemon Hardening

🔐 Do not expose Docker socket over TCP

BAD: exposes API to the world

dockerd -H tcp://0.0.0.0:2375

✅ **Restrict access to /var/run/docker.sock**

ls -l /var/run/docker.sock

- Add users carefully to docker group:

sudo usermod -aG docker <user>

✅ Best Practice: Only run docker commands as trusted users.

🧠 **9.4 Rootless Docker (Better Isolation)**

Run Docker **without root**:

curl -fsSL https://get.docker.com/rootless | sh

export PATH=$HOME/bin:$PATH

dockerd-rootless-setuptool.sh install

Benefits:

- Daemon runs as normal user
- No sudo access required
- Limits kernel attack surface

🚰 **9.5 Avoid --privileged Containers**

docker run --privileged alpine

This:

- Gives full access to host kernel capabilities
- Mounts host devices
- Completely bypasses container isolation

✅ NEVER use in production unless **absolutely** required (e.g., Docker-in-Docker).

🔐 **9.6 Least Privilege Container Execution**

✅ **Drop capabilities:**

docker run --cap-drop=ALL myapp

Or:

--cap-drop=NET_RAW --cap-add=CHOWN

✅ **Read-only file systems:**

docker run --read-only myapp

✅ **Non-root users:**

RUN adduser -D appuser

USER appuser

📦 **9.7 Hardened Base Images**

Use:

- **Distroless** base images
- **Minimal OS** (e.g., Alpine)
- **No package managers or shell**

Examples:

FROM gcr.io/distroless/static

✅ No bash, no curl, no package managers → harder to exploit.

🔍 **9.8 Image Vulnerability Scanning**

Use:

- Trivy
- Grype
- Snyk
- Docker Scout

Example:

trivy image devopsshack/app:latest

Check for:

- OS-level vulnerabilities (Alpine, Debian)
- Language-specific (npm, pip, gem)
- Misconfigurations

📋 9.9 Software Bill of Materials (SBOMs)

Generate SBOM:

syft devopsshack/app:latest -o json > sbom.json

Validate it:

cosign attest --predicate sbom.json ...

Required by many security compliance standards (NIST, SLSA).

🔐 9.10 Image Signing and Verification (Cosign)

Sign image:

cosign sign --key cosign.key devopsshack/app:latest

Verify:

cosign verify --key cosign.pub devopsshack/app:latest

Use GitHub Actions + Cosign to:

- Sign images after CI build
- Validate before deployment

⚠️ 9.11 Secrets Management

❌ **Anti-patterns:**

ENV AWS_SECRET_KEY=mysecret

✅ **Best Practices:**

- Use docker secrets (Swarm mode)
- Inject via environment variables in CI (docker run -e)
- Use runtime secrets manager like:
    - HashiCorp Vault
    - AWS Secrets Manager
    - Doppler

📋 9.12 Security Labels and Metadata

Add metadata to images:

LABEL maintainer="devopsshack"

LABEL org.opencontainers.image.source="https://github.com/devopsshack/app"

Useful for traceability, SBOM tooling, security scanners.

🛡️ 9.13 Enforce Runtime Profiles

SELinux or AppArmor:

docker run --security-opt apparmor=docker-default myapp

Seccomp:

docker run --security-opt seccomp=seccomp.json myapp

These reduce syscall attack surface.

📋 9.14 Docker Bench for Security (Audit Tool)

Run full audit on Docker host:

git clone https://github.com/docker/docker-bench-security.git

cd docker-bench-security

sudo ./docker-bench-security.sh

Scans for:

- Docker daemon config issues
- Container runtime risks
- Host OS hardening gaps

⚙️ 9.15 Container Network Isolation

- Use **custom bridge networks**
- Avoid containers on default bridge or host
- Use firewalls to restrict egress/ingress
- Add --internal flag to networks to prevent external access:

docker network create --internal isolated

🧱 9.16 Runtime Monitoring and Auditing

Tools:

