WEB DEVELOPMENT

Docker Intermediate/Advanced

Multi-Stage Builds & Image Optimization

Compose, Networking & Production Data · Security & CI/CD

Orchestration Orientation · Capstone Project

Philip Osztromok

Generated with Claude

Table of Contents

Docker Intermediate/Advanced — 9 Chapters

 Multi-Stage Builds

`docker1`'s Dockerfile chapter (`docker_06`) built a single-stage image. Three courses have since referenced multi-stage builds without ever teaching the pattern — `express2-8`'s deployment chapter, `node3-8`'s production chapter, and `pipelines1-6`'s Docker-in-CI chapter. This is the deep dive all three assumed.

What docker1's Single-Stage Dockerfile Leaves on the Table

A single-stage build bundles **everything** used during the build — dev dependencies, build tools, compilers, source files — directly into the *same* image that ends up running in production, even though most of that is only needed to *produce* the runnable output, not to actually run it. The result: a bloated image, and a larger attack surface sitting in production for no functional reason.

The Multi-Stage Pattern

A Dockerfile can contain **multiple `FROM` statements**, each starting a new, independent stage. A later stage can selectively copy just the specific build artifacts it needs from an earlier one via `COPY --from=<stage>` — discarding everything else (build tools, source code, dev dependencies) from the final image entirely.

Single-Stage vs. Multi-Stage Dockerfile

Single-Stage (docker1's approach)

```
FROM node:18
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build
CMD ["node", "dist/server.js"]
# final image still contains devDependencies, source, build tools
```

Multi-Stage

```
FROM node:18 AS builder
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build

FROM node:18-slim
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
CMD ["node", "dist/server.js"]
```

The final image is built `FROM node:18-slim` — a fresh, minimal starting point — and only ever receives the specific files the `COPY --from=builder` lines explicitly select. Everything else that existed in the `builder` stage (the full source tree, the TypeScript compiler, dev-only dependencies) never makes it into the image that actually ships.

Naming Stages & Selective Copying

`AS builder` names a stage so later stages can reference it by name rather than a fragile numeric index. `docker build --target builder` can also build *only* up to a named stage — useful for running tests against the build stage without needing the final lean image at all.

Concept	Purpose
`FROM ... AS name`	Starts a named stage
`COPY --from=name`	Copies specific files from a named stage into the current one
`--target name`	Builds only up to a specific named stage

A real-world illustration of the payoff: a Node.js/TypeScript app built single-stage might land around **1.2GB** once dev dependencies, the compiler, and source files are included — the same app built multi-stage, copying only the compiled output and production dependencies into a slim final image, can easily land under **150MB**.

Why This Pattern Recurs Across the Site

`express2-8`'s deployment chapter and `node3-8`'s production chapter both already used multi-stage Dockerfiles as part of their own deployment examples, and `pipelines1-6`'s CI pipeline built and pushed multi-stage images — all three assumed the reader already understood the pattern shown above. This chapter is the missing explanation those three examples were quietly built on top of.

💻 Coding Challenges

Challenge 1: Convert a Single-Stage Dockerfile

Convert this single-stage Dockerfile into a multi-stage one: `FROM python:3.12`, installs build dependencies, compiles a Cython extension, runs the app. The final image should only contain the compiled extension and runtime dependencies, not the build toolchain.

Goal: Practice splitting a build-and-run Dockerfile into a builder stage and a lean final stage.

→ Solution

Challenge 2: Explain the Size Reduction

Explain, in your own words, why copying only `/app/dist` and production `node_modules` from a builder stage produces a smaller image than copying the entire builder stage's filesystem.

Goal: Practice articulating exactly what a multi-stage build excludes and why that exclusion is the whole point.

→ Solution

Challenge 3: Use --target for Testing

Given this chapter's two-stage Dockerfile, write the `docker build` command that builds only the `builder` stage, useful for running tests against it without building the final slim image.

Goal: Practice using `--target` to build a specific intermediate stage.

→ Solution

⚠️ Gotcha: Copying More Than Necessary Defeats the Whole Pattern

It's easy to write `COPY --from=builder /app ./` — copying the *entire* builder stage's working directory — instead of the specific files actually needed. Doing this brings the source tree, dev dependencies, and build artifacts right back into the final image, quietly undoing the size and attack-surface reduction multi-stage builds exist to provide. Always name build stages explicitly with `AS` (an unnamed stage can only be referenced by a fragile numeric index that breaks if stages are reordered), and be deliberate about exactly which files each `COPY --from` line selects — the discipline of copying only what's needed is the entire value of this pattern, not an optional refinement.

🎯 What's Next

The next chapter is **Image Optimization & Layer Caching** — minimizing layers, `.dockerignore`, choosing a base image (alpine vs. slim vs. full), and build cache invalidation order.

📦 Image Optimization & Layer Caching

Chapter 1 reduced image size *between* stages. This chapter optimizes *within* a stage — fewer, smaller layers, a smart base image choice, and a build order that keeps rebuilds fast instead of reinstalling every dependency on every source change.

How Docker Layers Work

Every instruction in a Dockerfile — `RUN`, `COPY`, `ADD` — creates a new **layer**. Docker caches each layer, reusing it on the next build as long as nothing about that instruction (or anything before it) has changed. Understanding this caching behavior is the foundation for everything else in this chapter.