- **Falco** (syscall detection)
- **Sysdig Secure**
- **Docker Scout**

- **KubeArmor** (for Kubernetes)

Monitor:

- Unexpected execs (e.g., /bin/bash)
- File access to sensitive paths
- Outbound network spikes

🔁 9.17 CI/CD Security Controls

- Run builds in isolated Docker container
- Scan images **before** pushing
- Sign images
- Store images in private registries
- Require multi-factor auth on registry access
- Validate image at deploy-time

🔐 9.18 Secure Private Docker Registries

Use:

- Authentication (basic auth, LDAP, OIDC)
- HTTPS with TLS certificates
- Scan registry images periodically
- Rate limit pull/push
- Enable auditing

Self-hosted options:

- Harbor
- JFrog Artifactory
- GitHub Container Registry (GHCR)
- AWS ECR, GCP Artifact Registry

🔁 9.19 Defense in Depth Summary

Layer	Protection
Docker Daemon	Disable remote access, rootless mode
Images	Sign, scan, use minimal base
Runtime	Drop capabilities, run non-root
Network	Isolate, segment, firewall
Secrets	Inject securely, avoid baking in
Registry	Secure access, scan continuously
CI/CD	Scan, sign, verify pre-deploy

Module 10: Docker Internals & Advanced Concepts

⚙️ 10.1 What Actually Happens When You Run a Container?

When you execute:

docker run -it ubuntu bash

Here's what Docker does internally:

1. **CLI → Docker API**:
   docker CLI sends a request to Docker daemon (dockerd) via the UNIX socket (/var/run/docker.sock).
2. **Pulls Image**:
   Docker checks if ubuntu exists locally. If not, it pulls from Docker Hub.
3. **Creates Container**:
   - Allocates a **container ID**
   - Sets up an **isolated root filesystem**
   - Initializes **namespaces** (PID, NET, MNT, etc.)
   - Applies **cgroups** (resource limits)
4. **Starts Container**:
   - Launches **container process** (here, bash) inside the new environment
   - Logs, IO, and runtime metrics begin streaming

🔗 10.2 Linux Kernel Features Behind Containers

Docker is just a user-friendly API over core **Linux kernel primitives**:

🧱 1. Namespaces

Provide isolation between containers:

Namespace	Isolates
pid	Process IDs
mnt	Mount points
net	Network stack
uts	Hostname, domain
ipc	Interprocess comm.
user	UID/GID mappings

Namespace	Isolates
cgroup	Control groups

Each container has its own namespaces.

📦 2. Cgroups (Control Groups)

Control **resource usage**:

- CPU shares, quotas
- Memory limits
- Disk I/O limits
- Max processes

View via:

cat /proc/self/cgroup

🗂️ 3. Union File Systems

Docker layers images using **UnionFS**:

- Combines multiple read-only image layers with a top read-write layer
- Only changes go into the writable layer

Common drivers:

- overlay2 (default on most systems)
- aufs, btrfs, zfs (legacy)

Inspect with:

docker info | grep Storage

📄 10.3 Container Lifecycle Internals

Phase	Internals
create	Allocates container ID, filesystem, sets config
start	Spawns entrypoint process in isolated namespaces
stop	Sends SIGTERM, then SIGKILL after grace period
pause	Freezes process via cgroup freezer

Phase	Internals
kill	Sends signal (SIGKILL, etc.) to container PID
remove	Deletes filesystem, container metadata

🧠 10.4 Docker Architecture Refresher

[docker CLI]

↓

[Docker API]

↓

[Docker Daemon (dockerd)]

↓

[containerd] ← manages container lifecycle

↓

[runc] ← low-level OCI-compliant runtime

↓

[Linux Kernel]

- runc implements the OCI runtime spec
- containerd handles execution, image management, metrics

Docker is not the only player: Kubernetes, CRI-O, and Podman also use containerd + runc.

🧰 10.5 BuildKit – The Modern Docker Build Engine

BuildKit is a modern backend for docker build.