Minimizing Layers

Combining related commands into a **single** `RUN` instruction — using `&&` and line continuations — reduces the total layer count, and matters for more than just tidiness.

```
# Three separate layers — the apt cache still exists in an EARLIER layer
# even after this later RUN "deletes" it
RUN apt-get update
RUN apt-get install -y curl
RUN rm -rf /var/lib/apt/lists/*

# One layer — the cache never exists in the image's history at all
RUN apt-get update && \
  apt-get install -y curl && \
  rm -rf /var/lib/apt/lists/*
```

.dockerignore

A `.dockerignore` file excludes files from the **build context** sent to the Docker daemon — the same idea as `.gitignore`, applied to what a build is even allowed to see.

```
# .dockerignore
node_modules
.git
.env
*.log
```

This speeds up builds (a smaller context transfers faster) and, just as importantly, prevents accidentally `COPY`ing sensitive files — a local `.env` with real credentials, for instance — into an

image, the same "never let a secret leak into a built artifact" discipline this site has stressed repeatedly for source control and CI.

Choosing a Base Image: Alpine vs. Slim vs. Full

Base Image Type	Size	C Library	Compatibility Risk
Full (e.g. `node:18`)	Largest	glibc	Lowest — includes broad tooling, "just works"
Slim (e.g. `node:18-slim`)	Smaller	glibc	Low — drops non-essential packages, keeps glibc compatibility
Alpine (e.g. `node:18-alpine`)	Smallest	musl	Real — native modules compiled against glibc can fail or behave subtly differently

Alpine's small size comes from using **musl** libc instead of the more common **glibc** — a genuinely different C library, not just a stripped-down version of the same one. Some native Node.js addons or Python packages compiled against glibc can fail to run, or behave subtly differently, on Alpine — a well-known, real compatibility gotcha, not a theoretical edge case.

Build Cache Invalidation Order

The moment *anything* about an instruction changes, that layer's cache is invalidated — and so is **every layer after it**, even if those later instructions themselves didn't change at all. Instruction **order** in the Dockerfile determines how much gets rebuilt on a typical change.

Bad Ordering vs. Good Ordering

Bad: COPY . . First

```
COPY . .
RUN npm install
```

Any source file change invalidates the `COPY` layer — and therefore forces `npm install` to re-run on *every single build*, even when dependencies never changed.

Good: Dependencies First

```
COPY package*.json ./
RUN npm install
COPY . .
```

`npm install`'s cache stays valid across source-only changes — it only re-runs when `package.json`/`package-lock.json` actually change.

The general rule: order instructions from **least likely to change** to **most likely to change** — dependency manifests before source code, since dependencies change far less often than application code does during normal development.

💻 Coding Challenges

Challenge 1: Combine Layers With Cleanup

Rewrite this three-layer sequence into a single `RUN` instruction that never leaves the cleaned-up files in the image's history: `RUN apt-get update`, `RUN apt-get install -y git`, `RUN rm -rf /var/lib/apt/lists/*`.

Goal: Practice the combine-and-cleanup-in-one-layer pattern this chapter introduced.

→ Solution

Challenge 2: Reorder for Better Caching

A Python Dockerfile has `COPY . .` immediately followed by `RUN pip install -r requirements.txt`. Reorder it so source-code-only changes don't force a dependency reinstall.

Goal: Practice applying the dependency-manifest-first ordering rule to a Python project.

→ Solution

Challenge 3: Pick a Base Image

A team's Node.js app depends on a native addon compiled against glibc, and image size is a secondary concern to reliability. Which base image type should they choose, and why not the smallest option?

Goal: Practice weighing size against compatibility risk using this chapter's alpine/slim/full comparison.

→ Solution

⚠️ Gotcha: Deleting a File in a Later Layer Doesn't Shrink the Image

Each layer is a **diff** against the previous one — when a later `RUN rm ...` instruction deletes a file, Docker records that deletion as its own layer, but the file *still physically exists* in the earlier layer's diff, and the image still has to ship it. This is exactly why the three-separate-`RUN` example above doesn't actually save any space, despite the final command "removing" the apt cache — the cache is still sitting in the layer created by `apt-get update`. The fix is always the same: cleanup has to happen in the *same* `RUN` instruction that created the mess, never a separate, later one.

🎯 What's Next

The next chapter is **Docker Compose in Depth** — going beyond `docker1`'s intro: override files, profiles, healthchecks, `depends_on` with conditions, and explicitly named networks/volumes.

🧭 Docker Compose in Depth

`docker1`'s Compose chapter (`docker_10`) covered running a multi-container app. This chapter covers the features that turn a working Compose setup into a production-ready one: environment-specific config, optional services, real readiness checks, and predictable networking.

Override Files for Environment-Specific Config

Rather than duplicating an entire `docker-compose.yml` per environment, Compose merges a base file with an **override** file — `docker-compose.override.yml` is picked up automatically, or explicit files can be layered with `-f`.

```
# docker-compose.yml (base, shared)
services:
  api:
    image: myapp:latest
    environment:
      NODE_ENV: production

# docker-compose.override.yml (auto-merged locally)
services:
  api:
    build: .
    environment:
      NODE_ENV: development
    volumes:
      - .:/app

# explicit layering for a different environment
docker compose -f docker-compose.yml -f docker-compose.prod.yml up
```

Profiles

The `profiles` key marks a service as **optional** — it only starts when its profile is explicitly activated, keeping services like a debug tool or a one-off seed job out of the way during normal `up` runs.

```
services:
  seed-data:
    image: myapp-seed:latest
    profiles: ["debug"]

# normal `docker compose up` skips seed-data entirely
# explicitly activating it:
docker compose --profile debug up
```

Healthchecks

A container can be **running** while the application inside it hasn't actually finished starting up — a `healthcheck` block tells Docker how to verify genuine readiness, not just process liveness.

```
services:
  db:
    image: mysql:8
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 5s
      retries: 10
      start_period: 30s
```

depends_on With Conditions

Plain `depends_on: [db]` only waits for the `db` container to **start** — not for MySQL inside it to actually be accepting connections. This is one of Compose's most common real bugs: an app container starts, tries to connect to a database that's technically running but not yet ready, and crashes.

```
services:
  api:
    depends_on:
      db:
        condition: service_healthy
```

depends_on Without vs. With a Condition

> **Plain depends_on**
>
> Waits only for the `db` container process to start — the app can still connect before MySQL is ready, and crash.

> **depends_on with service_healthy**
>
> Waits for the healthcheck (this chapter's `mysqladmin ping`) to actually pass first — the app only starts once the database is genuinely ready.

Explicitly Named Networks & Volumes

By default, Compose auto-generates network/volume names prefixed with the project directory name. Giving them explicit names makes them predictable to reference from outside Compose, or to share deliberately across multiple Compose files/projects.

```
networks:
  backend:
    name: shop-backend-network

volumes:
  db-data:
    name: shop-db-data
```

Feature	Purpose
Override files	Layer environment-specific config on a shared base, without duplicating the whole file
`profiles`	Keep optional services out of normal `up` runs
`healthcheck`	Distinguish "running" from actually ready
`depends_on: condition: service_healthy`	Wait for real readiness, not just container start
Named networks/volumes	Predictable references outside Compose or across projects

💻 Coding Challenges

Challenge 1: Add a Healthcheck

Write a `healthcheck` block for a Redis service (image `redis:7`) that runs `redis-cli ping` every 5 seconds, allowing up to 5 retries.

Goal: Practice writing a healthcheck's `test`/`interval`/`retries` fields for a real service.

→ Solution

Challenge 2: Fix a depends_on Race Condition

A team's `api` service has `depends_on: [db]` and intermittently crashes on startup with a "connection refused" error against the database. Diagnose the cause and fix the Compose file.

Goal: Practice recognizing and fixing the classic `depends_on`-without-a-condition race condition.

→ Solution

Challenge 3: Design an Optional Debug Service

Add a service called `adminer` (a database admin UI) to a Compose file such that it never starts during a normal `docker compose up`, but can be started deliberately when needed.

Goal: Practice using `profiles` to keep an optional tool out of the default startup path.

→ Solution

⚠️ Gotcha: Override Files Don't Merge Every Key the Same Way

It's easy to assume an override file always merges *additively* with the base file — but that's not uniformly true. Keys like `environment` genuinely merge (the override's variables are added to/override individual keys from the base). Keys like `command` or `entrypoint`, however, typically **replace** the base's value entirely rather than merging with it — an override file that only intended to tweak one argument can silently discard the base's entire command if it redefines that key at all. Always check Compose's documented merge behavior for the specific key being overridden, rather than assuming every override is purely additive.

🎯 What's Next

The next chapter is **Container Networking Beyond Basics** — bridge vs. host vs. overlay networks, custom networks for service isolation, and DNS-based service discovery between containers.

Container Networking Beyond Basics

Chapter 3 touched named networks in passing. This chapter explains the actual Docker networking model underneath every Compose file on this site — why service names "just work" as hostnames, and how to deliberately wall services off from each other.

Bridge Networks (the Default)

Bridge is Docker's default network driver — each container gets its own network namespace and IP on a virtual bridge. But there's a critical distinction between two kinds of bridge network:

- The **default bridge network** (`docker0`) — containers can reach each other, but only by **IP address**. There's no automatic name-based DNS resolution on it at all.
- A **custom, user-defined bridge network** — containers get automatic **DNS resolution by container/service name**, no IP address needed.

This is exactly why Compose always creates a custom network automatically for every project — it's the reason a service-name-based connection string like `mysql://db:3306` works at all in every Compose example this site has used.

Host Networking

Host networking removes the network boundary entirely — a container shares the host machine's network namespace directly, binding to the host's ports with no mapping needed. The trade-off: maximum performance and simplicity, but **zero network isolation** — a real security consideration Chapter 6 covers properly. Rarely the right default choice.

Overlay Networks

Overlay networks answer "what about multiple physical/virtual hosts?" — they let containers running on *different* machines communicate as if they were on the same local network, the foundation multi-host orchestration (Docker Swarm, and conceptually Kubernetes) relies on. Chapter 8 covers orchestration properly; this is just naming where overlay networking fits in the bigger picture.

Custom Networks for Service Isolation

Multiple custom networks can deliberately **segment** which services can reach which — a genuine defense-in-depth technique, the same network-isolation principle `dbsec1-4` applied at the VM/bare-metal level, just expressed at the container level instead.

```
networks:
 frontend:
 backend:

services:
 web:
  networks: [frontend]
 api:
  networks: [frontend, backend]
 db:
  networks: [backend]
```

`web` can reach `api` (both on `frontend`), and `api` can reach `db` (both on `backend`) — but `web` has **no network path to `db` at all**, not merely "isn't supposed to" by convention. This is enforced at the network layer itself.