✅ **Advantages:**

- Parallelism
- Layer caching
- Secrets injection
- Better output control

Enable:

export DOCKER_BUILDKIT=1

docker build .

Secrets Example:

```
# syntax=docker/dockerfile:1.4

RUN --mount=type=secret,id=token curl -H "Authorization: $(cat /run/secrets/token)" https://private-api
```

🧱 10.6 DevContainers (Visual Studio Code + Docker)

DevContainers are **containerized dev environments** used via **VSCode Remote Containers** or **GitHub Codespaces**.

devcontainer.json Example:

```
{
  "name": "Node Dev",
  "build": {
    "dockerfile": "Dockerfile",
    "args": { "VARIANT": "16" }
  },
  "settings": {
    "terminal.integrated.shell.linux": "/bin/bash"
  },
  "extensions": [
    "dbaeumer.vscode-eslint",
    "esbenp.prettier-vscode"
  ]
}
```

Benefits:

- Same environment for everyone on the team
- Works on any machine with Docker + VSCode
- Prevents “works on my laptop” bugs

🧩 10.7 Docker Contexts and Remote Management

Use docker context to manage multiple environments:

```
docker context create remote \
  --docker "host=ssh://user@remote-host"
```

docker context use remote

docker ps

Great for:

- Managing remote Linux servers
- Switching between local and cloud environments
- Running builds on a remote BuildKit daemon

📦 10.8 Inspecting Docker Internals (Hands-on)

Get container PID:

docker inspect --format '{{.State.Pid}}' mycontainer

Then:

nsenter -t <PID> -a

You are now “inside” the container from the host.

View namespace info:

ls -l /proc/$(docker inspect ... | jq .[].State.Pid)/ns

View image layers:

docker history <image>

📋 10.9 Namespaces: Isolating Everything

Isolate hostname:

docker run -it busybox hostname

Inside: container_id, not host hostname.

Isolate PIDs:

docker run -it busybox sh

ps aux

You’ll only see a few processes inside container.

🧠 10.10 Image Layer Optimization Strategy

Tip	Benefit
Minimize RUN instructions	Fewer layers, faster builds
Sort commands by volatility	Better caching
Use .dockerignore	Smaller context
Use COPY package*.json before COPY . .	Faster build if code changes
Use multi-stage builds	Clean final image

🔄 10.11 Advanced Build Patterns

Build for multiple platforms:

docker buildx create --use

docker buildx build --platform linux/amd64,linux/arm64 \

-t myapp:multiarch --push .

Use target stages:

FROM node AS build

RUN npm ci && npm run build

FROM nginx

COPY --from=build /dist /usr/share/nginx/html

Then build only up to build stage:

docker build --target build .

📦 10.12 OCI (Open Container Initiative) Standards

Docker pushed for standardization → led to **OCI specs**:

Spec	Purpose
OCI Runtime Spec	How containers are started and run
OCI Image Spec	How images are structured and layered
OCI Distribution Spec	How registries store, fetch images

Runtimes like:

- Docker

- Podman
- CRI-O
  All adhere to OCI → compatible across ecosystems.

📋 10.13 Container Performance Metrics

Use tools like:

- docker stats
- cAdvisor
- containerd metrics
- sysdig

Monitor:

- CPU usage
- Memory footprint
- Network I/O
- Block I/O

Containers can be resource-hungry if not tuned (watch OOMKilled events).

Module 12: Real-World Use Cases for Docker

🔧 12.1 Why Use Docker in Real Projects?

Docker enables:

- Reproducibility of environments
- Isolation between services
- Portability across systems
- Developer onboarding in minutes
- Infrastructure consistency in CI/CD
- Fast provisioning in staging/prod

Let's explore real-world examples in dev, testing, CI/CD, microservices, and cloud-native architectures.

🍼 12.2 Use Case: Containerizing a Monolithic App

Imagine a legacy Java-based Spring Boot application.