DNS-Based Service Discovery Between Containers

On a custom network, Docker's embedded DNS server resolves each container's **service name** directly to its current IP — no manual `/etc/hosts` editing, no hardcoded addresses that break the moment a container restarts with a new IP.

Default Bridge vs. Custom Bridge

Default Bridge

Containers can reach each other, but only by IP address — no built-in name resolution.

Custom Bridge

Automatic DNS by container/service name — exactly what makes `mysql://db:3306`-style connection strings work.

Network Driver	Use Case
Default bridge	Legacy behavior; generally avoided in favor of custom bridges
Custom bridge	Single-host multi-container apps — automatic DNS by name
Host	Maximum performance, no isolation — rarely the right default
Overlay	Multi-host communication (Swarm, conceptually Kubernetes)

💻 Coding Challenges

Challenge 1: Explain Why a Service Name Resolves

Explain why `mysql://db:3306` works as a connection string inside an `api` container in a Compose project, but wouldn't work between two containers started with plain `docker run` on the default bridge network.

Goal: Practice connecting Compose's automatic custom-network creation to the DNS resolution it enables.

→ Solution

Challenge 2: Design an Isolated Network Layout

Design a Compose network layout for a system with a public-facing `web` service, an internal `api` service, and a `db` service, such that `web` can never reach `db` directly.

Goal: Practice applying the frontend/backend network-segmentation pattern to a new scenario.

→ Solution

Challenge 3: Diagnose a Connection Failure

A developer's `web` service can't reach their `db` service at all, even though both containers are "running." Using this chapter's material, list the most likely cause.

Goal: Practice recognizing network segmentation as the likely cause of an otherwise-confusing connection failure.

→ Solution

⚠️ Gotcha: "Both Running" Doesn't Mean "Can Reach Each Other"

It's a common early confusion to assume two containers can talk to each other simply because both show as `running` in `docker ps`. If they're on *different* networks — or one is on the default bridge while the other is on a custom network — there is genuinely **no network path between them at all**, not a permissions issue, not a firewall rule, just no route. This isn't a bug; it's the isolation feature working exactly as designed, the same isolation this chapter's frontend/backend example deliberately relied on. When two containers can't reach each other, checking which networks each one is actually attached to (`docker network inspect`) is one of the first things worth verifying, before assuming the problem lies somewhere else entirely.

🎯 What's Next

The next chapter is **Volumes & Data Management in Production** — named volumes vs. bind mounts vs. tmpfs, backup strategies for containerized data, and volume drivers.

Volumes & Data Management in Production

Containers are deliberately disposable — a container can be removed and recreated at any time. This chapter covers where data that *needs* to survive that actually lives, and how to back it up.

Named Volumes

A **named volume** is storage Docker manages itself, living in Docker's own storage area rather than directly inside the project directory. It's the recommended default for persistent application data — a database's files, uploaded content — and survives container removal entirely, remaining available to be reattached to a new container.

```
docker run -d --name db -v db-data:/var/lib/mysql mysql:8
```

Bind Mounts

A **bind mount** maps a specific *host* filesystem path directly into the container — full read/write coupling to the actual host filesystem. This is what makes local-development live-reloading work (mounting source code so container-side changes reflect on disk instantly), but it's a real liability in production: tightly coupled to one host's specific filesystem layout, and not portable if that container ever needs to run on a different machine.

```
# local development — live-reloading source code
docker run -v $(pwd):/app -p 3000:3000 myapp
```

tmpfs Mounts

A **tmpfs mount** stores data in the host's **memory only** — it's never written to disk at all, and its contents vanish the moment the container stops. Right for temporary or sensitive data that genuinely shouldn't persist — a session cache, credentials being processed transiently — a real security benefit (nothing ever touches disk) as well as a performance one.

```
docker run --tmpfs /app/cache myapp
```

Named Volume vs. Bind Mount vs. tmpfs

> **Named Volume / Bind Mount**
>
> Both persist to disk. Named volumes are Docker-managed and portable; bind mounts are tied to a specific host path.

> **tmpfs**
>
> Never touches disk at all — lives only in memory, gone the instant the container stops.

	Persists After Container Removal?	Best Fit
Named volume	Yes	Persistent application data (databases, uploads)
Bind mount	Yes (on the host path)	Local development, live-reloading source code
tmpfs	No — memory only	Sensitive/temporary data that must never touch disk

Backup Strategies for Containerized Data

Because a named volume lives in Docker's own managed storage area rather than an obvious project folder, backing it up needs a deliberate step: running a temporary container that mounts *both* the volume and a host backup directory, then archiving the contents across.

```
docker run --rm \
 -v db-data:/data \
 -v $(pwd)/backup:/backup \
 alpine tar czf /backup/db-data-$(date +%Y%m%d).tar.gz /data
```

The same discipline `dbsec1-8` established for database backups applies directly here — encrypt the resulting archive before it's stored anywhere, and periodically test that it actually restores, rather than assuming a successful-looking backup command means the data is genuinely recoverable.

Volume Drivers

A volume's actual storage backend is pluggable via **volume drivers** — `local` (the default, a directory on the host running the container) is the common case, but a driver plugin can back a volume with network storage (NFS) or a cloud provider's block storage instead. This matters in production deployments spanning multiple hosts, where a named volume needs to be more than just a folder tied to whichever specific machine happened to create it.

💻 Coding Challenges

Challenge 1: Choose the Right Mount Type

For each, recommend a named volume, bind mount, or tmpfs: (a) a PostgreSQL database's data directory in production, (b) source code during local development with live reload, (c) a directory holding decrypted API credentials only for the duration of one request.

Goal: Practice applying this chapter's three-way comparison to concrete scenarios.

→ Solution

Challenge 2: Write a Backup Command

Write the `docker run` command to back up a named volume called `uploads-data` into a compressed archive in the current host directory, named with today's date.

Goal: Practice the temporary-container backup pattern this chapter introduced.

→ Solution

Challenge 3: Diagnose Lost Data

A developer's app wrote uploaded files directly into `/app/uploads` inside a running container — no volume, no bind mount. After redeploying (removing and recreating the container), all uploaded files are gone. Explain what happened.

Goal: Practice recognizing why data written to a container's own writable layer, rather than a volume, doesn't survive container removal.

→ Solution

⚠️ Gotcha: Data Written Directly Into a Container Doesn't Survive It

A container's own writable layer is exactly as disposable as the container itself — any file written there without a volume or bind mount backing that path vanishes the moment the container is removed, even if it was never explicitly deleted by anyone. This is the single most fundamental reason volumes exist at all: **never store data that needs to survive inside a container's own filesystem** — always mount a named volume (or, for local dev, a bind mount) at any path the application writes data that actually matters to. A related, common practical annoyance with bind mounts specifically: files created inside the container (often as root) can end up with a UID that the host user can't read or write — worth checking early rather than assuming permissions will just line up.

🎯 What's Next

The next chapter is **Docker Security** — running as non-root, read-only filesystems, scanning images for vulnerabilities, and never baking credentials into an image.

🔒 Docker Security

Chapters 4–5 secured the network and the data. This chapter hardens the container itself — who it runs as, what it can modify at runtime, whether its own base image is carrying known vulnerabilities, and whether a secret was ever accidentally baked into it.

Running as Non-Root

Unless told otherwise, a container's main process runs as **root** inside the container. If an attacker ever manages to escape the application layer into the container's own OS layer (via a vulnerability or misconfiguration), running as root there is a meaningfully worse starting point than a non-privileged user — root inside the container has far more latitude to do damage.

```
FROM node:18-slim
RUN useradd --create-home appuser
USER appuser
WORKDIR /home/appuser/app
COPY --chown=appuser:appuser . .
CMD ["node", "server.js"]
```

Root vs. Non-Root Container Process

Root

A container-escape vulnerability, combined with a root process, hands an attacker significantly more capability inside that boundary.

Non-Root

The same escape scenario leaves the attacker with only the limited privileges of an unprivileged user — the blast radius stays contained.

Read-Only Filesystems

The `--read-only` flag (or `read_only: true` in Compose) makes a container's filesystem immutable at runtime, except for explicitly mounted writable paths — a `tmpfs` mount (Chapter 5) for anything

that genuinely needs to write, like `/tmp`.

```
docker run --read-only --tmpfs /tmp myapp
```

This limits what a compromised process can actually *do*: it can't modify the application's own code or binaries at runtime, and can't drop a malicious script onto disk to persist itself — a genuine defense-in-depth measure, not a substitute for fixing the underlying vulnerability.

Scanning Images for Vulnerabilities

A base image bundles OS packages and libraries that can carry their own known, publicly documented CVEs — exactly the outdated-components risk `owasp1-6` and `dbsec1-9` already covered, applied here specifically to a container image's own contents. Tools like **Trivy** (open source) or `docker scan` check an image's installed packages against known vulnerability databases.

```
trivy image myapp:latest
```

This belongs in the **build/CI pipeline** (Chapter 7, and the site's `pipelines1` course generally) as an automated, recurring check — not a one-time manual scan run once and forgotten while the base image quietly accumulates newly-disclosed vulnerabilities over time.

Never Baking Credentials Into an Image

A classic, real mistake: hardcoding a secret directly in a Dockerfile's `ENV` instruction, or `COPY`ing a `.env` file containing real credentials into the image. This is the exact same discipline `pipelines1-5` established for Git commits — a credential baked into an image is compromised the moment that image exists, regardless of what happens afterward.

Baked-In Secret vs. Runtime Injection

Bad: Hardcoded in the Dockerfile

```
ENV DATABASE_PASSWORD=Sup3rSecret
```

Good: Injected at Runtime

```
docker run -e DATABASE_PASSWORD="$DB_PASSWORD" myapp
# or a secrets manager / orchestrator-native secret
```

Security Measure	Protects Against
Non-root user	Escalated damage from a container-escape vulnerability
Read-only filesystem	A compromised process modifying code or persisting malware on disk
Image vulnerability scanning	Known, unpatched CVEs in the base image's own packages
No baked-in credentials	A secret being extractable from the image itself, indefinitely

💻 Coding Challenges

Challenge 1: Add a Non-Root User

Rewrite this Dockerfile snippet to run as a non-root user: `FROM python:3.12-slim`, `WORKDIR /app`, `COPY . .`, `CMD ["python", "app.py"]`.

Goal: Practice adding a dedicated user and switching to it before the app runs.

→ Solution

Challenge 2: Explain the Value of Read-Only

Explain what a `--read-only` container specifically prevents an attacker from doing, even after successfully exploiting a vulnerability in the running application.

Goal: Practice articulating read-only's defense-in-depth value precisely, not just "it's more secure."