Traditional Deployment:

- Java version mismatch
- Dependency hell
- OS-specific bugs

Dockerized Solution:

FROM eclipse-temurin:17-jdk

COPY target/app.jar /app.jar

CMD ["java", "-jar", "/app.jar"]

Benefits:

- Runs identically across environments
- Java version is locked and consistent
- Simplifies deployment using a single .jar image

🌐 12.3 Use Case: Microservices Architecture

A modern web stack may have:

- frontend (React/Vue)

- api-gateway
- auth-service
- user-service
- order-service
- redis, postgres

Compose-based Dev Environment:

```
version: "3.9"
services:
  api:
    build: ./api
    depends_on: [ db, redis ]
    environment:
      DB_HOST: db
  db:
    image: postgres
    volumes: [ "dbdata:/var/lib/postgresql/data" ]
  redis:
    image: redis
  frontend:
    build: ./frontend
    ports: [ "3000:3000" ]
volumes:
  dbdata:
```

Benefits:

- Easy to run full stack locally
- Inter-service DNS resolution (db, redis)
- Shared networks and volumes

🧪 12.4 Use Case: CI/CD Testing

Use Docker to isolate:

- Unit test environments

- Integration tests with databases
- E2E tests using Cypress or Selenium

Example: Node Test Runner

test:

image: node:20

volumes:

- .:/app

working_dir: /app

command: npm test

Can be triggered via:

bash

CopyEdit

docker-compose run test

🔐 12.5 Use Case: Secure Build Pipelines

In CI:

- Build Docker image
- Run unit tests
- Run Trivy scan
- Push image to registry
- Trigger deployment

GitHub Actions Snippet:

- uses: docker/build-push-action@v4

with:

context: .

push: true

tags: myorg/app:latest

- name: Scan

run: trivy image myorg/app:latest

☁️ 12.6 Use Case: Cloud Portability

A single Docker image can:

- Run locally with Docker Desktop
- Be deployed on:
  - AWS ECS/Fargate
  - GCP Cloud Run
  - Azure App Service
  - Kubernetes

Example:

docker build -t devopsshack/app:v1 .

docker push devopsshack/app:v1

Then deploy on any platform using the same image.

💿 12.7 Use Case: Running Databases Locally

Running MySQL or PostgreSQL natively is messy. Use Docker:

```
docker run --name pg \
 -e POSTGRES_PASSWORD=secret \
 -v pgdata:/var/lib/postgresql/data \
 -p 5432:5432 postgres:15
```

Benefits:

- Version-controlled DB
- Easy reset
- No polluting local system

🧑‍🏫 12.8 Use Case: Teaching, Demos & Developer Onboarding

Using DevContainers:

```
{
 "name": "Python Starter",
 "build": { "dockerfile": "Dockerfile" },
 "extensions": ["ms-python.python"]
}
```

VSCode loads a full dev environment on any machine via Docker.

Benefits:

- Instant reproducibility
- No “works on my laptop”
- Faster onboarding

🔄 12.9 Use Case: Backups and Data Migration

Use volumes + tarball backups:

```
docker run --rm \
  -v pgdata:/data \
  -v $(pwd):/backup \
  alpine tar czf /backup/db.tar.gz -C /data .
```

Restore:

```
bash
CopyEdit
docker run --rm \
  -v pgdata:/data \
  -v $(pwd):/backup \
  alpine tar xzf /backup/db.tar.gz -C /data
```

🧩 12.10 Use Case: Dev/Test Matrix Across Platforms

Use multi-arch builds:

```
docker buildx build \
  --platform linux/amd64,linux/arm64 \
  -t myorg/app:multiarch \
  --push .
```

Benefits:

- Support M1/M2 Macs and Linux x86 in one step
- Avoid runtime architecture bugs

📦 12.11 Use Case: Replacing VMs for Local Dev

Instead of spinning up:

- Ubuntu VM
- Apache + PHP
- MySQL

Use a single container:

docker run -it -p 80:80 php:8.2-apache

✅ Faster than Vagrant or VirtualBox. No virtualization overhead.

🧪 12.12 Use Case: E2E Automation Testing with Containers

- Run test server in Docker
- Launch headless browser in separate container
- Use a shared Docker network

Example:

docker network create testnet

docker run -d --network testnet --name web myweb:latest

docker run --rm --network testnet cypress/included

🔐 12.13 Use Case: Running Security Tools via Containers

Tools like:

- Trivy
- Nmap
- Nikto
- OWASP ZAP
- SSLyze

Can all be run as containers:

docker run --rm aquasec/trivy image myapp

docker run --rm instrumentisto/nmap scanme.nmap.org

No need to install them locally.

💉 12.14 Use Case: Building ML & Data Pipelines

Data scientists often use:

- Python
- Jupyter
- Pandas, NumPy
- Custom dependencies

Docker allows shipping the entire stack:

FROM jupyter/scipy-notebook

RUN pip install seaborn nltk scikit-learn

docker run -p 8888:8888 my-ml-env

Module 13: Docker Troubleshooting Playbook

🚨 13.1 Why Docker Breaks (And Often)

Containers are designed to be **isolated** and **ephemeral**, but real-world factors such as:

- Misconfigured mounts
- Crashes due to memory limits
- Image build failures
- Permission mismatches
- Host networking issues
- Secrets leakage

...can cause unexpected and frustrating issues.

This module helps you **systematically diagnose** and **resolve** these container nightmares.

🧠 13.2 Core Troubleshooting Workflow

When a Docker-related problem occurs:

1. **Check container status**

docker ps -a

2. **Get logs**

docker logs <container>

3. **Inspect container details**

docker inspect <container>

4. **Try interactive shell**

docker exec -it <container> sh

5. **Check host resources**

free -m && df -h && top

📋 13.3 Common Issue: Container Exits Immediately

🔍 Diagnosis:

docker ps -a

Check STATUS:

Exited (0) 3 seconds ago

🧪 **Root Causes:**

- No foreground process (e.g., CMD ends immediately)
- Entrypoint misconfigured
- Shell scripts without tail -f or sleep

✅ **Fix:**

CMD ["tail", "-f", "/dev/null"]

Or fix ENTRYPOINT/CMD to run long-lived service (e.g., web server).

🔁 **13.4 Issue: Container Keeps Restarting**

🔍 **Look for restart loop:**

docker ps --filter "status=restarting"

🧪 **Root Causes:**

- Crash in app process
- Port already in use
- Secrets/configs missing
- Memory OOM crash

✅ **Fix:**

- Add logging (docker logs -f)
- Remove bad restart policy during testing:

docker run --restart=on-failure ...

📦 **13.5 Issue: Image Build Fails (docker build)**

🔍 **Sample Error:**

Step 5/9 : RUN npm install

---> Running in ...

npm ERR! no such file or directory

✅ **Fix Steps:**

1. Use .dockerignore to reduce context
2. Split build steps for faster debug
3. Use docker build --progress=plain

4. Check for missing COPY, broken dependencies

🧹 13.6 Issue: Disk Space Full

🔍 Check usage:

docker system df

df -h

✅ Fix:

docker system prune -a

docker volume prune

docker image prune

Be careful — pruning may remove **unused volumes/images** still needed in CI/CD.

🔐 13.7 Issue: Permission Denied

🔍 Common Errors:

Permission denied

chown: invalid user

🧪 Root Causes:

- Host directory mounted with wrong UID
- Read-only bind mounts
- Non-root user in container can't write

✅ Fix:

docker run -v $(pwd):/data --user $(id -u):$(id -g) myimage

Or adjust ownership inside container:

RUN chown -R appuser:appuser /app

📎 13.8 Issue: Port Already in Use

🔍 Error:

bind: address already in use

✅ Fix:

Find the process:

lsof -i :3000

Kill it or map to another port:

docker run -p 3001:3000 myapp

🧠 **13.9 Issue: Container Logs Missing or Empty**

🔍 **Check logs:**

docker logs <container>

🧪 **Causes:**

- App doesn't log to stdout/stderr
- Logs written to file inside container
- Silent process crash

✅ **Fix:**

Update app to log to stdout:

console.log("App started")

Or redirect logs in Dockerfile:

CMD ["sh", "-c", "myapp >> /dev/stdout"]

💀 **13.10 Issue: OOMKilled (Out of Memory)**

🔍 **Diagnose:**

docker inspect <container> | grep -i oom

Check container logs:

docker logs <container>

✅ **Fix:**

- Increase container memory limit:

docker run --memory=1g ...