→ Solution

Challenge 3: Spot the Credential Leak

A Dockerfile includes `ENV API_KEY=abc123` in one layer, then a later layer runs `RUN unset API_KEY`. Explain why the API key is still compromised despite this.

Goal: Practice connecting Chapter 2's layer-diff mechanics to a real security consequence, not just a size one.

→ Solution

⚠️ Gotcha: A "Deleted" Secret Is Still Sitting in an Earlier Layer

Chapter 2 explained that deleting a file in a later layer doesn't shrink the image, because each layer is a diff and the earlier layer's contents are still there. Applied to secrets, this is a genuine security hazard, not just a size one: an `ENV` value or a `COPY`'d `.env` file set in one layer and "removed" in a later one is still fully extractable — anyone with the image can run `docker history`, or simply unpack the image's layers directly, and read the credential straight out of the earlier layer, regardless of what any later instruction did. The only real fix is the same one Git commits need: the secret should never enter any layer in the first place — inject it at runtime (an environment variable passed to `docker run`, a secrets manager, an orchestrator-native secret) instead of writing it into the Dockerfile at build time.

🎯 What's Next

The next chapter is **Docker in CI/CD** — reviewing and building on the CI/CD Pipelines course's own Docker-in-CI chapter, registry push/pull, and tagging strategy.

🚀 Docker in CI/CD

`pipelines1-6` already built and pushed a Docker image as part of a CI pipeline. This chapter reviews that material, goes deeper on registry mechanics, and formalizes a real tagging strategy — while folding in everything this course has covered since: multi-stage builds (Ch.1), optimized caching (Ch.2), and vulnerability scanning (Ch.6).

Recap: pipelines1-6's Docker-in-CI Chapter

`pipelines1-6` established the core shape: build an image tagged with the commit SHA, push it to a registry, and gate that job on the test suite already passing — "no image without passing tests." This chapter builds on exactly that foundation.

Container Registries

A **registry** is where built images actually live — Docker Hub is the public default, but production systems commonly use a private registry (a cloud provider's own, or a self-hosted one like Harbor). `docker login` authenticates; `docker push`/`docker pull` move images in and out.

```
# CI step — authenticating with registry credentials from CI secrets, never hardcoded
echo "$REGISTRY_PASSWORD" | docker login registry.example.com -u "$REGISTRY_USER" --password-stdin
```

Registry credentials are themselves secrets — the exact same discipline Chapter 6 established for never baking a credential into an image applies just as much to the credentials used to *push* that image: read from CI secrets at runtime, never hardcoded into a pipeline config file.

Tagging Strategy

`pipelines1-6`'s commit-SHA tagging is immutable and traceable — that exact tag always points to that exact commit, forever. A complete tagging strategy layers a few tag types together:

Tag Type	Purpose	Mutable?
Commit SHA (e.g. `a1b2c3d`)	Exact traceability to one commit	No — immutable
Semantic version (e.g. `v1.2.3`)	Marks a real release	No — should never be reused
`latest`	Convenience pointer to "the newest build"	Yes — and that's the problem

```
# tagging one build with multiple tags at once, on a release
docker build -t myapp:a1b2c3d -t myapp:v1.2.3 -t myapp:latest .
docker push myapp:a1b2c3d
docker push myapp:v1.2.3
docker push myapp:latest
```

latest vs. an Immutable SHA Tag

`latest`

Mutable and ambiguous — any subsequent push to `latest`, from any branch, silently changes what that tag points to.

Commit SHA

Permanently tied to one exact commit — pulling it a year from now returns exactly the same image, every time.

Putting It Together: A Full CI Build/Push Step

```
# 1. Multi-stage build (Ch.1), using registry cache from the last build (Ch.2)
docker build \
  --cache-from myapp:latest \
  -t myapp:$GIT_SHA \
  -t myapp:latest \
  .

# 2. Vulnerability scan gate (Ch.6) — fail the pipeline on critical findings
trivy image --exit-code 1 --severity CRITICAL myapp:$GIT_SHA

# 3. Push only after the scan passes
docker push myapp:$GIT_SHA
docker push myapp:latest
```

💻 Coding Challenges

Challenge 1: Write a Multi-Tag Build Command

Write a `docker build` command that tags one build with a commit SHA (`e5f6g7h`) and a semantic version (`v2.0.0`), for a release build — but deliberately without a `latest` tag.

Goal: Practice applying multiple tags to a single build in one command.

→ Solution

Challenge 2: Secure the Registry Login Step

A CI config hardcodes `docker login -u admin -p Sup3rSecret` directly in its pipeline file. Rewrite it to follow this chapter's credential-handling guidance.

Goal: Practice applying Chapter 6's never-hardcode-credentials discipline to registry authentication specifically.

→ Solution

Challenge 3: Diagnose a Production Incident

A production system is configured to always pull `myapp:latest`. After an unrelated build on a different branch pushed a broken image also tagged `latest`, production silently started running that broken build. Explain what went wrong and how this chapter's tagging strategy would have prevented it.

Goal: Practice connecting the `latest`-tag gotcha to a realistic, concrete incident.