- Optimize app memory usage (e.g., avoid memory leaks)

🧪 13.11 Issue: File Not Found in Container

🔍 **Error:**

/bin/sh: app.py: not found

✅ **Fix:**

- Check COPY path casing and location
- Use docker exec to inspect file paths
- Ensure working directory is set correctly:

WORKDIR /app

🔄 13.12 Issue: Bind Mount Doesn't Reflect Changes

🧪 **Causes:**

- Caching (Docker Desktop)
- VSCode interfering with bind mount
- Incorrect path mapping

✅ **Fix:**

- Use absolute paths
- Restart Docker Desktop
- Check if file is being cached:

docker run -v $PWD:/app ...

📶 13.13 Issue: Containers Can't Talk to Each Other

🧪 **Causes:**

- Not on the same user-defined network
- Using IPs instead of service names
- Firewall rules or host DNS settings

✅ **Fix:**

- Use a custom bridge network:

docker network create mynet

docker run --network mynet ...

- Communicate via container name (db:5432)

🛠️ 13.14 Useful Debugging Tools in Docker

Tool	Purpose
docker exec	Get a shell in running container
docker logs	Review container logs
docker inspect	Detailed metadata
nsenter	Attach to container namespaces
strace	Trace syscalls
lsof, top	Runtime diagnostics

🧰 13.15 Custom Troubleshooting Container

Use busybox or alpine as debug containers:

docker run -it --network mynet busybox sh

- Ping other containers
- wget, nslookup, ps, etc.

Module 14: Docker Best Practices & Anti-Patterns

✅ 14.1 Why Best Practices Matter

As teams grow and environments scale:

- Docker misuse leads to massive **image bloat**
- Dockerfiles become **non-reproducible**
- Containers become **insecure and fragile**
- CI/CD pipelines slow down

This module summarizes **field-tested strategies** and **dangerous anti-patterns** to help you build Docker the right way — clean, secure, optimized.

🧱 14.2 Dockerfile Design Best Practices

✅ 1. Always Pin Image Versions

✅ Good

FROM node:20-alpine

❌ Bad

FROM node:latest

Unpinned versions change over time, causing **non-reproducible builds**.

✅ 2. Use Multi-Stage Builds

FROM golang:1.21 AS builder

WORKDIR /src

COPY . .

RUN go build -o app

FROM scratch

COPY --from=builder /src/app /app

ENTRYPOINT ["/app"]

Keeps final images:

- Small

- Secure (no compilers or package managers)
- Faster to pull/deploy

✅ 3. Reduce Layers

✅ Combine into one RUN layer

RUN apt update && apt install -y curl && rm -rf /var/lib/apt/lists/*

- Minimizes final image size
- Improves cache usage

✅ 4. Use .dockerignore

node_modules

.git

coverage/

Prevents sending unnecessary files to Docker daemon during build — **speeds up builds** and reduces image context size.

✅ 5. Avoid Installing Debug Tools in Final Image

- Use multi-stage: build/debug in stage 1, copy output only to final
- No need for curl, vim, or bash in production images

✅ 6. Use Non-Root Users

RUN adduser -D appuser

USER appuser

- Prevents privilege escalation
- Required by some PaaS platforms (e.g., Cloud Run)

✅ 7. Use Minimal Base Images

Use Case	Image
General	alpine, debian:slim
Static binary	scratch

Use Case	Image
Node.js	node:20-alpine
Java	eclipse-temurin:<version>-jre

✅ 8. Use ENTRYPOINT for Executable

ENTRYPOINT ["python"]

CMD ["app.py"]

- Allows passing extra args to container (docker run myapp --debug)
- CMD can be overridden easily; ENTRYPOINT sets default command

---

🚫 14.3 Anti-Patterns to Avoid

❌ 1. Using ADD Instead of COPY

ADD . /app # ❌ BAD (unpacks archives, fetches URLs)

COPY . /app # ✅ Safe and predictable

Only use ADD when:

- You need to auto-extract .tar.gz
- You want to pull from remote URL (not recommended)

❌ 2. Hardcoding Secrets

ENV DB_PASSWORD=secret # ❌ Do NOT bake secrets into image

Instead, inject via environment variable or secret volume at runtime.

❌ 3. Running as Root

USER root # ❌ Exposes container if compromised

Always drop privileges using USER directive.

❌ 4. Using latest Tag in Production

- Unpredictable builds
- Might break silently if base image updates

Always pin to specific tag or digest:

FROM python@sha256:<digest>

❌ 5. Too Many Layers

RUN apt update

RUN apt install -y curl

RUN echo "done" # ❌ Creates 3 layers

Combine related steps:

RUN apt update && apt install -y curl && echo "done"

❌ 6. Huge Final Image Sizes

Causes:

- Copying entire repo (COPY . .)
- Not cleaning package caches
- Leaving unused files in image

✅ Use docker image ls and docker history to identify bloat.

🧪 14.4 Build Optimization Patterns

- Use --target to build only part of multi-stage image
- Use --platform to prebuild for M1/ARM:

docker buildx build --platform linux/amd64 .

- Run builds with DOCKER_BUILDKIT=1 for faster builds

🧠 14.5 Image Size Auditing

docker image ls

docker history <image>

Use tools:

- Dive: analyze image layer sizes
- Trivy: scan for vulnerabilities and misconfigurations

🔐 14.6 Security Best Practices

- Sign images using Cosign
- Scan regularly with Trivy
- Use distroless base if possible
- Run containers with:

--cap-drop=ALL --read-only --no-new-privileges

⚙️ 14.7 Operational Best Practices

- Use health checks in Dockerfile:

HEALTHCHECK CMD curl -f http://localhost:8080/health || exit 1

- Use volumes for data, not COPY:

docker run -v data:/app/data myapp

- Use tagging convention for versions:
  devopsshack/app:1.0.0, app:1.0.0-rc1, etc.

Module 15: Ultimate Docker Use Case Recap & Production Checklist

✅ 15.1 Final Recap of Docker's Capabilities

Docker is the glue of DevOps and modern software delivery pipelines. It enables:

Capability	Description
📦 Packaging	Bundle code + dependencies as portable images
🚀 Rapid Deployment	Fast container startups, no need for full OS boot
🧪 Testing Environments	Run tests in disposable, identical containers
🧱 Microservices Isolation	Service-level containerization for better separation and scaling
🔁 CI/CD Integration	Build, scan, test, and deploy from containers
☁️ Cloud Compatibility	Works on EKS, ECS, GKE, Cloud Run, Azure App Services, etc.
🛠️ Developer Onboarding	Use DevContainers or Compose to standardize dev envs
🔐 Security & Observability	Integrate with Trivy, Cosign, Prometheus, Falco, etc.

📋 15.2 Final Docker Production Checklist

🔐 Security

- Use USER non-root inside Dockerfile
- Drop Linux capabilities with --cap-drop=ALL
- Set --read-only filesystem for runtime
- Use HEALTHCHECK directive
- Set no-new-privileges: true in container config
- Enable image signing (e.g. cosign sign)
- Run Trivy scans pre-deployment
- Never bake secrets into the image (ENV, COPY, ARG)
- Use Kubernetes Secrets or Vault to inject credentials at runtime

---

📦 Image & Build Hygiene

- Pin base image tags (e.g. FROM node:20-alpine)