→ Solution

⚠️ Gotcha: Never Deploy Production Based on the latest Tag

The `latest` tag is just a mutable pointer — it doesn't mean "the newest stable release," it means "whatever was pushed to `latest` most recently, from wherever." A production system configured to always pull `latest` has no guarantee it's running anything specific at all — a stray build from an unrelated branch, a hotfix that was never meant to go live, or a broken in-progress push can silently become what production is running, with no deploy event, no record, and no easy way to know exactly what's live without inspecting the image directly. Production deployments should always reference an immutable, specific tag — a commit SHA or a semantic version — precisely because that guarantees pulling it always returns the exact same image, verifiably, every time.

🎯 What's Next

The next chapter is **Orchestration Beyond Compose** — why Compose doesn't scale to multi-host production, and a conceptual orientation to Kubernetes/Swarm as the next step.

🧩 Orchestration Beyond Compose

Chapter 4 named overlay networks as the answer to "what about multiple hosts?" without explaining what actually needs that answer. This chapter closes that loop: why Compose — no matter how deep this course has gone with it — structurally cannot manage a fleet of machines, and what genuinely does.

What Compose Is Actually Good At (Recap)

Everything Chapters 3–7 covered — override files, healthchecks, custom networks, named volumes, security hardening, CI integration — makes Compose a genuinely excellent tool for its actual scope: starting, networking, and managing a defined set of containers **on one machine**. That scope covers a huge share of real deployments.

Where Compose Stops Scaling

A `docker-compose.yml` describes services that run on **one host** — whichever machine `docker compose up` is run on. Compose has no built-in concept of "multiple hosts" at all:

- **No cross-machine scheduling** — Compose can't decide to run a service on a different machine than the one it's invoked on.
- **No real self-healing** — Compose's restart policies can restart a crashed *container* on the same host, but if the **entire host** goes down, nothing in Compose itself notices, let alone reschedules that workload elsewhere.
- **No fleet-wide rolling deployment** — deploying a new version across many machines isn't something Compose was built to coordinate.

What Orchestration Actually Adds

Orchestration treats a cluster of many machines as one pool of resources, with a **scheduler** deciding which machine runs which container — and automatically rescheduling workloads if a machine dies. This is self-healing at the *cluster* level, not just "restart this one container": an entire machine disappearing gets its workload redistributed elsewhere, automatically. This is a structurally different capability Compose was never designed to have — not a missing feature waiting to be added.

Docker Swarm

Swarm is Docker's own native orchestrator, with a genuinely similar mental model to Compose — a `docker-compose.yml` can often be deployed nearly as-is via `docker stack deploy`. Lower learning curve coming from Compose, though it's seen less industry-wide adoption than Kubernetes in recent years.

Kubernetes (Conceptual Orientation Only)

Kubernetes is the dominant orchestrator industry-wide, with a genuinely different mental model and a much steeper learning curve than Swarm or Compose — **Pods** (not quite the same thing as a single container), **Deployments**, **Services**, and a full declarative API rather than Compose's simpler file format. This chapter deliberately does *not* go deep on Kubernetes — just naming these core concepts so they're recognizable as the natural next step once orchestration needs genuinely outgrow this course's scope.

Compose vs. Swarm/Kubernetes

Compose

Single host, a defined set of containers, no cluster-level scheduling or self-healing.

Swarm / Kubernetes

A cluster of many machines treated as one pool, with automatic scheduling and self-healing across the whole fleet.

	Compose	Swarm	Kubernetes
Scope	Single host	Multi-host cluster	Multi-host cluster
Learning curve from Compose	—	Low	Steep
Industry adoption	Extremely common (dev, small deployments)	Declining	Dominant at scale

Compose Is Still Enough When

The deployment genuinely fits on one well-resourced machine — most side projects, single-server production deployments, and everything Chapters 1–7 of this course already cover.

Real Orchestration Is Needed When

True high availability across multiple machines is a genuine requirement, or the workload has outgrown what a single host can realistically serve.

💻 Coding Challenges

Challenge 1: Explain the Structural Limit

Explain, in your own words, why Compose's inability to reschedule work after a host fails is a structural limitation, not a missing feature that a future Compose update could simply add.

Goal: Practice articulating the actual scope Compose was designed for, versus what orchestration adds.

→ Solution

Challenge 2: Compose, Swarm, or Kubernetes?

A solo developer runs a small side project on one VPS. A company runs a large production system across dozens of machines with strict uptime requirements. Recommend an approach for each, using this chapter's comparison.

Goal: Practice matching orchestration complexity to actual, not assumed, scaling needs.

→ Solution

Challenge 3: Connect to a Prior Site Gotcha

This chapter's closing gotcha warns against adopting Kubernetes prematurely. Name a similar "don't do this before you need it" gotcha from another course on this site, and explain the shared underlying principle.

Goal: Practice recognizing a recurring judgment-call pattern across genuinely different technologies.

→ Solution

⚠️ Gotcha: Reaching for Kubernetes Prematurely Is a Real, Common Mistake

"Kubernetes for a single side project" is a well-known industry joke for a reason — adopting cluster orchestration before there's a genuine multi-host scaling or availability problem to solve adds real, substantial operational complexity (a cluster to run, a much steeper learning curve, an entirely different mental model) for a problem that doesn't yet exist. This is the exact same judgment call `mongodb2-6`'s closing exercise made about sharding a MongoDB collection that "comfortably fits on a single 3-node replica set with room to spare" — don't adopt distributed-systems complexity ahead of a real, demonstrated need. The underlying principle recurs across genuinely different technologies for the same reason every time: match infrastructure complexity to an actual, current requirement — not to where the project might theoretically be someday.