- Use .dockerignore to eliminate unwanted files
- Use multi-stage builds to strip build-only dependencies
- Avoid bloated tools in final stage (vim, curl, git)
- Keep layers minimal and combined where appropriate
- Use distroless or alpine wherever possible

🚀 **Deployment Readiness**

- Set proper ENTRYPOINT + CMD
- Map ports explicitly (EXPOSE 8080)
- Externalize configuration using ENV or --env-file
- Mount volumes for persistent data, not COPY
- Use tagging conventions (v1.2.3, release-2025-05)
- Push to trusted registry (Docker Hub, ECR, Harbor)

📊 **Observability & Monitoring**

- Expose /metrics for Prometheus (where applicable)
- Add logs to stdout/stderr (avoid file-based logs)
- Instrument readiness + liveness probes (if used with K8s)
- Use sidecars (e.g. Fluent Bit, Filebeat) if needed

🧪 **15.3 Cheat Sheet – Docker CLI Essentials**

```
# Build & run
docker build -t myapp .
docker run -d -p 8080:80 myapp

# Logs & inspect
docker logs -f <container>
docker inspect <container>
docker exec -it <container> sh

# Networking
```

```
docker network create mynet
docker run --network mynet ...

# Volumes
docker volume create pgdata
docker run -v pgdata:/var/lib/postgresql/data ...

# Cleanup
docker system prune -a
docker image prune
docker volume prune
```

🧠 15.4 Template: Secure Production Dockerfile (Node Example)

```
# Stage 1 - Build
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2 - Run
FROM node:20-alpine
RUN adduser -D appuser
USER appuser
WORKDIR /app
COPY --from=builder /app/dist ./
CMD ["node", "index.js"]
```

🔐 15.5 Template: Runtime Flags for Hardened Containers

```
docker run \
  --read-only \
  --cap-drop=ALL \
  --security-opt no-new-privileges \
  --memory=512m \
  --pids-limit=100 \
  --user 1000:1000 \
  -v /tmp:/tmp \
  -p 8080:8080 \
  myapp
```

📦 15.6 Template: Docker Compose with Secrets, Volumes, Limits

```
version: "3.9"
services:
  web:
    image: devopsshack/app:1.0.0
    ports:
      - "80:80"
    environment:
      DB_HOST: db
    secrets:
      - db_password
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: "0.5"
  db:
    image: postgres:15
    volumes:
```

```
    - pgdata:/var/lib/postgresql/data
  environment:
    POSTGRES_PASSWORD_FILE: /run/secrets/db_password
volumes:
  pgdata:
secrets:
  db_password:
    file: ./secrets/db_password.txt
```

🧩 15.7 Real-World Docker Toolchain Stack (2025+ Ready)

Area	Tool
🧱 Image Build	Docker BuildKit, Kaniko, Buildah
📦 Registry	Docker Hub, AWS ECR, Harbor
🔍 Scanning	Trivy, Grype, Snyk
📜 Signing	Cosign, Notary v2
🔁 CI/CD	GitHub Actions, GitLab, ArgoCD
🔐 Secrets	HashiCorp Vault, Doppler, SOPS
🧪 Tests	Testcontainers, Playwright, Cypress
☁️ Deploy	Kubernetes, ECS, GKE, Cloud Run
🔎 Monitoring	Prometheus, Grafana, Loki
🔐 Runtime Sec	Falco, AppArmor, Seccomp, gVisor

📦 15.8 Final Tips Before You Docker in Production

1. **Always scan before you ship**
   Don’t deploy an image without a vulnerability scan.
2. **Log everything to stdout/stderr**
   Avoid writing to log files inside container.
3. **Never bake secrets into Dockerfiles**
   Secrets should be runtime-injected only.

4. **Small images = fast deploys**
   Use alpine, slim, or distroless.
5. **Make your Dockerfile reproducible**
   Pin everything. No latest.
6. **Push only signed images**
   Use cosign sign --key ... before pushing to registry.
7. **Always tag and version**
   Never just push :latest unless you want to live dangerously.