🎯 What's Next

The final chapter is the **Capstone: Productionizing a Multi-Container App** — combining multi-stage builds, Compose, security hardening, and CI integration into one small real application.

🏁 Capstone: Productionizing a Multi-Container App

This capstone builds a small, real **Task API + database** system, combining multi-stage builds (Ch.1–2), Compose (Ch.3–5), security hardening (Ch.6), and CI integration (Ch.7) into one production-ready setup.

The Application: A Task API + Database

A Node.js `api` service backed by a PostgreSQL `db` service — small enough to stay readable here, real enough to exercise every chapter in this course.

Step 1: Multi-Stage, Optimized Build (Chapters 1–2)

```
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM node:18-slim
RUN useradd --create-home appuser
WORKDIR /home/appuser/app
COPY --from=builder --chown=appuser:appuser /app/dist ./dist
COPY --from=builder --chown=appuser:appuser /app/node_modules ./node_modules
USER appuser
CMD ["node", "dist/server.js"]
```

Dependencies are copied and installed *before* the source (Chapter 2's cache-ordering rule), the final stage only receives compiled output and production dependencies (Chapter 1), and the container already runs as a non-root user (Chapter 6, folded in early rather than bolted on later).

Step 2: Compose With Healthchecks & Named Resources (Chapters 3–5)

```
networks:
  backend:
    name: taskapi-backend

volumes:
  db-data:
    name: taskapi-db-data

services:
  db:
    image: postgres:16
    networks: [backend]
    volumes:
      - db-data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "postgres"]
      interval: 5s
      retries: 10

  api:
    build: .
    networks: [backend]
    depends_on:
      db:
        condition: service_healthy
```

The database's actual data lives in a named volume (Chapter 5), the two services share an explicitly named custom network (Chapter 4), and `api` waits for `db`'s healthcheck to genuinely pass (Chapter 3) rather than merely starting.

Step 3: Security Hardening (Chapter 6)

```
services:
  api:
    build: .
    read_only: true
    tmpfs:
      - /tmp
    environment:
      DATABASE_URL: ${DATABASE_URL} # from CI/host env, never hardcoded
```

The `api` container's filesystem is read-only at runtime, with `/tmp` mounted as tmpfs for anything genuinely needing to write. `DATABASE_URL` is injected from the environment, never baked into the image or the Compose file itself.

Step 4: CI Integration (Chapter 7)

```
# CI pipeline step
docker build --cache-from myapp/api:latest -t myapp/api:$GIT_SHA -t myapp/api:latest .
trivy image --exit-code 1 --severity CRITICAL myapp/api:$GIT_SHA
docker push myapp/api:$GIT_SHA
docker push myapp/api:latest  # convenience tag only — production deploys the SHA tag, never this one
```

Putting It All Together

Course Concept	Where It's Used in This App
Multi-stage builds (Ch.1)	Builder stage compiles; final stage ships only `dist` + production deps
Cache-ordering / base image choice (Ch.2)	Dependencies copied before source; slim final base
Healthchecks + `depends_on` (Ch.3)	`api` waits for `db`'s real readiness, not just container start
Named networks (Ch.4)	`taskapi-backend` — isolates and connects the two services
Named volumes (Ch.5)	`taskapi-db-data` — the database's data survives container removal
Non-root, read-only, no baked-in secrets (Ch.6)	`appuser`, `read_only: true`, `DATABASE_URL` from environment
SHA/latest tagging, registry push, scan gate (Ch.7)	The full CI build/scan/push sequence

💻 Coding Challenges

Challenge 1: Add a .dockerignore

Write a `.dockerignore` file for this capstone's build context, excluding `node_modules`, `.git`, and any local `.env` file.

Goal: Practice applying Chapter 2's `.dockerignore` guidance to this specific project.

→ Solution

Challenge 2: Add an Override File for Local Development

Write a `docker-compose.override.yml` that bind-mounts the local source directory into `api` for live-reloading during development, without changing the base `docker-compose.yml`.

Goal: Practice combining Chapter 3's override-file pattern with Chapter 5's bind-mount use case, layered on top of this capstone's base file.

→ Solution

Challenge 3: Explain the Registry Authentication Step

Extend Step 4's CI snippet with the registry login step, and explain why it must never hardcode credentials directly in the pipeline file.

Goal: Practice applying Chapter 7's credential-handling guidance to this capstone's own CI step.

→ Solution

⚠️ Gotcha: read_only and Persistent Data Aren't in Conflict

It can look contradictory at first glance: `db` writes real data to disk constantly, yet Chapter 6's `read_only` hardening seems to say containers shouldn't write anything. The resolution is that `read_only` locks down the container's **own** filesystem layer — it says nothing about a separately mounted **named volume**, which remains fully writable regardless. If `db` here were hardened with `read_only: true`, its data directory would still need an explicit writable exception (either accepting that the volume mount itself isn't affected by `read_only`, or adding a `tmpfs`/volume override for the specific paths that must write) — the two techniques from Chapters 5 and 6 are complementary, not contradictory, but combining them correctly requires understanding that a volume mount and the container's own filesystem are genuinely separate things.

🎉 Course Complete

That's the full **Docker Intermediate/Advanced** course — from multi-stage builds and image optimization through Compose in depth, container networking, production data management, security hardening, CI integration, and orchestration orientation, finishing with a real, production-shaped multi-container application. Together with **Docker for Beginners**, this completes both Docker courses on the site.